Nebula Graph Database 内核手册

v1.2.1

吴敏,张莹,黄晓丹

2021 Vesoft Inc.

Table of contents

1. 简介 4
- 1.1 关于本手册 4
- 1.2 欢迎使用 Nebula Graph 手册 5
- 1.3 基本概念 11
- 1.4 快速开始和常用链接 18
- 1.5 系统设计与架构 32

2. 查询语言 41
- 2.1 面向的读者 41
- 2.2 数据类型 42
- 2.3 函数与操作符 45
- 2.4 语言结构 60
- 2.5 语句语法 73

3. 编译、部署与运维 123
- 3.1 编译 123
- 3.2 安装 129
- 3.3 配置 139
- 3.4 账号权限管理 153
- 3.5 批量数据管理 164
- 3.6 监控与统计 180
- 3.7 源码开发和 API 186

4. 图算法 189
- 4.1 nebula-algorithm 189

5. 数据传输 195
- 5.1 Nebula Exchange 195
- 5.2 Nebula Importer 229
- 5.3 Nebula Flink Connector 230
- 5.4 Nebula Spark Connector 242

6. Nebula Graph Studio 255
- 6.1 Studio版本更新说明 255
- 6.2 认识 Nebula Graph Studio 259
- 6.3 安装与登录 264
- 6.4 快速开始 271
- 6.5 操作指南 282
- 6.6 故障排查 303

2021 Vesoft Inc.

7. 社区贡献 306
7.1 贡献文档 306
7.2 C++ 编码风格 307
7.3 如何提交代码和文档 308
7.4 PR 和 Commit Message 指南 311
8. 附录 312
8.1 Cypher 和 nGQL 312
8.2 Gremlin 和 nGQL 对比 317
8.3 SQL 和 nGQL 332
8.4 点标识符和分区 337
8.5 超级顶点（稠密点）处理 338

2021 Vesoft Inc.

1. 简介

1.1 关于本手册

此手册为 **Nebula Graph** 的用户手册，版本为1.2.1。详细版本更新信息参见 Release Notes。

1.1.1 面向的读者

本手册适用于 `算法工程师` 、 `数据科学家` 、 `软件开发人员` 和 `DBA` ，以及所有对 `图数据库` 感兴趣的人群。

如果在使用 **Nebula Graph** 的过程中有任何问题，欢迎在 Nebula Graph Community Slack 或官方论坛提问。 如果对本手册有任何建议或疑问，请在 GitHub 给我们留言。

1.1.2 格式约定

Nebula Graph 尚在持续开发中，本手册也将持续更新。 本手册使用如下语法惯例：

- `等宽字体`

  等宽字体用于表示**命令**，**需要用户输入的命令**及**接口**。

- **加粗字体**

  用于命令以及其他需要用户逐字输入的文字。

- `UPPERCASE fixed width`

  在查询语言中， `保留关键字` 和 `非保留关键字` 均使用大写等宽字体表示。

1.1.3 文件格式

本手册所有文件均采用 Markdown 编写，HTML 网站使用 mkdocs 自动生成。

最后更新: 2021年4月15日

2021 Vesoft Inc.

1.2 欢迎使用 Nebula Graph 手册

Nebula Graph 是一个分布式的可扩展的高性能的图数据库。

Nebula Graph 可以容纳百亿点和万亿条边，并达到毫秒级的时延。

1.2.1 前言

- 关于本手册
- 变更历史

1.2.2 概览 (入门用户)

- 简介
- 基本概念
  - 数据模型
  - 查询语言概览
- 快速开始和常用链接
  - 开始试用
  - 常见问题 FAQ
  - 编译源代码
  - 导入 .csv 文件
  - Nebula Graph SDK
- 系统设计与架构
  - 设计总览
  - 存储层架构
  - 查询引擎架构

1.2.3 查询语言 (所有用户)

- 数据类型
  - 基本数据类型
  - 类型转换
- 函数与操作符
  - 位运算
  - 内置函数
  - 比较运算
  - 聚合运算
  - 分页 (Limit)
  - 逻辑运算
  - 排序 (Order By)
  - 集合运算
  - 字符比较函数和运算符
  - uuid 函数

2021 Vesoft Inc.

- 语言结构
  - 字面值常量
    - 布尔类型
    - 数值类型
    - 字符串类型
  - 注释语法
  - 标识符大小写
  - 关键字和保留字
  - 管道
  - 属性引用
  - 标识符命名规则
  - 语句组合
  - 用户自定义变量

2021 Vesoft Inc.

- 语句语法
  - 数据定义语句 (DDL)
    - 新建图空间
    - 新建 Edge
    - 新建 Tag
    - 修改 Edge
    - 修改 Tag
    - 删除 Tag
    - 删除 Edge
    - 删除 Space
    - 索引
    - TTL (time-to-live)
  - 数据查询与操作语句 (DQL 和 DML)
    - 删除边
    - 删除顶点
    - 获取点和边属性 (Fetch)
    - 图遍历 (Go)
    - 插入边
    - 插入顶点
    - 查找数据 (Lookup)
    - 返回满足条件的语句 (Return)
    - 更新点
    - 更新边
    - Upsert 语法
    - 条件语句 (Where)
    - 返回结果语句 (Yield)

2021 Vesoft Inc.

- 辅助功能语句
  - Show 语句
    - Show Charset 语法
    - Show Collation 语法
    - Show Configs 语法
    - Show Create Spaces 语法
    - Show Create Tag/Edge 语法
    - Show Hosts 语法
    - Show Indexes 语法
    - Show Parts 语法
    - Show Roles 语法
    - Show Snapshots 语法
    - Show Spaces 语法
    - Show Tag/Edge 语法
    - Show Users 语法
  - Describe
  - Use
- 图算法
  - 查找路径

1.2.4 编译、部署与运维 (程序员和 DBA)

- 编译
  - 编译源代码
  - 使用 Docker 编译
- 安装
  - rpm 安装
  - 起停服务
  - 使用 Docker 安装
  - 集群部署

2021 Vesoft Inc.

- 配置
  - 系统要求
  - 配置持久化与优先级
  - CONFIG 语法
  - Graphd 配置
  - Storaged 配置
  - 命令行终端配置
  - Kernel 配置
  - 单机日志
- 账号权限管理
  - Alter User Syntax
  - 身份验证
  - Built-in Roles
  - Change Password
  - Create User
  - Drop User
  - Grant Role
  - LDAP
  - Revoke
- 批量数据管理
  - 离线数据导入
    - 读取 .csv 文件
    - Spark 导入工具
  - 离线数据转储
    - Dump Tool
  - 负载均衡和数据迁移
  - 集群快照
  - 长耗时任务管理(compact,flush)
  - Compact
- 监控与统计
  - metrics
  - meta 层运行统计 (metrics)
  - storage 运行统计 (metrics)
  - graph 层运行统计 (metrics)
  - 源码开发和 API
  - Key Value 接口
  - Nebula Graph 客户端

1.2.5 社区贡献 (开源社区爱好者)

- 贡献文档
- C++ 编程风格

2021 Vesoft Inc.

- 开发者文档风格
- 如何贡献

1.2.6 附录

- Gremlin V.S. nGQL
- Cypher V.S. nGQL
- SQL V.S. nGQL
- 点标识符和分区

1.2.7 其他

视频

- YouTube
- Bilibili

最后更新: 2020年8月11日

2021 Vesoft Inc.

1.3 基本概念

1.3.1 图数据建模

此文档介绍 **Nebula Graph** 建模及图模型的基本概念。

图空间

图空间 为彼此隔离的图数据，与 MySQL 中的 database 概念类似。

有向属性图

Nebula Graph将数据存储在有向属性图中。有向属性图是指点和边构成的图，这些边是有方向的。有向属性图表示为：

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

- **V**是点的集合。
- **E**是有向边的集合。
- P_V 是点的属性。
- P_E 是边的属性。

下表为篮球运动员数据集的结构示例，包括两种类型的点（**player**、**team**）和两种类型的边（**serve**、**follow**）。

类型	名称	属性名（数据类型）	说明
tag	**player**	name （string） age （int）	表示球员。
tag	**team**	name （string）	表示球队。
edge type	**serve**	start_year （int） end_year （int）	表示球员的行为。 该行为将球员和球队联系起来，方向是从球员到球队。
edge type	**follow**	degree （int）	表示球员的行为。 该行为将两个球员联系起来，方向是从一个球员到另一个球员。

点（Vertex）

点用于表示现实世界中的实体。在 **Nebula Graph** 中，点必须拥有唯一的标识符（即 `VID`）。在每个图空间中的 `VID` 必须是唯一的。

本例的数据中共包含 11 个点。

点类型——标签（Tag）

Nebula Graph 使用**标签**表示点类型；一个点可以同时有多种类型（Tag）。本例中有两种类型的点，其标签（类型）分别为 **player** 和 **team**。

2021 Vesoft Inc.

边（Edge）

边用来连接点，边通常表示两个点间的某种关系或行为，本例中的边为 **serve** 和 **follow**。

边类型（Edge Type）

每条边都有唯一的边类型。两个节点之间允许有多个相同或者不同类型的边。例如，以球员-球队的服役关系 **serve** 为例，（球员）点 `101`（表示一名球员）为起始点，（球队）点 `215`（表示一支球队）为目标点。点 `101` 有一条出边，而点 `215` 有一条入边。

边 rank

两个点之间的边除了必须有类型之外，还必须有 rank。边 rank 是用户分配的 64 位整数；如不指定，边 rank 默认值为 0。

四元组[起点、边类型、权重、终点]可以唯一表示一条边。

边 rank 决定了两个点之间相同类型的边的排序方式。边 rank 值较高的边排名靠前。

目前的排序依据为“二进制编码顺序“：即 0, 1, 2, ... 9223372036854775807, -9223372036854775808, -9223372036854775807, ..., -1。

点和边的属性（Property）

点和边无可可拥有属性，属性以键值对的方式描述。本例中，点 **player** 拥有属性 `id`、`name` 和 `age`，边 **follow** 则拥有属性 `degree`。

Schema

在 **Nebula Graph** 中，Schema 为标签及边对应的属性。与 `MySQL` 类似，**Nebula Graph** 是一种强 schema 的数据库，属性的名称和数据类型在数据写入前已确定。

最后更新: 2021年4月16日

2021 Vesoft Inc.

1.3.2 Nebula Graph 查询语言 (nGQL)

nGQL 概览

`nGQL` 是一种声明型的文本查询语言，目前尚在开发中。新版本会添加更多功能并优化已有功能。未来的语法规范可能会与现行的不一致。

目标

- 易学
- 易用
- 专注线上查询，同时为离线计算提供基础

特点

- 类 SQL，易学易用
- 可扩展
- 大小写不敏感
- 支持图遍历
- 支持模式匹配
- 支持聚合运算
- 支持图计算
- 支持分布式事务（开发中）
- 无嵌入支持组合语句，易于阅读

术语

- **图空间**：物理隔离的不同数据集
- **标签** ：拥有一种或多种属性
  - 每个标签都有一个人类可读的名称，并且每个标签内部都会分配一个 32 位的整数
  - 每个标签与一个属性列表相关联，每个属性都有一个名称和类型
  - 标签之间可存在依赖关系作为约束。例如，如果标签 S 依赖于标签 T，则除非标签 T 存在，否则标签 S 无法存在。
- **点** ：图数据中代表实体的点
  - 每个点都有一个唯一的 64 位（有符号整数）ID (**VID**)
  - 一个点可以拥有多个**标签**
- **边**：点之间的联系称为边
  - 每条边由唯一数组 标识
  - ***边类型*** 是人类可读的字符串，并且每条边内部都会分配一个 32 位的整数。边类型决定边上的属性（模式）
  - ***边 rank*** 是用户分配的不可变的 64 位带符号整数，决定两个顶点之间相同类型的边顺序。等级值较高的边排名靠前。如未指定，则默认等级值为零。目前的排序依据为“二进制编码顺序“：即 0, 1, 2, ... 9223372036854775807, -9223372036854775808, -9223372036854775807, ..., -1。
  - 每条边只能有一种类型
- **路径**： 多个点与边的**非分支**连接
  - 路径长度为该路径上的边数，比点数少 1
  - 路径可由一系列点，边类型及权重表示。一条边是一个长度为 1 的特殊路径

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


2021 Vesoft Inc.

查询语言规则概览

不熟悉 BNF 的读者可跳过本节

总览

- 整套语句可分为三部分：**查询**、**更改**、**管理**
- 每条语句均可返回一个数据集，每个数据集均包含一个 schema 和多条数据

语句组合

- 语句组合有两种方式：
  - 语句可使用管道函数 “**|**” 连接，前一条语句返回的结果可作为下一条语句的查询条件
  - 支持使用 “**;**” 批量输入多条语句，批处理时返回最后一条语句结果

数据类型

- 简单类型： **vid**、**double**、**int**、**bool**、**string** 和 **timestamp**
- **vid** ： 64 位有符号整数，用来表示点 ID
- 简单类型列表，如： **integer[]**, **double[]**, **string[]**
- **Map**: 键值对列表。键类型必须为 **字符**，值类型必须与给定 map
- **Object** (未来版本支持): 键值对列表。键类型必须为**字符**，值可以是任意简单类型
- **Tuple List**: *只适用于返回值*。由元数据和数据（多行）组成 。元数据包含列名和类型。

类型转换

- 一个简单的类型值可以隐式转换为列表
- 列表可以隐式转换为单列元组列表
  - "<type>_list" 可用来表示列名

常用 BNF

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

::=

<type> ::= |

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

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

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

::= <label>

::= (,)*

::= :<type>

::= ":"

::=

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

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

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

2021 Vesoft Inc.

查询语句

选择图空间

Nebula Graph 支持多图空间。不同图空间的数据彼此隔离。在进行查询前，需指定图空间。

USE

返回数据集

返回单个值或数据集

RETURN

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

创建标签

使用以下语句创建**新**标签

CREATE TAG ()

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

创建边类型

使用以下语句创建**新**的边类型

CREATE EDGE ()

:= <label>

插入点

使用以下语句插入一个或多个点

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

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

插入边

使用以下语句插入一条或多条边

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

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

更新点

使用以下语句更新点

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

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

2021 Vesoft Inc.

更新边

使用以下语句更新边

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

图遍历

根据指定条件遍历给定点的关联点，返回点 ID 列表或数组

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

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

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

WHERE 语句仅适用于最终返回结果，对中间结果不适用。

跳过 **STEP[S]** 表示 **一步**

从起始点出发一跳，遍历所有满足**WHERE** 语句的关联点，只返回满足 **WHERE** 语句的结果。

多跳查询时，**WHERE** 语句只适用于最终结果，对中间结果不适用。例如：

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

以上语句查询所有生日在 1988/1/1 之后的二度好友。

搜索

以下语句对满足筛选条件的点或边进行搜索。

FIND VERTEX WHERE [**YIELD**]

FIND EDGE WHERE [**YIELD**]

属性关联

属性关联很常见，如 **WHERE** 语句和 **YIELD** 语句。nGQL 采用如下方式定义属性关联：

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

<var> 以 “$” 开始，特殊变量有两类：$- 和 $$。

$- 为输入值， $$ 为目标值。

所有属性名以字母开头。个别系统属性以 “_” 开头。 “_” 保留值。

内建属性

- _id： 点 ID
- _type： 边类型
- _src： 边起始点 ID

2021 Vesoft Inc.

- _dst： 边终点 ID
- _rank： 边 rank

最后更新: 2020年8月11日

2021 Vesoft Inc.

1.4 快速开始和常用链接

1.4.1 快速入门

本手册将逐步指导您使用 **Nebula Graph**，包括：

- 安装
- 数据建模
- 增删改查
- 批量插入
- 数据导入工具

安装

我们推荐使用 docker compose 来安装 **Nebula Graph**， 具体步骤可以参考我们录制的操作视频。

除了使用 Docker，您还可以选择编译源码或者rpm/deb 包 方式安装 **Nebula Graph**。

如果您在安装过程中遇到任何问题，可以前往 Nebula Graph 官方论坛 提问，我们有专门的值班开发人员为您解答问题。

数据建模

在本手册中，我们将通过下图所示的数据集向您展示如何使用 **Nebula Graph** 数据库。

上图中有两个标签（**player**、**team**）以及两条边类型（**serve**、**follow**）。

2021 Vesoft Inc.

创建并使用图空间

Nebula Graph 中的图空间类似于传统数据库中创建的独立数据库，例如在 MySQL 中创建的数据库。 首先，您需要创建一个图空间并使用 (use) 它，然后才能执行其他操作。

您可以通过以下步骤创建并使用图空间：

1. 检查集群机器状态：

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

   状态 `online` 表示**存储服务进程** `storaged` 已经成功连接上**元数据服务进程** `metad`。

2. 输入以下语句创建图空间：

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

   这里:

   - `partition_num` ：指定一个副本中的分区数。通常为全集群硬盘数量的 5 倍。
   - `replica_factor` ：指定集群中副本的数量，通常生产环境为 3，测试环境可以为 1。由于采用多数表决原理，因此需为奇数。

   你可以通过 `SHOW HOSTS` 命令检查机器和 partition 分布情况：

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

   若发现机器都已在线 (online)，但 Leader distribution 分布不均(如上)，则可以通过命令 (`BALANCE LEADER`) 来触发 partition 重分布：

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

   具体解释可以见这里。

3. 输入以下语句来指定使用的图空间：

```
nebula> USE basketballplayer;
```

4. 现在，您可以通过以下语句查看刚创建的空间：

```
nebula> SHOW SPACES;
```

   返回以下信息：

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

定义数据的 SCHEMA

在 **Nebula Graph** 中，我们将具有相同属性的点分为一组，该组即为一个标签。 `CREATE TAG` 语句定义了一个标签，标签名称后面的括号中是标签的属性和属性类型。 `CREATE EDGE` 语句定义边类型，类型名称后面的括号中是边的属性和属性类型。

2021 Vesoft Inc.

您可以通过以下步骤创建标签和边类型：

1. 输入以下语句创建 **player** 标签：

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

2. 输入以下语句创建 **team** 标签：

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

3. 输入以下语句创建 **follow** 边类型：

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

4. 输入以下语句创建 **serve** 边类型：

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

5. 现在，您可以查看刚刚创建的标签和边类型。

   5.1 要获取刚创建的标签，请输入以下语句：

```
nebula> SHOW TAGS;
```

   返回以下信息：

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

   5.2 要显示刚创建的边类型，请输入以下语句：

```
nebula> SHOW EDGES;
```

   返回以下信息：

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

   5.3 要显示 **player** 标签的属性，请输入以下语句：

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

   返回以下信息：

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

   5.4 要获取 **follow** 边类型的属性，请输入以下语句：

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

   返回以下信息：

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


2021 Vesoft Inc.

创建索引

您可以使用 `CREATE INDEX` 为已有 Tag/Edge-type 创建索引。

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

上述语句在所有标签为 *player* 的顶点上为属性 *name* 创建了一个索引。

索引会影响写性能，第一次批量导入的时候，建议先导入数据，再批量重建索引；不推荐带索引批量导入，这样写性能会非常差。

详情参看索引文档。

增删改查

插入数据

您可以根据示意图中的关系插入点和边数据。

插入点

`INSERT VERTEX` 语句通过指定点的标签、属性、点 ID 和属性值来插入一个点。

您可以通过以下语句插入点：

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

注意：

1. 在上面插入的点中，关键词 `VALUES` 之后的数字是点的 ID（缩写为 `VID`，int64）。每个图空间中的 `VID` 必须是唯一的。
2. 最后插入的点(VID: 121)将在下文删除数据部分中删除。
3. 如果您想一次批量插入同类型的点，可以执行以下语句：

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

   反除号 `\` 用于实现多行语句。

插入边

`INSERT EDGE` 语句通过指定边类型名称、属性、起始点VID和目标点VID以及属性值来插入边。

您可以通过以下语句插入边：

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

同样的：如果您想一次批量插入多条同类型的边，可以执行以下语句：

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

读取数据

在 **Nebula Graph** 中插入数据后，您可以从图空间中查询到之前已经插入的数据。

`FETCH PROP ON` 语句从图空间检索数据。如果要获取点的数据，则必须指定点的标签和点 VID；如果要获取边数据，则必须指定边的类型、起始点 VID 和目标点 VID。

例如，要获取 `VID` 为 `100` 的选手的数据，请输入以下语句：

2021 Vesoft Inc.

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

返回以下信息：

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

例如，要获取 `VID 100` 和 `VID 200` 之间的类型为 `serve` 的边上的属性，请输入以下语句：

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

返回以下信息：

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

更新数据

您可以更新刚插入的点和边数据。

更新点数据

`UPDATE VERTEX` 语句用于更新点的属性。

指定要更新的点 `VID`，然后在等号右侧为其分配新值来更新点的全部或者部分属性。

以下示例说明如何将 `VID 100` 的属性 `name` 从 `Tim Duncan` 更改为 `Tim`。

输入以下语句更新 `name` 值：

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

要检查 `name` 值是否已更新，请输入以下语句：

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

返回以下信息：

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

更新边数据

类似的，`UPDATE EDGE` 语句用于更新边的属性。

通过指定边的起始点 VID 和目标点 VID，然后在等号右侧为其分配新值来更新边的属性。

以下示例展示了如何更改 `VID 100` 和 `VID 101` 之间，类型为 `follow` 的边的属性。

现在，我们将 `degree` 的值增加 `1`。

```
nebula> UPDATE EDGE 100 -> 101 OF follow SET degree = follow.degree + 1;
```

要检查 `degree` 的值是否已更新，请输入以下语句：

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

返回以下信息：

2021 Vesoft Inc.

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

UPSERT

`UPSERT` 用于插入新的顶点或边或更新现有的顶点或边。如果顶点或边不存在，则会新建该顶点或边。 `UPSERT` 是 `INSERT` 和 `UPDATE` 的组合。

例如：

```
nebula> INSERT VERTEX player(name, age) VALUES 111:("Ben Simmons", 22); -- 插入一个新点。
nebula> UPSERT VERTEX 111 SET player.name = "Dwight Howard", player.age = $^.player.age + 11 WHEN $^.player.name == "Ben Simmons" && $^.player.age > 20 YIELD $^.player.name AS Name,
$^.player.age AS Age; -- 对该点进行 UPSERT 操作。
======================
| Name          | Age |
======================
| Dwight Howard | 33  |
----------------------
```

详情查看 UPSERT 文档。

删除数据

删除点

`DELETE VERTEX` 语句通过指定点 VID 来删除点。**同时也会删除该点的所有标签，以及和该点相邻的所有入边和出边**。

要删除 `VID 121` 的点，请输入以下语句：

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

要检查是否删除了该点，请输入以下语句；

```
nebula> FETCH PROP ON player 121;
Execution succeeded (Time spent: 1571/1910 us)
```

上面返回结果为空，表示查询操作成功，但是由于数据已被删除，未能从图空间中查询到任何数据。

删除边

您可以从图空间中删除任何边。 `DELETE EDGE` 语句通过指定边的类型以及起始点 VID 和目标点 VID 来删除边。

要删除 `VID 100` 和 `VID 200` 之间的类型为 `follow` 的边，请输入以下语句：

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

其他

查询示例

本节提供了更多查询示例供您参考。

示例1

查询球员 `VID 100` 关注 (follow) 的其他球员。

输入以下语句：

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

返回以下信息：

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


2021 Vesoft Inc.

```
| 102        |
---------------
```

示例2

查询球员 `VID 100` 关注的球员，被关注球员年龄需大于 `35` 岁。 返回其姓名和年龄，并取别名为 **Teammate** 和 **Age**。

输入以下语句：

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

返回以下信息：

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

这里：

- `YIELD` 指定希望从查询中返回的结果。
- `$$` 表示边上的目的点。
- `\` 表示换行符。

示例3

查询球员 `100` 关注的球员所效力的球队。 有两种方法可获得相同的结果：

1. 使用 `管道(|)` 来组合两个查询语句

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

   返回如下信息：

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

   这里：

   `$^` 表示边的起始点。

   `|` 表示管道。

   `$-` 表示输入流。上一个查询的输出 `(id)` 作为下一个查询的输入 `($-.id)`。

2. 使用 `自定义的变量` 来组合两个查询语句

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

   返回以下信息：

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

当一条组合语句被整体提交给服务器后，该语句内的自定义变量就已结束生命周期。

2021 Vesoft Inc.

示例 4

如果您已为数据创建索引且重构索引，则可以使用 `LOOKUP ON` 关键字进行属性查询。

请参考索引文档创建索引。

如下示例返回名称为 `Tony Parker`，标签为 *player* 的顶点。

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

详情参考 LOOKUP 文档。

批量执行

通常测试时需要执行多条数据来准备环境，可以将所有语句放入一个 `.ngql` 文件中，如下所示。

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

在服务器：

```
$ cat schema.ngql | ./bin/nebula -u <user> -p <password>
```

或者 `Docker` 中

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

这里：

- 必须将 --addr 和 --port 更改为您自己的 IP 地址和端口号。
- 可以在这里下载 `basketballplayer.ngql` 文件。

批量导入

如果您要插入数百万条记录，建议使用 csv 导入工具 和 Spark 导入工具。

最后

如果您在使用 **Nebula Graph** 的过程中遇到任何问题，请前往我们的 官方论坛 提问，将有专门的值班开发人员为您解答问题。

如果您完成了本手册的全部操作，认为 **Nebula Graph** 是一款值得尝试的图数据库产品，恳请移步 GitHub 留下您珍贵的一颗星，这将鼓舞我们继续向前。

最后更新: 2021年5月17日

2021 Vesoft Inc.

1.4.2 常见问题

本文档列出了 **Nebula Graph** 常见问题。如果您没有在文档中找到需要的信息，请尝试在 Nebula Graph 官方论坛中的用户问答分类下进行搜索。

General Information

General Information 部分列出了关于 **Nebula Graph** 的概念性问题。

查询返回时间解释

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

以上述查询为例，Time spent 中前一个数字 `7431` 为数据库本身所花费的时间，即 query engine 从 console 收到这条查询语句，到存储拿到数据，并进行一系列计算所花的时间；后一个数字 `10406` 是从客户端角度看花费的时间，即 console 从发送请求，到收到响应，并将结果输出到屏幕的时间。

Trouble Shooting

Trouble Shooting 部分列出了 **Nebula Graph** 操作中的常见错误。

服务器参数配置

在 Nebula console 中运行

```
nebula> SHOW CONFIGS;
```

详细参考运行配置要求及 CONFIGS 语法。

配置文件

配置文件默认在 `/usr/local/nebula/etc/` 下。

配置参考这里及 CONFIGS 语法。

PARTITION 分布不均

参考这里。

日志和更改日志级别

日志文件默认在 `/usr/local/nebula/logs/` 下。

参见 graphd 日志 和 storaged 日志。

使用多块硬盘

修改 `/usr/local/nebula/etc/nebula-storage.conf`。例如

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

多块硬盘时可以逗号分隔多个目录，每个目录对应一个 RocksDB 实例，以有更好的并发能力。参考这里。

进程异常 CRASH

1. 检查硬盘空间 `df -h`

2021 Vesoft Inc.

没有磁盘空间，导致服务写文件失败，服务 crash，通过以上命令查看当前磁盘的使用情况，服务配置的 `--data_path` 目录是否是满的目录。

2. 检查内存是否足够 `free -h`

   服务使用过多内存，被系统杀掉，通过 `dmesg` 查看是否有 OOM 的记录，记录是否有 nebula 关键字。

3. 检查日志

使用 DOCKER 启动后，执行命令时报错

可能的原因是 Docker 的 IP 地址和默认配置中的监听地址不一致（默认是 172.17.0.2），因此这里需要修改默认配置中的监听地址。

1. 首先在容器中执行 `ifconfig` 命令，查看您的容器地址，这里假设您的容器地址是 172.17.0.3，那么就意味着您需要修改默认配置的 IP 地址。
2. 然后进入配置目录 `cd /usr/local/nebula/etc`，查找所有 IP 地址配置的位置（ `grep "172.17.0.2" . -r` ）。
3. 修改上一步查到的所有 IP 地址为您的容器地址（172.17.0.3）。
4. 最后重新启动所有服务（ `/usr/local/nebula/scripts/nebula.service restart all` ）。

单机先后加入两个不同集群

同一台主机先后用于单机测试和集群测试，storaged 服务无法正常启动（终端上显示的 storaged 服务的监听端口为红色）。查看 storaged 服务的日志（ `/usr/local/nebula/nebula-storaged.ERROR` ），若发现 "wrong cluster" 的报错信息，则可能的出错原因是单机测试和集群测试时的 Nebula Graph 生成的 cluster id 不一致，需要删除 Nebula Graph 安装目录（ `/usr/local/nebula` ）下的 cluster.id 文件和 data 目录，然后重启服务。

连接失败

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

检查服务是否存在

```
$ /usr/local/nebula/scripts/nebula.service status all
或者
nebula> SHOW HOSTS;
```

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

1. 检查硬盘空间 `df -h`
2. 检查日志目录 `/usr/local/nebula/logs/`
3. 修改允许打开的最大文件数 `ulimit -n 65536`

如何查看 NEBULA GRAPH 版本信息

使用 `curl http://ip:port/status` 命令获取 git_info_sha、binary 包的 commitID。

修改配置文件不生效

Nebula Graph 使用如下两种方式获取配置：

1. 从配置文件中（需要修改配置文件并重启服务）；
2. 从 Meta 服务中。通过 CLI 设置，并持久化保存在 Meta 服务中，详情参考这里；

修改了配置文件不生效，是因为默认情况下，**Nebula Graph** 的配置参数管理采用第二种方式 （Meta），如果希望采用第一种方式，需要在 `/usr/local/nebula/etc/` 配置文件 `metad.conf`、 `storaged.conf`、 `graphd.conf` 中分别添加 `--local_config=true` 选项。

修改 ROCKSDB BLOCK CACHE

更改 storage 的配置文件 `storaged.conf` （默认路径为 `/usr/local/nebula/etc/` ）并重启，例如：

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

参见这里

2021 Vesoft Inc.

使用 CENTOS 6.5 NEBULA 服务失败

在 CentOS 6.5 部署 Nebula Graph 失败，报错信息如下：

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

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

此时服务状态为：

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

出错原因：CentOS 6.5 系统内核版本为 2.6.32， `SO_REUSEPORT` 仅支持 `Linux 3.9` 及以上版本。

将系统升级到 CentOS 7.5 问题可自行解决。

MAX_EDGE_RETURNED_PER_VERTEX 和 WHERE 条件执行的优先顺序

如果已设置 max_edge_returned_per_vertex=10，使用 WHERE 进行过滤时，实际边数量大于 10，此时返回 10 条边，还是所有边？

Nebula Graph 先进行 WHERE 条件过滤，如果实际边数量少于 10， 则返回实际边数量。反之，则根据 max_edge_returned_per_vertex 限制返回 10 条边。

使用 FETCH 返回数据时，有时可以返回数据，有时返回为空

请检查是否在同一个节点启动了两个 storage，并且这两个 storage 配置的端口号相同。假如存在以上情况，请修改其中一个 storage 端口号，然后重新导入数据。

最后更新: 2020年7月21日

2021 Vesoft Inc.

1.4.3 CSV文件导入示例

以下示例将指导您如何使用 **Nebula Importer** 将 CSV 数据导入到 **Nebula Graph** 中。在此示例中，**Nebula Graph** 通过 `Docker` 和 `Docker Compose` 安装。我们将通过以下步骤引导您完成该示例：

- CSV文件导入示例
  - 启动 **Nebula Graph** 服务
  - 创建点和边的 Schema
  - 准备配置文件
  - 准备 CSV 数据
  - 导入 CSV 数据
    - 使用 Go-importer 导入 CSV 数据

启动 Nebula Graph 服务

您可以按照以下步骤启动 **Nebula Graph** 服务：

1. 在命令行界面上，进入 `nebula-docker-compose` 目录。
2. 执行以下命令以启动 **Nebula Graph** 服务：

```
$ sudo docker-compose up -d
```

1. 执行以下命令把 **Nebula Graph** 镜像文件下拉到本地：

```
$ sudo docker pull vesoft/nebula-console:nightly
```

1. 执行以下命令以连接 **Nebula Graph** 服务器：

```
$ sudo docker run --rm -ti --network=host vesoft/nebula-console:nightly --addr=127.0.0.1 --port=3699
```

注意：您必须确保 IP 地址和端口号配置正确。

创建点和边的 Schema

在输入 schema 之前，必须创建一个空间并使用它。在此示例中，我们创建一个 **nba** 空间并使用它。

我们使用以下命令创建两个标签和两个边类型：

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

nebula> CREATE TAG team (name string);

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

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

准备配置文件

您必须配置 `.yaml` 配置文件，该文件规定了 CSV 文件中数据的格式。在本例中，我们创建一个名为 `config.yaml` 的配置文件。

在此示例中，我们按以下方式配置 `config.yaml` 文件：

```
version: v1rc1
description: example
clientSettings:
  concurrency: 2 # number of graph clients
  channelBufferSize: 50
  space: nba
  connection:
    user: user
    password: password
```


2021 Vesoft Inc.

```
    address: 127.0.0.1:3699
logPath: ./err/test.log
files:
  - path: /home/nebula/serve.csv
    failDataPath: ./err/serve.csv
    batchSize: 10
    type: csv
    csv:
      withHeader: false
      withLabel: false
    schema:
      type: edge
      edge:
        name: serve
        withRanking: false
        props:
          - name: start_year
            type: int
          - name: end_year
            type: int
  - path: /home/nebula/follow.csv
    failDataPath: ./err/follow.csv
    batchSize: 10
    type: csv
    csv:
      withHeader: false
      withLabel: false
    schema:
      type: edge
      edge:
        name: follow
        withRanking: false
        props:
          - name: degree
            type: int
  - path: /home/nebula/player.csv
    failDataPath: ./err/player.csv
    batchSize: 10
    type: csv
    csv:
      withHeader: false
      withLabel: false
    schema:
      type: vertex
      vertex:
        tags:
          - name: player
            props:
              - name: name
                type: string
              - name: age
                type: int
  - path: /home/nebula/team.csv
    failDataPath: ./err/team.csv
    batchSize: 10
    type: csv
    csv:
      withHeader: false
      withLabel: false
    schema:
      type: vertex
      vertex:
        tags:
          - name: team
            props:
              - name: name
                type: string
```

注意：

- 在上面的配置文件中，您必须将 IP 地址和端口号更改为您自己的 IP 地址和端口号。
- 您必须将 CSV 文件的目录更改为您自己的目录，否则，**Nebula Importer** 无法找到 CSV 文件。

准备 CSV 数据

在此示例中，我们准备了四个 CSV 数据文件：`player.csv`、`team.csv`、`serve.csv` 以及 `follow.csv`。

`serve.csv` 文件中的数据如下：

```
100,200,1997,2016
101,201,1999,2018
102,203,2006,2015
102,204,2015,2019
103,204,2017,2019
104,200,2007,2009
```


2021 Vesoft Inc.

`follow.csv` 文件中的数据如下：

```
100,101,95
100,102,90
101,100,95
102,101,75
102,100,75
103,102,70
104,101,50
104,105,60
105,104,83
```

`player.csv` 文件中的数据如下：

```
100,Tim Duncan,42
101,Tony Parker,36
102,LaMarcus Aldridge,33
103,Rudy Gay,32
104,Marco Belinelli,32
105,Danny Green,31
106,Kyle Anderson,25
107,Aron Baynes,32
108,Boris Diaw,36
```

`team.csv` 文件中的数据如下：

```
200,Warriors
201,Nuggets
202,Rockets
203,Trail
204,Spurs
205,Thunders
206,Jazz
207,Clippers
208,Kings
```

注意：

- 在 `serve` 和 `follow` CSV 文件中，第一列是起始点 ID、第二列是目标点 ID，其他列与 `config.yaml` 文件一致。
- 在 `player` 和 `team` CSV 文件中，第一列是点 ID，其他列与 `config.yaml` 文件一致。

导入 CSV 数据

完成上述所有四个步骤后，您可以使用 `Docker` 或 `Go` 导入 CSV 数据。

使用 GO-IMPORTER 导入 CSV 数据

请参见Nebula Importer。

最后更新: 2021年4月25日

2021 Vesoft Inc.

1.5 系统设计与架构

1.5.1 Nebula Graph 的整体架构

一个完整的 **Nebula Graph** 部署集群包含三个服务，即 Query Service，Storage Service 和 Meta Service。每个服务都有其各自的可执行二进制文件，这些二进制文件既可以部署在同一组节点上，也可以部署在不同的节点上。

Meta Service

上图为 **Nebula Graph** 的架构图，其右侧为 Meta Service 集群，它采用 leader/follower 架构。Leader 由集群中所有的 Meta Service 节点选出，然后对外提供服务。Followers 处于待命状态并从 leader 复制更新的数据。一旦 leader 节点 down 掉，会再选举其中一个 follower 成为新的 leader。

Meta Service 不仅负责存储和提供图数据的 meta 信息，如 schema、partition 信息等，还同时负责指挥数据迁移及 leader 的变更等运维操作。

存储计算分离

在架构图中 Meta Service 的左侧，为 **Nebula Graph** 的主要服务，**Nebula Graph** 采用存储与计算分离的架构，虚线以上为计算，以下为存储。

存储计算分离有诸多优势，最直接的优势就是，计算层和存储层可以根据各自的情况弹性扩容、缩容。

存储计算分离还有另一个优势：使水平扩展成为可能。

此外，存储计算分离使得 Storage Service 可以为多种类型的个计算层或者计算引擎提供服务。当前 Query Service 是一个高优先级的计算层，而各种迭代计算框架会是另外一个计算层。

无状态计算层

现在来看下计算层，每个计算节点都运行着一个无状态的查询计算引擎，而节点彼此间无任何通信关系。计算节点仅从 Meta Service 读取 meta 信息，以及和 Storage Service 进行交互。这样设计使得计算层集群更容易使用 K8s 管理或部署在云上。

计算层的负载均衡有两种形式，最常见的方式是在计算层上加一个负载均衡 (balance)，第二种方法是将计算层所有节点的 IP 地址配置在客户端中，这样客户端可以随机选取计算节点进行连接。

每个查询计算引擎都能接收客户端的请求，解析查询语句，生成抽象语法树（AST）并将 AST 传递给执行计划器和优化器，最后再交由执行器执行。

2021 Vesoft Inc.

Shared-nothing 分布式存储层

Storage Service 采用 shared-nothing 的分布式架构设计，每个存储节点都有多个本地 KV 存储实例作为物理存储。**Nebula Graph** 采用多数派协议 Raft 来保证这些 KV 存储之间的一致性（由于 Raft 比 Paxos 更简洁，我们选用了 Raft ）。在 KVStore 之上是图语义层，用于将图操作转换为下层 KV 操作。

图数据（点和边）通过 Hash 的方式存储在不同 Partition 中。这里用的 Hash 函数实现很直接，即 vertex_id 取余 Partition 数。在 **Nebula Graph** 中，Partition 表示一个虚拟的数据集，这些 Partition 分布在所有的存储节点，分布信息存储在 Meta Service 中（因此所有的存储节点和计算节点都能获取到这个分布信息）。

最后更新: 2020年4月29日

2021 Vesoft Inc.

1.5.2 存储层设计

摘要

Nebula Graph 的 Storage 包含两个部分， 一是 meta 相关的存储， 称之为 `Meta Service` ，另一个是 data 相关的存储， 称之为 `Storage Service` 。 这两个服务是两个独立的进程，数据也完全隔离，当然部署也是分别部署， 不过两者整体架构相差不大。 如果没有特殊说明，本文中 Storage Service 代指 data 的存储服务。

架构

图一 storage service 架构图

如图1 所示，Storage Service 共有三层，最底层是 Store Engine，它是一个单机版 local store engine，提供了对本地数据的 `get` / `put` / `scan` / `delete` 操作，相关的接口放在 `KVStore/KVEngine.h` 文件里面，用户完全可以根据自己的需求定制开发相关 local store plugin，目前 **Nebula Graph** 提供了基于 RocksDB 实现的 Store Engine。

在 local store engine 之上，便是 Consensus 层，实现了 Multi Group Raft，每一个 Partition 都对应了一组 Raft Group，这里的 Partition 便是数据分片。目前 **Nebula Graph** 的分片策略采用了 `静态 Hash` 的方式。用户在创建 SPACE 时需指定 Partition 数，Partition 数量一旦设置便不可更改，一般来讲，Partition 数目要能满足业务将来的扩容需求。

在 Consensus 层上面也就是 Storage Service 的最上层，便是 Storage interfaces，这一层定义了一系列和图相关的 API。 这些 API 请求会在这一层被翻译成一组针对相应 Partition 的 kv 操作。正是这一层的存在，使得存储服务变成了真正的图存储，否则，Storage Service 只是一个 kv 存储。而 **Nebula Graph** 没把 kv 作为一个服务单独提出，其最主要的原因便是图查询过程中会涉及到大量计算，这些计算往往需要使用图的 schema，而 kv 层是没有数据 schema 概念，这样设计会比较容易实现计算下推。

2021 Vesoft Inc.

Schema & Partition

数据结构上，图的主要数据是点和边。但 **Nebula Graph** 存储的是属性图：除了点和边以外，还存储了对应的属性，以便更高效地使用属性过滤。

对于点来说，**Nebula Graph** 使用不同的 Tag 表示不同类型的点，同一个 VertexID 可以关联多个 Tag，而每一个 Tag 都有自己对应的属性。对应到 kv 存储里面，**Nebula Graph** 使用 vertexID + TagID 来表示 key，把相关的属性编码后放在 value 里面，具体 key 的 format 如图2 所示：

图二 Vertex Key Format

- `Type` ： 1 个字节，用来表示 key 类型，当前的类型有 data, index, system 等
- `Part ID` ：3 个字节，用来表示数据分片 Partition，此字段主要用于 **Partition 重新分布 (balance) 时方便根据前缀扫描整个 Partition 数据**
- `Vertex ID` ：8 个字节，用来表示点的 ID
- `Tag ID` ：4 个字节, 用来表示关联的某个 tag
- `Timestamp` ：8 个字节，对用户不可见，未来实现分布式事务 (MVCC) 时使用

在一个图中，每一条逻辑意义上的边，在 Nebula Graph 中会建模成两个独立的 key-value，分别称为 out-key 和 in-key。out-key 与这条边所对应的起点存储在同一个 partition 上，in-key 与这条边所对应的终点存储在同一个partition 上。通常来说，out-key 和 in-key 会分布在两个不同的 Partition 中。

两个点之间可能存在多种类型的边，**Nebula Graph** 用 Edge Type 来表示边类型。而同一类型的边可能存在多条，比如，定义一个 edge type "转账"，用户 A 可能多次转账给 B， 所以 **Nebula Graph** 又增加了一个 Rank 字段来做区分，表示 A 到 B 之间多次转账记录。 Edge key 的 format 如图 3 所示：

图三 Edge Key Format

- `Type` ：1 个字节，用来表示 key 的类型，当前的类型有 data, index, system 等。
- `Part ID` ：3 个字节，用来表示数据分片 Partition，此字段主要用于 **Partition 重新分布 (balance) 时方便根据前缀扫描整个 Partition 数据**
- `Vertex ID` ：8 个字节，出边里面用来表示源点的 ID， 入边里面表示目标点的 ID。
- `Edge Type` ：4 个字节，用来表示这条边的类型，如果大于 0 表示出边，小于 0 表示入边。
- `Rank` ：8 个字节，用来处理同一种类型的边存在多条的情况。用户可以根据自己的需求进行设置，这个字段可*存放交易时间、交易流水号、或某个排序*
- `Vertex ID` ：8 个字节，出边里面用来表示目标点的 ID， 入边里面表示源点的 ID。
- `Timestamp` ：8 个字节，对用户不可见，未来实现分布式做事务的时候使用。

针对 Edge Type 的值，若如果大于 0 表示出边，则对应的 edge key format 如图4 所示；若 Edge Type 的值小于 0，则对应的 edge key format 如图5 所示

2021 Vesoft Inc.

Type (1 byte)	Part ID (3 bytes)	srcVertex ID (8 bytes)	Edge Type (4 bytes)	Rank (8 bytes)	dstVertex ID (8 bytes)	Timestamp (8 bytes)

图4 出边的 Key Format

Type (1 byte)	Part ID (3 bytes)	dstVertex ID (8 bytes)	Edge Type (4 bytes)	Rank (8 bytes)	srcVertex ID (8 bytes)	Timestamp (8 bytes)

图5 入边的 Key Format

对于点或边的属性信息，有对应的一组 kv pairs，**Nebula Graph** 将它们编码后存在对应的 value 里。由于 **Nebula Graph** 使用强类型 schema，所以在解码之前，需要先去 Meta Service 中取具体的 schema 信息。另外，为了支持在线变更 schema，在编码属性时，会加入对应的 schema 版本信息。

数据的分片方式为对 Vertex ID 取模 。通过对 Vertex ID 取模，同一个点的所有*出边*，*入边*以及这个点上所有关联的 *Tag 信息*都会被分到同一个 Partition，这种方式大大地提升了查询效率。对于在线图查询来讲，最常见的操作便是从一个点开始向外 BFS（广度优先）拓展，于是拿一个点的出边或者入边是最基本的操作，而这个操作的性能也决定了整个遍历的性能。BFS 中可能会出现按照某些属性进行剪枝的情况，**Nebula Graph** 通过将属性与点边存在一起，来保证整个操作的高效。在实际的场景中，大部分情况都是属性图，并且实际中的 BFS 也需要进行大量的剪枝操作。

KVStore

对于KVStore的要求：

- **性能**；
- **以 library 的形式提供**：对于强 schema 的 **Nebula Graph** 来讲，计算下推需要 schema 信息，而计算下推实现的好坏，是 **Nebula Graph** 是否高效的关键；
- **数据强一致**：这是分布式系统决定的；
- **使用 C++实现**：这由团队的技术特点决定；

基于上述要求，**Nebula Graph** 实现了自己的 KVStore。当然，对于性能完全不敏感且不太希望搬迁数据的用户来说，**Nebula Graph** 也提供了整个 KVStore 层的 plugin，直接将 Storage Service 搭建在第三方的 KVStore 上面，目前官方提供的是 HBase 的 plugin。

Nebula Graph KVStore 主要采用 RocksDB 作为本地的存储引擎，对于多硬盘机器，为了充分利用多硬盘的并发能力，**Nebula Graph** 支持自己管理多块盘，用户只需配置多个不同的数据目录即可。

分布式 KVStore 的管理由 Meta Service 来统一调度，它记录了所有 Partition 的分布情况，以及当前机器的状态，当用户增减机器时，只需要通过 console 输入相应的指令，Meta Service 便能够生成整个 balance plan 并执行。（之所以没有采用完全自动 balance 的方式，主要是为了减少数据搬迁对于线上服务的影响，balance 的时机由用户自己控制，通常会在业务低谷进行。）

为了方便对于 WAL 进行定制，**Nebula Graph** KVStore 实现了自己的 WAL 模块，每个 partition 都有自己的 WAL，这样在追数据时，不需要进行 wal split 操作， 更加高效。 另外，为了实现一些特殊的操作，专门定义了 Command Log 这个类别，这些 log 只为了使用 Raft 来通知所有 replica 执行某一个特定操作，并没有真正的数据。除了 Command Log 外，**Nebula Graph** 还提供了一类日志来实现针对某个 Partition 的 atomic operation，例如 CAS，read-modify-write，它充分利用了Raft 串行的特性。

关于多图空间（space）的支持：一个 Nebula Graph KVStore 集群可以支持多个 space，每个 space 可设置自己的 partition 数和 replica 数。不同 space 在物理上是完全隔离的，而且在同一个集群上的不同 space 可支持不同的 store engine 及分片策略。

Raft

作为一个分布式系统，KVStore 的 replication、scale out 等功能需 Raft 的支持。主要介绍 Nebula Graph Raft 的一些特点以及工程实现。

2021 Vesoft Inc.

MULTI RAFT GROUP

由于 Raft 的日志不允许空洞，几乎所有的实现都会采用 Multi Raft Group 来缓解这个问题，因此 partition 的数目几乎决定了整个 Raft Group 的性能。但这也并不是说 Partition 的数目越多越好：每一个 Raft Group 内部都要存储一系列的状态信息，并且每一个 Raft Group 有自己的 WAL 文件，因此 Partition 数目太多会增加开销。此外，当 Partition 太多时， 如果负载没有足够高，batch 操作是没有意义的。比如，对于一个有 1万 TPS 的线上系统，即使它的每台机器上 partition 的数目超过 1万，但很有可能每个 partition TPS 只有 1，这样 batch 操作就失去了意义，还增加了 CPU 开销。

实现 Multi Raft Group 的最关键之处有两点，**第一是共享 Transport 层**，因为每一个 Raft Group 内部都需要向对应的 peer 发送消息，如果不能共享 Transport 层，连接的开销巨大；**第二是线程模型**，Multi Raft Group 一定要共享一组线程池，否则会造成系统的线程数目过多，导致大量的 context switch 开销。

BATCH

对于每个 Partition来说，由于串行写 WAL，为了提高吞吐，做 batch 是十分必要的。Nebula Graph 利用每个 part 串行的特点，做了一些特殊类型的 WAL，带来了一些工程上的挑战。

举个例子，Nebula Graph 利用 WAL 实现了无锁的 CAS 操作，而每个 CAS 操作需要之前的 WAL 全部 commit 之后才能执行，所以对于一个 batch，如果中间夹杂了几条 CAS 类型的 WAL, 还需要把这个 batch 分成粒度更小的几个 group，group 之间保证串行。还有，command 类型的 WAL 需要它后面的 WAL 在其 commit 之后才能执行，所以整个 batch 划分 group 的操作工程实现上比较有特色。

LEARNER

Learner 这个角色的存在主要是为了 `应对扩容` 时，新机器需要“追”相当长一段时间的数据，而这段时间有可能会发生意外。如果直接以 follower 的身份开始追数据，就会使得整个集群的 HA 能力下降。 Nebula Graph 里面 learner 的实现就是采用了上面提到的 command wal。 Leader 在写 wal 时如果碰到 add learner 的 command， 就会将 learner 加入自己的 peers，并把它标记为 learner，这样在统计多数派的时候，就不会算上 learner，但是日志还是会照常发送给它们。当然 learner 也不会主动发起选举。

TRANSFER LEADERSHIP

Transfer leadership 这个操作对于 balance 来讲至关重要，当把某个 Partition 从一台机器挪到另一台机器时，首先便会检查 source 是不是 leader，如果是的话，需要先把他挪到另外的 peer 上面；在搬迁数据完毕之后，通常还要把 leader 进行一次 balance，这样每台机器承担的负载也能保证均衡。

实现 transfer leadership， 需要注意的是 leader 放弃自己的 leadership，和 follower 开始进行 leader election 的时机。对于 leader 来讲，当 transfer leadership command 在 commit 的时候，它放弃 leadership；而对于 follower 来讲，当收到此 command 的时候就要开始进行 leader election， 这套实现要和 Raft 本身的 leader election 走一套路径，否则很容易出现一些难以处理的 corner case。

MEMBERSHIP CHANGE

为了避免脑裂，当一个 Raft Group 的成员发生变化时，需要有一个中间状态， 这个状态下 old group 的多数派与 new group 的多数派总是有 overlap，这样就防止了 old group 或者新 group 单方面做出决定，这就是论文中提到的 `joint consensus` 。为了更加简化，Diego Ongaro 在自己的博士论文中提出**每次增减一个 peer 的方式，以保证 old group 的多数派总是与 new group 的多数派有 overlap**。 Nebula Graph 的实现也采用了这个方式，只不过 add member 与 remove member 的实现有所区别，具体实现方式可以参考 Raft Part class 里面 `addPeer/removePeer` 的实现。

SNAPSHOT

Snapshot 如何与 Raft 流程结合起来，论文中并没有细讲，但是这一部分是一个 Raft 实现里最容易出错的地方，因为这里会产生大量的 corner case。

举一个例子，当 leader 发送 snapshot 过程中，如果 leader 发生了变化，该怎么办？ 这个时候，有可能 follower 只接到了一半的 snapshot 数据。 所以需要有一个 Partition 数据清理过程，由于多个 Partition 共享一份存储，因此如何清理数据又是一个很麻烦的问题。另外，snapshot 过程中，会产生大量的 IO，为了性能考虑，不希望这个过程与正常的 Raft 共用一个 IO threadPool，并且整个过程中，还需要使用大量的内存，如何优化内存的使用，对于性能十分关键。由于篇幅原因，并不会在本文对这些问题展开讲述，可以参考 `SnapshotManager` 的实现。

2021 Vesoft Inc.

Storage Service

在 KVStore 的接口之上，Nebula Graph 封装有图语义接口，主要的接口如下：

- `getNeighbors` ：查询一批点的出边或者入边，返回边以及对应的属性，并且需要支持条件过滤；
- `Insert vertex/edge` ：插入一条点或者边及其属性；
- `getProps` ：取一个点或者一条边的属性；

这一层会将图语义的接口转化成 kv 操作。为了提高遍历的性能，还要做并发操作。

Meta Service

在 KVStore 的接口上，Nebula Graph 也同时封装了一套 meta 相关的接口。Meta Service 不但提供了图 schema 的增删查改的功能，还提供了集群的管理功能以及用户鉴权相关的功能。Meta Service 支持单独部署，也支持使用多副本来保证数据的安全。

最后更新: 2020年5月14日

2021 Vesoft Inc.

1.5.3 查询引擎设计

在 **Nebula Graph** 中，Query Engine 用来处理 **Nebula Graph** 查询语言语句（nGQL）。本篇文章将带你了解 Nebula Query Engine 的架构。

上图为查询引擎的架构图，如果你对 SQL 的执行引擎比较熟悉，那么对上图一定不会陌生。**Nebula Graph** 的 Query Engine 架构图和现代 SQL 的执行引擎类似，只是在查询语言解析器和具体的执行计划有所区别。

Session Manager

Nebula Graph 权限管理采用基于角色的权限控制（Role Based Access Control）。客户端第一次连接到 Query Engine 时需作认证，当认证成功之后 Query Engine 会创建一个新 session，并将该 session ID 返回给客户端。所有的 session 统一由 Session Manager 管理。session 会记录当前的 graph space 信息及对该 space 的权限。此外，session 还会记录一些会话相关的配置信息，并临时保存同一 session 内的跨多个请求的一些信息。

客户端连接结束之后 session 会关闭，或者如果长时间没通信会切换为空闲状态。这个空闲时长是可以配置的。

客户端的每次请求都必须带上此 session ID，否则 Query Engine 会拒绝此请求。

Storage Engine 不管理 session，Query Engine 在访问存储引擎时，会带上 session 信息。

Parser

Query Engine 解析来自客户端的 nGQL 语句，分析器（parser）主要基于著名的 flex / bison 工具集。字典文件（lexicon）和语法规则（syntax）在 **Nebula Graph** 源代码的 `src/parser` 目录下。设计上，nGQL 的语法非常接近 SQL，目的是降低学习成本。

图数据库目前没有统一的查询语言国际标准，一旦 ISO/IEC 的图查询语言（GQL）委员会发布 GQL 国际标准，nGQL 会尽快去实现兼容。

Parser 构建产出的抽象语法树（Abstract Syntax Tree，简称 AST）会交给下一模块：Execution Planner。

Execution Planner

执行计划器（Execution Planner）负责将抽象树 AST 解析成一系列执行动作 action（可执行计划）。action 为最小可执行单元。例如，典型的 action 可以是获取某个节点的所有邻节点，或者获得某条边的属性，或基于特定过滤条件筛选节点或边。

当抽象树 AST 被转换成执行计划时，所有 ID 信息会被抽取出来以便执行计划的复用。这些 ID 信息会放置在当前请求 context 中，context 也会保存变量和中间结果。

2021 Vesoft Inc.

Optimization

经由 Execution Planner 产生的执行计划会交给优化框架 Optimization Framework。优化框架中注册有多个 Optimizer。Optimizer 会依次被调用对执行计划进行优化，这样每个 Optimizer 都有机会修改（优化）执行计划。

最后，优化过的执行计划可能和原始执行计划完全不一样，但是优化后的执行结果必须和原始执行计划的结果一样。

Execution

Query Engine 最后一步是去执行优化后的执行计划，这步是执行框架（Execution Framework）完成的。执行层的每个执行器一次只处理一个执行计划，计划中的 action 会依次执行。执行器也会做一些有限的局部优化，比如：决定是否并发执行。

针对不同 action，执行器将通过客户端与 meta service 或 storage engine 进行通信。

最后更新: 2020年5月14日

2021 Vesoft Inc.

2. 查询语言

2.1 面向的读者

本章介绍 **Nebula Graph** 的查询语言，适合所有使用 **Nebula Graph** 的用户。

2.1.1 示例数据

Nebula Graph 查询语句中使用的示例数据可以在此处下载。示例数据下载完成后可以通过 Nebula Graph Studio 把数据导入到 **Nebula Graph** 数据库中。

2.1.2 占位标识符和占位符值

Nebula Graph 查询语言 nGQL 参照以下标准设计：

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

在模板代码中，任何非关键词，文字或标点符号的标记均为占位标识符或占位符值。

符号标记使用见下表：

符号	含义
< >	必选项
::=	定义元素的公式
[ ]	可选项
{ }	明确指定的元素
|	表示在其左右两边任选一项
...	元素可重复多次

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.2 数据类型

2.2.1 数据类型

Nebula Graph 支持的内建数据类型如下：

数值型

整型

整型的关键字为 `int`，为 64 位**有符号**整型，范围是 `[-9223372036854775808, 9223372036854775807]`。整型常量支持多种格式：

1. 十进制，例如 `123456` 2. 十六进制，例如 `0xdeadbeaf` 3. 八进制，例如 `01234567`

双浮点型

双精度浮点数的关键字为 `double`，且没有上限和下限。

布尔型

布尔型关键字为 `bool`，字面常量为 `true` 和 `false`。

字符型

字符型关键字为 `string`，字面常量为双引号或单引号包围的任意长度的字符序列，字符串中间不允许换行。例如 `"Shaquile O'Neal"`，`'"This is a double-quoted literal string"'`。字符串内支持嵌入转义序列，例如：

1. `"\n\t\r\b\f"` 1. `"\110ello world"`

时间戳类型

- 时间戳类型的取值范围为 `1970-01-01 00:00:01 UTC` 到 `2262-04-11 23:47:16 UTC`
- 时间戳单位为秒
- 插入数据的时候，支持插入方式
  - 调用函数 now()
  - 时间字符串，例如："2019-10-01 10:00:00"
  - 直接输入时间戳，即从 1970-01-01 00:00:00 开始的秒数
- 当插入的时间格式为字符串时，做数据存储的时候，会先将时间根据服务端时区转化为 UTC 时间，读取的时候 console 会将存储的 UTC 时间戳转换为本地时间给用户。
- 底层存储数据类型为: **int64**

示例

先创建一个名为 school 的 tag

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

插入一个点，名为 "xiwang"，建校时间为 "2010-09-01 08:00:00"

```
nebula> INSERT VERTEX school(name, create_time) VALUES hash("xiwang"):("xiwang", "2010-09-01 08:00:00");
```

插入一个点，名为 "guangming"，建校时间为现在

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


2021 Vesoft Inc.

最后更新: 2020年6月8日

2021 Vesoft Inc.

2.2.2 类型转换

在 nGQL 中，类型转换分为隐式转换和显式转换。

隐式转换

在表达式中，兼容类型间可自动完成类型转换：

1. 以下类型均可隐式转换至 `bool` 类型：
    - 当且仅当字符串长度为 `0` 时，可被隐式转换为 `false`，否则为 `true`
    - 当且仅当整型数值为 `0` 时，可被隐式转换为 `false`，否则为 `true`
    - 当且仅当浮点类型数值为 `0.0` 时，可被隐式转换为 `false`，否则为 `true`
2. `int` 类型可隐式转换为 `double` 类型

显式转换

除隐式类型转换外，在符合语义的情况下，还可以使用显式类型转换。语法规则类似 C 语言：`(type_name)expression` 。例如，`YIELD length((string)(123)), (int)"123" + 1` 执行结果为 `3, 124` 。`YIELD (int)(TRUE)` 执行结果为 `1` 。`YIELD (int)("12ab3")` 则会转换失败。

最后更新: 2020年8月11日

2021 Vesoft Inc.

2.3 函数与操作符

2.3.1 位运算符

名称	*描述*
&	按位与
|	按位或
^	按位异或

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.3.2 内建函数

Nebula Graph 支持在表达式中调用如下类型的内建函数。

数学相关

函数	描述
double abs(double x)	返回绝对值
double floor(double x)	返回小于参数的最大整数（向下取整）
double ceil(double x)	返回大于参数的最小整数（向上取整）
double round(double x)	对参数取整，如果参数位于中间位置，则返回远离 0 的数字
double sqrt(double x)	返回参数的平方根
double cbrt(double x)	返回参数的立方根
double hypot(double x, double y)	返回一个直角三角形的斜边
double pow(double x, double y)	返回 x 的 y 次幂
double exp(double x)	计算 e 的 x 次幂
double exp2(double x)	返回 2 的指定次幂
double log(double x)	返回参数的自然对数
double log2(double x)	返回底数为 2 的对数
double log10(double x)	返回底数为 10 的对数
double sin(double x)	返回正弦函数值
double asin(double x)	返回反正弦函数值
double cos(double x)	返回余弦函数值
double acos(double x)	返回反余弦函数值
double tan(double x)	返回正切函数值
double atan(double x)	返回反正切函数值
int rand32()	返回 32bit 整型伪随机数
int rand32(int max)	返回 [0, max) 区间内的 32bit 整型伪随机数
int rand32(int min, int max)	返回 [min, max) 区间内的 32bit 整型伪随机数
int rand64()	返回 64bit 整型伪随机数
int rand64(int max)	返回 [0, max) 区间内的 64bit 整型伪随机数
int rand64(int min, int max)	返回 [min, max) 区间内的 64bit 整型伪随机数

2021 Vesoft Inc.

字符串相关

注意： 和 SQL 一样，nGQL 的字符索引（位置）从 `1` 开始，而不是类似 C 语言从 `0` 开始。

函数	描述
int strcasecmp(string a, string b)	大小写不敏感的字符串比较，相等时返回零，a > b 时返回值大于零，否则返回值小于零
string lower(string a)	将字符串转换为小写
string upper(string a)	将字符串转换为大写
int length(string a)	返回字符串长度（整数）（目前实现为，返回占用字节数）
string trim(string a)	删除字符串两端的空白字符（空格，换行，制表符等）
string ltrim(string a)	删除字符串起始的空白字符
string rtrim(string a)	删除字符串末尾的空白字符
string left(string a, int count)	返回 [1, count] 范围内的子串，若字符串长度小于 count ，则返回原字符串
string right(string a, int count)	返回 [size - count + 1, size] 范围内的子串，若字符串长度小于 count ，则返回原字符串
string lpad(string a, int size, string letters)	使用字符串 letters 从左侧填充字符串至其长度不小于 size
string rpad(string a, int size, string letters)	使用字符串 letters 从右侧填充字符串至其长度不小于 size
string substr(string a, int pos, int count)	从指定起始位置 pos 开始，获取长度为 count 的子串
int hash(string a)	对数值进行 hash，返回值类型为整数

函数 `substr` 返回结果**注释**：

- 如果 pos 等于 0 ，返回空串
- 如果 pos 绝对值大于原字符串的长度，返回空串
- 如果 pos 大于 0 ，返回 [pos, pos + count) 范围内的子串
- 如果 pos 小于 0 ，设起始位置 N 为 length(a) + pos + 1 ，返回 [N, N + count) 范围内的子串
- 如果 count 大于 length(a) ，则返回整个字符串

时间相关

函数	描述
int now()	返回当前时间戳

最后更新: 2020年5月12日

2021 Vesoft Inc.

2.3.3 比较函数和运算符

运算符	描述
=	赋值运算符
/	除法运算符
==	等于运算符
!=	不等于运算符
<	小于运算符
<=	小于或等于运算符
-	减法运算符
%	余数运算符
+	加法运算符
*	乘法运算符
-	负号运算符
udf_is_in()	比较函数，判断值是否在指定的列表中

比较运算的结果是 *true* 或 *false* 。

- ==

等于。String的比较大小写敏感。不同类的值不相同：

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

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

- >

大于：

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

- ≥

大于或等于：

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

- <

小于：

2021 Vesoft Inc.

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

- ≤

小于或等于：

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

- !=

不等于：

```
nebula> YIELD 1 != '1';
[ERROR (-8)]: A string type can not be compared with a non-string type.
```

- udf_is_in()

第一个参数为要比较的值。

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

nebula> GO FROM 100 OVER follow WHERE udf_is_in($$.player.name, "Tony Parker"); /*由于udf_is_in 后面可能变更，所以该示例可能失效。*/
===============
| follow._dst |
===============
| 101         |
---------------

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

最后更新: 2020年5月12日

2021 Vesoft Inc.

2.3.4 分组 (Group By)

`GROUP BY` 类似于 SQL。 只能与 `YIELD` 语句一起使用。

名称	描述
AVG()	返回参数的平均值
COUNT()	返回记录值总数
COUNT_DISTINCT())	返回独立记录值的总数
MAX()	返回最大值
MIN()	返回最小值
STD()	返回总体标准差
SUM()	返回总合
BIT_AND()	按位与
BIT_OR()	按位或
BIT_XOR()	按位异或
以上函数只作用于 int64 和 double。	

示例

```
nebula> GO FROM 100 OVER follow YIELD $$.player.name as Name | GROUP BY $-.Name YIELD $-.Name, COUNT(*);
-- 从节点 100 出发，查找其关注的球员并返回球员的姓名作为 Name，按照姓名对球员分组并统计每个分组的人数。
-- 返回以下结果：
===============================
| $-.Name           | COUNT(*) |
===============================
| Kyle Anderson     | 1        |
-------------------------------
| Tony Parker       | 1        |
-------------------------------
| LaMarcus Aldridge | 1        |
-------------------------------

nebula> GO FROM 101 OVER follow YIELD follow._src AS player, follow.degree AS degree | GROUP BY $-.player YIELD SUM($-.degree);
-- 从节点 101 出发找到其关注的球员，返回这些球员作为 player，边（follow）的属性值作为 degree，对这些球员分组并返回分组球员属性 degree 相加的值。
-- 返回以下结果：
==================
| SUM($-.degree) |
==================
| 186            |
------------------
```

最后更新: 2020年5月12日

2021 Vesoft Inc.

2.3.5 LIMIT 语法

`LIMIT` 用法与 `SQL` 中的相同，且只能与 `|` 结合使用。 `LIMIT` 子句接受一个或两个参数,两个参数的值都必须是零或正整数。

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

- **expressions**

  待排序的列或计算。

- **number_rows**

  number_rows 指定返回结果行数。例如，LIMIT 10 返回前 10 行结果。由于排序顺序会影响返回结果，所以使用 `ORDER BY` 时请注意排序顺序。

- **offset_value**

  可选选项，用来跳过指定行数返回结果，offset 从 0 开始。

当使用 `LIMIT` 时，请使用 `ORDER BY` 子句对返回结果进行唯一排序。否则，将返回难以预测的子集。

例如：

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

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.3.6 逻辑运算符

名称	描述
&&	逻辑与 AND
!	逻辑非 NOT
||	逻辑或 OR
XOR	逻辑异或 XOR

在 nGQL 中，非 0 数字将被视为 *true* 。逻辑运算符的优先级参见 Operator Precedence。

- &&

逻辑与 AND:

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

- !

逻辑非 NOT:

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

- ||

逻辑或 OR:

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

- XOR

逻辑异或 XOR:

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

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.3.7 运算符优先级

下面的列表展示了 nGQL 运算符的优先级（降序）。同一行的运算符拥有一致的优先级。

```
-（负数）
!
*, /, %
-, +
== , >=, >, <=, <, <>, !=
&&
||
=（赋值）
```

在一个表达式中，同等优先级的运算符将按照从左到右的顺序执行，唯一例外是赋值按照从右往左的顺序执行。但是，可以使用括号来修改执行顺序。

示例:

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

最后更新: 2020年5月12日

2021 Vesoft Inc.

2.3.8 Order By 函数

类似于 SQL， `ORDER BY` 可以进行升序（`ASC`）或降序（`DESC`）排序并返回结果，并且它只能在 `PIPE` 语句（`|`）中使用。

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

如果没有指明 ASC 或 DESC， `ORDER BY` 将默认进行升序排序。

示例

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

-- 取 4 个顶点并将他们以 age 从小到大的顺序排列，如 age 相同，则 name 按降序排列。
-- 返回如下结果:
=====================================
| VertexID | age | name              |
=====================================
| 103      | 32  | Rudy Gay          |
-------------------------------------
| 102      | 33  | LaMarcus Aldridge |
-------------------------------------
| 101      | 36  | Tony Parker       |
-------------------------------------
| 100      | 42  | Tim Duncan        |
-------------------------------------
```

(使用方法参见 FETCH 文档)

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

-- 从顶点 100 出发查找其关注的球员，返回球员的 age 和 name，age 按降序排列，如 age 相同，则 name 按升序排列。
-- 返回如下结果：
==========================
| age | name              |
==========================
| 36  | Tony Parker       |
--------------------------
| 33  | LaMarcus Aldridge |
--------------------------
| 25  | Kyle Anderson     |
--------------------------
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.3.9 集合操作（UNION，INTERSECT， MINUS）

您可以使用集合运算符 `UNION`、 `UNION ALL`、 `INTERSECT` 和 `MINUS` 进行多个组合查询。所有集合运算符具有相同的优先级。如果 nGQL 语句包含多个集合运算符，则 **Nebula Graph** 将对其进行从左到右处理，除非使用括号明确指定了顺序。

复合查询中， `GO` 语句返回的结果必须列数相同，且数据类型相同（比如均为数字类型或字符类型）。

UNION，UNION DISTINCT，UNION ALL

`UNION DISTINCT` （简称 `UNION` ）返回数据集 A 和 B 的并集（不包含重复元素）。

`UNION ALL` 返回数据集 A 和 B 的并集（包含重复元素）。 `UNION` 语法为

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

`<left>` 和 `<right>` 必须列数相同，且数据类型相同。如果数据类型不同，将按照类型转换进行转换。

示例

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

以上语句返回点 `1` 和 `2` （沿边 `e1` ）关联的唯一的点。

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

以上语句返回点 `1` 和 `2` 关联的所有点，其中存在重复点。

`UNION` 亦可与 `YIELD` 同时使用，例如以下语句：

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

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

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

以上语句返回

```
======================
| id  | col_1| col_2  |    -- UNION or UNION DISTINCT. The column names come from query 1
======================
| 104 |  1   |    2   |    -- line 1
----------------------
| 104 |  2   |    2   |    -- line 2
----------------------
| 215 |  4   |    3   |    -- line 3
----------------------
```

请注意第一行与第二行返回相同 id 的点，但是返回的值不同。 `DISTINCT` 检查返回结果中的重复值。所以第一行与第二行的返回结果不同。 `UNION ALL` 返回结果为

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


2021 Vesoft Inc.

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

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

INTERSECT

`INTERSECT` 返回集合 A 和 B 的交集（A ⋂ B）。

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

与 `UNION` 类似， `<left>` 和 `<right>` 必须列数相同，且数据类型相同。 此外，只返回 `<left>` 右 `<right>` 相同的行。例如：

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

返回

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

MINUS

返回 A - B 数据的差集，此处请注意运算顺序。例如：

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

返回

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

如果更改 `MINUS` 顺序

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

则返回

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

集合操作和管道的优先级

请注意当一条查询同时包含管道 `|` 和集合操作时，管道的优先级高于集合操作。管道用法请参考管道文档。语句 `GO FROM 1 UNION GO FROM 2 | GO FROM 3` 等价于语句 `GO FROM 1 UNION (GO FROM 2 | GO FROM 3)`。

例如：

2021 Vesoft Inc.

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

红色框内的语句先执行，然后再执行绿色框内的语句。

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

以上语句中，括号改变了执行的优先级，括号内的语句先执行。

最后更新: 2020年6月10日

2021 Vesoft Inc.

2.3.10 字符比较函数和运算符

名称	描述
CONTAINS	搜索包含指定字段的字符串，大小写敏感

- CONTAINS

`CONTAINS` 运算符用来搜索包含指定字段的字符串，且大小写敏感。 `CONTAINS` 运算符左右两侧必须为字符串类型。

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

nebula> GO FROM 107 OVER serve WHERE $$.team.name CONTAINS "Riors" \
        YIELD $^.player.name, serve.start_year, serve.end_year, $$.team.name; -- 以下语句返回为空。
```

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

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

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.3.11 UUID

`UUID` 用于生成全局唯一的标识符。

当顶点数量到达十亿级别时，用 `hash` 函数生成 vid 有一定的冲突概率。因此 **Nebula Graph** 提供 `UUID` 函数来避免大量顶点时的 vid 冲突。`UUID` 函数由 `Murmurhash` 与当前时间戳（单位为秒）组合而成。

`UUID` 产生的值会以 key-value 方式存储在 **Nebula Graph** 的 Storage 服务中，调用时会检查这个 key 是否存在或冲突。因此相比 hash，性能可能会更慢。

插入 `UUID`：

```
-- 使用 UUID 函数插入一个点。
nebula> INSERT VERTEX player (name, age) VALUES uuid("n0"):("n0", 21);
-- 使用 UUID 函数插入一条边。
nebula> INSERT EDGE follow(degree) VALUES uuid("n0") -> uuid("n1"): (90);
```

获取 `UUID`：

```
nebula> FETCH PROP ON player uuid("n0") YIELD player.name, player.age;
-- 返回以下值：
===============================================
| VertexID             | player.name | player.age |
===============================================
| -5057115778034027261 | n0          | 21         |
-----------------------------------------------
nebula> FETCH PROP ON follow uuid("n0") -> uuid("n1");
-- 返回以下值：
==============================================================================
| follow._src          | follow._dst         | follow._rank | follow.degree |
==============================================================================
| -5057115778034027261 | 4039977434270020867 | 0            | 90            |
------------------------------------------------------------------------------
```

结合 `Go` 使用 `UUID`：

```
nebula> GO FROM uuid("n0") OVER follow;
-- 返回以下值：
=======================
| follow._dst         |
=======================
| 4039977434270020867 |
-----------------------
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.4 语言结构

2.4.1 字面值常量

布尔字面值

布尔字面值 `TRUE` 和 `FALSE` 对大小写不敏感。

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

最后更新: 2020年4月29日

2021 Vesoft Inc.

数值字面值

数值字面值包括整数字面值和浮点字面值。

整数

整数为 64 位，并且可以用 `+` 和 `-` 来表明正负性。它们和 C 语言中的 `int64_t` 是一致的。

注意整数的最大值为 `9223372036854775807` 。输入任何大于此最大值的整数为语法错误。整数的最小值 `-9223372036854775808` 同理。

双浮点数

双浮点数和 C 语言中的 `double` 是一致的。

双浮点数上限和下限分为别 `-1.79769e+308` 和 `1.79769e+308` 。

科学计数法

Nebula Graph 中，支持用科学计数法表示 Double 类型。例如：

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

最后更新: 2020年5月14日

2021 Vesoft Inc.

字符串字面值

字符串由一串字节或字符组成，并由一对单引号 (') 或双引号 (") 包装。

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

一些转义字符 (\) 已被支持，如下表所示:

转义字符	对应的字符
\'	单引号 (')
\"	双引号 (")
\t	制表符
\n	换行符
\b	退格符
\	反斜杠 (\)

示例:

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

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

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.4.2 注释

Nebula Graph 支持四种注释方式：

- 在行末加 **#**
- 在行末加 --，使用 '--' 作注释时，需在其后加空格，即 '-- '。
- 在行末加 //，与 C 语言类似
- 添加 `/* */` 符号，其开始和结束序列无需在同一行，因此此类注释方式支持换行。

尚不支持嵌套注释。 注释方式示例如下：

```
nebula> -- Do nothing in this line
nebula> YIELD 1+1;     # 注释在本行末结束
nebula> YIELD 1+1;    -- 注释在本行末结束
nebula> YIELD 1+1;     // 注释在本行末结束
nebula> YIELD 1  /* 这是行内注释 */ + 1;
nebula> YIELD 11 +  \
/* 多行注释使用      \
隔开  \
*/ 12;
```

行内 `\` 表示换行符。

最后更新: 2020年5月13日

2021 Vesoft Inc.

2.4.3 标识符大小写

用户自定义的标识符为大小写敏感

下方示例语句有错，因为 `my_space` 和 `MY_SPACE` 为两个不同的变量名。

```
nebula> CREATE SPACE my_space;
nebula> USE MY_SPACE;   ---- my_space 和 MY_SPACE 是两个不同的 space
```

关键词和保留关键词为大小写不敏感

下面四条语句是等价的（因为 show 和 spaces 都是保留字）

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

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.4.4 关键字和保留字

关键字是在 nGQL 中具有重要意义的单词。保留关键字需引用方可使用。

非保留关键字无需引用可直接使用，且所有非保留字都会自动转换成小写，所以非保留字不区分大小写。保留关键字需使用反引号标注方可使用，例如 `AND`。

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

nebula> CREATE TAG SPACE(name string); -- SPACE 为非保留关键字
Execution succeeded

nebula> SHOW TAGS; -- 所有非保留字都会自动转换成小写
=============
| ID | Name |
=============
| 25 | space|
-------------
```

`TAG` 为保留字使用时必须使用反引号。 `SPACE` 为非保留字使用时无需加反引号。

```
nebula> CREATE TAG `TAG` (name string); -- 此处 TAG 为保留字
Execution succeeded
```

保留字

以下列表为 nGQL 中的保留字。

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


2021 Vesoft Inc.

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

非保留关键字

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

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.4.5 管道

nGQL 和 SQL 的主要区别之一是子查询的组合方式。

SQL 中的查询语句通常由子查询嵌套组成，而 nGQL 则使用类似于 shell 的管道方式 `PIPE(|)` 来组合子查询。

示例

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

如未使用 `YIELD`，则默认返回终点 `id`。

如果使用 `YIELD` 明确声明返回结果，则不会返回默认值 `id`。

`$-.` 后的别名必须为前一个子句 `YIELD` 定义的值，如本例中的 `dstid` 和 `Name`。

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.4.6 属性引用

`WHERE` 和 `YIELD` 可引用节点或边的属性。

引用点的属性

引用起点的属性

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

其中符号 `$^` 用于获取起点属性，`tag_name` 表示起点的 `tag`，`prop_name` 为指定属性的名称。

引用终点的属性

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

其中符号 `$$` 用于获取终点属性，`tag_name` 表示终点的 `tag`，`prop_name` 为指定属性的名称。

示例

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

该语句用于获取起点的属性名称和终点的属性年龄。

引用边

引用边的属性

使用如下方式获取边属性：

```
edge_type.edge_prop
```

此处，`edge_type` 为边的类型，`edge_prop` 为属性，例如：

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

引用边的内置属性

一条边有四个内置属性：

- _src: 边起点 ID
- _dst: 边终点 ID
- _type: 边类型
- _rank: 边的 rank 值

获取起点和终点 ID 可通过 `_src` 和 `_dst` 获取，这在显示图路径时经常会用到。

例如：

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

该语句通过引用 `follow._src` 作为起点 ID 和 `follow._dst` 作为终点 ID，返回起点 100 `follow` 的所有邻居节点。其中 `follow._src` 返回起点 ID，`follow._dst` 返回终点 ID。

2021 Vesoft Inc.

最后更新: 2020年6月23日

2021 Vesoft Inc.

2.4.7 标识符命名规则

Nebula Graph 中的标识符包括图空间、标签、边、别名、自定义变量等。标识符命名规则为：

- 标识符中允许的字符： ASCII: [*0-9,a-z,A-Z,*] (基本拉丁字母，数字 0 - 9，下划线)，不支持其他标点字符。
- 所有标识符必须以字母开头。
- 标识符区分大小写。
- 不可使用关键字或保留关键字做标识符。

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.4.8 语句组合

组合语句（或子查询）的方法有两种：

- 将多个语句放在一个语句中进行批处理，以英文分号（;）隔开，最后一个语句的结果将作为批处理的结果返回。
- 将多个语句通过运算符（|）连接在一起，类似于 shell 脚本中的管道。前一个语句得到的结果可以重定向到下一个语句作为输入。

注意复合语句并不能保证 `事务性`。 例如，由三个子查询组成的语句：A | B | C，其中 A 是读操作，B 是计算，C 是写操作。 如果任何部分在执行中失败，整个结果将未被定义--尚不支持调用回滚。

示例

分号复合语句

```
nebula> SHOW TAGS; SHOW EDGES;          -- 仅列出边
nebula> INSERT VERTEX player(name, age) VALUES 100:("Tim Duncan", 42); \
  INSERT VERTEX player(name, age) VALUES 101:("Tony Parker", 36); \
  INSERT VERTEX player(name, age) VALUES 102:("LaMarcus Aldridge", 33);  /* 通过复合语句插入多个点*/
```

管道复合语句

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

占位符 `$-.id` 获取第一个语句 `GO FROM 201 OVER edge_serve YIELD edge_serve._dst AS id` 返回的结果。

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.4.9 用户自定义变量

Nebula Graph 支持用户自定义变量，其格式为 `$var_name`。其中变量名 `var_name` 由字母字符组成。目前不建议使用任何其他字符。用户自定义变量可暂时将值存储在前一个语句中的自定义变量中，并在（随后的）另一个语句中使用，从而可将中间结果从一个语句传递到另一个语句。

用户定义变量只能在一次执行中使用（由分号 `;` 或管道 `|` 隔开的复合语句，并提交给服务器一起执行）。

请注意，自定义的变量仅在当前会话的当前查询有效。在一个语句中定义的自定义变量不能在其他客户端和其他执行中使用，也就是说定义语句和使用它的语句应该一起提交。当客户端发送执行时，变量会自动释放。

用户变量名称区分大小写。

示例：

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

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.5 语句语法

2.5.1 数据定义语句 (DDL)

修改 Edge

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

<alter_definition>::=
    ADD (prop_name data_type)
    | DROP (prop_name)
    | CHANGE (prop_name data_type)

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

`ALTER EDGE` 语句可改变边的结构，例如，可以添加或删除属性，更改已有属性的类型，也可将属性设置为 TTL（生存时间），或更改 TTL 时间。

注意： 修改边结构时，**Nebula Graph** 将自动检测是否存在索引。修改时需要两步判断。首先，判断这个边是否关联索引。其次，检查所有关联的索引，判断待删除或更改的 column item 是否存在于索引的 column 中，如果存在则拒绝修改。如果不存在，即使有关联的索引也允许修改。

请参考索引文档了解索引详情。

一个 `ALTER` 语句允许使用多个 `ADD`， `DROP`， `CHANGE` 语句，语句之间需用逗号隔开。但是不要在一个语句中添加，删除或更改相同的属性。如果必须进行此操作，请将其作为 `ALTER` 语句的子语句。

```
nebula> CREATE EDGE e1 (prop3 int, prop4 int, prop5 int);
nebula> ALTER EDGE e1 ADD (prop1 int, prop2 string),    /* 添加 prop1 */
              CHANGE (prop3 string),           /* 将 prop3 类型更改为字符 */
              DROP (prop4, prop5);              /* 删除 prop4 和 prop5 */

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

注意： `TTL_COL` 仅支持 `INT` 和 `TIMESTAMP` 类型的属性。

最后更新: 2020年7月28日

2021 Vesoft Inc.

修改 Tag

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

<alter_definition>::=
    ADD (prop_name data_type)
    | DROP (prop_name)
    | CHANGE (prop_name data_type)

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

`ALTER TAG` 语句可改变标签的结构，例如，可以添加或删除属性，更改已有属性的类型，也可将属性设置为 TTL（生存时间），或更改 TTL 时间。

注意： 修改标签结构时，**Nebula Graph** 将自动检测是否存在索引。修改时需要两步判断。首先，判断这个标签是否关联索引。其次，检查所有关联的索引，判断待删除或更改的 column item 是否存在于索引的 column 中，如果存在则拒绝修改。如果不存在，即使有关联的索引也允许修改。

请参考索引文档了解索引详情。

一个 `ALTER` 语句允许使用多个 `ADD`，`DROP`，`CHANGE` 语句，语句之间需用逗号隔开。但是不要在一个语句中添加，删除或更改相同的属性。如果必须进行此操作，请将其作为 `ALTER` 语句的子语句。

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

注意： `TTL_COL` 仅支持 `INT` 和 `TIMESTAMP` 类型的属性。

最后更新: 2020年7月28日

2021 Vesoft Inc.

CREATE SPACE 语法

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

以上语句用于创建一个新的图空间。不同的图空间是物理隔离的。

IF NOT EXISTS

创建图空间可使用 `IF NOT EXISTS` 关键字，这个关键字会自动检测对应的图空间是否存在，如果不存在则创建新的，如果存在则直接返回。

注意： 这里判断图空间是否存在只是比较图空间的名字(不包括属性)。

SPACE NAME 图空间名

- **space_name**

  图空间的名称在集群中标明了一个唯一的空间。命名规则详见 Schema Object Names

自定义图空间选项

在创建图空间的时候，可以传入如下四个自定义选项：

- *partition_num*

  partition_num 表示数据分片数量。默认值为 100。建议为硬盘数量的 5 倍。

- *replica_factor*

  replica_factor 表示副本数量。默认值是 1，生产集群建议为 3。由于采用多数表决原理，因此需为奇数。

- *charset*

  charset 表示字符集，定义了字符以及字符的编码，默认为 utf8。

- *collate*

  collate 表示字符序，定义了字符的比较规则，默认为 utf8_bin。

如果没有自定义选项，**Nebula Graph** 会使用默认的值（partition_number、replica_factor、charset 和 collate）来创建图空间。

示例

```
nebula> CREATE SPACE my_space_1; -- 使用默认选项创建图空间
nebula> CREATE SPACE my_space_2(partition_num=10); -- 使用默认 replica_factor 创建图空间
nebula> CREATE SPACE my_space_3(replica_factor=1);  -- 使用默认 partition_number 创建图空间
nebula> CREATE SPACE my_space_4(partition_num=10, replica_factor=1);
```

检查 PARTITION 分布正常

在某些大集群上，由于启动时间先后不一，可能会导致 partition 分布不均，可以通过如下命令（SHOW HOSTS）检查机器和分布。

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

若发现机器都已在线，但 partition 分布不均，可以通过如下命令 （BALANCE LEADER）来命令 partition 重分布。

```
nebula> BALANCE LEADER;
```

具体见 SHOW HOSTS 和 BALANCE。

最后更新: 2020年7月22日

2021 Vesoft Inc.

CREATE EDGE 语法

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

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

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

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

Nebula Graph 的图结构由带有属性的 tags 和 edges 组成。`CREATE EDGE` 使用一个给定的名称创建一个新的 edge type。

`CREATE EDGE` 语法有一些特点，在如下分块中将对这些特点进行讨论：

IF NOT EXISTS

创建 edge type 可使用 `IF NOT EXISTS` 关键字，这个关键字会自动检测对应的 edge type 是否存在，如果不存在则创建新的，如果存在则直接返回。

注意： 这里判断 edge type 是否存在只是比较 edge type 的名字(不包括属性)。

EDGE TYPE 名称

- **edge_name**

  edge type 的名称在图中必须**唯一**，且名称被定义后无法被修改。edge type 的命名规则和 space 的命名规则一致。参见 Schema Object Name。

属性名和数据类型

- **prop_name**

  prop_name 表示每个属性的名称。在每个 edge type 中必须唯一。

- **data_type**

  data_type 表示每个属性的数据类型。更多关于 **Nebula Graph** 支持的数据类型信息请参见 data-type。

  > NULL 和 NOT NULL 在创建 edge type 时不可用。(相比于关系型数据库)。

- **默认值约束**

  您可以在创建 edge type 时使用 `DEFAULT` 约束设置属性的默认值。如果没有指定其他值，那么会将默认值插入新的边。默认值可以为 **Nebula Graph** 支持的任一数据类型，且支持表达式。如果您不想使用默认值，也可以写一个用户指定的值。

  > 暂时不支持使用 `Alter` 更改默认值。

Time-to-Live (TTL) 语法

- TTL_DURATION

  TTL_DURATION 指定了 vertices 和 edges 的有效期，超过有效期的数据会失效。失效时间为 TTL_COL 设置的属性值加 TTL_DURATION 设置的秒数。

  > 如果 TTL_DURATION 的值为负或 0，则该 edge 不会失效。

- TTL_COL

  指定的列（或者属性）必须是 int64 或者 timestamp。

- 单 TTL 定义

  仅支持指定单个 TTL_COL 字段。

TTL 详细用法参见 TTL 文档。

示例

2021 Vesoft Inc.

```
nebula> CREATE EDGE follow(start_time timestamp, grade double);
nebula> CREATE EDGE noedge();  -- 属性为空

nebula> CREATE TAG player_with_default(name string, age int DEFAULT 20);  -- 默认年龄设置为 20 岁
nebula> CREATE EDGE follow_with_default(start_time timestamp DEFAULT 0, grade double DEFAULT 0.0);  -- 默认 start_time 设置为 0，默认 grade 设置为 0.0
```

```
nebula> CREATE EDGE marriage(location string, since timestamp)
    TTL_DURATION = 0, TTL_COL = "since"; -- 负值或 0 数据不会失效
```

最后更新: 2020年6月10日

2021 Vesoft Inc.

CREATE TAG 语法

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

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

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

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

Nebula Graph 的图结构由带有属性的 tags 和 edges 组成。`CREATE TAG` 使用一个给定的名称创建一个新的 tag。

`CREATE TAG` 语法有一些特点，在如下分块中将对这些特点进行讨论：

IF NOT EXISTS

创建 tag 可使用 `IF NOT EXISTS` 关键字，这个关键字会自动检测对应的 tag 是否存在，如果不存在则创建新的，如果存在则直接返回。

注意： 这里判断 tag 是否存在只是比较 tag 的名字(不包括属性)。

TAG 名称

- **tag_name**

  tags 的名称在图中必须**唯一**，且名称被定义后无法被修改。Tag 的命名规则和 space 的命名规则一致。参见 Schema Object Name。

属性名和数据类型

- **prop_name**

  prop_name 表示每个属性的名称。在每个 tag 中必须唯一。

- **data_type**

  data_type 表示每个属性的数据类型。更多关于 **Nebula Graph** 支持的数据类型信息请参见 data-type。

  NULL 和 NOT NULL 在创建 tag 时不可用。(相比于关系型数据库)。

- **默认值约束**

  您可以在创建标签/边时使用 `DEFAULT` 约束设置属性的默认值。如果没有指定其他值，那么会将默认值插入新的顶点。默认值可以为 **Nebula Graph** 支持的任一数据类型，且支持表达式。如果您不想使用默认值，也可以写一个用户指定的值。

  暂时不支持使用 `Alter` 更改默认值。

Time-to-Live (TTL) 语法

- TTL_DURATION

  TTL_DURATION 指定了 vertices 和 edges 的有效期，超过有效期的数据会失效。失效时间为 TTL_COL 设置的属性值加 TTL_DURATION 设置的秒数。

  如果 TTL_DURATION 的值为负或 0，则该 edge 不会失效。

- TTL_COL

  指定的列（或者属性）必须是 int64 或者 timestamp。

- 单 TTL 定义

  仅支持指定单个 TTL_COL 字段。

TTL 详细用法参见 TTL 文档。

示例

```
nebula> CREATE TAG course(name string, credits int);
nebula> CREATE TAG notag();  -- 属性为空
```


2021 Vesoft Inc.

```
nebula> CREATE TAG player_with_default(name string, age int DEFAULT 20);  -- 默认年龄设置为 20 岁
```

```
nebula> CREATE TAG woman(name string, age int,
   married bool, salary double, create_time timestamp)
   TTL_DURATION = 100, TTL_COL = "create_time"; -- 时间间隔是 100s，从 create_time 字段的值开始

nebula> CREATE TAG icecream(made timestamp, temperature int)
   TTL_DURATION = 100, TTL_COL = "made";
   --  超过 TTL_DURATION 数据即失效
```

最后更新: 2020年6月10日

2021 Vesoft Inc.

DROP EDGE 语法

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

仅支持有 DROP 权限的用户进行此操作。

注意： 删除边时 **Nebula Graph** 将判断相应边是否有关联的索引，如果有则拒绝删除。

请参考索引文档了解索引详情。

删除边可使用 `IF EXISTS` 关键词，这个关键字会自动检测对应的边是否存在，如果存在则删除，如果不存在则直接返回。

此操作将移除指定类型的所有边。

此操作仅删除 Schema 信息，硬盘中所有文件及目录均**未被直接删除**，数据会在下次 compaction 时删除。

最后更新: 2020年4月29日

2021 Vesoft Inc.

DROP TAG 语法

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

仅支持有 DROP 权限的用户进行此操作。

> 请谨慎进行此操作。

注意： 删除标签时 **Nebula Graph** 将判断相应标签是否有关联的索引，如果有则拒绝删除。

请参考索引文档了解索引详情。

删除标签可使用 `IF EXISTS` 关键字，这个关键字会自动检测对应的标签是否存在，如果存在则删除，如果不存在则直接返回。

一个节点可以有一个或多个标签（类型）。

删除所有标签后，节点将不可访问，同时与节点连接的边也不可使用。

删除单个标签后，节点仍可访问，但是已删除标签的属性不可访问。

此操作仅删除 Schema 信息，硬盘中所有文件及目录均**未被直接删除**，数据会在下次 compaction 时删除。

最后更新: 2020年4月29日

2021 Vesoft Inc.

DROP SPACE 语法

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

仅支持有 DROP 权限的用户进行此操作。

DROP SPACE 将删除指定 space 内的所有内容。

删除图空间可使用 `IF EXISTS` 关键字，这个关键字会自动检测对应的图空间是否存在，如果存在则删除，如果不存在则直接返回。

其他 space 不受影响。

该语句不会立即删除存储引擎中的所有文件和目录（并释放磁盘空间）。删除操作取决于不同存储引擎的实现。

> 请**谨慎**进行此操作。

最后更新: 2020年5月13日

2021 Vesoft Inc.

Schema 索引

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

schema 索引可用于快速处理图查询。**Nebula Graph** 支持两种类型的索引：**Tag 索引**和 **Edge Type 索引**。

多数图查询都从拥有共同属性的同一类型的点或边开始遍历。schema 索引使得这些全局检索操作在大型图上更为高效。

一般地，在使用 `CREATE TAG/EDGE` 语句将 Tag/Edge-type 创建好之后，即可为其创建索引。

创建索引

`CREATE INDEX` 用于为已有 Tag/Edge-type 创建索引。

注意：索引会影响写性能，第一次批量导入的时候，建议先导入数据，再批量重建索引；不推荐带索引批量导入，这样写性能会非常差。

创建单属性索引

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

上述语句在所有标签为 *player* 的顶点上为属性 *name* 创建了一个索引。

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

上述语句在 *follow* 边类型的所有边上为属性 *degree* 创建了一个索引。

创建组合索引

schema 索引还支持为相同 tag 或 edge 中的多个属性同时创建索引。这种包含多种属性的索引在 **Nebula Graph** 中称为组合索引。

注意： 目前尚不支持跨多个 tag 创建复合索引。

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

上述语句在所有标签为 *player* 的顶点上为属性 *name* 和 *age* 创建了一个复合索引。

列出索引

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

`SHOW INDEXES` 用于列出已创建完成的 Tag/Edge-type 的索引信息。使用以下命令列出索引：

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

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

返回索引信息

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

`DESCRIBE INDEX` 用于返回指定索引信息。例如，使用以下命令返回索引信息：

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

nebula> DESCRIBE TAG INDEX player_index_1;
```


2021 Vesoft Inc.

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

删除索引

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

`DROP INDEX` 用于删除指定名称的 Tag/Edge-type 索引。例如，使用以下命令删除名为 *player_index_0* 的索引：

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

重构索引

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

创建索引部分介绍了如何创建索引以提高查询性能。如果索引在插入数据之前创建，此时无需执行索引重构操作；如果创建索引时，数据库里已经存有数据，则不会自动对旧的数据进行索引，此时需要对整个图中与索引相关的数据执行索引重构操作以保证索引包含了之前的数据。若当前数据库没有对外提供服务，则可在索引重构时使用 `OFFLINE` 关键字加快重构速度。

重构完成后，可使用 `SHOW {TAG | EDGE} INDEX STATUS` 命令查看索引是否重构成功。例如：

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

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

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

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

使用索引

索引创建完成并插入相关数据后，即可使用 LOOKUP 语句进行数据查询。

通常无需指定在查询中具体使用的索引，**Nebula Graph** 会自行选择。

最后更新: 2020年7月21日

2021 Vesoft Inc.

TTL (time-to-live)

Nebula Graph 支持 **TTL**，在一定时间后自动从数据库中删除点或者边。过期数据会在下次 compaction 时被删除，在下次 compaction 前，query 会过滤掉过期的点和边。

ttl 功能需要 `ttl_col` 和 `ttl_duration` 一起使用。自从 `ttl_col` 指定的字段的值起，经过 `ttl_duration` 指定的秒数后，该条数据过期。即，到期阈值是 `ttl_col` 指定的 property 的值加上 `ttl_duration` 设置的秒数。其中 `ttl_col` 指定的字段的类型需为 integer 或者 timestamp。

TTL 配置

- `ttl_duration` 单位为秒，范围为 0 ~ max(int64)，当 `ttl_duration` 被设置为 0，则点的此 tag 属性不会过期。
- 当 `ttl_col` 指定的字段的值 + `ttl_duration` 值 < 当前时间时，该条数据此 tag 属性值过期。
- 当该条数据有多个 tag，每个 tag 的 ttl 单独处理。

设置 TTL

- 对已经创建的 tag，设置 TTL。

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

点 101 的 TAG t1 属性会在 now() 之后，经过 5s 后过期。

- 在创建 tag 时设置 TTL。

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

点 102 的 TAG t2 属性会在 2020年3月17日 18时33分51秒 CST （即时间戳为 1584441231），经过 100s 后过期。

- 当点有多个 TAG 时，各 TAG 的 TTL 相互独立。

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

5s 后, 点 Vertex 200 的 t1 属性过期。

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

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

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

删除 TTL

如果想要删除 TTL，可以设置 `ttl_col` 字段为空，或删除配置的 `ttl_col` 字段，或者设置 `ttl_duration` 为 0。

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

删除配置的 `ttl_col` 字段：

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

设置 ttl_duration 为 0：

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


2021 Vesoft Inc.

TTL 使用注意事项

- 如果某个属性被 `ttl_col` 引用，则不支持对其进行更改操作。

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

- 注意一个 tag 或 edge 不能同时拥有 TTL 和索引，只能二者择其一，即使 `ttl_col` 配置的字段与要创建索引的字段不同。

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

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

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

- 对 edge 配置 TTL 与 tag 类似。

最后更新: 2020年7月22日

2021 Vesoft Inc.

2.5.2 数据查询与操作语句 (DQL 和 DML)

Delete Edge 语法

`DELETE EDGE` 语句用于删除边。给定一个 edge 类型，及其起点与终点，**Nebula Graph** 支持删除这条边及其相关属性和 rank，也支持指定 rank 删除边，语法如下：

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

例如，

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

以上示例删除一条起点为 `100`， 终点为 `200`， 边类型为 `follow` 的边。

系统内部会找出与这条边相关联的属性，并将其全部删除。整个过程当前还无法保证原子性，因此当遇到失败时请重试。

最后更新: 2020年6月23日

2021 Vesoft Inc.

Delete Vertex 语法

Nebula Graph 支持给定点 ID（或 hash ID、UUID），删除这些顶点和与其相关联的入边和出边，语法如下：

```
DELETE VERTEX <vid_list>
```

例如，

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

以上示例删除 ID 为 `121` 的点。

系统内部会找出与这些顶点相关联的出边和入边，并将其全部删除，然后再删除点相关的信息。整个过程当前还无法保证原子性，因此若操作失败请重试。

最后更新: 2020年5月25日

2021 Vesoft Inc.

Fetch 语法

`FETCH` 语句用于获取点和边的属性。

获取点属性

`FETCH PROP ON` 可返回节点的一系列属性，目前已支持一条语句返回多个节点属性。 `FETCH` 获取点属性支持与管道 及用户自定义变量一起使用。

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

`*` 返回指定 `VID` 点的所有属性。

`<tag_name_list>::=[tag_name [, tag_name]]` 为标签名称，与 return_list 中的标签相同。

`<vertex_id_list>::=[vertex_id [, vertex_id]]` 是一组用 "," 分隔开的顶点 `VID` 列表。

`[YIELD [DISTINCT] <return_list>]` 为返回的属性列表， `YIELD` 语法参看 YIELD Syntax 。

示例

```
-- 返回节点 100 的所有属性。
nebula> FETCH PROP ON * 100;

-- 返回点100，201 的 player、team tag 上的所有属性。
nebula> FETCH PROP ON * 100, 102;

-- 返回节点 100、201 的所有属性。
nebula> FETCH PROP ON player, team 100, 201;

-- 如未指定 YIELD 字段，则返回节点 100，tag 为 player 的所有属性。
nebula> FETCH PROP ON player 100;

-- 返回节点 100 的姓名与年龄属性。
nebula> FETCH PROP ON player 100 YIELD player.name, player.age;

-- 通过 hash 生成 int64 节点 ID，返回其姓名和年龄属性。
nebula> FETCH PROP ON player hash("nebula")  YIELD player.name, player.age;

-- 支持与管道一起使用
nebula> YIELD 100 AS id | FETCH PROP ON player $-.id;

-- 沿边 follow 寻找节点 100 的所有近邻，返回其姓名和年龄属性。
nebula> GO FROM 100 OVER follow YIELD follow._dst AS id | FETCH PROP ON player $-.id YIELD player.name, player.age;

-- 与上述语法相同。
nebula> $var = GO FROM 100 OVER follow YIELD follow._dst AS id; FETCH PROP ON player $var.id YIELD player.name, player.age;

-- 获取 100、101、102 三个节点，返回姓名和年龄都不相同的记录。
nebula> FETCH PROP ON player 100,101,102 YIELD DISTINCT player.name, player.age;
```

获取边属性

使用 `FETCH` 获取边属性的用法与点属性大致相同，且可同时获取相同类型多条边的属性。

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

`<edge_type>` 指定边的类型，需与 `<return_list>` 相同。

`<vid> -> <vid>` 从起始节点到终止节点，多条边需使用逗号隔开。

`<rank>` 指定相同类型边 rank，可选。如未指定，则默认返回 rank 为 0 的边。

`[YIELD [DISTINCT] <return_list>]` 为返回的属性列表。

获取边属性示例

```
-- 本语句未指定 YIELD，因此获取从节点 100 到节点 200 边 serve 的所有属性。
nebula> FETCH PROP ON serve 100 -> 200;

-- 仅返回属性 start_year。
nebula> FETCH PROP ON serve 100 -> 200 YIELD serve.start_year;

-- 获取节点 100 出边 follow 的 degree 属性。
nebula> GO FROM 100 OVER follow YIELD follow.degree;

-- 同上述语句。
nebula> GO FROM 100 OVER follow YIELD follow._src AS s, serve._dst AS d \
 | FETCH PROP ON follow $-.s -> $-.d YIELD follow.degree;
```


2021 Vesoft Inc.

```
-- 同上述语句。
nebula> $var = GO FROM 100 OVER follow YIELD follow._src AS s, follow._dst AS d;\
 FETCH PROP ON follow $var.s -> $var.d YIELD follow.degree;
```

最后更新: 2020年8月11日

2021 Vesoft Inc.

GO 语法

`GO` 是 **Nebula Graph** 中最常用的关键字，可以指定过滤条件（如 `WHERE` ）遍历图数据并获取点和边的属性，还能以指定顺序（ `ORDER BY ASC | DESC` ）返回指定数目（ `LIMIT` ）的结果。

`GO` 的用法与 SQL 中的 `SELECT` 类似，重要区别是 `GO` 必须从遍历一系列的节点开始。

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

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

<edge_type_list>
   edge_type [, edge_type ...]

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

- `<N> STEPS` 指定查询 N 跳。当 `N` 为零时，**Nebula Graph** 不会获取任何边，因此返回结果为空。
- `M TO N STEPS` 指定查询 M 到 N 跳。当 `M` 为零时，返回结果与 `M` 为 1 时一样，即 `GO 0 TO 2` 与 `GO 1 TO 2` 返回相同结果。
- `<node_list>` 为逗号隔开的节点 ID，或特殊占位符 `$-.id` (参看 `PIPE` 用法)。
- `<edge_type_list>` 为图遍历返回的边类型列表。
- `WHERE <expression>` 指定被筛选的逻辑条件，WHERE 可用于起点，边及终点，同样支持逻辑关键词 AND、OR、NOT，详情参见 WHERE 的用法。
- `YIELD [DISTINCT] <return_list>` 以列的形式返回结果，并可对列进行重命名。详情参看 `YIELD` 用法。 `DISTINCT` 的用法与 SQL 相同。

示例

```
nebula> GO FROM 107 OVER serve;  \
   /* 从点 107 出发，沿边 serve，找到点 200，201 */
==============
| serve._dst |
==============
| 200        |
--------------
| 201        |
--------------
```

```
nebula> GO 2 STEPS FROM 103 OVER follow; \
  /* 返回点 103 的 2 度的好友 */
===============
| follow._dst |
===============
| 101         |
---------------
```

```
nebula> GO FROM 109 OVER serve  \
   WHERE serve.start_year > 1990       /* 筛选边 serve 的 start_year 属性  */ \
   YIELD $$.team.name AS team_name, serve.start_year as start_year;    /* 目标点 team 的 serve.start_year 属性 serve.start_year */
==========================
| team_name | start_year |
==========================
| Nuggets   | 2011       |
--------------------------
| Rockets   | 2017       |
--------------------------
```

```
nebula> GO FROM 100,102 OVER serve           \
        WHERE serve.start_year > 1995            /* 筛选边属性*/ \
        YIELD DISTINCT $$.team.name AS team_name, /* DISTINCT 与 SQL 用法相同 */ \
        serve.start_year as start_year,          /* 边属性 */ \
        $^.player.name AS player_name;           /* 起点 (player) 属性 */
===========================================
| team_name | start_year | player_name       |
===========================================
| Warriors  | 2001       | LaMarcus Aldridge |
-------------------------------------------
| Warriors  | 1997       | Tim Duncan        |
-------------------------------------------
```


2021 Vesoft Inc.

沿着多种类型的边进行遍历

目前 **Nebula Graph** 还支持 `GO` 沿着多条边遍历，语法为：

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

例如：

```
nebula> GO FROM <node_list> OVER edge1, edge2....  //沿着 edge1 和 edge2 遍历，或者
nebula> GO FROM <node_list> OVER *    //这里 * 意味着沿着任意类型的边遍历
```

> 请注意，当沿着多种类型边遍历时，对于使用过滤条件有特别限制(也即 `WHERE` 语句），比如 `WHERE edge1.prop1 > edge2.prop2` 这种过滤条件是不支持的。

对于返回的结果，如果存在多条边的属性需要返回，会把他们放在不同的行。比如：

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

返回如下结果：

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

没有的属性当前会填充默认值， 数值型的默认值为 0， 字符型的默认值为空字符串。bool 类型默认值为 false，timestamp 类型默认值为 0 (即 "1970-01-01 00:00:00")，double 类型默认值为 0.0。

当然也可以不指定 `YIELD`， 这时会返回每条边目标点的 vid。如果目标点不存在，同样用默认值(此处为 0)填充。比如 `GO FROM 100 OVER follow, serve;`，返回结果如下：

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

对于 `GO FROM 100 OVER *` 这样的例子来说，返回结果也和上面例子类似：不存在的属性或者 vid 使用默认值来填充。 请注意从结果中无法分辨每一行属于哪条边，未来版本会在结果中把 edge type 表示出来。

反向遍历

目前 **Nebula Graph** 支持使用关键词 `REVERSELY` 进行反向遍历，语法为：

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

例如：

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

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

============================
| FriendOf      | Team     |
============================
| Tony Parker   | Warriors |
----------------------------
```


2021 Vesoft Inc.

```
| Kyle Anderson | Warriors |
----------------------------
```

遍历所有关注 100 号球员的球员，找出这些球员服役的球队，筛选年龄大于 20 岁的球员并返回这些球员姓名和其服役的球队名称。如果此处不指定 `YIELD`，则默认返回每条边目标点的 `vid`。

双向遍历

目前 **Nebula Graph** 支持使用关键词 `BIDIRECT` 进行双向遍历，语法为：

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

例如：

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

上述语句同时返回 102 关注的球员及关注 102 的球员。

遍历 M 到 N 跳

Nebula Graph 支持遍历 M 到 N 跳。当 M 等于 N 时，`GO M TO N STEPS` 等同 `GO N STEPS`。语法为：

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

例如：

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

遍历从点 100 出发沿 serve 边 1 至 2 跳的点。

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

反向遍历从点 100 出发沿 follow 边 2 至 4 跳的点。

```
nebula> GO 4 TO 5 STEPS FROM 101 OVER follow BIDIRECT YIELD DISTINCT follow._dst;
===============
| follow._dst |
===============
| 100         |
---------------
| 102         |
---------------
| 104         |
---------------
| 105         |
---------------
| 107         |
---------------
| 113         |
---------------
```


2021 Vesoft Inc.

```
| 121        |
--------------
```

双向遍历从点 101 出发沿 follow 边 4 至 5 跳的点。

支持 INT 类型传入查询

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

例如：

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

最后更新: 2020年8月4日

2021 Vesoft Inc.

INSERT EDGE 语法

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

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

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

INSERT EDGE 用于插入从起点（ src_vid ）到终点（ dst_vid ）的一条边。

- `<edge_name>` 表示边类型，在进行 `INSERT EDGE` 操作前需创建好。
- `<prop_name_list>` 为指定边的属性列表。
- `<prop_value_list>` 须根据 列出属性，如无匹配类型，则返回错误。
- `rank` 指定边 rank，可在插入同一类型的多条边时使用，可选，不指定时默认为 0。

示例

```
nebula> CREATE EDGE e1();                   -- 创建空属性边 t1
nebula> INSERT EDGE e1 () VALUES 10->11:();    -- 插入一条从点 10 到点 11 的空属性边
nebula> INSERT EDGE e1 () VALUES 10->11@1:();  -- 插入一条从点 10 到点 11 的空属性边，rank 值为 1
```

```
nebula> CREATE EDGE e2 (name string, age int);                    -- 创建有两种属性的边 e2
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", 1);          -- 插入一条从点 11 到点 13 的有两条属性的边
nebula> INSERT EDGE e2 (name, age) VALUES \
12->13:("n1", 1), 13->14:("n2", 2);                          -- 插入两条边
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", "a13");      -- 错误操作，"a13" 不是 int 类型
```

同一条边可被多次插入或写入，读取时以最后一次插入为准。

```
-- 为插入边赋新值
insert edge with new version of values.
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", 12);
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", 13);
nebula> INSERT EDGE e2 (name, age) VALUES 11->13:("n1", 14); -- 读取最后插入的值
```

最后更新: 2020年6月23日

2021 Vesoft Inc.

INSERT VERTEX 语法

```
INSERT VERTEX <tag_name> [, <tag_name>, ...] (prop_name_list[, 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] ...]
```

INSERT VERTEX 可向 **Nebula Graph** 插入节点。

- `tag_name` 表示标签（节点类型），在进行 `INSERT VERTEX` 操作前需创建好。
- `prop_name_list` 指定标签的属性列表。
- `VID` 表示点 ID。每个图空间中的 `VID` 必须唯一。目前的排序依据为“二进制编码顺序“：即 0, 1, 2, ... 9223372036854775807, -9223372036854775808, -9223372036854775807, ..., -1。`VID` 支持手动指定 ID 或使用 `hash()` 函数生成。
- `prop_value_list` 须根据 `prop_name_list` 列出属性值，如无匹配类型，则返回错误。

示例

```
nebula> CREATE TAG t1();                  -- 创建空属性标签 t1
nebula> INSERT VERTEX t1 () VALUES 10:();    -- 插入空属性点 10
```

```
nebula> CREATE TAG t2 (name string, age int);                -- 创建有两种属性的标签 t2
nebula> INSERT VERTEX t2 (name, age) VALUES 11:("n1", 12);     -- 插入有两种属性的点 11
nebula> INSERT VERTEX t2 (name, age) VALUES 12:("n1", "a13");  -- 错误操作，"a13" 不是 int 类型
nebula> INSERT VERTEX t2 (name, age) VALUES 13:("n3", 12), 14:("n4", 8);    -- 插入两个点
```

```
nebula> CREATE TAG t1(i1 int);
nebula> CREATE TAG t2(s2 string);
nebula> INSERT VERTEX  t1 (i1), t2(s2) VALUES 21: (321, "hello");   -- 插入有两个标签的点 21
```

同一节点可被多次插入或写入，读取时以最后一次插入为准。

```
-- 为点 11 多次插入新值
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);  -- 读取最后插入的值
```

最后更新: 2020年8月11日

2021 Vesoft Inc.

LOOKUP 语法

`LOOKUP` 语句指定过滤条件对数据进行查询。`LOOKUP` 语句之后通常跟着 `WHERE` 子句。`WHERE` 子句用于向条件中添加过滤性的谓词，从而对数据进行过滤。

注意： 在使用 `LOOKUP` 语句之前，请确保已创建索引。查看索引文档了解有关索引的更多信息。

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

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

- `LOOKUP` 语句用于寻找点或边的集合。
- `WHERE` 指定被筛选的逻辑条件。同样支持逻辑关键词 AND、UNION、NOT，详情参见 WHERE 的用法。 **注意：** `WHERE` 子句在 `LOOKUP` 中暂不支持如下操作：
  - `$-` 和 `$^`
  - 在关系表达式中，暂不支持操作符两边都是field-name 的表达式，如 (tagName.column1 > tagName.column2)
  - 暂不支持运算表达式和 function 表达式中嵌套 AliasProp 表达式。
- `YIELD` 指定返回结果。如未指定，则在 `LOOKUP` 标签时返回点 ID，在 `LOOKUP` 边类型时返回边的起点 ID、终点 ID 和 ranking 值。

点查询

如下示例返回名称为 `Tony Parker`，标签为 *player* 的顶点。

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

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

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

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

边查询

如下示例返回 `degree` 为 90，边类型为 *follow* 的边。

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

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

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

nebula> LOOKUP ON follow WHERE follow.degree == 90 YIELD DISTINCT follow._src;
========================================
| SrcVID | DstVID | Ranking | follow._src |
========================================
| 121    | 116    | 0       | 121         |
----------------------------------------
| 140    | 114    | 0       | 140         |
----------------------------------------
```


2021 Vesoft Inc.

```
| 142    | 117    | 0       | 142         |
-------------------------------------------
| 133    | 114    | 0       | 133         |
-------------------------------------------
| 143    | 150    | 0       | 143         |
-------------------------------------------
| 114    | 103    | 0       | 114         |
-------------------------------------------
| 136    | 117    | 0       | 136         |
-------------------------------------------
| 127    | 114    | 0       | 127         |
-------------------------------------------
| 147    | 136    | 0       | 147         |
-------------------------------------------
| 118    | 120    | 0       | 118         |
-------------------------------------------
| 128    | 116    | 0       | 128         |
-------------------------------------------
| 138    | 115    | 0       | 138         |
-------------------------------------------
| 129    | 116    | 0       | 129         |
-------------------------------------------

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

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

FAQ

错误码411

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

错误码 411 表示 WHERE 过滤时没有有效的索引。Nebula Graph的索引使用的是最左匹配原则，即从最左边的为起点任何连续的索引都能匹配上。例如：

```
# 为标签t的前三个属性创建索引。
nebula> CREATE TAG INDEX example_index ON TAG t(p1, p2, p3);

# 无法匹配索引，因为不是从p1开始。
nebula> LOOKUP ON t WHERE p2 == 1 and p3 == 1;

# 可以匹配索引。
nebula> LOOKUP ON t WHERE p1 == 1;

# 可以匹配索引，因为p1和p2是连续的。
nebula> LOOKUP ON t WHERE p1 == 1 and p2 == 1;

# 可以匹配索引，因为p1、p2、p3是连续的。
nebula> LOOKUP ON t WHERE p1 == 1 and p2 == 1 and p3 == 1;
```

最后更新: 2021年5月14日

2021 Vesoft Inc.

Return 语法

Return 语句用于返回条件成立时的结果。如果条件不成立，则无返回结果。

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

- 为变量名称，示例：$var

示例

```
nebula> $A = GO FROM 100 OVER follow YIELD follow._dst AS dst; \
        $rA = YIELD $A.* WHERE $A.dst == 101; \
        RETURN $rA IF $rA is NOT NULL; \ /* $rA 为非空，返回 $rA */
        GO FROM $A.dst OVER follow; /* 因为 RETURN 语句返回了结果，所以GO FROM 语句不执行 */
=========
| $A.dst |
=========
| 101    |
---------
nebula> $A = GO FROM 100 OVER follow YIELD follow._dst AS dst; \
        $rA = YIELD $A.* WHERE $A.dst == 300; \
        RETURN $rA IF $rA is NOT NULL; \ /* $rA 为空，不返回任何值 */
        GO FROM $A.dst OVER follow; /* 因为 RETURN 语句无返回结果，所以 GO FROM 语句将执行 */
==============
| follow._dst |
==============
| 100         |
--------------
| 101         |
--------------
| 100         |
--------------
| 102         |
--------------
| 100         |
--------------
| 107         |
--------------
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

UPDATE EDGE 语法

Nebula Graph 支持 `UPDATE EDGE` 一条边的属性，支持 CAS 操作，支持返回相关的属性。`UPDATE EDGE` 一次只能更新一条边的一个 edge type 上的属性。

```
UPDATE EDGE <edge> SET <update_columns> [WHEN <condition>] [YIELD <columns>]
```

注意： `WHEN` 和 `YIELD` 是可选的。

- `edge` 表示需要更新的 edge，edge 的格式为 `<src> -> <dst> [@rank] OF <edge_type>`。
- `update_columns` 表示需要更新的 edge 上的属性。
- `condition` 是一些约束条件，只有满足这个条件，update 才会真正执行，支持表达式操作。
- `columns` 表示需要返回的 columns，此处 YIELD 可返回 update 以后最新的 columns 值。

举例如下：

```
nebula> UPDATE EDGE 100 -> 200@0 OF serve SET start_year = serve.start_year + 1 \
YIELD $^.player.name AS name, serve.start_year AS start;
```

最后更新: 2020年6月23日

2021 Vesoft Inc.

UPDATE VERTEX 语法

Nebula Graph 支持 `UPDATE VERTEX` 一个点的属性，支持 CAS 操作，支持返回相关的属性。`UPDATE VERTEX` 一次只能更新一个点的一个 tag 上的属性。

```
UPDATE VERTEX <vid> SET <update_columns> [WHEN <condition>] [YIELD <columns>]
```

注意： `WHEN` 和 `YIELD` 是可选的。

- `vid` 表示需要更新的 vertex ID。
- `update_columns` 表示需要更新的 tag 上的 columns，比如 `tag1.col1 = $^.tag2.col2 + 1` 表示把这个点的 `tag1.col1` 更新成 `tag2.col2 + 1`。

  注意： `$^` 表示 `UPDATE` 中需要更新的点。
- `condition` 是一些约束条件，只有满足这个条件，`UPDATE` 才会真正执行，支持表达式操作。
- `columns` 表示需要返回的 columns，此处 `YIELD` 可返回 update 以后最新的 columns 值。

举例如下：

```
nebula> UPDATE VERTEX 101 SET player.age = $^.player.age + 1 \
WHEN $^.player.name == "Tony Parker" \
YIELD $^.player.name AS name, $^.player.age AS age;
```

这个例子里面，101 有一个 tag，即 player。

```
nebula> UPDATE VERTEX 200 SET player.name = 'Cory Joseph' WHEN $^.team.name == 'Rocket';
[ERROR (-8)]: Maybe invalid tag or property in SET/YIELD clause!
```

`UPDATE VERTEX` 不支持多个 tag，故此处报错。

最后更新: 2020年6月10日

2021 Vesoft Inc.

UPSERT 语法

`UPSERT` 用于插入新的顶点或边或更新现有的顶点或边。如果顶点或边不存在，则会新建该顶点或边。`UPSERT` 是 `INSERT` 和 `UPDATE` 的组合。

`UPSERT` 操作相比于 `INSERT` 操作性能会低很多，因为 `UPSERT` 是在 partition 级别的 read-modify-write 串行化操作，因此不适用于大并发更改写入的场景。

- 如果顶点或边不存在，则会新建该顶点或边，无论 WHEN 条件是否满足，且未经 SET 指定的属性字段使用该字段的默认值，如果默认值不存在则报错；
- 如果该顶点或者边存在，并且 WHEN 条件满足，则会更新；
- 如果该顶点或者边存在, 并且 WHEN 条件不满足，则不会有任何操作。

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

- `vid` 表示需要更新的 vertex ID。
- `edge` 表示需要更新的 edge，edge 的格式为 `<src> -> <dst> [@rank] OF <edge_type>`。
- `update_columns` 表示需要更新的 tag 或 edge 上的 columns，比如 `tag1.col1 = $^.tag2.col2 + 1` 表示把这个点的 `tag1.col1` 更新成 `tag2.col2 + 1`。

  注意： `$^` 表示 `UPDATE` 中需要更新的点。

- `condition` 是一些约束条件，只有满足这个条件，`UPDATE` 才会真正执行，支持表达式操作。
- `columns` 表示需要返回的 columns，此处 `YIELD` 可返回 update 以后最新的 columns 值。

例如：

```
nebula> INSERT VERTEX player(name, age) VALUES 111:("Ben Simmons", 22); -- 插入一个新点。
nebula> UPSERT VERTEX 111 SET player.name = "Dwight Howard", player.age = $^.player.age + 11 WHEN $^.player.name == "Ben Simmons" && $^.player.age > 20 YIELD $^.player.name AS Name,
$^.player.age AS Age; -- 对该点进行 UPSERT 操作。
======================
| Name          | Age |
======================
| Dwight Howard | 33  |
----------------------
```

```
nebula> FETCH PROP ON * 111; -- 返回为空，点 111 不存在
Empty set (Time spent: 3.069/4.382 ms)
nebula> UPSERT VERTEX 111 SET player.age = $^.player.age + 1;
```

当点 111 不存在，player 的 age 有默认值时，点 111 的 player.age 为默认值 + 1；此处 player.age 未设置默认值，因此报错。

```
nebula> CREATE TAG person(followers int, age int DEFAULT 0); -- 创建示例 tag person

nebula> UPSERT VERTEX 300 SET person.followers = $^.person.age + 1,  person.age = 8; -- followers 为 1，age 为 8

nebula> UPSERT VERTEX 300 SET person.age = 8, person.followers = $^.person.age + 1; -- followers 为 9，age 为 8
```

最后更新: 2020年6月23日

2021 Vesoft Inc.

WHERE 语法

`WHERE` 子句可以为查询返回的数据指定搜索条件。 `WHERE` 子句语法如下：

```
WHERE <expression> [ AND | OR <expression> ...])
```

目前， `WHERE` 语句适用于 `GO` 和 `LOOKUP` 语句。注意部分 `WHERE` 过滤条件尚未在 `LOOKUP` 语句中支持。详情参考 LOOKUP 文档。

通常，筛选条件是关于节点、边的表达式的逻辑组合。

> 作为语法糖，逻辑与可用 `AND` 或 `&&`，同理，逻辑或可用 `OR` 或 `||` 表示。

示例

```
-- 边 follow 的 degree 属性大于 90。
nebula> GO FROM 100 OVER follow WHERE follow.degree > 90;
-- 返回以下值：
===============
| follow._dst |
===============
| 101         |
---------------

-- 找到与起点 player 104 的 age 值相等的点。
nebula> GO FROM 104 OVER follow WHERE $^.player.age == $$.player.age;
-- 返回以下值：
===============
| follow._dst |
===============
| 103         |
---------------

-- 多种逻辑组合。
nebula> GO FROM 100 OVER follow WHERE follow.degree > 90 OR $$.player.age != 33 AND $$.player.name != "Tony Parker";
-- 返回以下值：
===============
| follow._dst |
===============
| 101         |
---------------
| 106         |
---------------

--下面 WHERE 语句中的条件总是为 TRUE。
nebula> GO FROM 101 OVER follow WHERE 1 == 1 OR TRUE;
-- 返回以下值：
===============
| follow._dst |
===============
| 100         |
---------------
| 102         |
---------------
```

使用 WHERE 对边 RANK 筛选

`WHERE` 子句支持对边 rank 进行筛选。例如：

```
nebula> CREATE SPACE test;
nebula> USE test;
nebula> CREATE EDGE e1(p1 int);
nebula> CREATE TAG person(p1 int);
nebula> INSERT VERTEX person(p1) VALUES 1:(1);
nebula> INSERT VERTEX person(p1) VALUES 2:(2);
nebula> INSERT EDGE e1(p1) VALUES 1->2@0:(10);
nebula> INSERT EDGE e1(p1) VALUES 1->2@1:(11);
nebula> INSERT EDGE e1(p1) VALUES 1->2@2:(12);
nebula> INSERT EDGE e1(p1) VALUES 1->2@3:(13);
nebula> INSERT EDGE e1(p1) VALUES 1->2@4:(14);
nebula> INSERT EDGE e1(p1) VALUES 1->2@5:(15);
nebula> INSERT EDGE e1(p1) VALUES 1->2@6:(16);
nebula> GO FROM 1 OVER e1 WHERE e1._rank>2 YIELD e1._src, e1._dst, e1._rank AS Rank, e1.p1 | ORDER BY Rank DESC;
===================================
| e1._src | e1._dst | Rank | e1.p1 |
===================================
| 1       | 2       | 6    | 16    |
-----------------------------------
| 1       | 2       | 5    | 15    |
-----------------------------------
| 1       | 2       | 4    | 14    |
-----------------------------------
| 1       | 2       | 3    | 13    |
-----------------------------------
```


2021 Vesoft Inc.

最后更新: 2020年7月7日

2021 Vesoft Inc.

YIELD 子句、语句

`YIELD` 关键词可以在 `FETCH`、 `GO` 语句中作为子句使用，也可以在 `PIPE` (`|`) 中作为独立的语句使用，同时可以作为用于计算的单句使用。

作为子句

```
YIELD
    [DISTINCT]
    <col_name> [AS <col_alias>]
    [, <col_name> [AS <col_alias>] ...]
```

常用于返回由 `GO` （详情请参阅 GO 用法）语句生成的结果。

```
nebula> GO FROM 100 OVER follow YIELD $$.player.name AS Friend, $$.player.age AS Age;
==========================
| Friend            | Age |
==========================
| Tony Parker       | 36  |
--------------------------
| LaMarcus Aldridge | 33  |
--------------------------
| Kyle Anderson     | 25  |
--------------------------
```

例如， `$$.player.name` 用来获取目标点（`$$`）的属性。

作为语句

引用输入或者变量

- 可以在 `PIPE` 中使用 `YIELD` 语句。
- 可以用于引用变量。
- 对于那些不支持 `YIELD` 子句的语句，可以使用 `YIELD` 语句作为一个工具，控制输出。

```
YIELD
    [DISTINCT]
    <col_name> [AS <col_alias>]
    [, <col_name> [AS <col_alias>] ...]
    [WHERE <conditions>]
```

```
nebula> GO FROM 100 OVER follow YIELD follow._dst AS id | YIELD $-.* WHERE $-.id == 106;

=========
| $-.id |
=========
| 106   |
---------

nebula> $var1 = GO FROM 101 OVER follow; $var2 = GO FROM 105 OVER follow; YIELD DISTINCT $var1.* UNION YIELD $var2.*;

====================
| $var1.follow._dst |
====================
| 100               |
--------------------
| 102               |
--------------------
| 104               |
--------------------
| 116               |
--------------------
| 125               |
--------------------

nebula> GO 2 STEPS FROM 100 OVER follow YIELD follow._dst AS dst | YIELD DISTINCT $-.dst AS dst

=======
| dst |
=======
| 100 |
-------
| 102 |
-------
| 125 |
-------
```

作为独立的语句

- YIELD 语句可以独立使用，用于一些简单的计算。您可以使用 `AS` 重命名返回的列。

2021 Vesoft Inc.

```
nebula> YIELD 1 + 1;
=========
| (1+1) |
=========
| 2     |
---------

nebula> YIELD "Hel" + "\tlo" AS HELLO_1, ", World!" AS WORLD_2;
======================
| HELLO_1 | WORLD_2  |
======================
| Hel  lo  | , World! |
----------------------

nebula> YIELD hash("Tim") % 100;
====================
| (hash("Tim")%100) |
====================
| 42                |
--------------------
```

最后更新: 2021年4月15日

2021 Vesoft Inc.

2.5.3 辅助功能语句

SHOW 语句

SHOW CHARSET 语法

```
SHOW CHARSET
```

`SHOW CHARSET` 返回所有可用的字符集。目前支持两种类型：utf8、utf8mb4。其中默认字符集为 utf8。**Nebula Graph** 将 utf8 进行了扩展，utf8 同时支持 4 个字节的字符，因此，utf8 和 utf8mb4 是等价的。

```
nebula> SHOW CHARSET;
=======================================================
| Charset | Description   | Default collation | Maxlen |
=======================================================
| utf8    | UTF-8 Unicode | utf8_bin          | 4      |
-------------------------------------------------------
```

`SHOW CHARSET` 输出以下列：

- Charset 字符集名称。
- Description 字符集的描述。
- Default collation 字符集的默认排序规则。
- Maxlen 存储一个字符所需的最大字节数。

最后更新: 2020年4月29日

2021 Vesoft Inc.

SHOW COLLATION 语法

```
SHOW COLLATION
```

`SHOW COLLATION` 语句列出 **Nebula Graph** 目前支持的所有排序规则。目前支持四种排序规则：utf8_bin、utf8_general_ci、utf8mb4_bin、utf8mb4_general_ci。字符集为 utf8 时，默认 collate 为 utf8_bin；字符集为 utf8mb4 时，默认 collate 为 utf8mb4_bin。utf8_general_ci 和 utf8mb4_general_ci 都是忽略大小写的比较，行为同 MySQL 一致。

```
nebula> SHOW COLLATION;
======================
| Collation | Charset |
======================
| utf8_bin  | utf8    |
----------------------
```

`SHOW COLLATION` 输出有以下列：

- Collation 排序规则名称。
- Charset 与排序规则关联的字符集的名称。

最后更新: 2020年4月29日

2021 Vesoft Inc.

SHOW CONFIGS 语法

```
SHOW CONFIGS [graph|meta|storage]
```

`SHOW CONFIGS` 语句显示参数信息。 `SHOW CONFIGS` 输出以下列：module（模块信息）、name（参数名称）、type（参数类型）、mode（参数模式）和 value（参数值）。

例如：

```
nebula> SHOW CONFIGS graph;
==============================================================
| module | name                     | type  | mode    | value |
==============================================================
| GRAPH  | v                        | INT64 | MUTABLE | 0     |
--------------------------------------------------------------
| GRAPH  | minloglevel              | INT64 | MUTABLE | 2     |
--------------------------------------------------------------
| GRAPH  | slow_op_threshhold_ms    | INT64 | MUTABLE | 50    |
--------------------------------------------------------------
| GRAPH  | heartbeat_interval_secs  | INT64 | MUTABLE | 3     |
--------------------------------------------------------------
| GRAPH  | meta_client_retry_times  | INT64 | MUTABLE | 3     |
--------------------------------------------------------------
```

更多关于 `SHOW CONFIGS [graph|meta|storage]` 的信息，参见 configs syntax。

最后更新: 2020年5月12日

2021 Vesoft Inc.

SHOW CREATE SPACE 语法

```
SHOW CREATE SPACE <space_name>
```

`SHOW CREATE SPACE` 返回指定 space 及其创建语法。如果 space 包含默认值，则同时返回默认值。

```
nebula> SHOW CREATE SPACE basketballplayer;
==================================================================================================
| Space | Create Space                                                                           |
==================================================================================================
| basketballplayer  | CREATE SPACE gods (partition_num = 1, replica_factor = 1, charset = utf8, collate = utf8_bin) |
--------------------------------------------------------------------------------------------------
```

最后更新: 2021年4月16日

2021 Vesoft Inc.

SHOW CREATE TAGS/EDGES 语法

```
SHOW CREATE {TAG <tag_name> | EDGE <edge_name>}
```

`SHOW CREATE TAG` 和 `SHOW CREATE EDGE` 返回当前图空间中指定的 tag、edge type 及其创建语法。如果 tag 或 edge type 包含默认值，则同时返回默认值。

```
nebula> SHOW CREATE TAG player;
==================================================================================
| Tag    | Create Tag                                                            |
==================================================================================
| player | CREATE TAG player (
  name string,
  age int
) ttl_duration = 0, ttl_col = "" |
----------------------------------------------------------------------------------
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

SHOW HOSTS 语法

```
SHOW HOSTS
```

`SHOW HOSTS` 列出元服务器注册的所有存储主机。`SHOW HOSTS` 输出以下列：IP 地址、端口号、状态（online/offline）、leader 数量、leader 分布、partition 分布。

```
nebula> SHOW HOSTS;
==========================================================================================
| Ip         | Port  | Status | Leader count | Leader distribution  | Partition distribution |
==========================================================================================
| 172.28.2.1 | 44500 | online | 9            | basketballplayer: 9  | basketballplayer: 10   |
------------------------------------------------------------------------------------------
| 172.28.2.2 | 44500 | online | 0            |                      | basketballplayer: 10   |
------------------------------------------------------------------------------------------
| 172.28.2.3 | 44500 | online | 1            | basketballplayer: 1  | basketballplayer: 10   |
------------------------------------------------------------------------------------------
| Total      |       |        | 10           | basketballplayer: 10 | basketballplayer: 30   |
------------------------------------------------------------------------------------------
```

最后更新: 2021年4月16日

2021 Vesoft Inc.

SHOW INDEXES 语法

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

`SHOW INDEXES` 用于列出已创建完成的标签或边类型的索引信息。`SHOW INDEXES` 返回以下字段：索引 ID 和 索引名称。

例如：

```
nebula> SHOW TAG INDEXES;
==============================
| Index ID | Index Name     |
==============================
| 6        | player_index_1 |
------------------------------
| 7        | player_index_0 |
------------------------------
```

如何创建索引请参考 索引 文档。

最后更新: 2020年5月12日

2021 Vesoft Inc.

SHOW PARTS 语法

```
SHOW PARTS <part_id>
```

`SHOW PARTS` 列出指定 partition 的信息。`<part_id>` 为可选，如果不指定则返回所有 part 信息。

```
nebula> SHOW PARTS 1;
=============================================================
| Partition ID | Leader           | Peers            | Losts |
=============================================================
| 1            | 172.28.2.2:44500 | 172.28.2.2:44500 |       |
-------------------------------------------------------------
```

`SHOW PARTS` 输出以下列：

- Partition ID
- Leader
- Peers
- Losts

最后更新: 2020年5月12日

2021 Vesoft Inc.

SHOW ROLES 语法

```
SHOW ROLES IN <space_name>
```

`SHOW ROLES` 语句显示分配给用户账户的角色。 `SHOW ROLES` 输出以下列：用户账户和角色类型。

如果是 `GOD` 或 `ADMIN` 类型的用户, **Nebula Graph** 返回其权限内的所有用户角色。 如果是 `DBA` 、 `USER` 或 `GUEST` 类型的用户, **Nebula Graph** 仅返回其自身角色。

例如：

```
nebula> SHOW ROLES in NBA;
=====================
| Account | Role Type |
=====================
| userA   | ADMIN     |
----------------------
```

参考 Create User 创建用户，参考 Grant Role 为用户授予角色。

最后更新: 2020年5月13日

2021 Vesoft Inc.

SHOW SNAPSHOTS 语法

```
SHOW SNAPSHOTS
```

`SHOW SNAPSHOTS` 语句返回所有快照。

例如：

```
nebula> SHOW SNAPSHOTS;
=============================================================
| Name                          | Status | Hosts           |
=============================================================
| SNAPSHOT_2019_12_04_10_54_36 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_42 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_44 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------------
```

参考 这里 创建集群快照。

最后更新: 2020年5月12日

2021 Vesoft Inc.

SHOW SPACES 语法

```
SHOW SPACES
```

`SHOW SPACES` 列出 **Nebula Graph** 集群中的所有图空间。

例如：

```
nebula> SHOW SPACES;
====================
| Name             |
====================
| basketballplayer |
--------------------
```

参考这里创建图空间。

最后更新: 2021年4月16日

2021 Vesoft Inc.

SHOW TAGS/EDGES 语法

```
SHOW {TAGS | EDGES}
```

`SHOW TAGS` 和 `SHOW EDGES` 则返回当前图空间中被定义的 tag 和 edge type。

最后更新: 2020年4月29日

2021 Vesoft Inc.

SHOW USERS 语法

```
SHOW USERS
```

`SHOW USERS` 语句显示用户信息。 `SHOW USERS` 输出以下列：账户名。

最后更新: 2020年4月29日

2021 Vesoft Inc.

DESCRIBE 语法

```
DESCRIBE SPACE <space_name>
DESCRIBE TAG <tag_name>
DESCRIBE EDGE <edge_name>
DESCRIBE {TAG | EDGE} INDEX <index_name>
```

DESCRIBE 关键词的作用是获取关于 space, tag, edge 结构的信息。

同时需要注意的是，DESCRIBE 和 SHOW 也是不同的。 详细参见 SHOW 文档。

示例

获取指定 space 的信息，对应 `DESCRIBE SPACE`。

```
nebula> DESCRIBE SPACE basketballplayer;
=====================================================================
| ID |     Name                 | Partition number | Replica Factor |
=====================================================================
|  1 |     basketballplayer     |             100  |              1 |
---------------------------------------------------------------------
```

获取指定 tag 的信息，对应 `DESCRIBE TAG`。

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

获取指定 EDGE 的信息，对应 `DESCRIBE EDGE`。

```
nebula> DESCRIBE EDGE serve;
====================
| Field      | Type |
====================
| start_year | int  |
--------------------
| end_year   | int  |
--------------------
```

返回指定索引信息，对应 `DESCRIBE INDEX`。

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

最后更新: 2021年4月16日

2021 Vesoft Inc.

USE 语法

```
USE <graph_space_name>
```

在 **Nebula Graph** 中 `USE` 语句的作用是选择一个图空间来作为当前的工作图空间。`USE` 需要一些特定的权限来执行。

当前的图空间会保持默认直至当前会话结束或另一个 `USE` 语句被执行。

```
nebula> USE space1;
-- 遍历 space1。
nebula> GO FROM 1 OVER edge1;
-- 使用 space2。space2 中的数据与 space1 物理隔离。
nebula> USE space2;
nebula> GO FROM 2 OVER edge2;
-- 回到 space1。至此你不能从 space2 中读取数据。
nebula> USE space1;
```

和 SQL 不同的是，选取一个当前的工作图空间会阻止你访问其他图空间。遍历一个新的图空间的唯一方案是通过 `USE` 语句来切换工作图空间。

> SPACES 之间是 `完全隔离的`。不像 SQL 允许在一个语句中选择两个来自不同数据库的表单，在 **Nebula Graph** 中一次只能对一个图空间进行操作。

最后更新: 2020年4月29日

2021 Vesoft Inc.

2.5.4 图算法

FIND PATH 语法

`FIND PATH` 语法用于获取最短路径及全路径。

```
FIND SHORTEST | ALL PATH FROM <vertex_id_list> TO <vertex_id_list>
OVER <edge_type_list> [UPTO <N> STEPS]
```

`SHORTEST` 寻找最短路径关键词。

`ALL` 寻找全路径关键词。

`<vertex_id_list>::=[vertex_id [, vertex_id]]` 为节点列表，用逗号隔开。支持输入 `$-` 及变量 `$var`。

`<edge_type_list>` 指定边的类型，多种边类型用 `,` 隔开，用 `*` 表示所有边类型。

`<N>` 为跳数，默认值 5。

注意事项

- 当起点及终点是 ID 列表时，表示寻找从任意起点开始到终点的最短路径。
- 全路径会有环。

示例

在 console 中，路径显示方式为 `id <edge_name, rank> id`。

```
nebula> FIND SHORTEST PATH FROM 100 to 200 OVER *;
============================
| _path_ |
============================
| 100 <serve,0> 200
----------------------------
```

```
nebula> FIND ALL PATH FROM 100 to 200 OVER *;
=================================================================================================
| _path_ |
=================================================================================================
| 100 < serve,0> 200
-------------------------------------------------------------------------------------------------
| 100 <follow,0> 101 < serve,0> 200
-------------------------------------------------------------------------------------------------
| 100 <follow,0> 102 < serve,0> 200
-------------------------------------------------------------------------------------------------
| 100 <follow,0> 106 < serve,0> 200
-------------------------------------------------------------------------------------------------
```

最后更新: 2020年6月23日

2021 Vesoft Inc.

3. 编译、部署与运维

3.1 编译

3.1.1 使用源码编译

前言

我们已经针对多种不同的环境做过编译测试，包括 CentOS 6/7/8、Ubuntu 16.04/18.04/19.04、Fedora 28/29/30、GCC 7/8/9 以及较新版本的 LLVM/Clang。然而，由于编译环境及依赖的复杂性，很难保证覆盖到所有场景。如果在编译过程遇到任何问题，欢迎通过 Issue 或者 Pull Request 联系我们。

系统要求

以下为编译**Nebula Graph**机器所需配置要求，运行环境所需配置要求见这里。

- 处理器: x86_64
- 内存: 至少 4GB
- 存储空间: 至少 10GB
- Linux 内核: 2.6.32 或更高版本，通过命令 `uname -r` 查看
- glibc: 2.12 或更高版本，通过命令 `ldd --version` 查看
- GCC: 7.1.0 或更高版本，通过命令 `g++ --version` 查看
- CMake: 3.5.0 或更高版本，通过命令 `cmake --version` 查看
- 能够访问互联网

注意: Nebula Graph 目前仅支持 x86_64 架构。

快速编译步骤

安装系统依赖

请注意，安装系统依赖需要 root 权限。

CentOS，RedHat 和 Fedora 用户可以运行以下命令安装：

```
$ 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

# CentOS 8+, RedHat 8+ 以及 Fedora 用户，需要额外安装 libstdc++-static 和 libasan
$ yum install -y libstdc++-static libasan
```

Debian 及 Ubuntu 用户，执行以下命令安装：

```
$ apt-get update
$ apt-get install -y make \
                     m4 \
                     git \
                     wget \
```


2021 Vesoft Inc.

```
                    unzip \
                    xz-utils \
                    curl \
                    lsb-core \
                    build-essential \
                    libreadline-dev \
                    ncurses-dev \
                    cmake \
                    gettext
```

ArchLinux、Gentoo 或者 LFS 用户请自行安装。

在开始编译之前，请确保编译器和 CMake 版本满足要求：

```
$ g++ --version
$ cmake --version
```

否则，请分别参考 安装 GCC 或 安装 CMake 进行操作。

克隆源码

```
$ git clone https://github.com/vesoft-inc/nebula.git
```

如果不关心代码仓库的历史提交信息，您可进行_浅克隆_（Shallow clone）以加快下载速度：

```
$ git clone --depth=1 https://github.com/vesoft-inc/nebula.git
```

执行编译

```
$ cd nebula
$ mkdir build
$ cd build
$ cmake -DENABLE_TESTING=OFF -DCMAKE_BUILD_TYPE=Release ..
# 假设 cores 为核数，mem_gb 为内存大小（单位为 GB），N 取值建议为 cores 和 mem_gb/2 中的较小值
# Build type 建议选择 release 以加快编译速度
$ make -jN
# 默认安装目录为 /usr/local/nebula
$ sudo make install
# 如需启动服务，请复制 etc/ 目录下的配置文件
# 用于生产环境
$ cd /usr/local/nebula
$ cp etc/nebula-storaged.conf.production etc/nebula-storaged.conf
$ cp etc/nebula-metad.conf.production etc/nebula-metad.conf
$ sudo cp etc/nebula-graphd.conf.production etc/nebula-graphd.conf
# 用于试用
$ cd /usr/local/nebula
$ sudo cp etc/nebula-storaged.conf.default etc/nebula-storaged.conf
$ sudo cp etc/nebula-metad.conf.default etc/nebula-metad.conf
$ sudo cp etc/nebula-graphd.conf.default etc/nebula-graphd.conf
```

详情参考启动和停止 Nebula Graph 服务文档。

由于 **Nebula Graph** 使用了大量的 C++ 模板，尤其是 Folly，fbthrift 和 boost，因此编译会非常耗时。比如，如果使用 Intel E5-2697 v3 处理器，在 16 个任务并发运行的情况下，需要花费大约 4 分钟完成编译，总的 CPU 时间大约 35 分钟。

源码打包（可选）

- 如需将 **Nebula Graph** 打包至一个包，请使用以下命令：

```
cd nebula/package
./package.sh -v <version>
```

- 如需将 **Nebula Graph** 打包至多个包，请使用以下命令：

```
cd nebula/package
./package.sh -v <version> -n OFF
```

编译选项

除默认选项外，**Nebula Graph** 的编译系统还提供诸多选项来调整编译行为。

CMAKE 参数

可通过 `cmake -DArgument=Value ..` 调整 CMake 参数。

2021 Vesoft Inc.

ENABLE_WERROR

默认情况下，**Nebula Graph** 使用 `-Werror` 选项将编译过程中的告警当成错误。如果在编译过程中遇到了类似情况，可以通过将 `ENABLE_WERROR` 设置为 `OFF` 来暂时忽略此类错误。

ENABLE_TESTING

该选项允许用户开启或关闭单元测试的编译，默认开启。如果您只需要编译 **Nebula Graph** 服务模块，可以将该选项设置为 `OFF`。

ENABLE_ASAN

该选项允许用户开启或关闭 AddressSanitizer（内存相关错误检测器），默认关闭。

CMAKE_BUILD_TYPE

Nebula Graph 支持以下几种编译类型：

- `Debug`，启用调试信息，不启用优化选项，为默认编译类型
- `Release`，启用优化选项，不启用调试信息
- `RelWithDebInfo`，启用优化选项，且启用调试信息
- `MinSizeRel`，启用利于减小代码体积的优化选项，不启用调试信息

CMAKE_INSTALL_PREFIX

该选项用于指定执行 `make install` 命令时，**Nebula Graph** 的服务模块、配置文件以及工具集的安装路径，默认为 `/usr/local/nebula`。

CMAKE_CXX_COMPILER

通常情况下，CMake 会自动选择合适的编译器。但是，如果目标编译器不在默认的标准路径下，或者你想使用其他种类或路径下的编译器，请使用如下方式指定：

```
$ 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

`ccache` 可以加快编译过程，主要用于开发过程。如果系统中安装了 `ccache`，**Nebula Graph** 默认会自动启用该选项。

但是，如果你想禁用 `ccache`，将该选项设置成 `OFF` *可能* 是不够的。因为，在某些系统中，`ccache` 会_代理_当前编译器。此时，需要通过设置环境变量 `export CCACHE_DISABLE=true`，或者在 `~/.ccache/ccache.conf` 文件中添加 `disable=true`。后续 **Nebula Graph** 将隐藏这些细节。

另外，关于 `ccache` 的更多细节，请参考官方文档。

NEBULA_USE_LINKER

该选项允许我们使用不同的链接器。目前可用的选项是：`bfd`，`gold`，`lld`。其中，`bfd` 和 `gold` 隶属于 GNU binutils，`lld` 则需要安装 LLVM/Clang。此外，如果需要，还可以使用该参数指定链接器的绝对路径。

NEBULA_THIRDPARTY_ROOT

该选项用于显式指定 third party 所在路径。

手动安装 THIRD PARTY

在 configure/cmake 阶段，**Nebula Graph** 默认将预先编译好的 third party 下载到当前 build 目录。但是如果你想将其安装到其他路径（比如，安装到某个公共目录），你可以：

```
# 安装 third party 至 /opt 需要 root 权限，可使用 --prefix 改变安装路径
$ ../third-party/install-third-party.sh --prefix=/opt/vesoft/third-party
```


2021 Vesoft Inc.

如果不指定 `--prefix`，third party 的默认安装路径为 `/opt/vesoft/third-party`，且可为 **Nebula Graph** 的编译系统自动找到。否则，需使用上文所述的 `NEBULA_THIRDPARTY_ROOT` CMake 参数指定路径，或为该路径设置环境变量并导出。 **Nebula Graph** 查找并选择 third party 的优先级如下：

1. CMake 变量 `NEBULA_THIRDPARTY_ROOT`
2. build 路径下的 `third-party/install`
3. `NEBULA_THIRDPARTY_ROOT` 环境变量
4. `/opt/vesoft/third-party`

安装可用的 CMake

对于没有可用 CMake 安装的用户， 我们提供了可自动下载安装的脚本。在 build 目录下，运行：

```
$ ../third-party/install-cmake.sh cmake-install
CMake has been installed to prefix=cmake-install
Run 'source cmake-install/bin/enable-cmake.sh' to make it ready to use.
Run 'source cmake-install/bin/disable-cmake.sh' to disable it.

$ source cmake-install/bin/enable-cmake.sh
$ cmake --version
cmake version 3.15.5
```

此时可用的 CMake 已安装完成。你可以在任何时候使用 `source cmake-install/bin/disable-cmake.sh` 命令将其禁用。

安装可用的 GCC

对于没有可用 GCC 安装的用户， 我们提供了 GCC 和可自动下载安装的脚本。在 build 目录下，运行：

```
# 将 GCC 安装至 /opt 需要 root 权限，支持更改安装路径
$ ../third-party/install-gcc.sh --prefix=/opt
GCC-7.5.0 has been installed to /opt/vesoft/toolset/gcc/7.5.0
Performing usability tests
Performing regular C++14 tests...OK
Performing LeakSanitizer tests...OK
Run 'source /opt/vesoft/toolset/gcc/7.5.0/enable' to start using.
Run 'source /opt/vesoft/toolset/gcc/7.5.0/disable' to stop using.

# 注意路径和指定版本可能与你的环境不同
$ source /opt/vesoft/toolset/gcc/7.5.0/enable
# 此处仅设置了 PATH，以免污染库路径
# 如果需要可运行 'export LD_LIBRARY_PATH=/opt/vesoft/toolset/gcc/7.5.0/lib64:$LD_LIBRARY_PATH'

$ g++ --version
g++ (Nebula Graph Build) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
```

此时可用的 GCC 编译器已安装完成。你可以在任何时候使用 `source /opt/vesoft/toolset/gcc/7.5.0/disable` 命令将其禁用。

无网络编译

如果在编译源码时无法连接网络，则必须手动下载以上工具和依赖，包括 **Nebula Graph** 仓库中的 GCC 编译器，第三方库和 CMake。然后，将所有内容复制到你的机器上。以下是快速指南。详细信息请参考上述步骤。

首先，需要有一台可以连接外网的主机，并下载以下文件：

```
# 下载 GCC
# RedHat 或 CentOS 用户
$ wget https://oss-cdn.nebula-graph.com.cn/toolset/vesoft-gcc-7.5.0-CentOS-x86_64-glibc-2.12.sh
# Debian 或 Ubuntu 用户
$ wget https://oss-cdn.nebula-graph.com.cn/toolset/vesoft-gcc-7.5.0-Debian-x86_64-glibc-2.13.sh

# 下载 CMake
$ wget https://cmake.org/files/v3.15/cmake-3.15.5-Linux-x86_64.sh

# 下载第三方库
$ wget https://oss-cdn.nebula-graph.com.cn/third-party/vesoft-third-party-x86_64-libc-2.12-gcc-7.5.0-abi-11.sh
```

然后，将这些软件包复制到编译的机器并运行。

```
# 安装 GCC
# RedHat 或 CentOS 用户
$ sudo bash vesoft-gcc-7.5.0-CentOS-x86_64-glibc-2.12.sh
```


2021 Vesoft Inc.

```
# Debian 或 Ubuntu 用户
$ sudo bash vesoft-gcc-7.5.0-Debian-x86_64-glibc-2.13.sh

# 启用 GCC 安装
$ source /opt/vesoft/toolset/gcc/7.5.0/enable

# 安装 CMake
$ sudo bash cmake-3.15.5-Linux-x86_64.sh --skip-license --prefix=/opt/vesoft/toolset/cmake

# 将安装好的cmake的bin目录加到PATH里面

$ export PATH=/opt/vesoft/toolset/cmake:$PATH

# 安装第三方库
$ sudo bash vesoft-third-party-x86_64-libc-2.12-gcc-7.5.0-abi-11.sh
```

现在您就可以下载并编译 **Nebula Graph** 项目了。

卸载

卸载前，请先停止服务。在 `make` 目录下使用 `make uninstall` 命令即可卸载 **Nebula Graph**。请注意卸载之后配置文件不会删除。

FAQ

ERROR: INVALID ARGUMENT TYPE 'AUTO' TO UNARY EXPRESSION

当使用 Clang 9.0 编译 **Nebula Graph** 时，会发生该错误：

```
[  5%] Building CXX object src/common/fs/CMakeFiles/fs_obj.dir/FileUtils.cpp.o
In file included from src/common/fs/FileUtils.cpp:8:
In file included from src/common/fs/FileUtils.h:12:
src/common/base/StatusOr.h:57:19: error: invalid argument type 'auto' to unary expression
    static_assert(!is_status_v<T>, "`T' must not be of type `Status'");
                  ^~~~~~~~~~~~~~
src/common/fs/FileUtils.cpp:90:34: note: in instantiation of template class 'nebula::StatusOr<std::__cxx11::basic_string<char> >' requested here
StatusOr<std::string> FileUtils::readLink(const char *path) {
...
```

这是 Clang 9.0 引入的一个已知的 Bug，Clang 10.0（2020-05-25）尚未修复。

最后更新: 2020年8月11日

2021 Vesoft Inc.

3.1.2 通过 Docker 构建

Nebula Graph 提供整个编译环境的 docker 镜像 vesoft/nebula-dev，支持在本地更改源代码，构建并在容器中调试。执行以下步骤以开始快速开发：

从 Docker Hub 获取镜像

```
bash> docker pull vesoft/nebula-dev
```

运行 docker 容器并将本地源码目录挂载到容器工作目录 `/home/nebula` 中

```
bash> docker run --rm -ti \
  --security-opt seccomp=unconfined \
  -v /path/to/nebula/directory:/home/nebula \
  -w /home/nebula \
  vesoft/nebula-dev \
  bash
```

将 `/path/to/nebula/directory` 替换成你个人的 **源码路径**。

在容器内编译

```
docker> mkdir _build && cd _build
docker> cmake ..
docker> make
docker> make install
```

启动 Nebula Graph 服务

上述步骤完成后即可在容器内启动服务，默认安装目录为 `/usr/local/nebula/` 。

```
docker> cd /usr/local/nebula
```

重命名 **Nebula Graph** 服务的配置文件

```
docker> cp etc/nebula-graphd.conf.default etc/nebula-graphd.conf
docker> cp etc/nebula-metad.conf.default etc/nebula-metad.conf
docker> cp etc/nebula-storaged.conf.default etc/nebula-storaged.conf
```

启动服务

```
docker> ./scripts/nebula.service start all
docker> ./bin/nebula -u root -p nebula --port 3699 --addr="127.0.0.1"
nebula> SHOW HOSTS;
```

最后更新: 2020年7月28日

2021 Vesoft Inc.

3.2 安装

3.2.1 使用 rpm/deb 包安装 **Nebula Graph**

概览

本指南将指导您使用 `rpm/deb` 包来安装 **Nebula Graph**。

前提条件

请参考运行配置要求文档。

2021 Vesoft Inc.

安装 Nebula Graph

使用 `rpm/deb` 包来安装 **Nebula Graph**，需要完成以下步骤：

2021 Vesoft Inc.

1. 下载安装包
   - 方式一：通过阿里云 OSS 获取安装包。（国内用户可优先考虑使用 OSS 下载）

     a. 获取 release 版本，URL 格式如下：

```
* Centos 6: https://oss-cdn.nebula-graph.com.cn/package/${release_version}/nebula-${release_version}.el6-5.x86_64.rpm

* Centos 7: https://oss-cdn.nebula-graph.com.cn/package/${release_version}/nebula-${release_version}.el7-5.x86_64.rpm

* Ubuntu 1604: https://oss-cdn.nebula-graph.com.cn/package/${release_version}/nebula-${release_version}.ubuntu1604.amd64.deb

* Ubuntu 1804: https://oss-cdn.nebula-graph.com.cn/package/${release_version}/nebula-${release_version}.ubuntu1804.amd64.deb
```

     链接中 `${release_version}` 为具体的发布版本号，例如要下载 `1.2.1` `Centos 7.5` 的安装包，那么可以直接通过命令下载。

```
$ wget https://oss-cdn.nebula-graph.com.cn/package/1.2.1/nebula-1.2.1.el7-5.x86_64.rpm
```

     b. 获取 nightly (最新测试)版本，URL 格式如下：

```
* Centos 6: https://oss-cdn.nebula-graph.com.cn/package/nightly/${date}/nebula-${date}-nightly.el6-5.x86_64.rpm

* Centos 7: https://oss-cdn.nebula-graph.com.cn/package/nightly/${date}/nebula-${date}-nightly.el7-5.x86_64.rpm

* Ubuntu 1604: https://oss-cdn.nebula-graph.com.cn/package/nightly/${date}/nebula-${date}-nightly.ubuntu1604.amd64.deb

* Ubuntu 1804: https://oss-cdn.nebula-graph.com.cn/package/nightly/${date}/nebula-${date}-nightly.ubuntu1804.amd64.deb
```

     链接中 `${date}` 为具体的日期，例如要下载 `2020年4月1日` 的 `Centos 7.5` 的安装包，那么可以直接通过命令下载

```
$ wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2020.04.01/nebula-2020.04.01-nightly.el7-5.x86_64.rpm
```


2021 Vesoft Inc.

- 方式二：通过 GitHub 获取安装包

  a. 登录到 GitHub 并单击 rpm/deb 链接。

  b. 在 **Actions** 选项卡下，单击左侧的 **package**，显示所有可用的包。

  c. 单击列表顶部最新的包。

  d. 单击右上角 **Artifacts**， 选择要下载的安装包。

2. 安装 **Nebula Graph**

- 如果是 `rpm` 文件，使用以下命令安装 **Nebula Graph**：

```
sudo rpm -ivh nebula-2019.12.23-nightly.el6-5.x86_64.rpm
```

- 如果是 `deb` 文件，使用以下命令安装 **Nebula Graph**：

```
sudo dpkg -i nebula-2019.12.23-nightly.ubuntu1604.amd64.deb
```

- 如需安装至自定义目录，请使用以下命令：

```
rpm -ivh --prefix=${your_dir} nebula-graph-${version}.rpm
```


2021 Vesoft Inc.

注意：

1. 使用您自己的文件名替换以上命令中的文件名，否则以上命令可能执行失败。
2. **Nebula Graph** 默认会安装在 `/usr/local/nebula` 目录下。

启动 Nebula Graph 服务

详情参考启动和停止 Nebula Graph 服务文档。

卸载

卸载前，请先停止服务。如果使用 rpm 安装，使用 `rpm -qa | grep nebula` 命令搜索 `nebula`，然后把结果传给 rpm -e 同理即可卸载。使用 deb 安装则需通过 `dpkg` 卸载。请注意卸载之后配置文件不会删除。

```
# rpm
$ rpm -qa|grep nebula
nebula-graph-1.2.1-1.x86_64
$ sudo rpm -e nebula-graph-1.2.1-1.x86_64

# deb
$ dpkg -l|grep nebula
nebule-graph
$ dpkg -r nebule-graph
```

最后更新: 2021年4月15日

2021 Vesoft Inc.

3.2.2 启动和停止 Nebula Graph 服务

输入以下命令启动 Nebula Graph 服务

```
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
```

查看 Nebula Graph 服务

输入以下命令查看 **Nebula Graph** 服务：

```
sudo /usr/local/nebula/scripts/nebula.service status all
[INFO] nebula-metad: Running as 9576, Listening on 45500
[INFO] nebula-graphd: Running as 9679, Listening on 3699
[INFO] nebula-storaged: Running as 9812, Listening on 44500
```

连接 Nebula Graph 服务

输入以下命令连接 **Nebula Graph** 服务：

```
sudo /usr/local/nebula/bin/nebula -u <user> -p <password> [--addr=<graphd IP> --port=<graphd port>]

Welcome to Nebula Graph (Version 1.2.1)

nebula> SHOW HOSTS;
```

- -u 用户名称，默认值为 `root`
- -p 密码，用户 `root` 的默认密码为 `nebula`
- --addr 为 graphd IP, 默认使用 127.0.0.1
- --port 为 graphd port，默认值为 `3699`
- SHOW HOSTS 命令检查已成功连接的 storaged 服务

注意： `enable_authorize` 默认关闭，此时接受任意账号密码和无账号连接。如果启用，则默认用户名和密码分别为 root 和 nebula。 参见 Built-in Roles 文档。

停止 Nebula Graph 服务

输入以下命令停止 **Nebula Graph** 服务：

```
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
```

请注意不要通过 `kill -9` 关闭服务，否则可能较小概率导致数据丢失，建议使用以上方式停止服务。

启停单个 Nebula Graph 模块

可以使用脚本 nebula.service 来控制对单个模块的启停。

2021 Vesoft Inc.

```
sudo /usr/local/nebula/scripts/nebula.service
Usage: ./nebula.service [-v] [-c /path/to/config] <start|stop|restart|status|kill> <metad|graphd|storaged|all>
```

- -v 本脚本的详细调试信息
- -c 配置文件路径，默认为安装路径(/usr/local/nebula/)下的 `etc/` 目录。

最后更新: 2021年4月9日

2021 Vesoft Inc.

3.2.3 集群部署

在本文档中，我们将指导您使用部署 **Nebula Graph** 集群。同时我们也使用 Docker 部署好了集群，以便您可以在几分钟之内试用。

前提条件

在开始部署 **Nebula Graph** 集群前，确保您已在集群的每一台机器上安装最新版本的 **Nebula Graph**。安装方式参考：

- 包安装
- 源码编译安装

由于 **Nebula Graph** 依赖较多，推荐使用安装包安装。

本文档中，我们准备了 3 台装有 CentOS 7.5 系统的机器， IP 如下：

```
192.168.8.14 # cluster-14
192.168.8.15 # cluster-15
192.168.8.16 # cluster-16
```

要部署的 Nebula Graph 服务

在本文档中我们将部署以下 **Nebula Graph** 服务：

- 3 副本 `nebula-metad` 服务
- 3 副本 `nebula-storaged` 服务
- 3 副本 `nebula-graphd` 服务

```
- cluster-14: metad/storaged/graphd
- cluster-15: metad/storaged/graphd
- cluster-16: metad/storaged/graphd
```

修改配置文件

Nebula Graph 的所有配置文件均位于 `/usr/local/nebula/etc` 目录下，并且提供了三份默认配置。

NEBULA-METAD.CONF

部署集群时，需要根据每个节点上部署的服务修改相应配置文件中的两个参数： `local_ip` 和 `meta_server_addrs`。 `local_ip` 要修改成节点的 IP， `meta_server_addrs` 需要修改成集群上 meta 服务的 ip:port 对。多个 ip:port 对之间需要用逗号隔开。

`cluster-14` 上的两项配置示例如下所示：

```
# Peers
--meta_server_addrs=192.168.8.14:45500,192.168.8.15:45500,192.168.8.16:45500
# Local ip
--local_ip=192.168.8.14
# Meta daemon listening port
--port=45500
```

NEBULA-GRAPHD.CONF

部署集群时，需要为 graphd 服务配置 metad 的地址和端口 `meta_server_addrs`。 cluster-14 上的部分配置如下：

```
# Meta Server Address
--meta_server_addrs=192.168.8.14:45500,192.168.8.15:45500,192.168.8.16:45500
```

NEBULA-STORAGED.CONF

部署集群时，需要为 storaged 服务配置 metad 的地址和端口 `meta_server_addrs` 以及本机地址 `local_ip`。 cluster-14 上的部分配置如下：

```
# Meta server address
--meta_server_addrs=192.168.8.14:45500,192.168.8.15:45500,192.168.8.16:45500
# Local ip
--local_ip=192.168.8.14
```


2021 Vesoft Inc.

```
# Storage daemon listening port
--port=44500
```

启动集群

集群启动请参考 启停 Nebula Graph 服务文档。

测试集群

登录集群中的一台机器，执行如下命令：

```
[(none)]> SHOW HOSTS;
==========================================================================================================================
| Ip           | Port  | Status | Leader count | Leader distribution                                | Partition distribution               |
==========================================================================================================================
| 192.168.8.14 | 44500 | online | 0            | No valid partition                                 | No valid partition                   |
--------------------------------------------------------------------------------------------------------------------------
| 192.168.8.15 | 44500 | online | 3            | toy: 1, test: 1, basketballplayer: 1               | basketballplayer: 1, toy: 1, test: 1 |
--------------------------------------------------------------------------------------------------------------------------
| 192.168.8.16 | 44500 | online | 0            | No valid partition                                 | No valid partition                   |
--------------------------------------------------------------------------------------------------------------------------
| Total        |       |        | 3            | basketballplayer: 1, toy: 1, test: 1               | basketballplayer: 1, test: 1, toy: 1 |
--------------------------------------------------------------------------------------------------------------------------
```

最后更新: 2021年4月16日

2021 Vesoft Inc.

3.2.4 使用 Docker 安装

访问这里。

最后更新: 2020年5月26日

2021 Vesoft Inc.

3.3 配置

3.3.1 运行配置要求

生产环境

生产环境部署方式

- 3 个元数据服务进程 `metad`
- 至少 3 个存储服务进程 `storaged`
- 至少 3 个查询引擎服务进程 `graphd`

以上进程都无需独占机器。例如一个由 5 台机器组成的集群：A、B、C、D、E，可以如下部署：

- A：metad, storaged, graphd
- B：metad, storaged, graphd
- C：metad, storaged, graphd
- D：storaged, graphd
- E：storaged, graphd

> 同一个集群不要跨机房部署。 metad 每个进程都会创建一份元数据的存储副本，因此通常只需 3 个进程。storaged 进程数量不影响图空间数据的副本数量。

服务器配置要求(标准配置)

以 AWS EC2 c5d.12xlarge 为例：

- 处理器：48 core
- 内存：96 GB
- 存储：2 * 900 GB, NVMe SSD
- Linux 内核：3.9 或更高版本，通过命令 `uname -r` 查看
- glibc：2.12 或更高版本，通过命令 `ldd --version` 查看

操作系统配置见这里。

测试环境

- 1 个元数据服务进程 `metad`
- 至少 1 个存储服务进程 `storaged`
- 至少 1 个查询引擎服务进程 `graphd`

例如一个有 3 台机器的集群：A、B、C 可以如下部署：

- A：metad、storaged、graphd
- B：storaged、graphd
- C：storaged、graphd

2021 Vesoft Inc.

服务器配置要求(最低配置)

以 AWS EC2 c5d.xlarge 为例：

- 处理器：4 core
- 内存：8 GB
- 存储：100 GB, SSD

资源估算（3副本标准配置）

- 存储空间（全集群）：点和边数量 * 平均属性的字节数 * 6
- 内存（全集群）：点边数量 * 15 字节 + RocksDB 实例数量 * (write_buffer_size * max_write_buffer_number + rocksdb_block_cache), 其中 `etc/nebula-storaged.conf` 文件中 `--data_path` 项中的每个目录对应一个 RocksDB 实例
- 图空间 partition 数量：全集群硬盘数量 *（2 至 10 —— 硬盘越好该值越大）
- 内存和硬盘另预留 20% buffer。

关于机械硬盘和千兆网络

Nebula Graph 设计时主要针对的硬件设备是 NVMe SSD 和万兆网。没有对于机械磁盘和千兆网络做过适配，以下是一些需调整的参数：

- etc/nebula-storage.conf：
  - --raft_rpc_timeout_ms= 5000 至 10000
  - --rocksdb_batch_size= 4096 至 16384
  - --heartbeat_interval_secs = 30 至 60
  - --raft_heartbeat_interval_secs = 30 至 60
- etc/nebula-meta.conf：
  - --heartbeat_interval_secs 与 etc/nebula-storage.conf 该项相同
- Spark Writer:

```
rate: {
      timeout: 5000 至 10000
    }
```

- go-importer:
  - batchSize: 10 至 50
  - concurrency: 1 至 10
  - channelBufferSize：100 至 500
- partition 值为全集群硬盘数量 2 倍

最后更新: 2020年8月11日

2021 Vesoft Inc.

3.3.2 配置项持久化与优先级

配置持久化（生产用）

Nebula Graph 服务第一次启动的时候，会从本地读取配置文件（默认路径为 `/usr/local/nebula/etc/` ），然后所有配置项（包括动态更改的配置项）都会被**持久化**在 Meta Service 中。之后即使发生 **Nebula Graph** 重启，也都只会从 Meta Service 读取配置。

只从本地获取配置（调试用）

在部分调试场景中，需要从**本地**而不是 Meta Service 中获取配置，此时请在配置文件顶部添加 `--local_config=true` 。更改后需重启服务方可生效。

更改方式和读取优先级

Nebula Graph 的参数项也支持通过命令行终端命令(`UPDATE CONFIG` 语法)或者设置环境变量的方式来更改，读取优先级规则如下：

对于一个参数项：

- 默认的配置寻找优先级：`meta service` > 命令行 `UPDATE CONFIG` 语法 > 本地环境变量 > 本地配置文件。
- 如果 `--local_config=true` ：本地配置文件 > `meta service` > 本地环境变量。此时不推荐使用 `UPDATE CONFIG` 。

最后更新: 2020年5月29日

2021 Vesoft Inc.

3.3.3 CONFIGS 语法

显示配置

```
SHOW CONFIGS [graph|storage]
```

例如

```
nebula> SHOW CONFIGS;
=====================================================================
| module  | name                              | type   | mode    | value
=====================================================================
| GRAPH   | v                                 | INT64  | MUTABLE | 0
---------------------------------------------------------------------
| GRAPH   | minloglevel                       | INT64  | MUTABLE | 0
---------------------------------------------------------------------
...
```

获取变量

```
GET CONFIGS [graph|storage :] <var>
```

例如

```
nebula> GET CONFIGS storage:v;
=========================================
| module  | name | type  | mode    | value |
=========================================
| STORAGE | v    | INT64 | MUTABLE | 0     |
-----------------------------------------
```

更新变量

```
UPDATE CONFIGS [graph|storage :] <var> = <value>
```

更新的变量将持久化在 `meta service` 中。 如果配置模式为 `MUTABLE`，则更改将立即生效。 部分 RocksDB 参数需重启才会生效。 支持在 `UPDATE CONFIGS` 中使用算术计算。

例如：

```
nebula> UPDATE CONFIGS storage:heartbeat_interval_secs=1;
nebula> GET CONFIGS storage:heartbeat_interval_secs;
=============================================================
| module  | name                     | type  | mode    | value |
=============================================================
| STORAGE | heartbeat_interval_secs | INT64 | MUTABLE | 1     |
-------------------------------------------------------------
```

最后更新: 2020年7月27日

2021 Vesoft Inc.

3.3.4 Graph 配置

本文介绍 `graphd` 配置文件。配置文件默认在 `/usr/local/nebula/etc/` 目录下。如果您已指定 **Nebula Graph** 安装路径，则配置文件目录为 `$pwd/nebula/etc/` 。

- `*.default` 文件为**日常调试**使用，也是服务**启动时默认使用**的配置文件
- `*.production` 文件为**推荐生产时**使用的的文件，生产时请去掉 `.production` 后缀

Basics 基础配置

属性名	默认值	说明
`daemonize`	true	作为 daemon 进程运行
`pid_file`	"pids/nebula-graphd.pid"	进程 PID 文件

logging 日志相关

属性名	默认值	说明	动态修改
`log_dir`	logs (也即 `/usr/local/nebula/logs`)	graphd 日志文件存放路径，建议与 storaged 的数据目录放不同硬盘	
`minloglevel`	0	对应的日志级别分别为 INFO(DEBUG)，WARNING，ERROR，FATAL。通常在调试环境设置为 0，生产环境设置为 1，设置为 4 不打印任何日志。	UPDATE CONFIGS 命令修改，立刻生效
`v`	0	0-4: 当 minloglevel 设置为 0 时，可以进一步设置调试日志的详细程度，值越大越详细	UPDATE CONFIGS 命令修改，立刻生效
`logbufsecs`	0 （秒）	日志缓存时间	UPDATE CONFIGS 命令修改，立刻生效
`redirect_stdout`	true	将 stdout 和 stderr 重定向到单独的文件	
`stdout_log_file`	"stdout.log"	stdout 目标文件名	
`stderr_log_file`	"stderr.log"	stderr 目标文件名	
`slow_op_threshhold_ms`	50 (毫秒)	慢操作和慢查询需要打印日志的阈值	UPDATE CONFIGS 命令修改，立刻生效

例如，下面命令可以增加 graphd 的日志详细程度 v=1。

```
nebula> UPDATE CONFIGS graph:v=1;
```


2021 Vesoft Inc.

networking 网络通信相关

属性名	默认值	说明	动态修改
`meta_server_addrs`	"127.0.0.1:45500"	meta server 地址列表，格式为 ip1:port1, ip2:port2, ip3:port3	
`port`	3699	RPC 监听端口	
`meta_client_retry_times`	3	与 meta service 重试次数	UPDATE CONFIGS 命令修改，立刻生效
`heartbeat_interval_secs`	3 (秒)	与 meta service 心跳时长	UPDATE CONFIGS 命令修改，下个心跳周期生效
`client_idle_timeout_secs`	0	关闭 idle 连接前的时长（单位秒）， 0 为无穷大	
`session_idle_timeout_secs`	0	idle sessions 过期时长（单位秒），0 为无穷大	
`num_netio_threads`	0	networking 线程数，0为物理 CPU 核数	
`num_accept_threads`	1	接受进入连接的线程数	
`num_worker_threads`	0	执行用户请求的线程数，线程数为系统 CPU 核数	
`reuse_port`	true	开启内核（>3.9）`SO_REUSEPORT` 选项	
`listen_backlog`	1024	listen socket 的 backlog	
`listen_netdev`	"any"	监听的网络服务	
`ws_http_port`	13000	HTTP 协议监听端口 （内部使用）	
`ws_h2_port`	13002	HTTP/2 协议监听端口（内部使用）	
`ws_ip`	"127.0.0.1"	web service 绑定地址	

注意： `meta_server_addrs` 参数中推荐使用实际 IP， `127.0.0.1` 有时不会被正确解析。

authorization （安全相关）

属性名	默认值	说明
`enable_authorize`	false	开启身份验证
`auth_type`	password	password：帐密方式；ldap：LDAP 方式；cloud云：端方式

开启身份验证，仅能使用 root 账号登陆，例如：

```
/usr/local/nebula/bin/nebula -u root -p nebula --addr=127.0.0.1 --port=3699
```

未开启身份验证，可不填账号信息或使用任意账号进行登陆，例如：

```
/usr/local/nebula/bin/nebula --addr=127.0.0.1 --port=3699
```

最后更新: 2020年7月28日

2021 Vesoft Inc.

3.3.5 Storage 配置

本文介绍 `storaged` 配置文件。配置文件默认在 `/usr/local/nebula/etc/` 目录下。如果您已指定 **Nebula Graph** 安装路径，则配置文件目录为 `$pwd/nebula/etc/` 。

- `*.default` 文件为**日常调试**使用，也是服务**启动时默认使用**的配置文件
- `*.production` 文件为**推荐生产时**使用的的文件，生产时请去掉 `.production` 后缀

Basics 基础配置

属性名	默认值	说明
`daemonize`	true	作为 daemon 进程运行
`pid_file`	"pids/nebula-storaged.pid"	进程 PID 文件

logging 日志相关

属性名	默认值	说明	动态修改
`log_dir`	logs (也即 `/usr/local/nebula/logs`)	storaged 日志文件存放路径，建议与 data_path 放不同硬盘	
`minloglevel`	0	对应的日志级别分别为 INFO(DEBUG)，WARNING，ERROR，FATAL。通常在调试环境设置为 0，生产环境设置为 1，设置为 4 不打印任何日志。	UPDATE CONFIGS 命令修改，立刻生效
`v`	0	0-4: 当 minloglevel 设置为 0 时，可以进一步设置调试日志的详细程度，值越大越详细	UPDATE CONFIGS 命令修改，立刻生效
`logbufsecs`	0 （秒）	日志缓存时间	UPDATE CONFIGS 命令修改，立刻生效
`slow_op_threshhold_ms`	50 (毫秒)	慢操作和慢查询需要打印日志的阈值	UPDATE CONFIGS 命令修改，立刻生效

例如，下面命令可以增加 storage 的日志详细程度 v=1.

```
nebula> UPDATE CONFIGS storage:v=1;
```


2021 Vesoft Inc.

networking 网络通信相关

属性名	默认值	说明	动态修改
`meta_server_addrs`	"127.0.0.1:45500"	meta server 地址列表，格式为 ip1:port1, ip2:port2, ip3:port3	
`port`	44500	RPC 监听端口	
`reuse_port`	true	开启内核（>3.9) `SO_REUSEPORT` 选项	
`ws_http_port`	12000	HTTP 协议监听端口 （内部使用）	
`ws_h2_port`	12002	HTTP/2 协议监听端口 （内部使用）	
`ws_ip`	"127.0.0.1"	web service 绑定地址	
`heartbeat_interval_secs`	10 (秒)	与 `meta service` 的心跳，需和 `nebula-meta.conf` 文件中的该项**一致**	UPDATE CONFIGS 命令修改，下个心跳周期生效
`raft_heartbeat_interval_secs`	5 (秒)	RAFT 选举通信超时时间	修改配置文件重启
`raft_rpc_timeout_ms`	500 (ms)	RAFT RPC 通信超时	修改配置文件重启

注意： `meta_server_addrs` 参数中推荐使用实际 IP， `127.0.0.1` 有时不会被正确解析。

storage 数据持久化设置

属性名	默认值	说明
`data_path`	data/storage (也即 `/usr/local/nebula/data/storage/`)	本机数据持久化的根目录。 每个 partition 对应一个子目录。

多块硬盘时可以逗号分隔多个目录，每个目录对应一个 Rocksdb 实例，以有更好的并发能力。例如

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

RocksDB Options

属性名	默认值	说明	动态修改
rocksdb_batch_size	4096 (B)	Batch 写	
rocksdb_block_cache	1024 (MB)	block cache 大小。**建议为本机内存的1/3**	
rocksdb_disable_wal	true	使用 Rocksdb 的wal。默认使用 raft wal	
wal_ttl	14400 (秒)	RAFT wal 保留时间	UPDATE CONFIGS 命令修改，立刻生效
rocksdb_db_options	{}	json 类型，其中每个参数 key 和 value 均为 string 格式。解释见下	UPDATE CONFIGS 命令修改，**整个json覆盖**，立刻生效
rocksdb_column_family_options	{}	json 类型，其中每个参数 key 和 value 均为 string 格式。解释见下	UPDATE CONFIGS 命令修改，**整个json覆盖**，立刻生效
rocksdb_block_based_table_options	{}	json 类型，其中每个参数 key 和 value 均为 string 格式。解释见下	UPDATE CONFIGS 命令修改，**整个json覆盖**，立刻生效

ROCKSDB_DB_OPTIONS

2021 Vesoft Inc.

```
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
enable_write_thread_adaptive_yield
max_subcompactions              -- level0 到 level1 的 compact 启用多线程。默认为 1，动态修改重启后生效
max_background_jobs             -- 使用多线程进行 compact。默认为 1，动态修改重启后生效
```

以上参数都可通过 `UPDATE CONFIGS` 语法进行动态修改，也可以写在本地配置文件中。具体作用和修改是否需要重启请参考 RocksDB 手册。

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       -- 停止写入数据时候的自动 compact，默认 false。动态修改立刻生效。
```

以上参数都可通过 UPDATE CONFIGS 语法进行动态修改，也可以写在本地配置文件中。具体作用和修改是否需要重启请参考RocksDB手册。

以上参数可通过命令行如下设置:

```
nebula> UPDATE CONFIGS storage:rocksdb_column_family_options = \
        { disable_auto_compactions = false, level0_file_num_compaction_trigger = 10 };
        -- 该命令会整个覆盖 rocksdb_column_family_options，请注意是否会覆盖其他子项
nebula> UPDATE CONFIGS storage:rocksdb_db_options  = \
        { max_subcompactions = 10, max_background_jobs = 10};
nebula> UPDATE CONFIGS storage:max_edge_returned_per_vertex = 10; -- 该参数解释见下文
```

可在配置文件中如下设置:

```
    rocksdb_db_options = {"stats_dump_period_sec":"200", "write_thread_max_yield_usec":"600"}
    rocksdb_column_family_options = {"max_write_buffer_number":"4", "min_write_buffer_number_to_merge":"2", "max_write_buffer_number_to_maintain":"1"}
    rocksdb_block_based_table_options = {"block_restart_interval":"2"}
```

说明：稠密点处理（出入边数量极多的点）

针对图遍历过程中，遇到稠密点的情况，目前有两种截断策略：

1. 直接截断。设置 `enable_reservoir_sampling = false`，截取前 `max_edge_returned_per_vertex` 个边，多余的边不返回。
2. 蓄水池采样。对出边进行等概率采样，从未知数量的所有出边中，等概率采样 `max_edge_returned_per_vertex` 个边。对于出边数量超过 `max_edge_returned_per_vertex` 的稠密点，蓄水池采样方法的性能相对直接截取方法要差，但对某些业务场景更有应用价值。

例如：

```
nebula> UPDATE CONFIGS storage:enable_reservoir_sampling = false;
```

直接截断方法

属性名	默认值	说明	动态修改
max_edge_returned_per_vertex	2147483647	每个稠密点，最多返回多少条边，多余的边截断不返回	UPDATE CONFIGS 命令修改，立刻生效

2021 Vesoft Inc.

RESERVOIR SAMPLING 蓄水池采样算法方式

属性名	默认值	说明	动态修改
enable_reservoir_sampling	false	对于边采用蓄水池概率采样方式决定是否返回	UPDATE CONFIGS 命令修改，立刻生效

数据量大时的 storage 配置

如果数据量很大但内存不够，则推荐把 storage 配置中的 `enable_partitioned_index_filter` 设置为 `true`；但由于缓存了较少的 RocksDB 索引，性能会受影响。

最后更新: 2020年7月27日

2021 Vesoft Inc.

3.3.6 命令行终端设置

属性名	默认值	说明
addr	"127.0.0.1"	graphd IP
port	0	graphd port
u	""	用于身份验证的用户名
p	""	用于身份验证的密码
enable_history	false	是否保存历史命令
server_conn_timeout_ms	1000	连接超时时长，单位毫秒

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.3.7 操作系统参数配置

本文档为 **Nebula Graph** 系统参数配置参考。

ulimit

ULIMIT -C

此命令调整 core dump 文件可占用空间的最大值。建议设置为 *unlimited*，即：`ulimit -c unlimited`。

ULIMIT -N

此命令调整进程最大可用的文件句柄数。建议设置到 10 万以上，如： `ulimit -n 130000`。

内存

VM.SWAPPINESS

内核将一段时间内 idle 的匿名页交换至磁盘的*倾向性*。值越大，越倾向于换出。建议设置为 0，即在内存紧张时，优先淘汰 page cache。注意，swappiness 为 0，不代表不会换出。

VM.MIN_FREE_KBYTES

Linux 内核根据该选项计算内存相关水位值。如果系统物理内存较大，建议调高该值（比如，128GB 物理内存设置为 5GB）。否则，在内存紧张时，系统无法申请较大的连续物理内存。

VM.MAX_MAP_COUNT

此选项指定进程允许的最大 vma (virtual memory area) 数量。系统默认值通常为 65530，绝大多数情况是足够的。如果在内存消耗较大时出现内存申请失败，可以适当调高。

VM.OVERCOMMIT_MEMORY

该选项指定系统中申请的内存大于可用内存时的内核行为，建议使用系统默认值 0 或 1。不要设置为 2。

VM.DIRTY_*

这些选项控制系统脏页（dirty page cache）落盘的激进程度。对于写入密集的场景，可以根据需要（吞吐优先还是延时优先）做适当调整。建议使用系统默认值。

TRANSPARENT HUGE PAGE

为了更好的延时表现，建议将内存透明大页及锁片整理禁掉。选项分别为 `/sys/kernel/mm/transparent_hugepage/enabled` 和 `/sys/kernel/mm/transparent_hugepage/defrag`。例如：

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
swapoff -a && swapon -a
```

网络

NET.IPV4.TCP_SLOW_START_AFTER_IDLE

此选项默认值为 1，即 TCP 连接在 Idle 一段时间之后，发送窗口会被重设，并重新执行慢开始算法。建议将该选项设置为 0，尤其是长肥链路（即高延时大带宽）。

NET.CORE.SOMAXCONN

该选项与 listen 系统调用的 backlog 参数一起，指定“已完成三次握手但未被 accept”的最大 TCP 连接数。对于有大量突发连接建立的场景，建议设置为 1024 以上（默认值为 128）。

NET.IPV4.TCP_MAX_SYN_BACKLOG

最大 TCP 半连接数，设置原则同 `net.core.somaxconn`。

2021 Vesoft Inc.

NET.CORE.NETDEV_MAX_BACKLOG

网卡接受队列的最大长度。对于高吞吐场景，尤其是 10G 网卡，建议设置为 10000 以上（默认值为 1000）。

NET.IPV4.TCP_KEEPALIVE_*

TCP 连接的 keepalive 机制相关参数。对于使用 4 层透明 load balancer 的应用，如果出现 idle 连接意外断开的情况，可以将 `tcp_keepalive_time` 与 `tcp_keepalive_intvl` 调低。

NET.IPV4.TCP_RMEM/WMEM

TCP 连接的最小、默认、最大接受/发送 buffer 的大小。对于长肥链路，建议调大默认值为 bandwidth * RTT。

SCHEDULER

对于 SSD 设备，建议将 `/sys/block/DEV_NAME/queue/scheduler` 设置为 noop 或 none。

其他

KERNEL.CORE_PATTERN

建议设置为 core，并将 `kernel.core_uses_pid` 设置为 1。

相关命令使用说明

SYSCTL

- `sysctl conf_name` 查看当前参数值
- `sysctl -w conf_name=value` 修改参数值并立即生效
- `sysctl -p` 从相关配置文件加载参数值

ULIMIT 使用说明

`ulimit` 命令用于设置当前 shell session 的资源限制阈值。需要注意的是，

- `ulimit` 命令做出的修改仅对当前 session （及子进程）有效。
- `ulimit` 不能将某个资源的（soft）阈值调整至大于当前 hard 值。
- 普通用户不能通过该命令调整 hard 阈值（包括 `sudo` 方式）。
- 若要从系统层面修改，或者调整 hard 阈值，需要编辑 `/etc/security/limits.conf` 文件。但该方式需要重新登录才能生效。

PRLIMIT

`prlimit` 命令可以修改某个特定进程的资源限制阈值。结合 `sudo` 命令可以修改 hard 阈值。例如， `prlimit --nofile=130000 --pid=$$` 可将当前进程最大允许打开的文件数调整至 14000，并立即生效。请注意该命令仅在 RedHat 7u 或更新版本 OS 可用。

最后更新: 2020年6月30日

2021 Vesoft Inc.

3.3.8 单机日志

Nebula Graph 使用 glog 打印日志，使用 gflag 控制日志级别，并提供 HTTP 接口在运行时动态改变日志级别，以方便追踪问题。

日志位置

日志默认存放在 `/usr/local/nebula/logs/` 下。

注意： 如果在运行时删除了日志目录，会导致运行时的日志不继续输出，但不会影响服务。程序重启后可恢复正常。

参数说明

GLOG 中的两个主要参数

- minloglevel：范围为 0-4。其中 0-3 对应的日志级别分别为 INFO(DEBUG)、WARNING、ERROR、FATAL。通常在调试环境设置为 0，生产环境设置为 1，设置为 4 不打印任何日志。
- v：范围为 0-3。当 minloglevel 设置为 0 时，可以进一步设置调试日志的详细程度，值越大越详细。

配置文件

在配置文件中（通常在 `/usr/local/nebula/etc/` 下）可以找到 metad、graphd 和 storaged 的默认日志配置级别。

动态查看和修改日志级别

通过如下命令来查看当前所有的 gflags 参数（包括日志参数）。curl 仅在 `local_config` 设置为 `true` 时可用。

```
> curl ${ws_ip}:${ws_port}/get_flags
```

其中，

- `ws_ip` 为 HTTP 服务的 IP，可以在上述配置文件中找到。默认 IP 为 127.0.0.1。
- `ws_port` 为 HTTP 服务的端口号。`metad` 默认为 11000， `storaged` 默认为 12000， `graphd` 默认为 13000。

例如，查看 `storaged` 服务的 minloglevel 级别：

```
> curl 127.0.0.1:12000/get_flags | grep minloglevel  # storage
> curl 127.0.0.1:13000/get_flags                     # metad
```

也可以通过如下命令将日志级别更改为**最详细**。

```
> curl "http://127.0.0.1:12000/set_flags?flag=v&value=3"
> curl "http://127.0.0.1:12000/set_flags?flag=minloglevel&value=0"
```

在 console 中，使用如下命令获取当前日志级别并将日志级别设置为**最详细**。

```
nebula> GET CONFIGS graph:minloglevel;
nebula> UPDATE CONFIGS graph:minloglevel=0;
```

如需更改 storage 日志级别，将上述命令中的 `graph` 更换为 `storage` 即可。

注意： **Nebula Graph** 仅支持通过 console 修改 graph 和 storage 日志级别，meta 日志级别必须使用 curl 命令更改。

使用以下命令**关闭**所有的日志打印(仅保留 FATAL)。

```
> curl "http://127.0.0.1:12000/set_flags?flag=minloglevel&value=3"
```

最后更新: 2020年7月27日

2021 Vesoft Inc.

3.4 账号权限管理

3.4.1 ALTER USER 语法

```
ALTER USER <user_name> WITH PASSWORD <password>
```

使用 `ALTER USER` 语句修改 **Nebula Graph** 帐户。使用 `ALTER USER` 必须拥有全局的 `CREATE USER` 权限。尝试修改一个不存在的用户会发生错误。`ALTER` 无需密码校验。

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.4.2 身份验证

当客户端连接到 **Nebula Graph** 时，**Nebula Graph** 就会创建一个会话。会话用来存储有关连接的各种信息。每条会话只关联一个用户。身份验证即为将会话映射到特定用户的过程。一旦将会话映射到用户，便可以使用授权将一组权限与之关联。

Nebula Graph 支持两种身份验证方式，即本地和 LDAP。详细说明见下文。

本地身份验证

本地数据库存储用户名，加密密码，本地用户设置和远程 LDAP 用户设置。当用户尝试访问数据库时，将接受安全验证。

将 `nebula-graphd.conf` （默认目录 `/usr/local/nebula/etc/` ）文件中的 `--enable_authorize` 属性设置为 `true` 即可启用本地认证。

LDAP

轻型目录访问协议（LDAP）是用于访问目录服务的轻型客户端-服务器协议。 LDAP 中存储的用户优先级高于本地数据库用户。例如，如果两个本地和 LDAP 都有一个名为 “Amber” 的用户，则优先从 LDAP 读取该用户的设置和角色信息。

与本地身份验证不同，除需设置 `--enable_authorize` 参数之外， LDAP 还需要在 `nebula-graphd.conf` 文件中（默认目录 `/usr/local/nebula/etc/` ）配置相关参数方可启用。详细信息，请参见集成 LDAP 文档。

LDAP 参数

参数名	参数类型	默认值	说明
ldap_server	string	""	一系列 ldap 服务器地址。多个地址使用逗号隔开。
ldap_port	INT32	LDAP 服务器端口号。如未指定，则使用默认端口。	
ldap_scheme	string	"ldap"	仅支持 ldap。
ldap_tls	bool	false	开启/禁用 graphd 和 ldap 之间的 TLS 加密方式。
ldap_suffix	string	""	指定用于所有 ldap 操作的根后缀。
ldap_basedn	string	""	LDAP 的专有名称。
ldap_binddn	string	""	允许搜索基本 DN 的 LDAP 用户。
ldap_bindpasswd	string	""	绑定到 DN 的用户密码。
ldap_searchattribute	string	""	一系列必填属性。
ldap_searchfilter	string	""	定义搜索过滤规则。较 searchattribut 更为灵活。

|

最后更新: 2020年7月22日

2021 Vesoft Inc.

3.4.3 Built-in Roles

Nebula Graph 角色可分为以下几类：

- God
  - 初始 Root 用户（类似于 Linux 系统中的 Root，和 Windows 系统中的 Administrator）。
  - 拥有所有操作权限。
  - 一个集群只能有一个 God。God 可管理集群内所有 space。
  - Meta 服务在初始化时会默认创建一个 GOD 角色的 Account，名为 root。
  - God 角色由 meta 自动初始化，且不支持用户自行授权成为 God。
- Admin
  - 管理员用户。
  - 对权限内的 space 拥有 schema 和 data 的读/写权限。
  - 可对权限内的 space 进行用户授权。
- DBA
  - 对权限内的 space 拥有 schema 和 data 的读/写权限。
  - 没有对用户受权的权限。
- User
  - 对权限内的 space 拥有 data 的读/写权限。
  - 对权限内的 space 拥有 schema 只读权限。
- Guest
  - 对权限内的 space 拥有 schema 和 data 的只读权限。

如果开启用户权限开关，则默认用户名为 root，默认密码为 nebula，且用户名不可更改。将 `/usr/local/nebula/etc/nebula-graphd.conf` 文件中的 `enable_authorize` 设置为 `true` 即可打开权限开关。

未被分配角色的用户将无权访问该 space。一个用户在同一个 space 中只能分配一个角色。一个用户在不同 space 可拥有不同权限。

各角色的 Executor 权限见下表。

按操作权限划分。

OPERATION	STATEMENTS
Read space	Use, DescribeSpace
Write space	CreateSpace, DropSpace, CreateSnapshot, DropSnapshot, Balance, Admin, Config, Ingest, Download
Read schema	DescribeTag, DescribeEdge, DescribeTagIndex, DescribeEdgeIndex
Write schema	CreateTag, AlterTag, CreateEdge, AlterEdge, DropTag, DropEdge, CreateTagIndex, CreateEdgeIndex, DropTagIndex, DropEdgeIndex
Write user	CreateUser, DropUser, AlterUser
Write role	Grant, Revoke
Read data	Go, Set, Pipe, Match, Assignment, Lookup, Yield, OrderBy, FetchVertices, Find, FetchEdges, FindPath, Limit, GroupBy, Return
Write data	BuildTagIndex, BuildEdgeIndex, InsertVertex, UpdateVertex, InsertEdge, UpdateEdge, DeleteVertex, DeleteEdges
Special operation	Show, ChangePassword

2021 Vesoft Inc.

按操作划分。

OPERATION	GOD	ADMIN	DBA	USER	GUEST
Read space	Y	Y	Y	Y	Y
Write space	Y				
Read schema	Y	Y	Y	Y	Y
Write schema	Y	Y	Y		
Write user	Y				
Write role	Y	Y			
Read data	Y	Y	Y	Y	Y
Write data	Y	Y	Y	Y	
Special operation	Y	Y	Y	Y	Y
Show Jobs	Y	Y	Y	Y	Y

注意： Special Operation 为特殊操作，例如 `SHOW SPACE`，每个角色都可以执行，但其执行结果只显示 Account 权限内的结果。

最后更新: 2020年6月17日

2021 Vesoft Inc.

3.4.4 CHANGE PASSWORD 语法

```
CHANGE PASSWORD <user_name> FROM <old_psw> TO <new-psw>
```

`CHANGE PASSWORD` 更改 **Nebula Graph** 用户账户密码。更改密码需同时提供新密码和旧密码。

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.4.5 CREATE USER 语法

```
CREATE USER [IF NOT EXISTS] <user_name> [WITH PASSWORD <password>]
```

使用 `CREATE USER` 语句创建新的 **Nebula Graph** 帐户。使用 `CREATE USER` 必须拥有全局的 `CREATE USER` 权限。默认情况下，尝试创建一个已经存在的用户会发生错误。如果使用 `IF NOT EXISTS` 子句，则会提示用户名已存在。

例如：

```
nebula> CREATE USER user1 WITH PASSWORD 'nebula';
```

最后更新: 2020年6月23日

2021 Vesoft Inc.

3.4.6 DROP USER 语法

```
DROP USER [IF EXISTS] <user_name>
```

只有 `God` 和 `Admin` 用户有使用 DROP 语句的权限。 `DROP USER` 不会自动关闭任何已打开的客户端连接。

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.4.7 GRANT ROLE 语法

```
GRANT ROLE <role_type> ON <space> TO <user>
```

使用 `GRANT` 语句为 **Nebula Graph** 用户授予权限。使用 `GRANT` 必须拥有 `GRANT` 权限。

目前 **Nebula Graph** 包含五种角色权限：`GOD`、`ADMIN`、`DBA`、`USER` 和 `GUEST`。

通常，需要先使用 `CREATE USER` 创建帐户，然后再使用 `GRANT` 为其授予权限（假设你拥有 `CREATE` 和 `GRANT` 权限）。待授予的角色以及用户帐户必须存在，否则会发生错误。

`<space>` 必须指定为存在的图空间，否则会发生错误。

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.4.8 集成 LDAP

本文档介绍如何将 **Nebula Graph** 连接到 LDAP 服务器以进行身份验证（仅企业版可用）。

LDAP 介绍

集成 LDAP 后， **Nebula Graph** 即可共享在 LDAP 中定义的用户身份信息和密码。

安装 LDAP 插件

1. 构建 LDAP 服务器并插入相应的记录。

   例如，插入用户名 `test2`，密码 `passwdtest2`。然后使用以下命令查看用户：

   ```
   ldapsearch -x -b 'uid=test2,ou=it,dc=sys,dc=com'
   ```

2. 将 `auth_ldap.so` 文件放在安装路径的共享目录中。
3. 创建影子账户安装 `auth_ldap` 插件。

   以 root 身份登录 **Nebula Graph**， 密码为 `nebula`，身份验证 `auth_type` 类型为 `password`：

   ```
   ./bin/nebula -u root  -p nebula   --port 3699 --addr="127.0.0.1"
   ```

   创建影子账户：

   ```
   # 此处需要为影子账户 test2 赋权
   nebula> CREATE USER test2 WITH PASSWORD "";
   ```

   安装 `auth_ldap` 插件：

   ```
   nebula> INSTALL PLUGIN auth_ldap SONMAE "auth_ldap.so";
   ```

   查看插件是否成功安装：

   ```
   nebula> SHOW PLUGINS;
   ```

卸载 LDAP 插件

1. 以 root 身份登录 **Nebula Graph**， 密码为 `nebula`，身份验证 `auth_type` 类型为 `password`：

   ```
   ./bin/nebula -u root  -p nebula   --port 3699 --addr="127.0.0.1"
   ```

2. 执行以下命令卸载 `auth_ldap` 插件：

   ```
   nebula> UNINSTALL PLUGIN auth_ldap;
   ```

启用 LDAP 身份验证

Nebula Graph 默认禁用 LDAP 身份验证。执行以下操作启用 LDAP 身份验证：

开启身份验证开关

首先，开启身份验证开关。打开 `nebula-graphd.conf` 文件（默认目录为 `/usr/local/nebula/etc/`），然后找到 `--enable_authorize`，将其更改为 `true`：

```
########## Authorization ##########
# Enable authorization
--enable_authorize=true
```


2021 Vesoft Inc.

配置 NEBULA GRAPH

然后配置 **Nebula Graph** 以使用 LDAP。您可以使用两种 LDAP 方法来为 **Nebula Graph** 开启身份验证。

- **简单绑定**。打开 `nebula-graphd.conf` 文件（默认目录为 `/usr/local/nebula/etc/` ），找到 `Authentication` 部分，将 `auth_type` 更改为 `ldap` 并添加以下属性：

```
########## Authentication ##########
# User login authentication type, password for nebula authentication, ldap for ldap authentication, cloud for cloud authentication
--auth_type=ldap
--ldap_server=127.0.0.1
--ldap_port=389
--ldap_scheme=ldap
--ldap_prefix=uid=
--ldap_suffix=,ou=it,dc=sys,dc=com
```

- **搜索和绑定**. 打开 `nebula-graphd.conf` 文件（默认目录为 `/usr/local/nebula/etc/` ），找到 `Authentication` 部分，将 `auth_type` 更改为 `ldap` 并添加以下属性：

```
########## Authentication ##########
# User login authentication type, password for nebula authentication, ldap for ldap authentication, cloud for cloud authentication
--auth_type=ldap
--ldap_server=127.0.0.1
--ldap_port=389
--ldap_scheme=ldap
--ldap_prefix=uid=
--ldap_suffix=,ou=it,dc=sys,dc=com
```

重启服务

保存并关闭配置文件，然后重启服务：

```
/usr/local/nebula/scripts/nebula.service restart all
```

禁用 LDAP

将 `nebula-graphd.conf` 中的 `--enable_authorize` 参数设置为 `false` 然后重启服务即可在 **Nebula Graph** 中禁用 LDAP 身份验证。

使用 LDAP 连接 Nebula Graph

配置完成后，您可以使用 LDAP 身份验证连接到 **Nebula Graph**，运行以下命令：

```
./bin/nebula -u test2 -p passwdtest2 --port 3699 --addr="127.0.0.1"
```

最后更新: 2020年7月20日

2021 Vesoft Inc.

3.4.9 REVOKE 语法

```
REVOKE ROLE <role_type> ON <space> FROM <user>
```

使用 `REVOKE` 语句从 **Nebula Graph** 用户删除权限。使用 `REVOKE` 必须拥有 `REVOKE` 权限。

目前 **Nebula Graph** 包含五种角色权限：`GOD`、`ADMIN`、`DBA`、`USER` 和 `GUEST`。

待删除的角色以及用户帐户必须存在，否则会发生错误。

`<space>` 必须指定为存在的图空间，否则会发生错误。

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.5 批量数据管理

3.5.1 离线数据导入

Spark Writer

概述

Spark Writer 是 Nebula Graph 基于 Spark 的分布式数据导入工具，能够将多种数据仓库中的数据转化为图的点和边，并批量导入到图数据库中。目前支持的数据仓库有：

- HDFS，包括 Parquet、JSON、ORC 和 CSV 格式的文件
- HIVE

Spark Writer 支持并发导入多个 tag、edge，支持不同 tag/edge 配置不同的数据仓库。

软件要求

注意： 为确保 **Nebula Graph Spark Writer** 正常使用，请确保你的机器已安装：

- Spark 2.0 及以上版本
- Hive 2.3 及以上版本
- Hadoop 2.0 及以上版本

获取 SPARK WRITER

编译源码

```
git clone https://github.com/vesoft-inc/nebula.git
cd nebula/src/tools/spark-sstfile-generator
mvn compile package
```

或者直接下载

从云存储 OSS 下载

```
wget https://oss-cdn.nebula-graph.com.cn/jar-packages/sst.generator-1.2.1.jar
```

使用流程

基本流程分为以下几步：

1. 在 Nebula Graph 中创建图模型，构图
2. 编写数据文件
3. 编写输入源映射文件
4. 导入数据

构图

构图请参考快速试用中的示例构图。

注意：请先在 Nebula Graph 中完成构图（创建图空间和定义图数据 Schema），再通过本工具向 Nebula Graph 中写入数据。

数据示例

点

2021 Vesoft Inc.

顶点数据文件由一行一行的数据组成，文件中每一行表示一个点和它的属性。一般来说，第一列为点的 ID ——此列的名称将在后文的映射文件中指定，其他列为点的属性。

- **player** 顶点数据

```
{"id":100,"name":"Tim Duncan","age":42}
{"id":101,"name":"Tony Parker","age":36}
{"id":102,"name":"LaMarcus Aldridge","age":33}
```

边

边数据文件由一行一行的数据组成，文件中每一行表示一条边和它的属性。一般来说，第一列为起点 ID，第二列为终点 ID，起点 ID 列及终点 ID 列会在映射文件中指定。其他列为边属性。下面以 JSON 格式为例进行说明。

以边 边 ***follow*** 的数据为例：

- 无 rank 的边

```
{"source":100,"target":101,"likeness":95}
{"source":101,"target":100,"likeness":95}
{"source":101,"target":102,"likeness":90}
```

- 有 rank 的边

```
{"source":100,"target":101,"likeness":95,"ranking":2}
{"source":101,"target":100,"likeness":95,"ranking":1}
{"source":101,"target":102,"likeness":90,"ranking":3}
```

含有地理位置 Geo 的数据

Spark Writer 支持 Geo 数据导入，Geo 数据用 **latitude** 与 **longitude** 字段描述经纬度，数据类型为 double。

```
{"latitude":30.2822095,"longitude":120.0298785,"target":0,"dp_poi_name":"0"}
{"latitude":30.2813834,"longitude":120.0208692,"target":1,"dp_poi_name":"1"}
{"latitude":30.2807347,"longitude":120.0181162,"target":2,"dp_poi_name":"2"}
{"latitude":30.2812694,"longitude":120.0164896,"target":3,"dp_poi_name":"3"}
```

数据源文件

目前 Spark Writer 支持的数据源有：

- HDFS
- HIVE

HDFS 文件

支持的文件格式包括：

- Parquet
- JSON
- CSV
- ORC

Player 的 Parquet 示例如下：

```
+---+---+------------+
|age| id|        name|
+---+---+------------+
| 42|100| Tim Duncan |
| 36|101| Tony Parker|
+---+---+------------+
```

JSON 示例如下：

```
{"id":100,"name":"Tim Duncan","age":42}
{"id":101,"name":"Tony Parker","age":36}
```


2021 Vesoft Inc.

CSV 示例如下：

```
age,id,name
42,100,Tim Duncan
36,101,Tony Parker
```

数据库

Spark Writer 支持以数据库作为数据源，目前支持 HIVE。

Player 表结构如下：

col_name	data_type	comment
id	int	
name	string	
age	int	

编写配置文件

配置文件由 Spark 相关信息，Nebula 相关信息，以及 tags 映射 和 edges 映射块组成。Spark 信息配置了 Spark 运行的相关参数，Nebula 相关信息配置了连接 Nebula Graph 的用户名和密码等信息。 tags 映射和 edges 映射分别对应多个 tag/edge 的输入源映射，描述每个 tag/edge 的数据源等基本信息，不同 tag/edge 可以来自不同数据源。

输入源的映射文件示例：

```
{
  # Spark 相关信息配置
  # 参见： http://spark.apache.org/docs/latest/configuration.html
  spark: {
    app: {
      name: Spark Writer
    }

    driver: {
      cores: 1
      maxResultSize: 1G
    }

    cores {
      max: 16
    }
  }

  # Nebula Graph 相关信息配置
  nebula: {
    # 查询引擎 IP 列表
    addresses: ["127.0.0.1:3699"]

    # 连接 Nebula Graph 服务的用户名和密码
    user: user
    pswd: password

    # Nebula Graph 图空间名称
    space: test

    # thrift 超时时长及重试次数
    # 如未设置，则默认值分别为 3000 和 3
    connection {
      timeout: 3000
      retry: 3
    }

  # nGQL 查询重试次数
  # 如未设置，则默认值为 3
    execution {
      retry: 3
    }
  }

  # 处理标签
  tags: [

    # 从 HDFS 文件加载数据， 此处数据类型为 Parquet
    # tag 名称为 tag name 0
    #  HDFS Parquet 文件的中的 field_0、field_1、field_2 将写入 tag_name_0
    # 节点列为 vertex_key_field
    {
      name: tag_name_0
      type: parquet
```


2021 Vesoft Inc.

```
      path: hdfs path
      fields: {
        field_0: nebula_field_0,
        field_1: nebula_field_1,
        field_2: nebula_field_2
      }
      vertex: vertex_key_field
      batch : 16
    }

    # 与上述类似
    # 从 Hive 加载将执行命令 $ {exec} 作为数据集
    {
      name: tag_name_1
      type: hive
      exec: "select hive_field_0, hive_field_1, hive_field_2 from database.table"
      fields: {
        hive_field_0: nebula_field_0,
        hive_field_1: nebula_field_1,
        hive_field_2: nebula_field_2
      }
      vertex: vertex_id_field
    }
  ]

  # 处理边
  edges: [
    # 从 HDFS 加载数据，数据类型为 JSON
    # 边名称为 edge_name_0
    # HDFS JSON 文件中的 field_0、field_1、field 2 将被写入 edge_name_0
    # 起始字段为 source_field，终止字段为 target_field ，边权重字段为 ranking_field。
    {
      name: edge_name_0
      type: json
      path: hdfs_path
      fields: {
        field_0: nebula_field_0,
        field_1: nebula_field_1,
        field_2: nebula_field_2
      }
      source:  source_field
      target:  target_field
      ranking: ranking_field
    }

    # 从 Hive 加载将执行命令 $ {exec} 作为数据集
    # 边权重为可选
    {
      name: edge_name_1
      type: hive
      exec: "select hive_field_0, hive_field_1, hive_field_2 from database.table"
      fields: {
        hive_field_0: nebula_field_0,
        hive_field_1: nebula_field_1,
        hive_field_2: nebula_field_2
      }
      source:  source_id_field
      target:  target_id_field
    }
  ]
}
```

Spark 配置信息

下表给出了一些示例，所有可配置项请见 Spark Available Properties。

字段	默认值	是否必须	说明
spark.app.name	Spark Writer	否	app 名称
spark.driver.cores	1	否	驱动程序进程的核数，仅适用于群集模式
spark.driver.maxResultSize	1G	否	每个 Spark 操作（例如收集）中所有分区的序列化结果的上限（以字节为单位）。至少应为 1M，否则应为 0（无限制）
spark.cores.max	(not set)	否	当以“粗粒度”共享模式在独立部署群集或 Mesos 群集上运行时，跨群集（而非从每台计算机）请求应用程序的最大 CPU 核数。如果未设置，则默认值为 Spark 的独立集群管理器上的 `spark.deploy.defaultCores` 或 Mesos 上的 infinite（所有可用的内核）

2021 Vesoft Inc.

Nebula Graph 配置信息

字段	默认值	是否必须	说明
nebula.addresses	无	是	查询引擎的地址列表，逗号分隔
nebula.user	无	是	数据库用户名，默认为 user
nebula.pswd	无	是	数据库用户名对应密码，默认 user 密码为 password
nebula.space	无	是	导入数据对应的 space，本例中为 test
nebula.connection.timeout	3000	否	Thrift 连接超时时间
nebula.connection.retry	3	否	Thrift 连接重试次数
nebula.execution.retry	3	否	nGQL 语句执行重试次数

tags 和 edges 映射信息

tag 和 edge 映射的选项比较类似。下面先介绍相同的选项，再分别介绍 `tag 映射` 和 `edge 映射` 的特有选项。

- **相同的选项**
  - `type` 指定上文中提到的数据类型，目前支持 “Parquet”、"JSON"、"ORC" 和 “CSV”，大小写不敏感，必填
  - `path` 适用于 HDFS 数据源，指定HDFS 文件或目录的绝对路径，type 为 HDFS 时，必填
  - `exec` 适用于 Hive 数据源， 当执行查询语句 type 为 HIVE 时，必填
  - `fields` 将输入源列的列名映射为 tag / edge 的属性名，必填
- **tag 映射的特有选项**
  - `vertex` 指定某一列作为点的 ID 列，必填
- **edge 映射的特有选项**
  - `source` 指定输入源某一列作为**源点**的 ID 列，必填
  - `target` 指定某一列作为**目标点**的 ID 列，必填
  - 当插入边有 ranking 值， `ranking` 指定某一列作为边 ranking 列，选填

数据源映射

- **HDFS Parquet 文件**
  - `type` 指定输入源类型，当为 parquet 时大小写不敏感，必填
  - `path` 指定 HDFS 文件或目录的路径，必须是 HDFS 的绝对路径，必填
- **HDFS JSON 文件**
  - `type` 指定输入源类型，当为 JSON 时大小写不敏感，必填
  - `path` 指定 HDFS 文件或目录的路径，必须是 HDFS 的绝对路径，必填
- **HIVE ORC 文件**
  - `type` 指定输入源类型，当为 ORC时大小写不敏感，必填
  - `path` 指定 HDFS 文件或目录的路径，必须是 HDFS 的绝对路径，必填
- **HIVE CSV 文件**
  - `type` 指定输入源类型，当为 CSV 时大小写不敏感，必填
  - `path` 指定 HDFS 文件或目录的路径，必须是 HDFS 的绝对路径，必填
- **HIVE**
  - `type` 指定输入源类型，当为 HIVE 时大小写不敏感，必填
  - `exec` 指定 HIVE 执行查询的语句，必填

2021 Vesoft Inc.

执行命令导入数据

导入数据命令：

```
bin/spark-submit \
 --class com.vesoft.nebula.tools.generator.v2.SparkClientGenerator \
 --master ${MASTER-URL} \
 ${SPARK_WRITER_JAR_PACKAGE} -c conf/test.conf -h -d
```

参数说明：

Abbreviation	Required	Default	Description	示例
--class	yes		指定程序主类	
--master	yes		指定spark cluster master url，请参见 master-urls	e.g. `spark://23.195.26.187:7077`
-c / --config	yes		上文所编写的配置文件路径	
-h / --hive	no	false	用于指定是否支持 Hive	
-d / --directly	no	false	true 为客户端方式插入； false 为 sst 方式导入 (TODO)	
-D / --dry	no	false	检查配置文件是否正确	

性能测试结果

三台物理机 (56 核，250G 内存，万兆网，SSD），写 1 亿条数据（每条数据三个字段，每个 batch 64 条记录），用时 4 分钟（40万条/秒）。

最后更新: 2021年4月15日

2021 Vesoft Inc.

3.5.2 离线数据转储

Dump Tool

Dump Tool 是一个单机离线数据导出工具，可以用于导出或统计指定条件的数据。

如何获得

Dump Tool 源码位于 `nebula/src/tools/db_dump` 下，用户可以执行 `make db_dump` 命令来编译生成该工具。在使用本工具前，你可以使用 **Nebula Graph** CLI 的 SHOW HOSTS 命令查看分区的分布。使用 `vertex_id % partition_num` 来计算点对应的 key 位于哪个分区。

注意： Dump Tool 位于 rpm 包中，目录是 `nebula/bin/` 。该工具通过直接打开 RocksDB 转储数据，因此需要离线使用，关闭该 storaged 进程，并同时保持 meta_server 已启动。具体用法请参考下方说明。

如何使用

具体用法如下所示，用户可以通过执行不带参数的 `db_dump` 命令获得帮助。其中 `space` 参数是必须的，而 `db_path` 以及 `meta_server` 具有默认值，用户可以按照实际配置。 `vids` 、 `parts` 、 `tags` 、 `edges` 可以任意组合，导出你需要的数据。

```
  ./db_dump --space=<space name>

required:
       --space=<space name>
        # A space name must be given.

optional:
       --db_path=<path to rocksdb>
         # Path to the RocksDB data directory. If nebula was installed in `/usr/local/nebula`,
         # the db_path would be /usr/local/nebula/data/storage/nebula/
         # Default: ./

       --meta_server=<ip:port,...>
         # A list of meta severs' ip:port separated by comma.
         # Default: 127.0.0.1:45500

       --mode= scan | stat
         # scan: print to screen when records meet the condition, and also print statistics to screen in final.
         # stat: print statistics to screen.
         # Default: scan

       --vids=<list of vid>
         # A list of vids separated by comma. This parameter means vertex_id/edge_src_id
         # Would scan the whole space's records if it is not given.

       --parts=<list of partition id>
         # A list of partition ids separated by comma.
         # Would output all partitions if it is not given.

       --tags=<list of tag name>
         # A list of tag name separated by comma.

       --edges=<list of edge name>
         # A list of edge name separated by comma.

       --limit=<N>
         # A positive number that limits the output.
         # Would output all if set to 0 or negative.
         # Default: 1000
```

下面是一些示例：

```
// 指定 space 导出数据
./db_dump --space=space_name

// 指定space, db_path, meta_server
./db_dump --space=space_name --db_path=/usr/local/nebula/data/storage/nebula/ --meta_server=127.0.0.1:45513

// 指定 mode=stat(统计模式)，此时只返回统计信息，不打印数据
./db_dump --space=space_name --mode=stat --db_path=/usr/local/nebula/data/storage/nebula/ --meta_server=127.0.0.1:45513

// 指定 vid 导出该点以及以该点为起始点的边
./db_dump --space=space_name --mode=stat --db_path=/usr/local/nebula/data/storage/nebula/ --meta_server=127.0.0.1:45513 --vids=123,456

// 指定 tag 类型，导出具有该 tag 的点
./db_dump --space=space_name --mode=stat --db_path=/usr/local/nebula/data/storage/nebula/ --meta_server=127.0.0.1:45513 --tags=tag1,tag2
```

返回的数据格式：

2021 Vesoft Inc.

```
// 点, key: part_id, vertex_id, tag_name, value: <prop_list>
[vertex] key: 1, 0, poi value:mid:0,8191765721868409651,8025713627522363385,1993089399535188613,3926276052777355165,5123607763506443893,2990089379644866415,poi_name_0,上海,华东,
30.2824,120.016,poi_stat_0,poi_fc_0,poi_sc_0,0,poi_star_0,

// 边, key: part_id, src_id, edge_name, rank, dst_id, value: <prop_list>
[edge] key: 1, 0, consume_poi_reverse, 0, 656384 value:mid:656384,mid:0,7.19312,mid:656384,3897457441682646732,mun:656384,4038264117233984707,dun:656384,empe:656384,mobile:656384,gender:
656384,age:656384,rs:656384,fpd:656384,0.75313,1.34433,fpd:656384,0.03567,7.56212,

// 统计
==================================================
COUNT: 10           #本次导出记录数量
VERTEX COUNT: 1     #本次导出点数量
EDGE COUNT: 9       #本次导出边数量
TAG STATISTICS:     #本次导出tag统计
        poi : 1
EDGE STATISTICS:    #本次导出edge统计
        consume_poi_reverse : 9
==================================================
```

最后更新: 2020年6月23日

2021 Vesoft Inc.

3.5.3 存储服务的负载均衡和数据迁移

Nebula Graph 的服务可分为 graphd，storaged，metad。此文档中的 balance 仅针对 storaged 进行操作。目前，storaged 的扩缩容是通过 balance 命令来实现的。balance 命令有两种，一种需要迁移数据，命令为 **BALANCE DATA**；另一种不需要迁移数据，只改变 partition 的 leader 分布，来达到负载均衡的目的，命令为 **BALANCE LEADER**。

Balance data

以下举例说明 `BALANCE DATA` 的使用方式。本例将集群从 3 个实例（进程）扩展到 8 个实例（进程）：

STEP 1 准备

部署一个三副本的集群，1个 graphd，1个 metad，3个 storaged（具体部署方式请参考集群部署文档），通过 **SHOW HOSTS** 命令可以看到集群的状态信息：

Step 1.1

```
nebula> SHOW HOSTS;
=================================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=================================================================================================
| 192.168.8.210 | 34600 | online | 0            | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 0            | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 0            | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
Got 3 rows (Time spent: 5886/6835 us)
```

`SHOW HOSTS` 返回结果说明：

- IP, Port 表示当前的 storage 实例. 这个集群启动了 3 个 storaged 服务，并且没有任何数据。(192.168.8.210:34600，192.168.8.210:34700，192.168.8.210:34500)
- Status 表示当前实例的状态，目前有 online/offline 两种。当机器下线以后（metad 在一段间隔内收不到其心跳），将把其更改为 offline。这个时间间隔可以在启动 metad 的时候通过设置 `expired_threshold_sec` 来修改，当前默认值是 10 分钟。
- Leader count：表示当前实例 Raft leader 数目。
- Leader distribution：表示当前 leader 在每个图空间上的分布，目前尚未创建任何图空间。
- Partition distribution：不同 space 中 partition 的数目。

Step 1.2

创建一个名为 test 的图空间，包含 100 个 partition 和 3 个 replica。

```
nebula> CREATE SPACE test(PARTITION_NUM=100, REPLICA_FACTOR=3);
```

片刻后，使用 `SHOW HOSTS` 命令显示集群的分布。

```
nebula> SHOW HOSTS;
=================================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=================================================================================================
| 192.168.8.210 | 34600 | online | 0            | test: 0             | test: 100              |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 52           | test: 52            | test: 100              |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 48           | test: 48            | test: 100              |
-------------------------------------------------------------------------------------------------
```

STEP 2 上线新机器

启动 5 个新 storaged 进程进行扩容。启动完毕后，使用 `SHOW HOSTS` 命令查看新的状态：

```
nebula> SHOW HOSTS;
=================================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
=================================================================================================
| 192.168.8.210 | 34600 | online | 0            | test: 0             | test: 100              |
-------------------------------------------------------------------------------------------------
```


2021 Vesoft Inc.

```
| 192.168.8.210 | 34900 | online | 0           | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 35940 | online | 0           | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34920 | online | 0           | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 44920 | online | 0           | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 52          | test: 52            | test: 100              |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 48          | test: 48            | test: 100              |
-------------------------------------------------------------------------------------------------
| 192.168.8.210 | 34800 | online | 0           | No valid partition  | No valid partition     |
-------------------------------------------------------------------------------------------------
```

新启动的 storage instance 此时还没有任何 partition。

STEP 3 迁移数据

运行 `BALANCE DATA` 命令， 查看当前的 balance 计划 id。如果当前集群有新机器加入，则会生成一个新的计划 id。对于已经平衡的集群，重复运行 `BALANCE DATA` 不会有任何新操作。

```
nebula> BALANCE DATA;
=============
| ID         |
=============
| 1570761786 |
-------------
```

也可通过 `BALANCE DATA $id` 查看具 balance 的具体执行进度。

```
nebula> BALANCE DATA 1570761786;
==============================================================================
| balanceId, spaceId:partId, src->dst                          | status      |
==============================================================================
| [1570761786, 1:1, 192.168.8.210:34600->192.168.8.210:44920]   | succeeded   |
------------------------------------------------------------------------------
| [1570761786, 1:1, 192.168.8.210:34700->192.168.8.210:34920]   | succeeded   |
------------------------------------------------------------------------------
| [1570761786, 1:1, 192.168.8.210:34500->192.168.8.210:34800]   | succeeded   |
------------------------------------------------------------------------------
...
| Total:189, Succeeded:170, Failed:0, In Progress:19, Invalid:0 | 89.947090%  |
------------------------------------------------------------------------------
```

`BALANCE DATA $id` 返回结果说明：

- 第一列 balanceId, spaceId:partId, src->dst 表示一个具体的 balance task。 以 1570761786, 1:88, 192.168.8.210:34700->192.168.8.210:35940 为例：
  - 1570761786 为 balance ID
  - 1:88，1 表示当前的 spaceId，88 表示迁移的 partId
  - 192.168.8.210:34700->192.168.8.210:3594，表示数据从192.168.8.210:34700 搬迁至 192.168.8.210:35940
- 第二列表示当前 task 的运行状态，目前有 4 种状态
  - Succeeded：运行成功
  - Failed：运行失败
  - In progress：运行中
  - Invalid：无效的 task

最后一行为对所有 task 运行状态的统计，部分 partition 尚未完成迁移。

STEP 4 查看结果

大多数情况下，搬迁数据是个比较漫长的过程。但是搬迁过程不会影响已有服务。运行结束后，进度会提示 100%。如果有运行失败的 task，可再次运行 `BALANCE DATA` 命令进行修复。如果多次运行仍无法修复，请与社区联系 GitHub。最后，通过 `SHOW HOSTS` 查看运行后的 partition 分布。

```
nebula> SHOW HOSTS;
===================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
===================================================================================
| 192.168.8.210 | 34600 | online | 3            | test: 3             | test: 37               |
-----------------------------------------------------------------------------------
```


2021 Vesoft Inc.

```
| 192.168.8.210 | 34900 | online | 0           | test: 0            | test: 38              |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 35940 | online | 0           | test: 0            | test: 37              |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34920 | online | 0           | test: 0            | test: 38              |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 44920 | online | 0           | test: 0            | test: 38              |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 35          | test: 35           | test: 37              |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 24          | test: 24           | test: 37              |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34800 | online | 38          | test: 38           | test: 38              |
-------------------------------------------------------------------------------------------
Got 8 rows (Time spent: 5074/6488 us)
```

可以看到 partition 和对应的数据已均衡的分布至各个机器。

Balance stop

`BALANCE DATA STOP` 命令用于停止已经开始执行的 balance data 计划。如果没有正在运行的 balance 计划，则会返回错误信息。如果有正在运行的 balance 计划，则会返回计划对应的 ID。

> 由于每个 balance 计划对应若干个 balance task，`BALANCE DATA STOP` 不会停止已经开始执行的 balance task，只会取消后续的 task，已经开始的 task 将继续执行直至完成。

用户可以在 `BALANCE DATA STOP` 之后输入 `BALANCE DATA $id` 来查看已经停止的 balance 计划状态。

所有已经开始执行的 task 完成后，可以再次执行 `BALANCE DATA`，重新开始 balance。

如果之前停止的计划中有失败的 task，则会继续执行之前的计划，如果之前停止的计划中所有 task 都成功了，则会新建一个 balance 计划并开始执行。

批量缩容

Nebula Graph 支持指定需要下线的机器进行批量缩容。语法为 `BALANCE DATA REMOVE $host_list`，例如 `BALANCE DATA REMOVE 192.168.0.1:50000,192.168.0.2:50000`，将在本次 balance 过程中移除 192.168.0.1:50000，192.168.0.2:50000 两台机器。

> 如果移除指定机器后，不满足副本数要求（例如剩余机器数小于副本数，或者三副本中有一台已经离线，此时要求移除剩余两副本中的一个），Nebula Graph 将拒绝本次 balance 请求，并返回相关错误码。

Balance leader

`BALANCE DATA` 仅能 balance partition，但是 leader 分布仍然不均衡，这意味着旧服务过载，而新服务未得到充分使用。运行 `BALANCE LEADER` 重新分布 Raft leader：

```
nebula> BALANCE LEADER
```

片刻后，使用 `SHOW HOSTS` 命令查看，此时 Raft leader 已均匀分布至所有的实例。

```
nebula> SHOW HOSTS;
===========================================================================================
| Ip            | Port  | Status | Leader count | Leader distribution | Partition distribution |
===========================================================================================
| 192.168.8.210 | 34600 | online | 13           | test: 13            | test: 37               |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34900 | online | 12           | test: 12            | test: 38               |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 35940 | online | 12           | test: 12            | test: 37               |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34920 | online | 12           | test: 12            | test: 38               |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 44920 | online | 13           | test: 13            | test: 38               |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34700 | online | 12           | test: 12            | test: 37               |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34500 | online | 13           | test: 13            | test: 37               |
-------------------------------------------------------------------------------------------
| 192.168.8.210 | 34800 | online | 13           | test: 13            | test: 38               |
-------------------------------------------------------------------------------------------
Got 8 rows (Time spent: 5039/6346 us)
```


2021 Vesoft Inc.

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.5.4 集群快照

创建快照

`CREATE SNAPSHOT` 命令可对整个集群创建当前时间点的快照，快照名称由 meta server 的时间戳组成。当前版本如果快照创建失败，必须通过 `DROP SNAPSHOT` 命令清除无效的快照。

当前版本不支持对指定的图空间创建快照，当执行 `CREATE SNAPSHOT` 后，将对集群中的所有图空间创建快照。例如：

```
nebula> CREATE SNAPSHOT;
Execution succeeded (Time spent: 22892/23923 us)
```

查看快照

`SHOW SNAPSHOT` 命令可查看集群中所有快照状态（VALID 或 INVALID）、名称和创建快照时所有 storage server 的 IP 地址。例如：

```
nebula> SHOW SNAPSHOTS;
=======================================================
| Name                        | Status | Hosts          |
=======================================================
| SNAPSHOT_2019_12_04_10_54_36 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_42 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_44 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------
```

删除快照

`DROP SNAPSHOT` 命令可删除指定名称的快照，语法为：

```
DROP SNAPSHOT <snapshot-name>
```

可以通过 `SHOW SNAPSHOTS` 命令获取快照名称， `DROP SNAPSHOT` 既可以删除有效的快照，也可以删除创建失败的快照。

```
nebula> DROP SNAPSHOT SNAPSHOT_2019_12_04_10_54_36;
nebula> SHOW SNAPSHOTS;
=======================================================
| Name                        | Status | Hosts          |
=======================================================
| SNAPSHOT_2019_12_04_10_54_42 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------
| SNAPSHOT_2019_12_04_10_54_44 | VALID  | 127.0.0.1:77833 |
-------------------------------------------------------
```

此时删除的快照已不在快照列表中。

注意事项

- 当系统结构发生变化后，最好立刻创建快照，例如在 add host、drop host、create space、drop space、balance 等操作之后。
- 当前版本不支持对创建失败的快照进行自动垃圾回收，后续将在 meta server 中开发 cluster checker 功能，通过异步线程检查集群状态，并自动回收创建失败的快照垃圾文件
- 当前版本暂未提供用户指定快照路径的功能，快照将默认创建在 `data_path/nebula` 目录下。
- 当前版本暂未提供快照恢复功能，需要用户根据实际的生产环境编写 shell 脚本实现。实现逻辑也比较简单，拷贝各 engine server 的快照到指定的文件夹下，并将此文件夹设置为 `data_path`，然后启动集群即可。

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.5.5 作业管理（flush 和 compact）

作业特指在存储层运行的一些长任务。比如 `compact` 和 `flush` 等。管理指对作业进行管理。比如让作业排队执行、查看作业状态、停止作业、恢复作业等。

命令列表

SUBMIT JOB COMPACT

`SUBMIT JOB COMPACT` 命令触发长耗时的 `RocksDB compact` 操作。示例返回结果如下：

```
nebula> SUBMIT JOB COMPACT;
=============
| New Job Id |
=============
| 40         |
--------------
```

修改默认 compact 线程数量请参考这里。

SUBMIT JOB FLUSH

`SUBMIT JOB FLUSH` 命令将内存中的 RocksDB memfile 写入到硬盘中。

```
nebula> SUBMIT JOB FLUSH;
=============
| New Job Id |
=============
| 2          |
--------------
```

SHOW JOB

返回单个作业信息

命令 `SHOW JOB <job_id>` 用于返回对应 ID 作业及其所有任务。作业到达 Meta 层后，Meta 会将作业分成多个任务并发送至 storage 层。

```
nebula> SHOW JOB 40;
=========================================================================================
| Job Id(TaskId) | Command(Dest)              | Status   | Start Time        | Stop Time         |
=========================================================================================
| 40             | flush basketballplayer     | finished | 12/17/19 17:21:30 | 12/17/19 17:21:30 |
-----------------------------------------------------------------------------------------
| 40-0           | 192.168.8.5                | finished | 12/17/19 17:21:30 | 12/17/19 17:21:30 |
-----------------------------------------------------------------------------------------
```

执行上述命令将返回 1 到多行结果，取决于 space 所在的 storaged 个数。

返回结果说明：

- `40` 为当前作业 ID
- `flush basketballplayer` 表示在 `basketballplayer` 图空间上执行了 flush 操作
- `finished` 表示运行已结束，且成功。其他可能的状态有 Queue、running、failed、stopped
- `12/17/19 17:21:30` 表示开始时间，初始为空(Queue)，当且仅当状态变为 running 时，才会设这个值
- `12/17/19 17:21:30` 表示结束时间，如果为 Queue 或者 running 状态，这里会为空，当状态变为 finished、failed、stopped 时会设置此值
- `40-0` 表示当前作业 ID 是 40，任务 ID 是 0
- `192.168.8.5` 表示运行在 192.168.8.5 这台机器上
- `finished` 表示运行已结束，且成功。其他可能的状态有 Queue、running、failed、stopped
- `12/17/19 17:21:30` 表示开始时间，因为任务初始即为 running 状态，所以这里永不为空
- `12/17/19 17:21:30` 表示结束时间，如果为 running 状态，这里会为空，当状态变为 finished、failed、stopped 时会设置此值

注意： 作业状态有五种，分为是 QUEUE、RUNNING、FINISHED、FAILED、STOPPED。状态机转换见以下说明：

2021 Vesoft Inc.

```
Queue -- running -- finished -- removed
     \          \               /
      \          \ -- failed -- /
       \          \            /
        \ ---------- stopped -/
```

返回所有作业信息

命令 `SHOW JOBS` 用于列出所有未过期的作业信息。默认作业过期时长为一周。用户可通过 `job_expired_secs` 参数更改过期时长。

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

返回结果说明见上节返回单个作业信息。

STOP JOB

命令 `STOP JOB` 用于在停止未完成的作业。

```
nebula> STOP JOB 22;
========================
| STOP Result          |
========================
| stop 1 jobs 2 tasks |
------------------------
```

RECOVER JOB

命令 `RECOVER JOB` 用于重新执行失败的作业，并返回 recover 的作业数目。

```
nebula> RECOVER JOB;
=====================
| Recovered job num |
=====================
| 5 job recovered   |
---------------------
```

FAQ

`SUBMIT JOB` 使用 HTTP 端口。请检查 Storage 之间的 HTTP 端口是否正常。你可以使用如下命令调试。

```
curl "http://{storaged-ip}:12000/admin?space={test}&op=compact"
```

最后更新: 2021年4月16日

2021 Vesoft Inc.

3.5.6 Compact

本文档将为您介绍 Compact。

Nebula Graph KV 基于 RocksDB 的默认 compact 做了定制。**Nebula Graph** 提供两种类型的 compact 相应的操作方法：

- 调用 RocksDB 默认的 compact：主要目的是在写入时自动进行小规模的 sst 文件合并，以保证短时间内的读速度。通常在日间业务时自动触发。运行以下命令启动 RocksDB 默认 compact；

```
nebula> UPDATE CONFIGS storage:rocksdb_column_family_options = {disable_auto_compactions = true};
```

`disable_auto_compactions` 参数默认为关闭（即 `false`）。导入数据前，建议将其开启。导入数据后，请务必将其改回 `false`，然后执行 `SUBMIT JOB COMPACT` 命令主动触发 compact。

- 调用 **Nebula Graph** 自定义的 compact：主要目的是完成大规模的 sst 文件合并、TTL 等大规模后台操作。通常建议在凌晨业务低谷时进行。通过 `SUBMIT JOB COMPACT` 来主动触发。

另外，两种方法的线程数均可通过如下命令调整。

```
nebula> UPDATE CONFIGS storage:rocksdb_db_options  = \
        { max_subcompactions = 4, max_background_jobs = 4};
```

最后更新: 2020年8月11日

2021 Vesoft Inc.

3.6 监控与统计

3.6.1 Metrics Exposer

访问这里。

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.6.2 Meta Metrics

介绍

目前，**Nebula Graph** 支持通过 HTTP 方式来获取 Meta Service 层的基本性能指标。

每一个性能指标都由三部分组成，分别为指标名、统计类型、时间范围。

counter_name	statistic_type	time_range

指标名

每个指标名都由服务名加模块名构成，meta 只统计心跳信息，目前支持获取如下接口：

```
meta_heartbeat_qps
meta_heartbeat_error_qps
meta_heartbeat_latency
```

统计类型

目前支持的统计类型有 SUM、COUNT、AVG、RATE 和 P 分位数 (P99，P999， ... ，P999999)。其中：

- `_qps`、`_error_qps` 后缀的指标，支持 SUM、COUNT、AVG、RATE，但不支持 P 分位；
- `_latency` 后缀的指标，支持 SUM、COUNT、AVG、RATE，也支持 P 分位。

时间范围

时间范围目前只支持三种，分别为 60、600、3600，分别表示最近一分钟，最近十分钟和最近一小时。

通过 HTTP 接口获取相应的性能指标

下面是一些示例：

```
meta_heartbeat_qps.avg.60         // 最近一分钟心跳的平均 QPS
meta_heartbeat_error_qps.count.60   // 最近一分钟心跳的平均错误总计数量
meta_heartbeat_latency.avg.60     // 最近一分钟心中的平均延时
```

假设本地启动了一个 nebula meta service，同时启动时设置的 `ws_http_port` 端口号为 11000。通过 HTTP 的 **GET** 接口发送，方法名为 **get_stats**，参数为 stats 加对应的指标名字。下面是通过 HTTP 接口获取指标的示例：

```
# 获取一个指标
curl -G "http://127.0.0.1:11000/get_stats?stats=meta_heartbeat_qps.avg.60"
# meta_heartbeat_qps.avg.60=580

# 同时获取多个指标
curl -G "http://127.0.0.1:11000/get_stats?stats=meta_heartbeat_qps.avg.60,meta_heartbeat_error_qps.avg.60"
# meta_heartbeat_qps.avg.60=537
# meta_heartbeat_error_qps.avg.60=579

# 同时获取多个指标并以 json 格式返回
curl -G "http://127.0.0.1:11000/get_stats?stats=meta_heartbeat_qps.avg.60,meta_heartbeat_error_qps.avg.60&returnjson"
# [{"value":533,"name":"meta_heartbeat_qps.avg.60"},{"value":574,"name":"meta_heartbeat_error_qps.avg.60"}]

# 获取所有指标
curl -G "http://127.0.0.1:11000/get_stats?stats"
# 或
curl -G "http://127.0.0.1:11000/get_stats"
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.6.3 Storage Metrics

介绍

目前，**Nebula Graph** 支持通过 HTTP 方式来获取 Storage Service 层的基本性能指标。

每一个性能指标都由三部分组成，分别为指标名、统计类型、时间范围。

counter_name	statistic_type	time_range

下面将分别介绍这三部分。

指标名

每个指标名都由服务名加模块名构成，目前支持获取如下接口：

```
获取点的属性 storage_vertex_props
获取边的属性 storage_edge_props
插入一个点 storage_add_vertex
插入一条边 storage_add_edge
删除一个点 storage_del_vertex
更新一个点的属性 storage_update_vertex
更新一条边的属性 storage_update_edge
读取一个键值对 storage_get_kv
写入一个键值对 storage_put_kv
仅限内部使用 storage_get_bound
```

每一个接口都有三个性能指标，分别为延迟(单位为 us)、成功的 QPS、发生错误的 QPS，后缀名如下：

```
_latency
_qps
_error_qps
```

将接口名和相应指标连接在一起即可获得完整的指标名，例如 `storage_add_vertex_latency`、`storage_add_vertex_qps`、`storage_add_vertex_error_qps` 分别代表插入一个点的延迟、QPS 和发生错误的 QPS。

统计类型

目前支持的统计类型有 SUM、COUNT、AVG、RATE 和 P 分位数 (P99，P999， ... ，P999999)。其中：

- `_qps`、`_error_qps` 后缀的指标，支持 SUM、COUNT、AVG、RATE，但不支持 P 分位；
- `_latency` 后缀的指标，支持 SUM、COUNT、AVG、RATE，也支持 P 分位。

时间范围

时间范围目前只支持三种，分别为 60、600、3600，分别表示最近一分钟、最近十分钟和最近一小时。

通过 HTTP 接口获取相应的性能指标

根据上面的介绍，就可以写出一个完整的指标名称了，下面是一些示例：

```
storage_add_vertex_latency.avg.60        // 最近一分钟插入一个点的平均延时
storage_get_bound_qps.rate.600        // 最近十分钟获取邻点的 QPS
storage_update_edge_error_qps.count.3600   // 最近一小时更新一条边发生错误的总计数量
```

假设本地启动了一个 nebula storage service，同时启动时设置的 `ws_http_port` 端口号为 12000。通过 HTTP 的 **GET** 接口发送，方法名为 **get_stats**，参数为 stats 加对应的指标名字。下面是通过 HTTP 接口获取指标的示例：

```
# 获取一个指标
curl -G "http://127.0.0.1:12000/get_stats?stats=storage_vertex_props_qps.rate.60"
# storage_vertex_props_qps.rate.60=2674

# 同时获取多个指标
curl -G "http://127.0.0.1:12000/get_stats?stats=storage_vertex_props_qps.rate.60,storage_vertex_props_latency.avg.60"
# storage_vertex_props_qps.rate.60=2638
# storage_vertex_props_latency.avg.60=812

# 同时获取多个指标并以 json 格式返回
```


2021 Vesoft Inc.

```
curl -G "http://127.0.0.1:12000/get_stats?stats=storage_vertex_props_qps.rate.60,storage_vertex_props_latency.avg.60&returnjson"
# [{"value":2723,"name":"storage_vertex_props_qps.rate.60"},{"value":804,"name":"storage_vertex_props_latency.avg.60"}]

# 获取所有指标
curl -G "http://127.0.0.1:12000/get_stats?stats"
# 或
curl -G "http://127.0.0.1:12000/get_stats"
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.6.4 Graph Metrics

介绍

目前，**Nebula Graph** 支持通过 HTTP 方式来获取 Graph Service 层的基本性能指标。

每一个性能指标都由三部分组成，分别为指标名、统计类型、时间范围。

counter_name	statistic_type	time_range

指标名

每个指标名都由服务名加模块名构成，目前支持获取如下接口：

```
通过 storageClient 发送的请求，需要同时向多个 storage 并发多条消息时，按一次统计  graph_storageClient
通过 metaClient 发送的请求
graph_graph_all 客户端向 graph 发送的请求，当一条请求包含多条语句时，按一条计算 graph_metaClient
插入点 graph_insertVertex
插入边 graph_insertEdge
删除点 graph_deleteVertex
删除边 graph_deleteEdge //未支持
更新点的属性 graph_updateVertex
更新边的属性 graph_updateEdge
执行 go 命令 graph_go
查找最小路径或者全路径 graph_findPath
获取点属性，不统计获取点的总数，只统计执行命令的数量 graph_fetchVertex
获取边属性，不统计边的总数，只统计执行命令的数量 graph_fetchEdge
```

每一个接口都有三个性能指标，分别为延迟(单位为 us)、成功的 QPS、发生错误的 QPS，后缀名如下：

```
_latency
_qps
_error_qps
```

将接口名和相应指标连接在一起即可获得完整的指标名，例如 `graph_insertVertex_latency` 、 `graph_insertVertex_qps` 、 `graph_insertVertex_error_qps` 、分别代表插入一个点的延迟、QPS 和发生错误的 QPS。

统计类型

目前支持的统计类型有 SUM、COUNT、AVG、RATE 和 P 分位数 (P99，P999， ... ，P999999)。其中：

- `_qps` 、 `_error_qps` 后缀的指标，支持 SUM、COUNT、AVG、RATE，但不支持 P 分位；
- `_latency` 后缀的指标，支持 SUM、COUNT、AVG、RATE，也支持 P 分位。

时间范围

时间范围目前只支持三种，分别为 60、600、3600，分别表示最近一分钟，最近十分钟和最近一小时。

通过 HTTP 接口获取相应的性能指标

根据上面的介绍，就可以写出一个完整的指标名称了，下面是一些示例：

```
graph_insertVertex_latency.avg.60        // 最近一分钟插入点命令执行成功的平均延时
graph_updateEdge_error_qps.count.3600   // 最近一小时更新边命令失败的总计数量
```

假设本地启动了一个 nebula graph service，同时启动时设置的 `ws_http_port` 端口号为 13000。通过 HTTP 的 **GET** 接口发送，方法名为 **get_stats**，参数为 stats 加对应的指标名字。下面是通过 HTTP 接口获取指标的示例：

```
# 获取一个指标
curl -G "http://127.0.0.1:13000/get_stats?stats=graph_insertVertex_qps.rate.60"
# graph_insertVertex_qps.rate.60=3069

# 同时获取多个指标
curl -G "http://127.0.0.1:13000/get_stats?stats=graph_insertVertex_qps.rate.60, graph_deleteVertex_latency.avg.60"
# graph_insertVertex_qps.rate.60=3069
# graph_deleteVertex_latency.avg.60=837

# 同时获取多个指标并以 json 格式返回
curl -G "http://127.0.0.1:13000/get_stats?stats=graph_insertVertex_qps.rate.60, graph_deleteVertex_latency.avg.60&returnjson"
```


2021 Vesoft Inc.

```
# [{"value":2373,"name":"graph_insertVertex_qps.rate.60"},{"value":760,"name":"graph_deleteVertex_latency.avg.60"}]

# 获取所有指标
curl -G "http://127.0.0.1:13000/get_stats?stats"
# 或
curl -G "http://127.0.0.1:13000/get_stats"
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.7 源码开发和 API

3.7.1 KV 接口

接口示例

Nebula Graph storage 提供 key-value 接口，用户可以通过 StorageClient 进行 kv 的相关操作，请注意用户仍然需要通过 console 来创建 space。目前支持的接口有 Get 和 Put，接口如下。

```
folly::SemiFuture<StorageRpcResponse<storage::cpp2::ExecResponse>> put(
  GraphSpaceID space,
  std::vector<nebula::cpp2::Pair> values,
  folly::EventBase* evb = nullptr);

folly::SemiFuture<StorageRpcResponse<storage::cpp2::GeneralResponse>> get(
  GraphSpaceID space,
  const std::vector<std::string>& keys,
  folly::EventBase* evb = nullptr);
```

后续将提供 remove，removeRange 以及 scan 的方法。

下面结合示例说明 kv 接口的使用方法：

```
// Put 接口
std::vector<nebula::cpp2::Pair> pairs;
for (int32_t i = 0; i < 1000; i ++) {
    auto key = std::to_string(folly::Random::rand32(1000000000));
    auto value = std::to_string(folly::Random::rand32(1000000000));
    pairs.emplace_back(apache::thrift::FragileConstructor::FRAGILE,
                       std::move(key), std::move(value));
}

// 通过 StorageClient 发送请求，相应的参数为 spaceId，以及写入的键值对
auto future = storageClient->put(spaceId, std::move(pairs));
// 获取结果
auto resp = std::move(future).get();
```

```
// Get 接口
std::vector<std::string> keys;
for (auto& pair : pairs) {
    keys.emplace_back(pair.first);
}

// 通过 StorageClient 发送请求，相应的参数为 spaceId，以及要获取的 keys
auto future = storageClient->get(spaceId, std::move(keys));
// 获取结果
auto resp = std::move(future).get()
```

处理返回结果

用户可以通过检查 rpc 返回结果查看相应操作是否成功。此外由于每个 Nebula Graph storage 中都对数据进行了分片，因此如果对应的 Partition 失败了，也会返回每个失败的 Partition 的错误码。若任意一个 Partition 失败，则整个请求失败(resp.succeeded()为 false)，但是其他成功的 Partition 仍然会成功写入或读取。

用户可以进行重试，直至所有请求都成功。目前 StorageClient 不支持自动重试，用户可以根据错误码决定是否进行重试。

```
// 判断调用是否成功
if (!resp.succeeded()) {
    LOG(ERROR) << "Operation Failed";
    return;
}

// 失败的 Partition 以及相应的错误码
if (!resp.failedParts().empty()) {
    for (const auto& partEntry : resp.failedParts()) {
        LOG(ERROR) << "Operation Failed in " << partEntry.first << ", Code: "
                   << static_cast<int32_t>(partEntry.second);
    }
    return;
}
```


2021 Vesoft Inc.

读取返回值

对于 Get 接口，用户需要一些操作来获取相应的返回值。Nebula storage 是基于 Raft 的多副本，所有读写操作只能发送给对应 partition 的 leader。当一个 rpc 请求包含了多个跨 partition 的 get 时，Storage Client 会给访问这些 key 所对应的 Partition leader。每个 rpc 返回都单独保存在一个 unordered_map 中，目前还需要用户在这些 unordered_map 中遍历查找 key 是否存在。示例如下：

```
// 查找 key 对应的 value 是否在返回结果中，如果存在，则保存在 value 中
bool found = false;
std::string value;
// resp.responses()中是多个 storage server 返回的结果
for (const auto& result : resp.responses()) {
    // result.values 即为某个 storage server 返回的 key-value paris
    auto iter = result.values.find(key);
    if (iter != result.values.end()) {
        value = iter->second;
        found = true;
        break;
    }
}
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

3.7.2 Nebula Graph 支持的客户端

目前，**Nebula Graph** 支持如下客户端：

- Go 客户端
- Python 客户端
- Java 客户端

最后更新: 2020年4月29日

2021 Vesoft Inc.

4. 图算法

4.1 nebula-algorithm

4.1.1 什么是 nebula-algorithm

nebula-algorithm 是一款基于 GraphX 的 Spark 应用程序，提供了 PageRank 和 Louvain 社区发现的图计算算法。使用 nebula-algorithm，您能以提交 Spark 任务的形式对 Nebula Graph 数据库中的数据执行图计算。

目前 nebula-algorithm 仅提供了 PageRank 和 Louvain 社区发现算法。如果您有其他需求，可以参考本项目，编写 Spark 应用程序调用 GraphX 自带的其他图算法，如 LabelPropagation、ConnectedComponent 等。

实现方法

nebula-algorithm 根据以下方式实现图算法：

1. 从 Nebula Graph 数据库中读取图数据并处理成 DataFrame。
2. 将 DataFrame 转换为 GraphX 的图。
3. 调用 GraphX 提供的图算法（例如 PageRank）或者您自己实现的算法（例如 Louvain 社区发现）。

详细的实现方式，您可以参考 LouvainAlgo.scala 和 PageRankAlgo.scala。

PageRank 和 Louvain 简介

PAGERANK

GraphX 的 PageRank 算法基于 Pregel 计算模型，该算法流程包括 3 个步骤：

1. 为图中每个顶点（如网页）设置一个相同的初始 PageRank 值。
2. 第一次迭代：沿边发送消息，每个顶点收到所有关联边上邻接点（Adjacent Node）的信息，得到一个新的 PageRank 值；
3. 第二次迭代：用这组新的 PageRank 按不同算法模式对应的公式形成该顶点新的 PageRank。

关于 PageRank 的详细信息，参考 Wikipedia PageRank 页面。

LOUVAIN

Louvain 是基于模块度（Modularity）的社区发现算法，通过模块度来衡量一个社区的紧密程度，属于图的聚类算法。如果一个顶点加入到某一社区中使该社区的模块度相比其他社区有最大程度的增加，则该顶点就应当属于该社区。如果加入其它社区后没有使其模块度增加，则留在自己当前社区中。详细信息，您可以参考论文《Fast unfolding of communities in large networks》。

Louvain 算法包括两个阶段，其流程就是这两个阶段的迭代过程。

1. 阶段一：不断地遍历网络图中的顶点，通过比较顶点给每个邻居社区带来的模块度的变化，将单个顶点加入到能够使 Modulaity 模块度有最大增量的社区中。例如，顶点 v 分别加入到社区 A、B、C 中，使得三个社区的模块度增量为 -1、1、2，则顶点 v 最终应该加入到社区 C 中。
2. 阶段二：对第一阶段进行处理，将属于同一社区的顶点合并为一个大的超点重新构造网络图，即一个社区作为图的一个新的顶点。此时两个超点之间边的权重是两个超点内所有原始顶点之间相连的边权重之和，即两个社区之间的边权重之和。

整个 Louvain 算法就是不断迭代第一阶段和第二阶段，直到算法稳定（图的模块度不再变化）或者到达最大迭代次数。

2021 Vesoft Inc.

使用场景

您可以将 PageRank 算法应用于以下场景：

- 社交应用的相似度内容推荐：在对微博、微信等社交平台进行社交网络分析时，可以基于 PageRank 算法根据用户通常浏览的信息以及停留时间实现基于用户的相似度的内容推荐。
- 分析用户社交影响力：在社交网络分析时根据用户的 PageRank 值进行用户影响力分析。
- 文献重要性研究：根据文献的 PageRank 值评判该文献的质量，PageRank 算法就是基于评判文献质量的想法来实现设计。

您可以将 Louvain 算法应用于以下场景：

- 金融风控：在金融风控场景中根据用户行为特征进行团伙识别。
- 社交网络：基于网络关系中点对（一条边连接的两个顶点）之间关联的广度和强度进行社交网络划分；对复杂网络分析、电话网络分析人群之间的联系密切度。
- 推荐系统：基于用户兴趣爱好的社区发现，可以根据社区并结合协同过滤等推荐算法进行更精确有效的个性化推荐。

最后更新: 2021年4月7日

2021 Vesoft Inc.

4.1.2 使用限制

nebula-algorithm 目前仅支持 Nebula Graph v1.x，不支持 Nebula Graph v2.x。

最后更新: 2021年4月15日

2021 Vesoft Inc.

4.1.3 编译 nebula-algorithm

前提条件

编译 nebula-algorithm 之前，必须先编译 Nebula Spark Connector。

操作步骤

切换到 `nebula-java/tools/nebula-algorithm` 目录，并编译打包 nebula-algorithm。

```
$ cd nebula-java/tools/nebula-algorithm
$ mvn clean package -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

编译成功后，您可以在当前目录里看到如下目录结构。

```
.
├── README.md
├── pom.xml
├── src
│   ├── main
│   └── test
└── target
    ├── classes
    ├── classes.682519136.timestamp
    ├── maven-archiver
    ├── maven-status
    ├── nebula-algorithm-1.x.y-tests.jar
    ├── nebula-algorithm-1.x.y.jar
    ├── original-nebula-algorithm-1.x.y.jar
    ├── test-classes
    └── test-classes.682519136.timestamp
```

在 `target` 目录下，您可以看到 `nebula-algorithm-1.x.y.jar` 文件。

> **说明**：JAR 文件版本号会因 Nebula Java Client 的发布版本而异。您可以在 nebula-java 仓库的 Releases 页面 查看最新的 v1.x 版本。

后续操作

在使用 nebula-algorithm 时，您可以参考 `target/classes/application.conf` 根据实际情况修改配置文件。详细信息请参考 使用示例。

最后更新: 2021年4月7日

2021 Vesoft Inc.

4.1.4 使用示例

一般，您可以按以下步骤使用 nebula-algorithm：

1. 参考配置文件修改 nebula-algorithm 的配置。
2. 运行 nebula-algorithm。

本文以一个示例说明如何使用 nebula-algorithm。

示例环境

- 三台虚拟机，配置如下：
  - CPU：Intel(R) Xeon(R) Platinum 8260M CPU @ 2.30GHz
  - Processors：32
  - CPU Cores：16
  - 内存：128 GB
- 软件环境：
  - Spark：spark-2.4.6-bin-hadoop2.7 三个节点集群
  - yarn V2.10.0：三个节点集群
  - Nebula Graph V1.2.1：分布式部署，默认配置

示例数据

在本示例中，Nebula Graph 图空间名称为 algoTest，Schema 要素如下表所示。

要素	名称	属性
标签	`PERSON`	无
边类型	`FRIEND`	`likeness` (`double`)

前提条件

在操作之前，您需要确认以下信息：

- 已经完成 nebula-algorithm 编译。详细信息参考 编译 nebula-algorithm。
- Nebula Graph 数据库中已经有图数据。您可以使用不同的方式将其他来源的数据导入 Nebula Graph 数据库中，例如 Spark Writer。
- 当前机器上已经安装 Spark 并已启动 Spark 服务。

第 1 步. 修改配置文件

根据项目中的 `src/main/resources/application.conf` 文件修改 nebula-algorithm 配置。

```
{
  # Spark 相关设置
  spark: {
    app: {
        # Spark 应用程序的名称，可选项。默认设置为您即将运行的算法名称，例如 PageRank
        name: PageRank

        # Spark 中分区数量，可选项。
        partitionNum: 12
    }

    master: local

    # 可选项，如果这里未设置，则在执行 spark-submit spark 任务时设置
    conf: {
        driver-memory: 20g
```


2021 Vesoft Inc.

```
        executor-memory: 100g
        executor-cores: 3
        cores-max:6
    }
  }

  # Nebula Graph 相关配置
  nebula: {
    # 必需。Meta 服务的信息
    addresses: "127.0.0.1:45500" # 如果有多个 Meta 服务复本，则以英文逗号隔开
    user: root
    pswd: nebula
    space: algoTest
    # 必需。创建图空间时设置的分区数量。如果创建图空间时未设置分区数，则设置为默认值 100
    partitionNumber: 100
    # 必需。Nebula Graph 的边类型，如果有多种边类型，则以英文逗号隔开
    labels: ["FRIENDS"]

    hasWeight: true
    # 如果 hasWeight 配置为 true，则必须配置 weightCols。根据 labels 列出的边类型，按顺序在 weightCols 里设置对应的属性，一种边类型仅对应一个属性
    # 说明：nebula-algorithm 仅支持同构图，所以，weightCols 中列出的属性的数据类型必须保持一致而且均为数字类型
    weightCols: ["likeness"] # 如果 labels 里有多种边类型，则相应设置对应的属性，属性之间以英文逗号隔开
  }

  algorithm: {
    # 指定即将执行的算法，可以配置为 pagerank 或 louvain
    executeAlgo: louvain
    # 指定算法结果的存储路径
    path: /tmp

    # 如果选择的是 PageRank，则配置 pagerank 相关参数
    #pagerank: {
    #    maxIter: 20
    #    resetProb: 0.15  # 默认值为 0.15
    #}

    # 如果选择的是 louvain，则配置 louvain 相关参数
    #louvain: {
    #   maxIter: 20
    #    internalIter: 10
    #    tol: 0.5
   #}
  }
}
```

第 2 步. 执行 nebula-algorithm

运行以下命令，提交 nebula-algorithm 应用程序。

```
spark-submit --master "local" --class com.vesoft.nebula.tools.algorithm.Main /your-jar-path/nebula-algorithm-1.x.y.jar -p /your-application.conf-path/application.conf
```

其中，

- `--master` ：指定 Spark 集群中Master 进程的 URL。详细信息，参考 master-urls。
- `--class` ：指定 Driver 主类。
- 指定 nebula-algorithm JAR 文件的路径，JAR 文件版本号以您实际编译得到的 JAR 文件名称为准。
- `-p` ：Spark 配置文件文件路径。
- 其他：如果您未在配置文件中设置 Spark 的任务资源分配（ `conf` ）信息，您可以在这个命令中指定。例如，本示例中， `--driver-memory=20G --executor-memory=100G --executor-cores=3` 。

测试结果

按本示例设置的 Spark 任务资源分配，对于一个拥有一亿个数据的数据集：

- PageRank 的执行时间（PageRank 算法执行时间）为 21 分钟
- Louvain 的执行时间（Reader + Louvain 算法执行时间）为 1.3 小时

最后更新: 2021年4月9日

2021 Vesoft Inc.

5. 数据传输

5.1 Nebula Exchange

5.1.1 认识 Nebula Exchange

什么是 Nebula Exchange

Nebula Exchange（简称为 Exchange）是一款 Apache Spark™ 应用，用于在分布式环境中将集群中的数据批量迁移到 Nebula Graph 中，能支持多种不同格式的批式数据和流式数据的迁移。

Exchange 由 Reader、Processor 和 Writer 三部分组成。Reader 读取不同来源的数据返回 DataFrame 后，Processor 遍历 DataFrame 的每一行，根据配置文件中 `fields` 的映射关系，按列名获取对应的值。在遍历指定批处理的行数后，Writer 会将获取的数据一次性写入到 Nebula Graph 中。下图描述了 Exchange 完成数据转换和迁移的过程。

适用场景

Exchange 可被用于以下场景：

- 您想将来自 Kafka、Pulsar 平台的流式数据，如日志文件、网购数据、游戏内玩家活动、社交网站信息、金融交易大厅或地理空间服务，以及来自数据中心内所连接设备或仪器的遥测数据等转化为属性图的点或边数据，并导入 Nebula Graph 数据库。
- 您想从关系型数据库（如 MySQL）或者分布式文件系统（如 HDFS）中读取批式数据，如某个时间段内的数据，将它们转化为属性图的点或边数据，并导入 Nebula Graph 数据库。
- 您想将大批量数据生成 Nebula Graph 能识别的 SST 文件，再导入 Nebula Graph 数据库。

2021 Vesoft Inc.

产品优点

Exchange 具有以下优点：

- 适应性强：支持将多种不同格式或不同来源的数据导入 Nebula Graph 数据库，便于您迁移数据。
- 支持导入 SST：支持将不同来源的数据转换为 SST 文件，用于数据导入。

  说明：仅 Linux 系统支持导入 SST 文件。
- 支持断点续传：导入数据时支持断点续传，有助于您节省时间，提高数据导入效率。

  说明：目前仅迁移 Neo4j 数据时支持断点续传。
- 异步操作：会在源数据中生成一条插入语句，发送给查询服务，最后再执行 Nebula Graph 的插入操作。
- 灵活性强：支持同时导入多个标签和边类型，不同标签和边类型可以是不同的数据来源或格式。
- 统计功能：使用 Apache Spark™ 中的累加器统计插入操作的成功和失败次数。
- 易于使用，对用户友好：采用 HOCON（Human-Optimized Config Object Notation）配置文件格式，具有面向对象风格，便于理解和操作。

数据格式和来源

Exchange v1.x 支持将以下格式或来源的数据转换为 Nebula Graph v1.x 能识别的点和边数据：

- 存储在 HDFS 的数据，包括：
  - Apache Parquet
  - Apache ORC
  - JSON
  - CSV
- Apache HBase™
- 数据仓库：HIVE
- 图数据库：Neo4j 2.4.5-M1。仅 Neo4j 支持断点续传
- 关系型数据库：MySQL
- 流处理软件平台：Apache Kafka®
- 发布/订阅消息系统：Apache Pulsar 2.4.5

最后更新: 2021年4月7日

2021 Vesoft Inc.

使用限制

本文描述 Exchange v1.x 的一些使用限制。

NEBULA GRAPH 版本

Exchange v1.x 仅支持 Nebula Graph v1.x。请勿使用低版本的Exchange导入数据至高版本Nebula Graph中，例如使用Exchange v1.1导入数据至Nebula Graph v1.2.0。

如果您正在使用 Nebula Graph v2.x，请使用 Nebula Exchange v2.x。

使用环境

Exchange v1.x 支持以下操作系统：

- CentOS 7
- macOS

说明：仅 Linux 系统支持导入 SST 文件。

软件依赖

为保证 Exchange v1.x 正常工作，确认您的机器上已经安装以下软件：

- Apache Spark：2.3.0 及以上版本
- Java：1.8
- Scala：2.10.7、2.11.12、2.12.10

在以下使用场景，还需要部署 Hadoop Distributed File System (HDFS)：

- 以客户端形式迁移 HDFS 上的数据
- 以 SST 文件格式迁移数据

最后更新: 2021年4月21日

2021 Vesoft Inc.

名词解释

本文描述了您在使用 Exchange 时可能需要了解的名词解释。

- Nebula Exchange：在本手册中简称为 Exchange 或 Exchange v1.x，一款基于 Apache Spark™ 的 Spark 应用，用于批量数据迁移。它支持将多种不同来源和格式的数据文件转换为 Nebula Graph 能识别的点和边数据，再并发导入 Nebula Graph。
- Aparch Spark™：是专为大规模数据处理而设计的快速通用的计算引擎，是 Apache 软件基金会的一个开源项目。
- Driver Program：在本手册中简称为 Driver，是运行应用的 main 函数并且新建 SparkContext 实例的程序。

最后更新: 2021年4月7日

2021 Vesoft Inc.

常见问题

Exchange 支持哪些版本的 Nebula Graph？

您可以查看 Exchange 的 使用限制 获取 Exchange 对 Nebula Graph 的最新支持信息。

Exchange 与 Spark Writer 有什么关系？

Exchange 是在 Spark Writer 基础上开发的 Spark 应用程序，二者均适用于在分布式环境中将集群的数据批量迁移到 Nebula Graph 中，但是，后期的维护工作将集中在 Exchange 上。与 Spark Writer 相比，Exchange 有以下改进：

- 支持更丰富的数据源，如 MySQL、Neo4j、Hive、HBase、Kafka、Pulsar 等。
- 修复了 Spark Writer 的部分问题。例如，Spark 读取 HDFS 里的数据时，默认读取到的源数据均为 String 类型，可能与 Nebula Graph 定义的 Schema 不同，所以，Exchange 增加了数据类型的自动匹配和类型转换，当 Nebula Graph 定义的 Schema 中数据类型为非 String 类型（如 double）时，Exchange 会将 String 类型的源数据转换为对应的类型（如 double）。

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.1.2 编译 Exchange

按以下步骤编译 Exchange v1.x：

1. 克隆 `nebula-java` 源代码。

```
git clone -b v1.0 https://github.com/vesoft-inc/nebula-java.git
```

2. 切换到 `nebula-java` 目录，并打包 Nebula Java 1.x。

```
cd nebula-java
mvn clean install -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

3. 进入 `tools/exchange` 目录，并编译 Exchange v1.x。

```
cd nebula-java/tools/exchange
mvn clean package -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

编译成功后，您可以在当前目录里看到如下目录结构。

```
├── README.md
├── dependency-reduced-pom.xml
├── pom.xml
├── scripts
│   ├── README.md
│   ├── mock_data.py
│   ├── pulsar_producer.py
│   ├── requirements.txt
│   └── verify_nebula.py
├── src
│   └── main
└── target
    ├── classes
    ├── classes.timestamp
    ├── exchange-1.x.y-javadoc.jar
    ├── exchange-1.x.y-sources.jar
    ├── exchange-1.x.y.jar
    ├── generated-test-sources
    ├── maven-archiver
    ├── maven-status
    ├── original-exchange-1.x.y.jar
    ├── site
    ├── test-classes
    └── test-classes.timestamp
```

在 `target` 目录下，您可以看到 `exchange-1.x.y.jar` 文件。

> **说明**：JAR 文件版本号会因 Nebula Java Client 的发布版本而异。您可以在 nebula-java 仓库的 Releases 页面 查看最新的 v1.x 版本。

在迁移数据时，您可以参考 `target/classes/application.conf`、`target/classes/server_application.conf`、`target/classes/stream_application.conf` 根据实际情况修改配置文件。

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.1.3 操作指南

导入数据步骤

您可以按本文描述的步骤使用 Exchange 将指定来源的数据导入到 Nebula Graph 中。

前提条件

开始迁移数据之前，您需要确保以下信息：

- 已经安装部署了 Nebula Graph 并获取查询引擎所在服务器的 IP 地址、用户名和密码。
- 已经完成 Exchange 编译。详细信息，参考 编译 Nebula Exchange。
- 已经安装 Spark。
- 在 Nebula Graph 中创建图数据模式需要的所有信息，包括标签和边类型的名称、属性等。

操作步骤

按以下步骤将不同来源的数据导入 Nebula Graph 数据库：

1. 在 Nebula Graph 中构图，包括创建图空间、创建图数据模式（Schema）。
2. （可选）处理源数据。例如，在导入 Neo4j 时，为提高导出速度，在 Neo4j 数据库中为指定的标签属性创建索引。
3. 分别修改 Spark、Nebula Graph 以及点和边数据的配置文件。

   说明：完成 Exchange 编译后，进入 `nebula-java/tools/exchange` 目录，您可以参考 `target/classes/server_application.conf`、`target/classes/application.conf` 和 `target/classes/stream_application.conf` 文件修改配置文件。
4. （可选）检查配置文件是否正确。
5. 向 Nebula Graph 导入数据。
6. 在 Nebula Graph 中验证数据是否已经完整导入。
7. （可选）在 Nebula Graph 中重构索引。

关于详细操作步骤，根据数据来源不同，您可以参考相应的操作示例：

- 导入 Neo4j 数据
- 导入 HIVE 数据
- 导入 CSV 文件数据
- 导入 JSON 文件数据
- 导入 SST 文件数据

最后更新: 2021年4月7日

2021 Vesoft Inc.

导入 Neo4j 数据

您可以使用 Exchange 将 Neo4j 数据离线批量导入 Nebula Graph 数据库。

实现方法

Exchange 使用 **Neo4j Driver 4.0.1** 实现对 Neo4j 数据的读取。执行批量导出之前，您需要在配置文件中写入针对标签（label）和关系类型（Relationship Type）自动执行的 Cypher 语句，以及 Spark 分区数，提高数据导出性能。

Exchange 读取 Neo4j 数据时需要完成以下工作：

1. Exchange 中的 Reader 会将配置文件中 `exec` 部分的 Cypher `RETURN` 语句后面的语句替换为 `COUNT(*)`，并执行这个语句，从而获取数据总量，再根据 Spark 分区数量计算每个分区的起始偏移量和大小。
2. （可选）如果用户配置了 `check_point_path` 目录，Reader 会读取目录中的文件。如果当前处于续传的状态，Reader 会计算每个 Spark 分区应该有的偏移量和大小。
3. 在每个 Spark 分区里，Exchange 中的 Reader 会在 Cypher 语句后面添加不同的 `SKIP` 和 `LIMIT` 语句，调用 Neo4j Driver 并行执行，将数据分布到不同的 Spark 分区中。
4. Reader 最后将返回的数据处理成 DataFrame。

至此，Exchange 即完成了对 Neo4j 数据的导出。之后，数据被并行写入 Nebula Graph 数据库中。

整个过程如下图所示。

本文以一个示例说明如何使用 Exchange 将 Neo4j 的数据批量导入 Nebula Graph 数据库。

> **说明**：本文仅说明如何以客户端形式迁移数据。您也能通过 SST 文件将 Neo4j 的数据批量转入 Nebula Graph 数据库，具体操作基本相同，仅配置文件修改有差异，详细信息参考 导入 SST 文件。

示例场景

假设您在 Neo4j 数据库里有一个图数据集，需要使用 Exchange 将数据迁移到 Nebula Graph 数据库中。

2021 Vesoft Inc.

环境配置

以下为本示例中的环境配置信息：

- 服务器规格：
  - CPU：Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
  - CPU 内核数：14
  - 内存：251 GB
- Spark：单机版，2.4.6 pre-build for Hadoop 2.7
- Neo4j：3.5.20 Community Edition
- Nebula Graph 1.2.1，使用 Docker Compose 部署。详细信息，参考 使用 Docker Compose 部署 Nebula Graph

数据集信息

Neo4j 的数据集信息如下：

- 两种标签（此处命名为 tagA 和 tagB），共计 100 万个节点（node）。
- 一种连接上述两类点的关系类型（Relationship Type，此处命名为 edgeAB），共计 1000 万条关系边（Relationship）。

上述三者分别拥有如下表所示的属性。

要素名称	属性 idInt	属性 idString	属性 tboolean	属性 tdouble
tagA	`int`	`string`	`bool`	`double`
tagB	`int`	`string`	`bool`	`double`
edgeAB	`int`	`string`	`bool`	`double`

数据库信息

在本示例中 Neo4j 数据库和 Nebula Graph 数据库的信息如下：

- Neo4j 数据库：
  - 服务器地址及端口为：`bolt://127.0.0.1:7687`
  - 用户名：*neo4j*
  - 密码：*neo4j*
- Nebula Graph 数据库：
  - 服务器地址及端口为：`127.0.0.1:3699`
  - 用户名：本示例中 Nebula Graph 数据库没有启用身份认证，所以，用户名为 *user*
  - 密码：*password*

前提条件

开始迁移数据之前，您需要确保以下信息：

- 已经完成 Exchange 编译。详细信息，参考 编译 Exchange。
- 已经安装了 Spark。
- 在 Nebula Graph 中创建图数据模式所需的所有信息，包括标签和边类型的名称、属性等。

2021 Vesoft Inc.

操作步骤

步骤 1. 在 Nebula Graph 中构建图数据模式（Schema）

根据示例场景，按以下步骤完成 Nebula Graph 构建图数据模式：

1. 确认图数据模式要素：Neo4j 中的标签和关系类型及其属性与 Nebula Graph 中的数据模式要素对应关系如下表所示。

Neo4j 要素	Nebula Graph 要素	要素名称
Node	Tag	tagA 和 tagB
Relationship Type	Edge Type	edgAB

2. 确认 Nebula Graph 需要的分区数量：全集群硬盘数量 *（2 至 10）。本示例中假设为 10。
3. 确认 Nebula Graph 需要的副本数量。本示例中假设为 1。
4. 在 Nebula Graph 里创建一个图空间 **test**，并创建一个图数据模式，如下所示。

```
-- 创建图空间，本示例中假设只需要一个副本
nebula> CREATE SPACE test(partition_num=10, replica_factor=1);

-- 选择图空间 test
nebula> USE test;

-- 创建标签 tagA
nebula> CREATE TAG tagA(idInt int, idString string, tboolean bool, tdouble double);

-- 创建标签 tagB
nebula> CREATE TAG tagB(idInt int, idString string, tboolean bool, tdouble double);

-- 创建边类型 edgeAB
nebula> CREATE EDGE edgeAB(idInt int, idString string, tboolean bool, tdouble double);
```

关于 Nebula Graph 构图的更多信息，参考《Nebula Graph Database 手册》的 快速开始。

步骤 2. 配置源数据

为了提高 Neo4j 数据的导出速度，在 Neo4j 数据库中为 tagA 和 tagB 的 `idInt` 属性创建索引。详细信息，参考 Neo4j 用户手册。

步骤 3. 修改配置文件

完成 Exchange 编译后，进入 `nebula-java/tools/exchange` 目录，您可以参考 `target/classes/server_application.conf` 修改配置文件。在本示例中，文件被重命名为 `neo4j_application.conf`。以下仅详细说明点和边数据的配置信息，本次示例中未使用的配置项已被注释，但是提供了配置说明。Spark 和 Nebula Graph 相关配置，参考 Spark 参数、Nebula Graph 参数。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: Spark Writer
    }

    driver: {
      cores: 1
      maxResultSize: 1G
    }

    cores {
      max: 16
    }
  }

  # Nebula Graph 相关配置
  nebula: {
    address:{
      # 以下为 Nebula Graph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:3699"]
      meta:["127.0.0.1:45500"]
    }
    # 填写的账号必须拥有 Nebula Graph 相应图空间的写数据权限。
    user: user
    pswd: password
    # 填写 Nebula Graph 中需要写入数据的图空间名称。
    space: test
    connection {
      timeout: 3000
      retry: 3
    }
```


2021 Vesoft Inc.

```
  execution {
    retry: 3
  }
  error: {
    max: 32
    output: /tmp/errors
  }
  rate: {
    limit: 64M
    timeout: 1000
  }
}

# 处理点数据
tags: [
  # 设置标签相关信息
  {
  name: tagA
  # 设置 Neo4j 数据库服务器地址，String 类型，格式必须为 "bolt://ip:port"。
  server: "bolt://127.0.0.1:7687"

  # Neo4j 数据库登录账号和密码。
  user: neo4j
  password: neo4j

  # 传输是否加密，默认值为 false，表示不加密；设置为 true 时，表示加密。
  encryption: false

  # 设置源数据所在 Neo4j 数据库的名称。如果您使用 Community Edition Neo4j，不支持这个参数。
  # database: graph.db

  type: {
      # 指定数据来源，设置为 neo4j。
      source: neo4j
      # 指定点数据导入 Nebula Graph 的方式，
      # 可以设置为：client（以客户端形式导入）和 sst（以 SST 文件格式导入）。
      # 关于 SST 文件导入配置，参考文档：导入 SST 文件（https://
      # docs.nebula-graph.com.cn/nebula-exchange/
      # use-exchange/ex-ug-import-sst/）。
      sink: client
  }

  # 指定 Nebula Graph Schema 中标签对应的属性名称，以列表形式列出，
  # 与 tags.fields 列表中的属性名称一一对应，形成映射关系，
  # 多个属性名称之间以英文逗号（,）隔开。
  nebula.fields: [idInt, idString, tdouble, tboolean]

  # 指定源数据中与 Nebula Graph 标签对应的属性名称，
  # 以列表形式列出，多个属性名称之间以英文逗号（,）隔开，
  # 列出的属性名称必须与 exec 中列出的属性名称保持一致。
  fields       : [idInt, idString, tdouble, tboolean]

  # 将源数据中某个属性的值作为 Nebula Graph 点 VID 的来源，
  # 如果属性为 int 或者 long 类型，使用 vertex 设置 VID 列。
  vertex: idInt
  # 如果数据不是 int 类型，则添加 vertex.policy 指定 VID 映射策略，建议设置为 "hash"。
  # vertex {
  #     field: name
  #     policy: "hash"
  #}

  # Spark 的分区数量，默认值为 32。
  partition: 10

  # 单次写入 Nebula Graph 的点数据量，默认值为 256。
  batch: 2000

  # 设置保存导入进度信息的目录，用于断点续传，
  # 如果未设置，表示不启用断点续传。
  check_point_path: "file:///tmp/test"

  # 写入 Cypher 语句，从 Neo4j 数据库中检索打了某种标签的点的属性，并指定别名
  # Cypher 语句不能以英文分号（`;`）结尾。
  exec: "match (n:tagA) return n.idInt as idInt, n.idString as idString, n.tdouble as tdouble, n.tboolean as tboolean order by n.idInt"
  }
  # 如果有多个标签，则参考以上说明添加更多的标签相关信息。
]

# 处理边数据
edges: [
  {
    # 指定 Nebula Graph 中的边类型名称。
    name: edgeAB
    type: {
      # 指定数据来源，设置为 neo4j。
      source: neo4j

      # 指定边数据导入 Nebula Graph 的方式，
      # 可以设置为：client（以客户端形式导入）和 sst（以 SST 文件格式导入）。
      # 关于 SST 文件导入配置，参考文档：导入 SST 文件（https://
      # docs.nebula-graph.com.cn/nebula-exchange/
      # use-exchange/ex-ug-import-sst/）。
      sink: client
    }
```


2021 Vesoft Inc.

```
      # 设置 Neo4j 数据库的信息，包括 IP 地址、端口、用户名和密码。
      server: "bolt://127.0.0.1:7687"
      user: neo4j
      password: neo4j

      # 写入 Cypher 语句，表示从 Neo4j 数据库中查询关系属性。
      # Cypher 语句不能以英文分号（`;`）结尾。
      exec: "match (a:tagA)-[r:edgeAB]->(b:tagB) return a.idInt, b.idInt, r.idInt as idInt, r.idString as idString, r.tdouble as tdouble, r.tboolean as tboolean order by id(r)"

      # 在 fields 里指定 Neo4j 中关系的属性名称，其对应的 value
      # 会作为 Nebula Graph 中 edgeAB 的属性（nebula.fields）值来源
      # fields 和 nebula.fields 里的配置必须一一对应
      # 如果需要指定多个列名称，用英文逗号（,）隔开
      fields: [r.idInt, r.idString, r.tdouble, r.tboolean]
      nebula.fields: [idInt, idString, tdboule, tboolean]

      # 指定源数据中某个属性的值作为 Nebula Graph 边的起始点 VID。
      # 如果属性为 `int` 或者 `long` 类型，使用 source.field 设置起始点 VID 列
      # source.field: field_name
      # 如果不是上述类型的属性，则添加 source.policy 指定 VID 映射策略，建议设置为 "hash"。
      source: {
        field: a.idInt
        policy: "hash"
      }

      # 指定源数据中某个属性的值作为 Nebula Graph 边的终点 VID。
      # 如果属性为 `int` 或者 `long` 类型，使用 target.field 设置终点 VID 列
      # target.field: field_name
      # 如果不是上述类型的属性，则添加 target.policy 指定 VID 映射策略，建议设置为 "hash"。
      target: {
        field: b.idInt
        policy: "hash"
      }

      # 将源数据中某一列数据作为 Nebula Graph 中边的 Rank 值来源。
      ranking: idInt

      # Spark 的分区数量，默认值为 32。
      # 为减轻 Neo4j 的排序压力，将 partition 设置为 1
      partition: 1



      # 单次写入 Nebula Graph 的点数据量，默认值为 256
      batch: 1000

      # 设置保存导入进度信息的目录，用于断点续传。如果未设置，表示不启用断点续传。
      check_point_path: /tmp/test
    }
  ]
}
```

exec 配置说明

在配置源数据的 `tags.exec` 或者 `edges.exec` 参数时，需要写入 Cypher 查询语句。为了保证每次查询结果排序一致，并且为了防止在导入时丢失数据，强烈建议您在 Cypher 查询语句中加入 `ORDER BY` 子句，同时，为了提高数据导入效率，最好选取有索引的属性作为排序的属性。如果没有索引，您也可以观察默认的排序，选择合适的属性用于排序，以提高效率。如果默认的排序找不到规律，您可以根据点或关系的 ID 进行排序，并且将 `partition` 设置为一个尽量小的值，减轻 Neo4j 的排序压力。

说明：使用 `ORDER BY` 子句会延长数据导入的时间。

另外，Exchange 需要在不同 Spark 分区执行不同 `SKIP` 和 `LIMIT` 的 Cypher 语句，所以，在 `tags.exec` 和 `edges.exec` 对应的 Cypher 语句中不能含有 `SKIP` 和 `LIMIT` 子句。

tags.vertex 或 edges.vertex 配置说明

Nebula Graph 在创建点和边时会将 ID 作为唯一主键，如果主键已存在则会覆盖该主键中的数据。所以，假如将某个 Neo4j 属性值作为 Nebula Graph 的 ID，而这个属性值在 Neo4j 中是有重复的，就会导致“重复 ID”，它们对应的数据有且只有一条会存入 Nebula Graph 中，其它的则会被覆盖掉。由于数据导入过程是并发地往 Nebula Graph 中写数据，最终保存的数据并不能保证是 Neo4j 中最新的数据。

check_point_path 配置说明

如果启用了断点续传功能，为避免数据丢失，在断点和续传之间，数据库不应该改变状态，例如不能添加数据或删除数据，同时，不能更改 `partition` 数量配置。

步骤 4. （可选）检查配置文件是否正确

完成配置后，运行以下命令检查配置文件格式是否正确。关于参数的说明，参考 导入命令参数。

2021 Vesoft Inc.

```
$SPARK_HOME/bin/spark-submit --master "local[10]" --class com.vesoft.nebula.tools.importer.Exchange target/exchange-1.x.y.jar -c /path/to/conf/neo4j_application.conf -D
```

步骤 5. 向 Nebula Graph 导入数据

运行以下命令使用 Exchange 将 Neo4j 的数据迁移到 Nebula Graph 中。关于参数的说明，参考 导入命令参数。

```
$SPARK_HOME/bin/spark-submit --master "local[10]" --class com.vesoft.nebula.tools.importer.Exchange target/exchange-1.x.y.jar -c /path/to/conf/neo4j_application.conf
```

> **说明**：JAR 文件版本号以您实际编译得到的 JAR 文件名称为准。

步骤 6. （可选）验证数据

您可以在 Nebula Graph 客户端（例如 Nebula Graph Studio）里执行语句，确认数据是否已导入，例如：

```
GO FROM <tagA_VID> OVER edgeAB;
```

如果返回边终点（`edgeAB._dst`）即表明数据已导入。

您也可以使用 db_dump 工具统计数据是否已经全部导入。详细的使用信息参考 Dump Tool。

步骤 7. （可选）在 Nebula Graph 中重构索引

Exchange 导入数据时，并不会导入 Neo4j 数据库中的索引，所以，导入数据后，您可以在 Nebula Graph 中重新创建并重构索引。详细信息，参考《Nebula Graph Database 手册》。

性能说明

以下为单次测试数据，仅供参考。测试性能如下：

- 导入 100 万个点，耗时 9 s
- 导入 1,000 万条边，耗时 43 s
- 总耗时：52 s

2021 Vesoft Inc.

附录：NEO4J 3.5 COMMUNITY 和 NEBULA GRAPH 1.2.1 对比

Neo4j 和 Nebula Graph 在系统架构、数据模型和访问方式上都有一些差异，部分差异可能会对您使用 Exchange 迁移数据产生影响。下表列出了常见的异同。

2021 Vesoft Inc.

对比项		Neo4j 3.5 Conmunity	Nebula Graph 1.2.1	对使用 Exchange 的影响
系统架构	分布式	仅 Enterprise 支持	支持	无
	数据分片	不支持	支持	无
	开源协议	AGPL	Apache 2.0	无
	开发语言	Java	C++	无
	高可用	不支持	支持	无
数据模型	属性图	是	是	无
	Schema	Schema optional 或者 Schema free	强 Schema	• 必须在 Nebula Graph 中事先创建 Schema • Neo4j 中的 Schema 必须与 Nebula Graph 的保持一致
	点类型/边类型	Label（可以没有 Label，Label 不决定属性 Schema）	Tag/EdgeType（必须至少有一个 Tag，并且与 Schema 对应）	无
	点 ID/唯一主键	点允许无主键（会有多条重复记录）。由内置 `id()` 标识，或者由约束来保证主键唯一。	点必须有唯一标识符，称为 VID。VID 由应用程序生成。	主键相同的重复记录只保留最新的一份。
	属性索引	支持	支持	无法导入索引，必须在导入后再重新创建。
	约束（Constrains）	支持	不支持	不会导入约束。
	事务	支持	不支持	无
查询语句示例	列出所有 labels 或 tags	`MATCH (n) RETURN distinct labels(n);` `call db.labels();`	`SHOW TAGS;`	无
	插入指定类型的点	`CREATE (:Person {age: 16})`	`INSERT VERTEX <tag_name> (prop_name_list) VALUES <vid>: (prop_value_list)`	无
	更新点属性	`SET n.name = V`	`UPDATE VERTEX <vid> SET <update_columns>`	无

	查询指定点的属性	`MATCH (n)`
		`WHERE ID(n) = vid`
		`RETURN properties(n)`
			`FETCH PROP ON <tag_name> <vid>`	无

	查询指定点的某一类关系	`MATCH (n)-[r:edge_type]->() WHERE ID(n) = vid`	`GO FROM <vid> OVER <edge_type>`	无

	两点路径	`MATCH p =(a)-[]->(b)`
		`WHERE ID(a) = a_vid AND ID(b) = b_vid`
		`RETURN p`
			`FIND ALL PATH FROM <a_vid> TO <b_vid> OVER *`	无

最后更新: 2021年4月15日

2021 Vesoft Inc.

导入 SST 文件

Nebula Exchange 能将不同来源的数据转换成 SST 文件后再导入 Nebula Graph 数据库中。本文描述 Exchange 将源数据转换为 SST 文件并导入 Nebula Graph 的实现原理，并提供示例说明如何修改配置文件完成 SST 文件导入操作。

说明：仅 Linux 系统支持导入 SST 文件。

实现方法

Nebula Graph 底层使用 RocksDB 作为键值型存储引擎。RocksDB 是基于磁盘的存储引擎，数据以 Sorted String Table（SSTable）格式存放。SSTable 是一个内部包含了任意长度、排好序的键值对 <key,value> 集合的文件，用于高效地存储大量的键值型数据。

RocksDB 提供了一系列 API 用于创建及导入 SST 文件，有助于您快速导入海量数据。

处理 SST 文件的整个过程主要由 Exchange 的 Reader、sstProcessor 和 sstWriter 完成。整个数据处理过程如下所示：

1. Exchange 的 Reader 从数据源中读取数据。
2. sstProcessor 按照 Nebula Graph 要求的格式生成 SST 文件，存储到本地，并上传到 HDFS。SST 文件主要包含点和边两类数据，其中，
   - 表示点的键包括：分区信息、点 ID（VID）、标签类型信息和标签版本信息。
   - 表示边的键包括：分区信息、起点和终点 ID（`rsc_vid` 和 `dst_vid`）、边类型信息、边排序信息和边版本信息。
   - 对应的值主要包含各个属性键值对序列化信息。
3. SstFileWriter 创建 SST 文件：Exchange 会创建一个 SstFileWriter 对象，然后打开一个文件并插入数据。生成 SST 文件时，行数据必须严格按照增序进行写入。
4. 生成 SST 文件之后，RocksDB 通过 `IngestExternalFile()` 方法将 SST 文件导入到 Nebula Graph 之中。例如：

```
IngestExternalFileOptions ifo;
// Ingest the 2 passed SST files into the DB
Status s = db_->IngestExternalFile({"/home/usr/file1.sst", "/home/usr/file2.sst"}, ifo);
if (!s.ok()) {
  printf("Error while adding file %s and %s, Error %s\n",
         file_path1.c_str(), file_path2.c_str(), s.ToString().c_str());
  return 1;
}
```

调用 `IngestExternalFile()` 方法时，RocksDB 默认会将文件拷贝到数据目录，并且阻塞 RocksDB 写入操作。如果 SST 文件中的键范围覆盖了 Memtable 键的范围，则将 Memtable 落盘（flush）到硬盘。将 SST 文件放置在 LSM 树最优位置后，为文件分配一个全局序列号，并打开写操作。

2021 Vesoft Inc.

使用示例

不同来源的数据，导入 Nebula Graph 的操作与客户端形式导入操作基本相同，但是有以下差异：

- 环境里必须部署 HDFS。
- 在配置文件中，必须做以下修改：
  - 源数据的标签和边类型配置：`tags.type.sink` 和 `edges.type.sink` 必须配置为 `sst`。
  - Nebula Graph 相关配置里，需要添加 Nebula Graph 数据库 Meta 服务的 IP 地址和端口，并添加 SST 文件在本地和 HDFS 的存储路径。

```
# Nebula Graph 相关配置
nebula:  {
  address: {
    # 添加 Nebula Graph 数据库 Graph 服务的 IP 地址和端口
    graph: ["127.0.0.1:3699"]
    # 添加 Nebula Graph 数据库 Meta 服务的 IP 地址和端口
    meta:["127.0.0.1:45500"]
  }
  user: user
  pswd: password
  space: test
  path:{
    # 指定 SST 文件保存到本地的路径
    local:/Users/example/path
    # 指定上传 SST 文件的 HDFS 路径
    remote:/example/
  }

  connection {
    timeout: 3000
    retry: 3
  }

  execution {
    retry: 3
  }

  error: {
    max: 32
    output: /tmp/errors
  }

  rate: {
    limit: 64M
    timeout: 1000
  }
}
```

详细描述请参考不同数据源的操作示例：

- 导入 Neo4j 数据
- 导入 HIVE 数据
- 导入 CSV 文件数据
- 导入 JSON 文件数据
- 导入 HBase 数据[doc_TODO]
- 导入 HIVE 数据[doc_TODO]
- 导入 Kafka 数据[doc_TODO]
- 导入 MySQL 数据[doc_TODO]

最后更新: 2021年4月7日

2021 Vesoft Inc.

导入 HIVE 数据

本文以一个示例说明如何使用 Exchange 将存储在 HIVE 的数据导入 Nebula Graph。

数据集

本文以美国 Stanford Network Analysis Platform (SNAP) 提供的 Social Network: MOOC User Action Dataset 以及由公开网络上获取的不重复的 97 个课程名称作为示例数据集，包括：

- 两类点（`user` 和 `course`），共计 7,144 个点。
- 一种关系（`action`），共计 411,749 条边。

详细的数据集，您可以从 nebula-web-docker 仓库中下载。

在本示例中，该数据集已经存入 HIVE 中名为 `mooc` 的数据库中，以 `users`、`courses` 和 `actions` 三个表存储了所有点和边的信息。以下为各个表的结构。

```
scala> sql("describe mooc.users").show
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|  userid|   bigint|   null|
+--------+---------+-------+

scala> sql("describe mooc.courses").show
+----------+---------+-------+
|  col_name|data_type|comment|
+----------+---------+-------+
|  courseid|   bigint|   null|
|coursename|   string|   null|
+----------+---------+-------+

scala> sql("describe mooc.actions").show
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|actionid|   bigint|   null|
|   srcid|   bigint|   null|
|   dstid|   string|   null|
|duration|   double|   null|
|feature0|   double|   null|
|feature1|   double|   null|
|feature2|   double|   null|
|feature3|   double|   null|
|   label|  boolean|   null|
+--------+---------+-------+
```

说明：Hive 的 `bigint` 与 Nebula Graph 的 `int` 对应。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
  - CPU：1.7 GHz Quad-Core Intel Core i7
  - 内存：16 GB
- Spark：2.4.7，单机版
- Hadoop：2.9.2，伪分布式部署
- HIVE：2.3.7，Hive Metastore 数据库为 MySQL 8.0.22
- Nebula Graph：V1.2.0，使用 Docker Compose 部署。详细信息，参考 使用 Docker Compose 部署 Nebula Graph

2021 Vesoft Inc.

前提条件

开始导入数据之前，您需要确认以下信息：

- 已经完成 Exchange 编译。详细信息，参考 编译 Exchange。本示例中使用 Exchange v1.2.1。
- 已经安装 Spark。
- 已经安装并开启 Hadoop 服务，并已启动 Hive Metastore 数据库（本示例中为 MySQL）。
- 已经部署并启动 Nebula Graph，并获取：
  - Graph 服务、Meta 服务所在机器的 IP 地址和端口信息。
  - Nebula Graph 数据库的拥有写权限的用户名及其密码。
- 在 Nebula Graph 中创建图数据模式（Schema）所需的所有信息，包括标签和边类型的名称、属性等。

操作步骤

步骤 1. 在 Nebula Graph 中创建 Schema

按以下步骤在 Nebula Graph 中创建 Schema：

1. 确认 Schema 要素：Nebula Graph 中的 Schema 要素如下表所示。

要素	名称	属性
标签（Tag）	`user`	`userId int`
标签（Tag）	`course`	`courseId int, courseName string`
边类型（Edge Type）	`action`	`actionId int, duration double, label bool, feature0 double, feature1 double, feature2 double, feature3 double`

2. 在 Nebula Graph 里创建一个图空间 **hive**，并创建一个 Schema，如下所示。

```
-- 创建图空间
CREATE SPACE hive(partition_num=10, replica_factor=1);

-- 选择图空间 hive
USE hive;

-- 创建标签 user
CREATE TAG user(userId int);

-- 创建标签 course
CREATE TAG course(courseId int, courseName string);

-- 创建边类型 action
CREATE EDGE action (actionId int, duration double, label bool, feature0 double, feature1 double, feature2 double, feature3 double);
```

关于 Nebula Graph 构图的更多信息，参考《Nebula Graph Database 手册》的 快速开始。

步骤 2. 使用 Spark SQL 确认 HIVE SQL 语句

启动 spark-shell 环境后，依次运行以下语句，确认 Spark 能读取 HIVE 中的数据。

```
scala> sql("select userid from mooc.users").show
scala> sql("select courseid, coursename from mooc.courses").show
scala> sql("select actionid, srcid, dstid, duration, feature0, feature1, feature2, feature3, label from mooc.actions").show
```

以下为 `mooc.actions` 表中读出的结果。

```
+--------+-----+--------------------+--------+------------+------------+-----------+-----------+-----+
|actionid|srcid|               dstid|duration|    feature0|    feature1|   feature2|   feature3|label|
+--------+-----+--------------------+--------+------------+------------+-----------+-----------+-----+
|       0|    0|Environmental Dis...|     0.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       1|    0|  History of Ecology|     6.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       2|    0|      Women in Islam|    41.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       3|    0|  History of Ecology|    49.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       4|    0|      Women in Islam|    51.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       5|    0|Legacies of the A...|    55.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       6|    0|          ITP Core 2|    59.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       7|    0|The Research Pape...|    62.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|       8|    0|        Neurobiology|    65.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
```


2021 Vesoft Inc.

```
|       9|    0|           Wikipedia|   113.0|-0.319991479|-0.435701433|1.108826104|12.77723482|false|
|      10|    0|Media History and...|   226.0|-0.319991479|-0.435701433|0.607804941|149.4512115|false|
|      11|    0|             WIKISOO|   974.0|-0.319991479|-0.435701433|1.108826104|3.344522776|false|
|      12|    0|Environmental Dis...|  1000.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      13|    0|             WIKISOO|  1172.0|-0.319991479|-0.435701433|1.108826104|1.136866766|false|
|      14|    0|      Women in Islam|  1182.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      15|    0|  History of Ecology|  1185.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      16|    0|Human Development...|  1687.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      17|    1|Human Development...|  7262.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      18|    1|  History of Ecology|  7266.0|-0.319991479|-0.435701433|0.106783779|-0.06730924|false|
|      19|    1|      Women in Islam|  7273.0|-0.319991479|-0.435701433|0.607804941|0.936170765|false|
+--------+-----+--------------------+--------+------------+------------+-----------+-----------+-----+
only showing top 20 rows
```

步骤 3. 修改配置文件

完成 Exchange 编译后，进入 `nebula-java/tools/exchange` 目录，根据 `target/classes/application.conf` 文件修改 HIVE 数据源相关的配置文件。在本示例中，文件被重命名为 `hive_application.conf`。以下仅详细说明点和边数据的配置信息，本次示例中未使用的配置项已被注释，但是提供了配置说明。Spark 和 Nebula Graph 相关配置，参考 Spark 参数和 Nebula Graph 参数。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: Spark Writer
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores {
      max: 16
    }
  }
  # Nebula Graph 相关配置
  nebula: {
    address:{
      # 以下为 Nebula Graph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"
      # 不同地址之间以英文逗号 (,) 隔开
      graph:["127.0.0.1:3699"]
      meta:["127.0.0.1:45500"]
    }
    # 填写的账号必须拥有 Nebula Graph 相应图空间的写数据权限
    user: user
    pswd: password
    # 填写 Nebula Graph 中需要写入数据的图空间名称
    space: hive
    connection {
      timeout: 3000
      retry: 3
    }
    execution {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理标签
  tags: [
    # 设置标签相关信息
    {
      # Nebula Graph 中对应的标签名称。
      name: user
      type: {
        # 指定数据源文件格式，设置为 hive。
        source: hive
        # 指定点数据导入 Nebula Graph 的方式，
        # 可以设置为：client（以客户端形式导入）和 sst（以 SST 文件格式导入）。
        # 关于 SST 文件导入配置，参考文档：导入 SST 文件（https://
        # docs.nebula-graph.com.cn/nebula-exchange/
        # use-exchange/ex-ug-import-sst/）。
        sink: client
      }

      # 设置读取数据库 mooc 中 users 表数据的 SQL 语句
      exec: "select userid from mooc.users"

      # 在 fields 里指定 users 表中的列名称，其对应的 value
      # 会作为 Nebula Graph 中指定属性 userId (nebula.fields) 的数据源
      # fields 和 nebula.fields 里的配置必须一一对应
      # 如果需要指定多个列名称，用英文逗号 (,) 隔开
      fields: [userid]
      nebula.fields: [userId]
```


2021 Vesoft Inc.

```
      # 指定表中某一列数据为 Nebula Graph 中点 VID 的来源。
      # vertex.field 的值必须与上述 fields 中的列名保持一致。
      # 如果数据不是 int 类型，则添加 vertex.policy 指定 VID 映射策略，建议设置为 "hash"，参考以下 course 标签的设置。
      vertex: userid

      # 单次写入 Nebula Graph 的最大点数据量。
      batch: 256

      # Spark 分区数量
      partition: 32

      # isImplicit 的设置说明参考：https://github.com/vesoft-inc/nebula-java/
      # blob/v1.0/tools/exchange/src/main/resources/application.conf
      isImplicit: true
    }
    {
      name: course
      type: {
        source: hive
        sink: client
      }
      exec: "select courseid, coursename from mooc.courses"
      fields: [courseid, coursename]
      nebula.fields: [courseId, courseName]

      # 指定表中某一列数据为 Nebula Graph 中点 VID 的来源。
      # vertex.field 的值必须与上述 fields 中的列名保持一致。
      # 如果数据不是 int 类型，则添加 vertex.policy 指定 VID 映射策略，建议设置为 "hash"。
      vertex: {
        field: coursename
        policy: "hash"
      }
      batch: 256
      partition: 32
      isImplicit: true
    }

  ]

  # 处理边数据
  edges: [
    # 设置边类型 action 相关信息
    {
      # Nebula Graph 中对应的边类型名称。
      name: action

      type: {
        # 指定数据源文件格式，设置为 hive。
        source: hive

        # 指定边数据导入 Nebula Graph 的方式，
        # 可以设置为：client（以客户端形式导入）和 sst（以 SST 文件格式导入）。
        # 关于 SST 文件导入配置，参考文档：导入 SST 文件（https://
        # docs.nebula-graph.com.cn/nebula-exchange/
        # use-exchange/ex-ug-import-sst/）。
        sink: client
      }

      # 设置读取数据库 mooc 中 actions 表数据的 SQL 语句
      exec: "select actionid, srcid, dstid, duration, feature0, feature1, feature2, feature3, label from mooc.actions"

      # 在 fields 里指定 actions 表中的列名称，其对应的 value
      # 会作为 Nebula Graph 中 action 的属性（nebula.fields）值来源
      # fields 和 nebula.fields 里的配置必须一一对应
      # 如果需要指定多个列名称，用英文逗号（,）隔开
      fields: [actionid, duration, feature0, feature1, feature2, feature3, label]
      nebula.fields: [actionId, duration, feature0, feature1, feature2, feature3, label]

      # 在 source 里，将 actions 表中某一列作为边起点数据源
      # 在 target 里，将 actions 表中某一列作为边终点数据源
      # 如果数据源是 int 或 long 类型，直接指定列名
      # 如果数据源不是 int 类型，则添加 vertex.policy 指定 VID 映射策略，建议设置为 "hash"
      source: srcid
      target: {
        field: dstid
        policy: "hash"
      }

      # 单次写入 Nebula Graph 的最大点数据量。
      batch: 256

      # Spark 分区数量
      partition: 32
    }
  ]
}
```

步骤 4. （可选）检查配置文件是否正确

完成配置后，运行以下命令检查配置文件格式是否正确。关于参数的说明，参考 导入命令参数。

2021 Vesoft Inc.

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/hive_application.conf -h -D
```

步骤 5. 向 Nebula Graph 导入数据

运行以下命令将 HIVE 中的数据导入到 Nebula Graph 中。关于参数的说明，参考 导入命令参数。

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/hive_application.conf -h
```

步骤 6. （可选）验证数据

您可以在 Nebula Graph 客户端（例如 Nebula Graph Studio）里执行语句，确认数据是否已导入，例如：

```
GO FROM 1 OVER action;
```

如果返回边终点（`action._dst`）即表明数据已导入。

您也可以使用 db_dump 工具统计数据是否已经全部导入。详细的使用信息参考 Dump Tool。

步骤 7. （可选）在 Nebula Graph 中重构索引

导入数据后，您可以在 Nebula Graph 中重新创建并重构索引。详细信息，参考《Nebula Graph Database 手册》。

最后更新: 2021年4月15日

2021 Vesoft Inc.

导入 CSV 文件数据

本文以一个示例说明如何使用 Exchange 将存储在 HDFS 上的 CSV 文件数据导入 Nebula Graph。

如果您要向 Nebula Graph 导入本地 CSV 文件，参考 CSV 文件导入示例。

数据集

本文以美国 Stanford Network Analysis Platform (SNAP) 提供的 Social Network: MOOC User Action Dataset 以及由公开网络上获取的不重复的 97 个课程名称作为示例数据集，包括：

- 两类点（`user` 和 `course`），共计 7,144 个点。
- 一种关系（`action`），共计 411,749 条边。

详细的数据集，您可以从 nebula-web-docker 仓库中下载。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
  - CPU：1.7 GHz Quad-Core Intel Core i7
  - 内存：16 GB
- Spark：2.3.0，单机版
- Hadoop：2.9.2，伪分布式部署
- Nebula Graph：V1.2.1，使用 Docker Compose 部署。详细信息，参考 使用 Docker Compose 部署 Nebula Graph

前提条件

开始导入数据之前，您需要确认以下信息：

- 已经完成 Exchange 编译。详细信息，参考 编译 Exchange。本示例中使用 Exchange v1.2.1。
- 已经安装 Spark。
- 已经安装并开启 Hadoop 服务。
- 已经部署并启动 Nebula Graph，并获取：
  - Graph 服务、Meta 服务所在机器的 IP 地址和端口信息。
  - Nebula Graph 数据库的拥有写权限的用户名及其密码。
- 在 Nebula Graph 中创建图数据模式（Schema）所需的所有信息，包括标签和边类型的名称、属性等。

2021 Vesoft Inc.

操作步骤

步骤 1. 在 Nebula Graph 中创建 Schema

分析 CSV 文件中的数据，按以下步骤在 Nebula Graph 中创建 Schema：

1. 确认 Schema 要素：Nebula Graph 中的 Schema 要素如下表所示。

要素	名称	属性
标签（Tag）	`user`	`userId int`
标签（Tag）	`course`	`courseId int, courseName string`
边类型（Edge Type）	`action`	`actionId int, duration double, label bool, feature0 double, feature1 double, feature2 double, feature3 double`

2. 在 Nebula Graph 里创建一个图空间 **csv**，并创建一个 Schema，如下所示。

```
-- 创建图空间
CREATE SPACE csv(partition_num=10, replica_factor=1);

-- 选择图空间 csv
USE csv;

-- 创建标签 user
CREATE TAG user(userId int);

-- 创建标签 course
CREATE TAG course(courseId int, courseName string);

-- 创建边类型 action
CREATE EDGE action (actionId int, duration double, label bool, feature0 double, feature1 double, feature2 double, feature3 double);
```

关于 Nebula Graph 构图的更多信息，参考《Nebula Graph Database 手册》的 快速开始。

步骤 2. 处理 CSV 文件

确认以下信息：

1. CSV 文件已经根据 Schema 作了处理。详细操作请参考 Nebula Graph Studio 快速开始。

   说明：Exchange 支持上传有表头或者无表头的 CSV 文件。

2. CSV 文件必须存储在 HDFS 中，并已获取文件存储路径。

步骤 3. 修改配置文件

完成 Exchange 编译后，进入 `nebula-java/tools/exchange` 目录，根据 `target/classes/application.conf` 文件修改 CSV 数据源相关的配置文件。在本示例中，文件被重命名为 `csv_application.conf`。以下仅详细说明点和边数据的配置信息，本次示例中未使用的配置项已被注释，但是提供了配置说明。Spark 和 Nebula Graph 相关配置，参考 Spark 参数和 Nebula Graph 参数。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: Spark Writer
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores {
      max: 16
    }
  }
  # Nebula Graph 相关配置
  nebula: {
    address:{
      # 以下为 Nebula Graph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"
      # 不同地址之间以英文逗号 (,) 隔开
      graph:["127.0.0.1:3699"]
      meta:["127.0.0.1:45500"]
    }
    # 填写的账号必须拥有 Nebula Graph 相应图空间的写数据权限
    user: user
    pswd: password
    # 填写 Nebula Graph 中需要写入数据的图空间名称
    space: csv
```


2021 Vesoft Inc.

```
    connection {
      timeout: 3000
      retry: 3
    }
    execution {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理标签
  tags: [
    # 设置标签 course 相关信息
    {
      # Nebula Graph 中对应的标签名称。
      name: course
      type: {
        # 指定数据源文件格式，设置为 csv。
        source: csv
        # 指定点数据导入 Nebula Graph 的方式，
        # 可以设置为：client（以客户端形式导入）和 sst（以 SST 文件格式导入）。
        # 关于 SST 文件导入配置，参考文档：导入 SST 文件（https://
        # docs.nebula-graph.com.cn/nebula-exchange/
        # use-exchange/ex-ug-import-sst/）。
        sink: client
      }
      # CSV 文件所在的 HDFS 路径，String 类型，必须以 hdfs:// 开头。
      path: "hdfs://namenode_ip:port/path/to/course.csv"

      # 如果 CSV 文件里不带表头，则写入 [_c0, _c1, _c2, ... _cn]，
      # 表示 CSV 文件中的数据列名，作为 course 各属性值来源。
      # 如果 CSV 文件里有表头，则写入各列名。
      # fields 与 nebula.fields 的顺序必须一一对应。
      fields: [_c0, _c1]

      # 设置 Nebula Graph 中与 CSV 文件各列对应的属性名称，
      # fields 与 nebula.fields 的顺序必须一一对应。
      nebula.fields: [courseId, courseName]

      # Exchange 1.2.1 添加了 csv.fields 参数：
      # 如果配置了 csv.fields，无论 CSV 文件是否有表头，
      # fields 的配置必须与 csv.fields 的配置保持一致。
      # csv.fields: [courseId, courseName]

      # 指定 CSV 中的某一列数据为 Nebula Graph 中点 VID 的来源。
      # vertex.field 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
      # 如果数据不是 int 类型，则添加 vertex.policy 指定 VID 映射策略，建议设置为 "hash"。
      vertex: {
        field: _c1,
        policy: "hash"
      }

      # 标明数据源中数据分隔方式，默认为英文逗号（,）。
      separator: ","

      # 如果 CSV 文件中有表头，header 设置为 true。
      # 如果 CSV 文件中没有表头，header 设置为 false（默认值）。
      header: false

      # 单次写入 Nebula Graph 的最大点数据量。
      batch: 256

      # Spark 分区数量
      partition: 32

      # isImplicit 的设置说明参考：https://github.com/vesoft-inc/nebula-java/
      # blob/v1.0/tools/exchange/src/main/resources/application.conf
      isImplicit: true
    }

    # 设置标签 user 相关信息
    {
      name: user
      type: {
        source: csv
        sink: client
      }
      path: "hdfs://namenode_ip:port/path/to/user.csv"

      # Exchange 1.2.1 添加了 csv.fields 参数
      # 如果 CSV 文件里不带表头，但是配置了 csv.fields，
      # 则 fields 的配置必须与 csv.fields 保持一致，
      # Exchange 会将 csv.fields 里的设置作为表头。
      # fields 与 nebula.fields 的顺序必须一一对应。
      fields: [userId]

      # 设置 Nebula Graph 中与 CSV 文件各列对应的属性名称，
      # 必须与 fields 或者 csv.fields 的顺序一一对应。
```


2021 Vesoft Inc.

```
      nebula.fields: [userId]

      # 如果配置了 csv.fields，无论 CSV 文件是否有表头，
      # 均以这个参数指定的名称作为表头，
      # fields 的配置必须与 csv.fields 的配置保持一致，
      # 同时，vertex 的设置必须与 csv.fields 的设置相同。
      csv.fields: [userId]

      # vertex 的值必须与 fields 或者 csv.fields 中相应的列名保持一致。
      vertex: userId
      separator: ","
      header: false
      batch: 256
      partition: 32

      # isImplicit 设置说明，详见 https://github.com/vesoft-inc/nebula-java/blob
      # /v1.0/tools/exchange/src/main/resources/application.conf
      isImplicit: true
    }
  ]
  # 处理边数据
  edges: [
    # 设置边类型 action 相关信息
    {
      # Nebula Graph 中对应的边类型名称。
      name: action
      type: {
        # 指定数据源文件格式，设置为 csv。
        source: csv

        # 指定边数据导入 Nebula Graph 的方式，
        # 可以设置为：client（以客户端形式导入）和 sst（以 SST 文件格式导入）。
        # 关于 SST 文件导入配置，参考文档：导入 SST 文件（https://
        # docs.nebula-graph.com.cn/nebula-exchange/
        # use-exchange/ex-ug-import-sst/）。
        sink: client
      }

      # 指定 CSV 文件所在的 HDFS 路径，String 类型，必须以 hdfs:// 开头。
      path: "hdfs://namenode_ip:port/path/to/actions.csv"

      # 如果 CSV 文件里不带表头，
      # 则写入 [_c0, _c1, _c2, ... _cn]，
      # 依次表示 CSV 文件中各数据列，作为 action 各属性值来源。
      # 如果 CSV 文件里有表头，则写入各列名。
      # fields 与 nebula.fields 的顺序必须一一对应。
      fields: [_c0, _c3, _c4, _c5, _c6, _c7, _c8]

      # Nebula Graph 中 action 的属性名称，必须与 fields 里的列顺序一一对应。
      nebula.fields: [actionId, duration, feature0, feature1, feature2, feature3, label]

      # Exchange 1.2.1 添加了 csv.fields 参数：
      # 如果配置了 csv.fields，无论 CSV 文件是否有表头，
      # 均以这个参数指定的名称作为表头，
      # fields 的配置必须与 csv.fields 的配置保持一致。
      # csv.fields: [actionId, duration, feature0, feature1, feature2, feature3, label]

      # 边起点和边终点 VID 数据来源，
      # 如果不是 int 类型数据，则添加 policy 指定 VID 映射策略，建议设置为 "hash"。
      source: _c1
      target: {
        field: _c2
        policy: "hash"
      }

      # 标明数据源中数据分隔方式，默认为英文逗号（,）。
      separator: ","

      # 如果 CSV 文件中有表头，header 设置为 true。
      # 如果 CSV 文件中没有表头，header 设置为 false（默认）。
      header: false

      # 单次向 Nebula Graph 写入的最大边数据量。
      batch: 256

      # 设置 Spark 分区数量。
      partition: 32
      isImplicit: true
    }
  ]
  # 如果还有其他边，再添加其他边类型相关的设置。
}
```

步骤 4. （可选）检查配置文件是否正确

完成配置后，运行以下命令检查配置文件格式是否正确。关于参数的说明，参考 导入命令参数。

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/csv_application.conf -D
```


2021 Vesoft Inc.

步骤 5. 向 Nebula Graph 导入数据

运行以下命令将 CSV 文件数据导入到 Nebula Graph 中。关于参数的说明，参考 导入命令参数。

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.2.1.jar -c /path/to/conf/csv_application.conf
```

步骤 6. （可选）验证数据

您可以在 Nebula Graph 客户端（例如 Nebula Graph Studio）里执行语句，确认数据是否已导入，例如：

```
GO FROM 1 OVER action;
```

如果返回边终点（`action._dst`）即表明数据已导入。

您也可以使用 db_dump 工具统计数据是否已经全部导入。详细的使用信息参考 Dump Tool。

步骤 7. （可选）在 Nebula Graph 中重构索引

导入数据后，您可以在 Nebula Graph 中重新创建并重构索引。详细信息，参考《Nebula Graph Database 手册》。

最后更新: 2021年4月15日

2021 Vesoft Inc.

导入 JSON 文件数据

本文以一个示例说明如何使用 Exchange 将存储在 HDFS 上的 JSON 文件数据导入 Nebula Graph。

数据集

本示例所用 JSON 文件（test.json）结构为：`{"source":int, "target":int, "likeness":double}`，表示 `source` 与 `target` 之间一种 `like` 关系。共计 21,645 条数据。

以下为部分示例数据：

```
{"source":53802643,"target":87847387,"likeness":0.34}
{"source":29509860,"target":57501950,"likeness":0.40}
{"source":97319348,"target":50240344,"likeness":0.77}
{"source":94295709,"target":8189720,"likeness":0.82}
{"source":78707720,"target":53874070,"likeness":0.98}
{"source":23399562,"target":20136097,"likeness":0.47}
```

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
  - CPU：1.7 GHz Quad-Core Intel Core i7
  - 内存：16 GB
- Spark：2.3.0，单机版
- Hadoop：2.9.2，伪分布式部署
- Nebula Graph：V1.2.1，使用 Docker Compose 部署。详细信息，参考 使用 Docker Compose 部署 Nebula Graph

前提条件

开始迁移数据之前，您需要确认以下信息：

- 已经完成 Exchange 编译。详细信息，参考 编译 Exchange。本示例中使用 Exchange v1.0.1。
- 已经安装 Spark。
- 已经安装并开启 Hadoop 服务。
- 已经部署并启动 Nebula Graph，并获取：
  - Graph 服务、Meta 服务所在机器的 IP 地址和端口信息。
  - Nebula Graph 数据库的拥有写权限的用户名及其密码。
- 在 Nebula Graph 中创建图数据模式（Schema）所需的所有信息，包括标签和边类型的名称、属性等。

2021 Vesoft Inc.

操作步骤

步骤 1. 在 Nebula Graph 中创建 Schema

分析 JSON 文件中的数据，按以下步骤在 Nebula Graph 中创建 Schema：

1. 确认 Schema 要素：Nebula Graph 中的 Schema 要素如下表所示。

要素	名称	属性
标签（Tag）	`source`	`srcId int`
标签（Tag）	`target`	`dstId int`
边类型（Edge Type）	`like`	`likeness double`

2. 在 Nebula Graph 里创建一个图空间 **json**，并创建一个 Schema，如下所示。

```
-- 创建图空间
CREATE SPACE json (partition_num=10, replica_factor=1);

-- 选择图空间 json
USE json;

-- 创建标签 source
CREATE TAG source (srcId int);

-- 创建标签 target
CREATE TAG target (dstId int);

-- 创建边类型 like
CREATE EDGE like (likeness double);
```

关于 Nebula Graph 构图的更多信息，参考《Nebula Graph Database 手册》的 快速开始。

步骤 2. 处理 JSON 文件

分别创建点和边数据 JSON 文件。同时，JSON 文件必须存储在 HDFS 里，并获取文件存储路径。

> **说明**：本示例中仅使用一个 JSON 文件同时写入点和边数据，其中，表示 source 和 target 的部分点数据是重复的，所以，在写入数据时，这些点会被重复写入。向 Nebula Graph 插入点或边时，允许重复插入，但是最后读取时以最后一次写入的数据为准，所以，并不影响使用。在实际使用时，最好分别创建点和边数据文件，提高数据写入速度。

步骤 3. 修改配置文件

完成 Exchange 编译后，进入 `nebula-java/tools/exchange` 目录，根据 `target/classes/application.conf` 文件修改 果断 数据源相关的配置文件。在本示例中，文件被重命名为 `json_application.conf`。以下配置文件中提供了 JSON 源数据所有配置项。本次示例中未使用的配置项已被注释，但是提供了配置说明。Spark 和 Nebula Graph 相关配置，参考 Spark 参数和 Nebula Graph 参数。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: Spark Writer
    }

    driver: {
      cores: 1
      maxResultSize: 1G
    }

    cores {
      max: 16
    }
  }

  # Nebula Graph 相关配置
  nebula: {
    address:{
      # 以下为 Nebula Graph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"
      # 不同地址之间以英文逗号 (,) 隔开
      graph:["127.0.0.1:3699"]
      meta:["127.0.0.1:45500"]
    }
    # 填写的账号必须拥有 Nebula Graph 相应图空间的写数据权限
    user: user
    pswd: password

    # 填写 Nebula Graph 中需要写入数据的图空间名称
```


2021 Vesoft Inc.

```
  space: json

  connection {
    timeout: 3000
    retry: 3
  }

  execution {
    retry: 3
  }

  error: {
    max: 32
    output: /tmp/errors
  }

  rate: {
    limit: 1024
    timeout: 1000
  }
}

# 处理标签
tags: [
  # 设置标签 source 相关信息
  {
    # 设置为 Nebula Graph 中对应的标签名称
    name: source
    type: {
      # 指定数据源文件格式，设置为 json。
      source: json

      # 指定标签数据导入 Nebula Graph 的方式，
      # 可以设置为：client（以客户端形式导入）和 sst（以 SST 文件格式导入）。
      # 关于 SST 文件导入配置，参考文档：导入 SST 文件。
      sink: client
    }

    # JSON 文件所在的 HDFS 路径，String 类型，必须以 hdfs:// 开头。
    path: "hdfs://namenode_ip:port/path/to/test.json"

    # 在 fields 里指定 JSON 文件中 key 名称，其对应的 value
    # 会作为 Nebula Graph 中指定属性 srcId 的数据源
    # 如果需要指定多个值，用英文逗号（,）隔开
    fields: ["source"]
    nebula.fields: ["srcId"]

    # 将 JSON 文件中某个 key 对应的值作为 Nebula Graph 中点 VID 的来源
    # 如果 VID 源数据不是 int 类型，则使用以下内容来代替 vertex 的设置，在其中指定 VID 映射策略，建议设置为 "hash"。
    # vertex: {
    #   field: key_name_in_json
    #   policy: "hash"
    # }
    vertex: source

    batch: 256
    partition: 32

    # isImplicit 设置说明，详见 https://github.com/vesoft-inc/
    # nebula-java/blob/v1.0/tools/exchange/src/main/resources/
    # application.conf
    isImplicit: true
  }
  # 设置标签 target 相关信息
  {
    name: target
    type: {
      source: json
      sink: client
    }
    path: "hdfs://namenode_ip:port/path/to/test.json"
    fields: ["target"]
    nebula.fields: ["dstId"]
    vertex: "target"
    batch: 256
    partition: 32
    isImplicit: true
  }
# 如果还有其他标签，参考以上配置添加
]

# 处理边数据
edges: [
  # 设置边类型 like 相关信息
  {
    # Nebula Graph 中对应的边类型名称。
    name: like
    type: {
      # 指定数据源文件格式，设置为 json。
      source: json

      # 指定边数据导入 Nebula Graph 的方式，
      # 可以设置为：client（以客户端形式导入）和 sst（以 SST 文件格式导入）。
      # 关于 SST 文件导入配置，参考文档：导入 SST 文件（https://
```


2021 Vesoft Inc.

```
        # docs.nebula-graph.com.cn/nebula-exchange/
        # use-exchange/ex-ug-import-sst/) 。
        sink: client
      }

      # 指定 JSON 文件所在的 HDFS 路径，String 类型，必须以 hdfs:// 开头。
      path: "hdfs://namenode_ip:port/path/to/test.json"

      # 在 fields 里指定 JSON 文件中 key 名称，其对应的 value
      # 会作为 Nebula Graph 中指定属性 likeness 的数据源
      # 如果需要指定多个值，用英文逗号（,）隔开
      fields: ["likeness"]
      nebula.fields: ["likeness"]

      # 将 JSON 文件中某两个 key 对应的值作为 Nebula Graph 中边起点和边终点 VID 的来源
      # 如果 VID 源数据不是 int 类型，则使用以下内容来代替 source
      # 和/或 target 的设置，在其中指定 VID 映射策略，建议设置为 "hash"。
      # source: {
      #   field: key_name_in_json
      #   policy: "hash"
      # }
      # target: {
      #   field: key_name_in_json
      #   policy: "hash"
      # }
      source: "source"
      target: "target"

      batch: 256
      partition: 32
      isImplicit: true
    }
    # 如果还有其他边类型，参考以上配置添加
  ]
}
```

步骤 4. （可选）检查配置文件是否正确

完成配置后，运行以下命令检查配置文件格式是否正确。关于参数的说明，参考 导入命令参数。

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.0.1.jar -c /path/to/conf/json_application.conf -D
```

步骤 5. 向 Nebula Graph 导入数据

运行以下命令将 JSON 文件数据导入 Nebula Graph 中。关于参数的说明，参考 导入命令参数。

```
$SPARK_HOME/bin/spark-submit --master "local" --class com.vesoft.nebula.tools.importer.Exchange /path/to/exchange-1.0.1.jar -c /path/to/conf/json_application.conf
```

步骤 6. （可选）验证数据

您可以在 Nebula Graph 客户端（例如 Nebula Graph Studio）里执行语句，确认数据是否已导入，例如：

```
GO FROM 53802643 OVER like;
```

如果返回边终点（`like._dst`）即表明数据已导入。

您也可以使用 db_dump 工具统计数据是否已经全部导入。详细的使用信息参考 Dump Tool。

步骤 7. （可选）在 Nebula Graph 中重构索引

导入数据后，您可以在 Nebula Graph 中重新创建并重构索引。详细信息，参考《Nebula Graph Database 手册》。

最后更新: 2021年4月9日

2021 Vesoft Inc.

5.1.4 参数说明

Spark 参数

在使用 Exchange 导入数据时，您可以根据需要设置 Spark 参数，详细信息的 Spark 参数信息，参考《Apache Spark 文档》。下表仅提供部分参数的配置说明。实际应用时的参数设置，参考不同来源数据的 操作示例。

参数	默认值	数据类型	是否必需	说明
`spark.app.name`	Spark Writer	`string`	否	Spark Driver Program 名称。
`spark.driver.cores`	1	`int`	否	Driver 进程的核数，仅适用于集群模式。
`spark.driver.maxResultSize`	1G	`string`	否	每个 Spark 操作（例如收集）中所有分区的序列化结果的上限（以字节为单位）。最小值为 1M，设为 0 则表示无限制上限。
`spark.cores.max`	无	`int`	否	当以“粗粒度”共享模式在独立部署集群或 Mesos 集群上运行时，跨集群（而非从每台计算机）请求应用程序的最大 CPU 核数。如果未设置，则默认值为 Spark 的独立集群管理器上的 `spark.deploy.defaultCores` 或 Mesos 上的 infinite（所有可用的内核）。

最后更新: 2021年4月7日

2021 Vesoft Inc.

Nebula Graph 相关参数

下表列出在使用 Exchange 导入数据时您需要设置的 Nebula Graph 相关参数。实际应用时的参数设置，参考不同来源数据的 操作示例。

参数	默认值	数据类型	是否必需	说明
`nebula.address.graph`	无	`list[string]`	是	Nebula Graph 图数据库 Graph 服务的地址列表。如果有多个地址，以英文逗号（,）分隔。格式为 `"ip1:port","ip2:port","ip3:port"` 。
`nebula.address.meta`	无	`list[string]`	是	Nebula Graph 图数据库 Meta 服务的地址列表。如果有多个地址，以英文逗号（,）分隔。格式为 `"ip1:port","ip2:port","ip3:port"` 。
`nebula.user`	`user`	`string`	是	数据库用户名，默认为 `user` 。如果 Nebula Graph 启用了身份认证： - 如果未创建不同用户，使用 `root` 。 - 如果已经创建了不同的用户并且分配了指定空间的角色，则使用对该空间拥有写操作权限的用户。
`nebula.pswd`	`password`	`string`	是	数据库用户名对应的密码，默认 `user` 的密码为 `password` 。如果 Nebula Graph 启用了身份认证： - 使用 `root` 时，密码为 `nebula` 。 - 使用其他用户账号时，设置账号对应的密码。
`nebula.space`	无	`string`	是	导入数据对应的图空间（Space）名称。
`nebula.connection.timeout`	3000	`int`	否	Thrift 连接的超时时间，单位为 ms。
`nebula.connection.retry`	3	`int`	否	Thrift 连接重试次数。
`nebula.execution.retry`	3	`int`	否	nGQL 语句执行重试次数。
`nebula.error.max`	32	`int`	否	指定导入过程中的最大失败次数。当失败次数达到最大值时，提交的 Spark 作业将自动停止。
`nebula.error.output`	无	`string`	是	在 Nebula Graph 服务器上指定输出错误信息的日志路径。您可以在这个文件里查看发生的所有错误信息。

最后更新: 2021年4月7日

2021 Vesoft Inc.

导入命令参数

完成配置文件修改后，运行以下命令将指定来源的数据导入 Nebula Graph 数据库。

```
$SPARK_HOME/bin/spark-submit  --class com.vesoft.nebula.tools.importer.Exchange --master "local[10]" target/exchange-1.x.y.jar -c /path/to/conf/application.conf
```

说明：JAR 文件版本号以您实际编译得到的 JAR 文件名称为准。

下表列出了命令的相关参数。

参数	是否必需	默认值	说明
`--class`	是	无	指定 Driver 主类。
`--master`	是	无	指定 Spark 集群中 Master 进程的 URL。详细信息参考 master-urls。
`-c` / `--config`	是	无	指定配置文件的路径。
`-h` / `--hive`	否	`false`	添加这个参数表示支持从 HIVE 中导入数据。
`-D` / `--dry`	否	`false`	添加这个参数表示检查配置文件的格式是否符合要求，但不会校验 `tags` 和 `edges` 的配置项是否正确。正式导入数据时不能添加这个参数。

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.2 Nebula Importer

5.2.1 Nebula Importer导入CSV文件

请参见vesoft-inc/nebula-importer。

最后更新: 2021年4月25日

2021 Vesoft Inc.

5.3 Nebula Flink Connector

5.3.1 什么是 Nebula Flink Connector

Nebula Flink Connector 是一个自定义的 Flink 连接器，支持 Flink 从 Nebula Graph 图数据库中读取数据（source），或者将其他外部数据源读取的数据写入 Nebula Graph 图数据库（sink）。

您可以将 Nebula Flink Connector 应用于以下场景：

- 在不同的 Nebula Graph 集群之间迁移数据。
- 在同一个 Nebula Graph 集群内不同图空间之间迁移数据。
- Nebula Graph 与其他数据源之间迁移数据。

您可以参考以下文档使用 Nebula Flink Connector：

- 使用限制
- 自定义 source (NebulaSource)
- 自定义 sink (NebulaSink)
- 特殊说明

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.3.2 编译 Nebula Flink Connector

按以下步骤编译 Nebula Flink Connector v1.x：

1. 克隆 `nebula-java` 源代码。

```
git clone -b v1.0 https://github.com/vesoft-inc/nebula-java.git
```

2. 切换到 `nebula-java` 目录，并打包 Nebula Java 1.x。

```
cd nebula-java
mvn clean install -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

3. 进入 `tools/nebula-flink` 目录，并编译 Nebula Flink Connector v1.x。

```
cd nebula-java/tools/nebula-flink
mvn clean package -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

编译成功后，您可以在当前目录里看到如下目录结构。在 `target` 目录下，您可以看到 `nebula-flink-1.x.y.jar` 文件。将这个文件复制到本地 Maven 库的 `com/vesoft/nebula-flink/` 目录中。

```
├── README.md
├── pom.xml
├── src
│   ├── main
│   └── test
└── target
    ├── classes
    ├── generated-sources
    ├── generated-test-sources
    ├── maven-archiver
    ├── maven-status
    ├── nebula-flink-1.x.y-sources.jar
    ├── nebula-flink-1.x.y-test-sources.jar
    ├── nebula-flink-1.x.y-tests.jar
    ├── nebula-flink-1.x.y.jar
    ├── original-nebula-flink-1.x.y.jar
    └── test-classes
```

说明：JAR 文件版本号会因 Nebula Java Client 的发布版本而异。您可以在 nebula-java 仓库的 Releases 页面 查看最新的 v1.x 版本。

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.3.3 使用限制

本文描述 Nebula Flink Connector 的一些使用限制。

Nebula Graph 版本

Nebula Flink Connector 对于 Nebula Graph v1.x 及 Nebula Graph v2.x 为不同版本。

软件依赖

为保证 Nebula Flink Connector 正常工作，确认您的机器上已经安装以下软件：

- Apache Flink®：1.11 及以后版本
- JDK 8

最后更新: 2021年4月9日

2021 Vesoft Inc.

5.3.4 自定义 source (NebulaSource)

Nebula Flink Connector 支持以 `addSource` 或者 `createInput` 方式将 Nebula Graph 图数据库注册为 Flink 的数据源（source）。其中，通过 `addSource` 读取 source 数据得到的是 Flink 的 `DataStreamSource`，表示 DataStream 的起点，而通过 `createInput` 读取 source 数据得到的是 Flink 的 `DataSource`。`DataSource` 会作为进一步转换的数据集。`DataSource` 可以通过 `withParameters` 封装配置参数进行其他操作。

NebulaSource 的实现类图如下所示。

2021 Vesoft Inc.

addSource

`addSource` 方式通过 `NebulaSourceFunction` 类实现，该类继承自 `RichSourceFunction` 并实现了以下方法：

- `open` ：准备 Nebula Graph 的连接信息，并获取 Nebula Graph 图数据库 Meta 服务和 Storage 服务的连接。
- `close` ：在数据读取完成后释放资源，并断开与 Nebula Graph 图数据库服务的连接。
- `run` ：开始读取数据，并将数据填充到 `sourceContext` 。
- `cancel` ：取消 Flink 作业时调用这个方法以关闭资源。

createInput

`createInput` 方式通过 `NebulaInputFormat` 类实现，该类继承自 `RichInputFormat` 并实现了以下方法：

- `openInputFormat` ：准备 `inputFormat` 以获取连接。
- `closeInputFormat` ：数据读取完成后释放资源。断开与 Nebula Graph 图数据库服务的连接。
- `open` ：开始 `inputFormat` 的数据读取，将读取的数据转换为 Flink 的数据格式，构造迭代器。
- `close` ：在数据读取完成后打印读取日志。
- `reachedEnd` ：判断是否读取完成。
- `nextRecord` ：通过迭代器获取下一条数据.

应用实践

您可以按以下步骤使用 Nebula Flink Connector 读取 Nebula Graph 的图数据：

1. 构造 `NebulaSourceFunction` 和 `NebulaOutputFormat` 。
2. 通过 Flink 的 `addSource` 或者 `createInput` 方式将 Nebula Graph 注册为数据源。

在构造的 `NebulaSourceFunction` 和 `NebulaOutputFormat` 中，对客户端参数和执行参数作如下配置：

- `NebulaClientOptions` 需要配置：
  - Nebula Graph 图数据库 Meta 服务的 IP 地址及端口号。如果有多个服务，使用逗号分隔，例如 `“ip1:port1,ip2:port2"` 。
  - Nebula Graph 图数据库的账号及其密码。
- `VertexExecutionOptions` 需要配置：
  - 需要读取点数据的 Nebula Graph 图数据库中的图空间名称。
  - 需要读取的标签（点类型）名称。一次只能一个标签。
  - 要读取的标签属性。
  - 是否读取指定标签的所有属性，默认为 `false` 。如果配置为 `true` 则标签属性的配置无效。
  - 单次读取的数据量限值，默认为 2000 个点数据。
- `EdgeExecutionOptions` 需要配置：
  - 需要读取边数据的 Nebula Graph 图数据库中的图空间名称。
  - 需要读取的边类型。一次只能一个边类型。
  - 需要读取的边类型属性。
  - 是否读取指定边类型的所有属性，默认为 `false` 。如果配置为 `true` 则边类型属性的配置无效。
  - 单次读取的数据量限值，默认值为 2000 个边数据。

2021 Vesoft Inc.

假设需要读取点数据的 Nebula Graph 图数据库信息如下：

- Meta 服务为本地单副本部署，使用默认端口
- 图空间名称：`flinkSource`
- 标签：`player`
- 标签属性：`name` 和 `age`
- 单次最多读取 100 个点数据

以下为自定义 NebulaSource 的代码示例。

```
// 构造 Nebula Graph 客户端连接需要的参数
NebulaClientOptions nebulaClientOptions = new NebulaClientOptions
                .NebulaClientOptionsBuilder()
                .setAddress("127.0.0.1:45500")
                .build();
// 创建 connectionProvider
NebulaConnectionProvider metaConnectionProvider = new NebulaMetaConnectionProvider(nebulaClientOptions);

// 构造读取 Nebula Graph 数据需要的参数
List<String> cols = Arrays.asList("name", "age");
VertexExecutionOptions sourceExecutionOptions = new VertexExecutionOptions.ExecutionOptionBuilder()
                .setGraphSpace("flinkSource")
                .setTag(tag)
                .setFields(cols)
                .setLimit(100)
                .builder();

// 构造 NebulaInputFormat
NebulaInputFormat inputFormat = new NebulaInputFormat(metaConnectionProvider)
                .setExecutionOptions(sourceExecutionOptions);

// 方式 1 使用 createInput 方式将 Nebula Graph 注册为数据源
DataSource<Row> dataSource1 = ExecutionEnvironment.getExecutionEnvironment()
        .createInput(inputFormat);

// 方式 2 使用 addSource 方式将 Nebula Graph 注册为数据源
NebulaSourceFunction sourceFunction = new NebulaSourceFunction(metaConnectionProvider)
                .setExecutionOptions(sourceExecutionOptions);
 DataStreamSource<Row> dataSource2 = StreamExecutionEnvironment.getExecutionEnvironment()
        .addSource(sourceFunction);
```

示例程序

您可以参考 GitHub 上的示例程序 testNebulaSource 编写您自己的 Flink 应用程序。

以 testNebulaSource 为例：该程序以 Nebula Graph 图数据库为 source，以 Print 为 sink，从 Nebula Graph 图数据库中读取 59,671,064 条点数据后再打印。将该程序打包提交到 Flink 集群执行，结果如下图所示。

由上图可知，source 发送数据 59,671,064 条，sink 接收数据 59,671,064 条。

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.3.5 自定义 sink (NebulaSink)

Nebula Flink Connector 支持以 `DataStream.addSink` 的方式将 Flink 数据流写入 Nebula Graph 数据库。

> **说明**：Nebula Flink Connector 使用 Flink 1.11-SNAPSHOT 开发，这个版本已经不再支持使用 `writeUsingOutputFormat` 方式定义输出端的接口，源码如下。所以，在使用自定义 NebulaSink 时，请您务必使用 `DataStream.addSink` 方式。

```
/** @deprecated */
@Deprecated
@PublicEvolving
public DataStreamSink<T> writeUsingOutputFormat(OutputFormat<T> format) {
    return this.addSink(new OutputFormatSinkFunction(format));
    }
```

Nebula Flink Connector 中实现了自定义的 `NebulaSinkFunction`，开发者通过调用 `dataSource.addSink` 方法并将 `NebulaSinkFunction` 对象作为参数传入即可实现将 Flink 数据流写入 Nebula Graph 数据库中。

NebulaSink 的实现类图如下所示。

最重要的两个类是 `NebulaSinkFunction` `NebulaBatchOutputFormat`。

NebulaSinkFunction

`NebulaSinkFunction` 继承自 `AbstractRichFunction` 并实现了以下方法：

- `open`：调用 `NebulaBatchOutputFormat` 的 `open` 方法以准备资源。
- `close`：调用 `NebulaBatchOutputFormat` 的 `close` 方法以释放资源。
- `invoke`：是 NebulaSink 中的核心方法，调用 `NebulaBatchOutputFormat` 中的 `write` 方法写入数据。
- `flush`：调用 `NebulaBatchOutputFormat` 的 `flush` 方法提交数据。

2021 Vesoft Inc.

NebulaBatchOutputFormat

`NebulaBatchOutputFormat` 继承自 `AbstractNebulaOutPutFormat`，而后者继承自 `RichOutputFormat`，主要实现了以下方法：

- `open`：准备 Nebula Graph 数据库的 Graph 服务的连接，并初始化数据写入执行器 `nebulaBatchExecutor`。
- `close`：提交最后批次的数据，等待最后提交的回调结果并关闭服务连接等资源。
- `writeRecord`：核心方法，将数据写入 bufferedRow 中，并在达到配置的批量写入上限时提交写入。NebulaSink 的写入操作是异步的，所以需要执行回调来获取执行结果。
- `flush`：当 bufferedRow 存在数据时，将数据提交到 Nebula Graph 中。

在 `AbstractNebulaOutputFormat` 中调用了 `NebulaBatchExecutor`，用于数据的批量管理和批量提交，并通过定义回调函数接收批量提交的结果，代码如下：

```
/**
 * write one record to buffer
 */
@Override
public final synchronized void writeRecord(T row) throws IOException {
    nebulaBatchExecutor.addToBatch(row);

    if (numPendingRow.incrementAndGet() >= executionOptions.getBatch()) {
        commit();
    }
}

/**
 * put record into buffer
 *
 * @param record represent vertex or edge
 */
void addToBatch(T record) {
    boolean isVertex = executionOptions.getDataType().isVertex();

    NebulaOutputFormatConverter converter;
    if (isVertex) {
        converter = new NebulaRowVertexOutputFormatConverter((VertexExecutionOptions) executionOptions);
    } else {
        converter = new NebulaRowEdgeOutputFormatConverter((EdgeExecutionOptions) executionOptions);
    }
    String value = converter.createValue(record, executionOptions.getPolicy());
    if (value == null) {
        return;
    }
    nebulaBufferedRow.putRow(value);
}

/**
 * commit batch insert statements
 */
private synchronized void commit() throws IOException {
    graphClient.switchSpace(executionOptions.getGraphSpace());
    future = nebulaBatchExecutor.executeBatch(graphClient);
    // clear waiting rows
    numPendingRow.compareAndSet(executionOptions.getBatch(),0);
}

/**
 * execute the insert statement
 *
 * @param client Asynchronous graph client
 */
ListenableFuture executeBatch(AsyncGraphClientImpl client) {
    String propNames = String.join(NebulaConstant.COMMA, executionOptions.getFields());
    String values = String.join(NebulaConstant.COMMA, nebulaBufferedRow.getRows());
    // construct insert statement
    String exec = String.format(NebulaConstant.BATCH_INSERT_TEMPLATE, executionOptions.getDataType(), executionOptions.getLabel(), propNames, values);
    // execute insert statement
    ListenableFuture<Optional<Integer>> execResult = client.execute(exec);
    // define callback function
    Futures.addCallback(execResult, new FutureCallback<Optional<Integer>>() {
        @Override
        public void onSuccess(Optional<Integer> integerOptional) {
            if (integerOptional.isPresent()) {
                if (integerOptional.get() == ErrorCode.SUCCEEDED) {
                    LOG.info("batch insert Succeed");
                } else {
                    LOG.error(String.format("batch insert Error: %d",
                            integerOptional.get()));
                }
            } else {
                LOG.error("batch insert Error");
            }
        }

        @Override
        public void onFailure(Throwable throwable) {
            LOG.error("batch insert Error");
```


2021 Vesoft Inc.

```
            }
        });
        nebulaBufferedRow.clean();
        return execResult;
    }
```

由于 NebulaSink 的写入是批量、异步的，所以在最后业务结束关闭（`close`）资源之前需要将缓存中的批量数据提交且等待写入操作的完成，以防在写入提交之前提前关闭 Nebula Graph 的客户端，代码如下：

```
    /**
     * commit the batch write operator before release connection
     */
    @Override
    public  final synchronized void close() throws IOException {
        if(numPendingRow.get() > 0){
            commit();
        }
        while(!future.isDone()){
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                LOG.error("sleep interrupted, ", e);
            }
        }

        super.close();
    }
```

应用实践

Flink 将处理完成的数据 sink 到 Nebula Graph 数据库时，需要将 Flink 数据流进行 map 转换成 NebulaSink 可接收的数据格式。自定义 NebulaSink 的使用方式是通过 `addSink` 的形式，

您可以按以下步骤使用 Nebula Flink Connector 的 NebulaSink 向 Nebula Graph 写入数据：

1. 将 Flink 数据转换成 NebulaSink 可以接受的数据格式。
2. 将 `NebulaSinkFunction` 作为参数传给 `addSink` 方法来实现 Flink 数据流的写入。

在构造的 `NebulaSinkFunction` 中分别对客户端参数和执行参数作了如下配置：

- `NebulaClientOptions` 需要配置：
  - Nebula Graph 图数据库 Graph 服务的 IP 地址及端口号。如果有多个地址，使用英文逗号分隔。
  - Nebula Graph 图数据库的账号及其密码。
- `VertexExecutionOptions` 需要配置：
  - 需要写入点数据的 Nebula Graph 图数据库中的图空间名称。
  - 需要写入的标签（点类型）名称。
  - 需要写入的标签属性。
  - 需要写入的点 VID 所在 Flink 数据流 Row 中的索引。
  - 单次写入 Nebula Graph 的数据量限值，默认为 2000。
- `EdgeExecutionOptions` 需要配置：
  - 需要写入边数据的 Nebula Graph 图数据库中的图空间名称。
  - 需要写入的边类型。
  - 需要写入的边类型属性。
  - 需要写入的边起点 VID（src_Id）所在 Flink 数据流 Row 中的索引。
  - 需要写入的边终点 VID（dst_Id）所在 Flink 数据流 Row 中的索引。
  - 需要写入的边 rank 所在 Flink 数据流 Row 中的索引。如果不配置，则写入边数据时不带 rank 信息。
  - 单次写入的数据量限值，默认值为 2000。

2021 Vesoft Inc.

假设需要写入点数据的 Nebula Graph 图数据库信息如下：

- Graph 服务为本地单副本部署，使用默认端口
- 图空间名称：`flinkSink`
- 标签：`player`
- 标签属性：`name` 和 `age`

以下为自定义 NebulaSink 的代码示例。

```
// 构造 Nebula Graph 的 Graph 服务客户端连接需要的参数
NebulaClientOptions nebulaClientOptions = new NebulaClientOptions
                .NebulaClientOptionsBuilder()
                .setAddress("127.0.0.1:3699")
                .build();
NebulaConnectionProvider graphConnectionProvider = new NebulaGraphConnectionProvider(nebulaClientOptions);


// 构造 Nebula Graph 写入点数据的操作参数
List<String> cols = Arrays.asList("name", "age")
ExecutionOptions sinkExecutionOptions = new VertexExecutionOptions.ExecutionOptionBuilder()
                .setGraphSpace("flinkSink")
                .setTag(tag)
                .setFields(cols)
                .setIdIndex(0)
                .setBatch(2)
                .builder();

// 将点数据写入 Nebula Graph
dataSource.addSink(nebulaSinkFunction);
```

NEBULASINK 示例程序

您可以参考 GitHub 上的示例程序 testSourceSink 编写您自己的 Flink 应用程序。

以 testSourceSink 为例：该程序以 Nebula Graph 的图空间 `flinkSource` 作为 source，通过 Flink 读取进行 `map` 类型转换后的数据，再写入 Nebula Graph 另一个图空间 `flinkSink`，即 Nebula Graph 一个图空间 `flinkSource` 的数据流入另一个图空间 `flinkSink` 中。

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.3.6 特殊说明

Catalog

Flink 1.11.0 之前，如果依赖 Flink 的 source/sink 读写外部数据源时，用户必须手动读取对应数据系统的 Schema（模式）。例如，如果您要读写 Nebula Graph 的数据，则必须先保证明确地知晓 Nebula Graph 中的 Schema 信息。由此带来的问题是：当 Nebula Graph 中的 Schema 发生变化时，用户需要手动更新对应的 Flink 任务以保持类型匹配，否则，任何不匹配都会造成运行时报错使作业失败，整个操作冗余且繁琐，体验极差。

Flink 1.11.0 版本后，用户使用 Flink SQL 时可以自动获取表的 Schema 而不再需要输入 DDL，即 Flink 在不了解外部系统数据的 Schema 时仍能完成数据匹配。

目前 Nebula Flink Connector 已经支持数据的读写，要实现 Schema 的匹配则需要为 Flink Connector 实现 Catalog 管理。但是，为了确保 Nebula Graph 中的数据安全，Nebula Flink Connector 仅支持 Catalog 的读操作，不允许进行 Catalog 的修改和写入。

访问 Nebula Graph 指定类型的数据时，完整路径格式如下：`<graphSpace>.<VERTEX.tag>` 或者 `<graphSpace>.<EDGE.edge>`。

具体使用方式如下：

```
// 其中 address 可以配置为多个 IP 地址，格式为 "ip1:port,ip2:port"
String catalogName  = "testCatalog";
String defaultSpace = "flinkSink";
String username     = "root";
String password     = "nebula";
String address      = "127.0.0.1:45500";
String table        = "VERTEX.player"

// define Nebula catalog
Catalog catalog = NebulaCatalogUtils.createNebulaCatalog(catalogName, defaultSpace, address, username, password);
// define Flink table environment
StreamExecutionEnvironment bsEnv = StreamExecutionEnvironment.getExecutionEnvironment();
tEnv = StreamTableEnvironment.create(bsEnv);
// register customed nebula catalog
tEnv.registerCatalog(catalogName, catalog);
// use customed nebula catalog
tEnv.useCatalog(catalogName);

// show graph spaces of nebula
String[] spaces = tEnv.listDatabases();

// // show tags and edges of Nebula Graph
tEnv.useDatabase(defaultSpace);
String[] tables = tEnv.listTables();

// check tage player exist in defaultSpace
ObjectPath path = new ObjectPath(defaultSpace, table);
assert catalog.tableExists(path) == true

// get nebula tag schema
CatalogBaseTable table = catalog.getTable(new ObjectPath(defaultSpace, table));
table.getSchema();
```

关于 Catalog 接口的详细信息，参考 Flink-table 代码。

Exactly-once

Flink Connector 的 Exactly-once 是指 Flink 借助 checkpoint 机制保证每个输入事件只对最终结果影响一次，在数据处理过程中即使出现故障，也不会出现数据重复和丢失的情况。

为了提供端到端的 Exactly-once 语义，Flink 的外部数据系统也必须提供提交或回滚的方法，然后通过 Flink 的 checkpoint 机制协调。Flink 提供了实现端到端的 Exactly-once 的抽象，即实现二阶段提交的抽象类 `TwoPhaseCommitSinkFunction`。

要为数据输出端实现 Exactly-once，需要实现四个函数：

- `beginTransaction`：在事务开始前，在目标文件系统的临时目录中创建一个临时文件，随后可以在数据处理时将数据写入此文件。
- `preCommit`：预提交阶段。在这个阶段，刷新文件到存储，关闭文件不再写入。为下一个 checkpoint 的任何后续文件写入启动一个新事务。
- `commit`：提交阶段。在这个阶段，将预提交阶段的文件原子地移动到真正的目标目录。二阶段提交过程会增加输出数据可见性的延迟。
- `abort`：终止阶段。在这个阶段，删除临时文件。

2021 Vesoft Inc.

由以上函数可看出，Flink 的二阶段提交对外部数据源有要求，即 source 数据源必须具备重发功能，sink 数据池必须支持事务提交和幂等写。

Nebula Graph v1.2.1 不支持事务，但其写入操作是幂等的，即同一条数据的多次写入结果是一致的。因此可以通过 checkpoint 机制实现 Nebula Flink Connector 的 At-least-Once 机制，根据多次写入的幂等性可以间接实现 sink 的 Exactly-once。

要使用 NebulaSink 的容错性，请确保在 Flink 的执行环境中开启了 checkpoint 配置，代码如下所示。

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(10000) // checkpoint every 10000 msecs
   .getCheckpointConfig()
   .setCheckpointingMode(CheckpointingMode.AT_LEAST_ONCE);
```

最后更新: 2021年4月9日

2021 Vesoft Inc.

5.4 Nebula Spark Connector

5.4.1 什么是 Nebula Spark Connector

Nebula Spark Connector（在本手册中简称为 Spark Connector）是一个 Spark 连接器，提供了通过 Spark 标准形式读写 Nebula Graph 数据库的能力，由以下两部分组成：

- Reader：为您提供了一个 Spark SQL 接口，您可以使用 Spark SQL 接口编程读取 Nebula Graph 图数据，单次读取一个点或边类型的数据，并将读取的结果组装成 Spark 的 DataFrame。参考 Nebula Spark Connector Reader。
- Writer：为您提供了一个 Spark SQL 接口，您可以使用 Spark SQL 接口编程将 DataFrame 格式的数据逐条或批量写入 Nebula Graph。参考 Nebula Spark Connector Writer。

您可以将 Nebula Spark Connector 应用于以下场景：

- 在不同的 Nebula Graph 集群之间迁移数据。
- 在同一个 Nebula Graph 集群内不同图空间之间迁移数据。
- Nebula Graph 与其他数据源之间迁移数据。

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.4.2 编译 Nebula Spark Connector

按以下步骤编译 Nebula Spark Connector v1.x：

1. 克隆 `nebula-java` 源代码。

```
git clone -b v1.0 https://github.com/vesoft-inc/nebula-java.git
```

2. 切换到 `nebula-java` 目录，并打包 Nebula Java 1.x。

```
cd nebula-java
mvn clean install -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

3. 进入 `tools/nebula-spark` 目录，并编译 Nebula Spark Connector v1.x。

```
cd nebula-java/tools/nebula-spark
mvn clean package -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

编译成功后，您可以在当前目录里看到如下目录结构。在 `target` 目录下，您可以看到 `nebula-spark-1.x.y.jar` 文件。将这个文件复制到本地 Maven 库以下路径 `com/vesoft/nebula-spark/` 。

```
├── README.md
├── pom.xml
├── src
│   └── main
└── target
    ├── classes
    ├── classes.-1146581144.timestamp
    ├── generated-sources
    ├── maven-archiver
    ├── maven-status
    ├── nebula-spark-1.x.y-tests.jar
    ├── nebula-spark-1.x.y.jar
    └── original-nebula-spark-1.x.y.jar
```

说明：JAR 文件版本号会因 Nebula Java Client 的发布版本而异。您可以在 nebula-java 仓库的 Releases 页面 查看最新的 v1.x 版本。

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.4.3 使用限制

本文描述 Nebula Spark Connector 的使用限制。

Nebula Graph 版本

Nebula Spark Connector 对于 Nebula Graph v1.x，和 Nebula Graph v2.x 为不同版本。

软件依赖

Nebula Spark Connector 默认依赖以下软件：

- Apache Spark™ 2.3.0 及更高版本
- Scala
- Java：1.8

功能限制

目前 Nebula Spark Connector 在使用时有以下功能限制：

- Reader：目前无法用于读取 Nebula Graph 中属性为空的点和边数据。
- Writer：用于向 Nebula Graph 写入边数据时，作为起点与终点 VID 的数据必须同时为整数型或非数值型。

最后更新: 2021年4月9日

2021 Vesoft Inc.

5.4.4 Nebula Spark Connector Reader

什么是 Nebula Spark Connector Reader

Nebula Spark Connector Reader 是 Nebula Spark Connector 的组成部分，为您提供了 Spark SQL 接口，您可以使用 Spark SQL 接口编程读取 Nebula Graph 图数据，单次读取一个标签或边类型的数据，并将读取的结果组装成 Spark 的 DataFrame。读出的 DataFrame 可以通过 Nebula Spark Connector Writer 写入 Nebula Graph 数据库或实现不同图空间之间的数据迁移。

NEBULA SPARK CONNECTOR READER 实现原理

Spark SQL 是 Spark 中用于处理结构化数据的一个编程模块。它提供了一个称为 DataFrame 的编程抽象，并且可以充当分布式 SQL 查询引擎。Spark SQL 允许用户自定义数据源，支持对外部数据源进行扩展。通过 Spark SQL 读取到的数据格式是以命名列方式组织的分布式数据集 DataFrame，而且 Spark SQL 提供了众多 API 方便用户对 DataFrame 进行计算和转换，能对多种数据源使用 DataFrame 接口。

接口

Spark 使用 `org.apache.spark.sql` 调用外部数据源包。以下为 Spark SQL 提供的扩展数据源相关的接口。

- 基本接口，包括：
  - `BaseRelation`：表示具有已知 Schema 的元组的集合。所有继承了 `BaseRelation` 的子类都必须生成 `StructType` 格式的 Schema。换句话说， `BaseRelation` 定义了从数据源中读取的数据在 Spark SQL 的 DataFrame 中存储的数据格式。
  - `RelationProvider`：获取参数列表，根据给定的参数返回一个新的 `BaseRelation`。
  - `DataSourceRegister`：“注册数据源”的简称，在使用数据源时不用写数据源的全限定类名，而只需要写自定义的 `shortName` 即可。
- Providers 接口，包括：
  - `RelationProvider` ：从指定数据源中生成自定义的 `relation`。 `RelationProvider#createRelation` 会基于给定的参数生成新的 `relation`。
  - `SchemaRelationProvider` ：可以基于给定的参数和给定的 Schema 信息生成新的 `relation`。
- RDD 接口，包括：
  - `RDD[InternalRow]`：从数据源中扫描出来后，需要构造成 `RDD[Row]`。

Nebula Spark Connector Reader 根据 Nebula Graph 的数据源自定义了上述部分方法，从而实现自定义 Spark 外部数据源。

实现类图

在 Nebula Spark Connector Reader 中，作为 Spark SQL 的外部数据源，Nebula Graph 通过 `sparkSession.read` 的形式读取数据。该功能实现的类图展示如下图所示。

2021 Vesoft Inc.

处理流程如下：

1. 定义数据源 `NebulaRelationProvider` ：继承 `RelationProvider` 自定义 `relation`，继承 `DataSourceRegister` 注册外部数据源。
2. 定义 `NebulaRelation`，实现 Nebula Graph 图数据 Schema 的获取和数据转换方法。在 `NebulaRelation#getSchema` 方法中连接 Nebula Graph 的 Meta 服务获取配置的返回字段对应的 Schema 信息。
3. 定义 `NebulaRDD` 读取 Nebula Graph 图数据。其中， `NebulaRDD#compute` 方法定义了如何读取 Nebula Graph 图数据，主要涉及到扫描 Nebula Graph 图数据、将读到的 Nebula Graph 的行（Row）数据转换为 Spark 的 `InternalRow` 数据，以 `InternalRow` 组成 RDD 的一行，其中每一个 `InternalRow` 表示 Nebula Graph 中的一行数据，最终通过分区迭代的形式读取 Nebula Graph 所有数据并组装成最终的 DataFrame 结果数据。

应用示例

参考 Nebula Spark Connector Reader 应用示例。

最后更新: 2021年4月7日

2021 Vesoft Inc.

Nebula Spark Connector Reader 应用示例

本文以一个示例说明如何使用 Nebula Spark Connector Reader 读取 Nebula Graph 的点和边数据。

前提条件

使用 Nebula Spark Connector Reader 前，您需要确认以下信息：

- 您的机器上已经安装了以下软件：
  - Apache Spark™ 2.3.0 及更高版本
  - Scala
  - Java：1.8
- 已经成功编译 Nebula Spark Connector Reader，并已经将 `nebula-spark-1.x.y.jar` 复制到本地 Maven 库。详细信息参考 编译 Nebula Spark Connector
- 已经获取 Nebula Graph 数据库的以下信息：
  - 图空间名称和分区数量（如果创建图空间时未设置分区数量，则默认使用 100）
  - 标签和边类型的名称以及属性
  - Meta 服务所在机器的 IP 地址及端口号

2021 Vesoft Inc.

操作步骤

参考以下步骤使用 Nebula Spark Connector Reader：

1. 在 Maven 项目的 `pom.xml` 文件中加入 `nebula-spark` 依赖。

```
<dependency>
  <groupId>com.vesoft</groupId>
  <artifactId>nebula-spark</artifactId>
  <version>1.x.y</version>
</dependency>
```

说明：`<version>` 建议配置为最新发布的 Nebula Java Client 版本号。您可以在 nebula-java 仓库的 Releases 页面 查看最新的 v1.x 版本。

1. 构建 `SparkSession` 类。这是 Spark SQL 的编码入口。

```
val sparkConf = new SparkConf
sparkConf
  .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
  .registerKryoClasses(Array[Class[_]](classOf[TCompactProtocol]))
val sparkSession = SparkSession
  .builder()
  .config(sparkConf)
  .master("local")
  .getOrCreate()
```

其中，关于 `.master()` 的设置，参考 Spark 配置的 Master URLs。

2. 按以下说明修改配置，利用 Spark 读取 Nebula Graph 的点或者边数据，得到 DataFrame。

```
// 读取 Nebula Graph 的点数据
val vertexDataset: Dataset[Row] =
      sparkSession.read
        .nebula("127.0.0.1:45500", "spaceName", "100")
        .loadVerticesToDF("tag", "*")
vertexDataset.show()

// 读取 Nebula Graph 的边数据
val edgeDataset: Dataset[Row] =
      sparkSession.read
        .nebula("127.0.0.1:45500", "spaceName", "100")
        .loadEdgesToDF("edge", "field1,field2")
edgeDataset.show()
```

其中配置说明如下：

- `nebula(<address: String>, <space: String>, <partitionNum: String>)`，所有参数均为必需参数。

```
- `<address: String>`：配置为 Nebula Graph 数据库 metad 服务所在的服务器地址及端口，如果有多个 metad 服务复本，则配置为多个地址，以英文逗号分隔，例如 `"ip1:45500,ip2:45500"`。默认端口号为 45500。
- `<space: String>`: 配置为 Nebula Graph 的图空间名称。
- `<partitionNum: String>`：设置 Spark 的分区数量。建议设置为 Nebula Graph 中创建图空间时指定的 `partitionNum`，以确保一个 Spark 分区读取 Nebula Graph 图空间中一个分区的数据。如果您在创建 Nebula Graph 图空间时未指定分区数量，则使用默认值 100。
```

- `loadVerticesToDF(<tag: String>, <fields: String>)`，所有参数均为必需参数。

```
- `<tag: String>`：配置为指定 Nebula Graph 图空间中某个标签的名称。
- `<fields: String>`：配置为指定标签的属性名称，不允许为空。如果一个标签有多个属性，则以英文逗号分隔。如果指定了属性名称，表示只读取指定的属性。如果配置为 `*`，表示读取指定标签的所有属性。
```

- `loadEdgesToDF(<edge: String>, <fields: String>)`，所有参数均为必需参数。

```
- `<edge: String>`：配置为指定 Nebula Graph 图空间中某个边类型的名称。
- `<fields: String>`：配置为指定边类型的属性名称，不允许为空。如果一个边类型有多个属性，则以英文逗号分隔。如果指定了属性名称，表示只读取指定的属性，如果配置为 `*` 表示读取指定边类型的所有属性。
```


2021 Vesoft Inc.

以下为读取结果示例。

- 读取点数据

```
20/10/27 08:51:04 INFO DAGScheduler: Job 0 finished: show at Main.scala:61, took 1.873141 s
+---------+----------+---+
|_vertexId|      name|age|
+---------+----------+---+
|        0|  Tom55322| 19|
| 84541440|Tom4152378| 27|
| 67829760|  Tom24006| 10|
| 51118080|  Tom84165| 62|
| 34406400|  Tom17308|  1|
| 17694720|  Tom73089| 56|
|   983040|  Tom82311| 95|
| 68812800|  Tom61046| 93|
| 52101120|  Tom52116| 45|
| 18677760|   Tom4773| 18|
|  1966080|  Tom25979| 20|
| 69795840|  Tom92575|  9|
| 53084160|  Tom48645| 29|
| 36372480|  Tom20594| 86|
| 19660800|  Tom27071| 32|
|  2949120|    Tom630| 61|
| 70778880|  Tom82319| 78|
| 37355520|  Tom38207| 31|
| 20643840|  Tom56158| 73|
|  3932160|  Tom36933| 59|
+---------+----------+---+
only showing top 20 rows
```

- 读取边数据

```
20/10/27 08:56:57 INFO DAGScheduler: Job 4 finished: show at Main.scala:71, took 0.085975 s
+------+------+----------+--------+
|_srcId|_dstId|start_year|end_year|
+------+------+----------+--------+
|   101|   201|      2002|    2020|
|   102|   201|      2002|    2015|
+------+------+----------+--------+
```

最后更新: 2021年4月7日

2021 Vesoft Inc.

5.4.5 Nebula Spark Connector Writer

什么是 Nebula Spark Connector Writer

Nebula Spark Connector Writer 是 Nebula Spark Connector 的组成部分，为您提供了 Spark SQL 接口，您可以使用 Spark SQL 接口编程将 DataFrame 数据逐条或批量写入 Nebula Graph。

NEBULA SPARK CONNECTOR WRITER 实现原理

Nebula Spark Connector Writer 分别提供了两个接口，用于逐条或批量地将数据写入 Nebula Graph。

逐条写入数据

Nebula Spark Connector Writer 基于 Spark 的 `DataSourceV2` 接口实现单条数据写入，实现步骤如下：

1. 继承 `WriteSupport` 接口并重写 `createWriter` 方法，并使用这个方法创建自定义的 `DataSourceWriter` 接口。
2. 继承 `DataSourceWriter` 接口，创建 `NebulaDataSourceVertexWriter` 类和 `NebulaDataSourceEdgeWriter` 类。重写 `createWriterFactory` 方法并返回自定义的 `DataWriterFactory`。重写 `commit` 方法，用于提交整个事务。重写 `abort` 方法，用于做事务回滚。

   说明：Nebula Graph v1.2.1 不支持事务操作，所以，在这个实现中，`commit` 和 `abort` 无实质性操作。
3. 继承 `DataWriterFactory`，创建 `NebulaVertexWriterFactory` 类和 `NebulaEdgeWriterFactory` 类，重写 `createWriter` 方法返回自定义的 `DataWriter`。
4. 继承 `DataWriter`，创建 `NebulaVertexWriter` 类和 `NebulaEdgeWriter` 类。重写 `write` 方法，用于写出数据。重写 `commit` 方法，用于提交事务。重写 `abort` 方法，用于做事务回滚。

   说明：Nebula Graph v1.2.1 不支持事务操作，所以在 `DataWriter` 中，`commit` 和 `abort` 无实质性操作。

Nebula Spark Connector Writer 的实现类图如下：

2021 Vesoft Inc.

`NebulaVertexWriter` 和 `NebulaEdgeWriter的write` 方法中定义了具体写入逻辑。逐条写入数据的逻辑概括如下：

1. 创建客户端，连接 Nebula Graph 的 Graph 服务。
2. 指定即将写入数据的 Nebula Graph 图空间。
3. 构造 Nebula Graph 插入数据的 nGQL 语句。
4. 提交 nGQL 语句，执行写入操作。
5. 定义回调函数接收写入操作执行结果。

批量数据写入

Nebula Spark Connector Writer 批量写入数据的实现与 Nebula Exchange 类似，即通过对 DataFrame 进行 `map` 操作批量数据的累计提交。

应用示例

参考 Nebula Spark Connector Writer 应用示例。

最后更新: 2021年4月9日

2021 Vesoft Inc.

Nebula Spark Connector Writer 应用示例

本文以一个示例说明如何使用 Nebula Spark Connector Writer 向 Nebula Graph 写入点和边数据。

前提条件

使用 Nebula Spark Connector Writer 前，您需要确认以下信息：

- 您的机器上已经安装了以下软件：
  - Apache Spark™ 2.3.0 及更高版本
  - Scala
  - Java：1.8
- 已经成功编译 Nebula Spark Connector，并已经将 `nebula-spark-1.x.y.jar` 复制到本地 Maven 库。详细信息参考 编译 Nebula Spark Connector。
- 待写入的点和边数据源。在本示例中所用的数据源为 JSON 文件，您可以从 nebula-java 库 中下载。
- Nebula Graph 的 Graph 服务 IP 地址及端口号。在本示例中，对应的信息为 `127.0.0.1:3699`。
- 在 Nebula Graph 中创建 Schema，并获取以下信息：
  - 图空间名称和分区数量。在本示例中，对应的信息为 `nb` 和 `100`。
  - 点的信息，包括标签和 VID 映射策略（`hash`）。
  - 边的信息，包括起点和终点对应的标签，以及 VID 映射策略（`hash`）。
- （可选）如果是批量写入，需要确认单次写入的最大数据量，默认为 2000。详见本文 `batchInsert` 配置项说明。

操作步骤

参考以下步骤使用 Nebula Spark Connector Writer 向 Nebula Graph 写入数据。

第 1 步

在 Maven 项目的 POM 文件中加入 `nebula-spark` 依赖。

```
<dependency>
  <groupId>com.vesoft</groupId>
  <artifactId>nebula-spark</artifactId>
  <version>1.x.y</version>
</dependency>
```

说明：`<version>` 建议配置为最新发布的 Nebula Java Client 版本号。您可以在 nebula-java 仓库的 Releases 页面 查看最新的 v1.x 版本。

第 2 步

根据逐条或批量写入需求，参考以下示例在 Spark 应用程序中完成配置。

逐条写入数据

示例代码如下：

```
// 构造点和边数据的 DataFrame,
// 这里使用 nebula-java 库 v1.0 分支里 nebula-java/examples/src/main/resources 目录下的示例数据,
// 示例数据在本地的存储路径为 examples/src/main/resources
  val vertexDF = spark.read.json("examples/src/main/resources/vertex")
      vertexDF.show()
  val edgeDF = spark.read.json("examples/src/main/resources/edge")
      edgeDF.show()

// 写入点
  vertexDF.write
    .nebula("127.0.0.1:3699", "nb", "100")
    .writeVertices("player", "vertexId", "hash")

// 写入边
  edgeDF.write
    .nebula("127.0.0.1:3699", "nb", "100")
    .wirteEdges("follow", "src_id", "dst_id")
```


2021 Vesoft Inc.

示例代码中的配置说明如下：

- `nebula(address: String, space: String, partitionNum: String)`
  - `address` ：Nebula Graph 的 Graph 服务地址及端口，可以配置多个地址，以英文逗号分隔，如 `“ip1:port,ip2:port”`，端口默认为 `3699` 。
  - `space` ：Nebula Graph 中即将写入数据的图空间名称。
  - `partitionNum` ：在 Nebula Graph 中创建图空间时指定的 `partitionNum` 的值。如果未指定，这里填写 `100` 。
- `writeVertices(tag: String, vertexField: String, policy: String = "")`
  - `tag` ：点对应的 Nebula Graph 图空间中的标签名称。
  - `vertexField` ：DataFrame 中可作为 Nebula Graph 点 VID 的列。例如，如果 DataFrame 有三列，分别为 a、b、c，其中 a 列作为点 VID 列，则该参数设置为 `"a"` 。
  - `policy` ：如果 DataFrame 中 `vertexField` 列的数据类型非数值型，则需要配置 Nebula Graph 中 VID 的映射策略，即该参数设置为 `"hash"` 。如果 `vertexField` 列的数据类型为整数型，则不需要配置。
- `writeEdges(edge: String, srcVertexField: String, dstVertexField: String, policy: String = "")`
  - `edge` ：边对应的 Nebula Graph 图空间中的边类型名称。
  - `srcVertexField` 和 `dstVertexField` ：DataFrame 中可作为边起点和边终点的列。列值必须同为整数型或同为非数值型。
  - `policy` ：如果 DataFrame 中 `srcVertexField` 列和 `dstVertexField` 列的数据类型非数值型，则需要配置 Nebula Graph 中 VID 的映射策略，即该参数设置为 `"hash"` 。如果 `srcVertexField` 列和 `dstVertexField` 列的数据类型为整数型，则不需要配置。

批量写入数据

示例代码如下：

```
// 构造点和边数据的 DataFrame,
// 这里使用 nebula-java 库 v1.0 分支里 nebula-java/examples/src/main/resources 目录下的示例数据,
// 示例数据在本地的存储路径为 examples/src/main/resources
val vertexDF = spark.read.json("examples/src/main/resources/vertex")
  vertexDF.show()
val edgeDF = spark.read.json("examples/src/main/resources/edge")
  edgeDF.show()

// 批量写入点
new NebulaBatchWriterUtils()
      .batchInsert("127.0.0.1:3699", "nb", 2000)
      .batchToNebulaVertex(vertexDF, "player", "vertexId")

// 批量写入边
new NebulaBatchWriterUtils()
      .batchInsert("127.0.0.1:3699", "nb", 2000)
      .batchToNebulaEdge(edgeDF, "follow", "source", "target")
```


2021 Vesoft Inc.

示例代码中的配置说明如下：

- `batchInsert(address: String, space: String, batch: Int = 2000)` ：
  - `address` ：Nebula Graph 的 Graph 服务地址及端口，可以配置多个地址，以英文逗号分隔，如 `“ip1:port,ip2:port”`，端口默认为 `3699` 。
  - `space` ：Nebula Graph 中即将写入数据的图空间名称。
  - `batch` ：批量写入时一批次的最大数据量，可不配置，默认为 2000。
- `batchToNebulaVertex(data: DataFrame, tag: String, vertexField: String, policy: String = "")` ：
  - `data` ：待写入 Nebula Graph 的 DataFrame 数据。
  - `tag` ：Nebula Graph 图空间中对应的标签名称。
  - `vertexField` ：DataFrame 中可作为 Nebula Graph 点 VID 的列。例如，如果 DataFrame 有三列，分别为 a、b、c，其中a 列作为点 VID 列，则该参数设置为 `"a"` 。
  - `policy` ：如果 DataFrame 中 `vertexField` 列的数据类型非数值型，则需要配置 Nebula Graph 中 VID 的映射策略，即该参数设置为 `"hash"` 。如果 `vertexField` 列的数据类型为整数型，则不需要配置。
- `batchToNebulaEdge(data: DataFrame,  edge: String, srcVertexField: String, dstVertexField: String, rankField: String = "",  policy: String = "")` ：
  - `data` ：待写入 Nebula Graph 的 DataFrame 数据。
  - `edge` ：Nebula Graph 中对应的边类型。
  - `srcVertexField` 和 `dstVertexField` ：DataFrame 中可作为边起点和边终点的列。列值必须同为整数型或同为非数值型。
  - `rankField` ：DataFrame 中可作为边 `rank` 值的列，可选配。
  - `policy` ：可选。如果 DataFrame 中 `srcVertexField` 列和 `dstVertexField` 列的数据类型非数值型，则需要配置 Nebula Graph 中 VID 的映射策略，即该参数设置为 `"hash"` 。如果 `srcVertexField` 列和 `dstVertexField` 列的数据类型为整数型，则不需要配置。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6. Nebula Graph Studio

6.1 Studio版本更新说明

6.1.1 v1.2.7-beta（2021.02.22）

- 修复：
  - 修复数字超过精度时显示不准确的问题

6.1.2 v1.2.6-beta（2021.01.19）

- 功能增强:
  - 图探索
    - 选中节点后节点信息固定页面显示
    - 显示节点/边的配置持久化保存

6.1.3 v1.2.5-beta（2021.01.12）

- 修复：
  - 图探索
    - 修复回退步骤时节点渲染问题
    - 拓展时添加默认最大限制数量

6.1.4 v1.2.4-beta（2020.12.29）

- 功能增强:
  - 图探索
    - 支持查询结果导出图片
    - 支持双击节点快速拓展

6.1.5 v1.2.3-beta（2020.12.23）

- 修复：
  - 导入
    - 修复 csv 解析预览不正确的问题
    - 修复上传大容量文件被删减的问题

6.1.6 v1.2.2-beta（2020.12.17）

- 功能增强:
  - 图探索
    - 支持导入悬挂边
  - 修复
    - 图探索
      - 修复排序问题

2021 Vesoft Inc.

6.1.7 v1.2.1-beta（2020.11.02）

- 功能增强:
  - 控制台
    - 支持查看子图
    - 导入图探索支持 AS 子句

6.1.8 v1.2.0-beta（2020.10.19）

- 功能增强:
  - 支持Schema可视化管理
    - 支持对Space的查询、创建及删除
    - 支持对Space下点类型的查询、创建、编辑及删除
    - 支持对Space下边类型的查询、创建、编辑及删除
    - 支持对Space下索引的查询、创建及删除

6.1.9 v1.1.1-beta（2020.8.20）

- 功能增强:
  - 在导航栏中添加Nebula Graph源码库链接
- 修复
  - 图探索
    - 支持使用hash和uuid预处理string类型数据生成VID（或者支持输入string类型生成VID）
    - 修正double类型数据查询结果的精度

6.1.10 v1.1.0-beta（2020.8.18）

- 功能增强:
  - 图探索
    - 支持使用索引查询点
    - 支持先对VID进行预处理，再执行查询
  - 控制台
    - 支持将点查询结果导入图探索画板
- 修复
  - 图探索
    - 节点布尔类型属性返回值显示问题修复
- 优化
  - UI交互优化
  - 文案调整

6.1.11 v1.0.10-beta（2020.7.31）

- 功能增强:
  - 图探索 - 拓展增加limit选项
  - 控制台 - 支持CSV导出查询结果

2021 Vesoft Inc.

6.1.12 v1.0.9-beta（2020.7.9）

- 修复：
  - 图探索 - 修复探索在点击小图标后语句拼接错误，无法探索的问题
  - 控制台 - 结果表格列过多时支持横向滑动
  - 导入 - 选择文件时支持搜索，兼容timestamp数据类型的导入

6.1.13 v1.0.8-beta（2020.7.3）

- 功能增强:
  - 图探索 - 增加边属性选择显示
  - 控制台 - 增加查询时间开销显示

6.1.14 v1.0.7-beta（2020.7.1）

- 修复：
  - 控制台 最短路|全路径 查找显示问题修复

6.1.15 v1.0.6-beta（2020.6.13）

- 优化
  - 图探索显示优化

6.1.16 v1.0.5-beta（2020.5.29）

- 优化：
  - 图探索交互优化
  - 数据库长连接
  - 修改同时选中多点的热键冲突，改为 按住shift + click拖选
  - 文案及交互改善

6.1.17 v1.0.4-beta（2020.4.23）

- 优化：
  - 支持移除已探索出的点
  - 支持同时选择多条边进行探索
  - 支持同时点击选中多个点的操作：click + 按住cmd/win
- 其他：
  - 修复一些体验上的反馈## 2020.1.16 - v1.0.0-beta
- 功能：
  - 控制台
  - 导数据
  - 可视化

2021 Vesoft Inc.

6.1.18 v1.0.3-beta（2020.4.15）

- 功能
  - 导入功能优化, 支持将Studio部署在远端，用户可以从本地访问并导入数据，不再限制导入数据必须与服务共同部署在本机
  - 图探索节点拓展，支持用户选择正反向查询节点，选择拓展出新节点的颜色显示逻辑(步骤or类型)
- 不兼容变更
  - 为了适应项目可以远端部署，修改了文件结构，需要拉取最新代码：`git pull origin master && docker-compose pull`

6.1.19 v1.0.2-beta（2020.3.31）

- 功能
  - 图探索增加边属性显示及点标记配置

6.1.20 v1.0.1-beta（2020.3.2）

- 优化：
  - 支持控制台执行是space选择和缓存
  - 控制台多行编辑器显示

6.1.21 v1.0.0-beta（2020.1.16）

- 功能：
  - 控制台
  - 导数据
  - 可视化

最后更新: 2021年5月27日

2021 Vesoft Inc.

6.2 认识 Nebula Graph Studio

6.2.1 什么是 Nebula Graph Studio

Nebula Graph Studio（简称 Studio）是一款可以通过 Web 访问的图数据库可视化工具，搭配 Nebula Graph DBMS 使用，为您提供构图、数据导入、编写 nGQL 查询、图探索等一站式服务。即使没有图数据库操作经验，您也可以快速成为图专家。

发行版本

Studio 目前有两个发行版本：

- Docker 版本：您可以使用 Docker 服务部署 Studio，并连接到 Nebula Graph 数据库。详细信息参考 部署 Studio。
- 云服务版本：您可以在 Nebula Graph Cloud Service 上创建 Nebula Graph 数据库实例，并一键直连云服务版 Studio。详细信息参考 Nebula Graph Cloud Service 用户手册。

两个发行版本功能基本相同。但是，因为部署方式不同，会有不同的使用限制。详细信息，参考 使用限制。

产品功能

Studio 具备以下功能：

- GUI 设计，方便您管理 Nebula Graph 图数据：
  - 使用 **Schema** 管理功能，您可以使用图形界面完成 Schema（模式）创建，快速上手 Nebula Graph。
  - 使用 **控制台** 功能，您可以使用 nGQL 语句创建 Schema，并对数据执行增删改查操作。
  - 使用 **导入** 功能，通过简单的配置，您即能批量导入点和边数据，并能实时查看数据导入日志。
- 图探索，支持可视化展示图数据，使您更容易发现数据之间的关联性，提高数据分析和解读的效率。

适用场景

如果您有以下任一需求，都可以使用 Studio：

- 您有一份数据集，想进行可视化图探索或者数据分析。您可以使用 Docker Compose 或者 Nebula Graph Cloud Service 部署 Nebula Graph，再使用 Studio 完成可视化操作。
- 您已经安装部署了 Nebula Graph 数据库，并且已经导入数据集，想使用 GUI 工具执行 nGQL 语句查询、可视化图探索或者数据分析。
- 您刚开始学习 nGQL（Nebula Graph Query Language），但是不习惯用命令行工具，更希望使用 GUI 工具查看语句输出的结果。

身份验证

对于云服务版 Studio，只有 Nebula Graph 实例的创建者以及被授予操作权限的 Nebula Graph Cloud Service 用户可以登录 Studio。详细信息参考 Nebula Graph Cloud Service 用户手册。

对于 Docker 版 Studio，因为 Nebula Graph 默认不启用身份验证，所以，一般情况下您可以使用默认账号和密码（`user` 和 `password`）登录 Studio。当 Nebula Graph 启用了身份验证后，您只能使用指定的账号和密码登录 Studio。关于 Nebula Graph 的身份验证功能，参考 Nebula Graph 用户手册。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.2.2 名词解释

本文提供了您在使用 Studio 时可能需要知道的名词解释。

- Nebula Graph Studio：在本手册中简称为 Studio，是一款可以通过 Web 访问的图数据库可视化工具，搭配 Nebula Graph DBMS 使用，为您提供构图、数据导入、编写 nGQL 查询、图探索等一站式服务。
- Nebula Graph：一款开源图数据库管理系统（Graph Database Management System），擅长处理千亿个点和万亿条边的超大规模数据集。详细信息，参考 Nebula Graph 用户手册。
- Nebula Graph Cloud Service：Nebula Graph 图数据库的云服务平台（Database-as-a-Service，DBaaS），按需付费，支持一键部署 Nebula Graph，集成了 Nebula Graph Studio，并内置资源监控工具。详细信息，参考 Nebula Graph Cloud Service 用户手册。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.2.3 使用限制

本文描述了您在使用 Studio 时可能会受到的限制。

Nebula Graph 版本支持

目前 Studio v1.x 仅支持 Nebula Graph v1.x，不支持 Nebula Graph v2.x。

系统架构

Docker 版 Studio 目前仅支持 x86_64 架构。

数据上传

使用云服务版 Studio 上传数据有以下限制：

- 目前仅支持上传没有表头行的 CSV 文件，而且仅支持英文逗号（,）作为分隔符。
- 单个文件不得超过 100 MB。
- 单个实例上传文件总量不得超过 1 GB。
- 单个文件仅能保存 1 天。

使用 Docker 版 Studio 上传数据也仅支持上传无表头的 CSV 文件，但是，单个文件大小及保存时间不受限制，而且，数据总量以本地存储容量为准。

数据备份

目前仅支持在 **控制台** 上以 CSV 格式导出查询结果，不支持其他数据备份方式。

nGQL 支持

使用 Docker 版 Studio 时，除以下内容外，您可以在 **控制台** 上执行所有 nGQL 语句语法：

- `USE <space_name>`：您只能在 **Space** 下拉列表中选择图空间，不能运行这个语句选择图空间。
- **控制台** 上使用 nGQL 语句时，您可以直接回车换行，不能使用换行符。

使用云服务版 Studio 时，除以上限制外，您也不能在 **控制台** 上执行用户管理和角色管理相关的语句，包括：

- `CREATE USER`
- `ALTER USER`
- `CHANGE PASSWORD`
- `DROP USER`
- `GRANT ROLE`
- `REVOKE ROLE`

关于语句的详细信息，参考 Nebula Graph 用户手册。

浏览器支持

建议您使用最新版本的 Chrome 访问 Studio。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.2.4 版本更新

Studio 处于持续开发状态中。您可以通过 Studio 的版本更新记录 查看最新发布的功能。

云服务版 Studio

对于云服务版 Studio，以 Nebula Graph Cloud Service 上实际部署的版本为准，您不能自行更新 Studio 版本。当前公测环境里的 Studio 版本为 v1.1.1-beta。

Docker 版 Studio

对于 Docker 版 Studio，建议您每次都在 `nebula-web-docker` 目录下运行以下命令启动 Studio：

```
docker-compose pull && docker-compose up -d
```

成功连接 Docker 版 Studio 后，您可以在页面右上角点击版本号，再点击 **新发布**，前往查看 Studio 的版本更新记录。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.2.5 快捷键

本文列出了 Studio 支持的快捷键。

要完成的任务	操作
在 **控制台** 页面运行 nGQL 语句	按 Shift + Enter 键
在 **图探索** 页面选中多个点	按 Shift 键 + 鼠标单击
在 **图探索** 页面对某个点快速拓展	鼠标双击。从 v1.2.4-beta 版本开始支持。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.3 安装与登录

6.3.1 部署 Studio

云服务版 Studio 只能在 Nebula Graph Cloud Service 上使用。当您在 Nebula Graph Cloud Service 上创建 Nebula Graph 实例时即自动完成云服务版本 Studio 的部署，一键直连即可使用，不需要自己部署。详细信息参考《Nebula Graph Cloud Service 用户手册》。但是，您需要自己部署 Docker 版 Studio。本文描述如何部署 Docker 版 Studio。

前提条件

在部署 Docker 版 Studio 之前，您需要确认以下信息：

- Nebula Graph 服务已经部署并启动。详细信息，参考《Nebula Graph 用户手册》。

  说明：您可以使用多种方式部署并启动 Nebula Graph 服务。如果您刚开始使用 Nebula Graph，建议您使用 Docker Compose 部署 Nebula Graph。详细信息参考 使用 Docker Compose 部署 Nebula Graph。

- 在即将运行 Docker 版 Studio 的机器上安装并启动 Docker Compose。详细信息参考 Docker Compose 文档。

- （可选）在中国大陆从 Docker Hub 拉取 Docker 镜像的速度可能比较慢，您可以使用 `registry-mirrors` 参数配置加速镜像。例如，如果您要使用 Docker 中国区官方镜像、网易镜像和中国科技大学的镜像，则按以下格式配置 `registry-mirrors` 参数：

  ```
  {
  "registry-mirrors": [
    "https://registry.docker-cn.com",
    "http://hub-mirror.c.163.com",
    "https://docker.mirrors.ustc.edu.cn"
    ]
  }
  ```

  配置文件的路径和方法因您的操作系统和/或 Docker Desktop 版本而异。详细信息参考 Docker Daemon 配置文档。

2021 Vesoft Inc.

操作步骤

在命令行工具中按以下步骤依次运行命令，部署并启动 Docker 版 Studio，这里我们用 Nebula Graph 版本为 1.x 的进行演示：

1. 下载 Studio 的部署配置文件。

安装包	Nebula Graph版本
nebula-graph-studio-v1.tar.gz	1.x

2. 解压后进入 `nebula-graph-studio-v1` 目录。

```
cd nebula-graph-studio-v1
```

3. 拉取 Studio 的 Docker 镜像。

```
docker-compose pull
```

4. 构建并启动 Studio 服务。其中，`-d` 表示在后台运行服务容器。

```
docker-compose up -d
```

当屏幕返回以下信息时，表示 Docker 版 Studio 已经成功启动。

```
Creating docker_importer_1 ... done
Creating docker_client_1   ... done
Creating docker_web_1      ... done
Creating docker_nginx_1    ... done
```

5. 启动成功后，在浏览器地址栏输入 `http://ip address:7001`。

说明：在运行 Docker 版 Studio 的机器上，您可以运行 `ifconfig` 或者 `ipconfig` 获取本机 IP 地址。如果您使用这台机器访问 Studio，可以在浏览器地址栏里输入 `http://localhost:7001`。

如果您在浏览器窗口中能看到以下登录界面，表示您已经成功部署并启动 Studio。

后续操作

进入 Studio 登录界面后，您需要连接 Nebula Graph。详细信息，参考 连接数据库。

最后更新: 2021年5月28日

2021 Vesoft Inc.

6.3.2 连接数据库

在 Nebula Graph Cloud Service 上，创建 Nebula Graph 实例后，您可以一键直连云服务版 Studio。详细信息参考 Nebula Graph Cloud Service 用户手册。但是，对于 Docker 版 Studio，在成功启动 Studio 后，您需要配置连接 Nebula Graph。本文主要描述 Docker 版 Studio 如何连接 Nebula Graph 数据库。

前提条件

在连接 Nebula Graph 数据库前，您需要确认以下信息：

- Docker 版 Studio 已经启动。详细信息参考 部署 Studio。
- Nebula Graph 的 Graph 服务本机 IP 地址以及服务所用端口。默认端口为 `3699`。
- Nebula Graph 数据库登录账号信息，包括用户名和密码。

  说明：如果 Nebula Graph 已经启用了身份验证，并且已经创建了不同角色的用户，您只能使用被分配到的账号和密码登录数据库。如果未启用身份验证，您可以使用默认用户名（`user`）和默认密码（`password`）登录数据库。关于启用身份验证，参考 Nebula Graph 用户手册。

2021 Vesoft Inc.

操作步骤

按以下步骤连接 Nebula Graph 数据库：

2021 Vesoft Inc.

1. 在 Studio 的 **配置数据库** 页面上，输入以下信息：
   - **Host**：填写 Nebula Graph 的 Graph 服务本机 IP 地址及端口。格式为 `ip:port`。如果端口未修改，则使用默认端口 `3699`。

     说明：即使 Nebula Graph 数据库与 Studio 部署在同一台机器上，您也必须在 **Host** 字段填写这台机器的本机 IP 地址，而不是 `127.0.0.1` 或者 `localhost`。
   - **用户名** 和 **密码**：根据 Nebula Graph 的身份验证设置填写登录账号和密码。
     - 如果未启用身份验证，可以填写默认用户名 `user` 和默认密码 `password`。
     - 如果已启用身份验证，但是未创建账号信息，您只能以 GOD 角色登录，必须填写 `root` 及对应的密码 `nebula`。
     - 如果已启用身份验证，同时又创建了不同的用户并分配了角色，不同角色的用户使用自己的账号和密码登录。

2. 完成设置后，点击 **连接** 按钮。
   如果您能看到如下图所示的界面，表示您已经成功连接到 Nebula Graph 数据库。

2021 Vesoft Inc.

一次连接会话持续 30 分钟。如果您超过 30 分钟没有操作，会话即断开，您需要重新登录数据库。

后续操作

成功连接 Nebula Graph 数据库后，根据账号的权限，您可以选择执行以下操作：

- 如果您以拥有 GOD 或者 ADMIN 权限的账号登录，可以使用 **控制台** 或者 **Schema** 页面管理 Schema。
- 如果您以拥有 GOD、ADMIN、DBA 或者 USER 权限的账号登录，可以 批量导入数据 或者在 **控制台** 页面上运行 nGQL 语句插入数据。
- 如果您以拥有 GOD、ADMIN、DBA、USER 或者 GUEST 权限的账号登录，可以在 **控制台** 页面上运行 nGQL 语句读取数据或者在 **图探索** 页面上进行图探索或数据分析。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.3.3 清除连接

使用云服务版 Studio 时，您不能清除连接。

使用 Docker 版 Studio 时，如果您需要重新连接 Nebula Graph 数据库，可以清除当前连接后再重新配置数据库。

当 Docker 版 Studio 还连接在某个 Nebula Graph 数据库时，在工具栏中，选择 **设置 > 清除连接**。之后，如果浏览器上显示 **配置数据库** 页面，表示 Studio 已经成功断开了与 Nebula Graph 数据库的连接。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.4 快速开始

6.4.1 规划 Schema

在使用 Studio 之前，您需要先根据 Nebula Graph 数据库的要求规划您的 Schema（模式）。

Schema 至少要包含以下要素：

- 标签（Tag，即点类型），以及每种标签的属性。
- 边类型（Edge Type），以及每种边类型的属性。

本文以美国 Stanford Network Analysis Platform (SNAP) 提供的 Social Network: MOOC User Action Dataset 为基础，并在其中加入由公开网络上获取的不重复的 97 个课程名称，说明如何规划 Schema。

下表列出了 Schema 要素。

要素	名称	属性名称 (数据类型)	说明
标签	**user**	`userId` (`int`) 将使用 `userId` 生成这类点数据的 VID。	表示指定 MOOC 平台的用户。
标签	**course**	- `courseId` (`int`) - `courseName` (`string`) 本示例中将使用 `courseName` 的值通过 `Hash()` 函数生成这类点数据的 VID。因为 Nebula Graph 要求同一个图空间中所有点的 VID 必须始终唯一，而 `courseId` 与部分 **user** 类 VID 重复，所以，不能使用 `courseId` 生成 **course** 类点数据的 VID。	表示指定 MOOC 平台上的课程。
边类型	**action**	- `actionId` (`int`) - `duration` (`double`)：代表源数据中的 *timestamp* 数据，表示行为持续时间 - `label` (`bool`)：表示 **user** 完成一个行为后是否退出了课程 - `feature0` (`double`) - `feature1` (`double`) - `feature2` (`double`) - `feature3` (`double`)	表示用户参与课程的行为，分别用参与活动的持续时间、参与后用户是否退出了 MOOC 平台以及行为的四个维度（feature）来描述。其中，`label` 为 `true` 表示退出 MOOC 平台，为 `false` 表示未退出平台。

下图说明示例中 **user** 类点与 **course** 类点之间如何发生关系（**action**）。

2021 Vesoft Inc.

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.4.2 准备 CSV 文件

Studio 支持通过 CSV 文件批量导入点和边数据。目前仅支持上传 CSV 文件，而且每个 CSV 文件应分别表示点数据或边数据，同时，每个 CSV 文件中不能包含表头行。所以，您需要对源数据集作如下处理：

1. 分别生成 CSV 文件。数据之间仅能使用英文逗号（,）分隔。
   - user.csv：仅包括源数据中的 `userId` 数据。
   - course.csv：仅包括 `courseId` 和 `courseName` 数据。
   - actions.csv：包括 `action` 边类型所有属性对应的数据（`actionId`、`label`、`duration`、`feature0`、`feature1`、`feature2`、`feature3`），并加入边起始点 VID 和终点 VID 的来源（`userId` 和 `courseName`）。其中，因为 `label` 属性是布尔数值，所以，将 1 替换为 `TRUE`，将 0 替换为 `FALSE`。如下图所示。

```
actionId,userId,courseName,duration,feature0,feature1,feature2,feature3,label   ← 表头行
0,0,Environmental Disruptors of Development,0.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
1,0,History of Ecology,6.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
2,0,Women in Islam,41.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
3,0,History of Ecology,49.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
4,0,Women in Islam,51.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
5,0,Legacies of the Ancient World,55.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
6,0,ITP Core 2,59.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
7,0,The Research Paper: Octavia Butler's Kindred,62.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
8,0,Neurobiology,65.000 ,-0.319991479,-0.435701433,0.106783779,-0.06730924,FALSE
```

2. 删除所有 CSV 文件中的表头行。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.4.3 创建 Schema

在 Nebula Graph 中，您必须先有 Schema，再向其中写入点数据和边数据。本文描述如何使用 Nebula Graph 的 **控制台** 或 **Schema** 功能创建 Schema。

说明：您也可以使用 nebula-console 创建 Schema。详细信息，参考 使用 Docker Compose 部署 Nebula Graph 和 Nebula Graph 快速开始。

前提条件

在 Studio 上创建 Schema 之前，您需要确认以下信息：

- Studio 已经连接到 Nebula Graph 数据库。
- 您的账号拥有 GOD、ADMIN 或 DBA 权限。详细信息，参考 Nebula Graph 内置角色。
- 您已经规划好了 Schema 的要素。
- 已经创建了图空间。

  说明：本示例假设已经创建了图空间。如果您的账号拥有 GOD 权限，也可以在 **控制台** 或 **Schema** 上创建一个图空间。

使用 Schema 管理功能创建 Schema

按以下步骤使用 **Schema** 创建 Schema：

1. 创建标签。详细信息，参考 操作标签。
2. 创建边类型。详细信息，参考 操作边类型。

使用控制台创建 Schema

按以下步骤使用 **控制台** 创建 Schema：

1. 在工具栏里，点击 **控制台** 页签。
2. 在 **当前Space** 中选择一个图空间。在本示例中，选择 **mooc_actions**。

3. 在命令行中，依次输入以下语句，并点击 图标。

```
-- 创建标签 user，带有 1 个属性
CREATE TAG user (userId int);

-- 创建标签 course，带有两个属性
CREATE TAG course (courseId int, courseName string);

-- 创建边类型，带有 7 个属性
CREATE EDGE action (actionId int, duration double, label bool, feature0 double,
feature1 double, feature2 double, feature3 double);
```


2021 Vesoft Inc.

至此，您已经完成了 Schema 创建。您可以运行以下语句查看标签与边类型的定义是否正确、完整。

```
-- 列出当前图空间中所有标签
SHOW TAGS;

-- 列出当前图空间中所有边类型
SHOW EDGES;

-- 查看每种标签和边类型的结构是否正确
DESCRIBE TAG user;
DESCRIBE TAG course;
DESCRIBE EDGE action;
```

后续操作

创建 Schema 后，您可以开始 导入数据。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.4.4 导入数据

准备好 CSV 文件，创建了 Schema 后，您可以使用 **导入** 功能将所有点和边数据上传到 Studio，用于数据查询、图探索和数据分析。

前提条件

导入数据之前，需要确认以下信息：

- Studio 已经连接到 Nebula Graph 数据库。
- Nebula Graph 数据库里已经创建了 Schema。
- CSV 文件符合 Schema 要求。
- 您的账号拥有 GOD、ADMIN、DBA 或者 USER 的权限，能往图空间中写入数据。

2021 Vesoft Inc.

操作步骤

按以下步骤导入数据：

2021 Vesoft Inc.

1. 在工具栏里，点击 **导入** 页签。
2. 在 **选择Space** 页面，选择一个图空间，再点击 **下一步** 按钮。
3. 在 **上传文件** 页面，点击 **上传文件** 按钮，并选择需要的 CSV 文件。本示例中，选择 `user.csv`、`course.csv` 和 `actions.csv` 文件。

   说明：您可以一次选择多个 CSV 文件。

4. 在文件列表的 **操作** 列，点击 **预览** 或 **删除**，保证文件信息正确，之后，再点击 **下一步** 按钮。
5. 在 **关联点** 页面，点击 **+ 绑定数据源** 按钮，在对话框中选择点数据文件，并点击 **确认** 按钮。如本示例中的 `user.csv` 或 `course.csv` 文件。
6. 在 **数据源 X** 页签下，点击 **+ Tag** 按钮。
7. 在 **vertexId** 部分，完成以下操作：

   a. 在 **对应列标** 列，点击 **选择**。

   b. 在弹出对话框中，选择数据列。在本示例中，`user.csv` 中仅有一列数据用于生成代表用户的 VID，`course.csv` 中选择表示 `courseName` 信息的 **Column 1** 用于生成代表课程的 VID。

   说明：在同一个图空间中，VID 始终唯一，不可重复。关于 VID 的信息，参考 Nebula Graph 的点标识符和分区。

   c. 在 **ID Hash** 列，选择 VID 预处理方式：如果源数据是 `int` 类型数据，选择 **保持原值**；如果源数据是 `string`、`double` 或者 `bool` 类型数据，选择 **Hash**。

8. 在 **TAG 1** 部分，完成以下操作：

   a. 在 **TAG** 下拉列表中，选择数据源对应的标签名称。在本示例中，`user.csv` 文件对应选择 **user**；`course.csv` 文件对应选择 **course**。

   b. 在显示的属性列表中，点击 **选择**，为标签属性绑定源数据。在本示例中，`user` 标签没有属性，不需要选择数据源；`course` 标签的 `courseId` 属性对应 `course.csv` 文件中的 **Column 0** 列，**类型** 为 **int**，`courseName` 属性对应文件中的 **Column 1** 列，**类型** 为 **string**。

2021 Vesoft Inc.

9. (可选) 如果您有多个标签数据文件，重复步骤 5 到步骤 8。
10. 完成配置后，点击 **下一步**。
    界面提示 **配置验证成功**，表示标签数据源绑定成功。
11. 在 **关联边** 页面，点击 **+ 绑定数据源** 按钮，在对话框中选择边数据文件，并点击 **确认** 按钮。如本示例中的 `actions.csv` 文件。
12. 在 **Edge X** 页签的 **类型** 下拉列表中，选择边类型名称。本示例中，选择 **action**。
13. 根据边类型的属性，从 `actions.csv` 文件中选择相应的数据列。其中，**srcId** 和 **dstId** 分别表示边的起点与终点，所选择的数据及处理方式必须与相应的 VID 保持一致。本示例中，**srcId** 对应的是表示用户的 VID，**dstId** 对应的是表示课程的 VID。**rank** 为选填项，可以忽略。

14. 完成设置后，点击 **下一步** 按钮。
15. 在 **导入** 页面，点击 **导入** 按钮开始导入数据。在 **log** 页面上，您可以看到数据导入进度。导入所需时间因数据量而异。导入过程中，您可以点击 **终止导入** 停止数据导入。当 **log** 页面显示如图所示信息时，表示数据导入完成。

```
09/24 05:31:14 [INFO] statsmgr.go:61: Tick: Time(145.00s), Finished(382374), Failed(0), Latency AVG(26190us), Batches Req AVG(37630us), Rows AVG(2637.05/s)
09/24 05:31:19 [INFO] statsmgr.go:61: Tick: Time(150.00s), Finished(393694), Failed(0), Latency AVG(26323us), Batches Req AVG(37809us), Rows AVG(2624.62/s)
09/24 05:31:24 [INFO] statsmgr.go:61: Tick: Time(155.00s), Finished(406174), Failed(0), Latency AVG(26369us), Batches Req AVG(37875us), Rows AVG(2620.47/s)
09/24 05:31:26 [INFO] reader.go:180: Total lines of file(/upload-dir/actions.csv) is: 411749, error lines: 0
```


2021 Vesoft Inc.

后续操作

完成数据导入后，您可以开始 图探索。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.4.5 查询图数据

导入数据后，您可以开始使用 **控制台** 或者 **图探索** 查询图数据。

以查询代表“History of Chinese Women Through Time”课程的点的属性为例：

- 在 **控制台** 页面：运行 `FETCH PROP ON * hash("History of Chinese Women Through Time");`，数据库会返回这个点所有属性信息。返回结果后，点击 **导入图探索** 按钮，将点数据查询结果导入 **图探索** 进行可视化显示。

- 在 **图探索** 页面：点击 **开始探索** 按钮，在 **指定VID** 对话框中，输入 **"History of Chinese Women Through Time"**，在 **VID预处理** 选择 **Hash**，再点击 **添加** 按钮。**图探索** 画板里会显示这个点，将鼠标移到点上，您能看到这个点所有属性信息，如下图所示。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.5 操作指南

6.5.1 管理 Schema

操作图空间

Studio 连接到 Nebula Graph 数据库后，您可以创建或删除图空间。您可以使用 **控制台** 或者 **Schema** 操作图空间。本文仅说明如何使用 **Schema** 操作图空间。

支持版本

Studio v1.2.0-beta 及以后版本。请更新版本，详细操作参考 版本更新。

前提条件

操作图空间之前，您需要确保以下信息：

- Studio 已经连接到 Nebula Graph 数据库。
- 您当前登录的账号拥有创建或删除图空间的权限，即：
  - 如果 Nebula Graph 未开启身份验证，您以默认用户名 `user` 账号和默认密码 `password` 登录。
  - 如果 Nebula Graph 已开启身份验证，您以 `root` 账号及其密码登录。

创建图空间

按以下步骤使用 **Schema** 创建图空间：

1. 在工具栏里，点击 **Schema** 页签。
2. 在图空间列表上方，点击 **+ 创建** 按钮。
3. 在 **创建** 页面，完成以下配置：

   a. **名称**：指定图空间名称，本示例中设置为 `mooc_actions`。不可与已有的图空间名称重复。名称命名规则，参考《nGQL 用户手册》。

   b. **选填参数**：分别设置 `partition_num`、`replica_factor`、`charset` 或者 `collate` 的值。在本示例中，四个参数分别设置为 `10`、`1`、`utf8` 和 `utf8_bin`。详细信息，参考 `CREATE SPACE` 语法。

   在 **对应的nGQL语句** 面板上，您能看到上述设置对应的 nGQL 语句。如下所示：

   ```
   CREATE SPACE mooc_actions (partition_num = 10, replica_factor = 1, charset = utf8, collate = utf8_bin)
   ```

4. 配置确认无误后，点击 **创建** 按钮。 如果页面回到 **图空间列表**，而且列表中显示刚创建的图空间信息，表示图空间创建成功。

2021 Vesoft Inc.

删除图空间

按以下步骤使用 **Schema** 删除图空间：

1. 在工具栏里，点击 **Schema** 页签。
2. 在图空间列表里，找到需要删除的图空间，并在 **操作** 列中，点击 图标。

3. 在弹出对话框中，确认信息，并点击 **确认** 按钮。
   删除成功后，页面回到 **图空间列表**。

2021 Vesoft Inc.

后续操作

图空间创建成功后，您可以开始创建或修改 Schema，包括：

- 操作标签
- 操作边类型
- 操作索引

最后更新: 2021年4月7日

2021 Vesoft Inc.

操作标签（点类型）

在 Nebula Graph 数据库中创建图空间后，您需要创建标签（点类型）。您可以选择使用 **控制台** 或者 **Schema** 操作标签。本文仅说明如何使用 **Schema** 操作标签。

支持版本

Studio v1.2.0-beta 及以后版本。请更新版本，详细操作参考 版本更新。

前提条件

在 Studio 上操作标签之前，您必须确认以下信息：

- Studio 已经连接到 Nebula Graph 数据库。
- 图空间已经创建。
- 您当前登录的账号拥有 GOD、ADMIN 或者 DBA 的权限。

创建标签

按以下步骤使用 **Schema** 创建标签：

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称切换图空间。
4. 点击 **标签** 页签，并点击 **创建** 按钮。
5. 在 **创建** 页面上，完成以下设置：

   a. **名称**：按提示信息输入合规的标签名称。本示例中，输入 `course`。

   b. （可选）如果标签需要属性，在 **定义属性** 模块左上角，点击勾选框，并在展开的列表中，完成以下操作：
   - 输入属性名称、数据类型和默认值。 - 如果一个标签有多个属性，可以点击 **添加属性** 按钮，并定义属性。 - 如果要删除某个属性，在该属性所在行，点击 图标。

   c. （可选）标签未设置索引时，您可以设置 TTL：在 **设置TTL** 模块左上角，点击勾选框，并在展开的列表中设置 `TTL_COL` 和 `TTL_DURATION` 参数信息。关于这两个参数的详细信息，参考 TTL 配置。
6. 完成设置后，在 **对应的 nGQL**面板，您能看到与上述配置等价的 nGQL 语句。
7. 确认无误后，点击 **+ 创建** 按钮。 如果标签创建成功，**定义属性**面板会显示这个标签的属性列表。

2021 Vesoft Inc.

修改标签

按以下步骤使用 **Schema** 修改标签：

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称切换图空间。
4. 点击 **标签** 页签，找到需要修改的标签，并在 **操作** 列中，点击 图标。
5. 在 **编辑** 页面，您可以选择以下操作：
   - 如果要修改属性：在 **定义属性** 面板上，找到需要修改的属性，在右侧点击 **编辑**，再修改属性的数据类型和默认值。之后，点击 **确认** 或者 **取消** 完成修改。
   - 如果要删除属性：在 **定义属性** 面板上，找到需要删除的属性，在右侧点击 **删除**，经确认后，删除属性。
   - 如果要添加属性：在 **定义属性**面板上，点击 **添加属性** 按钮，添加属性信息。详细操作，参考 创建标签面板。
   - 如果配置了 TTL，要修改 TTL 信息：在 **设置TTL** 面板上，修改 `TTL_COL` 和 `TTL_DURATION` 配置。
   - 如果要删除已经配置的 TTL 信息：在 **设置TTL** 面板的左上角，点击勾选框，取消选择。
   - 如果要配置 TTL 信息：在 **使用TTL** 面板的右上角，点击勾选框，开始设置 TTL 信息。
6. 完成设置后，在 **对应的 nGQL** 面板上，您能看到修改后的 nGQL 语句。

删除标签

按以下步骤使用 **Schema** 删除标签：

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称切换图空间。
4. 点击 **标签** 页签，找到需要修改的标签，并在 **操作** 列中，点击 图标。

后续操作

标签创建成功后，您可以在 **控制台** 上逐条插入点数据，或者使用 **导入** 功能批量插入点数据。

最后更新: 2021年4月7日

2021 Vesoft Inc.

操作边类型

在 Nebula Graph 数据库中创建图空间后，您可能需要创建边类型。您可以选择使用 **控制台** 或者 **Schema** 操作边类型。本文仅说明如何使用 **Schema** 操作边类型。

支持版本

Studio v1.2.0-beta 及以后版本。请更新版本，详细操作参考 版本更新。

前提条件

在 Studio 上操作边类型之前，您必须确认以下信息：

- Studio 已经连接到 Nebula Graph 数据库。
- 图空间已经创建。
- 您当前登录的账号拥有 GOD、ADMIN 或者 DBA 的权限。

2021 Vesoft Inc.

创建边类型

按以下步骤使用 **Schema** 创建边类型：

2021 Vesoft Inc.

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称切换图空间。
4. 点击 **边类型** 页签，并点击 **创建** 按钮。
5. 在 **创建** 页面上，完成以下设置：

   a. **名称**：按提示信息输入合规的边类型名称。本示例中，输入 `action`。

   b. （可选）如果边类型需要属性，在 **定义属性**面板的左上角，点击勾选框，并在展开的列表中，完成以下操作：
   - 输入属性名称、数据类型和默认值。 - 如果一个边类型有多个属性，可以点击 **添加属性** 按钮，并定义属性。 - 如果要删除某个属性，在该属性所在行，点击 图标。

   c. （可选）边类型未设置索引时，您可以设置 TTL：在 **设置TTL**面板的左上角，点击勾选框，并在展开的列表中设置 `TTL_COL` 和 `TTL_DURATION` 参数信息。关于这两个参数的详细信息，参考 TTL 配置。

2021 Vesoft Inc.

6. 完成设置后，在 **对应的nGQL语句** 面板上，您能看到与上述配置等价的 nGQL 语句。

7. 确认无误后，点击 **+ 创建** 按钮。
   如果边类型创建成功，**定义属性**面板会显示这个边类型的属性列表。

2021 Vesoft Inc.

修改边类型

按以下步骤使用 **Schema** 修改边类型：

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称完成图空间切换。
4. 点击 **边类型** 页签，找到需要修改的边类型，并在 **操作** 列中，点击 图标。
5. 在 **编辑** 页面，您可以选择以下操作：
   - 如果要修改属性：在 **定义属性** 面板上，找到需要修改的属性，在右侧点击 **编辑**，再修改属性的数据类型或者默认值。修改完成后，点击 **确认** 或 **取消**。
   - 如果要删除属性：在 **定义属性** 面板上，找到需要删除的属性，在右侧点击 **删除**，经确认后，删除属性。
   - 如果要添加属性：在 **定义属性** 面板上，点击 **添加属性** 按钮，添加属性信息。
   - 如果要修改 TTL：在 **设置TTL** 面板上，修改或 `TTL_COL` 和 `TTL_DURATION` 设置。
   - 如果要删除所有已经配置的 TTL：在 **设置TTL** 面板的左上角，点击勾选框，取消选择。
   - 如果要设置 TTL：在 **设置TTL** 面板的左上角，点击勾选框，开始设置 TTL。
6. 完成设置后，在 **对应的nGQL语句** 面板上，您能看到修改后的 nGQL 语句。

删除边类型

按以下步骤使用 **Schema** 删除边类型：

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称切换图空间。
4. 点击 **边类型** 页签，找到需要修改的边类型，并在 **操作** 列中，点击 图标。

后续操作

边类型创建成功后，您可以在 **控制台** 上逐条插入边数据，或者使用 **导入** 功能批量插入边数据。

最后更新: 2021年4月7日

2021 Vesoft Inc.

操作索引

您可以为标签和边类型创建索引，使得图查询时可以从拥有共同属性的同一类型的点或边开始遍历，使大型图的查询更为高效。Nebula Graph 支持两种类型的索引：标签索引和边类型索引。您可以选择使用 **控制台** 或者 **Schema** 操作索引。本文仅说明如何使用 **Schema** 操作索引。

> **说明**：一般在创建了标签或者边类型之后即可创建索引，但是，索引会影响写性能，所以，建议您先导入数据，再批量重建索引。关于索引的详细信息，参考《nGQL 用户手册》。

支持版本

Studio v1.2.0-beta 及以后版本。请更新版本，详细操作参考 版本更新。

前提条件

在 Studio 上操作索引之前，您必须确认以下信息：

- Studio 已经连接到 Nebula Graph 数据库。
- 图空间、标签和边类型已经创建。
- 您当前登录的账号拥有 GOD、ADMIN 或者 DBA 的权限。

2021 Vesoft Inc.

创建索引

按以下步骤使用 **Schema** 创建索引：

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称切换图空间。
4. 点击 **索引** 页签，再点击 **创建** 按钮。
5. 在 **创建** 页面上，完成以下设置：

   a. **索引类型**：确认或修改索引类型，即 **标签** 或者 **边类型**。本示例中选择 **边类型**。

   b. **名称**：选择要创建索引的标签或边类型名称。本示例中选择 **action**。

   c. **索引名称**：按规定指定索引名称。本示例中输入 **action_index**。

   d. **索引属性**：点击 **添加**，在 **选择关联的属性** 列表里选择需要索引的属性，并点击 **确定** 按钮。如果需要关联多个属性，重复这一步操作。您可以按界面提示重排索引属性的顺序。本示例中选择 `label` 和 `actionId`。

   说明：索引属性的顺序会影响 `LOOKUP` 语句的查询结果。详细信息，参考 《nGQL 用户手册》。

6. 完成设置后，在 **对应的 nGQL**面板，您能看到与上述配置等价的 nGQL 语句。

7. 确认无误后，点击 **+ 创建** 按钮。 如果索引创建成功，**定义属性**面板会显示这个索引的属性列表。

2021 Vesoft Inc.

查看索引

按以下步骤使用 **Schema** 查看索引：

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称切换图空间。
4. 点击 **索引** 页签，在列表左上方，选择需要查看的索引类型。
5. 在列表中，找到需要查看的索引，点击索引所在行。界面上即列出索引相关的所有属性。

删除索引

按以下步骤使用 **Schema** 删除索引：

1. 在工具栏中，点击 **Schema** 页签。
2. 在 **图空间列表** 中，找到图空间，点击图空间名称或者在 **操作** 列中点击 图标。
3. 在 **当前图空间** 里确认图空间名称。您也可以通过选择图空间名称切换图空间。
4. 点击 **索引** 页签，找到需要修改的索引，并在 **操作** 列中，点击 图标。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.5.2 使用控制台

控制台界面

Studio 的控制台界面如下图所示。

下表列出了控制台界面上的各种功能。

编号	功能	说明
1	工具栏	点击 **控制台** 页签进入控制台页面。
2	选择图空间	在 **当前图空间** 列表中选择一个图空间。 **说明**：Studio 不支持直接在输入框中运行 `USE <space_name>` 语句。
3	输入框	在输入框中输入 nGQL 语句后，点击 按钮运行语句。您可以同时输入多个语句同时运行，语句之间以 `;` 分隔。
4	清空输入框	点击 按钮，清空输入框中已经输入的内容。
5	重复语句输入	点击 按钮，在语句运行记录列表里，点击其中一个语句，输入框中即自动输入该语句。列表里提供最近 15 次语句运行记录。
6	运行	在输入框中输入 nGQL 语句后，点击 按钮即开始运行语句。
7	语句运行状态	运行 nGQL 语句后，这里显示语句运行状态。如果语句运行成功，语句以绿色显示。如果语句运行失败，语句以红色显示。
8	结果窗口	显示语句运行结果。如果语句会返回结果，结果窗口会以表格形式呈现返回的结果。
9	导出CSV文件	运行 nGQL 语句返回结果后，点击 **导出CSV文件** 按钮即能将结果以 CSV 文件的形式导出。
10	图探索功能键	根据您运行的 nGQL 语句，您可以点击图探索功能键将返回的结果导入 **图探索** 进行可视化展现，例如 导入图探索 和 查看子图。

最后更新: 2021年4月7日

2021 Vesoft Inc.

导入图探索

您可以在 **控制台** 上使用 nGQL 语句查询得到点或边的信息，再借助 **导入图探索** 功能实现查询结果的可视化。

支持版本

Studio v1.2.1-beta 及以后版本。请更新版本，详细操作参考 版本更新。

前提条件

使用导入图探索前，您需要确认以下信息：

- Studio 已经连接到 Nebula Graph 数据库。详细信息参考 连接数据库。
- 已经导入数据集。详细操作参考 导入数据。

2021 Vesoft Inc.

导入边数据

按以下步骤将 **控制台** 查询得到的边数据结果导入 **图探索**：

2021 Vesoft Inc.

1. 在工具栏里，点击 **控制台** 页签。
2. 在 **当前Space** 中选择一个图空间。在本示例中，选择 **mooc_actions**。
3. 在命令行中，输入查询语句，并点击 图标。

   说明：查询结果中必须包括边起点和终点 VID 信息。

   查询语句示例如下：

```
nebula> GO FROM 56 OVER action YIELD action._src, action._dst;
```

   查询结果可以看到 `userId` 为 56 的用户参加了哪些课程。如下图所示。

4. 点击 **导入图探索** 按钮。
5. 在弹出对话框中，配置如下： a. 点击 **边类型**。

   b. 在 **Edge Type** 字段，填写边类型名称。在本示例中，填写 `action`。

   c. 在 **Src ID** 字段，选择查询结果中代表边起点 VID 的列名。在本示例中，选择 `action._src`。

   d. 在 **Dst ID** 字段，选择查询结果中代表边终点 VID 的列名。在本示例中，选择 `action._dst`。

   e. （可选）如果返回的边数据中有边权重（`rank`）信息，则在 **Rank** 字段，选择代表边权重的列名。如果 **Rank** 字段未设置，默认为 0。

   f. 完成配置后，点击 **导入** 按钮。

2021 Vesoft Inc.

6. 在 **图探索** 页面的弹出窗口中，选择数据插入方式：
   - **增量插入**：在画图板原来的数据基础上插入新的数据。
   - **清除插入**：清除画图板上原来的数据后，再插入新的数据。

数据插入成功后，您可以看到查询得到的边数据的可视化表现。

2021 Vesoft Inc.

导入点数据结果

按以下步骤将 **控制台** 查询得到的点数据结果导入 **图探索**：

1. 在工具栏里，点击 **控制台** 页签。
2. 在 **当前Space** 中选择一个图空间。在本示例中，选择 **mooc_actions**。
3. 在命令行中，输入查询语句，并点击 图标。

   说明：查询结果中必须包括点的 VID 信息。

   查询语句示例如下：

```
-- 对于本手册中所用数据集，
-- course 类点的 VID 由 courseName 经 hash() 函数处理得到
nebula> FETCH PROP ON * hash("Media History and Theory");
```

   查询得到 `courseId` 为 `8` 的课程信息。如下图所示。

4. 点击 **导入图探索** 按钮。
5. 在弹出对话框中，配置如下： a. 点击 **点**。

   b. 在 **Vertex ID** 字段，选择查询结果中代表点 VID 的列名。在本示例中，选择 `VertexID`。

   c. 完成配置后，点击 **导入** 按钮。

6. 在 **图探索** 上的弹出窗口中选择数据插入方式：
   - **增量插入**：在画图板原来的数据基础上插入新的数据。
   - **清除插入**：清除画图板上原来的数据后，再插入新的数据。

数据插入成功后，您可以看到查询得到的点数据的可视化表现。

后续操作

数据导入图探索后，您可以对数据进行拓展分析。

最后更新: 2021年4月7日

2021 Vesoft Inc.

查看子图

在 Studio 里，您可以在 **控制台** 上运行 `FIND SHORTEST | ALL PATH` 语句查询得到指定点之间的所有路径或最短路径，然后再通过 **查看子图** 功能将查询得到的路径导入 **图探索** 进行可视化展示。

关于 `FIND SHORTEST | ALL PATH` 语句的详细信息，参考 nGQL 用户手册。

支持版本

Studio v1.2.1-beta 及以后版本。请更新版本，详细操作参考 版本更新。

前提条件

在 **控制台** 上运行 `FIND PATH` 语句并查看子图之前，您需要确认以下信息：

- Studio 版本为 v1.2.1-beta 及以后版本。
- Studio 已经连接到 Nebula Graph 数据库。详细信息参考 连接数据库。
- 已经导入数据集。详细操作参考 导入数据。

操作步骤

按以下步骤在 **控制台** 运行 `FIND PATH` 语句并将结果导入 **图探索**：

1. 在工具栏里，点击 **控制台** 页签。
2. 在 **当前Space** 中选择一个图空间。在本示例中，选择 **mooc_actions**。
3. 在命令行中，输入 `FIND SHORTEST PATH` 或者 `FIND ALL PATH` 语句，并点击 图标。

   查询语句示例如下：

   ```
   -- 对于本手册中所用数据集，
   -- course 类点的 VID 由 courseName 经 Hash() 函数处理得到
   nebula> FIND ALL PATH FROM 1,2,4,6,42 to hash("History of Ecology"),hash("Neurobiology") OVER action;
   ```

   查询得到如下图所示路径信息。

4. 点击 **查看子图** 按钮。
5. （可选）如果 **图探索** 上画板上已有数据，则选择一种数据插入方式：
   - **增量插入**：在画图板原来的数据基础上插入新的数据。
   - **清除插入**：清除画图板上原来的数据后，再插入新的数据。

数据插入成功后，您可以看到查询结果的可视化表现。

2021 Vesoft Inc.

后续操作

数据导入图探索后，您可以对数据进行拓展分析。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.6 故障排查

6.6.1 连接数据库错误

问题描述

按 连接 Studio 文档操作，提示 **配置失败**。

可能的原因及解决方法

您可以按以下步骤排查问题。

第 1 步. 确认 HOST 字段的格式是否正确

必须填写 Nebula Graph 图数据库 Graph 服务的 IP 地址（`graph_server_ip`）和端口。如果未做修改，端口默认为 `3699`。即使 Nebula Graph 与 Studio 都部署在当前机器上，您也必须使用本机 IP 地址，而不能使用 `127.0.0.1`、`localhost` 或者 `0.0.0.0`。

第 2 步. 确认 用户名 和 密码 是否正确

如果 Nebula Graph 没有开启身份认证，您可以填写任意字符串登录。

如果已经开启身份认证，您必须使用分配的账号登录。

第 3 步. 确认 NEBULA GRAPH 服务是否正常

检查 Nebula Graph 服务状态。关于查看服务的操作：

- 如果在 Linux 服务器上通过编译部署的 Nebula Graph，参考 查看 Nebula Graph 服务。
- 如果使用 Docker Compose 部署的 Nebula Graph，参考 查看 Nebula Graph 服务状态和端口。

如果 Nebula Graph 服务正常，进入第 4 步继续排查问题。否则，请重启 Nebula Graph 服务。

> **说明**：如果您之前使用 `docker-compose up -d` 启动 Nebula Graph，必须运行 `docker-compose down` 命令停止 Nebula Graph。

第 4 步. 确认 GRAPH 服务的网络连接是否正常

在 Studio 机器上运行命令（例如 `telnet <graph_server_ip> 3699`）确认 Nebula Graph 的 Graph 服务网络连接是否正常。

如果连接失败，则按以下要求检查：

- 如果 Studio 与 Nebula Graph 在同一台机器上，检查端口是否已暴露。
- 如果两者不在同一台机器上，检查 Nebula Graph 服务器的网络配置，例如，防火墙、网关以及端口。

如果按上述步骤排查后仍无法连接 Nebula Graph 服务，请前往 Nebula Graph 官方论坛 咨询。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.6.2 无法访问 Studio

问题描述

我按照文档描述启动 Studio 后访问 `127.0.0.1:7001` 或者 `0.0.0.0:7001`，但是打不开页面，为什么？

可能的原因及解决方法

您可以按以下顺序排查问题。

第 1 步. 确认系统架构

需要确认部署 Studio 服务的机器是否为 x86_64 架构。目前 Studio 仅支持 x86_64 系统架构。

第 2 步. 检查 STUDIO 服务是否正常启动

运行 `docker-compose ps` 查看服务是否已经正常启动。

如果服务正常，返回结果如下。其中，`State` 列应全部显示为 `Up`。

```
      Name                          Command               State               Ports
-----------------------------------------------------------------------------------------------------
nebula-web-docker_client_1     ./nebula-go-api                  Up      0.0.0.0:32782->8080/tcp
nebula-web-docker_importer_1   nebula-importer --port=569 ...   Up      0.0.0.0:32783->5699/tcp
nebula-web-docker_nginx_1      /docker-entrypoint.sh ngin ...   Up      0.0.0.0:7001->7001/tcp, 80/tcp
nebula-web-docker_web_1        docker-entrypoint.sh npm r ...   Up      0.0.0.0:32784->7001/tcp
```

如果没有返回以上结果，则先停止 Studio 重新启动。详细信息，参考 部署 Studio。

> **说明**：如果您之前使用 `docker-compose up -d` 启动 Studio，必须运行 `docker-compose down` 命令停止 Studio。

第 3 步. 确认访问地址

如果 Studio 与浏览器在同一台机器上，您可以在浏览器里使用 `localhost:7001`、`127.0.0.1:7001` 或者 `0.0.0.0:7001` 访问 Studio。

如果两者不在同一台机器上，必须在浏览器里输入 `<studio_server_ip>:7001`。其中，`studio_server_ip` 是指部署 Studio 服务的机器的 IP 地址。

第 4 步. 确认网络连通性

运行 `curl <studio_server_ip>:7001 -I` 确认是否正常。如果返回 `HTTP/1.1 200 OK`，表示网络连通正常。

如果连接被拒绝，则按以下要求检查：

- 如果浏览器与 Studio 在同一台机器上，检查端口是否已暴露。
- 如果两者不在同一台机器上，检查 Studio 所在机器的网络配置，例如，防火墙、网关以及端口。

如果按上述步骤排查后仍无法访问 Studio，请您前往 Nebula Graph 官方论坛 咨询。

最后更新: 2021年4月7日

2021 Vesoft Inc.

6.6.3 常见问题

为什么我无法使用某个功能？

如果发现您无法使用某个功能，建议您按以下步骤排除问题：

1. 确认 Nebula Graph 是最新版本。如果您使用 Docker Compose 部署 Nebula Graph 数据库，建议您运行 `docker-compose pull && docker-compose up -d` 拉取最新的 Docker 镜像，并启动容器。
2. 确认 Studio 是最新版本。详细信息参考 版本更新。
3. 搜索 论坛 或 GitHub 的 nebula 和 nebula-web-docker 项目，确认是否已经有类似的问题。
4. 如果上述操作均未解决您的问题，欢迎您在论坛上提交问题。

Studio 支持 Nebula Graph v2.x 吗？

Studio v1.x 仅适用于 Nebula Graph v1.x。Studio v2.x 适用于 Nebula Graph v2.x。

Studio 是否会开源？

目前还未开源。

最后更新: 2021年4月7日

2021 Vesoft Inc.

7. 社区贡献

7.1 贡献文档

Nebula Graph 文档完全开源，欢迎更多贡献者帮助改进文档。**Nebula Graph** 文档使用 Markdown 语言，并参考了 Google 开发者文档风格指南 进行编写。

7.1.1 如何贡献

贡献文档有多种方式：

- 在 GitHub 上提 issue。
- Fork 仓库，在本地分支上做更改，然后提交 PR。

最后更新: 2020年5月9日

2021 Vesoft Inc.

7.2 C++ 编码风格

请参考 Google C++ Style Guide。

最后更新: 2020年4月29日

2021 Vesoft Inc.

7.3 如何提交代码和文档

7.3.1 Step 1: 通过 GitHub Fork

1. 访问 https://github.com/vesoft-inc/nebula
2. 点击右上角 `Fork` 按钮创建远程分支

7.3.2 Step 2: 将分支克隆到本地

定义本地工作目录：

```
# 定义工作目录
working_dir=$HOME/Workspace
```

将 `user` 设置为 GitHub 账户名：

```
user={GitHub账户名}
```

clone 代码：

```
mkdir -p $working_dir
cd $working_dir
git clone git@github.com:$user/nebula.git
# 或: git clone https://github.com/$user/nebula.git

cd $working_dir/nebula
git remote add upstream git@github.com:vesoft-inc/nebula.git
# 或: git remote add upstream https://github.com/vesoft-inc/nebula.git

# 由于没有写访问权限，请勿推送至上游主分支
git remote set-url --push upstream no_push

# 确认远程分支有效：
# 正确的格式为：
# origin    git@github.com:$(user)/nebula.git (fetch)
# origin    git@github.com:$(user)/nebula.git (push)
# upstream  https://github.com/vesoft-inc/nebula (fetch)
# upstream  no_push (push)
git remote -v
```

定义预提交 hook

请将 **Nebula Graph** 预提交挂钩链接到 `.git` 目录。

此挂钩检查提交格式，构建，文档生成等。

```
cd $working_dir/nebula/.git/hooks
ln -s ../../cpplint/bin/pre-commit.sh .
```

有时，预提交挂钩不能执行，在这种情况下，需要手动执行。

```
cd $working_dir/nebula/.git/hooks
chmod +x pre-commit
```

7.3.3 Step 3: 分支

更新本地主分支：

```
cd $working_dir/nebula
git fetch upstream
git checkout master
git rebase upstream/master
```

从主分支创建并切换分支：

```
git checkout -b myfeature
```


2021 Vesoft Inc.

注意 由于一个 PR 通常包含多个 commit，在合并至 master 时容易被挤压 (squash)，因此建议创建一个独立的分支进行更改。合并后，这个分支已无用处，因此可以使用上述 rebase 命令将本地 master 与 upstream 同步。此外，如果直接将 commit 提交至 master，则需要 hard reset 主分支，例如：

```
git fetch upstream
git checkout master
git reset --hard upstream/master
git push --force origin master
```

7.3.4 Step 4: 开发

代码风格

Nebula Graph 采用 `cpplint` 来确保其代码符合 Google 的代码风格指南。此检查器将在提交代码之前实现。

添加单元测试

请为你的新功能或 bugfix 添加单元测试。在修改的模块代码目录下面有个 test 目录，可以在里面添加单元测试，然后编译运行单元测试，提交的代码必须确保所有单元测试顺利通过。

请开启单元测试进行源码编译

详情请参考源码编译。

> 请确保已设置 `-DENABLE_TESTING = ON` 启用了单元测试的构建。

运行所有单元测试

在 `nebula` 根目录运行以下命令：

```
cd nebula/build
ctest -j$(nproc)
```

验证

- 替换二进制文件

编译好的三个服务的二进制文件在 `nebula/build/src/daemon/_build/` 目录下面，编译好的 console 在 `nebula/build/src/console/_build` 目录下面。可以把二进制替换到安装目录 `bin` 下面，重启服务并做验证。

7.3.5 Step 5: 保持分支同步

```
# 当处于 myfeature 分支时：
git fetch upstream
git rebase upstream/master
```

在其他协作者将 PR 合并到基础分支之后，您需要更新 head 分支。

Step 6: Commit

提交代码更改

```
git commit
```

Step 7: Push

代码更改完成或需要备份代码时，将本地仓库创建的分支 push 到 GitHub 端的远程仓库：

```
git push origin myfeature
```


2021 Vesoft Inc.

Step 8: 创建 pull request

1. 点击此处访问 fork 仓库https://github.com/$user/nebula (替换此处的 `$user` 用户名)。
2. 点击 `myfeature` 分支旁的 `Compare & pull request` 按钮。

Step 9: 代码审查

公开的 pull request 至少需要两人审查，代码审查包括查找 bug，审查代码风格等。

最后更新: 2020年5月20日

2021 Vesoft Inc.

7.4 PR 和 Commit Message 指南

本文档介绍的 PR 和 Commit Message 指南适用于所有 **Nebula Graph** 仓库。 所有提交至 `master` 分支的 commit 均必须遵循以下准则。

7.4.1 Commit Message

```
<type>(<scope>): <subject> // scope is optional, subject is must

                <body> // optional

                <footer> // optional
```

本指南参照 AngularJS commit 规则。

- `<Type>` 描述 commit 类型。
- `<subject>` 是 commit 的简短描述。
- 如需添加详细信息，请添加空白行，然后以段落格式进行添加。

Commit 类型

Type	Description
Feature	新功能
Fix	修复 bug
Doc	文档
Style	代码格式
Refactor	代码重构
Test	增加测试
Chore	构建过程或辅助工具的变动

7.4.2 Pull Request

提交 PR 时，请在标题中包含所有更改的详细信息，并确保标题简洁。

PR 标题必需简明概括更改信息。

对于显而易见的简单更改，可不必添加描述。如果 PR 涉及复杂更改，请对更改进行概述。如果 PR 修复了相关 issue，请关联。

Pull Request 模板

```
What changes were proposed in this pull request?

Why are the changes needed?

Does this PR introduce any user-facing change?

How was this patch tested?
```

最后更新: 2020年4月29日

2021 Vesoft Inc.

8. 附录

8.1 Cypher 和 nGQL

8.1.1 基本概念对比

概念名称	Cypher	nGQL
vertex, node	node	vertex
edge, relationship	relationship	edge
vertex type	label	tag
edge type	relationship type	edge type
vertex identifier	node id generated by default	vid
edge identifier	edge id generated by default	src, dst, rank

2021 Vesoft Inc.

8.1.2 图基本操作

操作	Cypher	nGQL
列出所有 labels/tags	MATCH (n) RETURN distinct labels(n); call db.labels();	SHOW TAGS
插入指定类型的点	CREATE (:Person {age: 16})	INSERT VERTEX `<tag_name>` (prop_name_list) VALUES `<vid>` : (prop_value_list)
插入指定类型的边	CREATE (src)-[:LIKES]->(dst) SET rel.prop = V	INSERT EDGE `<edge_type>` (prop_name_list) VALUES `<src_vid>` -> `<dst_vid>` [`@<rank>`]: (prop_value_list)
删除点	MATCH (n) WHERE ID(n) = vid DETACH DELETE n	DELETE VERTEX `<vid>`
删除边	MATCH ()-[r]->() WHERE ID(r)=edgeID DELETE r	DELETE EDGE `<edge_type>` `<src_vid>` -> `<dst_vid>` [`@<rank>`]
更新点属性	SET n.name = V	UPDATE VERTEX `<vid>` SET `<update_columns>`
查询指定点的属性	MATCH (n) WHERE ID(n) = vid RETURN properties(n)	FETCH PROP ON `<tag_name>` `<vid>`
查询指定边的属性	MATCH (n)-[r]->() WHERE ID(r)=edgeID return properties(r)	FETCH PROP ON `<edge_type>` `<src_vid>` -> `<dst_vid>` [`@<rank>`]
查询指定点的某一类关系	MATCH (n)-[r:edge_type]->() WHERE ID(n) = vid	GO FROM `<vid>` OVER `<edge_type>`
指定点的某一类反向关系	MATCH (n)<-[r:edge_type]-() WHERE ID(n) = vid	GO FROM `<vid>` OVER `<edge_type>` REVERSELY
指定点某一类关系第 N-Hop 查询	MATCH (n)-[r:edge_type*N]->() WHERE ID(n) = vid return r	GO N STEPS FROM `<vid>` OVER `<edge_type>`
两点路径	MATCH p =(a)-[]->(b) WHERE ID(a) = a_vid AND ID(b) = b_vid RETURN p	FIND ALL PATH FROM `<a_vid>` TO `<b_vid>` OVER *

8.1.3 示例查询

示例使用以下数据:

2021 Vesoft Inc.

- 插入数据

```
# 插入点
nebula> INSERT VERTEX character(name, age, type) VALUES hash("saturn"):("saturn", 10000, "titan"), hash("jupiter"):("jupiter", 5000, "god");

# 插入边
nebula> INSERT EDGE father() VALUES hash("jupiter")->hash("saturn"):();

// cypher
cypher> CREATE (src:character {name:"saturn", age: 10000, type:"titan"})
      > CREATE (dst:character {name:"jupiter", age: 5000, type:"god"})
      > CREATE (src)-[rel:father]->(dst)
 ```

- 删除点

```ngql
nebula> DELETE VERTEX hash("prometheus");

cypher> MATCH (n:character {name:"prometheus"})
        > DETACH DELETE n
```

- 更新点的属性

```
nebula> UPDATE VERTEX hash("jesus") SET character.type = 'titan';
```


2021 Vesoft Inc.

```
cypher> MATCH (n:character {name:"jesus"})
      > SET n.type = 'titan'
```

- 查看点的属性

```
nebula> FETCH PROP ON character hash("saturn");
  ==================================================
  | character.name | character.age | character.type |
  ==================================================
  | saturn         | 10000         | titan          |
  --------------------------------------------------

cypher> MATCH (n:character {name:"saturn"})
      > RETURN properties(n)
   ╭──────────────────────────────────────────────────────────────────────────────╮
   │ "properties(n)"                                                              │
   ├──────────────────────────────────────────────────────────────────────────────┤
   │ {"name":"saturn","type":"titan","age":10000}                                 │
   ╰──────────────────────────────────────────────────────────────────────────────╯
```

- 查询 hercules 祖父的姓名

```
nebula> GO 2 STEPS FROM hash("hercules") OVER father YIELD  $$.character.name;
    ====================
    | $$.character.name |
    ====================
    | saturn            |
    --------------------

cypher> MATCH (src:character{name:"hercules"})-[r:father*2]->(dst:character)
      > RETURN dst.name;
       ╭──────────────────╮
       │ "dst.name"       │
       ├──────────────────┤
       │ "satun"          │
       ╰──────────────────╯
```

- 查询 hercules 父亲的姓名

```
nebula> GO FROM hash("hercules") OVER father YIELD $$.character.name;
    ====================
    | $$.character.name |
    ====================
    | jupiter           |
    --------------------

cypher> MATCH (src:character{name:"hercules"})-[r:father]->(dst:character)
      > RETURN dst.name
       ╭──────────────────╮
       │ "dst.name"       │
       ├──────────────────┤
       │ "jupiter"        │
       ╰──────────────────╯
```

- 查询百岁老人的姓名

```
nebula> # coming soon

cypher> MATCH (src:character)
      > WHERE src.age > 100
      > RETURN src.name
       ╭──────────────────╮
       │ "src.name"       │
       ├──────────────────┤
       │   "saturn"       │
       ├──────────────────┤
       │  "jupiter"       │
       ├──────────────────┤
       │  "neptune"       │
       │ ─────────────────│
       │   "pluto"        │
       ╰──────────────────╯
```

- 找出 pluto 和谁住

```
nebula> GO FROM hash("pluto") OVER lives YIELD lives._dst AS place | GO FROM $-.place OVER lives REVERSELY WHERE \
      > $$.character.name != "pluto" YIELD $$.character.name AS cohabitants;
    ==============
    | cohabitants |
    ==============
    | cerberus    |
    --------------

cypher> MATCH (src:character{name:"pluto"})-[r1:lives]->()<-[r2:lives]-(dst:character)
```


2021 Vesoft Inc.

```
      > RETURN dst.name
      ╭──────────────╮
      │ "dst.name" │
      ├──────────────┤
      │ "cerberus" │
      ╰──────────────╯
```

- 查询 Pluto 的兄弟们以及他们的居住地

```
nebula> GO FROM hash("pluto") OVER brother YIELD brother._dst AS god | \
      > GO FROM $-.god OVER lives YIELD $^.character.name AS Brother, $$.location.name AS Habitations;
    ========================
    | Brother | Habitations |
    ========================
    | jupiter | sky         |
    ------------------------
    | neptune | sea         |
    ------------------------

cypher> MATCH (src:Character{name:"pluto"})-[r1:brother]->(bro:Character)-[r2:lives]->(dst)
      > RETURN bro.name, dst.name
      ╭──────────────────────────────────────────────╮
      │ "bro.name"     │ "dst.name" │
      ├──────────────────────────────────────────────┤
      │  "jupiter"     │   "sky"    │
      ├──────────────────────────────────────────────┤
      │  "neptune"     │  "sea"     │
      ╰──────────────────────────────────────────────╯
```

最后更新: 2020年6月23日

2021 Vesoft Inc.

8.2 Gremlin 和 nGQL 对比

8.2.1 Gremlin 介绍

Gremlin 是 Apache ThinkerPop 框架下的图遍历语言。Gremlin 可以是声明性的也可以是命令性的。虽然 Gremlin 是基于 Groovy 的，但具有许多语言变体，允许开发人员以 Java、JavaScript、Python、Scala、Clojure 和 Groovy 等许多现代编程语言原生编写 Gremlin 查询。

8.2.2 nGQL 介绍

Nebula Graph 的查询语言为 nGQL， 是一种类 SQL 的声明型的文本查询语言。相比 SQL， `nGQL` 具有如下特点：

- 类 SQL，易学易用
- 可扩展
- 关键词大小写不敏感
- 支持图遍历
- 支持模式匹配
- 支持聚合运算
- 支持图计算
- 支持分布式事务（开发中）
- 无嵌入支持组合语句，易于阅读

8.2.3 基本概念对比

名称	Gremlin	nGQL
vertex, node	vertex	vertex
edge, relationship	edge	edge
vertex type	label	tag
edge type	label	edge type
vertex id	vid	vid
edge id	eid	无

Gremlin 和 nGQL 均使用唯一标识符标记顶点和边。在 **Nebula Graph** 中，用户可以使用指定标识符、哈希或 uuid 函数自动生成标识符。

2021 Vesoft Inc.

8.2.4 图基本操作

名称	Gremlin	nGQL
新建图空间	g = TinkerGraph.open().traversal()	CREATE SPACE gods
查看点类型	g.V().label()	SHOW TAGS
插入指定类型点	g.addV(String vertexLabel).property()	INSERT VERTEX `<tag_name>` (prop_name_list) VALUES `<vid>`:(prop_value_list)
插入指定类型边	g.addE(String edgeLabel).from(v1).to(v2).property()	INSERT EDGE `<edge_name>` (`<prop_name_list>`) VALUES `<src_vid>` -> `<dst_vid>`: (`<prop_value_list>`)
删除点	g.V(`<vid>`).drop()	DELETE VERTEX `<vid>`
删除边	g.E(`<vid>`).outE(`<type>`).where(otherV().is(`<vid>`))drop()	DELETE EDGE `<edge_type>` `<src_vid>` -> `<dst_vid>`
更新点属性	g.V(`<vid>`).property()	UPDATE VERTEX `<vid>` SET `<update_columns>`
查看指定点	g.V(`<vid>`)	FETCH PROP ON `<tag_name>` `<vid>`
查看指定边	g.E(`<src_vid>` >> `<dst_vid>`)	FETCH PROP ON `<edge_name>` `<src_vid>` -> `<dst_vid>`
沿指定点查询指定边	g.V(`<vid>`).outE(`<edge>`)	GO FROM `<vid>` OVER `<edge>`
沿指定点反向查询指定边	g.V(`<vid>`).in(`<edge>`)	GO FROM `<vid>` OVER `<edge>` REVERSELY
沿指定点查询指定边 N 跳	g.V(`<vid>`).repeat(out(`<edge>`)).times(N)	GO N STEPS FROM `<vid>` OVER `<edge>`
返回指定两点路径	g.V(`<vid>`).repeat(out()).until(`<vid>`).path()	FIND ALL PATH FROM `<vid>` TO `<vid>` OVER *

8.2.5 示例查询

本节中的示例使用了 Janus Graph 的示例图 *The Graphs of Gods*。该图结构如下图所示。此处使用属性图模型描述罗马万神话中诸神关系。

2021 Vesoft Inc.

- 插入数据

```
# 插入点
nebula> INSERT VERTEX character(name, age, type) VALUES hash("saturn"):("saturn", 10000, "titan"), hash("jupiter"):("jupiter", 5000, "god");

gremlin> saturn = g.addV("character").property(T.id, 1).property('name', 'saturn').property('age', 10000).property('type', 'titan').next();
==>v[1]
gremlin> jupiter = g.addV("character").property(T.id, 2).property('name', 'jupiter').property('age', 5000).property('type', 'god').next();
==>v[2]
gremlin> prometheus = g.addV("character").property(T.id, 31).property('name',  'prometheus').property('age', 1000).property('type', 'god').next();
==>v[31]
gremlin> jesus = g.addV("character").property(T.id, 32).property('name',  'jesus').property('age', 5000).property('type', 'god').next();
==>v[32]

# 插入边
nebula> INSERT EDGE father() VALUES hash("jupiter")->hash("saturn"):();
gremlin> g.addE("father").from(jupiter).to(saturn).property(T.id, 13);
==>e[13][2-father->1]
```

- 删除数据

2021 Vesoft Inc.

```
nebula> DELETE VERTEX hash("prometheus");
gremlin> g.V(prometheus).drop();
```

- 更新数据

```
nebula> UPDATE VERTEX hash("jesus") SET character.type = 'titan';
gremlin> g.V(jesus).property('age', 6000);
```

- 查看数据

```
nebula> FETCH PROP ON character hash("saturn");
==================================================
| character.name | character.age | character.type |
==================================================
| saturn         | 10000         | titan          |
--------------------------------------------------

gremlin> g.V(saturn).valueMap();
==>[name:[saturn],type:[titan],age:[10000]]
```

- 查询 hercules 的祖父

```
nebula> LOOKUP ON character WHERE character.name == 'hercules' | \
     -> GO 2 STEPS FROM $-.VertexID OVER father YIELD $$.character.name;
====================
| $$.character.name |
====================
| saturn            |
--------------------

gremlin> g.V().hasLabel('character').has('name','hercules').out('father').out('father').values('name');
==>saturn
```

- 查询 hercules 的父亲

```
nebula> LOOKUP ON character WHERE character.name == 'hercules' | \
     -> GO FROM $-.VertexID OVER father YIELD $$.character.name;
====================
| $$.character.name |
====================
| jupiter           |
--------------------

gremlin> g.V().hasLabel('character').has('name','hercules').out('father').values('name');
==>jupiter
```

- 查询年龄大于 100 的人物

```
nebula> LOOKUP ON character WHERE character.age > 100 YIELD character.name, character.age;
=======================================================
| VertexID             | character.name | character.age |
=======================================================
| 6761447489613431910  | pluto          | 4000          |
-------------------------------------------------------
| -5860788569139907963 | neptune        | 4500          |
-------------------------------------------------------
| 4863977009196259577  | jupiter        | 5000          |
-------------------------------------------------------
| -4316810810681305233 | saturn         | 10000         |
-------------------------------------------------------

gremlin> g.V().hasLabel('character').has('age',gt(100)).values('name');
==>saturn
==>jupiter
==>neptune
==>pluto
```

- 查询和 pluto 一起居住的人物

2021 Vesoft Inc.

```
nebula> GO FROM hash("pluto") OVER lives YIELD lives._dst AS place | \
GO FROM $-.place OVER lives REVERSELY YIELD $$.character.name AS cohabitants;
==============
| cohabitants |
==============
| pluto       |
--------------
| cerberus    |
--------------

gremlin> g.V(pluto).out('lives').in('lives').values('name');
==>pluto
==>cerberus
```

- 从一起居住的人物中排除 pluto 本人

```
nebula> GO FROM hash("pluto") OVER lives YIELD lives._dst AS place | GO FROM $-.place OVER lives REVERSELY WHERE \
$$.character.name != "pluto" YIELD $$.character.name AS cohabitants;
==============
| cohabitants |
==============
| cerberus    |
--------------

gremlin> g.V(pluto).out('lives').in('lives').where(is(neq(pluto))).values('name');
==>cerberus
```

- Pluto 的兄弟们

```
# where do pluto's brothers live?

nebula> GO FROM hash("pluto") OVER brother YIELD brother._dst AS brother | \
GO FROM $-.brother OVER lives YIELD $$.location.name;
====================
| $$.location.name |
====================
| sky              |
--------------------
| sea              |
--------------------

gremlin> g.V(pluto).out('brother').out('lives').values('name');
==>sky
==>sea

# which brother lives in which place?

nebula> GO FROM hash("pluto") OVER brother YIELD brother._dst AS god | \
GO FROM $-.god OVER lives YIELD $^.character.name AS Brother, $$.location.name AS Habitations;
========================
| Brother | Habitations |
========================
| jupiter | sky         |
------------------------
| neptune | sea         |
------------------------

gremlin> g.V(pluto).out('brother').as('god').out('lives').as('place').select('god','place').by('name');
==>[god:jupiter, place:sky]
==>[god:neptune, place:sea]
```

8.2.6 高级查询

图探索

```
# gremlin 版本
gremlin> Gremlin.version();
==>3.3.5

# 返回所有点
gremlin> g.V();
==>v[1]
==>v[2]
...
nebula> # Coming soon

# 统计点数
gremlin> g.V().count();
==>12
nebula> # Coming soon

# 按照点边类型统计点边个数
gremlin> g.V().groupCount().by(label);
==>[character:9,location:3]
gremlin> g.E().groupCount().by(label);
==>[mother:1,lives:5,father:2,brother:6,battled:3,pet:1]
```


2021 Vesoft Inc.

```
nebula> # Coming soon

# 返回所有边
gremlin> g.E();
==>e[13][2-father->1]
==>e[14][2-lives->3]
...
nebula> # Coming soon

# 查询所有点类型
gremlin> g.V().label().dedup();
==>character
==>location

nebula> SHOW TAGS;
==================
| ID | Name      |
==================
| 15 | character |
------------------
| 16 | location  |
------------------

# 查询所有边类型
gremlin> g.E().label().dedup();
==>father
==>lives
...
nebula> SHOW EDGES;
================
| ID | Name    |
================
| 17 | father  |
----------------
| 18 | brother |
----------------
...

# 查询所有顶点的属性
gremlin> g.V().valueMap();
==>[name:[saturn],type:[titan],age:[10000]]
==>[name:[jupiter],type:[god],age:[5000]]
...
nebula> # Coming soon

# 查询 character 顶点属性
gremlin> g.V().hasLabel('character').valueMap();
==>[name:[saturn],type:[titan],age:[10000]]
==>[name:[jupiter],type:[god],age:[5000]]
...
```

边的遍历

名称	Gremlin	nGQL
指定点沿指定边的出顶点	out(\)	GO FROM \ OVER \
指定点沿指定边的入顶点	in(\)	GO FROM \ OVER \ REVERSELY
指定点沿指定边的双向顶点	both(\)	GO FROM \ OVER \ BIDIRECT

```
# 访问某个顶点沿某条边的 OUT 方向邻接点
gremlin> g.V(jupiter).out('brother');
==>v[8]
==>v[5]
nebula> GO FROM hash("jupiter") OVER brother;
========================
| brother._dst         |
========================
| 6761447489613431910  |
------------------------
| -5860788569139907963 |
------------------------

# 访问某个顶点沿某条边的 IN 方向邻接点
gremlin> g.V(jupiter).in('brother');
==>v[5]
==>v[8]
nebula> GO FROM hash("jupiter") OVER brother REVERSELY;
=======================
| brother._dst        |
=======================
| 4863977009196259577 |
-----------------------
| 4863977009196259577 |
-----------------------
```


2021 Vesoft Inc.

```
# 访问某个顶点沿某条边的双向邻接点
gremlin> g.V(jupiter).both('brother');
==>v[8]
==>v[5]
==>v[5]
==>v[8]
nebula> GO FROM hash("jupiter") OVER brother BIDIRECT;
=======================
| brother._dst        |
=======================
| 6761447489613431910 |
-----------------------
| -5860788569139907963|
| 4863977009196259577 |
-----------------------
| 4863977009196259577 |
-----------------------

# 2度 out 查询
gremlin> g.V(hercules).out('father').out('lives');
==>v[3]
nebula> GO FROM hash("hercules") OVER father YIELD father._dst AS id | \
GO FROM $-.id OVER lives;
=======================
| lives._dst          |
=======================
| -1121386748834253737 |
-----------------------
```

has 条件过滤

名称	Gremlin	nGQL
通过 ID 来过滤顶点	hasId(\)	FETCH PROP ON \
通过 label 和属性的名字和值过滤顶点和边	has(\, \, \)	LOOKUP \ | \ WHERE \

```
# 查询 ID 为 saturn 的顶点
gremlin> g.V().hasId(saturn);
==>v[1]
nebula> FETCH PROP ON * hash("saturn");
==========================================================================
| VertexID             | character.name | character.age | character.type |
==========================================================================
| -4316810810681305233 | saturn         | 10000         | titan          |
--------------------------------------------------------------------------

# 查询 tag 为 character 且 name 属性值为 hercules 的顶点

gremlin> g.V().has('character','name','hercules').valueMap();
==>[name:[hercules],type:[demigod],age:[30]]
nebula> LOOKUP ON character WHERE character.name == 'hercules' YIELD character.name, character.age, character.type;
==========================================================================
| VertexID            | character.name | character.age | character.type |
==========================================================================
| 5976696804486077889 | hercules       | 30            | demigod        |
--------------------------------------------------------------------------
```

返回结果限制

名称	Gremlin	nGQL
指定返回结果行数	limit()	LIMIT
获取后 n 个元素	tail()	ORDER BY \ DESC LIMIT
跳过前 n 个元素	skip()	LIMIT \

```
# 查询前两个顶点
gremlin> g.V().has('character','name','hercules').out('battled').limit(2);
==>v[9]
==>v[10]
nebula> GO FROM hash('hercules') OVER battled | LIMIT 2;
=======================
| battled._dst        |
=======================
| 5301335129822214 54  |
-----------------------
| -695163537569412701 |
-----------------------
```


2021 Vesoft Inc.

```
# 查询最后一个顶点
gremlin> g.V().has('character','name','hercules').out('battled').values('name').tail(1);
==>cerberus
nebula> GO FROM hash('hercules') OVER battled YIELD $$.character.name AS name | ORDER BY name | LIMIT 1;
============
| name     |
============
| cerberus |
------------

# 跳过第 1 个元素并返回一个元素
gremlin> g.V().has('character','name','hercules').out('battled').values('name').skip(1).limit(1);
==>hydra
nebula> GO FROM hash('hercules') OVER battled YIELD $$.character.name AS name | ORDER BY name | LIMIT 1,1;
=========
| name  |
=========
| hydra |
---------
```

路径查询

名称	Gremlin	nGQL
所有路径	path()	FIND ALL PATH
不包含环路	simplePath()	\
只包含环路	cyclicPath()	\
最短路径	\	FIND SHORTEST PATH

注意： Nebula Graph 需要起始点和终点方可返回路径， Gremlin 仅需要起始点。

```
# pluto 顶点到与其有直接关联的出边顶点的路径
gremlin> g.V().hasLabel('character').has('name','pluto').out().path();
==>[v[8],v[12]]
==>[v[8],v[2]]
==>[v[8],v[5]]
==>[v[8],v[11]]

# 查询点 pluto 到点 jupiter 的最短路径
nebula> LOOKUP ON character WHERE character.name== "pluto" YIELD character.name AS name | \
    FIND SHORTEST PATH FROM $-.VertexID TO hash("jupiter") OVER *;
=========================================================
| _path_                                                |
=========================================================
| 6761447489613431910 <brother,0> 4863977009196259577
---------------------------------------------------------
```

多度查询

名称	Gremlin	nGQL
指定重复执行的语句	repeat()	N STEPS
指定重复执行的次数	times()	N STEPS
指定循环终止的条件	until()	\
指定收集数据的条件	emit()	\

```
# 查询点 pluto 出边邻点
gremlin> g.V().hasLabel('character').has('name','pluto').repeat(out()).times(1);
==>v[12]
==>v[2]
==>v[5]
==>v[11]
nebula> LOOKUP ON character WHERE character.name== "pluto" YIELD character.name AS name | \
    GO FROM $-.VertexID OVER *;
=================================================================================================
| father._dst | brother._dst         | lives._dst           | mother._dst | pet._dst            | battled._dst |
=================================================================================================
| 0           | -5860788569139907963 | 0                    | 0           | 0                   | 0            |
-------------------------------------------------------------------------------------------------
| 0           | 4863977009196259577  | 0                    | 0           | 0                   | 0            |
-------------------------------------------------------------------------------------------------
```


2021 Vesoft Inc.

```
| 0          | 0                   | -4331657707562925133 | 0          | 0                   | 0          |
-------------------------------------------------------------------------------------------------------------
| 0          | 0                   | 0                    | 0          | 4594048193862126013 | 0          |
-------------------------------------------------------------------------------------------------------------

# 查询顶点 hercules 到顶点 cerberus 之间的路径
# 循环的终止条件是遇到名称是 cerberus 的顶点
gremlin> g.V().hasLabel('character').has('name','hercules').repeat(out()).until(has('name', 'cerberus')).path();
==>[v[6],v[11]]
==>[v[6],v[2],v[8],v[11]]
==>[v[6],v[2],v[5],v[8],v[11]]
...
nebula> # Coming soon

# 查询点 hercules 的所有出边可到达点的路径
# 且终点必须是 character 类型的点
gremlin> g.V().hasLabel('character').has('name','hercules').repeat(out()).emit(hasLabel('character')).path();
==>[v[6],v[7]]
==>[v[6],v[2]]
==>[v[6],v[9]]
==>[v[6],v[10]]
...
nebula> # Coming soon

# 查询两顶点 pluto 和 saturn 之间的最短路径
# 且最大深度为 3
gremlin> g.V('pluto').repeat(out().simplePath()).until(hasId('saturn').and().loops().is(lte(3))).hasId('saturn').path();
nebula> FIND SHORTEST PATH FROM hash('pluto') TO hash('saturn') OVER * UPTO 3 STEPS;
==========================================================================================
| _path_                                                                                 |
==========================================================================================
| 6761447489613431910 <brother,0> 4863977009196259577 <father,0> -4316810810681305233
------------------------------------------------------------------------------------------
```

查询结果排序

名称	Gremlin	nGQL
升序排列	order().by()	ORDER BY
降序排列	order().by(decr)	ORDER BY DESC
随机排列	order().by(shuffle)	\

```
# 查询 pluto 的兄弟并按照年龄降序排列
gremlin> g.V(pluto).out('brother').order().by('age', decr).valueMap();
==>[name:[jupiter],type:[god],age:[5000]]
==>[name:[neptune],type:[god],age:[4500]]
nebula> GO FROM hash('pluto') OVER brother YIELD $$.character.name AS Name, $$.character.age as Age | ORDER BY Age DESC;
=================
| Name    | Age  |
=================
| jupiter | 5000 |
-----------------
| neptune | 4500 |
-----------------
```

Group By

名称	Gremlin	nGQL
对结果集进行分组	group().by()	GROUP BY
去除相同元素	dedup()	DISTINCT
对结果集进行分组并统计	groupCount()	GROUP BY COUNT

注意： GROUP BY 函数只能与 YIELD 语句一起使用。

```
# 根据顶点类别进行分组并统计各个类别的数量
gremlin> g.V().group().by(label).by(count());
==>[character:9,location:3]
nebula> # Coming soon

# 查询点 jupiter 出边邻点，使用 name 分组并统计
gremlin> g.V(jupiter).out().group().by('name').by(count());
==>[sky:1,saturn:1,neptune:1,pluto:1]
nebula> GO FROM hash('jupiter') OVER * YIELD $$.character.name AS Name, $$.character.age as Age, $$.location.name | \
GROUP BY $-.Name YIELD $-.Name, COUNT(*);
```


2021 Vesoft Inc.

```
=====================
| $-.Name | COUNT(*) |
=====================
|         | 1        |
---------------------
| pluto   | 1        |
---------------------
| saturn  | 1        |
---------------------
| neptune | 1        |
---------------------

# 查找点 jupiter 出边到达的点并去重
gremlin> g.V(jupiter).out().hasLabel('character').dedup();
==>v[1]
==>v[8]
==>v[5]
nebula> GO FROM hash('jupiter') OVER * YIELD DISTINCT $$.character.name, $$.character.age, $$.location.name;
=========================================================
| $$.character.name | $$.character.age | $$.location.name |
=========================================================
| pluto             | 4000             |                  |
---------------------------------------------------------
| neptune           | 4500             |                  |
---------------------------------------------------------
| saturn            | 10000            |                  |
---------------------------------------------------------
|                   | 0                | sky              |
---------------------------------------------------------
```

where 条件过滤

名称	Gremlin	nGQL
where 条件过滤	where()	WHERE

过滤条件对比：

名称	Gremlin	nGQL
等于	eq(object)	==
不等于	neq(object)	!=
小于	lt(number)	<
小于等于	lte(number)	<=
大于	gt(number)	>
大于等于	gte(number)	>=
判断值是否在指定的列表中	within(objects...)	udf_is_in()

```
gremlin> eq(2).test(3);
==>false
nebula> YIELD 3 == 2;
==========
| (3==2) |
==========
| false  |
----------

gremlin> within('a','b','c').test('d');
==>false
nebula> YIELD udf_is_in('d', 'a', 'b', 'c');
=====================
| udf_is_in(d,a,b,c) |
=====================
| false              |
---------------------
```

```
# 找出 pluto 和谁住并排队他本人
gremlin> g.V(pluto).out('lives').in('lives').where(is(neq(pluto))).values('name');
 ==>cerberus
nebula> GO FROM hash("pluto") OVER lives YIELD lives._dst AS place | GO FROM $-.place OVER lives REVERSELY WHERE \
$$.character.name != "pluto" YIELD $$.character.name AS cohabitants;
===============
| cohabitants |
===============
```


2021 Vesoft Inc.

```
| cerberus    |
---------------
```

逻辑运算

名称	Gremlin	nGQL
Is	is()	==
Not	not()	!=
And	and()	AND
Or	or()	OR

```
# 查询年龄大于 30 的人物
gremlin> g.V().values('age').is(gte(30));
==>10000
==>5000
==>4500
==>30
==>45
==>4000
nebula> LOOKUP ON character WHERE character.age >= 30 YIELD character.age;
=======================================
| VertexID             | character.age |
=======================================
| -4316810810681305233 | 10000         |
---------------------------------------
| 4863977009196259577  | 5000          |
---------------------------------------
| -5860788569139907963 | 4500          |
---------------------------------------
| 5976696804486077889  | 30            |
---------------------------------------
| -6780323075177699500 | 45            |
---------------------------------------
| 6761447489613431910  | 4000          |
---------------------------------------

# 查询名称为 pluto 且年龄为 4000 的人物
gremlin> g.V().has('name','pluto').and().has('age',4000);
==>v[8]
nebula> LOOKUP ON character WHERE character.name == 'pluto' AND character.age == 4000;
======================
| VertexID           |
======================
| 6761447489613431910 |
----------------------

# 逻辑非的用法
gremlin> g.V().has('name','pluto').out('brother').not(values('name').is('neptune')).values('name');
==>jupiter
nebula> LOOKUP ON character WHERE character.name == 'pluto' YIELD character.name AS name | \
GO FROM $-.VertexID OVER brother WHERE $$.character.name != 'neptune' YIELD $$.character.name;
====================
| $$.character.name |
====================
| jupiter           |
--------------------
```

统计运算

名称	Gremlin	nGQL
求和	sum()	SUM()
最大值	max()	MAX()
最小值	min()	MIN()
平均值	mean()	AVG()

Nebula Graph 统计运算必须同 `GROUP BY` 一起使用。

```
# 计算所有 character 的年龄的总和
gremlin> g.V().hasLabel('character').values('age').sum();
==>23595
```


2021 Vesoft Inc.

```
nebula> # Coming soon

# 计算所有 character 的 brother 出边数的总和
gremlin> g.V().hasLabel('character').map(outE('brother').count()).sum();
==>6
nebula> # Coming soon

# 返回所有 character 的年龄中的最大值
gremlin> g.V().hasLabel('character').values('age').max();
==>10000
nebula> # Coming soon
```

路径选取与过滤

```
# 从路径中选取第 1 步和第 3 步的结果作为最终结果
gremlin> g.V(pluto).as('a').out().as('b').out().as('c').select('a','c');
==>[a:v[8],c:v[3]]
==>[a:v[8],c:v[1]]
...
nebula> # Coming soon

# 通过 by() 指定选取的维度
gremlin> g.V(pluto).as('a').out().as('b').out().as('c').select('a','c').by('name');
==>[a:pluto,c:sky]
==>[a:pluto,c:saturn]
...
nebula> # Coming soon

# 从 map 中选择指定 key 的值
gremlin> g.V().valueMap().select('name').dedup();
==>[saturn]
==>[jupiter]
...
nebula> # Coming soon
```

分支

```
# 查找所有类型为 'character' 的顶点
# name 属性为 'jupiter' 的顶点输出其 age 属性
# 否则输出顶点的 name 属性
gremlin> g.V().hasLabel('character').choose(values('name')).option('jupiter', values('age')).option(none, values('name'));
==>saturn
==>5000
==>neptune
...

# Lambda
gremlin> g.V().branch {it.get().value('name')}.option('jupiter', values('age')).option(none, values('name'));
==>saturn
==>5000
...

# Traversal
gremlin> g.V().branch(values('name')).option('jupiter', values('age')).option(none, values('name'));
==>saturn
==>5000

# Branch
gremlin> g.V().choose(has('name','jupiter'),values('age'),values('name'));
==>saturn
==>5000

# 基于 if then 进行分组
gremlin> g.V().hasLabel("character").groupCount().by(values("age").choose(
           is(lt(40)),constant("young"),
            choose(is(lt(4500)),
                   constant("old"),
                  constant("very old"))));
==>[young:4,old:2,very old:3]
```

> **Nebula Graph** 尚无类似功能。

合并

`coalesce()` 可以接受任意数量的遍历器（traversal），按顺序执行，并返回第一个能产生输出的遍历器的结果。

`optional()` 只能接受一个遍历器（traversal），如果该遍历器能产生一个结果，则返回该结果，否则返回调用 optionalStep 的元素本身。

`union()` 可以接受任意数量的遍历器，并能够将各个遍历器的输出合并到一起。

2021 Vesoft Inc.

```
# 如果类型为 monster 则返回类型否则返回 'Not a monster'
gremlin> g.V(pluto).coalesce(has('type','monster').values('type'),constant("Not a monster"));
==>Not a monster

# 按优先级寻找到顶点 jupiter 的以下边和邻接点，找到一个就停止
# 1、brother 出边和邻接点
# 2、father 出边和邻接点
# 3、father 入边和邻接点
gremlin> g.V(jupiter).coalesce(outE('brother'), outE('father'), inE('father')).inV().path().by('name').by(label);
==>[jupiter,brother,pluto]
==>[jupiter,brother,neptune]

# 查找顶点 pluto 的 father 出顶点，如果没有就返回 pluto 自己
gremlin> g.V(pluto).optional(out('father')).valueMap();
==>[name:[pluto],type:[god],age:[4000]]

# 寻找顶点 pluto 的出 father 顶点，邻接 brother 顶点，并将结果合并，最后打印出路径
gremlin> g.V(pluto).union(out('father'),both('brother')).path();
==>[v[8],v[2]]
==>[v[8],v[5]]
```

> **Nebula Graph** 尚无类似功能。

结果聚集与展开

```
# 收集第 1 步的结果到集合 x 中
# 注意：不影响后续结果
gremlin> g.V(pluto).out().aggregate('x');
==>v[12]
==>v[2]
...

# 通过 by() 指定聚集的维度
gremlin> g.V(pluto).out().aggregate('x').by('name').cap('x');
==>[tartarus,jupiter,neptune,cerberus]

# 查询与 pluto 的两度 OUT 邻居
# 并收集这些到 x 集合里面
# 最终以 name 属性展示其邻居
gremlin> g.V(pluto).out().aggregate('x').out().aggregate('x').cap('x').unfold().values('name');
==>tartarus
==>tartarus
...
```

> **Nebula Graph** 尚无类似功能。

模式匹配

`match()` 语句为图查询提供了一种基于模式匹配的方式，以便用更具描述性的方式进行图查询。match()语句通过多个模式片段 traversal fragments 来进行模式匹配。这些 traversal fragments 中会定义一些变量，只有满足所有用变量表示的约束的对象才能够通过。

```
# 对每一个顶点，用以下模式去匹配，满足则生成一个 map<String, Object>，不满足则过滤掉
# 模式1：a 为沿 father 出边指向 jupiter 的顶点
# 模式2：b 对应当前顶点 jupiter
# 模式3：c 对应创建 jupiter 的 brother 年龄为 4000 的 顶点
gremlin> g.V().match(__.as('a').out('father').has('name', 'jupiter').as('b'), __.as('b').in('brother').has('age', 4000).as('c'));
==>[a:v[6],b:v[2],c:v[8]]

# match() 语句可以与 select() 语句配合使用，从 Map<String, Object> 中选取部分结果
gremlin> g.V().match(__.as('a').out('father').has('name', 'jupiter').as('b'), __.as('b').in('brother').has('age', 4000).as('c')).select('a', 'c').by('name');
==>[a:hercules,c:pluto]

# match() 语句可以与 where() 语句配合使用，过滤结果
gremlin> g.V().match(__.as('a').out('father').has('name', 'jupiter').as('b'), __.as('b').in('brother').has('age', 4000).as('c')).where('a', neq('c')).select('a', 'c').by('name');
==>[a:hercules,c:pluto]
```

随机过滤

`sample()` 接受一个整数值，从前一步的遍历器中采样（随机）出最多指定数目的结果。

`coin()` 字面意思是抛硬币过滤，接受一个浮点值，该浮点值表示硬币出现正面的概率。

```
# 从所有顶点的出边中随机选择 2 条
gremlin> g.V().outE().sample(2);
==>e[15][2-brother->5]
==>e[18][5-brother->2]

# 从所顶点的 name 属性中随机选取 3 个
gremlin> g.V().values('name').sample(3);
```


2021 Vesoft Inc.

```
==>hercules
==>sea
==>jupiter

# 从所有的 character 中根据 age 随机选择 3 个
gremlin> g.V().hasLabel('character').sample(3).by('age');
==>v[1]
==>v[2]
==>v[6]

# 与 local 联合使用做随机漫游
# 从顶点 pluto 出发做 3 次随机漫游
gremlin> g.V(pluto).repeat(local(bothE().sample(1).otherV())).times(3).path();
==>[v[8],e[26][8-brother->5],v[5],e[18][5-brother->2],v[2],e[13][2-father->1],v[1]]

# 每个顶点按 0.5 的概率过滤
gremlin> g.V().coin(0.5);
==>v[1]
==>v[2]
...

# 输出所有 location 类顶点的 name 属性，否则输出 not a location
gremlin> g.V().choose(hasLabel('location'), values('name'), constant('not a location'));
==>not a location
==>not a location
==>sky
...
```

结果存取口袋 Sack

包含本地数据结构的遍历器称为口袋。`sack()` 将数据放入口袋，或者从口袋取出数据。每个遍历器的每个口袋都是通过 `withSack ()` 创建的。

```
# 创建一个包含常数 1 的口袋，并且在最终取出口袋中的值
gremlin> g.withSack(1).V().sack();
==>1
==>1
...
```

遍历栅栏 barrier

`barrier()` 在某个位置插入一个栅栏，以强制该位置之前的步骤必须都执行完成才可以继续往后执行。

```
# 利用隐式 barrier 计算特征向量中心性
# 包括 groupCount、cap，按照降序排序
gremlin> g.V().repeat(both().groupCount('m')).times(5).cap('m').order(local).by(values, decr);
```

局部操作 local

通过 Gremlin 进行图遍历通常是当前 step 处理前一 step 传递过来的对象流。很多操作是针对传递过来的对象流中的全部对象进行操作，但也有很多时候需要针对对象流中的单个对象而非对象流中的全部对象进行操作。这种对单个对象的局部操作，可以使用 `local()` 语句实现。

```
# 不使用 local()
gremlin> g.V().hasLabel('character').as('character').properties('age').order().by(value,decr).limit(2).value().as('age').select('character', 'age').by('name').by();
==>[character:saturn,age:10000]
==>[character:jupiter,age:5000]

# 使用 local()
gremlin> g.V().hasLabel('character').as('character').local(properties('age').order().by(value).limit(2)).value().as('age').select('character', 'age').by('name').by()
==>[character:saturn,age:10000]
==>[character:jupiter,age:5000]
==>[character:neptune,age:4500]
==>[character:hercules,age:30]
...

# 查询 monster 的属性 map
gremlin> g.V()hasLabel('character').has('type', 'type').propertyMap();
==>[name:[vp[name->nemean]],type:[vp[type->monster]],age:[vp[age->20]]]
==>[name:[vp[name->hydra]],type:[vp[type->monster]],age:[vp[age->0]]]
==>[name:[vp[name->cerberus]],type:[vp[type->monster]],age:[vp[age->0]]]

# 查询 monster 的属性个数
gremlin> g.V()hasLabel('character').has('type', 'monster').propertyMap().count(local);
==>3
==>3
==>3

# 数目最多的顶点类型的顶点数目
gremlin> g.V().groupCount().by(label).select(values).max(local);
==>9

# 所有顶点的属性列表中的第一个属性
```


2021 Vesoft Inc.

```
gremlin> g.V().valueMap().limit(local, 1);
==>[name:[saturn]]
==>[name:[jupiter]]
==>[name:[sky]]
...

# 不加 local
gremlin> g.V().valueMap().limit(1);
==>[name:[saturn],type:[titan],age:[10000]]

# 所有顶点作为一个集合，从中采样 2 个
gremlin> g.V().fold().sample(local,2);
==>[v[8],v[1]]
```

执行统计和分析

Gremlin 提供两种语句对执行的查询语句进行统计和分析：

- `explain()`，详细描述原始的 Gremlin 语句在编译期是如何转变为最终要执行的 step 集合的
- `profile()`，统计 Gremlin 语句执行过程中的每个 step 消耗的时间和通过的对象等统计信息

```
# explain()
gremlin> g.V().hasLabel('character').explain();
==>Traversal Explanation
==================================================================================
Original Traversal                 [GraphStep(vertex,[]), HasStep([~label.eq(character)])]

ConnectiveStrategy           [D]   [GraphStep(vertex,[]), HasStep([~label.eq(character)])]
MatchPredicateStrategy       [O]   [GraphStep(vertex,[]), HasStep([~label.eq(character)])]
...
StandardVerificationStrategy [V]   [TinkerGraphStep(vertex,[~label.eq(character)])]

Final Traversal                    [TinkerGraphStep(vertex,[~label.eq(character)])]

# profile()
gremlin> g.V().out('father').profile()
==>Traversal Metrics
Step                                                               Count  Traversers       Time (ms)    % Dur
=============================================================================================================
TinkerGraphStep(vertex,[])                                            12          12           0.644    45.66
VertexStep(OUT,[father],vertex)                                        2           2           0.534    37.83
NoOpBarrierStep(2500)                                                  2           2           0.233    16.51
                                                >TOTAL                 -           -           1.411        -
```

最后更新: 2020年7月15日

2021 Vesoft Inc.

8.3 SQL 和 nGQL

8.3.1 基本概念对比

概念名称	SQL	nGQL
vertex	\	vertex
edge	\	edge
vertex type	\	tag
edge type	\	edge type
vertex identifier	\	vid
edge identifier	edge id generated by default	src, dst, rank
column	column	\
row	row	\

8.3.2 语法对比

数据定义语言（DDL）

数据定义语言（DDL）用于定义数据库 schema。 DDL 语句可以创建或修改数据库的结构。

对比项	SQL	nGQL
创建图空间（数据库）	CREATE DATABASE `<database_name>`	CREATE SPACE `<space_name>`
列出图空间（数据库）	SHOW DATABASES	SHOW SPACES
使用图空间（数据库）	USE `<database_name>`	USE `<space_name>`
删除图空间（数据库）	DROP DATABASE `<database_name>`	DROP SPACE `<space_name>`
修改图空间（数据库）	ALTER DATABASE `<database_name>` alter_option	\
创建 tags/edges	\	CREATE TAG | EDGE `<tag_name>`
创建表	CREATE TABLE `<tbl_name>` (create_definition,...)	\
列出表列名	SHOW COLUMNS FROM `<tbl_name>`	\
列出 tags/edges	\	SHOW TAGS | EDGES
Describe tags/edge	\	DESCRIBE TAG | EDGE `<tag_name | edge_name>`
修改 tags/edge	\	ALTER TAG | EDGE `<tag_name | edge_name>`
修改表	ALTER TABLE `<tbl_name>`	\

索引

对比项	SQL	nGQL
创建索引	CREATE INDEX	CREATE {TAG | EDGE} INDEX
删除索引	DROP INDEX	DROP {TAG | EDGE} INDEX
列出索引	SHOW INDEX FROM	SHOW {TAG | EDGE} INDEXES
重构索引	ANALYZE TABLE	REBUILD {TAG | EDGE} INDEX `<index_name>` [OFFLINE]

2021 Vesoft Inc.

数据操作语言（DML）

数据操作语言（DML）用于操作数据库中的数据。

对比项	SQL	nGQL
插入数据	INSERT IGNORE INTO `<tbl_name>` [(col_name [, col_name] ...)] {VALUES | VALUE} [(value_list) [, (value_list)]	INSERT VERTEX `<tag_name>` (prop_name_list[, prop_name_list]) {VALUES | VALUE} vid: (prop_value_list[, prop_value_list]) INSERT EDGE `<edge_name>` (`<prop_name_list>`) VALUES | VALUE `<src_vid>` -> `<dst_vid>` [`@<rank>`] : (`<prop_value_list>`)
查询数据	SELECT	GO, FETCH
更新数据	UPDATE `<tbl_name>` SET field1=new-value1, field2=new-value2 [WHERE Clause]	UPDATE VERTEX `<vid>` SET `<update_columns>` [WHEN `<condition>`] UPDATE EDGE `<edge>` SET `<update_columns>` [WHEN `<condition>`]
删除数据	DELETE FROM `<tbl_name>` [WHERE Clause]	DELETE EDGE `<edge_type>` `<vid>` -> `<vid>` [`@<rank>`] [, `<vid>` -> `<vid>` ...] DELETE VERTEX `<vid_list>`
拼接数据	JOIN	`|`

数据查询语言（DQL）

数据查询语言（DQL）语句用于执行数据查询。本节说明如何使用 SQL 语句和 nGQL 语句查询数据。

```
SELECT
 [DISTINCT]
 select_expr [, select_expr] ...
 [FROM table_references]
 [WHERE where_condition]
 [GROUP BY {col_name | expr | position}]
 [HAVING  where_condition]
 [ORDER BY {col_name | expr | position} [ASC | DESC]]
```

```
GO [[<M> TO] <N> STEPS ] FROM <node_list>
 OVER <edge_type_list> [REVERSELY] [BIDIRECT]
 [WHERE where_condition]
 [YIELD [DISTINCT] <return_list>]
 [| ORDER BY <expression> [ASC | DESC]]
 [| LIMIT [<offset_value>,] <number_rows>]
 [| GROUP BY {col_name | expr | position} YIELD <col_name>]

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

<edge_type_list>
   edge_type [, edge_type ...]

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


2021 Vesoft Inc.

数据控制语言（DCL）

数据控制语言（DCL）包含诸如 `GRANT` 和 `REVOKE` 之类的命令，这些命令主要用来处理数据库系统的权限，其他控件。

对比项	SQL	nGQL
创建用户	CREATE USER	CREATE USER
删除用户	DROP USER	DROP USER
更改密码	SET PASSWORD	CHANGE PASSWORD
授予权限	GRANT `<priv_type>` ON [object_type] TO `<user>`	GRANT ROLE `<role_type>` ON `<space>` TO `<user>`
删除权限	REVOKE `<priv_type>` ON [object_type] TO `<user>`	REVOKE ROLE `<role_type>` ON `<space>` FROM `<user>`

8.3.3 数据模型

查询语句基于以下数据模型：

MySQL

2021 Vesoft Inc.

Nebula Graph

8.3.4 增删改查（CRUD）

本节介绍如何使用 SQL 和 nGQL 语句创建（C）、读取（R）、更新（U）和删除（D）数据。

插入数据

```
mysql> INSERT INTO player VALUES (100, 'Tim Duncan', 42);

nebula> INSERT VERTEX player(name, age) VALUES 100: ('Tim Duncan', 42);
```

查询数据

Find the player whose id is 100 and output the `name` property:

```
mysql> SELECT player.name FROM player WHERE player.id = 100;

nebula> FETCH PROP ON player 100 YIELD player.name;
```

更新数据

```
mysql> UPDATE player SET name = 'Tim';

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

删除数据

```
mysql> DELETE FROM player WHERE name = 'Tim';

nebula> DELETE VERTEX 121;
nebula> DELETE EDGE follow 100 -> 200;
```


2021 Vesoft Inc.

8.3.5 示例查询

示例 1

返回年龄超过 36 岁的球员。

```
mysql> SELECT player.name
FROM player
WHERE player.age < 36;
```

使用 nGQL 查询有些不同，因为您必须在过滤属性之前创建索引。更多信息请参见 索引文档。

```
nebula> CREATE TAG INDEX player_age ON player(age);
nebula> REBUILD TAG INDEX player_age OFFLINE;
nebula> LOOKUP ON player WHERE player.age < 36;
```

示例 2

查找球员 Tim Duncan 并返回他效力的所有球队。

```
mysql> SELECT a.id, a.name, c.name
FROM player a
JOIN serve b ON a.id=b.player_id
JOIN team c ON c.id=b.team_id
WHERE a.name = 'Tim Duncan';
```

```
nebula> CREATE TAG INDEX player_name ON player(name);
nebula> REBUILD TAG INDEX player_name OFFLINE;
nebula> LOOKUP ON player WHERE player.name == 'Tim Duncan' YIELD player.name AS name | GO FROM $-.VertexID OVER serve YIELD $-.name, $$.team.name;
```

示例 3

查找球员 Tim Duncan 的队友。

```
mysql> SELECT a.id, a.name, c.name
FROM player a
JOIN serve b ON a.id=b.player_id
JOIN team c ON c.id=b.team_id
WHERE c.name IN (SELECT c.name
FROM player a
JOIN serve b ON a.id=b.player_id
JOIN team c ON c.id=b.team_id
WHERE a.name = 'Tim Duncan');
```

在 nGQL 中，我们使用管道将上一条语句的输出作为下一条语句的输入。

```
nebula> GO FROM 100 OVER serve YIELD serve._dst AS Team | GO FROM $-.Team OVER serve REVERSELY YIELD $$.player.name;
```

最后更新: 2020年7月21日

2021 Vesoft Inc.

8.4 点标识符和分区

本文档提供有关点标识符（简称 `VID`）和分区的一些介绍。

在 **Nebula Graph** 中，点是用点标识符（即 `VID`）标识的。插入点时，必须指定 `VID`（int64）。`VID` 可以由应用程序生成，也可以使用 **Nebula Graph** 提供的哈希函数生成。

`VID` 在一个图空间中必须唯一。即在同一个图空间中，拥有相同 `VID` 的点被当做同一个点。不同图空间中的 `VID` 彼此独立。此外，一个 `VID` 可以拥有多种 `TAG`。

向 **Nebula Graph** 集群中插入数据时，点和边会分布到不同的分区中，而这些分区又分布在多台机器上。应用程序如果希望将某些点落在同一个分区中（也即在同一台机器上），可根据以下公式自行控制 `VID` 的生成。

`VID` 和分区的对应关系为：

```
VID mod partition_number = partition ID + 1
```

其中，

- `mod` 是取模操作。
- `partition_number` 是 `VID` 所处图空间的的分区数量，即 CREATE SPACE 语句中 `partition_num` 的值。
- `partition ID` 即该 `VID` 所在分区的 ID。

例如，如果有 100 个分区，那 `VID` 为 1、11、101、1001 的点将存储在同一个分区上。

此外， `partition ID` 和机器之间的对应关系是随机的。因此不可以假设任何两个分区分布在同一台机器上。

最后更新: 2020年8月11日

2021 Vesoft Inc.

8.5 超级顶点（稠密点）处理

8.5.1 原理介绍

在图论中，超级顶点（稠密点）是指一个点有着极多的相邻边。相邻边可以是出边（我指向谁）或者是入边（谁指向我）。

由于幂律分布的特点，超级顶点现象非常普遍。 例如社交网络中的影响力领袖（网红大V）、证券市场中的热门股票、银行系统中的四大行、交通网络中的枢纽站、互联网中的高流量网站等、电商网络中的爆款产品。

在 **Nebula Graph** 中，一个 `点` 和其 `属性` 是一个 `Key-Value` （以该点的 `VID` 以及其他元信息作为 `Key` ），其 `Out-Edge Key-Value` 和 `In-Edge Key-Value` 都存储在同一个 partition 中(具体原理详见存储架构)，并且以 LSM-tree 的形式组织存放在硬盘（和缓存）中。

因此不论是 `从该点出发的有向遍历` ，或者 `以该点为终点的有向遍历` ，都会涉及到大量的 `顺序 IO 扫描` （最理想情况，当完成Compact操作之后），或者大量的 `随机 IO` （有关于 `该点` 和其 `出入边` 频繁的写入)。

经验上说，当一个点的出入度超过 1 万时，就可以视为是稠密点。需要考虑一些特殊的设计和处理。

重复属性索引

在属性图中，除了网络拓扑结构中的超级顶点，还有一类情况类似于超级顶点————某**属性有极高重复率**，也即"相同的 `点类型 Tag` ，不同的 `顶点 VID` ，同一属性字段，拥有相同属性值"。

Nebula Graph 原生属性索引的设计复用了存储模块 RocksDB 的功能，这种情况下的索引会被建模为 `前缀相同的 Key` 。对于该属性的查找，（如果未能命中缓存，）会对应为硬盘上的“一次随机查找 + 一次前缀顺序扫描”，以找到对应的 `点 VID` （此后，通常会从该顶点开始图遍历，这样又会发生该点对应 Key-Value 的一次随机读+顺序扫描）。当重复率越高，扫描范围就越大。

关于属性索引的原理详细介绍在博客《分布式图数据库 Nebula Graph 的 Index 实践》。

经验上说，当重复属性值超过 1 万时，也需要特殊的设计和处理。

建议的办法

数据库端的常见办法

1. 截断: 只访问一定阈值的边，超过该阈值的其他边则不返回。
2. 采样：对于每条边有一定概率的返回。
3. Compact：重新组织 RocksDB 中数据的排列方式，减少随机读，增加顺序读。

应用端的常见办法

根据业务意义，将一些超级顶点拆分：

- 删除多条边，合并为一条

例如，一个转账场景： (账户A)-[转账]->(账户B)。每次转账建模为一条AB之间的边，那么(账户A)和(账户B)之间会有着数万十次转账的场景。

按日、周、或者月为粒度，合并陈旧的转账明细。也就是批量删除陈旧的边，改为少量的边“月总额”和“次数。而保留最近月的转账明细。

- 拆分相同类型的边，变为多种不同类型的边

例如，"(机场)<-[depart]-(航班)"场景，每个架次航班的离港，都建模为一条航班和机场之间的边。那么大型机场的离港航班会极多。

根据不同的"航空公司"将 "depart" 这个边类型拆分更细的边类型，如 "depart_ceair", "depart_csair" 等。在查询（图遍历）时，指定离港的航空公司。

- 切分顶点本身

例如，对于"(人)-[借款]->(银行)"的借款网络，某大型银行A的借款次数和借款人会非常的多。

2021 Vesoft Inc.

可以将该大行节点 A 拆分为多个相关联的子节点 A1、A2、A3,

```
(人1)-[借款]->(银行A1), (人2)-[借款]->(银行A2), (人2)-[借款]->(银行A3);
(银行A1)-[属于]->(银行A), (银行A2)-[属于]->(银行A), (银行A3)-[属于]->(银行A).
```

这里的 A1、A2、A3 既可以是 A 真实的三个分行（例如北京、上海、浙江），也可以是三个按某种规则设立的虚拟分行， 例如按借款金额划分 A1: 1-1000, A2: 1001-10000, A3: 10000+。 这样，查询时对于 A 的任何操作，都转变为为对于 A1、A2、A3 的三次单独操作。

最后更新: 2021年5月28日

2021 Vesoft Inc.

https://docs.nebula-graph.com.cn/1.2.1

2021 Vesoft Inc.