



NebulaGraph

NebulaGraph Database 手册

master

吴敏, 周瑶, 梁振亚, 黄凤仙

2023 vesoft Inc.

Table of contents

1. Macro Rendering Error	6
2. 简介	7
2.1 图	7
2.2 图数据库的市场概况	27
2.3 相关技术	38
2.4 Macro Rendering Error	51
2.5 数据模型	52
2.6 路径	54
2.7 点 VID	56
2.8 服务架构	58
3. 快速入门	74
3.1 快速入门概览	74
3.2 从云开始（免费试用）	80
3.3 本地部署	95
3.4 nGQL 命令汇总	110
4. nGQL 指南	130
4.1 nGQL 概述	130
4.2 数据类型	145
4.3 变量和复合查询	164
4.4 运算符	169
4.5 函数和表达式	182
4.6 通用查询语句	227
4.7 子句和选项	265
4.8 图空间语句	294
4.9 Tag 语句	302
4.10 Edge type 语句	310
4.11 点语句	316
4.12 边语句	323
4.13 原生索引	330
4.14 全文索引	341
4.15 子图和路径	349
4.16 查询调优与终止	356
4.17 Macro Rendering Error	362
5. 安装部署	363
5.1 Macro Rendering Error	363

5.2 编译与安装	364
5.3 存算合并版NebulaGraph	374
5.4 <i>Macro Rendering Error</i>	376
5.5 连接NebulaGraph服务	377
5.6 管理 Storage 主机	379
5.7 卸载NebulaGraph	380
6. 配置与日志	382
6.1 配置	382
6.2 日志	388
7. 监控	390
7.1 <i>Macro Rendering Error</i>	390
7.2 RocksDB 统计数据	391
8. 数据安全	392
8.1 验证和授权	392
8.2 SSL 加密	395
9. 备份与恢复	397
9.1 管理快照	397
10. 同步与迁移	399
10.1 BALANCE	399
11. 最佳实践	400
11.1 Compaction	400
11.2 <i>Macro Rendering Error</i>	402
11.3 图建模设计	403
11.4 系统设计建议	407
11.5 执行计划	408
11.6 超级顶点（稠密点）处理	409
11.7 启用 AutoFDO	411
11.8 实践案例	417
12. 客户端	419
12.1 客户端介绍	419
12.2 NebulaGraph Console	420
12.3 NebulaGraph CPP	424
12.4 NebulaGraph Java	426
12.5 NebulaGraph Python	428
12.6 NebulaGraph Go	430
13. NebulaGraph Cloud	431
13.1 什么是 NebulaGraph Cloud	431
13.2 Nebula Graph Cloud 阿里云版	432

14. NebulaGraph Dashboard (社区版)	439
14.1 什么是 NebulaGraph Dashboard (社区版)	439
14.2 部署 Dashboard 社区版	441
14.3 连接 Dashboard	444
14.4 Dashboard 页面介绍	445
14.5 <i>Macro Rendering Error</i>	451
15. NebulaGraph Importer	452
15.1 NebulaGraph Importer	452
15.2 有表头配置说明	459
15.3 无表头配置说明	462
16. NebulaGraph Exchange	465
16.1 认识 NebulaGraph Exchange	465
16.2 获取 NebulaGraph Exchange	468
16.3 参数说明	470
16.4 使用 NebulaGraph Exchange	472
16.5 Exchange 常见问题	553
17. NebulaGraph Operator	557
17.1 什么是 NebulaGraph Operator	557
17.2 使用流程	559
17.3 <i>Macro Rendering Error</i>	560
17.4 部署 NebulaGraph	561
17.5 通过 Nebular Operator 连接NebulaGraph	563
17.6 配置 NebulaGraph	568
17.7 <i>Macro Rendering Error</i>	575
17.8 NebulaGraph集群的滚动更新策略	576
17.9 故障自愈	577
17.10 常见问题	578
18. 图计算	579
18.1 算法简介	579
18.2 NebulaGraph Algorithm	593
18.3 NebulaGraph Analytics License	598
19. NebulaGraph Spark Connector	600
19.1 版本兼容性	600
19.2 适用场景	601
19.3 特性	601
19.4 更新说明	601
19.5 获取 NebulaGraph Spark Connector	601
19.6 使用方法	602

20. NebulaGraph Flink Connector	607
20.1 适用场景	607
20.2 更新说明	607
21. NebulaGraph Bench	608
21.1 适用场景	608
21.2 更新说明	608
22. 附录	609
22.1 <i>Macro Syntax Error</i>	609
22.2 <i>Macro Rendering Error</i>	610
22.3 <i>Macro Rendering Error</i>	611
22.4 导入工具选择	612
22.5 如何贡献代码和文档	613
22.6 <i>Macro Rendering Error</i>	617
22.7 思维导图	618
22.8 错误码	622

1. Macro Rendering Error

UndefinedError: 'comm' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 35, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'comm' is undefined
```

最后更新: May 4, 2023

2. 简介

- 第一课：图的概念（03 分 45 秒）

- 第二课：图的结构（02 分 24 秒）

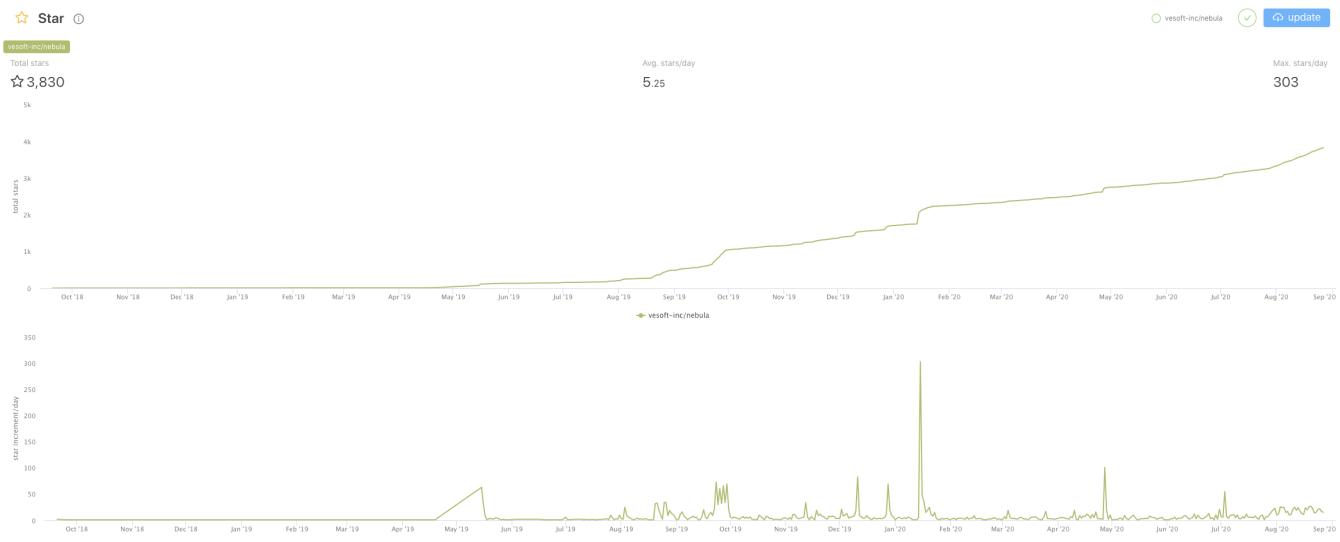
2.1 图

当前，从计算机行业巨头（例如 Amazon 和 Facebook）到小型研究团队，都投入了大量的资源探索图数据库在解决各种数据关系问题上的潜力。当然你也可以选择像他们这样进行尝试，现在可供选择的数据库有很多。那么图数据库究竟是什么？它可以做些什么？作为一类数据库，它在数据库领域里处于什么位置呢？要回答这些问题，我们首先得了解图。

图是计算机科学研究的主要领域之一。图能够高效地解决目前存在的诸多问题。本章将从图说起，继而说明图数据库的优点及其在现代应用程序开发中的巨大潜力，然后介绍分布式图数据库的区别和几种其他类型的数据库。

2.1.1 图、图片与图论

图无处不在。当听到图这个词时，很多人都会想到条形图或折线图，因为有时候我们确实会把它们称作图。从传统意义上来说，图是用来展示两个或多个数据系统之间的联系的。最简单的例子如下图，下图展示了NebulaGraph GitHub 仓库星星数量随时间推移的变化。

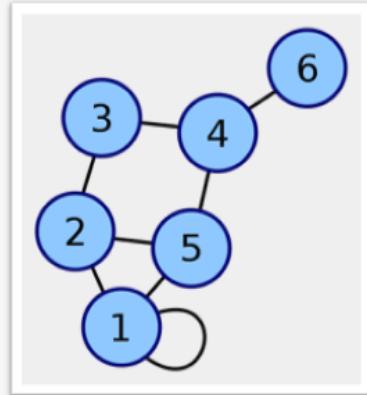


这是相对比较简单的一种图，横轴为时间，纵轴为星星数量。可以看到，星星数量是随着时间推移而上升的。这种类型的图通常称为折线图。折线图可以显示随时间（根据常用比例设置）而变化的连续数据。此处我们只给出了折线图的例子。当然图的形式有多种，比如饼图、条形图等。

还有一种“图”在日常口语中会更多的被提及，例如，“图像识别”，“美图秀”，“修图”等。例如下“图”的左边。



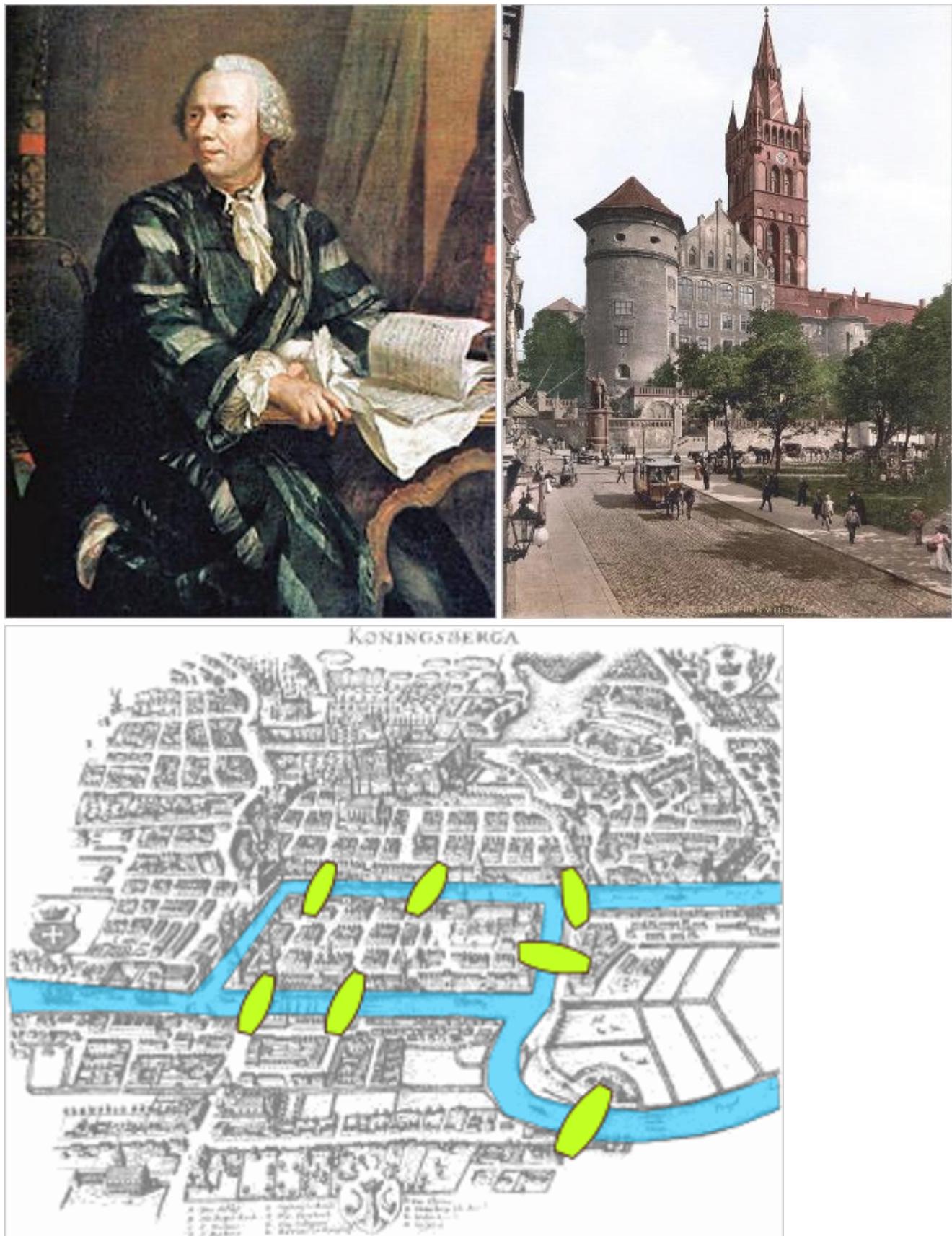
Graph: Image, Visual



Graph: Network, Connection, Linked Data

但是——总会有但是——我们在本书中讨论的图是另外一个概念——“图论”中的图。

图论始于 18 世纪初期的柯尼斯堡七桥问题。柯尼斯堡当时是普鲁士的城市，普雷格尔河穿过柯尼斯堡，不仅把柯尼斯堡分成了两部分，而且还在河中间形成了两个小岛。这就将整个城市分割成了四个区域，各区域由七座桥连接。在所有的桥都只能走一遍的前提下，如何能把这个地方所有的桥都走一遍呢？



有兴趣的话可以试试寻找这个小游戏的答案¹。下图为柯尼斯堡七座桥的简化图。



大数学家欧拉于 1735 年提出，并没有方法能圆满解决这个问题，并在第二年发表论文，证明符合条件的走法并不存在。这篇论文在圣彼得堡科学院发表，成为图论史上第一篇重要文献。

在论文中，欧拉把实际的抽象问题简化为平面上的点与边组合，将城市的四个区域抽象成点，将连接城市的七座桥抽象成连接点的边。



图中四个点代表柯尼斯堡的四个区域，点之间的线代表连接四个区域的七座桥。从图中不难看出，偶数座桥连接的区域可以轻松通过，因为来去可以选择不同的路线。奇数座桥连接的区域只能作为起点或者终点，因为同样的路线只能走一次。和节点相关联的边的条数称为节点度。现在可以证明，只有两个节点有奇数度，另外节点有偶数度时，也即两个区域必须有偶数座桥，剩下的区域有奇数座桥时，柯尼斯堡问题才能解决。欧拉论述了，由于柯尼斯堡七桥问题中所有区域都为奇数度，它无法实现符合题意的遍历。

欧拉发表的相关论文被认为是图论领域的第一篇文章，因此普遍认为欧拉是图论的创始人。

图论中的相关概念，我们将在后面的学习中提到。简单来说，图论就是研究图的学问。图是基本研究对象，用于表示实体与实体之间的关系。

在数学的分支图论中，图（Graph）是基本研究对象。在中文中，强调为“拓扑图”、“网络图”等。这一名词最早由西尔维斯特在 1878 年提出。他是著名的英国数学家、牛津大学几何教授，用图来表示数学和化学分子结构之间的关系。

一张图由一些小圆点（称为顶点或节点，即 Vertex）和连接这些圆点的直线或曲线（称为边，即 Edge）组成。



2.1.2 属性图

从数学角度来说，图论是研究建模对象之间关系结构的学科。但是从工业界使用的角度，通常会对基础的图模型进行扩展，称为属性图模型。属性图通常由以下几部分组成：

- 节点，即对象或实体。在本书中，通常简称为点（Vertex）。
- 节点之间的关系，在本书中，通常简称为边（Edge）。通常边是有方向或者无方向的，以表示两个实体之间有持续的关系。



A.无向图

B.有向图

- 此外，在节点和边上，还可以有属性（properties）。

在现实生活中，有很多属性图的例子。

在网络理论中，图可以用来做可视化的社会网络分析，研究社会实体之间的关系结构。例如企查查或者 BOSS 直聘这类的公司，用图来建模商业股权关系网络。这个网络中，点通常是一个自然人或者是一家企业，边通常是某自然人与某企业之间的股权关系。点上的属性可以是自然人姓名、年龄、身份证号等。边上的属性可以是投资金额、投资时间、董监高等职位关系。



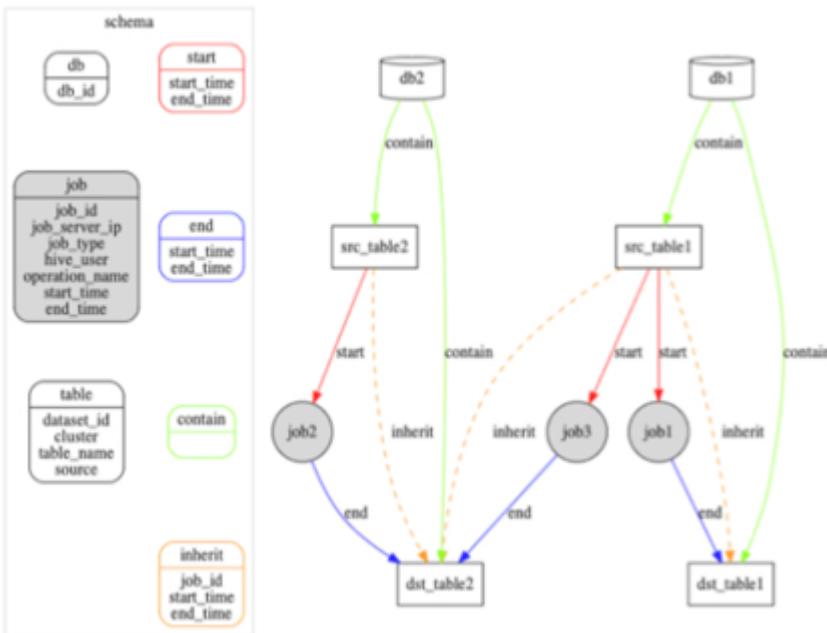
在一个股票市场里面，点可以是一家上市公司，边可以是上市公司之间的相关性。点的属性可以为股票代码、简称、市值、板块等；边的属性可以为股价的时间序列相关性系数²。



图关系还可以是类似《权力的游戏》这样电视剧中的人物关系网³: 点为人物, 边为人物之间的互动关系; 点的属性为人物姓名、年龄、阵营等, 边的属性(距离)为两个人物之间的互动次数, 互动越频繁距离越近。



图也可以用于 IT 系统内部的治理。例如，对于像微众银行这样的公司，通常有着非常庞大的数据仓库，以及相应的数仓管理工具。这些管理工具记录了数仓内 Hive 表之间通过 Job 实现的 ETL 关系⁴，这样的 ETL 关系，可以非常方便的用图的形式呈现和管理，当出现问题时也可以非常方便地追溯根源。



图也可以用于记录一个大型 IT 系统内部错综复杂的微服务之间的调用关系⁵，运维团队用其进行服务治理。这里每个点表示一个微服务，边表示两个微服务之间的调用关系；这样，运维人员可以方便地寻找可用性低于阈值 (99.99%) 的调用链路，或者发现那些出故障会影响面特别大的微服务节点。

服务治理

- 图谱数据
 - 将RPC服务调用关系写入图谱
 - 包含service、api、team等4类实体及5类关系
 - 点边数量在百万级别，实时写入
 - 用于服务链路治理和告警优化

```
//查找API com.sankuai.ia.search.api:SearchControllerV2.search过去七天可用率低于99.9%的链路的thrift调用，最大图遍历深度为10
GO 1 TO 10 STEPS FROM
hash('com.sankuai.ia.search.api:SearchControllerV2.
search') OVER call WHERE call.availability<0 AND
call.availability<1000000 AND
$$.api.type=="mtthrift" YIELD call._src,call._dst
```



```
//查找所有java类型服务提供的API，并统计其会影响的上游API的数量，从高到低排序看影响次数大小（调用的可用率小于4个9）
LOOKUP ON service WHERE service.type=="java"
| GO FROM $-.VertexID OVER provider YIELD
provider._dst AS java_api_id
| GO FROM $-.java_api_id OVER call REVERSELY WHERE
call.availability>0 AND call.availability<1000000
YIELD call._src AS api_src, call._dst AS api_dst
| GROUP BY $-.api_src YIELD $-.api_src AS api_id,
count(1) AS call_cnt
| ORDER BY call_cnt DESC
| FETCH PROP ON api $-.api_id YIELD
api.appkey,api.method,$-.call_cnt
```

图也可以用于提升代码开发效率。用图存放代码之间的函数调用关系⁵，可以提升研发团队审查和测试代码的效率。在这样的图中，每个点是代码中的一个函数或者变量，每个边是函数或者变量之间的调用关系。当有新提交的代码之时，人们可以更方便的看到可能会受到影响的其他接口，这样可以帮助测试人员更好的评估潜在的上线风险。

代码依赖分析

- 图谱数据
 - 将公司代码库中代码的依赖关系写入图谱
 - 包含method, field, class, interface等4类顶点，?类关系
 - 点边数量在千万级别，实时写入
- 用于QA精准测试
 - PR向代码仓库提交PR后，能查询出所修改代码能影响到的外部接口，并展示调用路径

```
//查找最外层method到某个method的所有无环路径
(GO 1 to 30 STEPS FROM 2946345526231222882 OVER
method_call_method REVERSELY YIELD DISTINCT
method_call_method._dst AS id
MINUS
GO 1 to 30 STEPS FROM 2946345526231222882 OVER
method_call_method REVERSELY YIELD DISTINCT
method_call_method._src as id )
| FIND NOLOOP path FROM $-.id TO
2946345526231222882 OVER method_call_method UPTO
30 STEPS
```



```
//确认两个method间是否有路径
FIND SINGLE SHORTEST PATH FROM hash("method1")
TO hash("method2") OVER method_call_method UPTO
30 STEPS
```

此外，相对于静态不发生变化的属性图，我们还可以通过增加一些时间信息，发掘出更多的使用场景。

例如，在一个银行间账户资金流向网络里面⁶，点是账户，边是账户之间的转账记录。边属性记录了转账的时间、金额等。同盾、邦盛、半云科技等公司采用图技术，可以方便地通过图的方式探索发现明显的资金挪用、“以贷还贷”、“团伙贷款”等现象。



同样的方法也可以用于探索发现加密货币的流向。



在一个黑产账户和设备网络中⁷，其中的点可以是账户、手机设备和 WIFI 网络，边是这些账户与手机设备之间的登录关系，以及手机设备和 WIFI 网络之间的接入关系。



这些登录记录的网络构成了黑产群体网络的团伙作案特征。360 数科⁷、快手⁸、微信⁹、知乎¹⁰、携程金融这些公司都通过图技术实时（毫秒级的）识别超过百万个的黑产社群。



更进一步，除了时间这个维度外，我们通过添加一些地理位置信息，还能发现属性图更多的应用场景。

例如新冠病毒的流行病学溯源¹¹，点是人物，边是人与人之间的接触；点属性为人物的身份证号、发病时间等信息，边属性为人物之间发生密切接触的时间和地理位置等。为卫生防疫部门快速识别高风险人群和其行为轨迹提供帮助。



地理位置与图的结合也可以用于一些 O2O 的场景，例如基于 POI（Point-of-Interest）的实时美食推荐¹²，使得美团这类本地生活服务平台公司能在消费者在打开 APP 的时候，实时推荐出更为合适的商家。





图还可以用于更深度的知识推理，华为、vivo、OPPO、微信、美团等公司，将图用于表征底层知识关系的数据模型。

由此可见，图可用于在物理、生物、社会和信息系统中建模许多类型的关系和过程，许多实际问题可以用图来表示。因此，图论成为运筹学、控制论、信息论、网络理论、社会科学、语言学、计算机科学等众多学科强有力的数据工具。

2.1.3 为什么要使用图数据库

虽然关系型数据库与 XML/JSON 等半结构类型的数据库，都可以用来描述图结构的数据模型，但是，图（数据库）不仅可以描述图结构与存储数据本身，更着眼于处理数据之间的关联（拓扑）关系。具体来说，图（数据库）有这么几个优点：

- 图是一种更直观、更符合人脑思考直觉的知识表示方式。这使得我们在抽象业务问题时，可以着眼于“业务问题本身”，而不是“如何将问题描述为数据库的某种特定结构（例如表格结构）”。

- 图更容易展现数据的特征，例如转账的路径、近邻的社区。例如，如果要分析《权力的游戏》中的人物派别关系和人物重要性，表的组织方式如下：



The screenshot shows a software interface for managing a graph database. At the top, there's a toolbar with icons for '节点' (Node), '边' (Edge), '配置' (Configure), '添加节点' (Add Node), '添加边' (Add Edge), '搜索/替换' (Search/Replace), '输入电子表格' (Input Excel Table), '输出表格' (Output Table), and '更多功能' (More Functions). A search bar labeled '过滤:' (Filter) and a column header 'Id' are also present. The main area is a table with the following columns: Id, Label, Interval, Modularity Class, betweenness, and community. The data rows represent various characters from the Game of Thrones universe, such as Arya Stark, Gendry, and Robb Stark, along with their respective values for the listed metrics.

Id	Label	Interval	Modularity Class	betweenness	community
-4364237027732478972	Arya-Stark	4	0.017495	0	0
-4973959390558533367	Gendry	0	0.0	0	0
-7025436182975816947	Hallis-Mollen	5	0.0	0	0
8249703968876499989	Raymun-Darry	0	0.0	0	0
-9085767779232784107	Hobb	3	0.0	1	1
7505218728385384214	Randyll-Tarly	3	0.0	1	1
5114251854412197144	Viserys-Targaryen	7	0.002858	2	2
-2958304360028784612	Jonos-Bracken	2	0.0	3	3
-435991958697142177	Lancel-Lannister	0	0.0	0	0
7195357273749773604	Robb-Stark	5	0.072984	0	0
1075531505424338989	Marq-Piper	4	0.006237	0	0
-3670573862518875597	Addam-Marbrand	2	0.0	0	0
3931316780000927284	Robert-Arryn	2	0.0	0	0
482603504180653879	Irri	7	0.000096	2	2
7181995877190281794	Jommo	7	0.0	2	2
1787258805162148944	Eddard-Stark	0	0.269604	0	0
-558158798715221422	Danwell-Frey	1	0.021389	4	4
-2434779847791595692	Mordane	4	0.001056	0	0
8409318129262644314	Clydas	3	0.0	1	1
-1738391870688186527	Karyl-Vance	2	0.010753	5	5
6902668443292429674	Hago	7	0.000068	2	2
95861621849685359	Mya-Stone	2	0.0	0	0
2025059960255100299	Porther	0	0.0	0	0
-1967336888960310122	Colemon	2	0.0	0	0
5894253429758865049	Ilyn-Payne	4	0.00032	0	0
-1356662602605843742	Pycelle	0	0.000321	0	0
382702656484230079	Gared	6	0.004328	1	1
1306559795724859089	Jorah-Mormont	7	0.012611	2	2
-96410199981447977	Cayn	0	0.000022	0	0
7160112206884551800	Ghella	2	0.0	0	0

Below the table are several buttons for data manipulation: '添加列' (Add Column), '合并列' (Merge Columns), '删除列' (Delete Column) with a dropdown, '清除列' (Clear Column) with a dropdown, '复制数据到其它列' (Copy Data to Other Column) with a dropdown, '填写数值到列' (Fill Value to Column) with a dropdown, '复制列数据' (Copy Column Data) with a dropdown, '从正则表达式中新建一个布尔列' (Create a Boolean Column from a Regular Expression) with a dropdown, '新建一列 (列表或者正则表达式匹配组合)' (Create a New Column (List or Regular Expression Match Combination)) with a dropdown, '布尔值求反' (Invert Boolean Value) with a dropdown, and '列转换为动态' (Convert Column to Dynamic) with a dropdown.

这显然不如下方图的组织方式直观：



特别是当某些中心节点被删除:



或者，增加一条边，可以彻底地改变整个图拓扑：



虽然只是个别数据的细微改变，图可以比表更直观地表现其中的重要而系统的信息。

- 图查询语言是针对图结构访问设计的，可以更加直观。例如，下面是一个 LDBC 中的查询示例，要求：查找某人（Person）在社交网络上发布的帖子（Posts）；查找相应的回复（Message，回复本身还会被多次回复）；发帖时间、回帖时间都满足一定条件；根据回帖数量对结果排序。



如果使用 PostgreSQL 编写查询语句：

```
--PostgreSQL
WITH RECURSIVE post_all(psa_threadid
    , psa_thread_creatorid, psa_messageid
    , psa_creationdate, psa_messagestype
    ) AS (
    SELECT m_messageid AS psa_threadid
        , m_creatorid AS psa_thread_creatorid
        , m_messageid AS psa_messageid
        , m_creationdate, 'Post'
    FROM message
    WHERE 1=1 AND m_c_replyof IS NULL -- post, not comment
    AND m_creationdate BETWEEN :startDate AND :endDate
UNION ALL
    SELECT psa.psa_threadid AS psa_threadid
        , psa.psa_thread_creatorid AS psa_thread_creatorid
        , m_messageid, m_creationdate, 'Comment'
    FROM message p, post_all psa
    WHERE 1=1 AND p.m_c_replyof = psa.psa_messageid
    AND m_creationdate BETWEEN :startdate AND :endDate
)
SELECT p.p_personid AS "person.id"
    , p.p_firstname AS "person.firstName"
    , p.p_lastname AS "person.lastName"
    , count(DISTINCT psa.psa_threadid) AS threadCount
END) AS messageCount
    , count(DISTINCT psa.psa_messageid) AS messageCount
FROM person p LEFT JOIN post_all psa ON (
    1=1 AND p.p_personid = psa.psa_thread_creatorid
    AND psa_creationdate BETWEEN :startdate AND :endDate
)
GROUP BY p.p_personid, p.p_firstname, p.p_lastname
ORDER BY messageCount DESC, p.p_personid
LIMIT 100;
```

如果使用为图专门设计的图语言 Cypher 编写查询语句：

```
--Cypher
MATCH (person:Person)-[:HAS_CREATOR]->(post:Post)-<[:REPLY_OF*0..]->(reply:Message)
WHERE post.creationDate >= $startDate AND post.creationDate <= $endDate
    AND reply.creationDate >= $startDate AND reply.creationDate <= $endDate
RETURN
    person.id, person.firstName, person.lastName, count(DISTINCT post) AS threadCount,
    count(DISTINCT reply) AS messageCount
ORDER BY
    messageCount DESC, person.id ASC
LIMIT 100
```

- 由于存储引擎和查询引擎可以针对图的结构专门设计，图的遍历（对应 SQL 中的 join）要高效得多。下图是知名产品 Neo4j 所做的一个对比¹²。

深度	关系型数据库的执行时间(s)	Neo4j的执行时间(s)	返回的记录条数
2	0.016	0.01	~2500
3	30.267	0.168	~110000
4	1543.505	1.359	~600000
5	未完成	2.132	~800000

关系数据库 vs 图数据库(多跳查询)

- 图数据库具有广泛的适用场景。例如数据集成（知识图谱）、个性化推荐、欺诈与威胁检测、风险分析与合规、身份（与控制权）验证、IT 基础设施管理、供应链与物流、社交网络研究等。
- 根据文献¹³ 的统计，使用图技术最多的领域，依次是：信息技术(IT)、学术界研究、金融、工业界实验室、政府、医疗健康、国防、制药业、零售与电子商务、交通运输、电信、保险。
- 2019 年，根据 Gartner 的问卷调研，27% 的客户（500 组）在使用图数据库，20% 有计划使用。

2.1.4 RDF

受篇幅所限，本章不讨论 RDF 数据模型。

1. 图片来源 <https://medium.freecodecamp.org/i-dont-understand-graph-theory-1c96572a1401>. ↪
2. <https://nebula-graph.com.cn/posts/stock-interrelation-analysis-jgrapht-nebula-graph/> ↪
3. <https://nebula-graph.com.cn/posts/game-of-thrones-relationship-networkx-gephi-nebula-graph/> ↪
4. <https://nebula-graph.com.cn/posts/practicing-nebula-graph-webank/> ↪
5. <https://nebula-graph.com.cn/posts/meituan-graph-database-platform-practice/> ↪ ↪
6. <https://zhuanlan.zhihu.com/p/90635957> ↪
7. <https://nebula-graph.com.cn/posts/graph-database-data-connections-insight/> ↪ ↪
8. <https://nebula-graph.com.cn/posts/kuaishou-security-intelligence-platform-with-nebula-graph/> ↪
9. <https://nebula-graph.com.cn/posts/nebula-graph-for-social-networking/> ↪
10. <https://mp.weixin.qq.com/s/K2QinpR5Rplw1teHpHtf4w> ↪
11. <https://nebula-graph.com.cn/posts/detect-corona-virus-spreading-with-graph-database/> ↪
12. <https://nebula-graph.com.cn/posts/meituan-graph-database-platform-practice/> ↪ ↪
13. <https://arxiv.org/abs/1709.03188> ↪

最后更新: May 4, 2023

2.2 图数据库的市场概况

既然已经讨论了什么是图，接下来让我们进一步认识基于图论和属性图模型发展起来的图数据库。

不同的图数据库在术语方面可能会略有不同，但是归根结底都是在讲点、边和属性。至于更多的功能，例如标签、索引、约束、TTL、长任务、存储过程和UDF等这些高级功能，在不同图数据库中，会存在明显的差异。

图数据库用图来存储数据，而图是最接近高度灵活、高性能的数据结构之一。图数据库是一种专门用于存储和检索庞大信息网的存储引擎，它能够高效地将数据存储为点和边，并允许对这些点边结构进行高性能的检索和查询。我们也可以为这些点和边添加属性。

图数据库几乎适用于存储所有领域的数据。因为在几乎所有领域中，事物之间都是由某种相关联的。图数据库支持存储实体之间的丰富关系，并且能够将这些关系完美地呈现出来，而无需像其他建模方式那样，将关系也当成实体存储。因此图数据库能够以最接近对数据直观认知的形式存储数据。

2.2.1 三方机构的统计和预测

DB-Engines 的统计

根据世界知名的数据库排名网站DB-Engines.com的统计，图数据库至2013年以来，一直是“增速最快”的数据库类别¹。

该网站根据一些指标来统计每种类别的数据库的流行度变化趋势，这些指标包括基于Google等搜索引擎的收录和趋势情况、主要IT技术论坛和社交网站上讨论的技术话题、招聘网站的职位变化等。该网站共收录了371种数据库产品，并分为12个类别。这12个类别中，图数据库这种类别的增速远远快于其他任何的类别。



Gartner 的预测

世界顶级智库Gartner早在2013年之前²，就将图数据库作为主要的“商业智能与分析技术趋势”。在那个时候，Big Data正火热的如日中天，数据科学家更是炙手可热的职位。

Figure 1. Hype Cycle for Business Intelligence and Analytics, 2013



直到最近，图数据库及相关的图技术依旧是“2021年十大数据与分析趋势”³：

Gartner Top 10 Data and Analytics Trends, 2021



Accelerating Change

- 1** Smarter, Responsible, Scalable AI
- 2** Composable Data and Analytics
- 3** Data Fabric Is the Foundation
- 4** From Big to Small and Wide Data



Operationalizing Business Value

- 5** XOps
- 6** Engineering Decision Intelligence
- 7** D&A as a Core Business Function



Distributed Everything

- 8** Graph Relates Everything
- 9** The Rise of the Augmented Consumer
- 10** D&A at the Edge

gartner.com/SmarterWithGartner

Source: Gartner
© 2021 Gartner, Inc. All rights reserved. CTMKT_1164473

Gartner

趋势八：图技术使一切产生关联（Graph Relates Everything）

图技术已成为许多现代数据和分析能力的基础，能够在不同的数据资产中发现人、地点、事物、事件和位置之间的关系。数据和分析领导者依靠图技术快速回答需要在了解情况并理解多个实体之间的联系和优势的性质后才能回答的复杂业务问题。

Gartner预测，到2025年图技术在数据和分析创新中的占比将从2021年的10%上升到80%。该技术将促进整个企业机构的快速决策。

可以注意到，Gartner 的预测比较好的吻合了 DB-Engines 的统计结论。技术的进步并不是完全线性的，通常会有一段快速发展的泡沫期，然后进入一段平台期，之后由于新的技术的出现产生新一轮的泡沫期，再经历一段平台期。以此往复螺旋形的循环发展。

对于市场规模的预测

根据 verifiedmarketresearch⁴, fnfresearch⁵, marketsandmarkets⁶, 以及 gartner⁷ 等智库的统计和预测, 图数据库市场 (包括云服务) 规模在2019年大约是8亿美元, 将在未来6年保持25%左右的年复合增长(CAGR)至 30-40 亿美元, 这大约对应于全球数据库市场 5-10% 的市场份额。



2.2.2 市场参与者

(第一代) 图数据库的先行者 **Neo4j**

虽然在 1970 年代, 人们已经提出了一些类似于“图”的数据模型和产品原型 (例如 CODASYL⁸) 和相应的图语言 G/G+ 语言⁹。但真正能够让“图数据库”这个概念流行起来, 不得不说到这个市场最主要的先行者 Neo4j, 甚至(标签)属性图和图数据库这两个主要术语就是 Neo4j 最早提出并实践的。

本小节关于 Neo4j 和其创造的图查询语言 Cypher 的历史内容主要摘录自 ISO WG3 的工作论文 “An overview of the recent history of Graph Query Languages”¹⁰ 和⁹, 本书作者根据最新两年的进展有删减和更新。

关于图查询语言 (Graph Query Language, GQL) 和国际标准的制定

熟悉数据库的读者可能都知道结构化查询语言SQL。通过使用SQL, 人们以接近自然语言的方式访问数据库。在 SQL 被广泛采用和标准化之前, 关系型数据库的市场是非常碎片和割裂的——各家厂商的产品都有完全不同的接入访问方式, 数据库产品自身的开发人员、数据库产品周边工具的开发人员、数据库最终的使用人员, 都不得不学习各个厂商的完全不同的产品, 在不同产品之间迁移极其困难。当1989年SQL-89标准被制定后, 整个关系型数据库的市场快速收敛到SQL-89上。这大大降低了上述各种人员的学习曲线。

类似的, 在图数据库领域, 图语言(GQL)承担了类似于SQL的作用, 是一种用户与图数据库主要的交互方式。但不同于SQL-89这种国际标准, GQL还没有任何国际标准。目前有两种主流的图语言:

Neo4j的Cypher (及其后续——ISO正在制定过程中的 GQL-standard 草案)和Apache TinkerPop的Gremlin。前者通常被称为声明式语言 (Declarative query language)——即用户只需要告诉系统“要什么”, 而不管“怎么做”; 后者通常被称为命令式语言 (Imperative query language), 用户会显式地指定系统的操作。

GQL国际标准正在制定过程中。

年表简述

- 2000 年, Neo4j 的创始人产生将数据建模成网络 (network) 的想法。
- 2001 年, Neo4j 开发了最早的核心部分代码。
- 2007 年, Neo4j 开始以一个公司的方式运作。
- 2009 年, Neo4j 团队借鉴 XPath 作为图查询语言, Gremlin¹¹最初也是基于这个想法。
- 2010 年, Neo4j 的员工 Marko Rodriguez 采用术语属性图 (Property Graph) 来描述 Neo4j 和 Tinkerpop / Gremlin 的数据模型。
- 2011 年, 第一个公开发行版本 Neo4j 1.4; 并发布了 Cypher 的第一个版本。
- 2012 年, Neo4j 1.8 为 Cypher 增加写入图的能力。Neo4j 2.0 增加了标签和索引, Cypher 成为一种声明式的语言。
- 2015 年, Neo4j 将 Cypher 开源为 openCypher。
- 2017 年, ISO WG3 工作组开始讨论如何将属性图查询能力引入 SQL。
- 2018 年 12 月, 从 Neo4j 3.5 开始其核心部分转为闭源。
- 2019 年, ISO 正式立项两个项目(ISO/IEC JTC 1 N 14279 和 ISO/IEC JTC 1/SC 32 N 3228), 启动关于图数据库语言国际标准的制定工作。
- 2021 年, Neo4j 完成 F 轮 3.25 亿美元的融资, 是整个数据库 (包括关系型) 历史上最大一轮融资。

NEO4J 的早期历史

Neo4j 和属性图这种数据模型, 最早构想于 2000 年。Neo4j 的创始人们当时在开发一个媒体管理系统, 所使用的数据库的 schema 经常会发生重大变化。为了支持这种灵活性, Neo4j 的联合创始人 Peter Neubauer, 受到 Informix Cocoon 的启发, 希望将系统能够建模为一种概念相互连接的网络。印度理工学院孟买分校的一群研究生们实现了最早的原型。Neo4j 的联合创始人 Emil Eifrem 和这些学生们花了一周的时间, 将 Peter 最初的想法扩展成为一个更抽象的模型: 节点通过关系连接, key-value 作为节点和关系的属性。这群人开发了一个 Java API 来和这种数据模型交互, 并在关系型数据库之上实现了一个抽象层。

虽然这种网络模型极大的提高了生产力, 但是性能一直很差。所以 Neo4j 联合创始人 Johan Svensson 花了不少精力, 为这种网络模型实现了一个原生的数据管理系统。这个就成为了 Neo4j。在最初的几年, Neo4j 作为一个内部产品很成功。在 2007 年, Neo4j 的知识产权转移给了一家独立的数据库公司。

在 Neo4j 的第一个公开发行版中 (Neo4j 1.4, 2011 年), 数据模型由节点和有类型的边构成, 节点和边都有 key-value 组成的属性。Neo4j 的早期版本没有任何的索引, 应用程序只能从根节点开始自己构造查询结构 (search structure)。因为这样对于应用程序非常笨重, Neo4j 2.0 (2013.12) 引入了一个新概念——点上的标签 (label)。基于点标签, Neo4j 可以为一些预定义的节点属性建立索引。

"节点"、"关系"、"属性"、"关系只能有一个标签"、"节点可以有零个或者多个标签", 以上这些概念构成了 Neo4j 属性图的数据模型定义。随着后来增加的索引功能, 让 Cypher 成为了与 Neo4j 交互的主要方式。因为这样应用程序开发者只需要关注于数据本身, 而不是上段提到的那个开发者自己构建的查询结构 (search structure)。

GREMLIN 的创造

Gremlin 是基于 Apache TinkerPop 开发的图语言, 其风格接近于一连串的函数 (过程) 调用。最初 Neo4j 的查询方式是通过 Java API。应用程序可以将查询引擎作为库(library)嵌入到应用程序中, 然后使用 API 来查询图。

就在这段时间, NoSQL 这个概念开始出现。NoSQL 型的数据库引擎一般用 REST 和 HTTP 来交互和查询。Neo4j 的早期员工 Tobias Lindaaker、Ivarsson、Peter Neubauer、Marko Rodriguez 用 XPath 作为图查询, Groovy 提供循环结构, 分支和计算 (等图灵完备的功能)。这个就是 Gremlin 最初的原型。2009 年 11 月发布了第一个版本。

后来, Marko 发现同时用两种不同的解析器 (XPath 和 Groovy) 有很多问题, 就将 Gremlin 改为基于 Groovy 的一种领域特定语言 (DSL)。

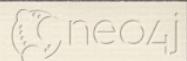
CYPHER 的创造

Gremlin 和 Neo4j 的 Java API 一样, 最初用于表达如何查询数据库的一种过程 (Procedural)。它允许更短的语法来表达查询, 也允许通过网络远程访问数据库。Gremlin 这种过程式的特性, 需要用户知道如何采用最好的办法查询结果, 这样对于应用程序开发人员来说仍旧有负担。与此同时, 在过去 30 年中, 声明式语言 SQL 取得了极大的成功: SQL 可以将“获取数据的声明方式”和“引擎如何获取数据”相分开, 所以 Neo4j 的工程师们希望开发一种声明式的图查询语言。

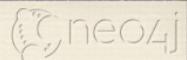
2010 年, Andrés Taylor 作为工程师加入 Neo4j。受 SQL 启发, 他启动了一个项目来开发图查询语言, 而这种新语言于 2011 年 Neo4j 1.4 发布, 这种新语言就是如今大多数图查询语言的先祖——Cypher。

Cypher 的语法基础, 是用 "ASCII艺术(ASCII art)" 来描述图模式。这种方式最初来源于 Neo4j 工程师团队在源代码中评注如何描述图模式。可以看下图的例子:

The Origin of Cypher



The Origin of Cypher



```

MATCH (query)-[:MODELED_AS]->(drawing),
      (code)-[:IMPLEMENTES]->(query),
      (drawing)-[:TRANSLATED_TO]->(ascii_art)
      (ascii_art)-[:IN_COMMENT_OF]->(code)
WHERE query.id = {query_id}
RETURN code.source
  
```

ASCII art 简单说，就是如何用可打印文本来描述点和边。Cypher 文本用 () 表示点，-[]-> 表示边。 (query)-[modeled as]->(drawing) 来表示起点 query， 终点 drawing， 边 modeled as， 这样一个最简单的图关系(也可以称为图模式)。

Cypher 第一个版本实现了对图的读取，但是需要用户说明从哪些节点开始查询。只有从这些节点开始，才可以支持图的模式匹配。

在后面的版本，2012 年 10 月发布的 Neo4j 1.8 中，Cypher 增加了修改图的能力。但查询还是需要指明从哪些节点开始。

2013 年 12 月，Neo4j 2.0 引入了 label 的概念，label 本质上是个索引。这样，查询引擎就可以利用索引，来选择模式所匹配到的节点，而不需要用户指定开始查询的节点。

随着 Neo4j 的普及, Cypher 有着广泛的开发者群体, 在各行各业的得到广泛的使用。至今仍是最受欢迎的图查询语言。

2015 年 9 月, Neo4j 发起成立了 openCypher Implementors Group (oCIG), 将 Cypher 开放为 openCypher, 通过开源的方式来治理和推进语言自身的演化。

后续

Cypher 启发了一系列后续的图查询语言, 包括

2015 年, Oracle 发布图引擎 PGX 使用的图语言 PGQL。

2016 年, Linked Data Benchmarking Council, LDBC 是一个行业知名的图性能基准评测机构。LDBC 发布 G-CORE

2018 年, 基于 Redis 的图库(library) RedisGraph 采用 Cypher 作为其图语言

2019 年, 国际标准组织 ISO 启动两个项目, 基于 openCypher, PGQL, GSQ₁₂, and G-CORE 等现有业界成果, 启动图语言国际标准的制定过程

2019 年, NebulaGraph 以 openCypher 为基础发布其扩展的图语言 NebulaGraph Query Language, nGQL。



分布式图数据库

大约 2005-2010 年, 随着 Google 云计算“三驾马车”的发布, 各种分布式的架构开始越来越流行, 其中就包括以开源方式运作的 Hadoop 和 Cassandra 等。这里包括几个方面的影响:

1. 由于数据量和计算量越来越大, 相比于单机(例如 Neo4j)或者小型机这种方案, 分布式系统的技术和成本优势更加明显; 而同时, 分布式系统使得应用程序在访问这成千上万台机器时, 就如同访问本地的系统一样, 不需要在代码层面进行过多改造;
2. 开源方式使得更多的人(包括代码开发者、数据科学家、产品经理等)以更加低成本和有效的方式参与新兴的技术, 并反馈给社区。

严格说, Neo4j 也提供了不少的分布式的功能, 但都和业界意义上的(对等、分片的)分布式系统有较大的不同:

- Neo4j 3.X 要求全量数据必须存放在单机中。虽然其也提供多机之间(Master-slave/slave)做全量复制和高可用, 但数据不可切分为不同子图存放。

Neo4j Causal Cluster



Cluster architecture

- Neo4j 4.X 允许在不同机器上各存放一部分数据（子图），然后在应用层需通过一定方式拼装后(其称为编织 Fabric)¹³，将读写分发到各个机器上。这种做法需要应用层代码有大量的参与和工作。例如，设计如何把不同子图应该放置在哪些机器上，如何将从各机器获取的部分结果重新编织为最终的结果。



其语法风格大体是

```
USE graphA # S1.1 从 Shard A 读
MATCH (movie:Movie)
Return movie.title AS title
UNION # S2. 在代理服务器 Join 结果
USE graphB # S1.2 从 Shard B 读
```

```
MATCH (movie:Movie)
RETURN movie.title AS title
```



第二代（分布式）图数据库：TITAN 和其后继者 JANUSGRAPH

2011 年，Aurelius 公司成立，致力于开发一个开源的分布式图数据库 Titan¹⁴。到 2015 年 Titan 的第一个正式版发布，Titan 后端可以支持多种主流的分布式存储架构（例如 Cassandra, HBase, Elasticsearch, BerkeleyDB），并可以复用 Hadoop 生态的诸多便利，前端以 Gremlin 为统一的查询语言。对于程序员使用、开发和社区参与都很方便。大规模的图，可以分片后存放在 HBase 或者 Cassandra 上（这些当时都已经是相对成熟的分布式存储方案），Gremlin 语言虽然略微冗长但相对功能完备。整个方案在当时(2011-2015)体现了不错的竞争力。

下图显示了 2012 年 - 2015 年，Titan 和 Neo4j 在 GitHub.com 上 star 的增长情况。



2015 年 Aurelius(Titan) 被 DataStax 收购，这之后 Titan 逐渐转变为一个闭源的商业产品 (DataStax Enterprise Graph)。

在 Aurelius(Titan) 被收购后，市场对于开源分布式的图数据库一直仍有比较强烈的需求，而当时市场上成熟和活跃的产品并不多。大数据时代，数据仍在远快于摩尔定律的速度，源源不断的产生。Linux 基金会以及一些技术巨头(Expero, Google, GRAKN.AI, Hortonworks, IBM and

Amazon) 在2017年，复制并分叉(fork)了原有的Titan项目，并启动为一个新项目 JanusGraph¹⁵。之后大多数的社区工作，包括开发、测试、发布和推广都逐步转移到了新的 JanusGraph。

下图显示了两个项目2012-2021年日常代码提交(pull request)的变化情况，可以观察到几点：

- 即使 Aurelius(Titan) 2015 年被收购后，其开源代码仍有一定的活跃度(左侧)，但增速已经明显放缓。这体现了社区的力量。
- 新项目 JanusGraph 项目在 2017 年 1 月启动后，其社区迅速活跃起来，短短一年时间就超越了 Titan 过去 5 年累计的 pull request 数量。而与此同时，Titan 开源项目就此停滞。



同期知名产品 ORIENTDB, TIGERGRAPH, ARANGODB, 和 DGRAPH

此后更多的厂商加入整个市场，除了由Linux基金会托管的 JanusGraph，还有一些由商业公司主导开发的分布式图数据，各方采用的数据模型和访问方式也有明显的不同。本文不做一一介绍，仅简单列出主要区别。

厂商名	创立时间	核心产品名	开源协议	数据模型	查询语言
OrientDB LTD (2017 年 被 SAP 收购)	2011	OrientDB	开源	文档 + KV + 图	OrientDB SQL (基 于SQL扩展的图能 力)
GraphQL (后改名 TigerGraph)	2012	TigerGraph	商业版本	图(分析)	GraphQL (类 SQL风格)
ArangoDB GmbH	2014	ArangoDB	Apache License 2.0	文档 + KV + 图	AQL (同时操作文 档, KV 和图)
DGraph Labs	2016	DGraph	Apache Public License 2.0 + Dgraph Community License	原 RDF, 后 改为 GraphQL	GraphQL+-

传统巨头微软、亚马逊和甲骨文纷纷入场

除了聚焦于图产品的厂商外，传统巨头也纷纷进入这个领域。

Microsoft Azure Cosmos DB¹⁶ 是一个在微软云上的多模数据库云服务，可以提供SQL、文档、图、key-value等多种能力； Amazon AWS Neptune¹⁷ 是一种由 AWS 提供图数据库云服务，可以提供属性图和 RDF 两种数据模型； Oracle graph¹⁸ 是关系型数据库巨头 Oracle 在图技术与图数据库方向的产品。

新一代开源分布式图数据库**NEBULAGRAPH**

在下一章，我们将正式介绍新一代开源分布式图数据库NebulaGraph。

1. https://db-engines.com/en/ranking_categories ↪
2. <https://www.yellowfinbi.com/blog/2014/06/yfcommunitynews-big-data-analytics-the-need-for-pragmatism-tangible-benefits-and-real-world-case-165305> ↪
3. <https://www.gartner.com/smarterwithgartner/gartner-top-10-data-and-analytics-trends-for-2021/> ↪
4. <https://www.verifiedmarketresearch.com/product/graph-database-market/> ↪
5. <https://www.globenewswire.com/news-release/2021/01/28/2165742/0/en/Global-Graph-Database-Market-Size-Share-to-Exceed-USD-4-500-Million-By-2026-Facts-Factors.html> ↪
6. <https://www.marketsandmarkets.com/Market-Reports/graph-database-market-126230231.html> ↪
7. <https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the> ↪
8. <https://www.amazon.com/Designing-Data-Intensive-Applications-Reliable-Maintainable/dp/1449373321> ↪
9. I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A Graphical Query Language Supporting Recursion. In Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data, pages 323–330. ACM Press, May 1987. ↪ ↪
10. "An overview of the recent history of Graph Query Languages". Authors: Tobias Lindaaker, U.S. National Expert. Date: 2018-05-14 ↪
11. Gremlin是基于Apache TinkerPop开发的图语言(<https://tinkerpop.apache.org/>)。 ↪
12. <https://docs.tigergraph.com/dev/gsql-ref> ↪
13. <https://neo4j.com/fosdem20/> ↪
14. <https://github.com/thinkaurelius/titan> ↪
15. <https://github.com/JanusGraph/janusgraph> ↪
16. <https://azure.microsoft.com/en-us/free/cosmos-db/> ↪
17. <https://aws.amazon.com/cn/neptune/> ↪
18. <https://www.oracle.com/database/graph/> ↪

最后更新: May 4, 2023

2.3 相关技术

本节主要介绍两个和分布式图数据库关系密切的领域，数据库方面和图技术方面。

2.3.1 数据库方面

关系型数据库

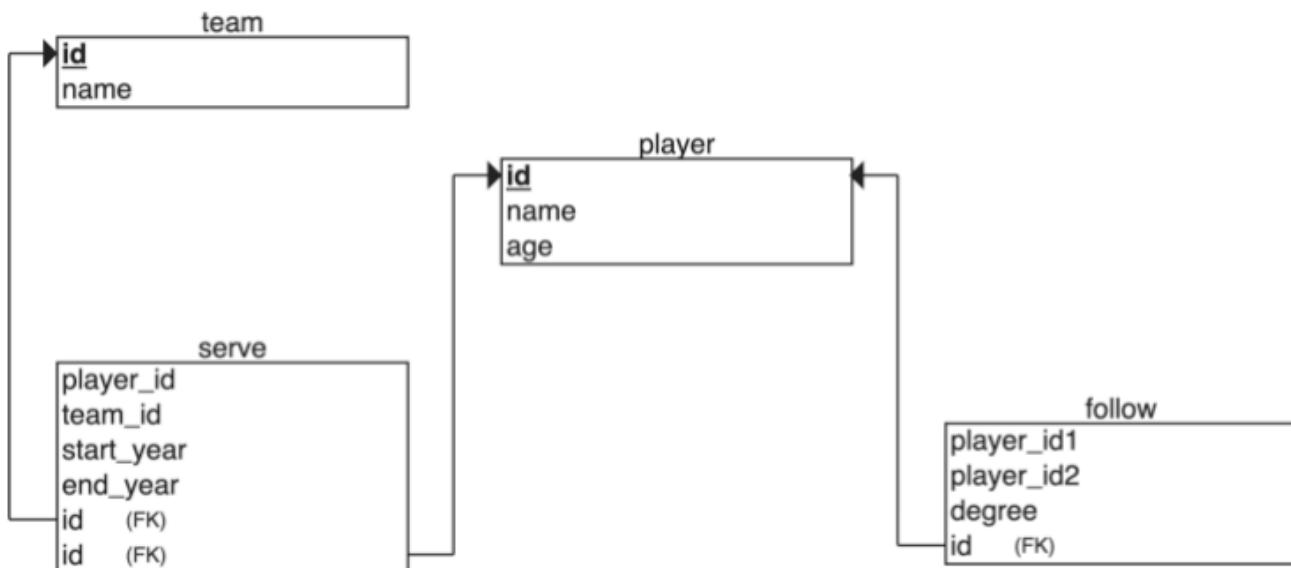
关系型数据库，是指采用了关系模型来组织数据的数据库。关系模型为二维表格模型，一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。说到关系型数据库，大多数人都会想到 MySQL。MySQL 是目前最流行的数据库管理系统之一，支持使用最常见的结构化查询语言（SQL）进行数据库操作，并以表格、行、列的形式存储数据。这种存储数据的方法源自埃德加·科德（Edgar Frank Codd）于 1970 年提出的关系型数据模型。

在关系型数据库中，可以为待存储的每种类型的数据创建一个表。例如，球员表用来存储所有的球员信息，球队表用来存储球队信息等。SQL 表中的每行数据都必须包含一个主键（primary key）。主键是该行数据的唯一标识符。一般地，主键作为字段 ID 都是随行数自增的。关系型数据库自问世以来一直为计算机行业提供着非常好的服务，并将未来很长的时间内继续服务下去。

如果你用过 Excel、WPS 或其他类似的应用，你就会大概了解到关系数据库是如何工作的。首先设置好列，然后在对应的列下添加行数据。你可以对某一列数据进行求平均值或其他聚合操作，这与在关系型数据库 MySQL 中求平均值的操作类似。而 Excel 中的数据透视表则相当于在关系型数据库 MySQL 中使用聚合函数和 CASE 语句对数据进行查询。一个 Excel 文件可以有多张表，一张表就相当于 MySQL 的一张表。一个 Excel 文件则类似于一个 MySQL 数据库。

关系型数据库中的关系

与图数据库不同，关系型数据库（或 SQL 型的数据库）中的边也是作为实体存储在专门的边表中的。先创建两个表，球员（player）和球队（team），然后再创建表 player_team 作为边表。边表通常由相关的表 join 而成。例如，此处的边表 player_team 就由球员表和球队表 join 而成。



这种存储边的方式在关联小型数据集时问题并不大，但是当关系型数据库中的关系太多时，问题就出现了。事实上，关系型数据库是非常“反关系的”。具体来说，当你只想查询一个球员的队友时，你必须对表中的所有数据进行 join 操作，然后再过滤掉你不需要的所有数据，当你的数据集达到一定规模时，这将给关系型数据库带来巨大压力。如果你想关联多张不同的表，可能在 join 爆炸（join bombs）前系统就已经无法响应了。

关系型数据库起源

上文提到，关系型数据模型最早是由 IBM 的工程师埃德加·科德（Edgar Frank Codd）于 1970 年提出的。科德写了几篇数据库管理系统的论文，论述了关系型数据模型的潜力。关系型数据模型不依赖于数据链接列表（网状数据或层级数据），而是更多依赖于数据集。他使用元组演算（tuple calculus）的数学方法论证了这些数据集能够完成与导航数据库管理系统相同的任务。唯一的要求是，关系型数据模型需要一种合适的查询语

言，以保证数据库的一致性要求。这就为后来声明型的结构化查询语言（SQL）提供了灵感来源。IBM 的 R 系统是关系型数据模型的最早使用者之一。然而，由前 IBM 员工拉里·埃里森创办的名叫软件开发实验室的小公司在市场上击败了 IBM。该公司的产品就是后来为我们熟知的 Oracle。

由于“关系数据库”在当时是一个比较时髦的词汇，因此许多数据库供应商都喜欢在其产品名称中使用这个词汇，尽管他们的产品实际上并不是关系型的。为了防止这种情况并减少关系型数据模型的错误使用，科德提出了著名科德 12 定律（Codd's 12 rules）。所有关系型数据系统都必须遵循科德 12 定律。

NoSQL 数据库

图数据库并不是可以克服关系型数据库缺点的唯一替代方案。现在市面上还有很多非关系型数据库的产品，这些产品都可以叫做 NoSQL。NoSQL 一词最早于上世纪 90 年代末提出，可以解释为“非 SQL”或“不仅是 SQL”，具体解释要根据语境判断。为便于理解，这里 NoSQL 可以解释成“非关系型数据库”。不同于关系型数据库，NoSQL 数据库提供的数据存储、检索机制并不是基于表关系建模的。NoSQL 数据库可以分为四类：

- 键值存储（key-value stores）
- 列式存储（column-family stores）
- 文档存储（document stores）
- 图数据库（graph databases）

下面将分别介绍这四类数据库。

键值存储

键值存储，顾名思义，就是使用键值对存储数据的数据库。不同于关系型数据库，键值存储是没有表和列的。如果一定要做类比，键值数据库本身就像一张有很多列（也就是键）的大表。在键值存储数据库中，数据（即键值对中的值）都是通过键来存储和查询的，通常用哈希列表来实现。这比传统的 SQL 数据库要简单得多，而且对于某些 web 应用来说，这就足够了。

键值模型对于 IT 系统来说优势在于简单、易部署。多数情况下，这种存储方式对非关联的数据很适用。如是只是存储数据而无需查询的话，使用这种存储方法就没有问题。但是如果 DBA 只对部分值进行查询或更新的时候，键值模型就显得效率低下了。常见的键值存储数据库有：Redis、Voldemort、Oracle BDB。

列式存储

NoSQL 数据库的列式存储与 NoSQL 数据库的键值存储有许多相似之处，因为列式存储仍然在使用键进行存储和检索。区别在于列式存储数据库中，列是最小的存储单元，每一列均由键、值以及用于版本控制和冲突解决的时间戳组成。这在分布式扩展时特别有用，因为在数据库更新时，可以使用时间戳定位过期数据。由于列式存储良好的扩展性，因此适用于非常大的数据集。常见的列式存储数据库有：HBase、Cassandra、HadoopDB 等。

文档存储

准确来说，NoSQL 数据库文档存储实际上也是基于键值的数据库，只不过对功能做了增强。数据仍然以键值的形式存储，但是文档存储中的值是结构化的文档，而不仅仅是一个字符串或单个值。也就是说，由于信息结构的增加，文档存储能够执行更优化的查询，并且使数据检索更加容易。因此，文档存储特别适合存储、索引并管理面向文档的数据或者类似的半结构化数据。

从技术上讲，作为一个半结构化的信息单元，文档存储中的文档可以是以任何形式可用的文档，包括 XML、JSON、YAML 等，这取决于数据库供应商的设计。比如，JSON 就是一种常见的选择。虽然 JSON 不是结构化数据的最佳选择，但是 JSON 型的数据在前端和后端应用中都可以使用。常见的文档存储数据库有：MongoDB、CouchDB、Terrastore 等。

图存储

最后一类 NoSQL 数据库是图数据库。本书重点讨论的NebulaGraph也是一种图数据库。虽然同为 NoSQL 型数据库，但是图数据库与上述 NoSQL 数据库有本质上的差异。图数据库以点、边、属性的形式存储数据。其优点在于灵活性高，支持复杂的图形算法，可用于构建复杂的关系图谱。我们将在随后的章节中详细讨论图数据库。不过在本章中，你只要知道图数据库是一种 NoSQL 类型的数据库就可以了。常见的图数据库有：NebulaGraph、Neo4j、OrientDB 等。

2.3.2 图技术方面

来看一张 2020 年的图技术全景¹



和图有关联的技术有很多，可以大致分为这么几类：

- 基础设施：包括图数据库、图计算(处理)引擎、图深度学习、云服务等。
 - 应用：包括可视化、知识图谱、反诈骗、网络安全、社交网络等。
 - 开发工具：包括图查询语言、建模工具、开发框架和库。
 - 电子书籍²和会议等。

图语言

在上一节中，我们介绍了图语言的历史。在这一节中，我们将对图语言的功能做一个分类。

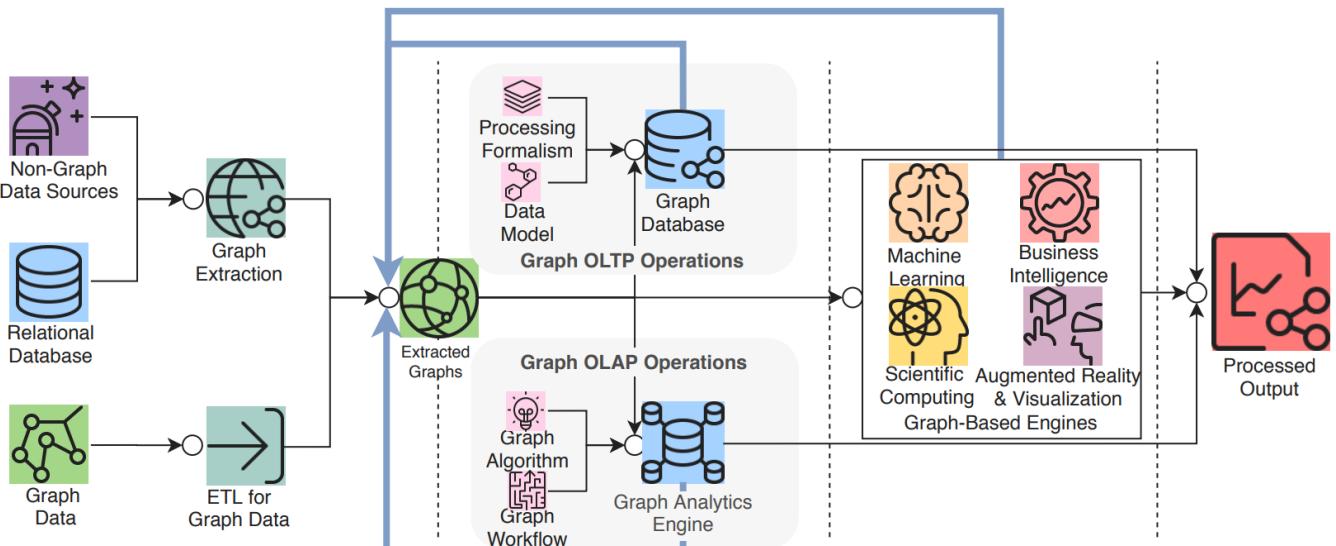
- 近邻查询: 查询给定点或者边的邻边、邻点, 或者是 K 跳的近邻。
 - 图模式匹配(Pattern matching): 找到一个/所有的子图, 满足给定的图模式; 这个问题非常接近于"子图同构映射(subgraph isomorphism)"——虽然两个看上去不同的图, 但其实是一模一样的³。

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

- 可达性（连通性）问题：最常见的可达性问题就是最短路径问题。泛化一些，这类问题通常用正则路径(Regular Path Query)的方式来描述——一系列联通的点组构成了一条路径，而该路径需要满足某种正则表达式。
- 分析型问题：通常与一些汇聚型算子相关（平均值、count、最大值、点的出入度），或者度量所有两两点之间的距离、某节点与其他节点之间的互动程度（介数中心性）等。

图数据库(Graph Database)与图处理(Graph processing)系统

一个图系统通常会涉及到复杂的数据流水线⁴，从数据源(左边)到处理输出(右边)，会经过多个数据加工处理环节和系统；大的模块可以分为 ETL模块，图数据库系统(Graph OLTP)，图处理系统(Graph OLAP)，基于图引擎的应用系统（BI、知识图谱等）。



虽然这两类系统都是与图数据和图技术相关的系统，也处理类似的目标，但是他们有着不同的起源和特长（及弱点）：

- （在线）图数据库目标是图的持久化存储管理、高效的子图操作。硬盘（及网络）是目标运行设备，物理/逻辑数据映射，数据完整性和（故障）一致性是主要目标。每一个请求通常只会涉及到全图的一小部分，通常可以在一台服务器上完成；单个请求时延通常在毫秒到秒级别，请求并发量通常在几千到几十万。早期的 Neo4j 是图数据库领域的起源之一。
- （离线）图处理系统目标是全图的大批量、并行、迭代、处理与分析，内存（及网络）是目标运行设备。每一个请求会涉及到所有的图节点，需要所有的服务器参与完成；单个请求的时延通常在分钟到小时（天），请求并发量通常为个位数。Google 的 Pregel⁵ 是图处理系统的典型起源代表，它的点中心编程抽象与BSP的运行模式构成的编程范式，相比之前 Hadoop Map-Reduce 是更为图友好的 API 抽象。



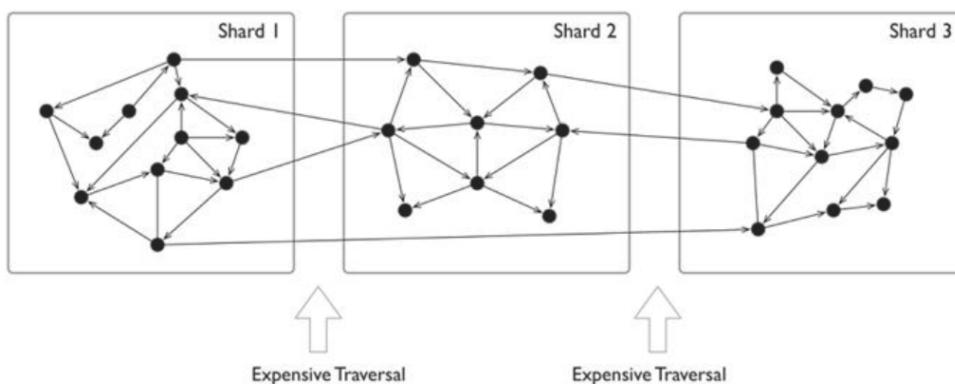
6

图的分片方式

对于一个大规模的图数据来说，是很难存放在单个服务器的内存中的，即使仅仅存放图结构本身也不够。而且通过增加单服务器的能力，其成本价格通常成指数级别上升。此外，随着数据量的增加，例如到达千亿级别的时候，已经超过了市面上所有商用服务器的容量能力。

与此对应的，另外一个经常使用的方案，是对数据进行分片，并将每个分片放置在不同的服务器上（并进行冗余备份），以此来增加可靠性和性能。对于一些 NoSQL 型的系统，例如 key-value 或者文档型的系统来说，这个分片方式是比较直观和自然的；通常可以根据 key 或者 docID，来将每个记录或者数据单元(key-value, doc)放在不同的服务器上。

但是图这种数据结构的分片通常不那么直观，这是因为通常图是“全联通”的，每个点通常只要6跳就可以联通到其他任何节点；而理论上早已证明图的划分问题是 NP 的。与此同时，当把整个图数据分散到多个服务器时，跨服务器的网络访问时延10倍于同一个服务器内部的硬件(内存)访问时间；因此对于一些深度优先遍历的场景，会发生大量的跨网络访问，导致整体时延极高。



7

另一方面，通常图有着明显的幂律分布；少量节点的邻边稠密程度远大于平均的节点，虽然处理这些节点通常可以在同一台服务器内，减少了跨网络访问，但这也意味着这些服务器压力会远大于平均。



因此，常见的图分片(Sharding)方式有几类：

- 偏应用层面的分片：应用层感知并控制每个点和边应该落在哪个分片上，一般来说可以根据点和边的类型（比如业务意义来人为）判断。将一组相同的点放在一个分片，另一组相同类型的点放在另一个分片。当然，为了高可靠，分片本身还可以做多副本。在应用使用时，从各个分片取回所要的点和边，然后在偏应用侧（或者某个代理服务器端），将取回的数据拼装成最终的结果。其典型代表是 Neo4j 4.x 的 Fabric。



- 使用分布式的缓存层：在硬盘之上增加内存缓存层，并对重要的部分分片和数据（例如图结构）进行缓存，并预热这部分缓存。
- 增加（只读）副本或视图（View）：为部分图分片增加只读的副本或者建立一个视图，将较重的读请求负载通过这些分片服务器承担。
- 进行细颗粒度的图划分：例如将点和边组成多个小分片（Partition），而不是一台服务器一个大分片（Sharding），再将关联性较强的 Partition 尽量放置在同一个服务器上⁸。



具体工程实践时，也会混合使用上述几种方式。通常，离线的图处理系统会通过一个ETL过程，将图进行一定程度的预处理以提高局部性；而在线图数据库系统通常会选择周期性的数据再平衡过程来提高数据局部性。

一些技术上的挑战

在文献⁹中，对于无处不在的大图和挑战做了详尽的调研，下面是其列出的十大图技术挑战：

- 可扩展性(软件可以处理更大的图规模): 包括大图的加载、更新、图计算和图遍历，触发器，超级节点；
- 可视化：可定制布局，大图的渲染，多层次展示，动态（更新）的展示
- 查询语言和编程 API：包括语言表达能力、标准兼容性、与现有系统的兼容性；子查询的设计和跨多图之间的关联查询
- 更快的图（及机器学习）算法
- 易用性（配置和使用）
- 性能指标与测试
- 更通用的图技术软件（例如，处理离线、在线、流式的计算）
- 图清洗（ETL）
- Debug 调试与测试

一些开源的单机图工具

对于图数据库通常会有一个误解，只要涉及到图结构的数据存取就需要存放在图数据库中，这是一种很大的浪费。

这就像也许你只需要一个 SQLite，却用了一个 Oracle。

当数据量并不大时，通常单机内存可以放下，例如数据量几千万的点边关系，使用一些单机的开源工具也可以取得很好的效果。

Note

下面是一些推荐的单机图库，也可以集成在你的应用程序里面。

- JGraphT¹⁰: 一个知名的开源 Java 图论库(library)，其实现了相当多的高效图算法。
- JUNG¹¹是BSD许可下用Java编写的开源图建模和可视化框架。该框架内置了许多布局算法，以及诸如图聚类和节点中心性度量之类的分析算法。
- igraph¹²: 一个轻量且功能强大的 Library，支持R、python、C
- NetworkX¹³: 数据科学家做图论分析第一选择， python。
- Cytoscape¹⁴: 功能强大的可视化开源图分析工具。
- Gephi¹⁵: 功能强大的可视化开源图分析工具。
- arrows.app¹⁶: 非常简单的脑图工具，用于可视化生成 Cypher 语句。

一些行业数据集和 Benchmark

LDBC

关联数据基准委员会（LDBC¹⁷，Linked Data Benchmark Council）是由Oracle、Intel等软硬件巨头和主流图数据库厂商Neo4j和TigerGraph等组成的非赢利机构，是图的基准指南制定者与测试结果发布机构，在行业内有着很高的影响力。

社交网络基准测试（SNB，Social Network Benchmark）是由关联数据基准委员会（LDBC）开发的面向图数据库的基准测试（Benchmark）之一，分为交互式查询（Interactive）和商业智能（BI）两个场景。其作用类似于 TPC-C, TPC-H 等测试在 SQL 型数据库中的功能，可以帮助用户比较多种图数据库产品的功能、性能、容量。

SNB 数据集模拟一个社交网络的人、发帖之间的关系，考虑了社交网络的分布属性、人的活跃度等等社交信息。



其标准数据量从 0.1 GB (scale factor 0.1) 到 1000 GB (sf1000)，也可以生成 10 TB, 100 TB 等更大的数据集；其点、边数量如下表。

Scale Factor	0.1	0.3	1	3	10	30	100	300	1000
# of Persons	1.5K	3.5K	11K	27K	73K	182K	499K	1.25M	3.6M
# of nodes	327.6K	908K	3.2M	9.3M	30M	88.8M	282.6M	817.3M	2.7B
# of edges	1.5M	4.6M	17.3M	52.7M	176.6M	540.9M	1.8B	5.3B	17B

2.3.3 一些趋势

虽然图的各种技术起源和目标并不相同，但在相互借鉴和融合

Convergence of Capabilities in the Graph DBMS Landscape



上云的趋势在加速，对于弹性能力提出更高要求

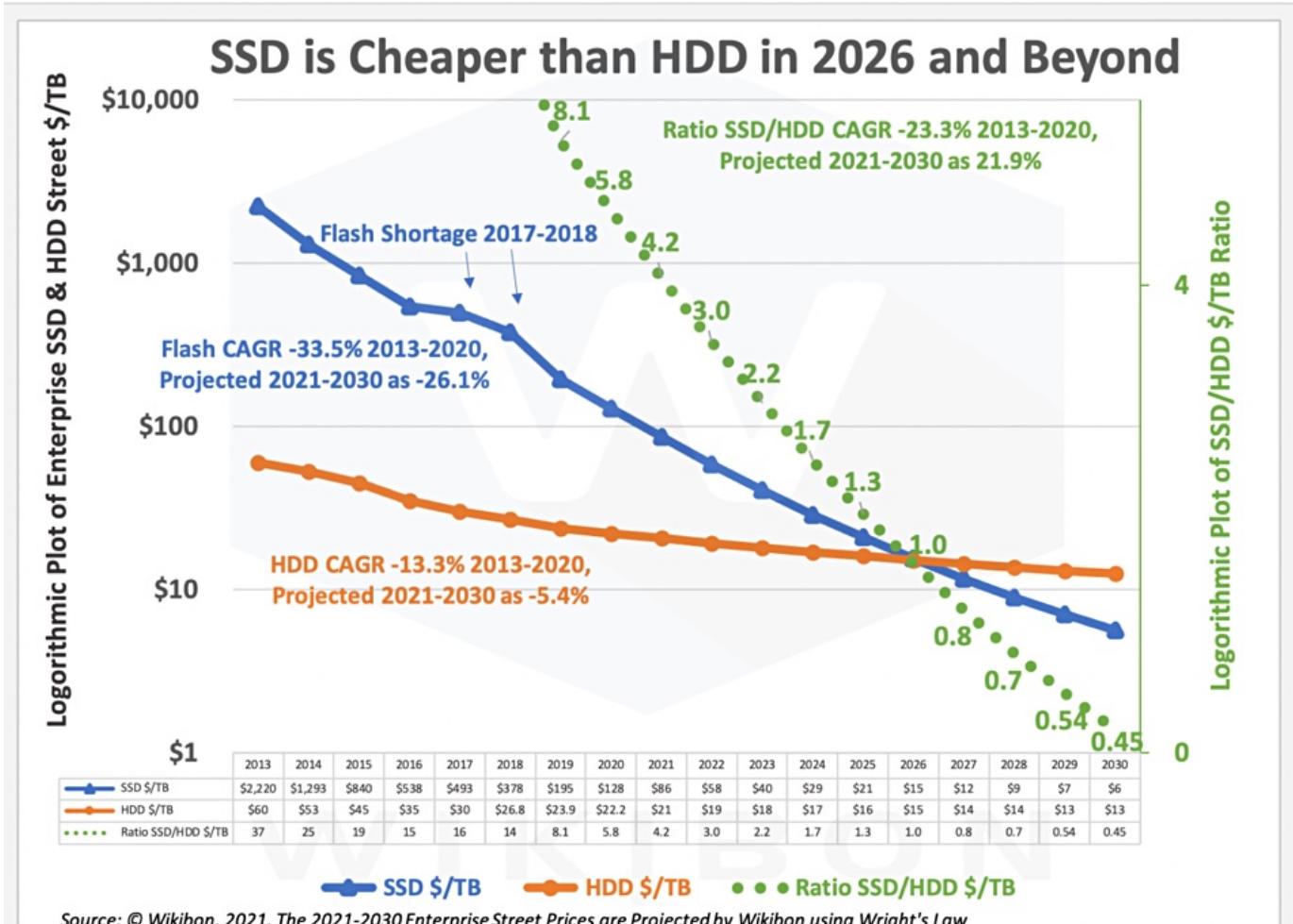
根据 Gartner 的预计，云服务一直保持较快的增速和渗透率¹⁸。大量的商业软件，正在从 10 年前完全私有本地逐步转向基于云服务的商业模式。云服务的一大优点是其提供了近乎无限的弹性能力；这也要求各种基于云基础设施的软件必须有更好的快速弹性扩缩容能力。



硬件趋势，SSD 将成为主流的持久化设备

硬件决定了软件的架构——从发现摩尔定律的 50 年代到进入多核的 00 年代，硬件发展趋势和速度一直深刻的决定了软件的架构。数据库类系统大多围绕“硬盘 + 内存”设计，高性能计算型系统大多围绕“内存 + CPU”设计，分布式系统面对千兆、万兆和 RDMA 网卡的设计也完全不同。

图基于拓扑的遍历有着极其明显的随机访问特点，因此大多数早期图数据库系统都采用了“大内存 + HDD”的架构——通过设计常驻在内存中的一些数据结构（例如B+树、Hash表等），在内存中实现随机访问目的，以优化图的拓扑遍历，再将这些随机访问转换成 HDD 所适合的顺序读写。整套软件的架构（包括存储和计算层）都必须基于和围绕这样的 IO 流程来展开。随着 SSD 价格的快速下降¹⁹，SSD 正在替代 HDD 成为持久化设备的主流。SSD 随机访问友好、IO 队列深、按块存取的特点与 HDD 高度顺序、随机时延极高、磁道易损坏的访问特点有着明显的不同。全部的软件架构也需要重新设计，这成为沉重的历史技术负担。

**Figure 4 - SSD/HDD Pricing Ratio 2013 - 2030**

Source: © Wikibon, 2021.

-
1. <https://graphaware.com/graphaware/2020/02/17/graph-technology-landscape-2020.html> ↵
 2. 学习目的(非商业用途)可以联系[作者]((mailto:min.wu@vesoft.com))获取电子版。 ↵
 3. https://en.wikipedia.org/wiki/Graph_isomorphism ↵
 4. The Future is Big Graphs! A Community View on Graph Processing Systems. <https://arxiv.org/abs/2012.06171> ↵
 5. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In Proceedings of the International Conference on Management of data (SIGMOD), pages 135–146, New York, NY, USA, 2010. ACM ↵
 6. <https://neo4j.com/graphacademy/training-iga-40/02-iga-40-overview-of-graph-algorithms/> ↵
 7. <https://livebook.manning.com/book/graph-powered-machine-learning/welcome/v-8/> ↵
 8. <https://www.arangodb.com/learn/graphs/using-smartgraphs-arangodb/> ↵
 9. <https://arxiv.org/abs/1709.03188> ↵
 10. <https://jgrapht.org/> ↵
 11. <https://github.com/jrtom/jung> ↵
 12. <https://igraph.org/> ↵
 13. <https://networkx.org/> ↵
 14. <https://cytoscape.org/> ↵
 15. <https://gephi.org/> ↵
 16. <https://arrows.app/> ↵
 17. https://github.com/ldbc/ldbc_snb_docs ↵
 18. <https://cloudcomputing-news.net/news/2019/apr/15/public-cloud-soaring-to-331b-by-2022-according-to-gartner/> ↵
 19. <https://blocksandfiles.com/2021/01/25/wikibon-ssds-vs-hard-drives-wrights-law/> ↵
-

最后更新: May 4, 2023

2.4 Macro Rendering Error

UndefinedError: 'comm' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 19, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'comm' is undefined
```

最后更新: May 4, 2023

2.5 数据模型

本文介绍NebulaGraph的数据模型。数据模型是一种组织数据并说明它们如何相互关联的模型。

2.5.1 数据模型

NebulaGraph数据模型使用 6 种基本的数据模型：

- 图空间（Space）

图空间用于隔离不同团队或者项目的数据。不同图空间的数据是相互隔离的，可以指定不同的存储副本数、权限、分片等。

- 点（Vertex）

点用来保存实体对象，特点如下：

- 点是用点标识符（VID）标识的。VID 在同一图空间中唯一。VID 是一个 int64，或者 fixed_string(N)。
- 点可以有 0 到多个 Tag。

Compatibility

NebulaGraph 2.x 及以下版本中的点必须包含至少一个 Tag。

- 边（Edge）

边是用来连接点的，表示两个点之间的关系或行为，特点如下：

- 两点之间可以有多条边。
- 边是有方向的，不存在无向边。
- 四元组 <起点 VID、Edge type、边排序值 (rank)、终点 VID> 用于唯一标识一条边。边没有 EID。
- 一条边有且仅有一个 Edge type。
- 一条边有且仅有一个 Rank，类型为 int64，默认值为 0。

关于

Rank 可以用来区分 Edge type、起始点、目的点都相同的边。该值完全由用户自己指定。

读取时必须自行取得全部的 Rank 值后排序过滤和拼接。

不支持诸如 next()，pre()，head()，tail()，max()，min()，lessThan()，moreThan() 等函数功能，也不能通过创建索引加速访问或者条件过滤。

- 标签（Tag）

Tag 由一组事先预定义的属性构成。

- 边类型（Edge type）

Edge type 由一组事先预定义的属性构成。

- 属性（Property）

属性是指以键值对（Key-value pair）形式表示的信息。

Note

Tag 和 Edge type 的作用，类似于关系型数据库中“点表”和“边表”的表结构。

2.5.2 有向属性图

NebulaGraph 使用有向属性图模型，指点和边构成的图，这些边是有方向的，点和边都可以有属性。

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。

Note

NebulaGraph 中没有无向边，只支持有向边。

Incompatibility

由于 NebulaGraph 3.4.1 的数据模型中，允许存在“悬挂边”，因此在增删时，用户需自行保证“一条边所对应的起点和终点”的存在性。详见 [INSERT VERTEX](#)、[DELETE VERTEX](#)、[INSERT EDGE](#)、[DELETE EDGE](#)。

不支持 openCypher 中的 MERGE 语句。

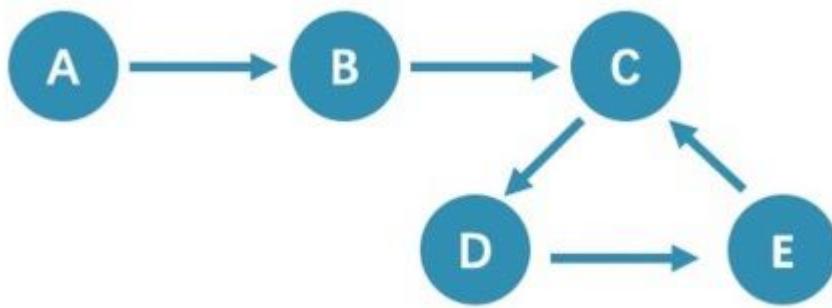
最后更新: May 4, 2023

2.6 路径

图论中一个非常重要的概念是路径，路径是指一个有限或无限的边序列，这些边连接着一系列的点。

路径的类型分为三种：`walk`、`trail`、`path`。关于路径的详细说明，请参见[维基百科](#)。

本文以下图为为例进行简单介绍。



2.6.1 walk

`walk`类型的路径由有限或无限的边序列构成。遍历时点和边可以重复。

查看示例图，由于C、D、E构成了一个环，因此该图包含无限个路径，例如`A->B->C->D->E`、`A->B->C->D->E->C`、`A->B->C->D->E->C->D`。

Note

GO语句采用的是`walk`类型路径。

2.6.2 trail

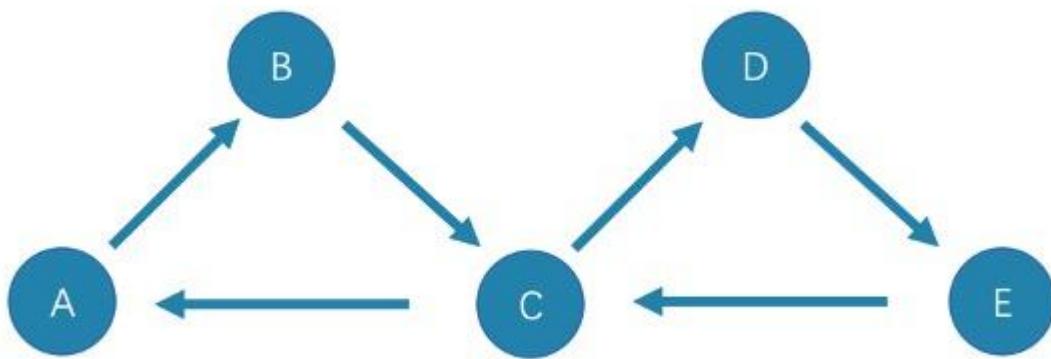
`trail`类型的路径由有限的边序列构成。遍历时只有点可以重复，边不可以重复。柯尼斯堡七桥问题的路径类型就是`trail`。

查看示例图，由于边不可以重复，所以该图包含有限个路径，最长路径由5条边组成：`A->B->C->D->E->C`。

Note

`MATCH`、`FIND PATH`和`GET SUBGRAPH`语句采用的是`trail`类型路径。

在`trail`类型中，还有`cycle`和`circuit`两种特殊的路径类型，以下图为为例对这两种特殊的路径类型进行介绍。



- **cycle**

cycle 是封闭的 **trail** 类型的路径，遍历时边不可以重复，起点和终点重复，并且没有其他点重复。在此示例图中，最长路径由三条边组成：
 $A \rightarrow B \rightarrow C \rightarrow A$ 或 $C \rightarrow D \rightarrow E \rightarrow C$ 。

- **circuit**

circuit 也是封闭的 **trail** 类型的路径，遍历时边不可以重复，除起点和终点重复外，可能存在其他点重复。在此示例图中，最长路径为：
 $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow C \rightarrow A$ 。

2.6.3 path

path 类型的路径由有限的边序列构成。遍历时点和边都不可以重复。

查看示例图，由于点和边都不可以重复，所以该图包含有限个路径，最长路径由 4 条边组成： $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ 。

2.6.4 视频

用户也可以观看视频了解路径的相关概念。

[Path \(03 分 09 秒\)](#)

最后更新: May 4, 2023

2.7 点 VID

在一个图空间中，一个点由点的 ID 唯一标识，即 VID 或 Vertex ID。

2.7.1 VID 的特点

- VID 数据类型只可以为定长字符串 `FIXED_STRING(<N>)` 或 `INT64`。一个图空间只能选用其中一种 VID 类型。
- VID 在一个图空间中必须唯一，其作用类似于关系型数据库中的主键（索引+唯一约束）。但不同图空间中的 VID 是完全独立无关的。
- 点 VID 的生成方式必须由用户自行指定，系统不提供自增 ID 或者 UUID。
- VID 相同的点，会被认为是同一个点。例如：
- VID 相当于一个实体的唯一标号，例如一个人的身份证号。Tag 相当于实体所拥有的类型，例如“滴滴司机”和“老板”。不同的 Tag 又相应定义了两组不同的属性，例如“驾照号、驾龄、接单量、接单小号”和“工号、薪水、债务额度、商务电话”。
- 同时操作相同 VID 并且相同 Tag 的两条 `INSERT` 语句（均无 `IF NOT EXISTS` 参数），晚写入的 `INSERT` 会覆盖先写入的。
- 同时操作包含相同 VID 但是两个不同 TAG A 和 TAG B 的两条 `INSERT` 语句，对 TAG A 的操作不会影响 TAG B。
- VID 通常会被（`LSM-tree` 方式）索引并缓存在内存中，因此直接访问 VID 的性能最高。

2.7.2 VID 使用建议

- NebulaGraph 1.x 只支持 VID 类型为 `INT64`，从 2.x 开始支持 `INT64` 和 `FIXED_STRING(<N>)`。在 `CREATE SPACE` 中通过参数 `vid_type` 可以指定 VID 类型。
- 可以使用 `id()` 函数，指定或引用该点的 VID。
- 可以使用 `LOOKUP` 或者 `MATCH` 语句，来通过属性索引查找对应的 VID。
- 性能上，直接通过 VID 找到点的语句性能最高，例如 `DELETE xxx WHERE id(xxx) = "player100"`，或者 `GO FROM "player100"` 等语句。通过属性先查找 VID，再进行图操作的性能会变差，例如 `LOOKUP | GO FROM $-.ids` 等语句，相比前者多了一次内存或硬盘的随机读（`LOOKUP`）以及一次序列化（`|`）。

2.7.3 VID 生成建议

VID 的生成工作完全交给应用端，有一些通用的建议：

- （最优）通过有唯一性的主键或者属性来直接作为 VID；属性访问依赖于 VID；
- 通过有唯一性的属性组合来生成 VID，属性访问依赖于属性索引。
- 通过 `snowflake` 等算法生成 VID，属性访问依赖于属性索引。
- 如果个别记录的主键特别长，但绝大多数记录的主键都很短的情况下，不要将 `FIXED_STRING(<N>)` 的 N 设置成超大，这会浪费大量内存和硬盘，也会降低性能。此时可通过 `BASE64`, `MD5`, `hash` 编码加拼接的方式来生成。
- 如果用 `hash` 方式生成 `int64` VID：在有 10 亿个点的情况下，发生 `hash` 冲突的概率大约是 $1/10$ 。边的数量与碰撞的概率无关。

2.7.4 定义和修改 VID 与其数据类型

VID 的数据类型必须在[创建图空间](#)时定义，且一旦定义无法修改。

VID 必须在[插入点](#)时设置，且一旦设置无法修改。

2.7.5 “查询起始点”(`start vid`) 与全局扫描

绝大多数情况下，NebulaGraph的查询语句（`MATCH`、`GO`、`LOOKUP`）的执行计划，必须要通过一定方式找到查询起始点的 VID（`start vid`）。

定位 `start vid` 只有两种方式:

1. 例如 `GO FROM "player100" OVER` 是在语句中显式的指明 `start vid` 是 "`player100`";
 2. 例如 `LOOKUP ON player WHERE player.name == "Tony Parker"` 或者 `MATCH (v:player {name:"Tony Parker"})`，是通过属性 `player.name` 的索引来定位到 `start vid`；
-

最后更新: May 4, 2023

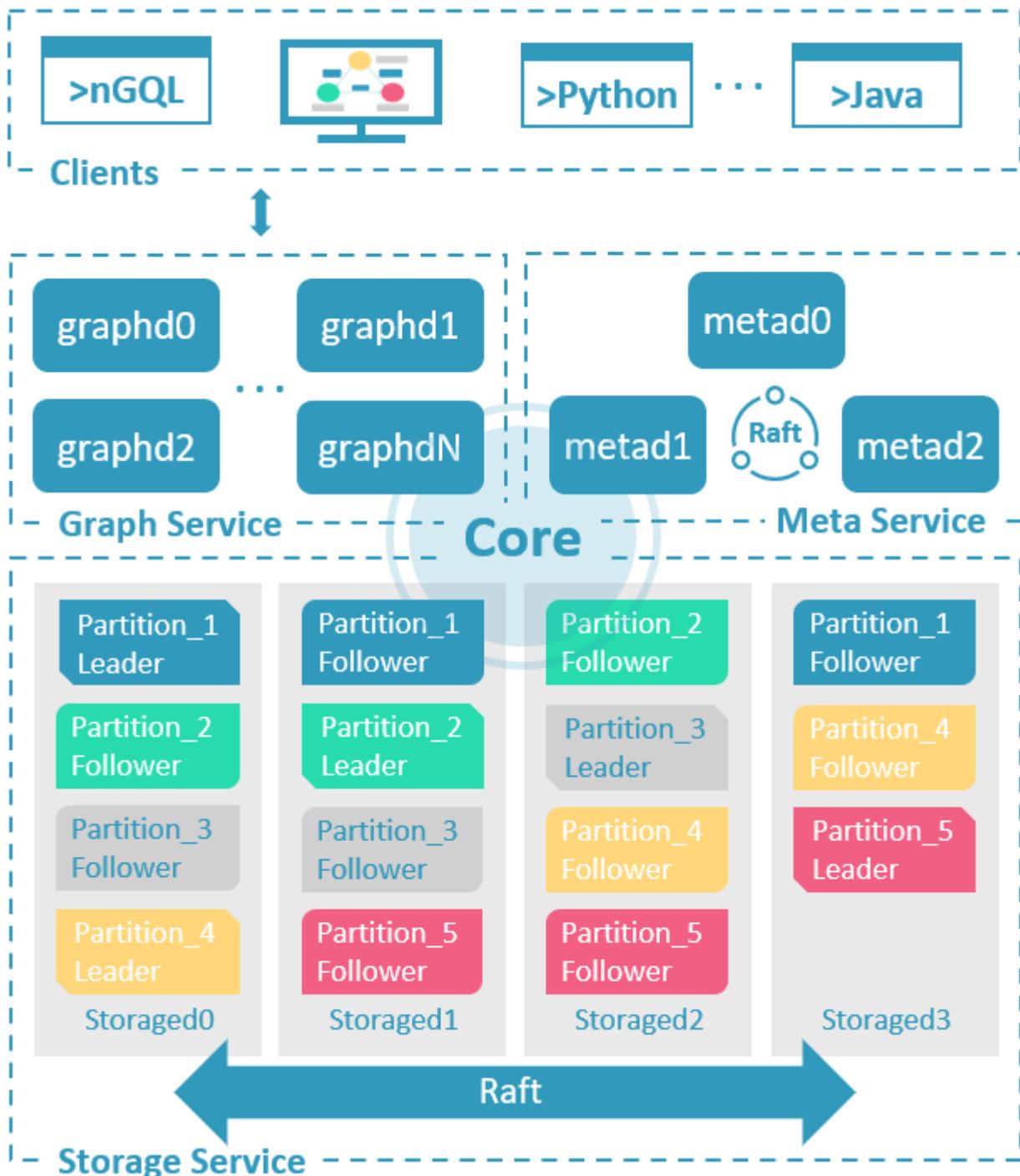
2.8 服务架构

2.8.1 NebulaGraph架构总览

NebulaGraph由三种服务构成：Graph 服务、Meta 服务和 Storage 服务，是一种存储与计算分离的架构。

每个服务都有可执行的二进制文件和对应进程，用户可以使用这些二进制文件在一个或多个计算机上部署NebulaGraph集群。

下图展示了NebulaGraph集群的经典架构。



Meta 服务

在NebulaGraph架构中，Meta 服务是由 `nebula-metad` 进程提供的，负责数据管理，例如 Schema 操作、集群管理和用户权限管理等。

Meta 服务的详细说明，请参见 [Meta 服务](#)。

Graph 服务和 Storage 服务

NebulaGraph采用计算存储分离架构。Graph 服务负责处理计算请求，Storage 服务负责存储数据。它们由不同的进程提供，Graph 服务是由 `nebula-graphd` 进程提供，Storage 服务是由 `nebula-storaged` 进程提供。计算存储分离架构的优势如下：

- 易扩展

分布式架构保证了 Graph 服务和 Storage 服务的灵活性，方便扩容和缩容。

- 高可用

如果提供 Graph 服务的服务器有一部分出现故障，其余服务器可以继续为客户端提供服务，而且 Storage 服务存储的数据不会丢失。服务恢复速度较快，甚至能做到用户无感知。

- 节约成本

计算存储分离架构能够提高资源利用率，而且可根据业务需求灵活控制成本。

- 更多可能性

基于分离架构的特性，Graph 服务将可以在更多类型的存储引擎上单独运行，Storage 服务也可以为多种目的计算引擎提供服务。

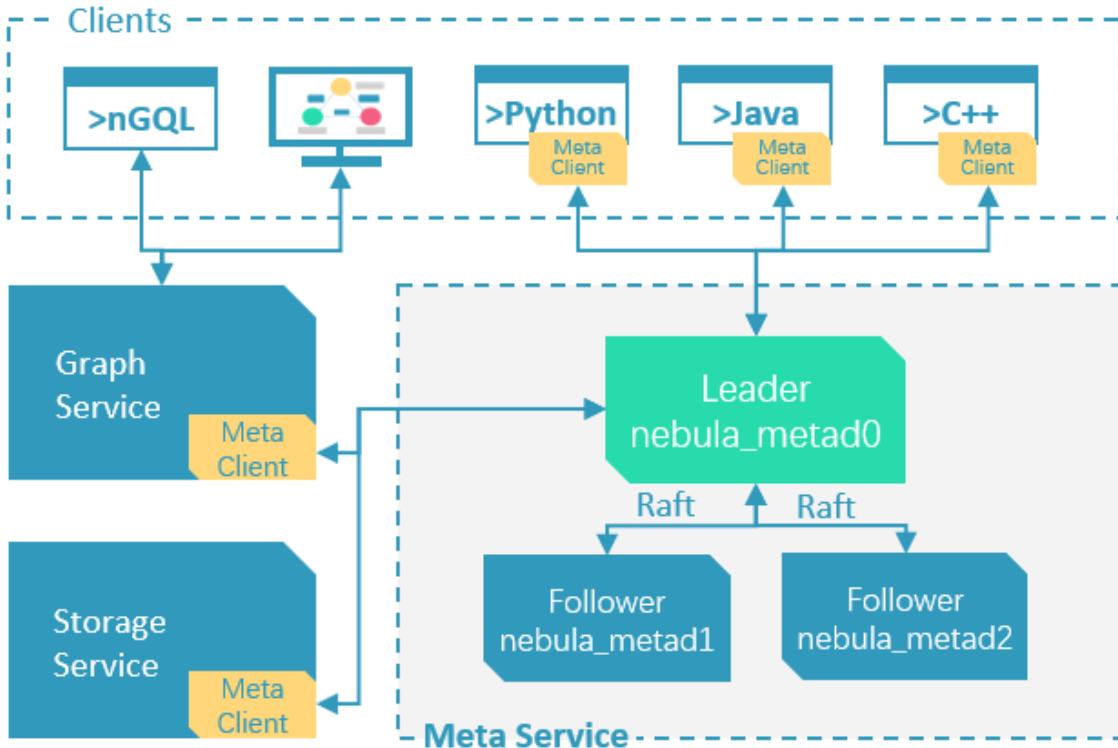
Graph 服务和 Storage 服务的详细说明，请参见 [Graph 服务](#) 和 [Storage 服务](#)。

最后更新: May 4, 2023

2.8.2 Meta 服务

本文介绍 Meta 服务的架构和功能。

Meta 服务架构



Meta 服务是由 nebula-metad 进程提供的，用户可以根据场景配置 nebula-metad 进程数量：

- 测试环境中，用户可以在NebulaGraph集群中部署 1 个或 3 个 nebula-metad 进程。如果要部署 3 个，用户可以将它们部署在 1 台机器上，或者分别部署在不同的机器上。
- 生产环境中，建议在NebulaGraph集群中部署 3 个 nebula-metad 进程。请将这些进程部署在不同的机器上以保证高可用。

所有 nebula-metad 进程构成了基于 Raft 协议的集群，其中一个进程是 leader，其他进程都是 follower。

leader 是由多数派选举出来，只有 leader 能够对客户端或其他组件提供服务，其他 follower 作为候补，如果 leader 出现故障，会在所有 follower 中选举出新的 leader。

Note

leader 和 follower 的数据通过 Raft 协议保持一致，因此 leader 故障和选举新 leader 不会导致数据不一致。更多关于 Raft 的介绍见 [Storage 服务](#)。

Meta 服务功能

管理用户账号

Meta 服务中存储了用户的账号和权限信息，当客户端通过账号发送请求给 Meta 服务，Meta 服务会检查账号信息，以及该账号是否有对应的请求权限。

更多NebulaGraph的访问控制说明，请参见[身份验证](#)。

管理分片

Meta 服务负责存储和管理分片的位置信息，并且保证分片的负载均衡。

管理图空间

NebulaGraph支持多个图空间，不同图空间内的数据是安全隔离的。Meta 服务存储所有图空间的元数据（非完整数据），并跟踪数据的变更，例如增加或删除图空间。

管理 **SCHEMA** 信息

NebulaGraph是强类型图数据库，它的 Schema 包括 Tag、Edge type、Tag 属性和 Edge type 属性。

Meta 服务中存储了 Schema 信息，同时还负责 Schema 的添加、修改和删除，并记录它们的版本。

更多NebulaGraph的 Schema 信息，请参见[数据模型](#)。

管理 **TTL** 信息

Meta 服务存储 TTL (Time To Live) 定义信息，可以用于设置数据生命周期。数据过期后，会由 Storage 服务进行处理，具体过程参见 [TTL](#)。

管理作业

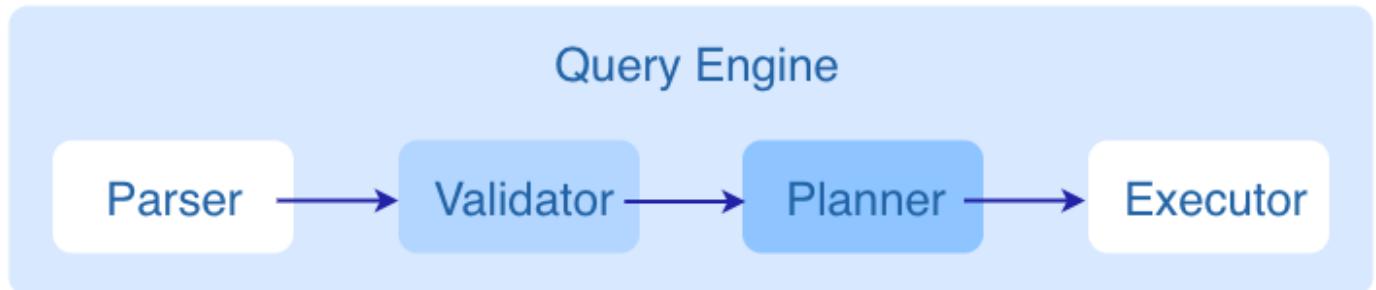
Meta 服务中的作业管理模块负责作业的创建、排队、查询和删除。

最后更新: May 4, 2023

2.8.3 Graph 服务

Graph 服务主要负责处理查询请求，包括解析查询语句、校验语句、生成执行计划以及按照执行计划执行四个大步骤，本文将基于这些步骤介绍 Graph 服务。

Graph 服务架构



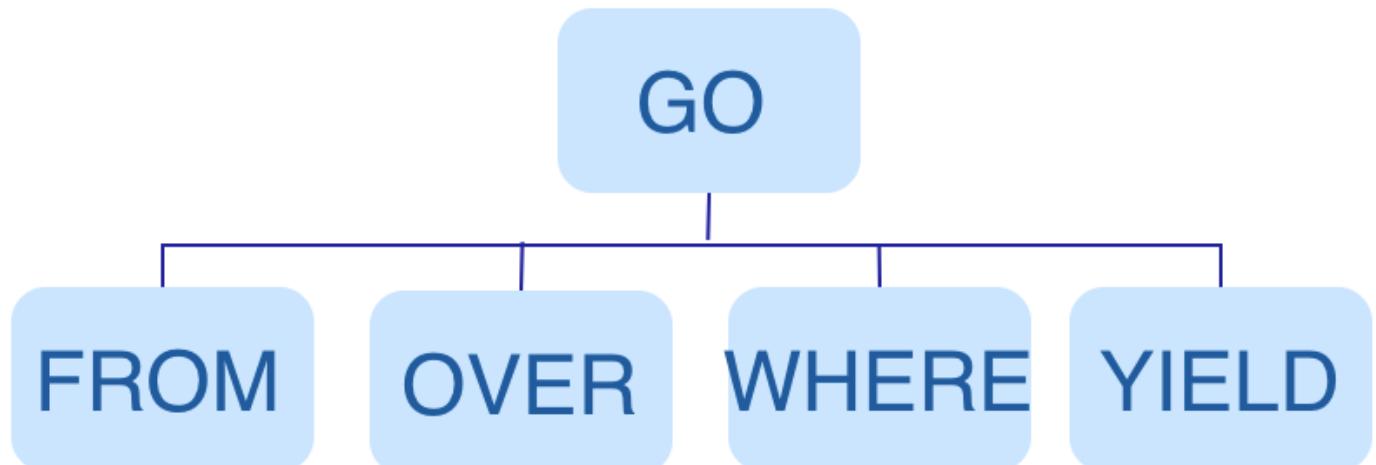
查询请求发送到 Graph 服务后，会由如下模块依次处理：

1. **Parser**: 词法语法解析模块。
2. **Validator**: 语义校验模块。
3. **Planner**: 执行计划与优化器模块。
4. **Executor**: 执行引擎模块。

Parser

Parser 模块收到请求后，通过 Flex（词法分析工具）和 Bison（语法分析工具）生成的词法语法解析器，将语句转换为抽象语法树（AST），在语法解析阶段会拦截不符合语法规则的语句。

例如 GO FROM "Tim" OVER Like WHERE properties(edge).likeness > 8.0 YIELD dst(edge) 语句转换的 AST 如下。



Validator

Validator 模块对生成的 AST 进行语义校验，主要包括：

- 校验元数据信息

校验语句中的元数据信息是否正确。

例如解析 `OVER`、`WHERE` 和 `YIELD` 语句时，会查找 Schema 校验 Edge type、Tag 的信息是否存在，或者插入数据时校验插入的数据类型和 Schema 中的是否一致。

- 校验上下文引用信息

校验引用的变量是否存在或者引用的属性是否属于变量。

例如语句 `$var = GO FROM "Tim" OVER like YIELD dst(edge) AS ID; GO FROM $var.ID OVER serve YIELD dst(edge)`，Validator 模块首先会检查变量 `var` 是否定义，其次再检查属性 `ID` 是否属于变量 `var`。

- 校验类型推断

推断表达式的结果类型，并根据子句校验类型是否正确。

例如 `WHERE` 子句要求结果是 `bool`、`null` 或者 `empty`。

- 校验 `*` 代表的信息

查询语句中包含 `*` 时，校验子句时需要将 `*` 涉及的 Schema 都进行校验。

例如语句 `GO FROM "Tim" OVER * YIELD dst(edge), properties(edge).likeness, dst(edge)`，校验 `OVER` 子句时需要校验所有的 Edge type，如果 Edge type 包含 `like` 和 `serve`，该语句会展开为 `GO FROM "Tim" OVER like,serve YIELD dst(edge), properties(edge).likeness, dst(edge)`。

- 校验输入输出

校验管道符 `(|)` 前后的一致性。

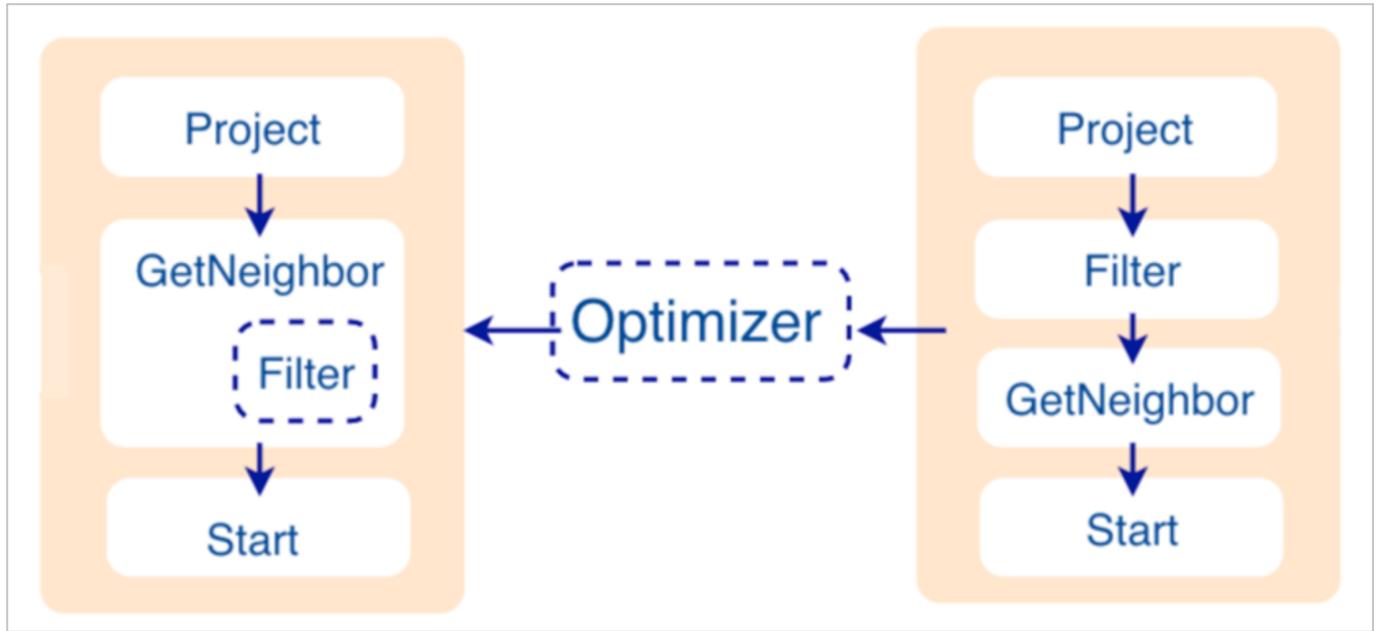
例如语句 `GO FROM "Tim" OVER like YIELD dst(edge) AS ID | GO FROM $-.ID OVER serve YIELD dst(edge)`，Validator 模块会校验 `$-.ID` 在管道符左侧是否已经定义。

校验完成后，Validator 模块还会生成一个默认可执行，但是未进行优化的执行计划，存储在目录 `src/planner` 内。

Planner

如果配置文件 `nebula-graphd.conf` 中 `enable_optimizer` 设置为 `false`，Planner 模块不会优化 Validator 模块生成的执行计划，而是直接交给 Executor 模块执行。

如果配置文件 `nebula-graphd.conf` 中 `enable_optimizer` 设置为 `true`，Planner 模块会对 Validator 模块生成的执行计划进行优化。如下图所示。



- 优化前

如上图右侧未优化的执行计划，每个节点依赖另一个节点，例如根节点 Project 依赖 Filter、Filter 依赖 GetNeighbor，最终找到叶子节点 Start，才能开始执行（并非真正执行）。

在这个过程中，每个节点会有对应的输入变量和输出变量，这些变量存储在一个哈希表中。由于执行计划不是真正执行，所以哈希表中每个 key 的 value 值都为空（除了 Start 节点，起始数据会存储在该节点的输入变量中）。哈希表定义在仓库 nebula-graph 内的 src/context/ExecutionContext.cpp 中。

例如哈希表的名称为 ResultMap，在建立 Filter 这个节点时，定义该节点从 ResultMap["GN1"] 中读取数据，然后将结果存储在 ResultMap["Filter2"] 中，依次类推，将每个节点的输入输出都确定好。

- 优化过程

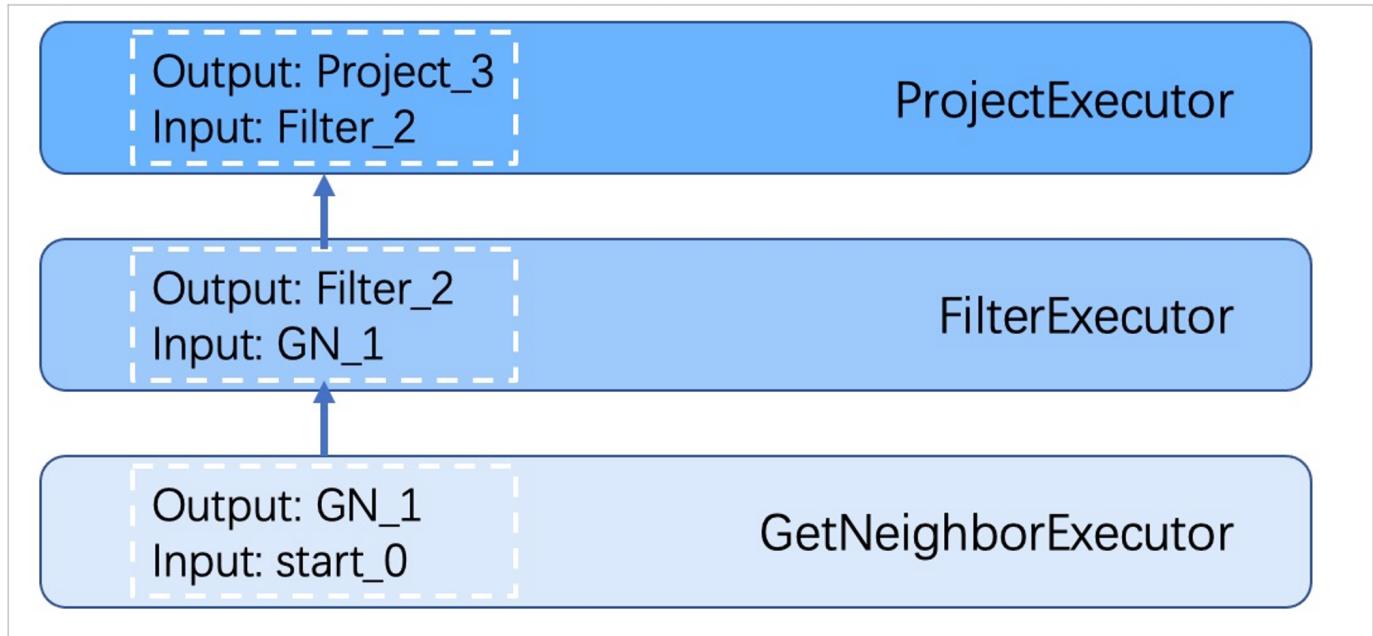
Planner 模块目前的优化方式是 RBO (rule-based optimization)，即预定义优化规则，然后对 Validator 模块生成的默认执行计划进行优化。新的优化规则 CBO (cost-based optimization) 正在开发中。优化代码存储在仓库 nebula-graph 的目录 src/optimizer/ 内。

RBO 是一个自底向上的探索过程，即对于每个规则而言，都会由执行计划的根节点（示例是 Project）开始，一步步向下探索到最底层的节点，在过程中查看是否可以匹配规则。

如上图所示，探索到节点 Filter 时，发现依赖的节点是 GetNeighbor，匹配预先定义的规则，就会将 Filter 融入到 GetNeighbor 中，然后移除节点 Filter，继续匹配下一个规则。在执行阶段，当算子 GetNeighbor 调用 Storage 服务的接口获取一个点的邻边时，Storage 服务内部会直接将不符合条件的边过滤掉，这样可以极大地减少传输的数据量，该优化称为过滤下推。

Executor

Executor 模块包含调度器 (Scheduler) 和执行器 (Executor)，通过调度器调度执行计划，让执行器根据执行计划生成对应的执行算子，从叶子节点开始执行，直到根节点结束。如下图所示。



每一个执行计划节点都一一对应一个执行算子，节点的输入输出在优化执行计划时已经确定，每个算子只需要拿到输入变量中的值进行计算，最后将计算结果放入对应的输出变量中即可，所以只需要从节点 Start 一步步执行，最后一个算子的输出变量会作为最终结果返回给客户端。

代码结构

NebulaGraph的代码层次结构如下：

```

|-src
 |--graph
   |--context //校验期和执行期上下文
   |--executor //执行算子
   |--gc //垃圾收集器
   |--optimizer //优化规则
   |--planner //执行计划结构
   |--scheduler //调度器
   |--service //对外服务管理
   |--session //会话管理
   |--stats //运行指标
   |--util //基础组件
   |--validator //语句校验
   |--visitor //visitor表达式

```

视频

用户也可以通过视频全方位了解NebulaGraph的查询引擎。

- [nMeetup·上海 | 全面解析 Query Engine \(33 分 30 秒\)](#)

最后更新: May 4, 2023

2.8.4 Storage 服务

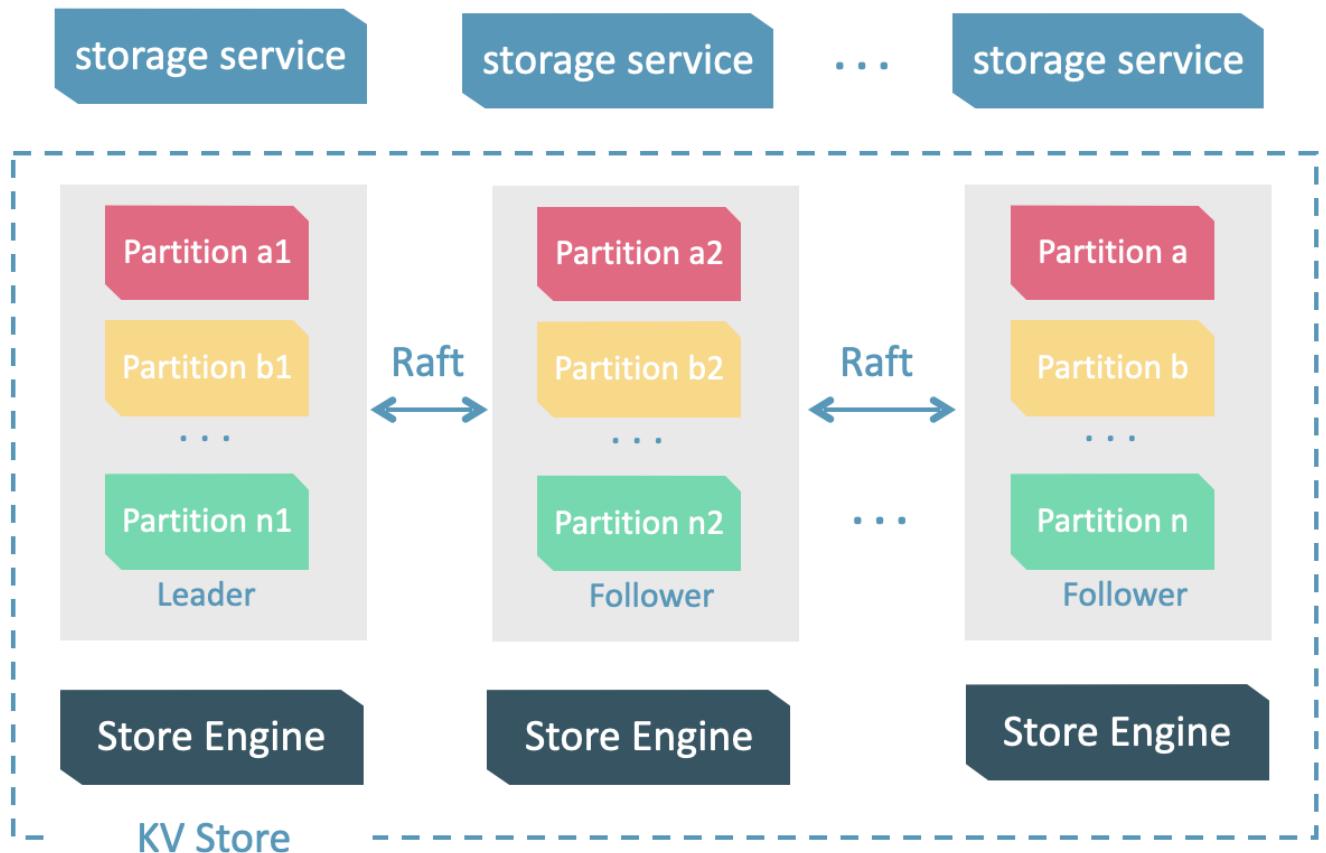
NebulaGraph的存储包含两个部分，一个是 Meta 相关的存储，称为 Meta 服务，在前文已有介绍。

另一个是具体数据相关的存储，称为 Storage 服务。其运行在 `nebula-storaged` 进程中。本文仅介绍 Storage 服务的架构设计。

优势

- 高性能（自研 KVStore）
- 易水平扩展（Shared-nothing 架构，不依赖 NAS 等硬件设备）
- 强一致性（Raft）
- 高可用性（Raft）
- 支持向第三方系统进行同步（例如[全文索引](#)）

Storage 服务架构



Storage 服务是由 `nebula-storaged` 进程提供的，用户可以根据场景配置 `nebula-storaged` 进程数量，例如测试环境 1 个，生产环境 3 个。

所有 `nebula-storaged` 进程构成了基于 Raft 协议的集群，整个服务架构可以分为三层，从上到下依次为：

- Storage interface 层

Storage 服务的最上层，定义了一系列和图相关的 API。API 请求会在这一层被翻译成一组针对分片的 KV 操作，例如：

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

正是这一层的存在，使得 Storage 服务变成了真正的图存储，否则 Storage 服务只是一个 KV 存储服务。

- Consensus 层

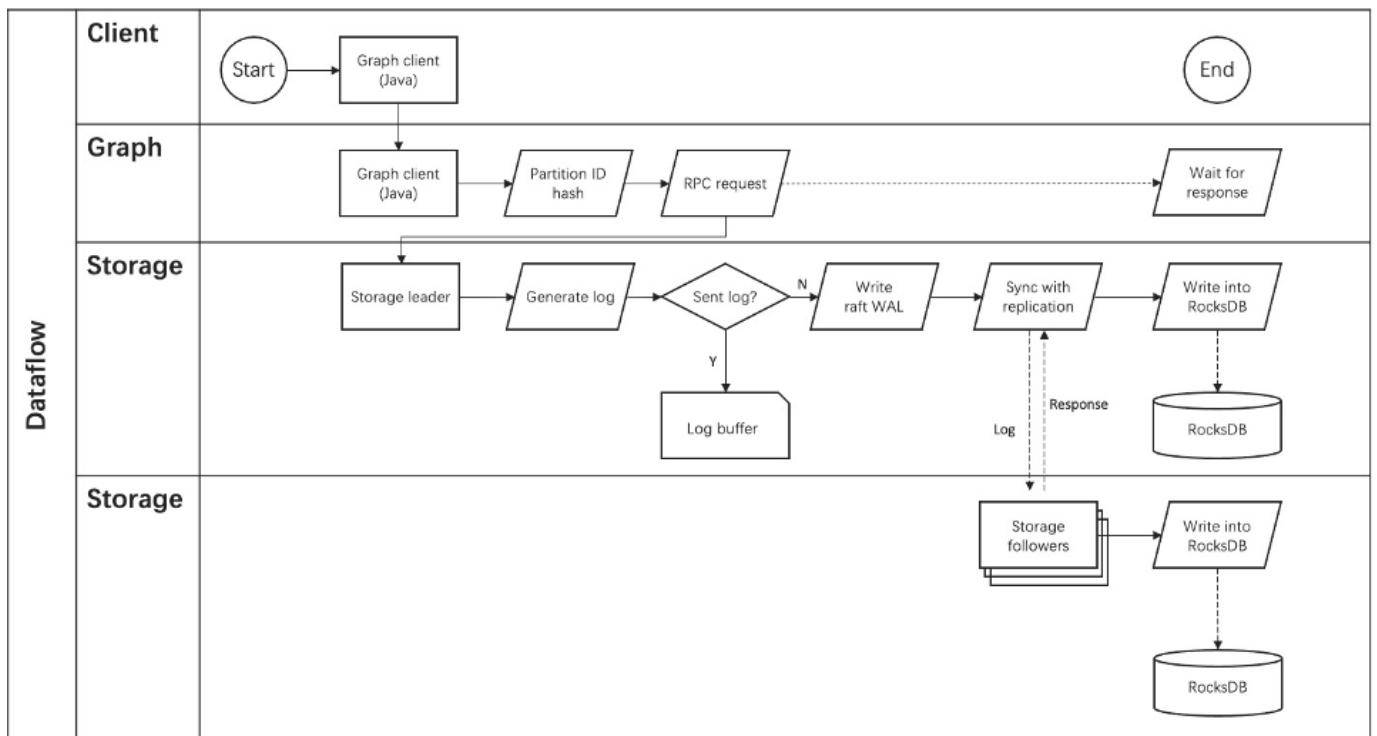
Storage 服务的中间层，实现了 **Multi Group Raft**，保证强一致性和高可用性。

- Store Engine 层

Storage 服务的最底层，是一个单机版本本地存储引擎，提供对本地数据的 `get`、`put`、`scan` 等操作。相关接口存储在 `KVStore.h` 和 `KVEngine.h` 文件，用户可以根据业务需求定制开发相关的本地存储插件。

下文将基于架构介绍 Storage 服务的部分特性。

Storage 写入流程



KVStore

NebulaGraph 使用自行开发的 KVStore，而不是其他开源 KVStore，原因如下：

- 需要高性能 KVStore。
- 需要以库的形式提供，实现高效计算下推。对于强 Schema 的NebulaGraph来说，计算下推时如何提供 Schema 信息，是高效的关键。
- 需要数据强一致性。

基于上述原因，NebulaGraph 使用 RocksDB 作为本地存储引擎，实现了自己的 KVStore，有如下优势：

- 对于多硬盘机器，NebulaGraph 只需配置多个不同的数据目录即可充分利用多硬盘的并发能力。
- 由 Meta 服务统一管理所有 Storage 服务，可以根据所有分片的分布情况和状态，手动进行负载均衡。

Note

不支持自动负载均衡是为了防止自动数据搬迁影响线上业务。

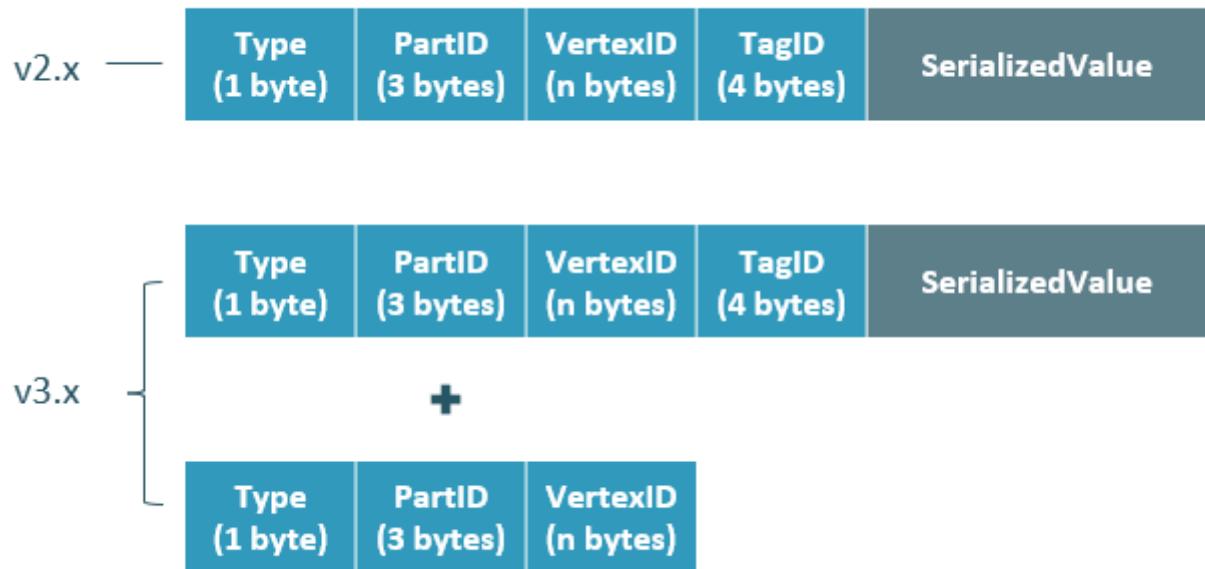
- 定制预写日志（WAL），每个分片都有自己的 WAL。
- 支持多个图空间，不同图空间相互隔离，每个图空间可以设置自己的分片数和副本数。

数据存储格式

图存储的主要数据是点和边，NebulaGraph将点和边的信息存储为 `key`，同时将点和边的属性信息存储在 `value` 中，以便更高效地使用属性过滤。

- 点数据存储格式

相比NebulaGraph 2.x 版本，3.x 版本在开启无 `Tag` 的点配置后，每个点多了一个不含 `TagID` 字段并且无 `value` 的 `key`。



字段	说明
Type	key 类型。长度为 1 字节。
PartID	数据分片编号。长度为 3 字节。此字段主要用于 Storage 负载均衡 (balance) 时方便根据前缀扫描整个分片的数据。
VertexID	点 ID。当点 ID 类型为 int 时，长度为 8 字节；当点 ID 类型为 string 时，长度为创建图空间时指定的 <code>fixed_string</code> 长度。
TagID	点关联的 Tag ID。长度为 4 字节。
SerializedValue	序列化的 value，用于保存点的属性信息。

- 边数据存储格式

Type (1 byte)	PartID (3 bytes)	VertexID (n bytes)	EdgeType (4 bytes)	Rank (8 bytes)	VertexID (n bytes)	PlaceHolder (1 byte)	SerializedValue
------------------	---------------------	-----------------------	-----------------------	-------------------	-----------------------	-------------------------	-----------------

字段	说明
Type	key 类型。长度为 1 字节。
PartID	数据分片编号。长度为 3 字节。此字段主要用于 Storage 负载均衡 (balance) 时方便根据前缀扫描整个分片的数据。
VertexID	点 ID。前一个 VertexID 在出边里表示起始点 ID，在入边里表示目的点 ID；后一个 VertexID 出边里表示目的点 ID，在入边里表示起始点 ID。
Edge type	边的类型。大于 0 表示出边，小于 0 表示入边。长度为 4 字节。
Rank	用来处理两点之间有多个同类型边的情况。用户可以根据自己的需求进行设置，例如存放交易时间、交易流水号等。长度为 8 字节。
PlaceHolder	预留字段。长度为 1 字节。
SerializedValue	序列化的 value，用于保存边的属性信息。

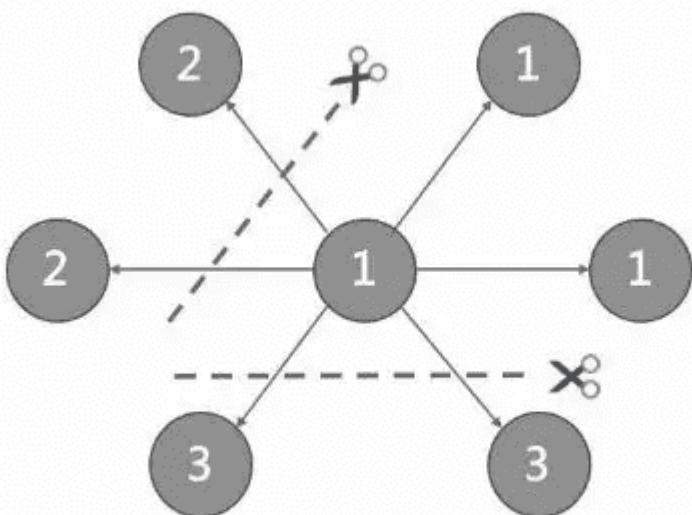
属性说明

NebulaGraph 使用强类型 Schema。

对于点或边的属性信息，NebulaGraph 会将属性信息编码后按顺序存储。由于属性的长度是固定的，查询时可以根据偏移量快速查询。在解码之前，需要先从 Meta 服务中查询具体的 Schema 信息（并缓存）。同时为了支持在线变更 Schema，在编码属性时，会加入对应的 Schema 版本信息。

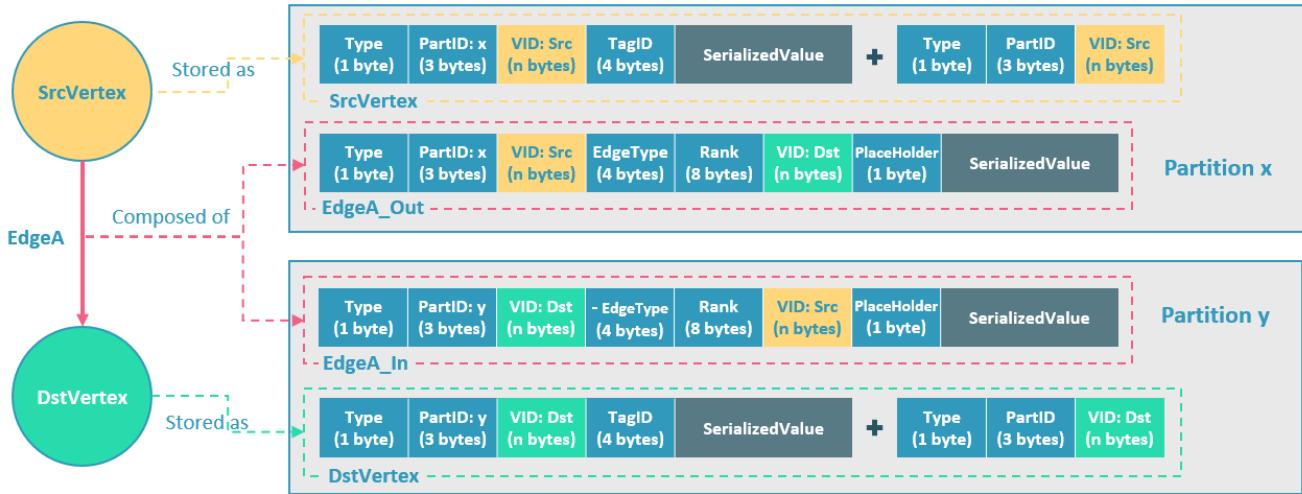
数据分片

由于超大规模关系网络的节点数量高达百亿到千亿，而边的数量更会高达万亿，即使仅存储点和边两者也远大于一般服务器的容量。因此需要有方法将图元素切割，并存储在不同逻辑分片（Partition）上。NebulaGraph 采用边分割的方式。



切边与存储放大

NebulaGraph中逻辑上的一条边对应着硬盘上的两个键值对（key-value pair），在边的数量和属性较多时，存储放大现象较明显。边的存储方式如下图所示。



上图以最简单的两个点和一条边为例，起点 SrcVertex 通过边 EdgeA 连接目的点 DstVertex，形成路径 (SrcVertex)-[EdgeA]->(DstVertex)。这两个点和一条边会以 6 个键值对的形式保存在存储层的两个不同分片，即 Partition x 和 Partition y 中，详细说明如下：

- 点 SrcVertex 的键值保存在 Partition x 中。
- 边 EdgeA 的第一份键值，这里用 EdgeA_Out 表示，与 SrcVertex 一同保存在 Partition x 中。key 的字段有 Type、PartID (x)、VID (Src, 即点 SrcVertex 的 ID)、EdgeType (符号为正，代表边方向为出)、Rank (0)、VID (Dst, 即点 DstVertex 的 ID) 和 PlaceHolder。SerializedValue 即 Value，是序列化的边属性。
- 点 DstVertex 的键值保存在 Partition y 中。
- 边 EdgeA 的第二份键值，这里用 EdgeA_In 表示，与 DstVertex 一同保存在 Partition y 中。key 的字段有 Type、PartID (y)、VID (Dst, 即点 DstVertex 的 ID)、EdgeType (符号为负，代表边方向为入)、Rank (0)、VID (Src, 即点 SrcVertex 的 ID) 和 PlaceHolder。SerializedValue 即 Value，与 EdgeA_Out 中该部分的完全相同。

EdgeA_Out 和 EdgeA_In 以方向相反的两条边的形式存在于存储层，二者组合成了逻辑上的一条边 EdgeA。EdgeA_Out 用于从起点开始的遍历请求，例如 (a)-[]->()；EdgeA_In 用于指向目的点的遍历请求，或者说从目的点开始，沿着边的方向逆序进行的遍历请求，例如 ()-[]->(a)。

如 EdgeA_Out 和 EdgeA_In 一样，NebulaGraph冗余了存储每条边的信息，导致存储边所需的实际空间翻倍。因为边对应的 key 占用的硬盘空间较小，但 value 占用的空间与属性值的长度和数量成正比，所以，当边的属性值较大或数量较多时候，硬盘空间占用量会比较大。

分片算法

分片策略采用静态 **Hash** 的方式，即对点 VID 进行取模操作，同一个点的所有 Tag、出边和入边信息都会存储到同一个分片，这种方式极大地提升了查询效率。

Note

创建图空间时需指定分片数量，分片数量设置后无法修改，建议设置时提前满足业务将来的扩容需求。

多机集群部署时，分片分布在集群内的不同机器上。分片数量在 CREATE SPACE 语句中指定，此后不可更改。

如果需要将某些点放置在相同的分片（例如在一台机器上），可以参考[公式或代码](#)。

下文用简单代码说明 VID 和分片的关系。

```
// 如果 ID 长度为 8，为了兼容 1.0，将数据类型视为 int64。
uint64_t vid = 0;
if (id.size() == 8) {
    memcpy(static_cast<void*>(&vid), id.data(), 8);
} else {
    MurmurHash2 hash;
    vid = hash(id.data());
}
PartitionID pId = vid % numParts + 1;
```

简单来说，上述代码是将一个固定的字符串进行哈希计算，转换成数据类型为 `int64` 的数字（`int64` 数字的哈希计算结果是数字本身），将数字取模，然后加 1，即：

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

示例的部分参数说明如下。

参数	说明
%	取模运算。
numParts	VID 所在图空间的分片数，即 <code>CREATE SPACE</code> 语句中的 <code>partition_num</code> 值。
pId	VID 所在分片的 ID。

例如有 100 个分片，VID 为 1、101 和 1001 的三个点将会存储在相同的分片。分片 ID 和机器地址之间的映射是随机的，所以不能假定任何两个分片位于同一台机器上。

Raft

关于 RAFT 的简单介绍

分布式系统中，同一份数据通常会有多个副本，这样即使少数副本发生故障，系统仍可正常运行。这就需要一定的技术手段来保证多个副本之间的一致性。

基本原理：Raft 就是一种用于保证多副本一致性的协议。Raft 采用多个副本之间竞选的方式，赢得“超过半数”副本投票的（候选）副本成为 Leader，由 Leader 代表所有副本对外提供服务；其他 Follower 作为备份。当该 Leader 出现异常后（通信故障、运维命令等），其余 Follower 进行新一轮选举，投票出一个新的 Leader。Leader 和 Follower 之间通过心跳的方式相互探测是否存活，并以 Raft-wal 的方式写入硬盘，超过多个心跳仍无响应的副本会被认为发生故障。

Note

因为 Raft-wal 需要定期写硬盘，如果硬盘写能力瓶颈会导致 Raft 心跳失败，导致重新发起选举。硬盘 IO 严重堵塞情况下，会导致长期无法选举出 Leader。

读写流程：对于客户端的每个写入请求，Leader 会将该写入以 Raft-wal 的方式，将该条同步给其他 Follower，并只有在“超过半数”副本都成功收到 Raft-wal 后，才会返回客户端该写入成功。对于客户端的每个读取请求，都直接访问 Leader，而 Follower 并不参与读请求服务。

故障流程：场景 1：考虑一个配置为单副本（图空间）的集群；如果系统只有一个副本时，其本身就是 Leader；如果其发生故障，系统将完全不可用。场景 2：考虑一个配置为 3 副本（图空间）的集群；如果系统有 3 个副本，其中一个副本是 Leader，其他 2 个副本是 Follower；即使原 Leader 发生故障，剩下两个副本仍可投票出一个新的 Leader（以及一个 Follower），此时系统仍可使用；但是当这 2 个副本中任一者再次发生故障后，由于投票人数不足，系统将完全不可用。

Note

Raft 多副本的方式与 HDFS 多副本的方式是不同的，Raft 基于“多数派”投票，因此副本数量不能是偶数。

MULTI GROUP RAFT

由于 Storage 服务需要支持集群分布式架构，所以基于 Raft 协议实现了 Multi Group Raft，即每个分片的所有副本共同组成一个 Raft group，其中一个副本是 leader，其他副本是 follower，从而实现强一致性和高可用性。Raft 的部分实现如下。

由于 Raft 日志不允许空洞，NebulaGraph 使用 Multi Group Raft 缓解此问题，分片数量较多时，可以有效提高 NebulaGraph 的性能。但是分片数量太多会增加开销，例如 Raft group 内部存储的状态信息、WAL 文件，或者负载过低时的批量操作。

实现 Multi Group Raft 有 2 个关键点：

- 共享 Transport 层

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，会导致连接的开销巨大。

- 共享线程池

如果不共享一组线程池，会造成系统的线程数过多，导致大量的上下文切换开销。

批量（BATCH）操作

NebulaGraph 中，每个分片都是串行写日志，为了提高吞吐，写日志时需要做批量操作，但是由于 NebulaGraph 利用 WAL 实现一些特殊功能，需要对批量操作进行分组，这是 NebulaGraph 的特色。

例如无锁 CAS 操作需要之前的 WAL 全部提交后才能执行，如果一个批量写入的 WAL 里包含了 CAS 类型的 WAL，就需要拆分成粒度更小的几个组，还要保证这几组 WAL 串行提交。

LEADER 切换（TRANSFER LEADERSHIP）

leader 切换对于负载均衡至关重要，当把某个分片从一台机器迁移到另一台机器时，首先会检查分片是不是 leader，如果是的话，需要先切换 leader，数据迁移完毕之后，通常还要重新 **均衡 leader 分布**。

对于 leader 来说，提交 leader 切换命令时，就会放弃自己的 leader 身份，当 follower 收到 leader 切换命令时，就会发起选举。

成员变更

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，该状态下新旧 group 的多数派需要有重叠的部分，这样就防止了新的 group 或旧的 group 单方面做出决定。为了更加简化，Diego Ongaro 在自己的博士论文中提出每次只增减一个 peer 的方式，以保证新旧 group 的多数派总是有重叠。NebulaGraph 也采用了这个方式，只不过增加成员和移除成员的实现有所区别。具体实现方式请参见 Raft Part class 里 addPeer/removePeer 的实现。

与 HDFS 的区别

Storage 服务基于 Raft 协议实现的分布式架构，与 HDFS 的分布式架构有一些区别。例如：

- Storage 服务本身通过 Raft 协议保证一致性，副本数量通常为奇数，方便进行选举 leader，而 HDFS 存储具体数据的 DataNode 需要通过 NameNode 保证一致性，对副本数量没有要求。
- Storage 服务只有 leader 副本提供读写服务，而 HDFS 的所有副本都可以提供读写服务。
- Storage 服务无法修改副本数量，只能在创建图空间时指定副本数量，而 HDFS 可以调整副本数量。
- Storage 服务是直接访问文件系统，而 HDFS 的上层（例如 HBase）需要先访问 HDFS，再访问到文件系统，远程过程调用（RPC）次数更多。

总而言之，Storage 服务更加轻量级，精简了一些功能，架构没有 HDFS 复杂，可以有效提高小块存储的读写性能。

最后更新: May 4, 2023

3. 快速入门

3.1 快速入门概览

用户可通过 Docker Desktop、云、本地三种部署方式快速入门NebulaGraph。快速入门将介绍如何通过 Docker Desktop、云、本地三种部署方式简单地使用NebulaGraph，包括部署、连接NebulaGraph，以及基础的增删改查操作。

3.1.1 使用 Docker Desktop 一键部署

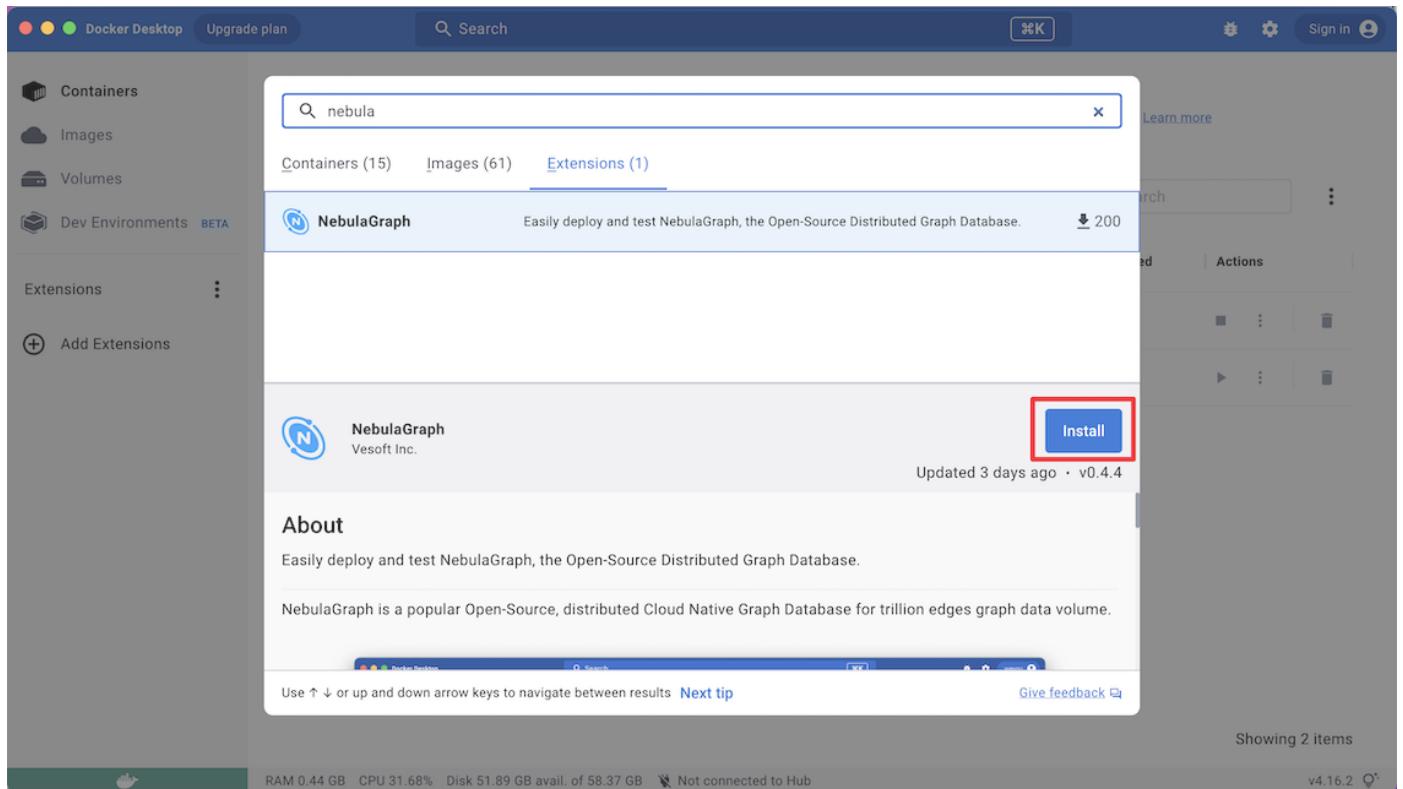
按照以下步骤可以快速在 Docker Desktop 中部署 NebulaGraph。

1. 安装 Docker Desktop。

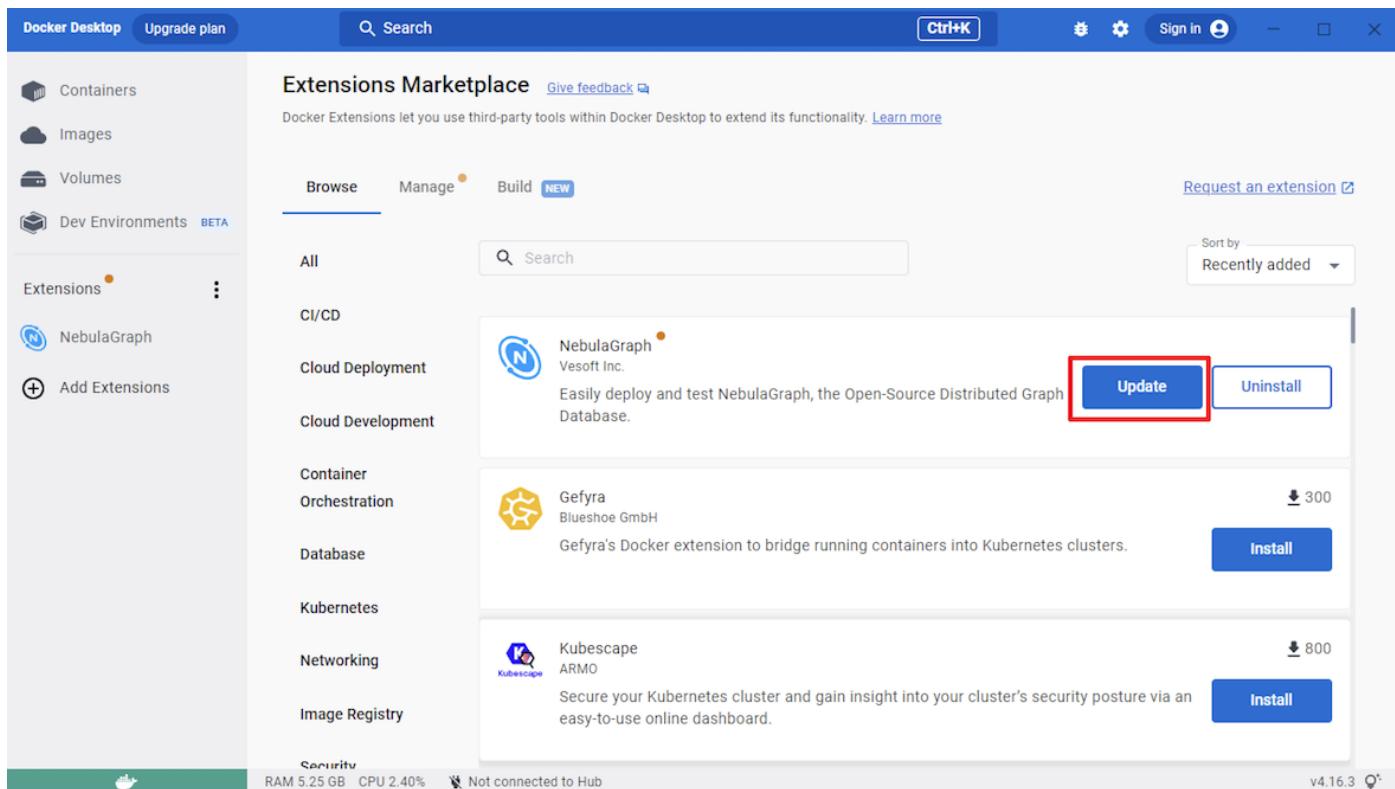
Caution

如果在 Windows 端安装 Docker Desktop 需[安装 WSL 2](#)。

2. 在仪表盘中单击 Extensions 或 Add Extensions 打开Extensions Marketplace 搜索NebulaGraph，也可以点击 **NebulaGraph** 在 Docker Desktop 打开。
3. 导航到NebulaGraph的扩展市场。
4. 点击 Install 下载NebulaGraph。



5. 在有更新的时候，可以点击 Update 更新到最新版本。



视频介绍



3.1.2 云上部署

使用流程图



操作步骤

按照以下步骤可以快速在云上部署并且使用NebulaGraph。

1. [创建云服务实例](#)
2. [连接NebulaGraph](#)
3. [管理NebulaGraph数据](#)

更多方式

快速入门使用基于阿里云的自管云服务，即 NebulaGraph Cloud 阿里云版。您还可以使用其他云服务。更多云服务的详情，参见[什么是 NebulaGraph Cloud](#)。

3.1.3 本地部署

使用流程图



操作步骤

按照以下步骤可以快速在本地部署并且使用NebulaGraph。

1. 安装NebulaGraph
2. 启动NebulaGraph
3. 连接NebulaGraph
4. 注册 Storage 服务
5. 使用基本 nGQL (CURD 操作)

更多方式

快速入门使用 RPM 或 DEB 文件安装NebulaGraph，您还可以使用其他本地部署方式安装NebulaGraph。关于其它部署方式及相应的准备工作，参见文档目录安装部署。

3.1.4 视频

用户也可以观看视频快速了解NebulaGraph的相关概念和操作。

NebulaGraph Academy 系列课程

- 第一课：图的概念 (03 分 45 秒)

- 第二课：图的结构 (02 分 24 秒)

请访问 [Bilibili 空间](#)，查看更多视频。

热点视频

- Foesa 小学姐课堂——NebulaGraph那些磨人的概念 (04 分 20 秒)

- Foesa 小学姐课堂——path 的三种类型 (03 分 09 秒)

- Foesa 小学姐课堂——悬挂边 (02 分 27 秒)

- Foesa 小学姐课堂——自环 (02 分 53 秒)

- Nebula Explore Demo Show (02 分 53 秒)

最后更新: May 4, 2023

3.2 从云开始（免费试用）

3.2.1 创建云服务实例

用户可以在云服务上创建NebulaGraph实例，本文介绍如何在阿里云上快速创建NebulaGraph实例。

背景信息

NebulaGraph支持在多个云平台上部署NebulaGraph，本文只介绍如何在 NebulaGraph Cloud 阿里云版创建NebulaGraph实例。更多云服务平台的支持情况，参见 [NebulaGraph Cloud](#)。



使用 NebulaGraph Cloud 阿里云版创建的实例对应 3.4.0 版本的NebulaGraph。

账号与权限

- 准备[阿里云账号](#)。
- 如果使用 RAM 用户创建实例，需为其添加以下权限：
 - AliyunECSFullAccess
 - AliyunVPCFullAccess
 - AliyunROSFullyAccess
 - AliyunCloudMonitorFullAccess
 - AliyunComputeNestUserFullAccess

资源与费用

NebulaGraph Cloud 阿里云版支持免费试用和付费使用，二者的详细说明如下表。

项目	免费试用版	付费版
云资源归属	阿里云官方账号	用户账号
云资源费用	阿里云承担	用户承担
云资源使用时长	30 天（不可续期）	用户创建实例时选择
NebulaGraph许可证费用	试用期内免费	用户承担
NebulaGraph服务许可证有效期	30 天	用户创建实例时选择
许可证过期后数据是否保留	否（因云资源会同时到期）	是
云资源到期后数据是否保留	否	否
数据盘快照费用	阿里云承担	用户承担（可关闭功能）

数据备份

在创建服务实例时，系统会默认开启数据盘自动快照备份，用于周期性备份NebulaGraph数据。

创建付费版服务实例前需[开通快照](#)。免费试用实例使用的是阿里云官方账号下的资源，已开通快照，无需用户手动开通。

默认的快照备份策略为每天 00:00 创建快照。

创建快照会产生费用，详情参见[快照计费](#)。

如需删除快照、修改或删除自动快照策略，参见[快照文档](#)。

套餐版本

NebulaGraph Cloud 阿里云版支持如下套餐版本。

套餐版本	说明
基础版	将所有NebulaGraph服务节点部署在 1 台 ECS 服务器上。
标准版	将 Graph 服务和 Storage 服务分别部署在不同的 ECS 服务器上，每个服务都是单节点（1 台 ECS）。将 NebulaGraph Explorer 等生态工具混合部署在 1 台 ECS 服务器上。
高可用版	将 Graph 服务和 Storage 服务分别部署在不同的 ECS 服务器上，每个服务都包含 3 节点（3 台 ECS）。将 NebulaGraph Explorer 等生态工具混合部署在 1 台 ECS 服务器上。

付费版和免费试用版服务实例支持的套餐版本有所不同，详情参见创建实例页面。

创建付费版服务实例

1. 登录[阿里云控制台](#)。
2. 打开云市场的 [NebulaGraph产品页](#)。
3. 选择套餐版本和购买时长（即NebulaGraph服务的许可证有效期），并单击立即购买。



当前云市场显示的部分价格并非实际售价，购买时需通过右侧的钉钉客服确认实际价格。

4. 在创建服务实例页，保持选择模板处的选择不变。如需切换部署架构，可改变选中的选项，重新选择模板。



改变模板会改变之前选择的套餐版本，软件费用（NebulaGraph许可证费用）和创建服务实例需要的资源也会改变。

5. （可选）设置服务实例名称。默认值为服务实例 ID。
6. 选择要创建服务实例的地域。
7. 在付费模式设置区域，指定 ECS 服务器的付费方式。默认为按量付费。
 - 按量付费：按照计费周期计费，在每个结算周期生成账单并从账户中扣除相应费用。详情参见[按量付费](#)。
 - 包年包月：先付费后使用。详情参见[包年包月](#)。选择包年包月模式需要指定购买时长周期和购买时长。购买时长周期当前仅支持 **Month**，即按月购买。
8. 在 **NebulaGraph**配置区域，完成数据盘和 ECS 服务器密码设置。
9. 在基础设施配置区域，完成以下设置。
 - a. 选择专有网络 **VPC** 实例 **ID**。

如果下拉列表为空，先单击其右侧的新建专有网络，完成专有网络创建。
 - b. 选择交换机可用区。
 - c. 选择业务网络交换机的实例 **ID**。

如果下拉列表为空，先单击其右侧的新建交换机，完成交换机创建。
10. 完成权限确认，并选中我同意授权服务商（杭州悦数科技有限公司）获取上述权限以提供代运维服务。
11. 在页面底部，单击下一步：确认订单。
12. 在服务条款区域，勾选我已阅读并同意《商品在线协议》《云市场平台服务协议》《计算巢服务协议》。

13. 在页面底部，单击立即支付。
14. 在支付页面，选择支付方式，之后单击页面底部的支付。
15. 在支付完成页面，单击管理控制台。
16. 在已购买的服务页面，单击要创建的实例右侧的立即进入计算巢部署或查看部署详情。

Note

如果已购买的服务列表中未显示要创建的实例，刷新页面。

17. 在服务实例详情页查看实例的状态，确保状态为部署中。

部署的平均耗时为 10 分钟。完成后实例的状态变为已部署。

创建免费版试用版服务实例

Danger

免费试用实例不可在生产环境中使用。

1. 登录阿里云计算巢[推荐服务](#)页面，搜索 **NebulaGraph**。
2. 在 **NebulaGraph**集群版卡片上，单击免费试用。
3. 系统会为首次创建NebulaGraph服务实例的用户弹出申请对话框。在申请权限对话框中填写申请人信息。带有红色星号 (*) 的为必填项。

Note

申请通过后需从第 1 步重新开始。

4. (可选) 设置服务实例名称。默认值为服务实例 ID。
5. 选择要创建实例的地域。
6. 在 **NebulaGraph**配置区域，完成数据盘和 ECS 服务器密码设置。
7. 在基础设施配置区域，选择交换机可用区。
8. 完成权限确认，并选中我同意授权服务商（杭州悦数科技有限公司）获取上述权限以提供代运维服务。
9. 在页面底部，单击下一步：确认订单。
10. 在服务条款区域，勾选我已阅读并同意《计算巢服务协议》。
11. 在页面底部，单击开始免费试用。
12. 在提交成功页面，单击去列表查看。
13. 在实例列表中查看目标实例的状态，确保状态为部署中。

部署的平均耗时为 10 分钟。完成后实例的状态变为已部署。

常见问题

Q: 服务实例的状态显示为部署失败怎么处理?

1. 如果使用 RAM 账号创建的实例，确认为该账号授予了本文前提条件中指定的权限。
2. 如果权限符合要求，[删除](#)创建失败的实例，尝试重新创建。
3. 如果仍然创建失败，到 [NebulaGraph论坛](#)寻求帮助。

下一步

[连接 NebulaGraph](#)

最后更新: May 4, 2023

3.2.2 连接NebulaGraph

创建好云服务实例后，可以连接NebulaGraph。本文介绍如何使用NebulaGraph Explorer快速连接NebulaGraph。

前提条件

[创建云服务实例](#)

操作步骤

本文使用 **NebulaGraph Explorer** 快速连接NebulaGraph。操作步骤如下：

1. 登录[服务实例管理](#)页面。
2. 查看已创建的实例。
 - 对于正式版实例，单击私有部署服务页签。
 - 对于免费试用版实例，单击试用服务页签。
3. 在实例列表中，单击目标实例的服务实例**ID**，或其右侧操作列的详情。
4. 在概览页签的基本信息区域，查看**nebula_private_ip**及**explorer_portal**信息。
5. 单击**explorer_portal**对应链接，进入NebulaGraph Explorer登录页面。
6. 填写登录信息，单击登录。
 - **Host:** `nebula_private_ip地址:9669`，例如 `192.168.98.160:9669`。
 - 用户名：`root`。
 - 密码：任意密码。



Note

用户可修改登录用户名及密码，详情参见[用户管理](#)。

更多连接方式

用户可根据自己的需求选择其他连接方式。关于更多连接方式，参见[连接NebulaGraph](#)。

下一步

[管理NebulaGraph数据](#)

最后更新: May 4, 2023

3.2.3 管理NebulaGraph数据

nGQL 是 NebulaGraph 创建的声明式图查询语言。用户可以使用 nGQL 语句对 NebulaGraph 数据库进行增删改查。本文介绍如何通过 NebulaGraph Cloud 中 NebulaGraph Explorer 的控制台功能快速使用 Nebula Graph 基本语句。

Note

用户可以在 NebulaGraph Explorer 中通过可视化查询 NebulaGraph 数据。本文介绍通过控制台功能使用 nGQL 语句增删改查 NebulaGraph 数据。关于如何可视化使用 NebulaGraph Explorer 查询数据，参见[开始探索](#)和[探索拓展](#)，或者观看视频。

前提条件

[连接NebulaGraph](#)

基本流程



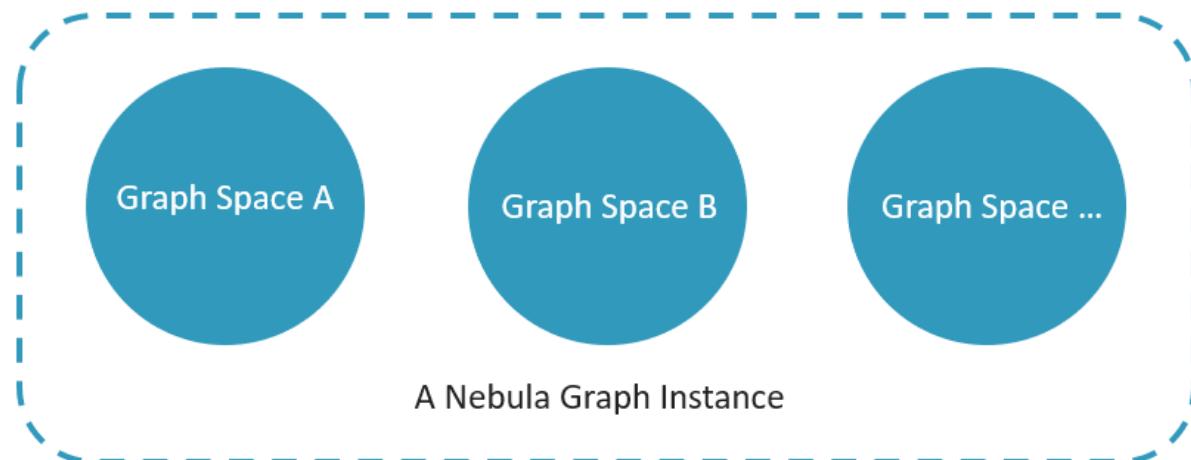
操作入口

在 NebulaGraph Explorer 页面的右上方，单击 进入控制台页面。在控制台命令行中执行 nGQL 语句。

使用说明

图空间和 SCHEMA

一个 NebulaGraph 实例由一个或多个图空间组成。每个图空间都是物理隔离的，用户可以在同一个实例中使用不同的图空间存储不同的数据集。

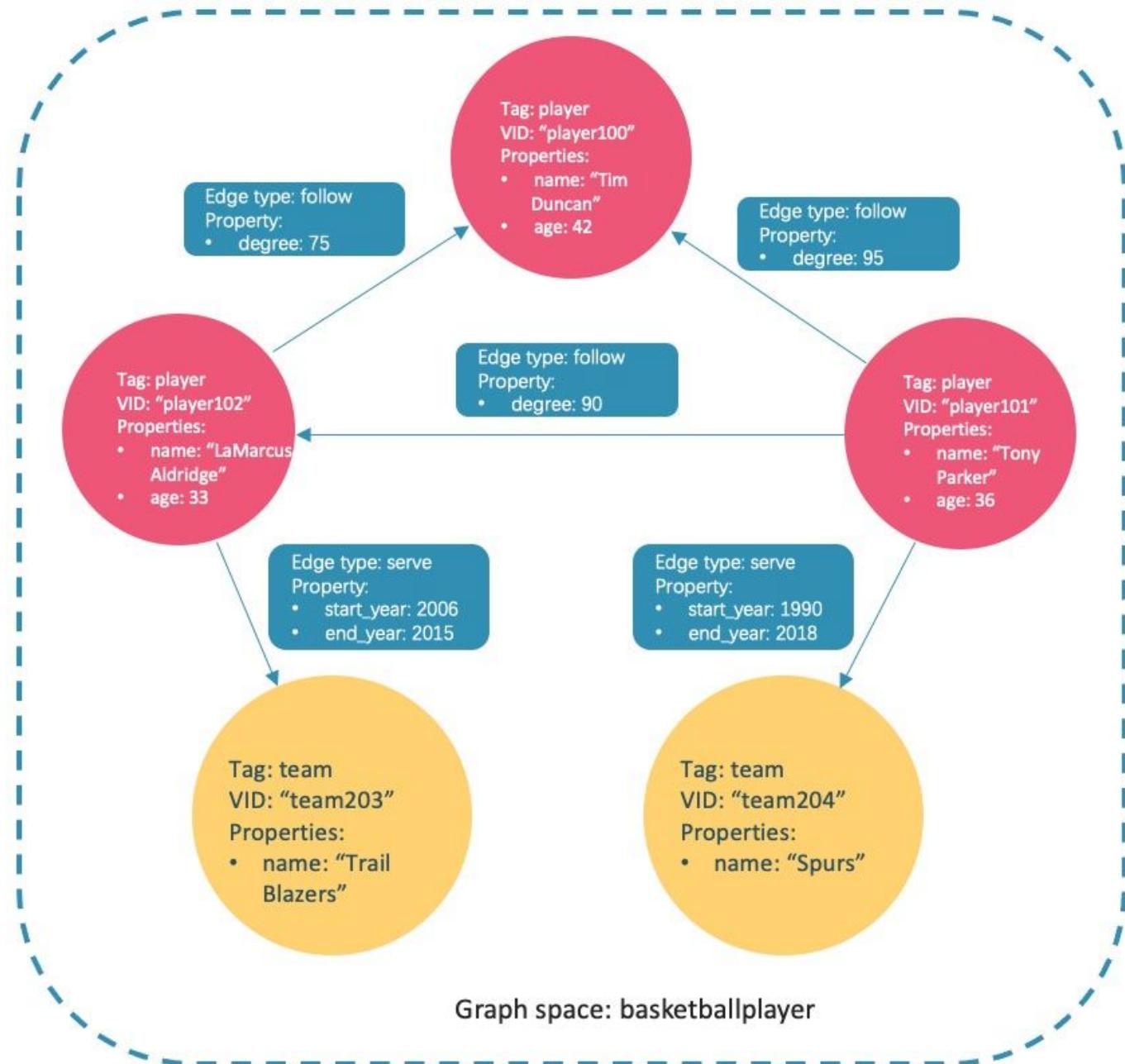


为了在图空间中插入数据，需要为图数据库定义一个 Schema。NebulaGraph的 Schema 是由如下几部分组成。

组成部分	说明
点 (Vertex)	表示现实世界中的实体。一个点可以有 0 到多个标签。
标签 (Tag)	点的类型，定义了一组描述点类型的属性。
边 (Edge)	表示两个点之间有方向的关系。
边类型 (Edge type)	边的类型，定义了一组描述边的类型的属性。

更多信息，请参见[数据结构](#)。

本文将使用下图的数据集演示基础操作的语法。



异步实现创建和修改

Caution

在NebulaGraph中，下列创建和修改操作是异步实现的。要在下一个心跳周期之后才能生效，否则访问会报错。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

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

Note

默认心跳周期是 10 秒。修改心跳周期参数 `heartbeat_interval_secs`，请参见[配置简介](#)。

第一步：创建和选择图空间

[NGQL 语法](#)

- 创建图空间

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

参数详情请参见 [CREATE SPACE](#)。

- 列出创建成功的图空间

```
nebula> SHOW SPACES;
```

- 选择数据库

```
USE <graph_space_name>;
```

示例

1. 执行如下语句创建名为 basketballplayer 的图空间。

```
nebula> CREATE SPACE basketballplayer(partition_num=15, replica_factor=1, vid_type=fixed_string(30));
```

Note

如果报错提示 [ERROR (-1005)]: Host not enough!，请检查是否已[添加 Storage 主机](#)。

2. 执行命令 SHOW HOSTS 检查分片的分布情况，确保平衡分布。

Host	Port	Status	Leader count	Leader distribution	Partition distribution	Version
------	------	--------	--------------	---------------------	------------------------	---------

"storaged0"	9779	"ONLINE"	5	"basketballplayer:5"	"basketballplayer:5"	"3.4.1"
"storaged1"	9779	"ONLINE"	5	"basketballplayer:5"	"basketballplayer:5"	"3.4.1"
"storaged2"	9779	"ONLINE"	5	"basketballplayer:5"	"basketballplayer:5"	"3.4.1"

如果 **Leader distribution** 分布不均匀, 请执行命令 `BALANCE LEADER` 重新分配。更多信息, 请参见 [Storage 负载均衡](#)。

3. 选择图空间 `basketballplayer`。

```
nebula[(none)]> USE basketballplayer;
```

用户可以执行命令 `SHOW SPACES` 查看创建的图空间。

```
nebula> SHOW SPACES;
+-----+
| Name |
+-----+
| "basketballplayer" |
+-----+
```

第二步: 创建 Tag 和 Edge type

NGQL 语法

```
CREATE {TAG | EDGE} [IF NOT EXISTS] {<tag_name> | <edge_type_name>}
(
    <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
    [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
)
[TTL_DURATION = <ttl_duration>]
[TTL_COL = <prop_name>]
[COMMENT = '<comment>'];
```

参数详情请参见 [CREATE TAG](#) 和 [CREATE EDGE](#)。

示例

创建 Tag: `player` 和 `team`, 以及 Edge type: `follow` 和 `serve`。说明如下表。

名称	类型	属性
player	Tag	name (string), age (int)
team	Tag	name (string)
follow	Edge type	degree (int)
serve	Edge type	start_year (int), end_year (int)

```
nebula> CREATE TAG player(name string, age int);
nebula> CREATE TAG team(name string);
nebula> CREATE EDGE follow(degree int);
nebula> CREATE EDGE serve(start_year int, end_year int);
```

第三步: 插入数据

用户可以使用 `INSERT` 语句, 基于现有的 Tag 插入点, 或者基于现有的 Edge type 插入边。

NGQL 语法

- 插入点

```
INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...]
VALUES <vid>; ([prop_value_list])

tag_props:
tag_name ([prop_name_list])

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

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

`vid` 是 `Vertex ID` 的缩写，`vid` 在一个图空间中是唯一的。参数详情请参见 [INSERT VERTEX](#)。

- 插入边

```
INSERT EDGE [IF NOT EXISTS] <edge_type> ( <prop_name_list> ) VALUES
<src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list>
[, <src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> ), ...];
<prop_name_list> ::= [<prop_name> [, <prop_name> ] ...]
<prop_value_list> ::= [<prop_value> [, <prop_value> ] ...]
```

参数详情请参见 [INSERT EDGE](#)。

示例

- 插入代表球员和球队的点。

```
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42);
nebula> INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36);
nebula> INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);
nebula> INSERT VERTEX team(name) VALUES "team203":("Trail Blazers"), "team204":("Spurs");
```

- 插入代表球员和球队之间关系的边。

```
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player100":(95);
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player102":(90);
nebula> INSERT EDGE follow(degree) VALUES "player102" -> "player100":(75);
nebula> INSERT EDGE serve(start_year, end_year) VALUES "player101" -> "team204":(1999, 2018), "player102" -> "team203":(2006, 2015);
```

第四步：查询数据

- `GO` 语句可以根据指定的条件遍历数据库。`GO` 语句从一个或多个点开始，沿着一条或多条边遍历，返回 `YIELD` 子句中指定的信息。
- `FETCH` 语句可以获得点或边的属性。
- `LOOKUP` 语句是基于索引的，和 `WHERE` 子句一起使用，查找符合特定条件的数据。
- `MATCH` 语句是查询图数据最常用的，可以灵活的描述各种图模式，但是它依赖索引去匹配NebulaGraph中的数据模型，性能也需要调优。

NGQL 语法

- GO

```
GO [[<N> TO] <N> [STEP|STEPS]] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
YIELD [DISTINCT] <return_list>
[ { SAMPLE <sample_list> | <limit_by_list_clause> } ]
[ | GROUP BY {<col_name> | expression} | <position> ] YIELD <col_name>]
```

```
[] ORDER BY <expression> [{ASC | DESC}]
[] LIMIT [<offset>,<number_rows>];
```

- **FETCH**

- 查询 Tag 属性

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
<vid> [, vid ...]
YIELD <return_list> [AS <alias>];
```

- 查询边属性

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

- **LOOKUP**

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

<return_list>
<prop_name> [AS <col_alias>] [, <prop_name> [AS <prop_alias>] ...];
```

- **MATCH**

```
MATCH <pattern> [<clause_1>] RETURN <output> [<clause_2>];
```

GO 语句示例

- 从 VID 为 player101 的球员开始，沿着边 follow 找到连接的球员。

```
nebula> GO FROM "player101" OVER follow YIELD id($$);
+-----+
| id($$) |
+-----+
| "player100" |
```

```
| "player102" |
+-----+
```

- 从 VID 为 player101 的球员开始，沿着边 follow 查找年龄大于或等于 35 岁的球员，并返回他们的姓名和年龄，同时重命名对应的列。

```
nebula> GO FROM "player101" OVER follow WHERE properties($$).age >= 35 \
    YIELD properties($$).name AS Teammate, properties($$).age AS Age;
+-----+-----+
| Teammate | Age |
+-----+-----+
| "Tim Duncan" | 42 |
+-----+
```

子句/符号	说明
YIELD	指定该查询需要返回的值或结果。
\$\$	表示边的终点。
\	表示换行继续输入。

- 从 VID 为 player101 的球员开始，沿着边 follow 查找连接的球员，然后检索这些球员的球队。为了合并这两个查询请求，可以使用管道符或临时变量。

- 使用管道符

```
nebula> GO FROM "player101" OVER follow YIELD dst(edge) AS id | \
    GO FROM $-.id OVER serve YIELD properties($$).name AS Team, \
    properties($$).name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| "Trail Blazers" | "LaMarcus Aldridge" |
+-----+
```

子句/符号	说明
\$^	表示边的起点。
	组合多个查询的管道符，将前一个查询的结果集用于后一个查询。
\$-	表示管道符前面的查询输出的结果集。

- 使用临时变量



当复合语句作为一个整体提交给服务器时，其中的临时变量会在语句结束时被释放。

```
nebula> $var = GO FROM "player101" OVER follow YIELD dst(edge) AS id; \
    GO FROM $var.id OVER serve YIELD properties($$).name AS Team, \
    properties($$).name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| "Trail Blazers" | "LaMarcus Aldridge" |
+-----+
```

FETCH 语句示例

查询 VID 为 player100 的球员的属性。

```
nebula> FETCH PROP ON player "player100" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 42, name: "Tim Duncan"} |
+-----+
```

Note

`LOOKUP` 和 `MATCH` 的示例在下文的 [索引](#) 部分查看。

其他操作

修改点和边

用户可以使用 `UPDATE` 语句或 `UPSERT` 语句修改现有数据。

`UPSERT` 是 `UPDATE` 和 `INSERT` 的结合体。当使用 `UPSERT` 更新一个点或边，如果它不存在，数据库会自动插入一个新的点或边。

Note

每个 `partition` 内部，`UPSERT` 操作是一个串行操作，所以执行速度比执行 `INSERT` 或 `UPDATE` 慢很多。其仅在多个 `partition` 之间有并发。

nGQL 语法

- `UPDATE` 点

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

- `UPDATE` 边

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

- `UPSERT` 点或边

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

示例

- 用 `UPDATE` 修改 VID 为 `player100` 的球员的 `name` 属性，然后用 `FETCH` 语句检查结果。

```
nebula> UPDATE VERTEX "player100" SET player.name = "Tim";
nebula> FETCH PROP ON player "player100" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
```

```
+-----+
| {age: 42, name: "Tim"} |
+-----+
```

- 用 `UPDATE` 修改某条边的 `degree` 属性，然后用 `FETCH` 检查结果。

```
nebula> UPDATE EDGE ON follow "player101" -> "player100" SET degree = 96;
nebula> FETCH PROP ON follow "player101" -> "player100" YIELD properties(edge);
+-----+
| properties(EDGE) |
+-----+
| {degree: 96}     |
+-----+
```

- 用 `INSERT` 插入一个 VID 为 `player111` 的点，然后用 `UPSERT` 更新它。

```
nebula> INSERT VERTEX player(name,age) VALUES "player111":("David West", 38);
nebula> UPSERT VERTEX "player111" SET player.name = "David", player.age = $^.player.age + 11 \
    WHEN $.player.name == "David West" AND $.player.age > 20 \
    YIELD $.player.name AS Name, $.player.age AS Age;
+-----+
| Name   | Age |
+-----+
| "David" | 49 |
+-----+
```

删除点和边

nGQL 语法

- 删除点

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

- 删除边

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

示例

- 删除点

```
nebula> DELETE VERTEX "player111", "team203";
```

- 删除边

```
nebula> DELETE EDGE follow "player101" -> "team204";
```

使用索引

用户可以通过 `CREATE INDEX` 语句为 `Tag` 和 `Edge type` 增加索引。

 使用索引必读

`MATCH` 和 `LOOKUP` 语句的执行都依赖索引，但是索引会导致写性能大幅降低。请不要随意在生产环境中使用索引，除非很清楚使用索引对业务的影响。

必须为“已写入但未构建索引”的数据重建索引，否则无法在 `MATCH` 和 `LOOKUP` 语句中返回这些数据。参见[重建索引](#)。

nGQL 语法

- 创建索引

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

- 重建索引

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

Note

为没有指定长度的变量属性创建索引时，需要指定索引长度。在 utf-8 编码中，一个中文字符占 3 字节，请根据变量属性长度设置合适的索引长度。例如 10 个中文字符，索引长度需要为 30。详情请参见[创建索引](#)。

基于索引的 **LOOKUP** 和 **MATCH** 示例

确保 **LOOKUP** 或 **MATCH** 有一个索引可用。如果没有，请先创建索引。

找到 Tag 为 `player` 的点的信息，它的 `name` 属性值为 `Tony Parker`。

```
// 为 name 属性创建索引 player_index_1。
nebula> CREATE TAG INDEX IF NOT EXISTS player_index_1 ON player(name(20));

// 重建索引确保能对已存在数据生效。
nebula> REBUILD TAG INDEX player_index_1
+-----+
| New Job Id |
+-----+
| 31          |
+-----+

// 使用 LOOKUP 语句检索点的属性。
nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
    YIELD properties(vertex).name AS name, properties(vertex).age AS age;
+-----+-----+
| name      | age   |
+-----+-----+
| "Tony Parker" | 36   |
+-----+-----+


// 使用 MATCH 语句检索点的属性。
nebula> MATCH (v:player{name:"Tony Parker"}) RETURN v;
+-----+
| v           |
+-----+
| {"player101" :player{age: 36, name: "Tony Parker"}) |
+-----+
```

最后更新: May 4, 2023

3.3 本地部署

3.3.1 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 3, in top-level template code
  File "docs-2.0/reuse/source_install-nebula-graph-by-rpm-or-deb.md", line 8, in top-level template code
    {{ ent.ent_begin }}
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

3.3.2 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 3, in top-level template code
  File "docs-2.0/reuse/source_manage-service.md", line 3, in top-level template code
    {{ ent.ent_begin }}
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

3.3.3 步骤 3: 连接NebulaGraph

本文介绍如何使用原生命令行客户端 Nebula Console 连接 NebulaGraph。



首次连接到NebulaGraph后，必须先[注册 Storage 服务](#)，才能正常查询数据。

NebulaGraph支持多种类型的客户端，包括命令行客户端、可视化界面客户端和流行编程语言客户端。详情参见[客户端列表](#)。

前提条件

-NebulaGraph服务已启动。

- 运行 Nebula Console 的机器和运行NebulaGraph的服务器网络互通。
- Nebula Console 的版本兼容NebulaGraph的版本。



版本相同的 Nebula Console 和NebulaGraph兼容程度最高，版本不同的 Nebula Console 连接NebulaGraph时，可能会有兼容问题，或者无法连接并报错 `incompatible version between client and server`。

操作步骤

- 在 Nebula Console [下载页面](#)，确认需要的版本，单击 **Assets**。



建议选择最新版本。

- 在 **Assets** 区域找到机器运行所需的二进制文件，下载文件到机器上。

- (可选) 为方便使用，重命名文件为 `nebula-console`。



在 Windows 系统中，请重命名为 `nebula-console.exe`。

- 在运行 Nebula Console 的机器上执行如下命令，为用户授予 `nebula-console` 文件的执行权限。



Windows 系统请跳过此步骤。

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

- 在命令行界面中，切换工作目录至 `nebula-console` 文件所在目录。

6. 执行如下命令连接NebulaGraph。

- Linux 或 macOS

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

- Windows

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

参数说明如下。

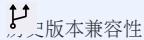
参数	说明
-h/-help	显示帮助菜单。
-addr/-address	设置要连接的 Graph 服务的 IP 地址。默认地址为 127.0.0.1。
-P/-port	设置要连接的 Graph 服务的端口。默认端口为 9669。
-u/-user	设置NebulaGraph账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 root）。
-p/-password	设置用户名对应的密码。未启用身份认证时，密码可以填写任意字符。
-t/-timeout	设置整数类型的连接超时时间。单位为毫秒， 默认值为 120。
-e/-eval	设置字符串类型的 nGQL 语句。连接成功后会执行一次该语句并返回结果，然后自动断开连接。
-f/-file	设置存储 nGQL 语句的文件的路径。连接成功后会执行该文件内的 nGQL 语句并返回结果，执行完毕后自动断开连接。
-enable_ssl	连接NebulaGraph时使用 SSL 加密。
-ssl_root_ca_path	指定 CA 证书的存储路径。
-ssl_cert_path	指定 CRT 证书的存储路径。
-ssl_private_key_path	指定私钥文件的存储路径。

更多参数参见[项目仓库](#)。

最后更新: May 4, 2023

3.3.4 注册 Storage 服务

首次连接到NebulaGraph后，需要先添加 **Storage** 主机，并确认主机都处于在线状态。



从NebulaGraph 3.0.0 版本开始，必须先使用 `ADD HOSTS` 添加主机，才能正常通过 **Storage** 服务读写数据。

前提条件

已连接NebulaGraph服务。

操作步骤

1. 添加 Storage 主机。

执行如下命令添加主机：

```
ADD HOSTS <ip>:<port> [<ip>:<port> ...];
```

示例：

```
nebula> ADD HOSTS 192.168.10.100:9779, 192.168.10.101:9779, 192.168.10.102:9779;
```



请确保添加的主机 IP 和配置文件 `nebula-storaged.conf` 中 `local_ip` 配置的 IP 一致，否则会导致添加 **Storage** 主机失败。关于配置文件的详情，参见[配置管理](#)。

2. 检查主机状态，确认全部在线。

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution | Version |
+-----+-----+-----+-----+-----+-----+
| "192.168.10.100" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" | "3.4.1" |
| "192.168.10.101" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" | "3.4.1" |
| "192.168.10.102" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" | "3.4.1" |
+-----+-----+-----+-----+-----+-----+
```

在返回结果的 **Status** 列，可以看到所有 **Storage** 主机都在线。

最后更新: May 4, 2023

3.3.5 步骤 4: 使用常用 nGQL (CRUD 命令)

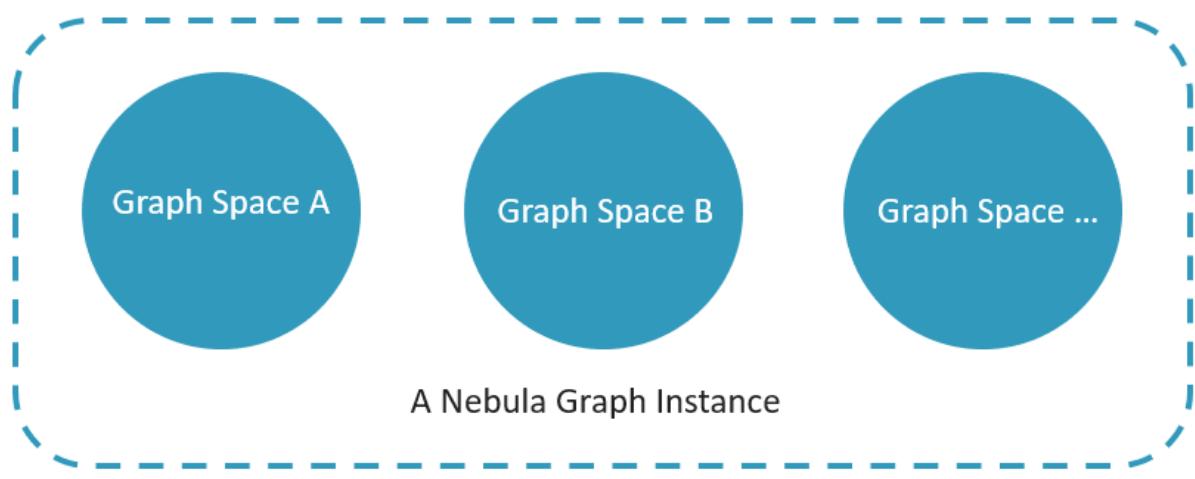
本文介绍NebulaGraph查询语言的基础语法，包括用于 Schema 创建和常用增删改查操作的语句。

如需了解更多语句的用法，参见 [nGQL 指南](#)。

使用说明

图空间和 SCHEMA

一个NebulaGraph实例由一个或多个图空间组成。每个图空间都是物理隔离的，用户可以在同一个实例中使用不同的图空间存储不同的数据集。

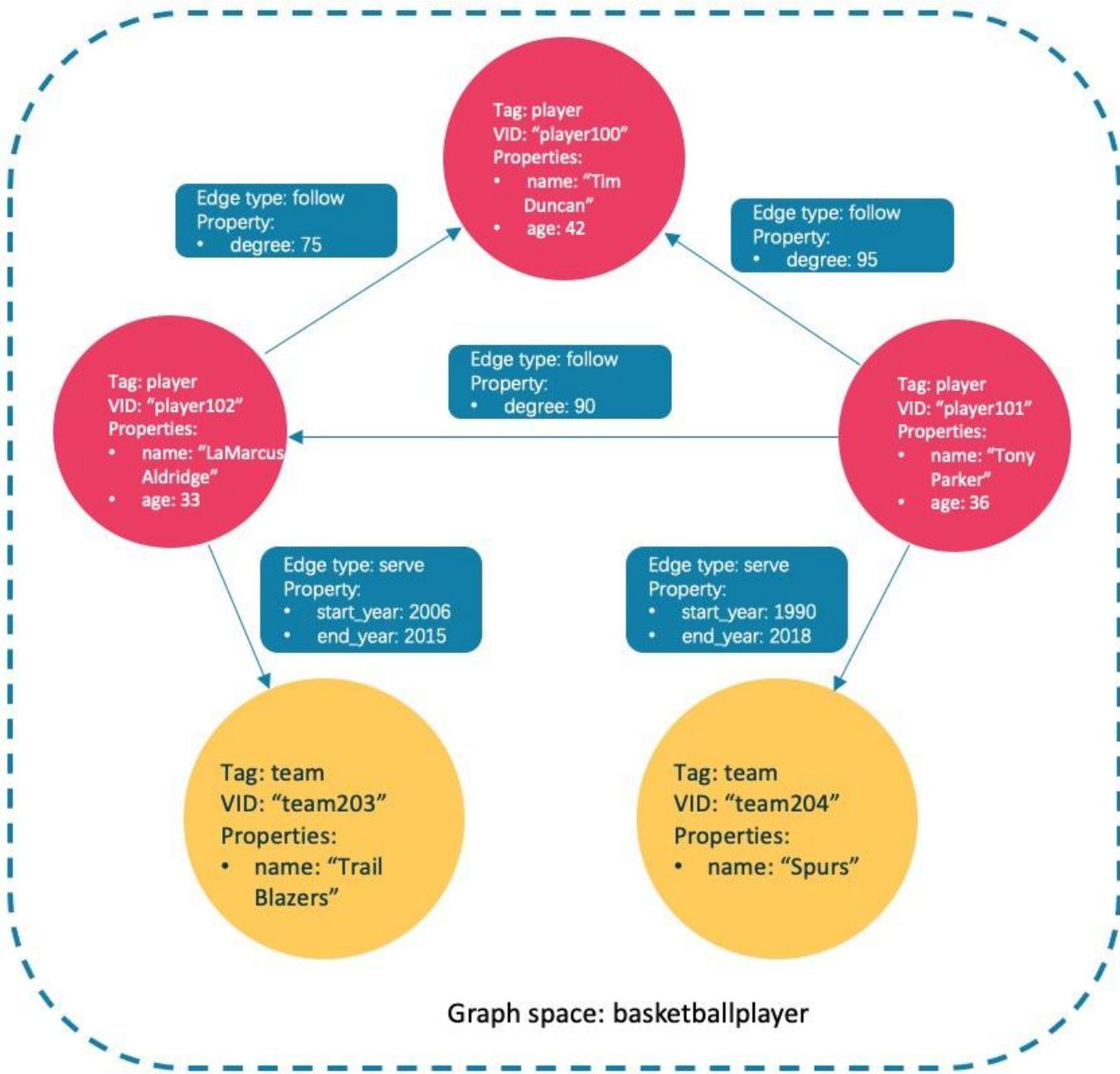


为了在图空间中插入数据，需要为图数据库定义一个 Schema。NebulaGraph的 Schema 是由如下几部分组成。

组成部分	说明
点 (Vertex)	表示现实世界中的实体。一个点可以有 0 到多个标签。
标签 (Tag)	点的类型，定义了一组描述点类型的属性。
边 (Edge)	表示两个点之间有方向的关系。
边类型 (Edge type)	边的类型，定义了一组描述边的类型的属性。

更多信息，请参见[数据结构](#)。

本文将使用下图的数据集演示基础操作的语法。



异步实现创建和修改

Caution

在NebulaGraph中，下列创建和修改操作是异步实现的。要在下一个心跳周期之后才能生效，否则访问会报错。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

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

Note

默认心跳周期是 10 秒。修改心跳周期参数 `heartbeat_interval_secs`，请参见[配置简介](#)。

第一步：创建和选择图空间

[nGQL 语法](#)

- 创建图空间

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

参数详情请参见 [CREATE SPACE](#)。

- 列出创建成功的图空间

```
nebula> SHOW SPACES;
```

- 选择数据库

```
USE <graph_space_name>;
```

示例

1. 执行如下语句创建名为 basketballplayer 的图空间。

```
nebula> CREATE SPACE basketballplayer(partition_num=15, replica_factor=1, vid_type=fixed_string(30));
```

Note

如果报错提示 [ERROR (-1005)]: Host not enough!，请检查是否已[添加 Storage 主机](#)。

2. 执行命令 SHOW HOSTS 检查分片的分布情况，确保平衡分布。

Host	Port	Status	Leader count	Leader distribution	Partition distribution	Version
------	------	--------	--------------	---------------------	------------------------	---------

"storaged0"	9779	"ONLINE"	5	"basketballplayer:5"	"basketballplayer:5"	"3.4.1"
"storaged1"	9779	"ONLINE"	5	"basketballplayer:5"	"basketballplayer:5"	"3.4.1"
"storaged2"	9779	"ONLINE"	5	"basketballplayer:5"	"basketballplayer:5"	"3.4.1"

如果 **Leader distribution** 分布不均匀, 请执行命令 `BALANCE LEADER` 重新分配。更多信息, 请参见 [Storage 负载均衡](#)。

3. 选择图空间 `basketballplayer`。

```
nebula[(none)]> USE basketballplayer;
```

用户可以执行命令 `SHOW SPACES` 查看创建的图空间。

```
nebula> SHOW SPACES;
+-----+
| Name |
+-----+
| "basketballplayer" |
+-----+
```

第二步: 创建 Tag 和 Edge type

NGQL 语法

```
CREATE {TAG | EDGE} [IF NOT EXISTS] {<tag_name> | <edge_type_name>}
(
    <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
    [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
)
[TTL_DURATION = <ttl_duration>]
[TTL_COL = <prop_name>]
[COMMENT = '<comment>'];
```

参数详情请参见 [CREATE TAG](#) 和 [CREATE EDGE](#)。

示例

创建 Tag: `player` 和 `team`, 以及 Edge type: `follow` 和 `serve`。说明如下表。

名称	类型	属性
player	Tag	name (string), age (int)
team	Tag	name (string)
follow	Edge type	degree (int)
serve	Edge type	start_year (int), end_year (int)

```
nebula> CREATE TAG player(name string, age int);
nebula> CREATE TAG team(name string);
nebula> CREATE EDGE follow(degree int);
nebula> CREATE EDGE serve(start_year int, end_year int);
```

第三步: 插入数据

用户可以使用 `INSERT` 语句, 基于现有的 Tag 插入点, 或者基于现有的 Edge type 插入边。

NGQL 语法

- 插入点

```
INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...]
VALUES <vid>; ([prop_value_list])

tag_props:
tag_name ([prop_name_list])

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

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

`vid` 是 Vertex ID 的缩写, `vid` 在一个图空间中是唯一的。参数详情请参见 [INSERT VERTEX](#)。

- 插入边

```
INSERT EDGE [IF NOT EXISTS] <edge_type> ( <prop_name_list> ) VALUES
<src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list>
[, <src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> ), ...];
<prop_name_list> ::= [<prop_name> [, <prop_name>] ...]
<prop_value_list> ::= [<prop_value> [, <prop_value>] ...]
```

参数详情请参见 [INSERT EDGE](#)。

示例

- 插入代表球员和球队的点。

```
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42);
nebula> INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36);
nebula> INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);
nebula> INSERT VERTEX team(name) VALUES "team203":("Trail Blazers"), "team204":("Spurs");
```

- 插入代表球员和球队之间关系的边。

```
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player100":(95);
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player102":(90);
nebula> INSERT EDGE follow(degree) VALUES "player102" -> "player100":(75);
nebula> INSERT EDGE serve(start_year, end_year) VALUES "player101" -> "team204":(1999, 2018), "player102" -> "team203":(2006, 2015);
```

第四步：查询数据

- GO** 语句可以根据指定的条件遍历数据库。**GO** 语句从一个或多个点开始，沿着一条或多条边遍历，返回 **YIELD** 子句中指定的信息。
- FETCH** 语句可以获得点或边的属性。
- LOOKUP** 语句是基于索引的，和 **WHERE** 子句一起使用，查找符合特定条件的数据。
- MATCH** 语句是查询图数据最常用的，可以灵活的描述各种图模式，但是它依赖索引去匹配NebulaGraph中的数据模型，性能也需要调优。

NGQL 语法

- GO

```
GO [[<N> TO] <N> [STEP|STEPS]] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
YIELD [DISTINCT] <return_list>
[ { SAMPLE <sample_list> | <limit_by_list_clause> } ]
[ | GROUP BY {<col_name> | expression} | <position> ] YIELD <col_name>]
```

```
[] ORDER BY <expression> [{ASC | DESC}]
[] LIMIT [<offset>,<number_rows>];
```

- **FETCH**

- 查询 Tag 属性

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
<vid> [, vid ...]
YIELD <return_list> [AS <alias>];
```

- 查询边属性

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

- **LOOKUP**

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

<return_list>
<prop_name> [AS <col_alias>] [, <prop_name> [AS <prop_alias>] ...];
```

- **MATCH**

```
MATCH <pattern> [<clause_1>] RETURN <output> [<clause_2>];
```

GO 语句示例

- 从 VID 为 player101 的球员开始，沿着边 follow 找到连接的球员。

```
nebula> GO FROM "player101" OVER follow YIELD id($$);
+-----+
| id($$) |
+-----+
| "player100" |
```

```
| "player102" |
+-----+
```

- 从 VID 为 player101 的球员开始，沿着边 follow 查找年龄大于或等于 35 岁的球员，并返回他们的姓名和年龄，同时重命名对应的列。

```
nebula> GO FROM "player101" OVER follow WHERE properties($$).age >= 35 \
    YIELD properties($$).name AS Teammate, properties($$).age AS Age;
+-----+-----+
| Teammate | Age |
+-----+-----+
| "Tim Duncan" | 42 |
+-----+
```

子句/符号	说明
YIELD	指定该查询需要返回的值或结果。
\$\$	表示边的终点。
\	表示换行继续输入。

- 从 VID 为 player101 的球员开始，沿着边 follow 查找连接的球员，然后检索这些球员的球队。为了合并这两个查询请求，可以使用管道符或临时变量。

- 使用管道符

```
nebula> GO FROM "player101" OVER follow YIELD dst(edge) AS id | \
    GO FROM $-.id OVER serve YIELD properties($$).name AS Team, \
    properties($$).name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| "Trail Blazers" | "LaMarcus Aldridge" |
+-----+
```

子句/符号	说明
\$^	表示边的起点。
	组合多个查询的管道符，将前一个查询的结果集用于后一个查询。
\$-	表示管道符前面的查询输出的结果集。

- 使用临时变量



当复合语句作为一个整体提交给服务器时，其中的临时变量会在语句结束时被释放。

```
nebula> $var = GO FROM "player101" OVER follow YIELD dst(edge) AS id; \
    GO FROM $var.id OVER serve YIELD properties($$).name AS Team, \
    properties($$).name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| "Trail Blazers" | "LaMarcus Aldridge" |
+-----+
```

FETCH 语句示例

查询 VID 为 player100 的球员的属性。

```
nebula> FETCH PROP ON player "player100" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 42, name: "Tim Duncan"} |
+-----+
```

Note

`LOOKUP` 和 `MATCH` 的示例在下文的 [索引](#) 部分查看。

其他操作

修改点和边

用户可以使用 `UPDATE` 语句或 `UPSERT` 语句修改现有数据。

`UPSERT` 是 `UPDATE` 和 `INSERT` 的结合体。当使用 `UPSERT` 更新一个点或边，如果它不存在，数据库会自动插入一个新的点或边。

Note

每个 `partition` 内部，`UPSERT` 操作是一个串行操作，所以执行速度比执行 `INSERT` 或 `UPDATE` 慢很多。其仅在多个 `partition` 之间有并发。

nGQL 语法

- `UPDATE` 点

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

- `UPDATE` 边

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

- `UPSERT` 点或边

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

示例

- 用 `UPDATE` 修改 VID 为 `player100` 的球员的 `name` 属性，然后用 `FETCH` 语句检查结果。

```
nebula> UPDATE VERTEX "player100" SET player.name = "Tim";
nebula> FETCH PROP ON player "player100" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
```

```
| {age: 42, name: "Tim"} |
+-----+
|
```

- 用 `UPDATE` 修改某条边的 `degree` 属性，然后用 `FETCH` 检查结果。

```
nebula> UPDATE EDGE ON follow "player101" -> "player100" SET degree = 96;
nebula> FETCH PROP ON follow "player101" -> "player100" YIELD properties(edge);
+-----+
| properties(EDGE) |
+-----+
| {degree: 96}      |
+-----+
```

- 用 `INSERT` 插入一个 VID 为 `player111` 的点，然后用 `UPSERT` 更新它。

```
nebula> INSERT VERTEX player(name,age) VALUES "player111":("David West", 38);
nebula> UPSERT VERTEX "player111" SET player.name = "David", player.age = $^.player.age + 11 \
    WHEN $^.player.name == "David West" AND $^.player.age > 20 \
    YIELD $^.player.name AS Name, $^.player.age AS Age;
+-----+-----+
| Name   | Age  |
+-----+-----+
| "David" | 49   |
+-----+-----+
```

删除点和边

nGQL 语法

- 删除点

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

- 删除边

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

示例

- 删除点

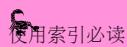
```
nebula> DELETE VERTEX "player111", "team203";
```

- 删除边

```
nebula> DELETE EDGE follow "player101" -> "team204";
```

使用索引

用户可以通过 `CREATE INDEX` 语句为 `Tag` 和 `Edge type` 增加索引。



使用索引必读

`MATCH` 和 `LOOKUP` 语句的执行都依赖索引，但是索引会导致写性能大幅降低。请不要随意在生产环境中使用索引，除非很清楚使用索引对业务的影响。

必须为“已写入但未构建索引”的数据重建索引，否则无法在 `MATCH` 和 `LOOKUP` 语句中返回这些数据。参见[重建索引](#)。

nGQL 语法

- 创建索引

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

- 重建索引

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

Note

为没有指定长度的变量属性创建索引时，需要指定索引长度。在 utf-8 编码中，一个中文字符占 3 字节，请根据变量属性长度设置合适的索引长度。例如 10 个中文字符，索引长度需要为 30。详情请参见[创建索引](#)。

基于索引的 **LOOKUP** 和 **MATCH** 示例

确保 LOOKUP 或 MATCH 有一个索引可用。如果没有，请先创建索引。

找到 Tag 为 player 的点的信息，它的 name 属性值为 Tony Parker。

```
// 为 name 属性创建索引 player_index_1。
nebula> CREATE TAG INDEX IF NOT EXISTS player_index_1 ON player(name(20));

// 重建索引确保能对已存在数据生效。
nebula> REBUILD TAG INDEX player_index_1
+-----+
| New Job Id |
+-----+
| 31          |
+-----+

// 使用 LOOKUP 语句检索点的属性。
nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
    YIELD properties(vertex).name AS name, properties(vertex).age AS age;
+-----+-----+
| name      | age   |
+-----+-----+
| "Tony Parker" | 36   |
+-----+-----+


// 使用 MATCH 语句检索点的属性。
nebula> MATCH (v:player{name:"Tony Parker"}) RETURN v;
+-----+
| v      |
+-----+
| ("player101" :player{age: 36, name: "Tony Parker"}) |
+-----+
```

最后更新: May 4, 2023

3.4 nGQL 命令汇总

3.4.1 函数

• 数学函数

函数	说明
double abs(double x)	返回 x 的绝对值。
double floor(double x)	返回小于或等于 x 的最大整数。
double ceil(double x)	返回大于或等于 x 的最小整数。
double round(double x)	返回离 x 最近的整数值，如果 x 恰好在中间，则返回离 0 较远的整数。
double sqrt(double x)	返回 x 的平方根。
double cbrt(double x)	返回 x 的立方根。
double hypot(double x, double y)	返回直角三角形（直角边长为 x 和 y）的斜边长。
double pow(double x, double y)	返回 x^y 的值。
double exp(double x)	返回 e^x 的值。
double exp2(double x)	返回 2^x 的值。
double log(double x)	返回以自然数 e 为底 x 的对数。
double log2(double x)	返回以 2 为底 x 的对数。
double log10(double x)	返回以 10 为底 x 的对数。
double sin(double x)	返回 x 的正弦值。
double asin(double x)	返回 x 的反正弦值。
double cos(double x)	返回 x 的余弦值。
double acos(double x)	返回 x 的反余弦值。
double tan(double x)	返回 x 的正切值。
double atan(double x)	返回 x 的反正切值。
double rand()	返回 [0,1) 内的随机浮点数。
int rand32(int min, int max)	返回 [min, max] 内的一个随机 32 位整数。 用户可以只传入一个参数，该参数会判定为 max，此时 min 默认为 0。如果不传入参数，此时会从带符号的 32 位 int 范围内随机返回。
int rand64(int min, int max)	返回 [min, max] 内的一个随机 64 位整数。 用户可以只传入一个参数，该参数会判定为 max，此时 min 默认为 0。如果不传入参数，此时会从带符号的 64 位 int 范围内随机返回。
bit_and()	逐位做 AND 操作。
bit_or()	逐位做 OR 操作。
bit_xor()	逐位做 XOR 操作。
int size()	返回列表或映射中元素的数量，或字符串的长度。
int range(int start, int end, int step)	返回 [start,end] 中指定步长的值组成的列表。步长 step 默认为 1。
int sign(double x)	返回 x 的正负号。如果 x 为 0，则返回 0。如果 x 为负数，则返回 -1。如果 x 为正数，则返回 1。
double e()	返回自然对数的底 e (2.718281828459045)。
double pi()	返回数学常数π (3.141592653589793)。
double radians()	将角度转换为弧度。radians(180) 返回 3.141592653589793。

• 聚合函数

函数	说明
<code>avg()</code>	返回参数的平均值。
<code>count()</code>	语法: <code>count({expr *})</code> 。 <code>count()</code> 返回总行数（包括 <code>NULL</code> ）。 <code>count(expr)</code> 返回满足表达式的非空值的总数。 <code>count()</code> 和 <code>size()</code> 是不同的。
<code>max()</code>	返回参数的最大值。
<code>min()</code>	返回参数的最小值。
<code>collect()</code>	<code>collect()</code> 函数返回一个符合表达式返回结果的列表。该函数可以将多条记录或值合并进一个列表，实现数据聚合。
<code>std()</code>	返回参数的总体标准差。
<code>sum()</code>	返回参数的和。

• 字符串函数

函数	说明
int strcasecmp(string a, string b)	比较两个字符串（不区分大小写）。当 <code>a=b</code> 时，返回 0，当 <code>a>b</code> 是，返回大于 0 的数，当 <code>a<b</code> 时，返回小于 0 的数。
string lower(string a)	返回小写形式的字符串。
string toLower(string a)	和 <code>lower()</code> 相同。
string upper(string a)	返回大写形式的字符串。
string toUpper(string a)	和 <code>upper()</code> 相同。
int length(a)	返回给定字符串的长度或路径的长度，单位分别是字节和跳数。
string trim(string a)	删除字符串头部和尾部的空格。
string ltrim(string a)	删除字符串头部的空格。
string rtrim(string a)	删除字符串尾部的空格。
string left(string a, int count)	返回字符串左侧 <code>count</code> 个字符组成的子字符串。如果 <code>count</code> 超过字符串 <code>a</code> 的长度，则返回字符串 <code>a</code> 。
string right(string a, int count)	返回字符串右侧 <code>count</code> 个字符组成的子字符串。如果 <code>count</code> 超过字符串 <code>a</code> 的长度，则返回字符串 <code>a</code> 。
string lpad(string a, int size, string letters)	在字符串 <code>a</code> 的左侧填充 <code>letters</code> 字符串，并返回 <code>size</code> 长度的字符串。
string rpad(string a, int size, string letters)	在字符串 <code>a</code> 的右侧填充 <code>letters</code> 字符串，并返回 <code>size</code> 长度的字符串。
string substr(string a, int pos, int count)	从字符串 <code>a</code> 的指定位置 <code>pos</code> 开始（不包括 <code>pos</code> 位置的字符），提取右侧的 <code>count</code> 个字符，组成新的字符串并返回。
string substring(string a, int pos, int count)	和 <code>substr()</code> 相同。
string reverse(string)	逆序返回字符串。
string replace(string a, string b, string c)	将字符串 <code>a</code> 中的子字符串 <code>b</code> 替换为字符串 <code>c</code> 。
list split(string a, string b)	在子字符串 <code>b</code> 处拆分字符串 <code>a</code> ，返回一个字符串列表。
concat()	<code>concat()</code> 函数至少需要两个或以上字符串参数，并将所有参数连接成一个字符串。 语法： <code>concat(string1,string2,...)</code>
concat_ws()	<code>concat_ws()</code> 函数将两个或以上字符串参数与预定义的分隔符（ <code>separator</code> ）相连接。
extract()	<code>extract()</code> 从指定字符串中提取符合正则表达式的子字符串。
json_extract()	<code>json_extract()</code> 将指定 JSON 字符串转换为 <code>map</code> 类型。

- 日期时间函数

函数	说明
int now()	根据当前系统返回当前时间戳。
timestamp timestamp()	根据当前系统返回当前时间戳。
date date()	根据当前系统返回当前日期（UTC 时间）。
time time()	根据当前系统返回当前时间（UTC 时间）。
datetime datetime()	根据当前系统返回当前日期和时间（UTC 时间）。

- Schema 相关函数

- 原生 nGQL 语句适用

函数	说明
id(vertex)	返回点 ID。数据类型和点 ID 的类型保持一致。
map properties(vertex)	返回点的所有属性。
map properties(edge)	返回边的所有属性。
string type(edge)	返回边的 Edge type。
src(edge)	返回边的起始点 ID。数据类型和点 ID 的类型保持一致。
dst(edge)	返回边的目的点 ID。数据类型和点 ID 的类型保持一致。
int rank(edge)	返回边的 rank。
vertex	返回点的信息。包括点 ID、Tag、属性和值。
edge	返回边的信息。包括 Edge type、起始点 ID、目的点 ID、rank、属性和值。
vertices	返回子图中的点的信息。详情参见 GET SUBGRAPH 。
edges	返回子图中的边的信息。详情参见 GET SUBGRAPH 。
path	返回路径信息。详情参见 FIND PATH 。

- openCypher 兼容语句适用

函数	说明
id(<vertex>)	返回点 ID。数据类型和点 ID 的类型保持一致。
list tags(<vertex>)	返回点的 Tag，与 labels() 作用相同。
list labels(<vertex>)	返回点的 Tag，与 tags() 作用相同，用于兼容 openCypher 语法。
map properties(<vertex_or_edge>)	返回点或边的所有属性。
string type(<edge>)	返回边的 Edge type。
src(<edge>)	返回边的起始点 ID。数据类型和点 ID 的类型保持一致。
dst(<edge>)	返回边的目的点 ID。数据类型和点 ID 的类型保持一致。
vertex startNode(<path>)	获取一条边或一条路径并返回它的起始点 ID。
string endNode(<path>)	获取一条边或一条路径并返回它的目的点 ID。
int rank(<edge>)	返回边的 rank。

• 列表函数

函数	说明
keys(expr)	返回一个列表，包含字符串形式的点、边或映射的所有属性。
labels(vertex)	返回点的 Tag 列表。
nodes(path)	返回路径中所有点的列表。
range(start, end [, step])	返回 [start,end] 范围内固定步长的列表，默认步长 step 为 1。
relationships(path)	返回路径中所有关系的列表。
reverse(list)	返回将原列表逆序排列的新列表。
tail(list)	返回不包含原列表第一个元素的新列表。
head(list)	返回列表的第一个元素。
last(list)	返回列表的最后一个元素。
reduce()	将表达式逐个应用于列表中的元素，然后和累加器中的当前结果累加，最后返回完整结果。

• 类型转换函数

函数	说明
bool toBoolean()	将字符串转换为布尔。
float toFloat()	将整数或字符串转换为浮点数。
string toString()	将任意数据类型转换为字符串类型。
int toInteger()	将浮点或字符串转换为整数。
set toSet()	将列表或集合转换为集合。
int hash()	hash() 函数返回参数的哈希值。其参数可以是数字、字符串、列表、布尔值、NULL 等类型的值，或者计算结果为这些类型的表达式。

• 谓词函数

谓词函数只返回 `true` 或 `false`，通常用于 `WHERE` 子句中。

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

函数	说明
exists()	如果指定的属性在点、边或映射中存在，则返回 <code>true</code> ，否则返回 <code>false</code> 。
any()	如果指定的谓词适用于列表中的至少一个元素，则返回 <code>true</code> ，否则返回 <code>false</code> 。
all()	如果指定的谓词适用于列表中的每个元素，则返回 <code>true</code> ，否则返回 <code>false</code> 。
none()	如果指定的谓词不适用于列表中的任何一个元素，则返回 <code>true</code> ，否则返回 <code>false</code> 。
single()	如果指定的谓词适用于列表中的唯一一个元素，则返回 <code>true</code> ，否则返回 <code>false</code> 。

- 条件表达式函数

函数	说明
CASE	使用条件来过滤 nGQL 查询语句的结果，常用于 YIELD 和 RETURN 子句中。 CASE 表达式会遍历所有条件，并在满足第一个条件时停止读取后续条件，然后返回结果。 如果不满足任何条件，将通过 ELSE 子句返回结果。如果没有 ELSE 子句且不满足任何条件，则返回 NULL。
coalesce()	返回所有表达式中第一个非空元素。

3.4.2 通用查询语句

- MATCH

```
MATCH <pattern> [<clause_1>] RETURN <output> [<clause_2>];
```

模式	示例	说明
匹配点	(v)	用户可以在一对括号中使用自定义变量来表示模式中的点。例如 (v)。
匹配 Tag	MATCH (v:player) RETURN v	用户可以在点的右侧用 :<tag_name> 表示模式中的 Tag。
匹配多 Tag	MATCH (v:player:team) RETURN v	用户可以用英文冒号 (:) 匹配多 Tag 的点。
匹配点的属性	MATCH (v:player{name:"Tim Duncan"}) RETURN v MATCH (v) WITH v, properties(v) as props, keys(properties(v)) as kk WHERE [i in kk where props[i] == "Tim Duncan"] RETURN v	用户可以在 Tag 的右侧用 {<prop_name>: <prop_value>} 表示模式中点的属性；或者不指定 Tag 直接匹配点的属性。
匹配单点 ID	MATCH (v) WHERE id(v) == 'player101' RETURN v	用户可以使用点 ID 去匹配点。id() 函数可以检索点的 ID。
匹配多点 ID	MATCH (v:player { name: 'Tim Duncan' })--(v2) WHERE id(v2) IN ["player101", "player102"] RETURN v2	要匹配多个点的 ID，可以用 WHERE id(v) IN [vid_list]。
匹配连接的点	MATCH (v:player{name:"Tim Duncan"})--(v2) RETURN v2.player.name AS Name	用户可以使用 -- 符号表示两个方向的边，并匹配这些边连接的点。用户可以在 -- 符号上增加 < 或 > 符号指定边的方向。
匹配路径	MATCH p=(v:player{name:"Tim Duncan"})-->(v2) RETURN p	连接起来的点和边构成了路径。用户可以使用自定义变量命名路径。
匹配边	MATCH (v:player{name:"Tim Duncan"})-[e]-(v2) RETURN e MATCH ()-<[e]-() RETURN e	除了用 -- 、 --> 、 <> 表示未命名的边之外，用户还可以在方括号中使用自定义变量命名边。例如 -[e]-。
匹配 Edge type	MATCH ()-[e:follow]->() RETURN e	和点一样，用户可以用 :<edge_type> 表示模式中的 Edge type，例如 -[e:follow]-。
匹配边的属性	MATCH (v:player{name:"Tim Duncan"})- [e:follow{degree:95}]->(v2) RETURN e MATCH ()-[e]->() WITH e, properties(e) as props, keys(properties(e)) as kk WHERE [i in kk where props[i] == 90] RETURN e	用户可以用 {<prop_name>: <prop_value>} 表示模式中 Edge type 的属性，例如 [e:follow{likeness:95}]；或者不指定 Edge type 直接匹配边的属性。
匹配多个 Edge type	MATCH (v:player{name:"Tim Duncan"})-[e:follow :serve]->(v2) RETURN e	使用 可以匹配多个 Edge type，例如 [e:follow :serve]。第一个 Edge type 前的英文冒号 (:) 不可省略，后续 Edge type 前的英文冒号可以省略，例如 [e:follow serve]。
匹配多条边	MATCH (v:player{name:"Tim Duncan"})-[]->(v2)< [e:serve]->(v3) RETURN v2, v3	用户可以扩展模式，匹配路径中的多条边。
匹配定长路径	MATCH p=(v:player{name:"Tim Duncan"})- [e:follow*2]->(v2) RETURN DISTINCT v2 AS Friends	用户可以在模式中使用 :<edge_type>*<hop> 匹配定长路径。hop 必须是一个非负整数。e 的数据类型是列表。
匹配变长路径	MATCH p=(v:player{name:"Tim Duncan"})- [e:follow*1..3]->(v2) RETURN v2 AS Friends	minHop：可选项。表示路径的最小长度。minHop 必须是一个非负整数，默认值为 1。 maxHop：可选项。表示路径的最大长度。maxHop 必须是一个非负整数，默认为无穷大。e 的数据类型是列表。
匹配多个 Edge type 的变长路径	MATCH p=(v:player{name:"Tim Duncan"})-[e:follow serve*2]->(v2) RETURN DISTINCT v2	用户可以在变长或定长模式中指定多个 Edge type。hop、minHop 和 maxHop 对所有 Edge type 都生效。e 的数据类型是列表。
检索点或边的信息	MATCH (v:player{name:"Tim Duncan"}) RETURN v MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) RETURN e	使用 RETURN {<vertex_name> <edge_name>} 检索点或边的所有信息。
检索点 ID		使用 id() 函数检索点 ID。

模式	示例	说明
	<code>MATCH (v:player{name:"Tim Duncan"}) RETURN id(v)</code>	
检索 Tag	<code>MATCH (v:player{name:"Tim Duncan"}) RETURN labels(v)</code>	使用 <code>labels()</code> 函数检索点上的 Tag 列表。 检索列表 <code>labels(v)</code> 中的第 N 个元素，可以使用 <code>labels(v)[n-1]</code> 。
检索点或边的单个属性	<code>MATCH (v:player{name:"Tim Duncan"}) RETURN v.player.age</code>	使用 <code>RETURN <vertex_name> <edge_name>.property</code> 检索单个属性。 使用 AS 设置属性的别名。
检索点或边的所有属性	<code>MATCH p=(v:player{name:"Tim Duncan"})-[]-(v2) RETURN properties(v2)</code>	使用 <code>properties()</code> 函数检索点或边的所有属性。
检索 Edge type	<code>MATCH p=(v:player{name:"Tim Duncan"})-[e]-() RETURN DISTINCT type(e)</code>	使用 <code>type()</code> 函数检索匹配的 Edge type。
检索路径	<code>MATCH p=(v:player{name:"Tim Duncan"})-[*3]-() RETURN p</code>	使用 <code>RETURN <path_name></code> 检索匹配路径的所有信息。
检索路径中的点	<code>MATCH p=(v:player{name:"Tim Duncan"})-[]-(v2) RETURN nodes(p)</code>	使用 <code>nodes()</code> 函数检索路径中的所有点。
检索路径中的边	<code>MATCH p=(v:player{name:"Tim Duncan"})-[]-(v2) RETURN relationships(p)</code>	使用 <code>relationships()</code> 函数检索路径中的所有边。
检索路径长度	<code>MATCH p=(v:player{name:"Tim Duncan"})-[*..2]->(v2) RETURN p AS Paths, length(p) AS Length</code>	使用 <code>length()</code> 函数检索路径的长度。

• OPTIONAL MATCH

模式	示例	说明
作为 MATCH 语句的可选项去匹配图数据库中的模式	<code>MATCH (m)-[]-(n) WHERE id(m)=="player100" OPTIONAL MATCH (n)-[]-(l) RETURN id(m),id(n),id(l)</code>	如果图数据库中没有对应的模式，对应的列返回 NULL。

• LOOKUP

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

模式	示例	说明
检索点	<code>LOOKUP ON player WHERE player.name == "Tony Parker" YIELD player.name AS name, player.age AS age</code>	返回 Tag 为 player 且 name 为 Tony Parker 的点。
检索边	<code>LOOKUP ON follow WHERE follow.degree == 90 YIELD follow.degree</code>	返回 Edge type 为 follow 且 degree 为 90 的边。
通过 Tag 列出所有点	<code>LOOKUP ON player YIELD properties(vertex),id(vertex)</code>	查找所有 Tag 为 player 的点 VID。
通过 Edge type 列出边	<code>LOOKUP ON Like YIELD edge AS e</code>	查找 Edge type 为 like 的所有边的信息。
统计点	<code>LOOKUP ON player YIELD id(vertex) YIELD COUNT(*) AS Player_Count</code>	统计 Tag 为 player 的点。
统计边	<code>LOOKUP ON Like YIELD id(vertex) YIELD COUNT(*) AS Like_Count</code>	统计 Edge type 为 like 的边。

• GO

```
GO [[<M> TO] <N> {STEP|STEPS} ] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
YIELD [DISTINCT] <return_list>
[ {SAMPLE <sample_list> | LIMIT <limit_list>} ]
[] GROUP BY {col_name | expr | position} YIELD <col_name>
[] ORDER BY <expression> [{ASC | DESC}]]
[] LIMIT [<offset_value>,] <number_rows>]
```

示例

```
GO FROM "player102" OVER serve YIELD dst(edge)
```

说明

返回 player102 所属队伍。

```
GO 2 STEPS FROM "player102" OVER follow YIELD dst(edge)
```

返回距离 player102 两跳的朋友。

```
GO FROM "player100", "player102" OVER serve WHERE properties(edge).start_year > 1995 YIELD DISTINCT
properties($$).name AS team_name, properties(edge).start_year AS start_year, properties($^).name AS player_name
```

添加过滤条件。

```
GO FROM "player100" OVER follow, serve YIELD properties(edge).degree, properties(edge).start_year
```

遍历多个 Edge type。属性没有值时，会显示 NULL。

```
GO FROM "player100" OVER follow REVERSELY YIELD src(edge) AS destination
```

返回 player100 入方向的邻居点。

```
GO FROM "player100" OVER follow REVERSELY YIELD src(edge) AS id | GO FROM $-.id OVER serve WHERE
properties($^).age > 20 YIELD properties($^).name AS FriendOf, properties($$).name AS Team
```

查询 player100 的朋友和朋友所属队伍。

```
GO FROM "player102" OVER follow YIELD dst(edge) AS both
```

返回 player102 所有邻居点。

```
GO 2 STEPS FROM "player100" OVER follow YIELD src(edge) AS src, dst(edge) AS dst, properties($$).age AS age |
GROUP BY $-.dst YIELD $-.dst AS dst, collect_set($-.src) AS src, collect($-.age) AS age
```

根据年龄分组。

• FETCH

- 获取点的属性值

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
<vid> [, vid ...]
YIELD <return_list> [AS <alias>]
```

示例

```
FETCH PROP ON player "player100" YIELD properties(vertex)
```

说明

在 FETCH 语句中指定 Tag 获取对应点的属性值。

```
FETCH PROP ON player "player100" YIELD player.name AS name
```

使用 YIELD 子句指定返回的属性。

```
FETCH PROP ON player "player101", "player102", "player103" YIELD
properties(vertex)
```

指定多个点 ID 获取多个点的属性值，点之间用英文逗号（,）分隔。

```
FETCH PROP ON player, t1 "player100", "player103" YIELD
properties(vertex)
```

在 FETCH 语句中指定多个 Tag 获取属性值。Tag 之间用英文逗号（,）分隔。

```
FETCH PROP ON * "player100", "player106", "team200" YIELD
properties(vertex)
```

在 FETCH 语句中使用 * 获取当前图空间所有标签里，点的属性值。

- 获取边的属性值

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

示例

```
FETCH PROP ON serve "player100" -> "team204" YIELD properties(edge)
```

说明

获取连接 player100 和 team204 的边 serve 的所有属性值。

```
FETCH PROP ON serve "player100" -> "team204" YIELD serve.start_year
```

使用 YIELD 子句指定返回的属性。

```
FETCH PROP ON serve "player100" -> "team204", "player133" -> "team202" YIELD
properties(edge)
```

指定多个边模式 (<src_vid> -> <dst_vid>[@<rank>]) 获取多个边的属性值。模式之间用英文逗号（,）分隔。

```
FETCH PROP ON serve "player100" -> "team204"@1 YIELD properties(edge)
```

要获取 rank 不为 0 的边，请在 FETCH 语句中设置 rank。

```
GO FROM "player101" OVER follow YIELD follow._src AS s, follow._dst AS d |
FETCH PROP ON follow $-.s -> $-.d YIELD follow.degree
```

返回从点 player101 开始的 follow 边的 degree 值。

```
$var = GO FROM "player101" OVER follow YIELD follow._src AS s, follow._dst AS
d; FETCH PROP ON follow $var.s -> $var.d YIELD follow.degree
```

自定义变量构建查询。

- SHOW

语句	语法	示例	说明
SHOW CHARSET	SHOW CHARSET	SHOW CHARSET	显示当前的字符集。
SHOW COLLATION	SHOW COLLATION	SHOW COLLATION	显示当前的排序规则。
SHOW CREATE SPACE	SHOW CREATE SPACE <space_name>	SHOW CREATE SPACE basketballplayer	显示指定图空间的创建语句。
SHOW CREATE TAG/EDGE	SHOW CREATE {TAG <tag_name> EDGE <edge_name>}	SHOW CREATE TAG player	显示指定 Tag/Edge type 的基本信息。
SHOW HOSTS	SHOW HOSTS [GRAPH STORAGE META]	SHOW HOSTS SHOW HOSTS GRAPH	显示 Graph、Storage、Meta 服务主机信息、版本信息。
SHOW INDEX STATUS	SHOW {TAG EDGE} INDEX STATUS	SHOW TAG INDEX STATUS	重建原生索引的作业状态，以便确定重建索引是否成功。
SHOW INDEXES	SHOW {TAG EDGE} INDEXES	SHOW TAG INDEXES	列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。
SHOW PARTS	SHOW PARTS [<part_id>]	SHOW PARTS	显示图空间中指定分片或所有分片的信息。
SHOW ROLES	SHOW ROLES IN <space_name>	SHOW ROLES in basketballplayer	显示分配给用户的角色信息。
SHOW SNAPSHOTS	SHOW SNAPSHOTS	SHOW SNAPSHOTS	显示所有快照信息。
SHOW SPACES	SHOW SPACES	SHOW SPACES	显示现存的图空间。
SHOW STATS	SHOW STATS	SHOW STATS	显示最近 STATS 作业收集的图空间统计信息。
SHOW TAGS/EDGES	SHOW TAGS EDGES	SHOW TAGS、SHOW EDGES	显示当前图空间内的所有 Tag/Edge type。
SHOW USERS	SHOW USERS	SHOW USERS	显示用户信息。
SHOW SESSIONS	SHOW SESSIONS	SHOW SESSIONS	显示所有会话信息。
SHOW SESSIONS	SHOW SESSION <Session_Id>	SHOW SESSION 1623304491050858	指定会话 ID 进行查看。
SHOW QUERIES	SHOW [ALL] QUERIES	SHOW QUERIES	查看当前 Session 中正在执行的查询请求信息。
SHOW META LEADER	SHOW META LEADER	SHOW META LEADER	显示当前 Meta 集群的 leader 信息。

3.4.3 子句和选项

子句	语法	示例	说明
GROUP BY	GROUP BY <var> YIELD <var>, <aggregation_function(var)>	GO FROM "player100" OVER follow BIDIRECT YIELD \$\$.player.name as Name GROUP BY \$-.Name YIELD \$-.Name as Player, count(*) AS Name_Count	查找所有连接到 player100 的点，并根据他们的姓名进行分组，返回姓名的出现次数。
LIMIT	YIELD <var> [LIMIT [<offset_value>,<number_rows>]	GO FROM "player100" OVER follow REVERSELY YIELD \$\$.player.name AS Friend, \$\$.player.age AS Age ORDER BY \$-.Age, \$-.Friend LIMIT 1, 3	从排序结果中返回第 2 行开始的 3 行数据。
SKIP	RETURN <var> [SKIP <offset>] [LIMIT <number_rows>]	MATCH (v:player{name:"Tim Duncan"}) --> (v2) RETURN v2.player.name AS Name, v2.player.age AS Age ORDER BY Age DESC SKIP 1	用户可以单独使用 SKIP <offset> 设置偏移量，后面不需要添加 LIMIT <number_rows>。
SAMPLE	<go_statement> SAMPLE <sample_list>;	GO 3 STEPS FROM "player100" OVER * YIELD properties(\$\$).name AS NAME, properties(\$\$).age AS Age SAMPLE [1,2,3];	在结果集中均匀取样并返回指定数量的数据。
ORDER BY	<YIELD clause> ORDER BY <expression> [ASC DESC] [, <expression> [ASC DESC] ...]	FETCH PROP ON player "player100", "player101", "player102", "player103" YIELD player.age AS age, player.name AS name ORDER BY \$-.age ASC, \$-.name DESC	ORDER BY 子句指定输出结果的排序规则。
RETURN	RETURN {<vertex_name> <edge_name> <vertex_name>.<property> <edge_name>.<property> ...}	MATCH (v:player) RETURN v.player.name, v.player.age LIMIT 3	返回点的属性为 name 和 age 的前三行值。
TTL	CREATE TAG <tag_name>(<property_name_1> <property_value_1>, <property_name_2> <property_value_2>, ...) ttl_duration=<value_int>, ttl_col = <property_name>	CREATE TAG t2(a int, b int, c string) ttl_duration= 100, ttl_col = "a"	创建 Tag 并设置 TTL 选项。
WHERE	WHERE {<vertex edge_alias>.<property_name> > = <...> <value>...}	MATCH (v:player) WHERE v.player.name == "Tim Duncan" XOR (v.player.age < 30 AND v.player.name == "Yao Ming") OR NOT (v.player.name == "Yao Ming" OR v.player.name == "Tim Duncan") RETURN v.player.name, v.player.age	WHERE 子句可以根据条件过滤输出结果，通常用于 GO 和 LOOKUP 语句， MATCH 和 WITH 语句。
YIELD	YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...] [WHERE <conditions>];	GO FROM "player100" OVER follow YIELD dst(edge) AS ID FETCH PROP ON player \$-.ID YIELD player.age AS Age YIELD AVG(\$-.Age) as Avg_Age, count(*)as Num_friends	查找 player100 关注的 player，并计算他们的平均年龄。
WITH	MATCH \$expressions WITH {nodes() labels() ...}	MATCH p=(v:player{name:"Tim Duncan"})--() WITH nodes(p) AS n UNWIND n AS n1 RETURN DISTINCT n1	WITH 子句可以获取并处理查询前半部分的结果，并将处理结果作为输入传递给查询的后半部分。
UNWIND	UNWIND <list> AS <alias> <RETURN clause>	UNWIND [1,2,3] AS n RETURN n	拆分列表。

3.4.4 图空间语句

语句	语法	示例	说明
CREATE SPACE	<code>CREATE SPACE [IF NOT EXISTS] <graph_space_name> ([partition_num = <partition_number>,] [replica_factor = <replica_number>,] vid_type = {FIXED_STRING(<N>) INT[64]}) [COMMENT = '<comment>']</code>	<code>CREATE SPACE my_space_1 (vid_type=FIXED_STRING(30))</code>	创建一个新的图空间。
CREATE SPACE	<code>CREATE SPACE <new_graph_space_name> AS <old_graph_space_name></code>	<code>CREATE SPACE my_space_4 as my_space_3</code>	克隆现有图空间的 Schema。
USE	<code>USE <graph_space_name></code>	<code>USE space1</code>	指定一个图空间，或切换到另一个图空间，将其作为后续查询的工作空间。
SHOW SPACES	<code>SHOW SPACES</code>	<code>SHOW SPACES</code>	列出NebulaGraph示例中的所有图空间。
DESCRIBE SPACE	<code>DESC[RIBE] SPACE <graph_space_name></code>	<code>DESCRIBE SPACE basketballplayer</code>	显示指定图空间的信息。
CLEAR SPACE	<code>CLEAR SPACE [IF EXISTS] <graph_space_name></code>	清空图空间中的点和边，但不会删除图空间本身以及其中的 Schema 信息。	
DROP SPACE	<code>DROP SPACE [IF EXISTS] <graph_space_name></code>	<code>DROP SPACE basketballplayer</code>	删除指定图空间的所有内容。

3.4.5 TAG 语句

语句	语法	示例	说明
CREATE TAG	<code>CREATE TAG [IF NOT EXISTS] <tag_name> (<prop_name> <data_type> [NULL NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>'] [{, <prop_name> <data_type> [NULL NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]) [TTL_DURATION = <ttdl_duration>] [TTL_COL = <prop_name>] [COMMENT = '<comment>']</code>	<code>CREATE TAG woman(name string, age int, married bool, salary double, create_time timestamp) TTL_DURATION = 100, TTL_COL = "create_time"</code>	通过指定名称创建一个 Tag。
DROP TAG	<code>DROP TAG [IF EXISTS] <tag_name></code>	<code>DROP TAG test;</code>	删除当前工作空间内所有点上的指定 Tag。
ALTER TAG	<code>ALTER TAG <tag_name> <alter_definition> [, alter_definition] ... [ttl_definition [, ttl_definition] ...] [COMMENT = '<comment>']</code>	<code>ALTER TAG t1 ADD (p3 int, p4 string)</code>	修改 Tag 的结构。例如增删属性、修改数据类型，也可以为属性设置、修改 TTL (Time-To-Live)。
SHOW TAGS	<code>SHOW TAGS</code>	<code>SHOW TAGS</code>	显示当前图空间内的所有 Tag 名称。
DESCRIBE TAG	<code>DESC[RIBE] TAG <tag_name></code>	<code>DESCRIBE TAG player</code>	查看指定 Tag 的详细信息，例如字段名称、数据类型等。
DELETE TAG	<code>DELETE TAG <tag_name_list> FROM <VID></code>	<code>DELETE TAG test1 FROM "test"</code>	删除指定点上的指定 Tag。

3.4.6 Edge type 语句

语句	语法	示例	说明
CREATE EDGE	<code>CREATE EDGE [IF NOT EXISTS] <edge_type_name> (<prop_name> <data_type> [NULL NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>'] [{, <prop_name> <data_type> [NULL NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']}] ...) [TTL_DURATION = <tll_duration>] [TTL_COL = <prop_name>] [COMMENT = '<comment>']</code>	<code>CREATE EDGE e1(p1 string, p2 int, p3 timestamp) TTL_DURATION = 100, TTL_COL = "p2"</code>	指定名称创建一个 Edge type。
DROP EDGE	<code>DROP EDGE [IF EXISTS] <edge_type_name></code>	<code>DROP EDGE e1</code>	删除当前工作空间内的指定 Edge type。
ALTER EDGE	<code>ALTER EDGE <edge_type_name> <alter_definition> [, alter_definition] ... [ttl_definition [, ttl_definition] ...] [COMMENT = '<comment>']</code>	<code>ALTER EDGE e1 ADD (p3 int, p4 string)</code>	修改 Edge type 的结构。
SHOW EDGES	<code>SHOW EDGES</code>	<code>SHOW EDGES</code>	显示当前图空间内的所有 Edge type 名称。
DESCRIBE EDGE	<code>DESC[RIBE] EDGE <edge_type_name></code>	<code>DESCRIBE EDGE follow</code>	查看指定 Edge type 的详细信息，例如字段名称、数据类型等。

3.4.7 点语句

语句	语法	示例	说明
INSERT VERTEX	<code>INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...] VALUES <vid>: ([prop_value_list])</code>	<code>INSERT VERTEX t2 (name, age) VALUES "13":("n3", 12), "14": ("n4", 8)</code>	在NebulaGraph实例的指定图空间中插入一个或多个点。
DELETE VERTEX	<code>DELETE VERTEX <vid> [, <vid> ...]</code>	<code>DELETE VERTEX "team1"</code>	删除点，以及点关联的出边和入边。
UPDATE VERTEX	<code>UPDATE VERTEX ON <tag_name> <vid> SET <update_prop> [WHEN <condition>] [YIELD <output>]</code>	<code>UPDATE VERTEX ON player "player101" SET age = age + 2</code>	修改点上 Tag 的属性值。
UPsert VERTEX	<code>UPsert VERTEX ON <tag> <vid> SET <update_prop> [WHEN <condition>] [YIELD <output>]</code>	<code>UPsert VERTEX ON player "player667" SET age = 31</code>	结合 UPDATE 和 INSERT，如果点存在，会修改点的属性值；如果点不存在，会插入新的点。

3.4.8 边语句

语句	语法	示例	说明
INSERT EDGE	<code>INSERT EDGE [IF NOT EXISTS] <edge_type> (<prop_name_list>) VALUES <src_vid> -> <dst_vid>[@<rank>] : (<prop_value_list>) [, <src_vid> -> <dst_vid>[@<rank>] : (<prop_value_list>), ...]</code>	<code>INSERT EDGE e2 (name, age) VALUES "11"->"13": ("n1", 1)</code>	在NebulaGraph实例的指定图空间中插入一条或多条边。
DELETE EDGE	<code>DELETE EDGE <edge_type> <src_vid> -> <dst_vid>[@<rank>] [, <src_vid> -> <dst_vid>[@<rank>] ...]</code>	<code>DELETE EDGE serve "player100" -> "team204"@0</code>	删除边。一次可以删除一条或多条边。
UPDATE EDGE	<code>UPDATE EDGE ON <edge_type> <src_vid> -> <dst_vid>[@<rank>] SET <update_prop> [WHEN <condition>] [YIELD <output>]</code>	<code>UPDATE EDGE ON serve "player100" -> "team204"@0 SET start_year = start_year + 1</code>	修改边上 Edge type 的属性。
UPsert EDGE	<code>UPSERT EDGE ON <edge_type> <src_vid> -> <dst_vid> [@rank] SET <update_prop> [WHEN <condition>] [YIELD <properties>]</code>	<code>UPSERT EDGE on serve "player666" -> "team200"@0 SET end_year = 2021</code>	结合 UPDATE 和 INSERT，如果边存在，会更新边的属性；如果边不存在，会插入新的边。

3.4.9 索引

- 原生索引

索引配合 `LOOKUP` 和 `MATCH` 语句使用。

语句	语法	示例	说明
<code>CREATE INDEX</code>	<code>CREATE {TAG EDGE} INDEX [IF NOT EXISTS] <index_name> ON {<tag_name> <edge_name>} ([<prop_name_list>]) [COMMENT = '<comment>']</code>	<code>CREATE TAG INDEX player_index ON player()</code>	对 Tag、EdgeType 或其属性创建原生索引。
<code>SHOW CREATE INDEX</code>	<code>SHOW CREATE {TAG EDGE} INDEX <index_name></code>	<code>show create tag index index_2</code>	创建 Tag 或者 Edge type 时使用的 nGQL 语句，其中包含索引的详细信息，例如其关联的属性。
<code>SHOW INDEXES</code>	<code>SHOW {TAG EDGE} INDEXES</code>	<code>SHOW TAG INDEXES</code>	列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。
<code>DESCRIBE INDEX</code>	<code>DESCRIBE {TAG EDGE} INDEX <index_name></code>	<code>DESCRIBE TAG INDEX player_index_0</code>	查看指定索引的信息，包括索引的属性名称（Field）和数据类型（Type）。
<code>REBUILD INDEX</code>	<code>REBUILD {TAG EDGE} INDEX [<index_name_list>]</code>	<code>REBUILD TAG INDEX single_person_index</code>	重建索引。索引功能不会自动对其创建之前已存在的存量数据生效。
<code>SHOW INDEX STATUS</code>	<code>SHOW {TAG EDGE} INDEX STATUS</code>	<code>SHOW TAG INDEX STATUS</code>	查看索引名称和对应的状态。
<code>DROP INDEX</code>	<code>DROP {TAG EDGE} INDEX [IF EXISTS] <index_name></code>	<code>DROP TAG INDEX player_index_0</code>	删除当前图空间中已存在的索引。

- 全文索引

语法	示例	说明
<code>SIGN IN TEXT SERVICE [<elastic_ip:port> [, <username>, <password>]], <elastic_ip:port>, ...]</code>	<code>SIGN IN TEXT SERVICE (127.0.0.1:9200)</code>	NebulaGraph 的全文索引是基于 Elasticsearch 实现，部署 Elasticsearch 集群之后，可以使用 <code>SIGN IN</code> 语句登录 Elasticsearch 客户端。
<code>SHOW TEXT SEARCH CLIENTS</code>	<code>SHOW TEXT SEARCH CLIENTS</code>	列出文本搜索客户端。
<code>SIGN OUT TEXT SERVICE</code>	<code>SIGN OUT TEXT SERVICE</code>	退出所有文本搜索客户端。
<code>CREATE FULLTEXT {TAG EDGE} INDEX <index_name> ON {<tag_name> <edge_name>} ([<prop_name_list>])</code>	<code>CREATE FULLTEXT TAG INDEX nebula_index_1 ON player(name)</code>	创建全文索引。
<code>SHOW FULLTEXT INDEXES</code>	<code>SHOW FULLTEXT INDEXES</code>	显示全文索引。
<code>REBUILD FULLTEXT INDEX</code>	<code>REBUILD FULLTEXT INDEX</code>	重建全文索引。
<code>DROP FULLTEXT INDEX <index_name></code>	<code>DROP FULLTEXT INDEX nebula_index_1</code>	删除全文索引。
<code>LOOKUP ON {<tag> <edge_type>} WHERE <expression> [YIELD <return_list>]</code>	<code>LOOKUP ON player WHERE FUZZY(player.name, "Tim Duncan", AUTO, OR) YIELD player.name</code>	使用查询选项。

3.4.10 子图和路径

类型	语法	示例	说明
子图	GET SUBGRAPH [WITH PROP] [<step_count> {STEP STEPS}] FROM {<vid>, <vid>...} [{IN OUT BOTH} <edge_type>, <edge_type>...] YIELD [VERTICES AS <vertex_alias>] ,EDGES AS <edge_alias>]	GET SUBGRAPH 1 STEPS FROM "player100" YIELD VERTICES AS nodes, EDGES AS relationships	指定 Edge type 的起始点可以到达的点和边的信息，返回子图信息。
路径	FIND { SHORTEST ALL NOLOOP } PATH [WITH PROP] FROM <vertex_id_list> TO <vertex_id_list> OVER <edge_type_list> [REVERSELY BIDIRECT] [<WHERE clause>] [UPTO <N> {STEP STEPS}] YIELD path as <alias> [ORDER BY \$-.path] [LIMIT <M>]	FIND SHORTEST PATH FROM "player102" TO "team204" OVER * YIELD path as p	查找指定起始点和目的点之间的路径。返回的路径格式类似于 (<vertex_id>)-[<edge_type_name>@<rank>]->(<vertex_id>)。

3.4.11 查询调优

类型	语法	示例	说明
EXPLAIN	EXPLAIN [format="row" "dot"] <your_nGQL_statement>	EXPLAIN format="row" SHOW TAGS EXPLAIN format="dot" SHOW TAGS	输出 nGQL 语句的执行计划，但不会执行 nGQL 语句。
PROFILE	PROFILE [format="row" "dot"] <your_nGQL_statement>	PROFILE format="row" SHOW TAGS EXPLAIN format="dot" SHOW TAGS	执行 nGQL 语句，然后输出执行计划和执行概要。

3.4.12 运维

- **BALANCE**

语法	说明
BALANCE LEADER	启动任务均衡分布所有图空间中的 leader。该命令会返回任务 ID。

- **作业管理**

语法	说明
SUBMIT JOB COMPACT	触发 RocksDB 的长耗时 compact 操作。
SUBMIT JOB FLUSH	将内存中的 RocksDB memfile 写入硬盘。
SUBMIT JOB STATS	启动一个作业，该作业对当前图空间进行统计。作业完成后，用户可以使用 SHOW STATS 语句列出统计结果。
SHOW JOB <job_id>	显示当前图空间内指定作业和相关任务的信息。Meta 服务将 SUBMIT JOB 请求解析为多个任务，然后分配给进程 nebula-storaged。
SHOW JOBS	列出当前图空间内所有未过期的作业。
STOP JOB	停止当前图空间内未完成的作业。
RECOVER JOB	重新执行当前图空间内失败的作业，并返回已恢复的作业数量。

- **终止查询**

语法	示例	说明
KILL QUERY (session=<session_id>, plan=<plan_id>)	KILL QUERY(SESSION=1625553545984255,PLAN=163)	在一个会话中执行命令终止另一个会话中的查询。

最后更新: May 4, 2023

4. nGQL 指南

4.1 nGQL 概述

4.1.1 什么是 nGQL

nGQL是NebulaGraph使用的的声明式图查询语言，支持灵活高效的图模式，而且 nGQL 是为开发和运维人员设计的类 SQL 查询语言，易于学习。

nGQL是一个进行中的项目，会持续发布新特性和优化，因此可能会出现语法和实际操作不一致的问题，如果遇到此类问题，请提交 [issue](#) 通知 NebulaGraph团队。NebulaGraph 3.0 及更新版本正在支持 [openCypher 9](#)。

nGQL 可以做什么

- 支持图遍历
- 支持模式匹配
- 支持聚合
- 支持修改图
- 支持访问控制
- 支持聚合查询
- 支持索引
- 支持大部分 openCypher 9 图查询语法（不支持修改和控制语法）

示例数据 Basketballplayer

用户可以下载NebulaGraph示例数据 [basketballplayer 文件](#)，然后使用 [NebulaGraph Console](#)，使用选项 -f 执行脚本。

Note

导入示例数据前，确保已执行 ADD HOSTS 命令将 Storage 主机增加至集群中。更多信息，请参见[管理 Storage 主机](#)。

占位标识符和占位符值

NebulaGraph查询语言 nGQL 参照以下标准设计：

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

在模板代码中，任何非关键字、字面值或标点符号的标记都是占位符标识符或占位符值。

本文中 nGQL 语法符号的说明如下。

符号	含义
< >	语法元素的名称。
:	定义元素的公式。
[]	可选元素。
{ }	显式的指定元素。
	所有可选的元素。
...	可以重复多次。

例如创建点的 nGQL 语法:

```
INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...]
VALUES <vid>: ([prop_value_list])

tag_props:
  tag_name ([prop_name_list])

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

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

示例语句:

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

关于 openCypher 兼容性

原生 **NGQL** 和 **OPENCYCER** 的关系

原生 nGQL 是由 NebulaGraph 自行创造和实现的图查询语言。openCypher 是由 openCypher Implementers Group 组织所开源和维护的图查询语言，最新版本为 openCypher 9。

由于 nGQL 语言部分兼容了 openCypher，这个部分在本文中称为 openCypher 兼容语句。



nGQL 语言 = 原生 nGQL 语句 + openCypher 兼容语句

NGQL 完全兼容 **OPENCYCER 9** 吗?

不。



nGQL 设计目标仅为兼容部分的 DQL 语句(match, optional match, with 等)。

不计划兼容任何 DDL, DML, DCL;

不计划兼容 Bolt 协议;

不计划兼容 APOC 与 GDS。

在本文搜索 "compatibility" 或者 "兼容性" 查看具体不兼容的细节。

在 **Issues** 中已经列出已知的兼容错误。如果发现这种类型的新问题，请提交问题并附带 **incompatible** 标签。

NGQL 和 **OPENCYpher 9** 的主要差异有哪些?

类别	openCypher 9	nGQL
Schema	弱 Schema	强 Schema
相等运算符	=	==
数学求幂	^	使用 <code>pow(x, y)</code> 替代 ^。
边 Rank	无此概念	用 <code>@rank</code> 设置。
语句	-	不支持 openCypher 9 的所有 DML 语句（如 <code>CREATE</code> 、 <code>MERGE</code> 等），不支持所有的 DCL，和支持部分 <code>MATCH</code> 、 <code>OPTIONAL MATCH</code> 语法和函数。
语句文本换行	换行符	\ + 换行符
Label 与 Tag 是不同的概念	Label 用于寻找点（点的索引）。	Tag 用于定义点的一种类型及相应的属性，无索引功能。
预编译与参数化查询	支持	仅支持参数化查询。

↑ Compatibility

请注意 **openCypher 9** 和 **Cypher** 在语法和许可上有不同:

1. Cypher 要求所有 Cypher 语句必须“显式地在一个事务中”执行，而 openCypher 没有这样的要求。另外，nGQL 没有事务及隔离性。
2. Cypher 企业版功能有多种的约束（constraints），包括 Unique node property constraints、Node property existence constraints、Relationship property existence constraints、Node key constraints。OpenCypher 标准中没有约束。而 nGQL 是强 Schema 系统，前述的约束大多通过 Schema 定义可实现（包括 NOT NULL），唯一不能支持的功能是“属性值唯一性”（UNIQUE constraint）。
3. Cypher 有 APoC，openCypher 9 没有 APoC。Cypher 有 Bolt 协议支持要求，openCypher 9 没有。

哪里可以找到更多 **NGQL** 的示例?

用户可以在 GitHub 的 **features** 目录内查看超过 2500 条 nGQL 示例。

features 目录内包含很多 **features** 格式的文件，每个文件都记录了使用 nGQL 的场景和示例。例如:

```
Feature: Basic match

Background:
  Given a graph with space named "basketballplayer"

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

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

```
Feature: Comparison of where clause

Background:
  Given a graph with space named "basketballplayer"

Scenario: push edge props filter down
  When profiling query:
```

```
"""
GO FROM "player100" OVER follow
WHERE properties(edge).degree IN [v IN [95,99] WHERE v > 0]
YIELD dst(edge), properties(edge).degree
"""

Then the result should be, in any order:
| follow_dst | follow.degree |
| "player101" | 95           |
| "player125" | 95           |

And the execution plan should be:
| id | name      | dependencies | operator info |
| 0  | Project   | 1            |               |
| 1  | GetNeighbors | 2          | {"filter": "(properties(edge).degree IN [v IN [95,99] WHERE (v>0)])"} |
| 2  | Start     |              |               |

```

示例中的关键字说明如下。

关键字	说明
Feature	描述当前文档的主题。
Background	描述当前文档的背景信息。
Given	描述执行示例语句的前提条件。
Scenario	描述具体场景。如果场景之前有 @skip 标识，表示这个场景下示例语句可能无法正常工作，请不要在生产环境中使用该示例语句。
When	描述要执行的 nGQL 示例语句。可以是 executing query 或 profiling query。
Then	描述执行 When 内语句的预期返回结果。如果返回结果和文档不同，请提交 issue 通知 NebulaGraph 团队。
And	描述执行 When 内语句的副作用或执行计划。
@skip	跳过这个示例。通常表示测试代码还没有准备好。

欢迎[增加更多 tck case](#)，在 CI/CD 中自动回归所使用的语句。

是否支持 TINKERPOP GREMLIN?

不支持。也没有计划。

是否支持 W3C 的 RDF (SPARQL) 或 GRAPHQL 等?

不支持。也没有计划。

NebulaGraph 的数据模型是属性图，是一个强 Schema 系统，不支持 RDF 标准。

nGQL 也不支持 SPARQL 和 GraphQL。

最后更新: May 4, 2023

4.1.2 模式

模式（pattern）和图模式匹配，是图查询语言的核心功能，本文介绍NebulaGraph设计的各种模式，部分还未实现。

单点模式

点用一对括号来描述，通常包含一个名称。例如：

(a)

示例为一个简单的模式，描述了单个点，并使用变量 a 命名该点。

多点关联模式

多个点通过边相连是常见的结构，模式用箭头来描述两个点之间的边。例如：

(a)-[]->(b)

示例为一个简单的数据结构：两个点和一条连接两个点的边，两个点分别为 a 和 b，边是有方向的，从 a 到 b。

这种描述点和边的方式可以扩展到任意数量的点和边，例如：

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

这样的一系列点和边称为 路径（path）。

只有在涉及某个点时，才需要命名这个点。如果不涉及这个点，则可以省略名称，例如：

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

Tag 模式



nGQL 中的 Tag 概念与 openCypher 中的 Label 有一些不同。例如，必须创建一个 Tag 之后才能使用它，而且 Tag 还定义了属性的类型。

模式除了简单地描述图中的点之外，还可以描述点的 Tag。例如：

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

模式也可以描述有多个 Tag 的点，例如：

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

属性模式

点和边是图的基本结构。nGQL 在这两种结构上都可以增加属性，方便实现更丰富的模型。

在模式中，属性的表示方式为：用花括号括起一些键值对，用英文逗号分隔，并且需要指定属性所属的 Tag 或者 Edge type。

例如一个点有两个属性：

(a:Player{name: "Tim Duncan", age: 42})

在这个点上可以有一条边是：

(a)-[e:follow{degree: 95}]->(b)

边模式

描述一条边最简单的方法是使用箭头连接两个点。

可以用以下方式描述边以及它的方向性。如果不关心边的方向，可以省略箭头，例如：

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

和点一样，边也可以命名。一对方括号用于分隔箭头，变量放在两者之间。例如：

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

和点上的 Tag 一样，边也可以有类型。描述边的类型，例如：

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

和点上的 Tag 不同，一条边只能有一种 Edge type。但是如果我们想描述多个可选 Edge type，可以用管道符号 (|) 将可选值分开，例如：

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

和点一样，边的名称可以省略，例如：

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

变长模式

在图中指定边的长度来描述多条边（以及中间的点）组成的一条长路径，不需要使用多个点和边来描述。例如：

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

该模式描述了 3 点 2 边组成的图，它们都在一条路径上（长度为 2），等价于：

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

也可以指定长度范围，这样的边模式称为 variable-length edges，例如：

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

*3..5 表示最小长度为 3，最大长度为 5。

该模式描述了 4 点 3 边、5 点 4 边或 6 点 5 边组成的图。

也可以忽略最小长度，只指定最大长度，例如：

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



必须指定最大长度，不支持仅指定最小长度 ((a)-[*3..]->(b)) 或都不指定 ((a)-[*]->(b))。

路径变量

一系列连接的点和边称为 路径。nGQL 允许使用变量来命名路径，例如：

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

可以在 MATCH 语句中使用路径变量。

4.1.3 注释

本文介绍 nGQL 中的注释方式。

Examples

```
nebula> # 这行什么都不做。
nebula> RETURN 1+1;      # 这条注释延续到行尾。
nebula> RETURN 1+1;      // 这条注释延续到行尾。
nebula> RETURN 1 /* 这是一条行内注释 */ + 1 == 2;
nebula> RETURN 11 +
/* 多行注释           \
用反斜线来换行。    \
*/ 12;
```

nGQL 语句中的反斜线 (\) 代表换行。

OpenCypher 兼容性

- 在 nGQL 中，用户必须在行末使用反斜线 (\) 来换行，即使是在使用 /* */ 符号的多行注释内。
- 在 openCypher 中不需要使用反斜线换行。

```
/* openCypher 风格：
这条注释
延续了不止
一行 */
MATCH (n:Label)
RETURN n;
```

```
/* 原生 nGQL 风格： \
这条注释      \
延续了不止      \
一行 */      \
MATCH (n:tag) \
RETURN n;
```

最后更新: May 4, 2023

4.1.4 大小写区分

标识符区分大小写

以下语句会出现错误，因为 `my_space` 和 `MY_SPACE` 是两个不同的图空间。

```
nebula> CREATE SPACE IF NOT EXISTS my_space (vid_type=FIXED_STRING(30));
nebula> use MY_SPACE;
[ERROR (-1005)]: SpaceNotFound:
```

关键字不区分大小写

以下语句是等价的，因为 `show` 和 `spaces` 是关键字。

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

函数不区分大小写

函数名称不区分大小写，例如 `count()`、`COUNT()`、`couNT()` 是等价的。

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

最后更新: May 4, 2023

4.1.5 关键字

关键字在 nGQL 中有重要意义，分为保留关键字和非保留关键字。建议不要在 Schema 中使用关键字。

如果必须使用关键字：

- 当非保留关键字均为小写字母时，如果将其作为标识符时可以不使用引号；当非保留关键字包含大写字母时并将其作为标识符时，需要用反引号（`）包围，例如 `Comment`。
- 保留关键字或特殊字符作为标识符时，需要用反引号（`）包围，例如 `AND`。



关键字不区分大小写。

```
nebula> CREATE TAG TAG(name string);
[ERROR (-1004)]: SyntaxError: syntax error near `TAG'

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

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

nebula> CREATE TAG 中文(简体 string);
Execution succeeded

nebula> CREATE TAG `¥%特殊字符&*+*/` (`q-! () = wer` string);
Execution succeeded
```

保留关键字

```
ACROSS
ADD
ALTER
AND
AS
ASC
ASCENDING
BALANCE
BOOL
BY
CASE
CHANGE
COMPACT
CREATE
DATE
DATETIME
DELETE
DESC
DESCENDING
DESCRIBE
DISTINCT
DOUBLE
DOWNLOAD
DROP
DURATION
EDGE
EDGES
EXISTS
EXPLAIN
FETCH
FIND
FIXED_STRING
FLOAT
FLUSH
FORMAT
FROM
GET
GO
GRANT
IF
IGNORE_EXISTED_INDEX
IN
INDEX
INDEXES
INGEST
INSERT
INT
```

```
INT16
INT32
INT64
INT8
INTERSECT
IS
LIMIT
LIST
LOOKUP
MAP
MATCH
MINUS
NO
NOT
NOT_IN
NULL
OF
OFFSET
ON
OR
ORDER
OVER
OVERWRITE
PROFILE
PROP
REBUILD
RECOVER
REMOVE
RESTART
RETURN
REVERSELY
REVOKE
SET
SHOW
STEP
STEPS
STOP
STRING
SUBMIT
TAG
TAGS
TIME
TIMESTAMP
TO
UNION
UPDATE
UPSERT
UPTO
USE
VERTEX
VERTICES
WHEN
WHERE
WITH
XOR
YIELD
```

非保留关键字

```
ACCOUNT
ADMIN
ALL
ANY
ATOMIC_EDGE
AUTO
BIDIRECT
BOTH
CHARSET
CLIENTS
COLLATE
COLLATION
COMMENT
CONFIGS
CONTAINS
DATA
DBA
DEFAULT
ELASTICSEARCH
ELSE
END
ENDS
ENDS_WITH
FORCE
FULLTEXT
FUZZY
GOD
GRAPH
GROUP
GROUPS
GUEST
HDFS
```

```
HOST
HOSTS
INTO
IS_EMPTY
IS_NOT_EMPTY
IS_NOT_NULL
IS_NULL
JOB
JOBS
KILL
LEADER
LISTENER
META
NOLOOP
NONE
NOT_CONTAINS
NOT_ENDS_WITH
NOT_STARTS_WITH
OPTIONAL
OUT
PART
PARTITION_NUM
PARTS
PASSWORD
PATH
PLAN
PREFIX
QUERIES
QUERY
REDUCE
REGEXP
REPLICA_FACTOR
RESET
ROLE
ROLES
SAMPLE
SEARCH
SERVICE
SESSION
SESSIONS
SHORTEST
SIGN
SINGLE
SKIP
SNAPSHOT
SNAPSHOTS
SPACE
SPACES
STARTS
STARTS_WITH
STATS
STATUS
STORAGE
SUBGRAPH
TEXT
TEXT_SEARCH
THEN
TOP
TTL_COL
TTL_DURATION
UNWIND
USER
USERS
UUID
VALUE
VALUES
VID_TYPE
WILDCARD
ZONE
ZONES
FALSE
TRUE
```

最后更新: May 4, 2023

4.1.6 nGQL 风格指南

nGQL 没有严格的构建格式要求，但根据恰当而统一的风格创建 nGQL 语句有利于提高可读性、避免歧义。在同一组织或项目中使用相同的 nGQL 风格有利于降低维护成本，规避因格式混乱或误解造成的问题。本文为写作 nGQL 语句提供了风格参考。

↑ Compatibility

nGQL 风格与 [Cypher Style Guide](#) 不同。

换行

1. 换行写子句。

不推荐:

```
GO FROM "player100" OVER follow REVERSELY YIELD src(edge) AS id;
```

推荐:

```
GO FROM "player100" \
OVER follow REVERSELY \
YIELD src(edge) AS id;
```

2. 换行写复合语句中的不同语句。

不推荐:

```
GO FROM "player100" OVER follow REVERSELY YIELD src(edge) AS id | GO FROM $-.id \
OVER serve WHERE properties($^).age > 20 YIELD properties($^).name AS FriendOf, properties($$).name AS Team;
```

推荐:

```
GO FROM "player100" \
OVER follow REVERSELY \
YIELD src(edge) AS id | \
GO FROM $-.id OVER serve \
WHERE properties($^).age > 20 \
YIELD properties($^).name AS FriendOf, properties($$).name AS Team;
```

3. 子句长度超过 80 个字符时，在合适的位置换行。

不推荐:

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

推荐:

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

Note

即使子句不超过 80 个字符，如需换行后有助于理解，也可将子句再次分行。

标识符命名

在 nGQL 语句中，关键字、标点符号、空格以外的字符内容都是标识符。推荐的标识符命名方式如下。

1. 使用单数名词命名 Tag，用原型动词或动词短语构成 Edge type。

不推荐：

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

推荐：

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

2. 标识符用蛇形命名法，以下划线（_）连接单词，且所有字母小写。

不推荐：

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

推荐：

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

3. 语法关键词大写，变量小写。

不推荐：

```
match (v:player) return V limit 5;
```

推荐：

```
MATCH (v:player) RETURN v LIMIT 5;
```

Pattern

1. 分行写 Pattern 时，在表示边的箭头右侧换行，而不是左侧。

不推荐：

```
MATCH (v:player{name: "Tim Duncan", age: 42}) \
-[e:follow]->()-[e2:serve]->()-<--(v2) \
RETURN v, e, v2;
```

推荐：

```
MATCH (v:player{name: "Tim Duncan", age: 42})-[e:follow]-> \
()-[e2:serve]->()-<--(v2) \
RETURN v, e, v2;
```

2. 将无需查询的点和边匿名化。

不推荐：

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

推荐：

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

3. 将非匿名点放在匿名点的前面。

不推荐：

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

推荐:

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

字符串

字符串用双引号包围。

不推荐:

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

推荐:

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

Note

字符串中需要嵌套单引号或双引号时，用反斜线 (\) 转义。例如:

```
RETURN '\"The database is amazing,\\\" the user says.\"';
```

结束语句

1. 用英文分号 (;) 结束 nGQL 语句。

不推荐:

```
FETCH PROP ON player "player100" YIELD properties(vertex)
```

推荐:

```
FETCH PROP ON player "player100" YIELD properties(vertex);
```

2. 使用管道符 (|) 分隔的复合语句，仅在最后一行末用英文分号结尾。在管道符前使用英文分号会导致语句执行失败。

不支持:

```
GO FROM "player100" \
OVER follow \
YIELD dst(edge) AS id; | \
GO FROM $-.id \
OVER serve \
YIELD properties($$).name AS Team, properties($^).name AS Player;
```

支持:

```
GO FROM "player100" \
OVER follow \
YIELD dst(edge) AS id | \
GO FROM $-.id \
OVER serve \
YIELD properties($$).name AS Team, properties($^).name AS Player;
```

3. 在包含自定义变量的复合语句中，用英文分号结束定义变量的语句。不按规则加分号或使用管道符结束该语句会导致执行失败。

不支持:

```
$var = GO FROM "player100" \
OVER follow \
YIELD dst(edge) AS id \
GO FROM $var.id \
OVER serve \
YIELD properties($$).name AS Team, properties($^).name AS Player;
```

也不支持:

```
$var = GO FROM "player100" \
OVER follow \
YIELD dst(edge) AS id | \
GO FROM $var.id \
OVER serve \
YIELD properties($$).name AS Team, properties($^).name AS Player;
```

支持:

```
$var = GO FROM "player100" \
OVER follow \
YIELD dst(edge) AS id; \
GO FROM $var.id \
OVER serve \
YIELD properties($$).name AS Team, properties($^).name AS Player;
```

最后更新: May 4, 2023

4.2 数据类型

4.2.1 数值

nGQL 支持整数和浮点数。

整数

nGQL 支持带符号的 64 位整数 (INT64)、32 位整数 (INT32)、16 位整数 (INT16) 和 8 位整数 (INT8)。

类型	声明关键字	范围
INT64	INT64 或 INT	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
INT32	INT32	-2,147,483,648 ~ 2,147,483,647
INT16	INT16	-32,768 ~ 32,767
INT8	INT8	-128 ~ 127

浮点数

nGQL 支持单精度浮点 (FLOAT) 和双精度浮点 (DOUBLE)。

类型	声明关键字	范围	精度
FLOAT	FLOAT	3.4E +/- 38	6~7 位
DOUBLE	DOUBLE	1.7E +/- 308	15~16 位

nGQL 支持科学计数法，例如 `1e2`、`1.1e2`、`.3e4`、`1.e4`、`-1234E-10`。

Note

不支持 MySQL 中的 DECIMAL 数据类型。

数值的读写

在写入和读取不同类型的数据时，nGQL 的行为遵守以下规则：

数值类型	设置为 VID	设置为属性类型	读取该类型的属性值得到的类型
INT64	支持	支持	INT64
INT32	不支持	支持	INT64
INT16	不支持	支持	INT64
INT8	不支持	支持	INT64
FLOAT	不支持	支持	DOUBLE
DOUBLE	不支持	支持	DOUBLE

例如，nGQL 不支持设置 INT8 类型的 VID，但支持将 TAG 或 Edge type 的某个属性类型设置为 INT8。当使用 nGQL 语句读取 INT8 类型的属性时，获取到的值的类型为 INT64。

- NebulaGraph 支持写入多种进制的数值：
- 十进制，例如 123456。
- 十六进制，例如 0x1e240。
- 八进制，例如 0361100。

但 NebulaGraph 会将写入的非十进制数值解析为十进制的值保存。读取到的值为十进制。

例如，属性 score 的类型为 INT，通过 INSERT 语句为其赋值 0xb，使用 FETCH 等语句查询该属性值获取到的结果是 11，即将十六进制的 0xb 转换为十进制后的值。

- 将 FLOAT/DOUBLE 类型的数值插入 INT 类型的列，会将数值四舍五入取整。

最后更新: May 4, 2023

4.2.2 布尔

NebulaGraph 使用关键字 `BOOL` 声明布尔数据类型，可选值为 `true` 或 `false`。

nGQL 支持以如下方式使用布尔值：

- 将属性值的数据类型定义为布尔。
- 在 `WHERE` 子句中用布尔值作为判断条件。

最后更新: May 4, 2023

4.2.3 字符串

NebulaGraph 支持定长字符串和变长字符串。

声明与表示方式

nGQL 中的字符串声明方式如下:

- 使用关键字 `STRING` 声明变长字符串。
- 使用关键字 `FIXED_STRING(<length>)` 声明定长字符串, `<length>` 为字符串长度, 例如 `FIXED_STRING(32)`。

字符串的表示方式为用双引号或单引号包裹, 例如 `"Hello, Cooper"` 或 `'Hello, Cooper'`。

字符串读写

nGQL 支持以如下方式使用字符串:

- 将 `VID` 的数据类型定义为定长字符串。
- 将变长字符串设置为 Schema 名称, 包括图空间、Tag、Edge type 和属性的名称。
- 将属性值的数据类型定义为定长或变长字符串。

例如:

- 将属性值的类型定义为定长字符串

```
nebula> CREATE TAG IF NOT EXISTS t1 (p1 FIXED_STRING(10));
```

- 将属性值的类型定义为变长字符串

```
nebula> CREATE TAG IF NOT EXISTS t2 (p2 STRING);
```

如果尝试写入的定长字符串超出长度限制:

- 当该定长字符串为属性值时, 写入会成功, NebulaGraph 将截断字符串, 仅存入符合长度限制的部分。
- 当该定长字符串为 VID 时, 写入会失败, NebulaGraph 将报错。

转义字符

字符串中不支持直接换行, 可以使用转义字符实现, 例如:

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

OpenCypher 兼容性

openCypher、Cypher 和 nGQL 之间有一些细微区别, 例如下面 openCypher 的示例, 不能将单引号替换为双引号。

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

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

Cypher 的返回结果同时支持单引号和双引号, nGQL 遵循 Cypher 的方式。

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

最后更新: May 4, 2023

4.2.4 日期和时间类型

本文介绍日期和时间的类型，包括 DATE 、 TIME 、 DATETIME 、 TIMESTAMP 和 DURATION 。

注意事项

- 在插入时间类型的属性值时，NebulaGraph会根据[配置文件](#)中 `timezone_name` 参数指定的时区，将该 DATE 、 TIME 、 DATETIME 转换成相应的世界协调时间（UTC）时间。

Note

如需修改当前时区，请同时修改所有服务的配置文件中的 `timezone_name` 参数。

- 函数 `date()` 、 `time()` 和 `datetime()` 可以指定期区进行转换，例如 `datetime("2017-03-04 22:30:40.003000+08:00")` 或 `datetime("2017-03-04T22:30:40.003000[Asia/Shanghai]")` 。
- 函数 `date()` 、 `time()` 、 `datetime()` 和 `timestamp()` 可以用空值获取当前的日期或时间。
- 函数 `date()` 、 `time()` 、 `datetime()` 和 `duration()` 可以用属性名称获取自身的某一个具体属性值，例如 `date().month` 获取当前月份、`time("02:59:40").minute` 获取传入时间的分钟数。

Note

设置时间的年份为负数时，需要使用 Map 类型数据。

openCypher 兼容性

- 支持年、月、日、时、分、秒、毫秒、微秒，不支持纳秒。
- 不支持函数 `localdatetime()` 。
- 不支持大部分字符串时间格式，支持 `YYYY-MM-DDThh:mm:ss` 和 `YYYY-MM-DD hh:mm:ss` 。
- 支持单个数字的字符串时间格式，例如 `time("1:1:1")` 。

DATE

DATE 包含日期，但是不包含时间。NebulaGraph检索和显示 DATE 的格式为 `YYYY-MM-DD` 。支持的范围是 -32768-01-01 到 32767-12-31 。

`date()` 支持的属性名称包括 `year` 、 `month` 和 `day` 。`date()` 支持输入 `YYYY` 、 `YYYY-MM` 或 `YYYY-MM-DD` ，未输入的月份或日期默认为 01 。

```
nebula> RETURN DATE({year:-123, month:12, day:3});
+-----+
| date({year:-123,month:12,day:3}) |
+-----+
| -123-12-03 |
+-----+
nebula> RETURN DATE("23333");
+-----+
| date("23333") |
+-----+
| 23333-01-01 |
+-----+
```

TIME

TIME 包含时间，但是不包含日期。NebulaGraph检索和显示 TIME 的格式为 `hh:mm:ss.msmsmsususus` 。支持的范围是 00:00:00.000000 到 23:59:59.999999 。

`time()` 支持的属性名称包括 `hour` 、 `minute` 和 `second` 。

DATETIME

`DATETIME` 包含日期和时间。NebulaGraph 检索和显示 `DATETIME` 的格式为 `YYYY-MM-DDThh:mm:ss.msmsmsususus`。支持的范围是 `-32768-01-01T00:00:00.000000` 到 `32767-12-31T23:59:59.999999`。

- `datetime()` 支持的属性名称包括 `year`、`month`、`day`、`hour`、`minute` 和 `second`。
- `datetime()` 可将 `TIMESTAMP` 类型的日期值转换成 `DATETIME` 类型的日期值。`TIMESTAMP` 类型的日期值取值范围: `0~9223372036`。
- `datetime()` 支持 `int` 类型的参数, 该 `int` 参数表示时间戳。

```
# 获取当前时间。
nebula> RETURN datetime();
+-----+
| datetime() |
+-----+
| 2022-08-29T06:37:08.933000 |
+-----+

# 获取当前时间的小时。
nebula> RETURN datetime().hour;
+-----+
| datetime().hour |
+-----+
| 6 |
+-----+

# 将时间戳转换成 DATETIME 类型的格式。
nebula> RETURN datetime(timestamp(1625469277));
+-----+
| datetime(timestamp(1625469277)) |
+-----+
| 2021-07-05T07:14:37.000000 |
+-----+

nebula> RETURN datetime(1625469277);
+-----+
| datetime(1625469277) |
+-----+
| 2021-07-05T07:14:37.000000 |
+-----+
```

TIMESTAMP

`TIMESTAMP` 包含日期和时间。支持的范围是 UTC 时间的 `1970-01-01T00:00:01` 到 `2262-04-11T23:47:16`。

`TIMESTAMP` 还有以下特点:

- 以时间戳形式存储和显示。例如 `1615974839`, 表示 `2021-03-17T17:53:59`。
- 查询 `TIMESTAMP` 类型属性值的方式包括时间戳整数和 `timestamp()` 函数。
- 插入 `TIMESTAMP` 类型属性值的方式包括时间戳整数、`timestamp()` 函数和 `now()` 函数。
- `timestamp()` 函数支持传入空值获取当前时间戳; 同时支持传入整数以标识该整数为时间戳, 整数取值: `0~9223372036`。
- `timestamp()` 函数可将 `DATETIME` 类型的日期值转换成 `TIMESTAMP` 类型的日期值, 且传入的 `DATETIME` 类型的日期值为 `string` 类型。
- 底层存储的数据格式为 **64** 位 `int`。

```
# 传入当前时间。
nebula> RETURN timestamp();
+-----+
| timestamp() |
+-----+
| 1625469277 |
+-----+

# 传入指定时间。
nebula> RETURN timestamp("2022-01-05T06:18:43");
+-----+
| timestamp("2022-01-05T06:18:43") |
+-----+
| 1641363523 |
+-----+

# 传入 datetime()。
nebula> RETURN timestamp(datetime("2022-08-29T07:53:10.939000"));
+-----+
| timestamp(datetime("2022-08-29T07:53:10.939000")) |
+-----+
```

+-----+
1661759590
+-----+

Note

传入 `timestamp()` 函数的时间字符串不支持包含毫秒和微秒，但是通过 `timestamp(datetime())` 传入的时间字符串支持包含毫秒和微秒。

DURATION

`DURATION` 是一段连续的时间，由 `years`、`months`、`days`、`hours`、`minutes`、`seconds` 六个 Key 自由组合成的 Map 类型数据表示。例如 `duration({years: 12, months: 5, days: 14, hours: 16, minutes: 12, seconds: 70})`。

`DURATION` 还有以下特点：

- 不支持为 `DURATION` 类型数据创建索引。
- 可以用于对指定时间进行计算。

示例

1. 创建 Tag，名称为 `date1`，包含 `DATE`、`TIME` 和 `DATETIME` 三种类型。

```
nebula> CREATE TAG IF NOT EXISTS date1(p1 date, p2 time, p3 datetime);
```

2. 插入点，名称为 `test1`。

```
nebula> INSERT VERTEX date1(p1, p2, p3) VALUES "test1":(date("2021-03-17"), time("17:53:59"), datetime("2017-03-04T22:30:40.003000[Asia/Shanghai]"));
```

3. 查询 `test1` 的属性 `p1` 是否为 `2021-03-17`。

```
nebula> MATCH (v:date1) RETURN v.date1.p1 == date("2021-03-17");
+-----+
| (v.date1.p1==date("2021-03-17")) |
+-----+
| true |
+-----+
```

4. 获取 `test1` 的属性 `p1` 的月份。

```
nebula> CREATE TAG INDEX IF NOT EXISTS date1_index ON date1(p1);
nebula> REBUILD TAG INDEX date1_index;
nebula> MATCH (v:date1) RETURN v.date1.p1.month;
+-----+
| v.date1.p1.month |
+-----+
| 3 |
+-----+
```

5. 查找 Tag `date1` 中属性 `p3` 小于 `2023-01-01T00:00:00.000000` 的值。

```
nebula> MATCH (v:date1) \
WHERE v.date1.p3 < datetime("2023-01-01T00:00:00.000000") \
RETURN v.date1.p3;
+-----+
| v.date1.p3 |
+-----+
| 2017-03-04T14:30:40.003000 |
+-----+
```

6. 创建 Tag，名称为 `school`，包含 `TIMESTAMP` 类型。

```
nebula> CREATE TAG IF NOT EXISTS school(name string, found_time timestamp);
```

7. 插入点，名称为 `DUT`，存储时间为 `"1988-03-01T08:00:00"`。

```
# 时间戳形式插入，1988-03-01T08:00:00 对应的时间戳为 573177600，转换为 UTC 时间为 573206400。
nebula> INSERT VERTEX school(name, found_time) VALUES "DUT":("DUT", 573206400);
```

```
# 日期和时间格式插入。
nebula> INSERT VERTEX school(name, found_time) VALUES "DUT":("DUT", timestamp("1988-03-01T08:00:00"));
```

8. 插入点，名称为 `dut`，用 `now()` 或 `timestamp()` 函数存储时间。

```
# 用 now() 函数存储时间
nebula> INSERT VERTEX school(name, found_time) VALUES "dut":("dut", now());

# 用 timestamp() 函数存储时间
nebula> INSERT VERTEX school(name, found_time) VALUES "dut":("dut", timestamp());
```

还可以使用 `WITH` 语句设置具体日期时间或进行计算，例如：

```
nebula> WITH time({hour: 12, minute: 31, second: 14, millisecond:111, microsecond: 222}) AS d RETURN d;
+-----+
| d |
+-----+
| 12:31:14.111222 |
+-----+

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

nebula> WITH date('1984-10-11') as x, duration({years: 12, days: 14, hours: 99, minutes: 12}) as d \
    RETURN x + d AS sum, x - d AS diff;
+-----+-----+
| sum | diff |
+-----+-----+
| 1996-10-29 | 1972-09-23 |
+-----+-----+
```

最后更新: May 4, 2023

4.2.5 NULL

默认情况下，插入点或边时，属性值可以为 `NULL`，用户也可以设置属性值不允许为 `NULL`（`NOT NULL`），即插入点或边时必须设置该属性的值，除非创建属性时已经设置默认值。

`NULL` 的逻辑操作

`AND`、`OR`、`XOR` 和 `NOT` 的真值表如下。

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

OpenCypher 兼容性

NebulaGraph中，`NULL` 的比较和操作与 openCypher 不同，后续也可能会有变化。

`NULL` 的比较

NebulaGraph中，`NULL` 的比较操作不兼容 openCypher。

`NULL` 的操作和返回

NebulaGraph中，对 `NULL` 的操作以及返回结果不兼容 openCypher。

示例

使用 `NOT NULL`

创建 Tag，名称为 `player`，指定属性 `name` 为 `NOT NULL`。

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

使用 `SHOW` 命令查看创建 Tag 语句，属性 `name` 为 `NOT NULL`，属性 `age` 为默认的 `NULL`。

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

插入点 `Kobe`，属性 `age` 可以为 `NULL`。

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

使用 `NOT NULL` 并设置默认值

创建 Tag, 名称为 `player`, 指定属性 `age` 为 `NOT NULL`, 并设置默认值 `18`。

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

插入点 `Kobe`, 只设置属性 `name`。

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

查询点 `Kobe`, 属性 `age` 为默认值 `18`。

```
nebula> FETCH PROP ON player "Kobe" YIELD properties(vertex);
+-----+
| properties(VERTEX)      |
+-----+
| {age: 18, name: "Kobe"} |
+-----+
```

最后更新: May 4, 2023

4.2.6 列表

列表（List）是复合数据类型，一个列表是一组元素的序列，可以通过元素在序列中的位置访问列表中的元素。

列表用左方括号（[）和右方括号（]）包裹多个元素，各个元素之间用英文逗号（,）隔开。元素前后的空格在列表中被忽略，因此可以使用换行符、制表符和空格调整格式。

OpenCypher 兼容性

复合数据类型（例如 List、Set、Map）不能存储为点或边的属性。

列表操作

对列表进行操作可以使用预设的[列表函数](#)，也可以使用下标表达式过滤列表内的元素。

下标表达式语法

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

nGQL 的下标支持从前往后查询，从 0 开始，0 表示第一个元素，1 表示第二个元素，以此类推；也支持从后往前查询，从 -1 开始，-1 表示最后一个元素，-2 表示倒数第二个元素，以此类推。

- [M]: 表示下标为 M 的元素。
- [M..N]: 表示 $M \leq$ 下标 $< N$ 的元素。N 为 0 时，返回为空。
- [M..]: 表示 $M \leq$ 下标 的元素。
- [..N]: 表示 下标 $< N$ 的元素。N 为 0 时，返回为空。

Note

- 越界的下标返回为空，未越界的可以正常返回。
- $M \geq N$ 时，返回为空。
- 查询单个元素时，如果 M 为 null，返回报错 BAD_TYPE；范围查询时，M 或 N 为 null，返回为 null。

示例

```
# 返回列表 [1,2,3]
nebula> RETURN List[1, 2, 3] AS a;
+-----+
| a   |
+-----+
| [1, 2, 3] |
+-----+

# 返回列表 [1,2,3,4,5] 中位置下标为 3 的元素。列表的位置下标是从 0 开始，因此返回的元素为 4。
nebula> RETURN range(1,5)[3];
+-----+
| range(1,5)[3] |
+-----+
| 4           |
+-----+

# 返回列表 [1,2,3,4,5] 中位置下标为-2 的元素。列表的最后一个元素的位置下标是-1，因此-2 是指倒数第二个元素，即 4。
nebula> RETURN range(1,5)[-2];
+-----+
| range(1,5)[-2] |
+-----+
| 4           |
+-----+

# 返回列表 [1,2,3,4,5] 中下标位置从 0 到 3 (不包括 3) 的元素。
nebula> RETURN range(1,5)[0..3];
+-----+
| range(1,5)[0..3] |
+-----+
```

```

| [1, 2, 3] |
+-----+
# 返回列表 [1,2,3,4,5] 中位置下标大于 2 的元素。
nebula> RETURN range(1,5)[3..] AS a;
+-----+
| a |
+-----+
| [4, 5] |
+-----+

# 返回列表内下标小于 3 的元素。
nebula> WITH list[1, 2, 3, 4, 5] AS a \
    RETURN a[..3] AS r;
+-----+
| r |
+-----+
| [1, 2, 3] |
+-----+

# 筛选列表 [1,2,3,4,5] 中大于 2 的元素，将这些元素分别做运算并返回。
nebula> RETURN [n IN range(1,5) WHERE n > 2 | n + 10] AS a;
+-----+
| a |
+-----+
| [13, 14, 15] |
+-----+

# 返回列表内第一个至倒数第二个（包括）的元素。
nebula> YIELD list[1, 2, 3][0..-1] AS a;
+-----+
| a |
+-----+
| [1, 2] |
+-----+

# 返回列表内倒数第三个至倒数第一个（不包括）的元素。
nebula> YIELD list[1, 2, 3, 4, 5][-3..-1] AS a;
+-----+
| a |
+-----+
| [3, 4] |
+-----+

# 设置变量，返回列表内下标为 1、2 的元素。
nebula> $var = YIELD 1 AS f, 3 AS t; \
    YIELD list[1, 2, 3][$var.f..$var.t] AS a;
+-----+
| a |
+-----+
| [2, 3] |
+-----+

# 越界的下标返回为空，未越界的可以正常返回。
nebula> RETURN list[1, 2, 3, 4, 5] [0..10] AS a;
+-----+
| a |
+-----+
| [1, 2, 3, 4, 5] |
+-----+

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

# [0..0] 时返回为空。
nebula> RETURN list[1, 2, 3, 4, 5] [0..0] AS a;
+-----+
| a |
+-----+
| [] |
+-----+

# M ≥ N 时，返回为空。
nebula> RETURN list[1, 2, 3, 4, 5] [3..1] AS a;
+-----+
| a |
+-----+
| [] |
+-----+

# 范围查询时，下标有 null 时，返回为 null。
nebula> WITH list[1,2,3] AS a \
    RETURN a[0..null] as r;
+-----+
| r |
+-----+
| _NULL_ |
+-----+

# 将列表 [1,2,3,4,5] 中的元素分别做运算，然后将列表去掉表头并返回。

```

```

nebula> RETURN tail([n IN range(1, 5) | 2 * n - 10]) AS a;
+-----+
| a      |
+-----+
| [-6, -4, -2, 0] |
+-----+

# 将列表 [1,2,3] 中的元素判断为真，然后返回。
nebula> RETURN [n IN range(1, 3) WHERE true | n] AS r;
+-----+
| r      |
+-----+
| [1, 2, 3] |
+-----+

# 返回列表 [1,2,3] 的长度。
nebula> RETURN size(list[1,2,3]);
+-----+
| size([1,2,3]) |
+-----+
| 3            |
+-----+

# 将列表 [92,90] 中的元素做运算，然后在 where 子句中进行条件判断。
nebula> GO FROM "player100" OVER follow WHERE properties(edge).degree NOT IN [x IN [92, 90] | x + $$.player.age] \
    YIELD dst(edge) AS id, properties(edge).degree AS degree;
+-----+-----+
| id      | degree |
+-----+-----+
| "player101" | 95    |
| "player102" | 90    |
+-----+-----+

# 将 MATCH 语句的查询结果作为列表中的元素进行运算并返回。
nebula> MATCH p = (n:player{name:"Tim Duncan"})-[:follow]->(m) \
    RETURN [n IN nodes(p) | n.player.age + 100] AS r;
+-----+
| r      |
+-----+
| [142, 136] |
| [142, 141] |
+-----+

```

OpenCypher 兼容性

- 在 openCypher 中，查询越界元素时返回 null，而在 nGQL 中，查询单个越界元素时返回 OUT_OF_RANGE。

```

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

```

- 复合数据类型（例如 set、map、list）不能存储为点或边的属性。
- 建议修改图建模方式：将复合数据类型建模为点的邻边，而不是该点的自身属性，每条邻边可以动态增删，并且可以设置邻边的 Rank 值来控制邻边的顺序。
- List 中不支持 pattern，例如 [(src)-[]->(m) | m.name]。

最后更新: May 4, 2023

4.2.7 集合

集合（Set）是复合数据类型，集合中是一组元素，与列表（List）不同的是，集合中的元素是无序的，且不允许重复。

集合用左花括号（{）和右花括号（}）包裹多个元素，各个元素之间用英文逗号（,）隔开。元素前后的空格在集合中被忽略，因此可以使用换行符、制表符和空格调整格式。

OpenCypher 兼容性

- 复合数据类型（例如 List、Set、Map）不能存储为点或边的属性。
- 在 OpenCypher 中，集合不是一个数据类型，而在 nGQL 中，用户可以使用集合。

示例

```
# 返回集合 {1,2,3}。
nebula> RETURN set{1, 2, 3} AS a;
+-----+
| a      |
+-----+
| {3, 2, 1} |
+-----+

# 返回集合 {1,2,1}，因为集合不允许重复元素，会返回 {1,2}，且顺序是无序的。
nebula> RETURN set{1, 2, 1} AS a;
+-----+
| a      |
+-----+
| {2, 1} |
+-----+

# 判断集合中是否有指定元素 1。
nebula> RETURN 1 IN set{1, 2} AS a;
+-----+
| a      |
+-----+
| true   |
+-----+

# 计算集合中的元素数量。
nebula> YIELD size(set{1, 2, 1}) AS a;
+---+
| a |
+---+
| 2 |
+---+

# 返回目标点属性值组成的集合。
nebula> GO FROM "player100" OVER follow \
    YIELD set{properties($$).name,properties($$).age} as a;
+-----+
| a      |
+-----+
| {36, "Tony Parker"}  |
| {41, "Manu Ginobili"} |
+-----+
```

最后更新: May 4, 2023

4.2.8 映射

映射 (Map) 是复合数据类型。一个映射是一组键值对 (Key-Value) 的无序集合。在映射中, Key 是字符串类型, Value 可以是任何数据类型。用户可以通过 `map['<key>']` 的方法获取映射中的元素。

映射用左花括号 ({}) 和右花括号 (}) 包裹多个键值对, 各个键值对之间用英文逗号 (,) 隔开。键值对前后的空格在映射中被忽略, 因此可以使用换行符、制表符和空格调整格式。

OpenCypher 兼容性

- 复合数据类型 (例如 List、Set、Map) 不能存储为点或边的属性。
- 不支持映射投影 (map projection)。

示例

```
# 返回简单的映射。
nebula> YIELD map{key1: 'Value1', key2: 'Value2'} as a;
+-----+
| a |
+-----+
| {key2: "Value2", key1: "Value1"} |
+-----+

# 返回列表类型的映射。
nebula> YIELD map{listKey: [{inner: 'Map1'}, {inner: 'Map2'}]} as a;
+-----+
| a |
+-----+
| {listKey: [{inner: "Map1"}, {inner: "Map2"}]} |
+-----+

# 返回混合类型的映射。
nebula> RETURN map{a: LIST[1,2], b: SET{1,2,1}, c: "hee"} as a;
+-----+
| a |
+-----+
| {a: [1, 2], b: {2, 1}, c: "hee"} |
+-----+

# 返回映射中的指定元素。
nebula> RETURN map{a: LIST[1,2], b: SET{1,2,1}, c: "hee"}["b"] AS b;
+-----+
| b |
+-----+
| {2, 1} |
+-----+

# 判断映射中是否有指定key, 暂不支持判断value。
nebula> RETURN "a" IN MAP{a:1, b:2} AS a;
+-----+
| a |
+-----+
| true |
+-----+
```

最后更新: May 4, 2023

4.2.9 类型转换

类型转换是指将表达式的类型转换为另一个类型。

NebulaGraph支持显式地转换类型。详情参见[类型转换函数](#)。

示例

```
nebula> UNWIND [true, false, 'true', 'false', NULL] AS b \
    RETURN toBoolean(b) AS b;
+-----+
| b      |
+-----+
| true   |
+-----+
| false  |
+-----+
| true   |
+-----+
| false  |
+-----+
| _NULL_ |
+-----+  
  
nebula> RETURNtoFloat(1), toFloat('1.3'), toFloat('1e3'), toFloat('not a number');
+-----+-----+-----+-----+
| toFloat(1) | toFloat("1.3") | toFloat("1e3") | toFloat("not a number") |
+-----+-----+-----+-----+
| 1.0       | 1.3        | 1000.0     | _NULL_     |
+-----+-----+-----+-----+
```

最后更新: May 4, 2023

4.2.10 地理位置

地理位置（GEOGRAPHY）是由经纬度构成的表示地理空间信息的数据类型。NebulaGraph当前支持[简单地理要素](#)中的 Point、LineString 和 Polygon 三种地理形状。支持 [SQL-MM 3](#) 中的部分核心 geo 解析、构造、格式设置、转换、谓词和度量等函数。

GEOGRAPHY

GEOGRAPHY 的基本类型是点，由经纬度确定一个点，例如 "POINT(3 8)" 表示经度为 3°，纬度为 8°。多个点可以构成线段或多边形。

类型	示例	说明
Point	"POINT(3 8)"	点类型
LineString	"LINESTRING(3 8, 4.7 73.23)"	线段类型
Polygon	"POLYGON((0 1, 1 2, 2 3, 0 1))"	多边形类型

Note

请勿直接插入上述类型的 GEOGRAPHY 数据，例如 `INSERT VERTEX any_shape(geo) VALUES "1":("POINT(1 1)")`，需要使用 [geo 函数](#)指定数据类型后才能插入，例如 `INSERT VERTEX any_shape(geo) VALUES "1":(ST_GeogFromText("POINT(1 1)"));`。

示例

```
//创建 Tag, 允许存储任意形状地理位置数据类型。
nebula> CREATE TAG IF NOT EXISTS any_shape(geo geography);

//创建 Tag, 只允许存储点形状地理位置数据类型。
nebula> CREATE TAG IF NOT EXISTS only_point(geo geography(point));

//创建 Tag, 只允许存储线段形状地理位置数据类型。
nebula> CREATE TAG IF NOT EXISTS only_linestring(geo geography(linestring));

//创建 Tag, 只允许存储多边形形状地理位置数据类型。
nebula> CREATE TAG IF NOT EXISTS only_polygon(geo geography(polygon));

//创建 Edge type, 允许存储任意形状地理位置数据类型。
nebula> CREATE EDGE IF NOT EXISTS any_shape_edge(geo geography);

//创建存储多边形地理位置的点。
nebula> INSERT VERTEX any_shape(geo) VALUES "103":(ST_GeogFromText("POLYGON((0 1, 1 2, 2 3, 0 1))"));

//创建存储多边形地理位置的边。
nebula> INSERT EDGE any_shape_edge(geo) VALUES "201"->"302":(ST_GeogFromText("POLYGON((0 1, 1 2, 2 3, 0 1))"));

//查询点 103 的属性 geo。
nebula> FETCH PROP ON any_shape "103" YIELD ST_ASText(any_shape.geo);
+-----+
| ST_ASText(any_shape.geo) |
+-----+
| "POLYGON((0 1, 1 2, 2 3, 0 1))" |
+-----+

//查询边 201->302 的属性 geo。
nebula> FETCH PROP ON any_shape_edge "201"->"302" YIELD ST_ASText(any_shape_edge.geo);
+-----+
| ST_ASText(any_shape_edge.geo) |
+-----+
| "POLYGON((0 1, 1 2, 2 3, 0 1))" |
+-----+

//为 geo 属性创建索引并使用 LOOKUP 查询。
nebula> CREATE TAG INDEX IF NOT EXISTS any_shape_geo_index ON any_shape(geo);
nebula> REBUILD TAG INDEX any_shape_geo_index;
nebula> LOOKUP ON any_shape YIELD ST_ASText(any_shape.geo);
+-----+
| ST_ASText(any_shape.geo) |
+-----+
| "POLYGON((0 1, 1 2, 2 3, 0 1))" |
+-----+
```

为 geo 属性创建索引时，还可以指定 geo 索引的参数。说明如下。

参数	默认值	说明
s2_max_level	30	S2 cell 用于填充的最大等级。取值：1 ~ 30。设置为小于默认值时，意味着会使用较大的单元格进行填充。
s2_max_cells	8	S2 cell 用于填充的最大数量，可以限制填充的工作量。取值：1 ~ 30。对于复杂形状的区域（例如细矩形），可以使用更大的值。

Note

指定如上两个参数对 Point 类型属性没有影响，Point 类型属性的 s2_max_level 强制为 30。

```
nebula> CREATE TAG INDEX IF NOT EXISTS any_shape_geo_index ON any_shape(geo) with (s2_max_level=30, s2_max_cells=8);
```

最后更新: May 4, 2023

4.3 变量和复合查询

4.3.1 复合查询（子句结构）

复合查询将来自不同请求的数据放在一起，然后进行过滤、分组或者排序等，最后返回结果。

NebulaGraph 支持三种方式进行复合查询（或子查询）：

- (`openCypher` 兼容语句) 连接各个子句，让它们在彼此之间提供中间结果集。
- (原生 nGQL) 多个查询可以合并处理，以英文分号 (;) 分隔，返回最后一个查询的结果。
- (原生 nGQL) 可以用管道符 (|) 将多个查询连接起来，上一个查询的结果可以作为下一个查询的输入。

OpenCypher 兼容性

在复合查询中，请不要混用 `openCypher` 兼容语句和原生 nGQL 语句，例如 `MATCH ... | GO ... | YIELD ...`。

- 如果使用 `openCypher` 兼容语句 (`MATCH`、`RETURN`、`WITH` 等)，请不要使用管道符或分号组合子句。
- 如果使用原生 nGQL 语句 (`FETCH`、`GO`、`LOOKUP` 等)，必须使用管道符或分号组合子句。

复合查询不支持事务

例如一个查询由三个子查询 A、B、C 组成，A 是一个读操作，B 是一个计算操作，C 是一个写操作，如果在执行过程中，任何一个操作执行失败，则整个结果是未定义的：没有回滚，而且写入的内容取决于执行程序。



`openCypher` 没有事务要求。

示例

- `openCypher` 兼容语句

```
# 子句连接多个查询。
nebula> MATCH p=(v:player{name:"Tim Duncan"})--() \
    WITH nodes(p) AS n \
```

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

- 原生 nGQL（分号）

```
# 只返回边。
nebula> SHOW TAGS; SHOW EDGES;

# 插入多个点。
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42); \
    INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36); \
    INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);
```

- 原生 nGQL（管道符）

```
# 管道符连接多个查询。
nebula> GO FROM "player100" OVER follow YIELD dst(edge) AS id | \
    GO FROM $-.id OVER serve YIELD properties($$).name AS Team, \
    properties($$).name AS Player;
+-----+-----+
| Team | Player |
+-----+-----+
| "Spurs" | "Tony Parker" |
| "Hornets" | "Tony Parker" |
| "Spurs" | "Manu Ginobili" |
+-----+-----+
```

最后更新: May 4, 2023

4.3.2 自定义变量

NebulaGraph允许将一条语句的结果作为自定义变量传递给另一条语句。

OpenCypher 兼容性

当引用一个变量的点、边或路径，需要先给它命名。例如：

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

示例中的 v 就是自定义变量。



在同一个 MATCH 语句的模式中，不能重复使用边变量。例如 e 不能重复被写在模式 p=(v1)-[e*2..2]->(v2)-[e*2..2]->(v3) 中。

原生 nGQL

nGQL 扩展的自定义变量可以表示为 \$var_name， var_name 由字母、数字或下划线（_）构成，不允许使用其他字符。

自定义变量仅在当前执行（本复合查询中）有效，执行结束后变量也会释放，不能在其他客户端、执行、session 中使用之前的自定义变量。

用户可以在复合查询中使用自定义变量。复合查询的详细信息请参见[复合查询](#)。



- 自定义变量区分大小写。
- 在包含扩展的自定义变量的复合语句中，用英文分号 ; 结束定义变量的语句。详情参见 [nGQL 风格指南](#)。

示例

```
nebula> $var = GO FROM "player100" OVER follow YIELD dst(edge) AS id; \
GO FROM $var.id OVER serve YIELD properties($$).name AS Team, \
properties($$).name AS Player;
+-----+
| Team | Player |
+-----+
| "Spurs" | "Tony Parker" |
| "Hornets" | "Tony Parker" |
| "Spurs" | "Manu Ginobili" |
+-----+
```

最后更新: May 4, 2023

4.3.3 引用属性

用户可以在 WHERE 和 YIELD 子句中引用点或边的属性。

Note

本功能仅适用于原生 nGQL 的 GO 语句。

引用点的属性

起始点

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

参数	说明
\$^	起始点
tag_name	点的 Tag 名称
prop_name	Tag 内的属性名称

目的点

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

参数	说明
\$\$	目的点
tag_name	点的 Tag 名称
prop_name	Tag 内的属性名称

引用边的属性

引用自定义的边属性

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

参数	说明
edge_type	Edge type
prop_name	Edge type 的属性名称

引用内置的边属性

除了自定义的边属性，每条边还有如下四种内置属性：

参数	说明
_src	边的起始点
_dst	边的目的点
_type	边的类型内部编码，正负号表示方向：正数为正向边，负数为逆向边
_rank	边的 rank 值

示例

```
# 返回起始点的 Tag player 的 name 属性值和目的点的 Tag player 的 age 属性值。
nebula> GO FROM "player100" OVER follow YIELD $^.player.name AS startName, $$ .player.age AS endAge;
+-----+-----+
| startName | endAge |
+-----+-----+
| "Tim Duncan" | 36 |
| "Tim Duncan" | 41 |
+-----+-----+

# 返回 Edge type follow 的 degree 属性值。
nebula> GO FROM "player100" OVER follow YIELD follow.degree;
+-----+
| follow.degree |
+-----+
| 95 |
+-----+

# 返回 EdgeType 是 follow 的起始点 VID、目的点 VID、EdgeType 编码（正数为正向边，负数为逆向边），和边的 rank 值。
nebula> GO FROM "player100" OVER follow YIELD follow._src, follow._dst, follow._type, follow._rank;
+-----+-----+-----+-----+
| follow._src | follow._dst | follow._type | follow._rank |
+-----+-----+-----+-----+
| "player100" | "player101" | 17 | 0 |
| "player100" | "player125" | 17 | 0 |
+-----+-----+-----+-----+
```

版本兼容性

从NebulaGraph 2.6.0 起支持了新的 [Schema 相关函数](#)。以上示例在NebulaGraph 3.4.1 中的近似写法如下：

```
GO FROM "player100" OVER follow YIELD properties($^).name AS startName, properties($$).age AS endAge;
GO FROM "player100" OVER follow YIELD properties(edge).degree;
GO FROM "player100" OVER follow YIELD src(edge), dst(edge), type(edge), rank(edge);
```

NebulaGraph 3.4.1 兼容新旧语法。

最后更新: May 4, 2023

4.4 运算符

4.4.1 比较符

NebulaGraph支持的比较符如下。

符号	说明
=	赋值
+	加法
-	减法
*	乘法
/	除法
==	相等
!=, <>	不等于
>	大于
>=	大于等于
<	小于
<=	小于等于
%	取模
-	负数符号
IS NULL	为 NULL
IS NOT NULL	不为 NULL
IS EMPTY	不存在
IS NOT EMPTY	存在

比较操作的结果是 `true` 或者 `false`。

Note

- 比较不同类型的值通常没有定义，结果可能是 `NULL` 或其它。
- `EMPTY` 当前仅用于判断，不支持函数或者运算操作，包括且不限于 `GROUP BY`、`count()`、`sum()`、`max()`、`hash()`、`collect()`、`+`、`*`。

OpenCypher 兼容性

openCypher 中没有 `EMPTY`，因此不支持在 `MATCH` 语句中使用 `EMPTY`。

示例

==

字符串比较时，会区分大小写。不同类型的值不相等。

Note

nGQL 中的相等符号是 `==`，openCypher 中的相等符号是 `=`。

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

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

>

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

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

>=

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

<

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

<=

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

!=

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

IS [NOT] NULL

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

```
nebula> RETURN length(NULL), size(NULL), count(NULL), NULL IS NULL, NULL IS NOT NULL, sin(NULL), NULL + NULL, [1, NULL] IS NULL;
+-----+-----+-----+-----+-----+-----+-----+-----+
| length(NULL) | size(NULL) | count(NULL) | NULL IS NULL | NULL IS NOT NULL | sin(NULL) | (NULL+NULL) | [1,NULL] IS NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+
| __NULL__ | __NULL__ | 0 | true | false | __NULL__ | __NULL__ | false |
+-----+-----+-----+-----+-----+-----+-----+-----+
nebula> WITH {name: null} AS `map` \
    RETURN `map`.name IS NOT NULL;
+-----+
| map.name IS NOT NULL |
+-----+
| false |
+-----+
nebula> WITH {name: 'Mats', name2: 'Pontus'} AS map1, \
    {name: null} AS map2, {notName: 0, notName2: null } AS map3 \
    RETURN map1.name IS NULL, map2.name IS NOT NULL, map3.name IS NULL;
+-----+-----+-----+
| map1.name IS NULL | map2.name IS NOT NULL | map3.name IS NULL |
+-----+-----+-----+
| false | false | true |
+-----+-----+-----+
nebula> MATCH (n:player) \
    RETURN n.player.age IS NULL, n.player.name IS NOT NULL, n.player.empty IS NULL;
+-----+-----+-----+
| n.player.age IS NULL | n.player.name IS NOT NULL | n.player.empty IS NULL |
+-----+-----+-----+
| false | true | true |
| false | true | true |
...
...
```

IS [NOT] EMPTY

```
nebula> RETURN null IS EMPTY;
+-----+
| NULL IS EMPTY |
+-----+
| false          |
+-----+  
  
nebula> RETURN "a" IS NOT EMPTY;
+-----+
| "a" IS NOT EMPTY |
+-----+
| true           |
+-----+  
  
nebula> GO FROM "player100" OVER * WHERE properties($$).name IS NOT EMPTY YIELD dst(edge);
+-----+
| dst(EDGE)   |
+-----+
| "team204"   |
| "player101" |
| "player125" |
+-----+
```

最后更新: May 4, 2023

4.4.2 布尔符

NebulaGraph支持的布尔符如下。

符号	说明
AND	逻辑与
OR	逻辑或
NOT	逻辑非
XOR	逻辑异或

对于以上运算的优先级, 请参见[运算优先级](#)。

对于带有 NULL 的逻辑运算, 请参见 [NULL](#)。



非 0 数字不能转换为布尔值。

最后更新: May 4, 2023

4.4.3 管道符

nGQL 支持使用管道符 (|) 将多个查询组合起来。

openCypher 兼容性

管道符仅适用于原生 nGQL。

语法

nGQL 和 SQL 之间的一个主要区别是子查询的组成方式。

- 在 SQL 中，子查询是嵌套在查询语句中的。
- 在 nGQL 中，子查询是通过类似 shell 中的管道符 (|) 实现的。

示例

```
nebula> GO FROM "player100" OVER follow \
    YIELD dst(edge) AS dstid, properties($$).name AS Name | \
    GO FROM $-.dstid OVER follow YIELD dst(edge);

+-----+
| dst(EDGE) |
+-----+
| "player100" |
| "player102" |
| "player125" |
| "player100" |
+-----+
```

必须在 YIELD 子句中为需要的返回结果设置别名，才能在管道符右侧使用引用符 \$-，例如示例中的 \$-.dstid。

性能提示

NebulaGraph 中的管道对性能有影响，以 A | B 为例，体现在以下几个方面：

1. 管道是同步操作。也即需要管道之前的子句 A 执行完毕后，数据才能整体进入管道子句。
2. 管道本身是需要序列化和反序列化的，这个是单线程执行的。
3. 如果 A 发大量数据给 |，整个查询请求的总体时延可能会非常大。此时可以尝试拆分这个语句：
 - a. 应用程序发送 A，
 - b. 将收到的返回结果在应用程序拆分，
 - c. 并发发送给多个 graphd，
 - d. 每个 graphd 执行部分 B。

这样通常比单个 graphd 执行完整地 A | B 要快很多。

最后更新: May 4, 2023

4.4.4 引用符

nGQL 提供引用符来表示 WHERE 和 YIELD 子句中的属性，或者复合查询中管道符之前的语句输出结果。

openCypher 兼容性

引用符仅适用于原生 nGQL。

引用符列表

引用符	说明
\$^	引用起始点。更多信息请参见 引用属性 。
\$\$	引用目的点。更多信息请参见 引用属性 。
\$-	引用复合查询中管道符之前的语句输出结果。更多信息请参见 管道符 。

示例

```
# 返回起始点和目的点的年龄。
nebula> GO FROM "player100" OVER follow YIELD properties($^).age AS SrcAge, properties($$).age AS DestAge;
+-----+-----+
| SrcAge | DestAge |
+-----+-----+
| 42     | 36      |
| 42     | 41      |
+-----+-----+

# 返回 player100 追随的 player 的名称和团队。
nebula> GO FROM "player100" OVER follow \
    YIELD dst(edge) AS id | \
    GO FROM $-.id OVER serve \
    YIELD properties($^).name AS Player, properties($$).name AS Team;
+-----+-----+
| Player   | Team    |
+-----+-----+
| "Tony Parker" | "Spurs" |
| "Tony Parker" | "Hornets" |
| "Manu Ginobili" | "Spurs" |
+-----+-----+
```

最后更新: May 4, 2023

4.4.5 集合运算符

合并多个请求时，可以使用集合运算符，包括 `UNION`、`UNION ALL`、`INTERSECT` 和 `MINUS`。

所有集合运算符的优先级相同，如果一个 nGQL 语句中有多个集合运算符，NebulaGraph 会从左到右进行计算，除非用括号指定顺序。



集合运算符前后的查询语句中定义的变量名及顺序必需保持一致，例如 `RETURN a,b,c UNION RETURN a,b,c` 中的 `a,b,c` 的名称及顺序需要保持一致。

UNION、UNION DISTINCT、UNION ALL

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

- 运算符 `UNION DISTINCT`（或使用缩写 `UNION`）返回两个集合 A 和 B 的并集，不包含重复的元素。
- 运算符 `UNION ALL` 返回两个集合 A 和 B 的并集，包含重复的元素。
- `left` 和 `right` 必须有相同数量的列和数据类型。如果需要转换数据类型，请参见[类型转换](#)。

示例

```
# 返回两个查询结果的并集，不包含重复的元素。
nebula> GO FROM "player102" OVER follow YIELD dst(edge) \
    UNION \
    GO FROM "player100" OVER follow YIELD dst(edge);
+-----+
| dst(EDGE) |
+-----+
| "player100" |
| "player101" |
| "player125" |
+-----+

nebula> MATCH (v:player) \
    WITH v.player.name AS n \
    RETURN n ORDER BY n LIMIT 3 \
    UNION \
    UNWIND ["Tony Parker", "Ben Simmons"] AS n \
    RETURN n;
+-----+
| n |
+-----+
| "Amar'e Stoudemire" |
| "Aron Baynes" |
| "Ben Simmons" |
| "Tony Parker" |
+-----+

# 返回两个查询结果的并集，包含重复的元素。
nebula> GO FROM "player102" OVER follow YIELD dst(edge) \
    UNION ALL \
    GO FROM "player100" OVER follow YIELD dst(edge);
+-----+
| dst(EDGE) |
+-----+
| "player100" |
| "player101" |
| "player101" |
| "player125" |
+-----+

nebula> MATCH (v:player) \
    WITH v.player.name AS n \
    RETURN n ORDER BY n LIMIT 3 \
    UNION ALL \
    UNWIND ["Tony Parker", "Ben Simmons"] AS n \
    RETURN n;
+-----+
| n |
+-----+
| "Amar'e Stoudemire" |
| "Aron Baynes" |
| "Ben Simmons" |
| "Tony Parker" |
| "Ben Simmons" |
+-----+

# UNION 也可以和 YIELD 语句一起使用，去重时会检查每一行的所有列，每列都相同时才会去重。

```

```
nebula> GO FROM "player102" OVER follow \
    YIELD dst(edge) AS id, properties(edge).degree AS Degree, properties($$).age AS Age \
    UNION /* DISTINCT */ \
    GO FROM "player100" OVER follow \
    YIELD dst(edge) AS id, properties(edge).degree AS Degree, properties($$).age AS Age;
+-----+-----+-----+
| id | Degree | Age |
+-----+-----+-----+
| "player100" | 75 | 42 |
| "player101" | 75 | 36 |
| "player101" | 95 | 36 |
| "player125" | 95 | 41 |
+-----+-----+-----+
```

INTERSECT

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

- 运算符 `INTERSECT` 返回两个集合 A 和 B 的交集。
- `left` 和 `right` 必须有相同数量的列和数据类型。如果需要转换数据类型，请参见[类型转换](#)。

示例

```
# 返回两个查询结果的交集。
nebula> GO FROM "player102" OVER follow \
    YIELD dst(edge) AS id, properties(edge).degree AS Degree, properties($$).age AS Age \
    INTERSECT \
    GO FROM "player100" OVER follow \
    YIELD dst(edge) AS id, properties(edge).degree AS Degree, properties($$).age AS Age;
+-----+-----+
| id | Degree | Age |
+-----+-----+
+-----+-----+
nebula> MATCH (v:player)-[e:follow]->(v2) \
    WHERE id(v) = "player102" \
    RETURN id(v2) AS id, e.degree AS Degree, v2.player.age AS Age \
    INTERSECT \
    MATCH (v:player)-[e:follow]->(v2) \
    WHERE id(v) = "player100" \
    RETURN id(v2) AS id, e.degree AS Degree, v2.player.age AS Age;
+-----+-----+
| id | Degree | Age |
+-----+-----+
+-----+-----+
nebula> UNWIND [1,2] AS a RETURN a \
    INTERSECT \
    UNWIND [1,2,3,4] AS a \
    RETURN a;
+---+
| a |
+---+
| 1 |
| 2 |
+---+
```

MINUS

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

运算符 `MINUS` 返回两个集合 A 和 B 的差异，即 $A - B$ 。请注意 `left` 和 `right` 的顺序， $A - B$ 表示在集合 A 中，但是不在集合 B 中的元素。

示例

```
# 返回在第一个查询结果中，但是不在第二个查询结果中的元素。
nebula> GO FROM "player100" OVER follow YIELD dst(edge) \
    MINUS \
    GO FROM "player102" OVER follow YIELD dst(edge);
+-----+
| dst(EDGE) |
+-----+
| "player125" |
+-----+
nebula> GO FROM "player102" OVER follow YIELD dst(edge) AS id \
    MINUS \
    GO FROM "player100" OVER follow YIELD dst(edge) AS id;
+-----+
| id |
+-----+
| "player100" |
```

```
+-----+
nebula> MATCH (v:player)-[e:follow]->(v2) \
    WHERE id(v) == "player102" \
    RETURN id(v2) AS id \
    MINUS \
    MATCH (v:player)-[e:follow]->(v2) \
    WHERE id(v) == "player100" \
    RETURN id(v2) AS id;
+-----+
| id |
+-----+
| "player100" |
+-----+
```



```
+-----+
nebula> UNWIND [1,2,3] AS a RETURN a \
    MINUS \
    WITH 4 AS a \
    RETURN a;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

集合运算符和管道符的优先级

当查询包含集合运算符和管道符 (|) 时，**管道符**的优先级高。例如 GO FROM 1 UNION GO FROM 2 | GO FROM 3 相当于 GO FROM 1 UNION (GO FROM 2 | GO FROM 3)。

示例

```
+-----+
nebula> GO FROM "player102" OVER follow \
    YIELD dst(edge) AS play_dst \
    UNION \
    GO FROM "team200" OVER serve REVERSELY \
    YIELD src(edge) AS play_src \
    | GO FROM $-.play_src OVER follow YIELD dst(edge) AS play_dst;
+-----+
| play_dst |
+-----+
| "player100" |
| "player101" |
| "player117" |
| "player105" |
+-----+
```

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

该查询会先执行红框内的语句，然后执行绿框的 UNION 操作。

圆括号可以修改执行的优先级，例如：

```
+-----+
nebula> (GO FROM "player102" OVER follow \
    YIELD dst(edge) AS play_dst \
    UNION \
    GO FROM "team200" OVER serve REVERSELY \
    YIELD src(edge) AS play_dst) \
    | GO FROM $-.play_dst OVER follow YIELD dst(edge) AS play_dst;
```

该查询中，圆括号包裹的部分先执行，即先执行 UNION 操作，再将结果结合管道符进行下一步操作。

最后更新: May 4, 2023

4.4.6 字符串运算符

NebulaGraph支持使用字符串运算符进行连接、搜索、匹配运算。支持的运算符如下。

名称	说明
+	连接字符串。
CONTAINS	在字符串中执行搜索。
(NOT) IN	字符串是否匹配某个值。
(NOT) STARTS WITH	在字符串的开头执行匹配。
(NOT) ENDS WITH	在字符串的结尾执行匹配。
正则表达式	通过正则表达式匹配字符串。

Note

所有搜索或匹配都区分大小写。

示例

+

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

CONTAINS

CONTAINS 要求待运算的左右两边都是字符串类型。

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

(NOT) IN

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

true	true	__NULL__	
------	------	----------	--

(NOT) STARTS WITH

```
nebula> RETURN 'apple' STARTS WITH 'app', 'apple' STARTS WITH 'a', 'apple' STARTS WITH toUpper('a');
+-----+-----+-----+
| ("apple" STARTS WITH "app") | ("apple" STARTS WITH "a") | ("apple" STARTS WITH toUpper("a")) |
+-----+-----+-----+
| true          | true          | false         |
+-----+-----+-----+
```



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

(NOT) ENDS WITH

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

正则表达式



当前仅 `opencypher` 兼容语句（`MATCH`、`WITH` 等）支持正则表达式，原生 `nGQL` 语句（`FETCH`、`GO`、`LOOKUP` 等）不支持正则表达式。

`NebulaGraph` 支持使用正则表达式进行过滤，正则表达式的语法是继承自 `std::regex`，用户可以使用语法 `=~ '<regexp>'` 进行正则表达式匹配。例如：

```
nebula> RETURN "384748.39" =~ "\d+(\.\d{2})?";
+-----+
| ("384748.39"=~"\d+(\.\d{2})?") |
+-----+
| true          |
+-----+
```



```
nebula> MATCH (v:player) WHERE v.player.name =~ 'Tony.*' RETURN v.player.name;
+-----+
| v.player.name |
+-----+
| "Tony Parker" |
+-----+
```

最后更新: May 4, 2023

4.4.7 列表运算符

NebulaGraph支持使用列表（List）运算符进行运算。支持的运算符如下。

名称	说明
+	连接列表。
IN	元素是否存在于列表中。
[]	使用下标操作符访问列表中的元素。

示例

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

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

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

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

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

最后更新: May 4, 2023

4.4.8 运算符优先级

nGQL 运算符的优先级从高到低排列如下（同一行的运算符优先级相同）：

- - (负数)
- ! 、 NOT
- * 、 / 、 %
- - 、 +
- == 、 >= 、 > 、 <= 、 < 、 <> 、 !=
- AND
- OR 、 XOR
- = (赋值)

如果表达式中有相同优先级的运算符，运算是从左到右进行，只有赋值操作是例外（从右到左运算）。

运算符的优先级决定运算的顺序，要显式修改运算顺序，可以使用圆括号。

示例

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

openCypher 兼容性

在 openCypher 中，比较操作可以任意连接，例如 $x < y \leq z$ 等价于 $x < y \text{ AND } y \leq z$ 。

在 nGQL 中， $x < y \leq z$ 等价于 $(x < y) \leq z$ ， $(x < y)$ 的结果是一个布尔值，再将布尔值和 z 比较，最终结果是 NULL。

最后更新: May 4, 2023

4.5 函数和表达式

4.5.1 内置数学函数

本文介绍NebulaGraph支持的数学函数。

abs()

abs() 返回指定数字的绝对值。

语法: `abs(<expression>)`

- **expression**: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN abs(-10);
+-----+
| abs(-(10)) |
+-----+
| 10          |
+-----+
```



```
nebula> RETURN abs(5-6);
+-----+
| abs((5-6)) |
+-----+
| 1           |
+-----+
```

floor()

floor() 返回小于或等于指定数字的最大整数。

语法: `floor(<expression>)`

- **expression**: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN floor(9.9);
+-----+
| floor(9.9) |
+-----+
| 9.0          |
+-----+
```

ceil()

ceil() 返回大于或等于指定数字的最小整数。

语法: `ceil(<expression>)`

- **expression**: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN ceil(9.1);
+-----+
| ceil(9.1) |
+-----+
```

```
+-----+
| 10.0 |
+-----+
```

round()

round() 返回指定数字四舍五入后的值。极端情况下请注意浮点数的精度问题。

语法: `round(<expression>, <digit>)`

- **expression**: 结果的数据类型为 `double` 的表达式。
- **digit**: 小数位数。小于 0 时，在小数点左侧做四舍五入。数据类型为 `int`。
- 返回类型: `double`。

示例:

```
nebula> RETURN round(314.15926, 2);
+-----+
| round(314.15926, 2) |
+-----+
| 314.16               |
+-----+

nebula> RETURN round(314.15926, -1);
+-----+
| round(314.15926, -1) |
+-----+
| 310.0                |
+-----+
```

sqrt()

sqrt() 返回指定数字的平方根。

语法: `sqrt(<expression>)`

- **expression**: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN sqrt(9);
+-----+
| sqrt(9) |
+-----+
| 3.0      |
+-----+
```

cbrt()

cbrt() 返回指定数字的立方根。

语法: `cbrt(<expression>)`

- **expression**: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN cbrt(8);
+-----+
| cbrt(8) |
+-----+
| 2.0      |
+-----+
```

hypot()

hypot() 返回直角三角形的斜边长。

语法: `hypot(<expression_x>,<expression_y>)`

- `expression_x`、`expression_y`: 结果的数据类型为 `double` 的表达式。表示直角三角形的边长 `x` 和 `y`。
- 返回类型: `double`。

示例:

```
nebula> RETURN hypot(3,2*2);
+-----+
| hypot(3,(2*2)) |
+-----+
| 5.0             |
+-----+
```

pow()

pow() 返回指定数字的幂 (x^y)。

语法: `pow(<expression_x>,<expression_y>,)`

- `expression_x`: 结果的数据类型为 `double` 的表达式。表示底数 `x`。
- `expression_y`: 结果的数据类型为 `double` 的表达式。表示指数 `y`。
- 返回类型: `double`。

示例:

```
nebula> RETURN pow(3,3);
+-----+
| pow(3,3) |
+-----+
| 27         |
+-----+
```

exp()

exp() 返回自然常数 `e` 的幂 (e^x)。

语法: `exp(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。表示指数 `x`。
- 返回类型: `double`。

示例:

```
nebula> RETURN exp(2);
+-----+
| exp(2)          |
+-----+
| 7.38905609893065 |
+-----+
```

exp2()

exp2() 返回2的幂 (2^x)。

语法: `exp2(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。表示指数 `x`。
- 返回类型: `double`。

示例:

```
nebula> RETURN exp2(3);
+-----+
| exp2(3) |
+-----+
| 8.0     |
+-----+
```

log()

log() 返回以自然数 e 为底的对数 $(\log_e N)$ 。

语法: `log(<expression>)`

- **expression**: 结果的数据类型为 `double` 的表达式。表示真数 N。
- 返回类型: `double`。

示例:

```
nebula> RETURN log(8);
+-----+
| log(8)   |
+-----+
| 2.0794415416798357 |
+-----+
```

log2()

log2() 返回以 2 为底的对数 $(\log_2 N)$ 。

语法: `log2(<expression>)`

- **expression**: 结果的数据类型为 `double` 的表达式。表示真数 N。
- 返回类型: `double`。

示例:

```
nebula> RETURN log2(8);
+-----+
| log2(8) |
+-----+
| 3.0     |
+-----+
```

log10()

log10() 返回以 10 为底的对数 $(\log_{10} N)$ 。

语法: `log10(<expression>)`

- **expression**: 结果的数据类型为 `double` 的表达式。表示真数 N。
- 返回类型: `double`。

示例:

```
nebula> RETURN log10(100);
+-----+
| log10(100) |
+-----+
| 2.0        |
+-----+
```

sin()

sin() 返回指定数字的正弦值。可以使用函数 `radians()` 将角度转化为弧度。

语法: `sin(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN sin(3);
+-----+
| sin(3)      |
+-----+
| 0.1411200080598672 |
+-----+
```

asin()

`asin()` 返回指定数字的反正弦值。可以使用函数 `radians()` 将角度转化为弧度。

语法: `asin(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN asin(0.5);
+-----+
| asin(0.5)      |
+-----+
| 0.5235987755982989 |
+-----+
```

cos()

`cos()` 返回指定数字的余弦值。可以使用函数 `radians()` 将角度转化为弧度。

语法: `cos(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN cos(0.5);
+-----+
| cos(0.5)      |
+-----+
| 0.8775825618903728 |
+-----+
```

acos()

`acos()` 返回指定数字的反余弦值。可以使用函数 `radians()` 将角度转化为弧度。

语法: `acos(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN acos(0.5);
+-----+
| acos(0.5)      |
+-----+
```

```
| 1.0471975511965979 |
+-----+
```

tan()

tan() 返回指定数字的正切值。可以使用函数 `radians()` 将角度转化为弧度。

语法: `tan(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN tan(0.5);
+-----+
| tan(0.5)      |
+-----+
| 0.5463024898437905 |
+-----+
```

atan()

atan() 返回指定数字的反正切值。可以使用函数 `radians()` 将角度转化为弧度。

语法: `atan(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。
- 返回类型: `double`。

示例:

```
nebula> RETURN atan(0.5);
+-----+
| atan(0.5)      |
+-----+
| 0.4636476090008061 |
+-----+
```

rand()

rand() 返回 [0,1) 内的随机浮点数。

语法: `rand()`

- 返回类型: `double`。

示例:

```
nebula> RETURN rand();
+-----+
| rand()      |
+-----+
| 0.6545837172298736 |
+-----+
```

rand32()

rand32() 返回指定范围 (`[min, max]`) 内的随机 32 位整数。

语法: `rand32(<expression_min>,<expression_max>)`

- `expression_min`: 结果的数据类型为 `int` 的表达式。表示最小值 `min`。
- `expression_max`: 结果的数据类型为 `int` 的表达式。表示最大值 `max`。
- 返回类型: `int`。
- 用户可以只传入一个参数, 该参数会判定为 `max`, 此时 `min` 默认为 0。如果不传入参数, 此时会从带符号的 32 位 `int` 范围内随机返回。

示例:

```
nebula> RETURN rand32(1,100);
+-----+
| rand32(1,100) |
+-----+
| 63           |
+-----+
```

rand64()

`rand64()` 返回指定范围 (`[min, max]`) 内的随机 64 位整数。

语法: `rand64(<expression_min>,<expression_max>)`

- `expression_min`: 结果的数据类型为 `int` 的表达式。表示最小值 `min`。
- `expression_max`: 结果的数据类型为 `int` 的表达式。表示最大值 `max`。
- 返回类型: `int`。
- 用户可以只传入一个参数, 该参数会判定为 `max`, 此时 `min` 默认为 0。如果不传入参数, 此时会从带符号的 64 位 `int` 范围内随机返回。

示例:

```
nebula> RETURN rand64(1,100);
+-----+
| rand64(1,100) |
+-----+
| 34           |
+-----+
```

bit_and()

`bit_and()` 返回按位进行 AND 运算后的结果。

语法: `bit_and(<expression_1>,<expression_2>)`

- `expression_1`、`expression_2`: 结果的数据类型为 `int` 的表达式。
- 返回类型: `int`。

示例:

```
nebula> RETURN bit_and(5,6);
+-----+
| bit_and(5,6) |
+-----+
| 4           |
+-----+
```

bit_or()

`bit_or()` 返回按位进行 OR 运算后的结果。

语法: `bit_or(<expression_1>,<expression_2>)`

- `expression_1`、`expression_2`: 结果的数据类型为 `int` 的表达式。
- 返回类型: `int`。

示例:

```
nebula> RETURN bit_or(5,6);
+-----+
| bit_or(5,6) |
+-----+
| 7           |
+-----+
```

bit_xor()

`bit_xor()` 返回按位进行 XOR 运算后的结果。

语法: `bit_xor(<expression_1>,<expression_2>)`

- `expression_1`、`expression_2`: 结果的数据类型为 `int` 的表达式。
- 返回类型: `int`。

示例:

```
nebula> RETURN bit_xor(5,6);
+-----+
| bit_xor(5,6) |
+-----+
| 3           |
+-----+
```

size()

`size()` 返回列表或映射中元素的数量，或者返回字符串的长度。

语法: `size({<expression>}|<string>)`

- `expression`: 列表或映射的表达式。
- `string`: 指定的字符串。
- 返回类型: `int`。

示例:

```
nebula> RETURN size([1,2,3,4]);
+-----+
| size([1,2,3,4]) |
+-----+
| 4           |
+-----+
```

```
nebula> RETURN size("basketballplayer") as size;
+-----+
| size |
+-----+
| 16   |
+-----+
```

range()

`range()` 返回指定范围 (`[start,end]`) 中指定步长的值组成的列表。

语法: `range(<expression_start>,<expression_end>[,<expression_step>])`

- `expression_start`: 结果的数据类型为 `int` 的表达式。表示起始值 `start`。
- `expression_end`: 结果的数据类型为 `int` 的表达式。表示结束值 `end`。
- `expression_step`: 结果的数据类型为 `int` 的表达式。表示步长 `step`，默认值为 1。
- 返回类型: `list`。

示例:

```
nebula> RETURN range(1,3*3,2);
+-----+
| range(1,(3*3),2) |
+-----+
| [1, 3, 5, 7, 9] |
+-----+
```

sign()

`sign()` 返回指定数字的正负号。如果数字为 0，则返回 0；如果数字为负数，则返回 -1；如果数字为正数，则返回 1。

语法: `sign(<expression>)`

- `expression`: 结果的数据类型为 `double` 的表达式。
- 返回类型: `int`。

示例:

```
nebula> RETURN sign(10);
+-----+
| sign(10) |
+-----+
| 1         |
+-----+
```

e()

`e()` 返回自然对数的底 e (2.718281828459045)。

语法: `e()`

- 返回类型: `double`。

示例:

```
nebula> RETURN e();
+-----+
| e()   |
+-----+
| 2.718281828459045 |
+-----+
```

pi()

`pi()` 返回数学常数π (3.141592653589793)。

语法: `pi()`

- 返回类型: `double`。

示例:

```
nebula> RETURN pi();
+-----+
| pi() |
+-----+
```

```
| 3.141592653589793 |
+-----+
```

radians()

radians() 返回指定角度的弧度。

语法: `radians(<angle>)`

- 返回类型: `double`。

示例:

```
nebula> RETURN radians(180);
+-----+
| radians(180)      |
+-----+
| 3.141592653589793 |
+-----+
```

最后更新: May 4, 2023

4.5.2 聚合函数

本文介绍NebulaGraph支持的聚合函数。

avg()

avg() 返回参数的平均值。

语法: `avg(<expression>)`

- 返回类型: `double`。

示例:

```
nebula> MATCH (v:player) RETURN avg(v.player.age);
+-----+
| avg(v.player.age) |
+-----+
| 33.294117647058826 |
+-----+
```

count()

count() 返回参数的数量。

- (原生 nGQL) 用户可以同时使用 `count()` 和 `GROUP BY` 对传参进行分组和计数，再使用 `YIELD` 返回结果。
- (openCypher 方式) 用户可以使用 `count()` 对指定的值进行计数，再使用 `RETURN` 返回结果。不需要使用 `GROUP BY`。

语法: `count({<expression> | *})`

- `count(*)` 返回总行数（包括 `NULL`）。
- 返回类型: `int`。

示例:

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

返回 player101 follow 的人，以及 follow player101 的人，即双向查询。

使用 `count()` 和 `GROUP BY` 进行分组和计数。

```
nebula> GO FROM "player101" OVER follow BIDIRECT \
    YIELD properties($$).name AS Name \
    | GROUP BY $-.Name YIELD $-.Name, count(*);
+-----+-----+
| $-.Name | count(*) |
+-----+-----+
| "LaMarcus Aldridge" | 2 |
| "Tim Duncan" | 2 |
| "Marco Belinelli" | 1 |
| "Manu Ginobili" | 1 |
| "Boris Diaw" | 1 |
| "Dejounte Murray" | 1 |
+-----+-----+
```

使用 `count()` 进行计数。

```
nebula> MATCH (v1:player)-[:follow]-(v2:player) \
    WHERE id(v1)= "player101" \
    RETURN v2.player.name AS Name, count(*) as cnt ORDER BY cnt DESC;
+-----+-----+
| Name | cnt |
+-----+-----+
| "LaMarcus Aldridge" | 2 |
| "Tim Duncan" | 2 |
| "Boris Diaw" | 1 |
| "Manu Ginobili" | 1 |
| "Dejounte Murray" | 1 |
| "Marco Belinelli" | 1 |
+-----+-----+
```

上述示例的返回结果有两列：

- `$-.Name`：查询结果包含的姓名。
- `count(*)`：姓名出现的次数。

因为测试数据集 `basketballplayer` 中没有重复的姓名，`count(*)` 列中数字 2 表示该行的人和 `player101` 是互相 `follow` 的关系。

```
# 方法一：统计数据库中的年龄分布情况。
nebula> LOOKUP ON player \
    YIELD player.age AS playerage \
    | GROUP BY $-.playerage \
    YIELD $-.playerage AS age, count(*) AS number \
    | ORDER BY $-.number DESC, $-.age DESC;
+-----+-----+
| age | number |
+-----+-----+
| 34  | 4      |
| 33  | 4      |
| 30  | 4      |
| 29  | 4      |
| 38  | 3      |
+-----+-----+
...
# 方法二：统计数据库中的年龄分布情况。
nebula> MATCH (n:player) \
    RETURN n.player.age AS age, count(*) AS number \
    ORDER BY number DESC, age DESC;
+-----+-----+
| age | number |
+-----+-----+
| 34  | 4      |
| 33  | 4      |
| 30  | 4      |
| 29  | 4      |
| 38  | 3      |
+-----+-----+
...
# 统计 Tim Duncan 关联的边数。
nebula> MATCH (v:player{name:"Tim Duncan"}) -[e]- (v2) \
    RETURN count(e);
+-----+
| count(e) |
+-----+
| 13      |
+-----+
# 多跳查询，统计 Tim Duncan 关联的边数，返回两列（不去重和去重）。
nebula> MATCH (n:player {name : "Tim Duncan"})-[]-(friend:player)-[]-(fof:player) \
    RETURN count(fof), count(DISTINCT fof);
+-----+-----+
| count(fof) | count(distinct fof) |
+-----+-----+
| 4          | 3          |
+-----+-----+
```

max()

`max()` 返回参数的最大值。

语法： `max(<expression>)`

- 返回类型：与原参数相同。

示例：

```
nebula> MATCH (v:player) RETURN max(v.player.age);
+-----+
| max(v.player.age) |
+-----+
| 47                |
+-----+
```

min()

`min()` 返回参数的最小值。

语法: `min(<expression>)`

- 返回类型: 与原参数相同。

示例:

```
nebula> MATCH (v:player) RETURN min(v.player.age);
+-----+
| min(v.player.age) |
+-----+
| 20 |
+-----+
```

collect()

`collect()` 返回一个符合表达式返回结果的列表。该函数可以将多条记录或值合并进一个列表，实现数据聚合。

语法: `collect(<expression>)`

- 返回类型: `list`。

示例:

```
nebula> UNWIND [1, 2, 1] AS a \
    RETURN a;
+---+
| a |
+---+
| 1 |
| 2 |
| 1 |
+---+

nebula> UNWIND [1, 2, 1] AS a \
    RETURN collect(a);
+-----+
| collect(a) |
+-----+
| [1, 2, 1] |
+-----+

nebula> UNWIND [1, 2, 1] AS a \
    RETURN a, collect(a), size(collect(a));
+-----+-----+
| a | collect(a) | size(collect(a)) |
+-----+-----+
| 2 | [2]      | 1           |
| 1 | [1, 1]    | 2           |
+-----+-----+

# 降序排列，限制输出行数为 3，然后将结果输出到列表中。
nebula> UNWIND ["c", "b", "a", "d"] AS p \
    WITH p AS q \
    ORDER BY q DESC LIMIT 3 \
    RETURN collect(q);
+-----+
| collect(q) |
+-----+
| ["d", "c", "b"] |
+-----+

nebula> WITH [1, 1, 2, 2] AS coll \
    UNWIND coll AS x \
    WITH DISTINCT x \
    RETURN collect(x) AS ss;
+-----+
| ss   |
+-----+
| [1, 2] |
+-----+

nebula> MATCH (n:player) \
    RETURN collect(n.player.age);
+-----+
| collect(n.player.age) |
+-----+
| [32, 32, 34, 29, 41, 40, 33, 25, 40, 37, ... |
+-----+

# 基于年龄聚合姓名。
nebula> MATCH (n:player) \
    RETURN n.player.age AS age, collect(n.player.name);
+-----+
| age | collect(n.player.name) |
+-----+
```

```

| 24 | ["Giannis Antetokounmpo"] |
| 20 | ["Luka Doncic"] |
| 25 | ["Joel Embiid", "Kyle Anderson"] |
+-----+
...
nebula> GO FROM "player100" OVER serve \
    YIELD properties($$).name AS name \
    | GROUP BY $-.name \
    YIELD collect($-.name) AS name;
+-----+
| name |
+-----+
| ["Spurs"] |
+-----+
nebula> LOOKUP ON player \
    YIELD player.age AS playerage \
    | GROUP BY $-.playerage \
    YIELD collect($-.playerage) AS playerage;
+-----+
| playerage |
+-----+
| [22] |
| [47] |
| [43] |
| [25, 25] |
+-----+
...

```

std()

std() 返回参数的总体标准差。

语法: `std(<expression>)`

- 返回类型: `double`。

示例:

```

nebula> MATCH (v:player) RETURN std(v.player.age);
+-----+
| std(v.player.age) |
+-----+
| 6.423895701687502 |
+-----+

```

sum()

sum() 返回参数的和。

语法: `sum(<expression>)`

- 返回类型: 与原参数相同。

示例:

```

nebula> MATCH (v:player) RETURN sum(v.player.age);
+-----+
| sum(v.player.age) |
+-----+
| 1698 |
+-----+

```

聚合示例

```

nebula> GO FROM "player100" OVER follow YIELD dst(edge) AS dst, properties($$).age AS age \
    | GROUP BY $-.dst \
    YIELD \
    $-.dst AS dst, \
    toInteger((sum($-.age)/count($-.age)) + avg(distinct $-.age)+1) AS statistics;
+-----+
| dst      | statistics |
+-----+-----+
| "player125" | 84.0   |
| "player101" | 74.0   |
+-----+-----+

```

最后更新: May 4, 2023

4.5.3 内置字符串函数

本文介绍NebulaGraph支持的字符串函数。

注意事项

- 字符串的表示方式为用双引号或单引号包裹。
- 和 SQL 一样, nGQL 的字符索引(位置)从 1 开始。但是 C 语言的字符索引是从 0 开始的。

strcasncmp()

`strcasncmp()` 比较两个字符串(不区分大小写)。

语法: `strcasncmp(<string_a>, <string_b>)`

- `string_a`、`string_b`: 待比较的字符串。
- 返回类型: `int`。
- 当 `string_a = string_b` 时, 返回 0, 当 `string_a > string_b` 是, 返回大于 0 的数, 当 `string_a < string_b` 时, 返回小于 0 的数。

示例:

```
nebula> RETURN strcasncmp("a", "aa");
+-----+
| strcasncmp("a", "aa") |
+-----+
| -97 |
+-----+
```

lower() 和 toLower()

`lower()` 和 `toLower()` 都可以返回指定字符串的小写形式。

语法: `lower(<string>)`、`toLower(<string>)`

- `string`: 指定的字符串。
- 返回类型: `string`。

示例:

```
nebula> RETURN lower("Basketball_Player");
+-----+
| lower("Basketball_Player") |
+-----+
| "basketball_player" |
+-----+
```

upper() 和 toUpper()

`upper()` 和 `toUpper()` 都可以返回指定字符串的大写形式。

语法: `upper(<string>)`、`toUpper(<string>)`

- `string`: 指定的字符串。
- 返回类型: `string`。

示例:

```
nebula> RETURN upper("Basketball_Player");
+-----+
| upper("Basketball_Player") |
+-----+
```

```
+-----+  
| "BASKETBALL_PLAYER" |  
+-----+
```

length()

`length()` 返回:

- 指定字符串的长度，单位：字节。
- 路径的长度，单位：跳。

语法: `length({<string>}|<path>)`

- `string` : 指定的字符串。
- `path` : 指定的路径，使用变量表示。
- 返回类型: `int`。

示例:

```
+-----+  
| length("basketball") |  
+-----+  
| 10 |  
+-----+
```

```
+-----+  
| length(p) |  
+-----+  
| 1 |  
| 1 |  
| 1 |  
+-----+
```

trim()

`trim()` 删除指定字符串头部和尾部的空格。

语法: `trim(<string>)`

- `string` : 指定的字符串。
- 返回类型: `string`。

示例:

```
+-----+  
| trim(" basketball player ") |  
+-----+  
| "basketball player" |  
+-----+
```

ltrim()

`ltrim()` 删除字符串头部的空格。

语法: `ltrim(<string>)`

- `string` : 指定的字符串。
- 返回类型: `string`。

示例:

```
+-----+  
| ltrim(" basketball player ") |  
+-----+
```

```
+-----+
| "basketball player "      |
+-----+
```

rtrim()

rtrim() 删除字符串尾部的空格。

语法: `rtrim(<string>)`

- `string`: 指定的字符串。
- 返回类型: `string`。

示例:

```
+-----+
| rtrim(" basketball player ") |
+-----+
| " basketball player"      |
+-----+
```

left()

left() 返回指定字符串头部若干个字符组成的子字符串。

语法: `left(<string>,<count>)`

- `string`: 指定的字符串。
- `count`: 指定从头部开始的字符数量。如果 `count` 超过字符串的长度，则返回字符串本身。
- 返回类型: `string`。

示例:

```
+-----+
| left("basketball_player",6) |
+-----+
| "basket"                   |
+-----+
```

right()

right() 返回指定字符串尾部若干个字符组成的子字符串。

语法: `right(<string>,<count>)`

- `string`: 指定的字符串。
- `count`: 指定从尾部开始的字符数量。如果 `count` 超过字符串的长度，则返回字符串本身。
- 返回类型: `string`。

示例:

```
+-----+
| right("basketball_player",6) |
+-----+
| "player"                    |
+-----+
```

lpad()

lpad() 在指定字符串的头部填充字符串至指定长度，并返回结果字符串。

语法: `lpad(<string>,<count>,<letters>)`

- `string`: 指定的字符串。
- `count`: 指定从尾部开始将要返回的字符数量。如果 `count` 少于 `string` 字符串的长度，则只返回 `string` 字符串从前到后的 `count` 个字符。
- `letters`: 从头部填充的字符串。
- 返回类型: `string`。

示例:

```
nebula> RETURN lpad("abcd",10,"b");
+-----+
| lpad("abcd",10,"b") |
+-----+
| "bbbbbbabcd" |
+-----+
```



```
nebula> RETURN lpad("abcd",3,"b");
+-----+
| lpad("abcd",3,"b") |
+-----+
| "abc" |
+-----+
```

rpad()

`rpad()` 在指定字符串的尾部填充字符串至指定长度，并返回结果字符串。

语法: `rpad(<string>,<count>,<letters>)`

- `string`: 指定的字符串。
- `count`: 指定从头部开始将要返回的字符数量。如果 `count` 少于 `string` 字符串的长度，则只返回 `string` 字符串从前到后的 `count` 个字符。
- `letters`: 从尾部填充的字符串。
- 返回类型: `string`。

示例:

```
nebula> RETURN rpad("abcd",10,"b");
+-----+
| rpad("abcd",10,"b") |
+-----+
| "abcd#####b" |
+-----+
```



```
nebula> RETURN rpad("abcd",3,"b");
+-----+
| rpad("abcd",3,"b") |
+-----+
| "abc" |
+-----+
```

substr() 和 substring()

`substr()` 和 `substring()` 从指定字符串的指定位置开始（不包括开始位置的字符），提取后面的若干个字符，组成新的字符串并返回。

语法: `substr(<string>,<pos>,<count>)` 、`substring(<string>,<pos>,<count>)`

- `string`: 指定的字符串。
- `pos`: 指定开始的位置，即字符索引，数据类型为 `int`。
- `count`: 指定从开始位置往后提取的字符数量。
- 返回类型: `string`。

SUBSTR() 和 SUBSTRING() 的返回说明

- 如果 pos 为 0，表示从指定字符串头部开始提取（包括第一个字符）。
- 如果 pos 大于最大字符索引，则返回空字符串。
- 如果 pos 是负数，则返回 BAD_DATA。
- 如果省略 count，则返回从 pos 位置开始到字符串末尾的子字符串。
- 如果 count 为 0，则返回空字符串。
- 使用 NULL 作为任何参数会出现错误。

 **openCypher 兼容性**

在 openCypher 中，如果字符串 a 为 null，会返回 null。

示例：

```
nebula> RETURN substr("abcdefg",2,4);
+-----+
| substr("abcdefg",2,4) |
+-----+
| "cdef"               |
+-----+

nebula> RETURN substr("abcdefg",0,4);
+-----+
| substr("abcdefg",0,4) |
+-----+
| "abcd"               |
+-----+

nebula> RETURN substr("abcdefg",2);
+-----+
| substr("abcdefg",2) |
+-----+
| "cdefg"              |
+-----+
```

reverse()

reverse() 逆序返回指定的字符串。

语法： reverse(<string>)

- string：指定的字符串。
- 返回类型：string。

示例：

```
nebula> RETURN reverse("abcdefg");
+-----+
| reverse("abcdefg") |
+-----+
| "gfedcba"          |
+-----+
```

replace()

replace() 将指定字符串中的子字符串 a 替换为字符串 b。

语法: `replace(<string>, <substr_a>, <string_b>)`

- `string`: 指定的字符串。
- `substr_a`: 子字符串 `a`。
- `string_b`: 字符串 `b`。
- 返回类型: `string`。

示例:

```
nebula> RETURN replace("abcdefg", "cd", "AAAAA");
+-----+
| replace("abcdefg", "cd", "AAAAA") |
+-----+
| "abAAAAAefg" |
+-----+
```

split()

`split()` 将子字符串 `b` 识别为分隔符, 分隔指定字符串, 并返回分隔后的字符串列表。

语法: `split(<string>, <substr>)`

- `string`: 指定的字符串。
- `substr`: 子字符串 `b`。
- 返回类型: `list`。

示例:

```
nebula> RETURN split("basketballplayer", "a");
+-----+
| split("basketballplayer", "a") |
+-----+
| ["b", "sketb", "llpl", "yer"] |
+-----+
```

concat()

`concat()` 返回所有参数连接成的字符串。

语法: `concat(<string1>, <string2>, ...)`

- 函数至少需要两个或以上字符串参数。如果字符串参数只有一个, 则返回该字符串参数本身。
- 如果任何一个的字符串参数为 `NULL`, 则 `concat()` 函数返回值为 `NULL`。
- 返回类型: `string`。

示例:

```
//连接 1, 2, 3
nebula> RETURN concat("1", "2", "3") AS r;
+-----+
| r   |
+-----+
| "123" |
+-----+

//字符串参数有 NULL
nebula> RETURN concat("1", "2", NULL) AS r;
+-----+
| r   |
+-----+
| __NULL__ |
+-----+

nebula> GO FROM "player100" over follow \
    YIELD concat(src(edge), properties($^).age, properties($$).name, properties(edge).degree) AS A;
+-----+
| A   |
+-----+
```

```
+-----+
| "player10042Tony Parker95" |
| "player10042Manu Ginobili95" |
+-----+
```

concat_ws()

`concat_ws()` 返回用分隔符 (`separator`) 连接的所有字符串。

语法: `concat_ws(<separator>,<string1>,<string2>,...)`

- 函数至少需要两个或以上字符串参数。
- 如果分隔符为 `NULL` 时, `concat_ws()` 函数才返回 `NULL`。
- 如果分隔符不为 `NULL`, 字符串参数只有一个, 则返回该字符串参数本身。
- 字符串参数存在 `NULL` 值时, 忽略 `NULL` 值, 继续连接下一个参数。

示例:

```
//分隔符为 +, 连接 a, b, c。
nebula> RETURN concat_ws("+" , "a" , "b" , "c") AS r;
+-----+
| r      |
+-----+
| "a+b+c" |
+-----+

//分隔符为 NULL。
nebula> RETURN concat_ws(NULL , "a" , "b" , "c") AS r;
+-----+
| r      |
+-----+
| _NULL_ |
+-----+

//分隔符为 +, 字符串参数有 NULL。
nebula> RETURN concat_ws("+" , "a" , NULL , "b" , "c") AS r;
+-----+
| r      |
+-----+
| "a+b+c" |
+-----+

//分隔符为+。字符串参数只有一个
nebula> RETURN concat_ws("+" , "a") AS r;
+-----+
| r      |
+-----+
| "a"   |
+-----+

nebula> GO FROM "player100" over follow \
    YIELD concat_ws(" ",src(edge), properties($^).age, properties($$).name, properties(edge).degree) AS A;
+-----+
| A          |
+-----+
| "player100 42 Tony Parker 95" |
| "player100 42 Manu Ginobili 95" |
+-----+
```

extract()

`extract()` 从指定字符串中提取符合正则表达式的子字符串。

语法: `extract(<string>,"<regular_expression>")`

- `string`: 指定的字符串。
- `regular_expression`: 正则表达式。
- 返回类型: `list`。

示例:

```
nebula> MATCH (a:player)-[b:serve]-(c:team{name: "Lakers"}) \
    WHERE a.player.age > 45 \
    RETURN extract(a.player.name, "\w+") AS result;
```

```
+-----+
| result |
+-----+
| ["Shaquille", "O", "Neal"] |
+-----+  
  
nebula> MATCH (a:player)-[b:serve]-(c:team{name: "Lakers"}) \
    WHERE a.player.age > 45 \
    RETURN extract(a.player.name, "hello") AS result;
+-----+
| result |
+-----+
| [] |
+-----+
```

json_extract() 函数

`json_extract()` 将指定 JSON 字符串转换为 `map` 类型。

语法: `extract(<string>)`

- `string`: 指定字符串, 为 JSON 格式。
- 返回类型: `map`。



- 目前仅支持 `Bool`、`Double`、`Int`、`String` 和 `NULL` 类型数据。
- 仅支持深度为 1 的 `Map` 嵌套, 如果嵌套深度为 2 及以上, 嵌套项保留为空。

示例:

```
+-----+
| result |
+-----+
| {a: 1, b: {}, c: {d: true}} |
+-----+
```

最后更新: May 4, 2023

4.5.4 内置日期时间函数

NebulaGraph支持以下内置日期时间函数。

函数	说明
<code>int now()</code>	根据当前系统返回当前时间戳。
<code>timestamp timestamp()</code>	根据当前系统返回当前时间戳。
<code>date date()</code>	根据当前系统返回当前日期（UTC 时间）。
<code>time time()</code>	根据当前系统返回当前时间（UTC 时间）。
<code>datetime datetime()</code>	根据当前系统返回当前日期和时间（UTC 时间）。
<code>map duration()</code>	持续时间。可以用于对指定时间进行计算。

详细信息参见[日期和时间类型](#)。

示例

```
nebula> RETURN now(), timestamp(), date(), time(), datetime();
```

now()	timestamp()	date()	time()	datetime()
1640057560	1640057560	2021-12-21	03:32:40.351000	2021-12-21T03:32:40.351000

最后更新: May 4, 2023

4.5.5 Schema 相关函数

本文介绍NebulaGraph支持的 **Schema** 相关的函数。

Schema 相关的函数分为两类:

- 适用于原生 nGQL 语句
- 适用于 openCypher 兼容语句

原生 **nGQL** 语句适用

原生 nGQL 语句的 `YIELD` 和 `WHERE` 子句中可以使用如下介绍的函数。



由于 `vertex`、`edge`、`vertices`、`edges`、`path` 属于关键字，使用时需要用 `AS <alias>` 设置别名才能正常使用。例如 `GO FROM "player100" OVER follow YIELD edge AS e;`。

`ID(VERTEX)`

`id(vertex)` 返回点 ID。

语法: `id(vertex)`

- 返回类型: 和点 ID 的类型保持一致。

示例:

```
nebula> LOOKUP ON player WHERE player.age > 45 YIELD id(vertex);
+-----+
| id(VERTEX) |
+-----+
| "player144" |
| "player140" |
+-----+
```

`PROPERTIES(VERTEX)`

`properties(vertex)` 返回点的所有属性。

语法: `properties(vertex)`

- 返回类型: `map`。

示例:

```
nebula> LOOKUP ON player WHERE player.age > 45 \
    YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 47, name: "Shaquille O'Neal"} |
| {age: 46, name: "Grant Hill"} |
+-----+
```

用户也可以使用属性引用符（`$^` 和 `$$`）替代函数 `properties()` 中的 `vertex` 参数来获取点的所有属性。

- `$^` 表示探索开始时的点数据。例如 `GO FROM "player100" OVER follow reversely YIELD properties($^)` 中，`$^` 指 `player100` 这个点。
- `$$` 表示探索结束的终点数据。

`properties($^)` 和 `properties($$)` 一般用于 `GO` 语句中。更多信息，请参见[属性引用符](#)。

Caution

用户可以通过 `properties().<property_name>` 来获取点的指定属性。但是不建议使用这种方式获取指定属性，因为 `properties()` 函数返回所有属性，这样会降低查询性能。

PROPERTIES(EDGE)

`properties(edge)` 返回边的所有属性。

语法: `properties(edge)`

- 返回类型: `map`。

示例:

```
nebula> GO FROM "player100" OVER follow \
    YIELD properties(edge);
+-----+
| properties(EDGE) |
+-----+
| {degree: 95}     |
| {degree: 95}     |
+-----+
```

Warning

用户可以通过 `properties(edge).<property_name>` 来获取边的指定属性。但是不建议使用这种方式获取指定属性，因为 `properties(edge)` 函数返回边的所有属性，这样会降低查询性能。

TYPE(EDGE)

`type(edge)` 返回边的 Edge type。

语法: `type(edge)`

- 返回类型: `string`。

示例:

```
nebula> GO FROM "player100" OVER follow \
    YIELD src(edge), dst(edge), type(edge), rank(edge);
+-----+-----+-----+-----+
| src(EDGE) | dst(EDGE) | type(EDGE) | rank(EDGE) |
+-----+-----+-----+-----+
| "player100" | "player101" | "follow" | 0
| "player100" | "player125" | "follow" | 0
+-----+-----+-----+-----+
```

SRC(EDGE)

`src(edge)` 返回边的起始点 ID。

语法: `src(edge)`

- 返回类型: 和点 ID 的类型保持一致。

示例:

```
nebula> GO FROM "player100" OVER follow \
    YIELD src(edge), dst(edge);
+-----+-----+
| src(EDGE) | dst(EDGE) |
+-----+-----+
| "player100" | "player101" |
| "player100" | "player125" |
+-----+-----+
```

Note

`src(edge)` 和 `properties($^)` 查找起始点的语义不同。`src(edge)` 始终表示图数据库中边的起始点 ID，而 `properties($^)` 表示探索开始时的点数据，例如示例中 `GO FROM "player100"` 中 `player100` 这个点的数据。

DST(EDGE)

`dst(edge)` 返回边的目的点 ID。

语法: `dst(edge)`

- 返回类型: 和点 ID 的类型保持一致。

示例:

```
nebula> GO FROM "player100" OVER follow \
    YIELD src(edge), dst(edge);
+-----+-----+
| src(EDGE) | dst(EDGE) |
+-----+-----+
| "player100" | "player101" |
| "player100" | "player125" |
+-----+-----+
```

Note

`dst(edge)` 始终表示图数据库中边的目的点 ID。

RANK(EDGE)

`rank(edge)` 返回边的 rank。

语法: `rank(edge)`

- 返回类型: `int`。

示例:

```
nebula> GO FROM "player100" OVER follow \
    YIELD src(edge), dst(edge), rank(edge);
+-----+-----+-----+
| src(EDGE) | dst(EDGE) | rank(EDGE) |
+-----+-----+-----+
| "player100" | "player101" | 0 |
| "player100" | "player125" | 0 |
+-----+-----+-----+
```

VERTEX

`vertex` 返回点的信息。包括点 ID、Tag、属性和值。需要用 `AS <alias>` 设置别名。

语法: `vertex`

示例:

```
nebula> LOOKUP ON player WHERE player.age > 45 YIELD vertex AS v;
+-----+
| v |
+-----+
| ("player144" :player{age: 47, name: "Shaquille O'Neal"}) |
| ("player140" :player{age: 46, name: "Grant Hill"}) |
+-----+
```

EDGE

`edge` 返回边的信息。包括 Edge type、起始点 ID、目的点 ID、rank、属性和值。需要用 `AS <alias>` 设置别名。

语法: `edge`

示例:

```
nebula> GO FROM "player100" OVER follow YIELD edge AS e;
+-----+
| e |
+-----+
| [:follow "player100"->"player101" @0 {degree: 95}] |
| [:follow "player100"->"player125" @0 {degree: 95}] |
+-----+
```

VERTICES

vertices 返回子图中的点的信息。详情参见 [GET SUBGRAPH](#)。

EDGES

edges 返回子图中的边的信息。详情参见 [GET SUBGRAPH](#)。

PATH

path 返回路径信息。详情参见 [FIND PATH](#)。

openCypher 兼容语句适用

openCypher 兼容语句的 RETURN 和 WHERE 子句中可以使用如下介绍的函数。

ID()

id() 返回点 ID。

语法: `id(<vertex>)`

- 返回类型: 和点 ID 的类型保持一致。

示例:

```
nebula> MATCH (v:player) RETURN id(v);
+-----+
| id(v) |
+-----+
| "player129" |
| "player115" |
| "player106" |
| "player102" |
...
```

TAGS() 和LABELS()

tags() 和**labels()** 返回点的 Tag。

语法: `tags(<vertex>)` 、 `labels(<vertex>)`

- 返回类型: list。

示例:

```
nebula> MATCH (v) WHERE id(v) == "player100" \
  RETURN tags(v);
+-----+
| tags(v) |
+-----+
| ["player"] |

```

PROPERTIES()

properties() 返回点或边的所有属性。

语法: `properties(<vertex_or_edge>)`

- 返回类型: map。

示例:

```
nebula> MATCH (v:player)-[e:follow]-() RETURN properties(v),properties(e);
+-----+-----+
| properties(v) | properties(e) |
+-----+-----+
| {age: 31, name: "Stephen Curry"} | {degree: 90} |
| {age: 47, name: "Shaquille O'Neal"} | {degree: 100} |
| {age: 34, name: "LeBron James"} | {degree: 13} |
...
...
```

TYPE()

type() 返回边的 Edge type。

语法: `type(<edge>)`

- 返回类型: `string`。

示例:

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
    RETURN type(e);
+-----+
| type(e) |
+-----+
| "serve" |
| "follow" |
| "follow" |
+-----+
```

SRC()

src() 返回边的起始点 ID。

语法: `src(<edge>)`

- 返回类型: 和点 ID 的类型保持一致。

示例:

```
nebula> MATCH ()-[e]-(v:player{name:"Tim Duncan"}) \
    RETURN src(e);
+-----+
| src(e) |
+-----+
| "player125" |
| "player113" |
| "player102" |
...
...
```

DST()

dst() 返回边的目的点 ID。

语法: `dst(<edge>)`

- 返回类型: 和点 ID 的类型保持一致。

示例:

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
    RETURN dst(e);
+-----+
| dst(e) |
+-----+
| "team204" |
| "player101" |
| "player125" |
+-----+
```

<!-- 方便计算下推, 用于POC, 暂不对外发布, 无英文文档 -->

NONE_DIRECT_SRC()

none_direct_src() 返回的是不指定边方向的的起点 ID, 而是指定边的编码的四元组 (`_src`、`_dst`、`_rank`、`_type`) 中 `_src` 字段。

语法: `none_direct_src(<edge>)`

示例：

```
nebula> MATCH (v:player)-[e:follow]->(v2) \
    WHERE none_direct_src(e) \
    IN ["player100", "player101"] \
    RETURN v2.player.age;
+-----+
| v2.player.age |
+-----+
| 36           |
| 41           |
| 42           |
| 33           |
| 41           |
+-----+
```

none_direct_dst()

none_direct_dst() 返回的是不指定边方向的的终点 ID，而是指定边的编码的四元组（`_src`、`_dst`、`_rank`、`_type`）中`_dst`字段。

语法：`none_direct_dst(<edge>)`

示例：

```
```ngql
nebula> MATCH (v:player)-[e:follow]->(v2) \
 WHERE none_direct_dst(e) \
 IN ["player100", "player101"] \
 RETURN v2.player.age;
+-----+
| v2.player.age |
+-----+
| 36 |
| 42 |
| 42 |
| 42 |
| 42 |
| ... |
+-----+
```

-->

### startNode()

startNode() 获取一条路径并返回它的起始点信息，包括点 ID、Tag、属性和值。

语法：`startNode(<path>)`

示例：

```
```ngql
nebula> MATCH p = (a :player {name : "Tim Duncan"})-[r:serve]-(t) \
    RETURN startNode(p);
+-----+
| startNode(p)           |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+
```

ENDNODE()

endNode() 获取一条路径并返回它的目的点信息，包括点 ID、Tag、属性和值。

语法：`endNode(<path>)`

示例：

```
nebula> MATCH p = (a :player {name : "Tim Duncan"})-[r:serve]-(t) \
    RETURN endNode(p);
+-----+
| endNode(p)           |
+-----+
| ("team204" :team{name: "Spurs"}) |
+-----+
```

RANK()

rank() 返回边的 rank。

语法：`rank(<edge>)`

- 返回类型：int。

示例:

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
    RETURN rank(e);
+-----+
| rank(e) |
+-----+
| 0      |
| 0      |
| 0      |
+-----+
```

最后更新: May 4, 2023

4.5.6 列表函数

本文介绍NebulaGraph支持的列表（List）函数。部分列表函数在原生 nGQL 语句和 openCypher 兼容语句中的语法不同。

注意事项

- 和 SQL 一样，nGQL 的字符索引（位置）从 1 开始。但是 C 语言的字符索引是从 0 开始的。

通用

RANGE()

range() 返回指定整数范围 [start,end] 内固定步长的列表。

语法: `range(start, end [, step])`

- `step` : 可选参数。步长。默认为 1。
- 返回类型: list。

示例:

```
nebula> RETURN range(1,9,2);
+-----+
| range(1,9,2)   |
+-----+
| [1, 3, 5, 7, 9] |
+-----+
```

REVERSE()

reverse() 返回将原列表逆序排列的新列表。

语法: `reverse(<list>)`

- 返回类型: list。

示例:

```
nebula> WITH [NULL, 4923, 'abc', 521, 487] AS ids \
    RETURN reverse(ids);
+-----+
| reverse(ids)           |
+-----+
| [487, 521, "abc", 4923, __NULL__] |
+-----+
```

TAIL()

tail() 返回不包含原列表第一个元素的新列表。

语法: `tail(<list>)`

- 返回类型: list。

示例:

```
nebula> WITH [NULL, 4923, 'abc', 521, 487] AS ids \
    RETURN tail(ids);
+-----+
| tail(ids)           |
+-----+
| [4923, "abc", 521, 487] |
+-----+
```

HEAD()

head() 返回列表的第一个元素。

语法: `head(<list>)`

- 返回类型: 与原列表内的元素类型相同。

示例:

```
nebula> WITH [NULL, 4923, 'abc', 521, 487] AS ids \
    RETURN head(ids);
+-----+
| head(ids) |
+-----+
| __NULL__ |
+-----+
```

LAST()

`last()` 返回列表的最后一个元素。

语法: `last(<list>)`

- 返回类型: 与原列表内的元素类型相同。

示例:

```
nebula> WITH [NULL, 4923, 'abc', 521, 487] AS ids \
    RETURN last(ids);
+-----+
| last(ids) |
+-----+
| 487       |
+-----+
```

REDUCE()

`reduce()` 将表达式逐个应用于列表中的元素，然后和累加器中的当前结果累加，最后返回完整结果。该函数将遍历给定列表中的每个元素 `e`，在 `e` 上运行表达式并和累加器的当前结果累加，将新的结果存储在累加器中。这个函数类似于函数式语言（如 `Lisp` 和 `Scala`）中的 `fold` 或 `reduce` 方法。

↑ openCypher 兼容性

在 openCypher 中，`reduce()` 函数没有定义。nGQL 使用了 Cypher 方式实现 `reduce()` 函数。

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

- `accumulator`: 在遍历列表时保存累加结果。
- `initial`: 为 `accumulator` 提供初始值的表达式或值。
- `variable`: 为列表引入一个变量，决定使用列表中的哪个元素。
- `list`: 列表或列表表达式。
- `expression`: 该表达式将对列表中的每个元素运行一次，并将结果累加至 `accumulator`。

- 返回类型: 取决于提供的参数，以及表达式的语义。

示例:

```
nebula> RETURN reduce(totalNum = -4 * 5, n IN [1, 2] | totalNum + n * 2) AS r;
+---+
| r |
+---+
| -14 |
+---+
```



```
nebula> MATCH p = (n:player{name:"LeBron James"})-[:follow]-(m) \
    RETURN nodes(p)[0].player.age AS src1, nodes(p)[1].player.age AS dst2, \
    reduce(totalAge = 100, n IN nodes(p) | totalAge + n.player.age) AS sum;
+-----+-----+-----+
| src1 | dst2 | sum |
+-----+-----+-----+
| 34   | 31   | 165  |
| 34   | 29   | 163  |
```

```
| 34 | 33 | 167 |
| 34 | 26 | 160 |
| 34 | 34 | 168 |
| 34 | 37 | 171 |
+-----+-----+-----+
nebula> LOOKUP ON pLayer WHERE player.name == "Tony Parker" YIELD id(vertex) AS VertexID \
| GO FROM $.VertexID over follow \
WHERE properties(edge).degree != reduce(totalNum = 5, n IN range(1, 3) | properties($$).age + totalNum + n) \
YIELD properties($$).name AS id, properties($$).age AS age, properties(edge).degree AS degree;
+-----+-----+-----+
| id | age | degree |
+-----+-----+-----+
| "Tim Duncan" | 42 | 95 |
| "LaMarcus Aldridge" | 33 | 90 |
| "Manu Ginobili" | 41 | 95 |
+-----+-----+-----+
```

原生 nGQL 语句适用

KEYS()

keys() 返回一个列表，包含字符串形式的点、边的所有属性。

语法: `keys({vertex | edge})`

- 返回类型: list。

示例:

```
nebula> LOOKUP ON player \
WHERE player.age > 45 \
YIELD keys(vertex);
+-----+
| keys(VERTEX) |
+-----+
| ["age", "name"] |
| ["age", "name"] |
+-----+
```

LABELS()

labels() 返回点的 Tag 列表。

语法: `labels(verte)`

- 返回类型: list。

示例:

```
nebula> FETCH PROP ON * "player101", "player102", "team204" \
YIELD labels(vertex);
+-----+
| labels(VERTEX) |
+-----+
| ["player"] |
| ["player"] |
| ["team"] |
+-----+
```

openCypher 兼容语句适用

KEYS()

keys() 返回一个列表，包含字符串形式的点、边或映射的所有属性。

语法: `keys(<vertex_or_edge>)`

- 返回类型: list。

示例:

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
RETURN keys(e);
+-----+
| keys(e) |
+-----+
```

```
| ["end_year", "start_year"] |  
| ["degree"] |  
| ["degree"] |  
+-----+
```

LABELS()

`labels()` 返回点的 Tag 列表。

语法: `labels(<vertex>)`

- 返回类型: list。

示例：

```
nebula> MATCH (v)-[e:serve]->() \
    WHERE id(v)=="player100" \
    RETURN labels(v);
```

+-----+
Labels(v)
+-----+
["player"]
+-----+

NODES()

`nodes()` 返回路径中所有点的列表。包括点 ID、Tag、属性和值。

语法: nodes(<path>)

- 返回类型: list。

示例：

RELATIONSHIPS()

`relationships()` 返回路径中所有关系的列表。

语法: relationships(<path>)

- 返回类型：list。

示例：

```
nebuLa> MATCH p=(v:player{name:"Tim Duncan"})-->(v2) \
    RETURN relationships(p);
+-----+
| relationships(p)
+-----+
| [[{:serve "player100" ->"team204" @ {end_year: 2016, start_year: 1997}}]]
| [[{:follow "player100" ->"player101" @ {degree: 95}}]]
| [[{:follow "player100" ->"player125" @ {degree: 95}}]]
+
|
```

最后更新: May 4, 2023

4.5.7 类型转换函数

本文介绍NebulaGraph支持的类型转换函数。

toBoolean()

`toBoolean()` 将字符串转换为布尔。

语法: `toBoolean(<value>)`

- 返回类型: `bool`。

示例:

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

toFloat()

`toFloat()` 将整数或字符串转换为浮点数。

语法: `toFloat(<value>)`

- 返回类型: `float`。

示例:

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

toString()

`toString()` 将任意非复合数据类型数据转换为字符串类型。

语法: `toString(<value>)`

- 返回类型: `string`。

示例:

```
nebula> RETURN toString(9669) AS int2str, toString(null) AS null2str;
+-----+-----+
| int2str | null2str |
+-----+-----+
| "9669" | __NULL__ |
+-----+-----+
```

toInteger()

`toInteger()` 将浮点或字符串转换为整数。

语法: `toInteger(<value>)`

- 返回类型: `int`。

示例:

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

toSet()

toSet() 将列表或集合转换为集合。

语法: `toSet(<value>)`

- 返回类型: `set`。

示例:

```
nebula> RETURN toSet(list[1,2,3,1,2]) AS list2set;
+-----+
| list2set |
+-----+
| {3, 1, 2} |
+-----+
```

hash()

hash() 返回参数的哈希值。其参数可以是数字、字符串、列表、布尔值、NULL 等类型的值，或者计算结果为这些类型的表达式。

hash() 函数采用 MurmurHash2 算法，种子 (**seed**) 为 `0xc70f6907UL`。用户可以在 [MurmurHash2.h](#) 中查看其源代码。

在 Java 中的调用方式如下:

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

语法: `hash(<string>)`

- 返回类型: `int`。

示例:

```
nebula> RETURN hash("abcde");
+-----+
| hash("abcde") |
+-----+
| 811036730794841393 |
+-----+

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

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

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

最后更新: May 4, 2023

4.5.8 条件表达式函数

本文介绍NebulaGraph支持的条件表达式函数。

CASE

CASE 表达式使用条件来过滤传参。和 openCypher 一样，nGQL 提供两种形式的 CASE 表达式：简单形式和通用形式。

CASE 表达式会遍历所有条件，并在满足第一个条件时停止读取后续条件，然后返回结果。如果不满足任何条件，将通过 ELSE 子句返回结果。如果没有 ELSE 子句且不满足任何条件，则返回 NULL。

简单形式

- 语法

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



CASE 表达式一定要用 END 结尾。

参数	说明
comparer	用于与 value 进行比较的值或者有效表达式。
value	和 comparer 进行比较，如果匹配，则满足此条件。
result	如果 value 匹配 comparer，则返回该 result。
default	如果没有条件匹配，则返回该 default。

- 示例

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

```
+-----+
| result |
+-----+
| 1      |
+-----+
```



```
nebula> GO FROM "player100" OVER follow \
YIELD properties($$).name AS Name, \
CASE properties($$).age > 35 \
WHEN true THEN "Yes" \
WHEN false THEN "No" \
ELSE "Nah" \
END \
AS Age_above_35;
```

Name	Age_above_35
"Tony Parker"	"Yes"
"Manu Ginobili"	"Yes"

通用形式

- 语法

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

参数	说明
condition	如果条件 condition 为 true，表示满足此条件。
result	condition 为 true，则返回此 result。
default	如果没有条件匹配，则返回该 default。

- 示例

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

```
+-----+
| result |
+-----+
| 1      |
+-----+
```

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

简单形式和通用形式的区别

为了避免误用简单形式和通用形式，用户需要了解它们的差异。请参见如下示例：

```
nebula> GO FROM "player100" OVER follow \
YIELD properties($$).name AS Name, properties($$).age AS Age, \
CASE properties($$).age \
WHEN properties($$).age > 35 THEN "Yes" \
ELSE "No" \
END \
AS Age_above_35;
```

示例本意为当玩家年龄大于 35 时输出 Yes。但是查看输出结果，年龄为 36 时输出的却是 No。

这是因为查询使用了简单形式的 CASE 表达式，比较对象是 \$\$.player.age 和 \$\$.player.age > 35。当年龄为 36 时：

- \$\$.player.age 的值为 36，数据类型为 int。
- \$\$.player.age > 35 的值为 true，数据类型为 boolean。

这两种数据类型无法匹配，不满足条件，因此返回 No。

coalesce()

coalesce() 返回所有表达式中第一个非空元素。

语法: `coalesce(<expression_1>[,<expression_2>...])`

- 返回类型: 与原元素类型相同。

示例:

```
nebula> RETURN coalesce(null,[1,2,3]) as result;
+-----+
| result |
+-----+
| [1, 2, 3] |
+-----+

nebula> RETURN coalesce(null) as result;
+-----+
| result |
+-----+
| __NULL__ |
+-----+
```

最后更新: May 4, 2023

4.5.9 谓词函数

谓词函数只返回 `true` 或 `false`，通常用于 `WHERE` 子句中。

NebulaGraph 支持以下谓词函数。

函数	说明
<code>exists()</code>	如果指定的属性在点、边或映射中存在，则返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>any()</code>	如果指定的谓词适用于列表中的至少一个元素，则返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>all()</code>	如果指定的谓词适用于列表中的每个元素，则返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>none()</code>	如果指定的谓词不适用于列表中的任何一个元素，则返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>single()</code>	如果指定的谓词适用于列表中的唯一一个元素，则返回 <code>true</code> ，否则返回 <code>false</code> 。

Note

如果列表为空，或者列表中的所有元素都为空，则返回 `NULL`。

Incompatibility

在 openCypher 中只定义了函数 `exists()`，其他几个函数依赖于具体实现。

语法

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

示例

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

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

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

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

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

```

| "LeBron James" | "Dejounte Murray" | false |
| "LeBron James" | "Chris Paul" | true |
| "LeBron James" | "Kyrie Irving" | true |
| "LeBron James" | "Carmelo Anthony" | true |
| "LeBron James" | "Dwyane Wade" | false |
+-----+-----+-----+
nebula> MATCH p = (n:player{name:"LeBron James"})-[:follow]->(m) \
    RETURN single(n IN nodes(p) WHERE n.player.age > 40) AS b;
+-----+
| b |
+-----+
| true |
+-----+
nebula> MATCH (n:player) \
    RETURN exists(n.player.id), n IS NOT NULL;
+-----+-----+
| exists(n.player.id) | n IS NOT NULL |
+-----+-----+
| false | true |
...
nebula> MATCH (n:player) \
    WHERE exists(n['name']) \
    RETURN n;
+-----+
| n |
+-----+
| {"player105":player{age: 31, name: "Danny Green"}}, {"player109":player{age: 34, name: "Tiago Splitter"}}, {"player111":player{age: 38, name: "David West"}}
...

```

最后更新: May 4, 2023

4.5.10 geo 函数

geo 函数用于生成地理位置 (GEOGRAPHY) 数据类型的值或对其执行操作。

关于地理位置数据类型说明请参见[地理位置](#)。

函数说明

函数	返回类型	说明
ST_Point(longitude, latitude)	GEOGRAPHY	创建包含一个点的地理位置。
ST_GeogFromText(wkt_string)	GEOGRAPHY	返回与传入的 WKT 字符串形式相对应的 GEOGRAPHY。
ST_ASText(geography)	STRING	返回传入的 GEOGRAPHY 的 WKT 字符串形式。
ST_Centroid(geography)	GEOGRAPHY	以单点 GEOGRAPHY 的形式返回传入的 GEOGRAPHY 的形心。
ST_IsValid(geography)	BOOL	返回传入的 GEOGRAPHY 是否有效。
ST_Intersects(geography_1, geography_2)	BOOL	返回传入的两个 GEOGRAPHY 是否有交集。
ST_Covers(geography_1, geography_2)	BOOL	返回 geography_1 是否完全包含 geography_2。如果 geography_2 中没有位于 geography_1 外部的点，返回 True。
ST_CoveredBy(geography_1, geography_2)	BOOL	返回 geography_2 是否完全包含 geography_1。如果 geography_1 中没有位于 geography_2 外部的点，返回 True。
ST_DWithin(geography_1, geography_2, distance)	BOOL	如果 geography_1 中至少有一个点与 geography_2 中的一个点的距离小于或等于 distance 参数（以米为单位）指定的距离，则返回 True。
ST_Distance(geography_1, geography_2)	FLOAT	返回两个非空 GEOGRAPHY 之间的最短距离（以米为单位）。
S2_CellIdFromPoint(point_geography)	INT	返回覆盖点 GEOGRAPHY 的 S2 单元 ID。
S2_CoveringCellIds(geography)	ARRAY<INT64>	返回覆盖传入的 GEOGRAPHY 的 S2 单元 ID 的数组。

示例

```
nebula> RETURN ST_ASText(ST_Point(1,1));
+-----+
| ST_ASText(ST_Point(1,1)) |
+-----+
| "POINT(1 1)"           |
+-----+

nebula> RETURN ST_ASText(ST_GeogFromText("POINT(3 8)"));
+-----+
| ST_ASText(ST_GeogFromText("POINT(3 8)")) |
+-----+
| "POINT(3 8)"           |
+-----+

nebula> RETURN ST_ASTEXT(ST_Centroid(ST_GeogFromText("LineString(0 1,1 0)")));
+-----+
| ST_ASTEXT(ST_Centroid(ST_GeogFromText("LineString(0 1,1 0)")) |
+-----+
| "POINT(0.5000380800773782 0.5000190382261059)"           |
+-----+

nebula> RETURN ST_IsValid(ST_GeogFromText("POINT(3 8)"));
+-----+
| ST_IsValid(ST_GeogFromText("POINT(3 8)")) |
+-----+
| true                         |
+-----+

nebula> RETURN ST_Intersects(ST_GeogFromText("LineString(0 1,1 0"),ST_GeogFromText("LineString(0 0,1 1)"));
+-----+
| ST_Intersects(ST_GeogFromText("LineString(0 1,1 0"),ST_GeogFromText("LineString(0 0,1 1)")) |
+-----+
| true                         |
+-----+
```

```
+-----+
nebula> RETURN ST_Covers(ST_GeogFromText("POLYGON((0 0,10 0,10 10,0 10,0 0))"),ST_Point(1,2));
+-----+
| ST_Covers(ST_GeogFromText("POLYGON((0 0,10 0,10 10,0 10,0 0))"),ST_Point(1,2)) |
+-----+
| true |
+-----+

nebula> RETURN ST_CoveredBy(ST_Point(1,2),ST_GeogFromText("POLYGON((0 0,10 0,10 10,0 10,0 0))"));
+-----+
| ST_CoveredBy(ST_Point(1,2),ST_GeogFromText("POLYGON((0 0,10 0,10 10,0 10,0 0))") |
+-----+
| true |
+-----+

nebula> RETURN ST_dwithin(ST_GeogFromText("Point(0 0)",ST_GeogFromText("Point(10 10)",20000000000.0));
+-----+
| ST_dwithin(ST_GeogFromText("Point(0 0)",ST_GeogFromText("Point(10 10)",20000000000) |
+-----+
| true |
+-----+

nebula> RETURN ST_Distance(ST_GeogFromText("Point(0 0)",ST_GeogFromText("Point(10 10)")));
+-----+
| ST_Distance(ST_GeogFromText("Point(0 0)",ST_GeogFromText("Point(10 10)")) |
+-----+
| 1.5685230187677438e+06 |
+-----+

nebula> RETURN S2_CellIdFromPoint(ST_GeogFromText("Point(1 1)"));
+-----+
| S2_CellIdFromPoint(ST_GeogFromText("Point(1 1)") |
+-----+
| 1153277837650709461 |
+-----+

nebula> RETURN S2_CoveringCellIds(ST_GeogFromText("POLYGON((0 1, 1 2, 2 3, 0 1))"));
+-----+
| S2_CoveringCellIds(ST_GeogFromText("POLYGON((0 1, 1 2, 2 3, 0 1))") |
+-----+
| [1152391494368201343, 1153466862374223872, 1153554823304445952, 1153836298281156608, 1153959443583467520, 1154240918560178176, 1160503736791990272, 116059169772212352] |
+-----+
```

最后更新: May 4, 2023

4.5.11 自定义函数

openCypher 兼容性

NebulaGraph 3.4.1 不支持自定义函数（UDF）和存储过程。

最后更新: May 4, 2023

4.6 通用查询语句

4.6.1 MATCH

`MATCH` 语句提供基于模式（Pattern）匹配的搜索功能，其通过定义一个或多个模式，允许在 NebulaGraph 中查找与模式匹配的数据。在检索到匹配的数据后，用户可以使用 `RETURN` 子句将其作为结果返回。

在本文中，我们将使用名为 `basketballplayer` 的测试数据集来演示 `MATCH` 语句的使用。

语法

`MATCH` 语句的语法相较于其他查询语句（如 `GO` 和 `LOOKUP`）更具灵活性。在进行查询时，`MATCH` 语句使用的路径类型是 `trail`，这意味着点可以重复出现，但边不能重复。

`MATCH` 语法的基本结构如下：

```
MATCH <pattern> [<clause_1>] RETURN <output> [<clause_2>];
```

- `pattern`：`MATCH` 语句支持匹配一个或多个模式，多个模式之间用英文逗号（,）分隔。例如 `(a)-[]->(b),(c)-[]->(d)`。`Pattern` 的详细说明请参见[模式](#)。
- `clause_1`：支持 `WHERE`、`WITH`、`UNWIND`、`OPTIONAL MATCH` 子句，也可以使用 `MATCH` 作为子句。
- `output`：定义需要返回输出结果的列表名称。可以使用 `AS` 设置列表的别名。
- `clause_2`：支持 `ORDER BY`、`LIMIT` 子句。

↑ 版本兼容性

- 从 3.5.0 版本开始，`MATCH` 语句支持全表扫描，即在不使用任何索引或者过滤条件的情况下可遍历图中点或边。在此之前的版本中，`MATCH` 语句在某些情况下需要索引才能执行查询或者需要使用 `LIMIT` 限制输出结果数量。
- 从 3.0.0 版本开始，为了区别不同 Tag 的属性，返回属性时必须额外指定 Tag 名称。即从 `RETURN <变量名>.<属性名>` 改为 `RETURN <变量名>.<Tag名>.<属性名>`。

使用说明

- 尽量避免执行全表扫描，因为这可能导致查询性能下降；并且如果在进行全表扫描时内存不足，可能会导致查询失败，系统会提示报错。建议使用具有过滤条件或指定 Tag、边类型的查询，例如 `MATCH (v:player) RETURN v.player.name AS Name` 语句中的 `v:player` 和 `v.player.name`。
- 可为 Tag、Edge type 或 Tag、Edge type 的某个属性创建索引，以提高查询性能。例如，用户可以为 `player` Tag 创建索引，或者为 `player` Tag 的 `name` 属性创建索引。有关索引的使用及注意事项，请参见[使用索引必读](#)。
- 目前 `MATCH` 语句无法查询到悬挂边。

使用模式

匹配点

用户可以在一对括号中使用自定义变量来表示模式中的点。例如 `(v)`。

```
nebula> MATCH (v) \
    RETURN v \
    LIMIT 3;
+-----+
| v |
+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
| ("player106" :player{age: 25, name: "Kyle Anderson"}) |
| ("player115" :player{age: 40, name: "Kobe Bryant"}) |
+-----+
```

匹配 TAG

 版本兼容性

在NebulaGraph 3.0.0 之前，匹配 Tag 的前提是 Tag 本身有索引或者 Tag 的某个属性有索引，否则，用户无法基于该 Tag 执行 MATCH 语句。从 NebulaGraph 3.0.0 开始，匹配 Tag 可以不创建索引，但需要使用 LIMIT 限制输出结果数量。从NebulaGraph 3.5.0 开始，MATCH 语句支持全表扫描，无需为 Tag 或 Tag 的某个属性创建索引，或者使用 LIMIT 限制输出结果数量，即可执行 MATCH 语句。

用户可以在点的右侧用 :<tag_name> 表示模式中的 Tag。

```
nebula> MATCH (v:player) \
    RETURN v;
+-----+
| v |
+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
| ("player106" :player{age: 25, name: "Kyle Anderson"}) |
| ("player115" :player{age: 40, name: "Kobe Bryant"}) |
...
...
```

需要匹配拥有多个 Tag 的点，可以用英文冒号 (:)。

```
nebula> CREATE TAG actor (name string, age int);
nebula> INSERT VERTEX actor(name, age) VALUES "player100":("Tim Duncan", 42);
nebula> MATCH (v:player:actor) \
    RETURN v;
+-----+
| v |
+-----+
| ("player100" :actor{age: 42, name: "Tim Duncan"} :player{age: 42, name: "Tim Duncan"}) |
+-----+
```

匹配点的属性

用户可以在 Tag 的右侧用 {<prop_name>: <prop_value>} 表示模式中点的属性。

```
# 使用属性 name 搜索匹配的点。
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN v;
+-----+
| v |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+
```

使用 WHERE 子句也可以实现相同的操作：

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

 openCypher 兼容性

在 openCypher 9 中，= 是相等运算符，在 nGQL 中，== 是相等运算符，= 是赋值运算符。

使用 WHERE 子句直接匹配点的属性。

```
nebula> MATCH (v) \
    WITH v, properties(v) as props, keys(properties(v)) as kk \
    WHERE [i in kk where props[i] == "Tim Duncan"] \
    RETURN v;
+-----+
| v |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+
```

匹配点 ID

用户可以使用点 ID 去匹配点。`id()` 函数可以检索点的 ID。

```
nebula> MATCH (v) \
    WHERE id(v) == 'player101' \
    RETURN v;
+-----+
| v |
+-----+
| {"player101" :player{age: 36, name: "Tony Parker"}} |
+-----+
```

要匹配多个点的 ID，可以用 `WHERE id(v) IN [vid_list]` 或者 `WHERE id(v) IN {vid_list}`。

```
nebula> MATCH (v:player { name: 'Tim Duncan' })--(v2) \
    WHERE id(v2) IN ["player101", "player102"] \
    RETURN v2;
+-----+
| v2 |
+-----+
| {"player101" :player{age: 36, name: "Tony Parker"}) |
| {"player101" :player{age: 36, name: "Tony Parker"}) |
| {"player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----+
```



```
nebula> MATCH (v) WHERE id(v) IN {"player100", "player101"} \
    RETURN v.player.name AS name;
+-----+
| name |
+-----+
| "Tony Parker" |
| "Tim Duncan" |
+-----+
```

匹配连接的点

用户可以使用 `--` 符号表示两个方向的边，并匹配这些边连接的点。



在 nGQL 1.x 中，`--` 符号用于行内注释，从 nGQL 2.x 起，`--` 符号表示出边或入边，不再用于注释。

```
nebula> MATCH (v:player{name:'Tim Duncan"})--(v2:player) \
    RETURN v2.player.name AS Name;
+-----+
| Name |
+-----+
| "Manu Ginobili" |
| "Manu Ginobili" |
| "Dejounte Murray" |
...
```

用户可以在 `--` 符号上增加 `<` 或 `>` 符号指定边的方向。

```
# -->表示边从 v 开始，指向 v2。对于点 v 来说是出边，对于点 v2 来说是入边。
nebula> MATCH (v:player{name:'Tim Duncan"})->(v2:player) \
    RETURN v2.player.name AS Name;
+-----+
| Name |
+-----+
| "Tony Parker" |
| "Manu Ginobili" |
+-----+
```

如果需要判断目标点，可以使用 CASE 表达式。

```
nebula> MATCH (v:player{name:'Tim Duncan"})--(v2) \
    RETURN \
    CASE WHEN v2.team.name IS NOT NULL \
    THEN v2.team.name \
    WHEN v2.player.name IS NOT NULL \
    THEN v2.player.name END AS Name;
+-----+
| Name |
+-----+
| "Manu Ginobili" |
| "Manu Ginobili" |
| "Spurs" |
+-----+
```

```
| "Dejounte Murray" |
...
```

如果需要扩展模式，可以增加更多点和边。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-->(v2)<--(v3) \
    RETURN v3.player.name AS Name;
+-----+
| Name
+-----+
| "Dejounte Murray"
| "LaMarcus Aldridge"
| "Marco Belinelli"
...
...
```

如果不需引用点，可以省略括号中表示点的变量。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-->()<--(v3) \
    RETURN v3.player.name AS Name;
+-----+
| Name
+-----+
| "Dejounte Murray"
| "LaMarcus Aldridge"
| "Marco Belinelli"
...
...
```

匹配路径

连接起来的点和边构成了路径。用户可以使用自定义变量命名路径。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-->(v2) \
    RETURN p;
+-----+
| p
+-----+
| <"player100" :player{age: 42, name: "Tim Duncan"}-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})> |
| <"player100" :player{age: 42, name: "Tim Duncan"}-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})> |
| <"player100" :player{age: 42, name: "Tim Duncan"}-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})> |
+-----+
...
```

↑ openCypher 兼容性

在 nGQL 中，@ 符号表示边的 rank，在 openCypher 中，没有 rank 概念。

匹配边

```
nebula> MATCH ()<-[e]-() \
    RETURN e \
    LIMIT 3;
+-----+
| e
+-----+
| [:follow "player101"->"player102" @0 {degree: 90}] |
| [:follow "player103"->"player102" @0 {degree: 70}] |
| [:follow "player135"->"player102" @0 {degree: 80}] |
+-----+
```

匹配 EDGE TYPE

和点一样，用户可以用 :<edge_type> 表示模式中的 Edge type，例如 -[e:follow]-。

↑ 版本兼容性

在 NebulaGraph 3.0.0 之前，匹配 Edge Type 的前提是 Edge type 本身有对应属性的索引，否则，用户无法基于 Edge Type 执行 MATCH 语句。从 NebulaGraph 3.0.0 开始，匹配 Edge Type 可以不创建索引，但需要使用 LIMIT 限制输出结果数量，并且必须指定边的方向。从 NebulaGraph 3.5.0 开始，无需为 Edge Type 创建索引或者使用 LIMIT 限制输出结果数量，即可使用 MATCH 语句匹配边。

```
nebula> MATCH ()-[e:follow]->() \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player102"->"player100" @0 {degree: 75}] |
```

```
| [:follow "player102"->"player101" @0 {degree: 75}] |
| [:follow "player129"->"player116" @0 {degree: 90}] |
...
```

匹配边的属性

用户可以用 {<prop_name>: <prop_value>} 表示模式中 Edge type 的属性，例如 [e:follow{likeness:95}]。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e:follow{degree:95}]->(v2) \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player100"->"player101" @0 {degree: 95}] |
| [:follow "player100"->"player125" @0 {degree: 95}] |
+-----+
```

使用 WHERE 子句直接匹配边的属性。

```
nebula> MATCH ()-[e]->() \
    WITH e, properties(e) as props, keys(properties(e)) as kk \
    WHERE [i in kk where props[i] == 90] \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player125"->"player100" @0 {degree: 90}] |
| [:follow "player140"->"player114" @0 {degree: 90}] |
| [:follow "player133"->"player144" @0 {degree: 90}] |
| [:follow "player133"->"player114" @0 {degree: 90}] |
...
+-----+
```

匹配多个 EDGE TYPE

使用 | 可以匹配多个 Edge type，例如 [e:follow|:serve]。第一个 Edge type 前的英文冒号（:）不可省略，后续 Edge type 前的英文冒号可以省略，例如 [e:follow|serve]。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e:follow|:serve]->(v2) \
    RETURN e;
+-----+
| e
+-----+
| [:follow "player100"->"player101" @0 {degree: 95}] |
| [:follow "player100"->"player125" @0 {degree: 95}] |
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] |
+-----+
```

匹配多条边

用户可以扩展模式，匹配路径中的多条边。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[]->(v2)<-[e:serve]-(v3) \
    RETURN v2, v3;
+-----+-----+
| v2      | v3
+-----+-----+
| ("team204" :team{name: "Spurs"}) | ("player104" :player{age: 32, name: "Marco Belinelli"}) |
| ("team204" :team{name: "Spurs"}) | ("player101" :player{age: 36, name: "Tony Parker"}) |
| ("team204" :team{name: "Spurs"}) | ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
...
```

匹配定长路径

用户可以在模式中使用 :<edge_type>*<hop> 匹配定长路径。hop 必须是一个非负整数。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \
    RETURN DISTINCT v2 AS Friends;
+-----+
| Friends
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
| ("player125" :player{age: 41, name: "Manu Ginobili"}) |
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----+
```

如果 hop 为 0，模式会匹配路径上的起始点。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) -[*0]-> (v2) \
    RETURN v2;
+-----+
```

```
+-----+  
| v2 |  
+-----+  
| ("player100" :player{age: 42, name: "Tim Duncan"}) |  
+-----+
```

Note

在对匹配的多跳边进行过滤时，如对 `-[e:follow*2]->` 中的 `e` 进行过滤，此时的 `e` 不再是单条边的数据类型，而是一个包含多条边的列表，例如：

以下语句可以运行但是没有返回数据，因为 `e` 是一个列表，没有 `.degree` 的属性。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \  
 WHERE e.degree > 1 \  
 RETURN DISTINCT v2 AS Friends;
```

这是正确的表达：

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \  
 WHERE ALL(e_ IN e WHERE e_.degree > 0) \  
 RETURN DISTINCT v2 AS Friends;
```

进一步，这是表达对多跳边的第一跳的边属性过滤的表达：

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \  
 WHERE e[0].degree > 98 \  
 RETURN DISTINCT v2 AS Friends;
```

匹配变长路径

用户可以在模式中使用 `:<edge_type>*[minHop..maxHop]` 匹配变长路径。

Caution

如果未设置 `maxHop` 可能会导致 graph 服务 OOM，请谨慎执行该命令。

参数	说明
<code>minHop</code>	可选项。表示路径的最小长度。 <code>minHop</code> 必须是一个非负整数，默认值为 1。
<code>maxHop</code>	可选项。表示路径的最大长度。 <code>maxHop</code> 必须是一个非负整数，默认值为无穷大。

如果未指定 `minHop` 和 `maxHop`，仅设置了 `:<edge_type>*`，则二者都应用默认值，即 `minHop` 为 1，`maxHop` 为无穷大。

```
+-----+  
| Friends |  
+-----+  
| ("player125" :player{age: 41, name: "Manu Ginobili"}) |  
| ("player101" :player{age: 36, name: "Tony Parker"}) |  
...  
  
+-----+  
| Friends |  
+-----+  
| ("player101" :player{age: 36, name: "Tony Parker"}) |  
| ("player125" :player{age: 41, name: "Manu Ginobili"}) |  
| ("player100" :player{age: 42, name: "Tim Duncan"}) |  
...  
  
+-----+  
| Friends |  
+-----+  
| ("player125" :player{age: 41, name: "Manu Ginobili"}) |  
| ("player101" :player{age: 36, name: "Tony Parker"}) |  
| ("player100" :player{age: 42, name: "Tim Duncan"}) |  
...
```

用户可以使用 `DISTINCT` 关键字聚合重复结果。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*1..3]->(v2:player) \
    RETURN DISTINCT v2 AS Friends, count(v2);
+-----+-----+
| Friends | count(v2) |
+-----+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) | 1 |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | 4 |
| ("player101" :player{age: 36, name: "Tony Parker"}) | 3 |
| ("player125" :player{age: 41, name: "Manu Ginobili"}) | 3 |
+-----+-----+
```

如果 `minHop` 为 0，模式会匹配路径上的起始点。例如，与上个示例相比，下面的示例设置 `minHop` 为 0。此时，因为表示 "Tim Duncan" 的点是路径的起始点，所以它在结果集中的计数为 5，比在上个示例的结果中多计一次。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*0..3]->(v2:player) \
    RETURN DISTINCT v2 AS Friends, count(v2);
+-----+-----+
| Friends | count(v2) |
+-----+-----+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) | 1 |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | 5 |
| ("player125" :player{age: 41, name: "Manu Ginobili"}) | 3 |
| ("player101" :player{age: 36, name: "Tony Parker"}) | 3 |
+-----+-----+
```

Note

当在模式中使用变量 `e` 匹配定长或者变长路径时，例如 `-[e:follow*0..3]->`，不支持在其他模式中引用 `e`。例如，不支持以下语句：

```
nebula> MATCH (v:player)-[e:like*1..3]->(n) \
    WHERE (n)-[e*1..4]->(:player) \
    RETURN v;
```

匹配多个 EDGE TYPE 的变长路径

用户可以在变长或定长模式中指定多个 Edge type。`hop`、`minHop` 和 `maxHop` 对所有 Edge type 都生效。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow|serve*2]->(v2) \
    RETURN DISTINCT v2;
+-----+
| v2 |
+-----+
| ("team204" :team{name: "Spurs"}) |
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
| ("team215" :team{name: "Hornets"}) |
| ("player125" :player{age: 41, name: "Manu Ginobili"}) |
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----+
```

匹配多个模式

用户可以用英文逗号 (,) 分隔多个模式。

```
nebula> CREATE TAG INDEX IF NOT EXISTS team_index ON team(name(20));
nebula> REBUILD TAG INDEX team_index;
nebula> MATCH (v1:player{name:"Tim Duncan"}), (v2:team{name:"Spurs"}) \
    RETURN v1,v2;
+-----+-----+
| v1 | v2 |
+-----+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | ("team204" :team{name: "Spurs"}) |
+-----+-----+
```

匹配最短路径

用户可以使用 `allShortestPaths` 返回起始点到目标点的所有最短路径。

```
nebula> MATCH p = allShortestPaths((a:player{name:"Tim Duncan"})-[e*..5]-(b:player{name:"Tony Parker"})) \
    RETURN p;
+-----+
| p |
+-----+
| <"player100" :player{age: 42, name: "Tim Duncan"}>-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})> |
| <"player100" :player{age: 42, name: "Tim Duncan"}>-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})> |
+-----+
```

用户可以使用 `shortestPath` 返回起始点到目标点的任意一条最短路径。

```
nebula> MATCH p = shortestPath((a:player{name:"Tim Duncan"})-[e*..5]-(b:player{name:"Tony Parker"})) \
    RETURN p;
+-----+
| p |
+-----+
| <("player100" :player{age: 42, name: "Tim Duncan"})<-[follow@0 {degree: 95}]-("player101" :player{age: 36, name: "Tony Parker"})> |
+-----+
```

多MATCH检索

不同的模式有不同的筛选条件时，可以使用多 `MATCH`，会返回模式完全匹配的行。

```
nebula> MATCH (m)-[]->(n) WHERE id(m)=="player100" \
    MATCH (n)-[]->(l) WHERE id(n)=="player125" \
    RETURN id(m),id(n),id(l);
+-----+-----+-----+
| id(m) | id(n) | id(l) |
+-----+-----+-----+
| "player100" | "player125" | "team204" |
| "player100" | "player125" | "player100" |
+-----+-----+-----+
```

OPTIONAL MATCH检索

参见 [OPTIONAL MATCH](#)。



NebulaGraph 3.4.1 中 `MATCH` 语句的性能和资源占用得到了优化。但对性能要求较高时，仍建议使用 `GO`, `LOOKUP`, `|` 和 `FETCH` 等来替代 `MATCH`。

最后更新: May 4, 2023

4.6.2 OPTIONAL MATCH



目前 OPTIONAL MATCH 为 Beta 功能，后续可能会有一定优化调整。

OPTIONAL MATCH 通常与 MATCH 语句一起使用，作为 MATCH 语句的可选项去匹配命中的模式，如果没有命中对应的模式，对应的列返回 NULL。

openCypher 兼容性

本文操作仅适用于 nGQL 中的 openCypher 方式。

使用限制

OPTIONAL MATCH 子句中暂不支持使用 WHERE 子句。

示例

MATCH 语句中使用 OPTIONAL MATCH 的示例如下：

```
nebula> MATCH (m)-[]->(n) WHERE id(m)=="player100" \
    OPTIONAL MATCH (n)-[]->(l) \
    RETURN id(m),id(n),id(l);
```

id(m)	id(n)	id(l)
"player100"	"team204"	NULL
"player100"	"player101"	"team204"
"player100"	"player101"	"team215"
"player100"	"player101"	"player100"
"player100"	"player101"	"player102"
"player100"	"player101"	"player125"
"player100"	"player125"	"team204"
"player100"	"player125"	"player100"

而使用多 MATCH，不使用 OPTIONAL MATCH 时，会返回模式完全匹配的行。示例如下：

```
nebula> MATCH (m)-[]->(n) WHERE id(m)=="player100" \
    MATCH (n)-[]->(l) \
    RETURN id(m),id(n),id(l);
```

id(m)	id(n)	id(l)
"player100"	"player101"	"team204"
"player100"	"player101"	"team215"
"player100"	"player101"	"player100"
"player100"	"player101"	"player102"
"player100"	"player101"	"player125"
"player100"	"player125"	"team204"
"player100"	"player125"	"player100"

最后更新: May 4, 2023

4.6.3 LOOKUP

LOOKUP 根据索引遍历数据。用户可以使用 LOOKUP 实现如下功能：

- 根据 WHERE 子句搜索特定数据。
- 通过 Tag 列出点：检索指定 Tag 的所有点 ID。
- 通过 Edge type 列出边：检索指定 Edge type 的所有边的起始点、目的点和 rank。
- 统计包含指定 Tag 的点或属于指定 Edge type 的边的数量。

OpenCypher 兼容性

本文操作仅适用于原生 nGQL。

注意事项

- 索引会导致写性能大幅降低。请不要随意在生产环境中使用索引，除非很清楚使用索引对业务的影响。
- 通过 Explain 命令查看选择的索引。



在 2.5.0 版本之前，如果用 LOOKUP 语句基于指定属性查询时该属性没有索引，系统将报错，而不会使用其它索引。

前提条件

请确保 LOOKUP 语句有至少一个索引可用。

如果已经存在相关的点、边或属性，必须在新创建索引后 **重建索引**，才能使其生效。

语法

```
LOOKUP ON {<vertex_tag> | <edge_type>}
[WHERE <expression> [AND <expression> ...]]
YIELD [DISTINCT] <return_list> [AS <alias>]
[<clause>];

<return_list>
  <prop_name> [AS <col_alias>] [, <prop_name> [AS <prop_alias>] ...];
```

- WHERE <expression>**：指定遍历的过滤条件，还可以结合布尔运算符 AND 和 OR 一起使用。详情请参见 [WHERE](#)。
- YIELD**：定义需要返回的输出。详情请参见 [YIELD](#)。
- DISTINCT**：聚合输出结果，返回去重后的结果集。
- AS**：设置别名。
- <clause>**：支持 ORDER BY、LIMIT 子句。

WHERE 语句限制

在 LOOKUP 语句中使用 WHERE 子句，不支持如下操作：

- \$- 和 \$^ 。
 - 在关系表达式中，不支持运算符两边都有字段名，例如 `tagName.prop1 > tagName.prop2` 。
 - 不支持运算表达式和函数表达式中嵌套 `AliasProp` 表达式。
 - 不支持 XOR 运算符。
 - 不支持除 STARTS WITH 之外的字符串操作。
 - 不支持过滤 `rank()` 。
 - 不支持图模式.

检索点

返回 Tag 为 player 且 name 为 Tony Parker 的点。

```

nebula> CREATE TAG INDEX IF NOT EXISTS index_player ON player(name(30), age);

nebula> REBUILD TAG INDEX index_player;
+-----+
| New Job Id |
+-----+
| 15          |
+-----+

nebula> LOOKUP ON player \
    WHERE player.name == "Tony Parker" \
        YIELD id(vertex);
+-----+
| id(VERTEX)   |
+-----+
| "player101"  |
+-----+


nebula> LOOKUP ON player \
    WHERE player.name == "Tony Parker" \
        YIELD properties(vertex).name AS name, properties(vertex).age AS age;
+-----+
| name      | age |
+-----+
| "Tony Parker" | 36 |
+-----+


nebula> LOOKUP ON player \
    WHERE player.age > 45 \
        YIELD id(vertex);
+-----+
| id(VERTEX)   |
+-----+
| "player144"  |
| "player140"  |
+-----+


nebula> LOOKUP ON player \
    WHERE player.name STARTS WITH "B" \
        AND player.age IN [22,30] \
        YIELD properties(vertex).name, properties(vertex).age;
+-----+
| properties(VERTEX).name | properties(VERTEX).age |
+-----+
| "Ben Simmons"          | 22                |
| "Blake Griffin"        | 30                |
+-----+


nebula> LOOKUP ON player \
    WHERE player.name == "Kobe Bryant" \
        YIELD id(vertex) AS VertexID, properties(vertex).name AS name | \
        GO FROM $-VertexID OVER serve \
        YIELD $.name, properties(edge).start_year, properties(edge).end_year, properties($$).name;
+-----+
| $.name      | properties(EDGE).start_year | properties(EDGE).end_year | properties($$).name |
+-----+
| "Kobe Bryant" | 1996                  | 2016                  | "Lakers"           |
+-----+

```

检索边

返回 Edge type 为 follow 且 degree 为 90 的边。

```
nebula> CREATE EDGE INDEX IF NOT EXISTS index_follow ON follow(degree);

nebula> REBUILD EDGE INDEX index_follow;
+-----+
| New Job Id |
+-----+
| 62          |
+-----+

nebula> LOOKUP ON follow \
    WHERE follow.degree == 90 YIELD edge AS e;
+-----+
| e      |
+-----+
| [:follow "player109"->"player125" @ {degree: 90}] |
| [:follow "player118"->"player120" @ {degree: 90}] |
| [:follow "player118"->"player131" @ {degree: 90}] |
...
+-----+
| e      |
+-----+
| [:follow "player109"->"player125" @ {degree: 90}] |
| [:follow "player118"->"player120" @ {degree: 90}] |
| [:follow "player118"->"player131" @ {degree: 90}] |
...
+-----+
| properties(edge).degree |
+-----+
| 90          |
| 90          |
...
+-----+
| properties(edge).degree as degree \
| ORDER BY $-.degree \
| LIMIT 10;
+-----+
| degree   |
+-----+
| -1        |
| -1        |
| 9         |
| 10        |
| 13        |
| 50        |
| 55        |
| 60        |
| 70        |
| 70        |
+-----+
| $-.DstVID | properties(edge).start_year | properties(edge).end_year | properties($$).name |
+-----+
| "player105" | 2018           | "Spurs"          |
| "player105" | 2009           | "Cavaliers"       |
| "player105" | 2018           | "Raptors"         |
+-----+
```

通过 Tag 列出所有的对应的点 / 通过 Edge type 列出边

如果需要通过 Tag 列出所有的点，或通过 Edge type 列出边，则 Tag、Edge type 或属性上必须有至少一个索引。

例如一个 Tag player 有属性 name 和 age , 为了遍历所有包含 Tag player 的点 ID, Tag player、属性 name 或属性 age 中必须有一个已经创建索引。

- 查找所有 Tag 为 player 的点 VID。

```
nebula> CREATE TAG IF NOT EXISTS player(name string,age int);
nebula> CREATE TAG INDEX IF NOT EXISTS player_index on player();

nebula> REBUILD TAG INDEX player_index;
+-----+
| New Job Id |
+-----+
| 66          |
+-----+

nebula> INSERT VERTEX player(name,age) \
VALUES "player100":("Tim Duncan", 42), "player101":("Tony Parker", 36);

# 列出所有的 player。类似于 MATCH (n:player) RETURN id(n) /*, n */.

nebula> LOOKUP ON player YIELD id(vertex);
+-----+
| id(VERTEX) |
+-----+
| "player100" |
| "player101" |
...
... 从结果中返回最前面的 4 行数据。
nebula> LOOKUP ON player YIELD id(vertex) | LIMIT 4;
+-----+
| id(VERTEX) |
+-----+
| "player105" |
| "player109" |
| "player111" |
| "player118" |
+-----+
```

- 查找 Edge type 为 follow 的所有边的信息。

```
nebula> CREATE EDGE IF NOT EXISTS follow(degree int);
nebula> CREATE EDGE INDEX IF NOT EXISTS follow_index on follow();

nebula> REBUILD EDGE INDEX follow_index;
+-----+
| New Job Id |
+-----+
| 88          |
+-----+

nebula> INSERT EDGE follow(degree) \
VALUES "player100"->"player101":(95);

# 列出所有的 follow 边。类似于 MATCH (s)-[e:follow]->(d) RETURN id(s), rank(e), id(d) /*, type(e) */.

nebula> LOOKUP ON follow YIELD edge AS e;
+-----+
| e           |
+-----+
| [:follow "player105"->"player100" @0 {degree: 70}] |
| [:follow "player105"->"player116" @0 {degree: 80}] |
| [:follow "player109"->"player100" @0 {degree: 80}] |
...
...
```

统计点或边

统计 Tag 为 player 的点和 Edge type 为 follow 的边。

```
nebula> LOOKUP ON player YIELD id(vertex)|\
YIELD COUNT(*) AS Player_Number;
+-----+
| Player_Number |
+-----+
| 51            |
+-----+

nebula> LOOKUP ON follow YIELD edge AS e| \
YIELD COUNT(*) AS Follow_Number;
+-----+
| Follow_Number |
+-----+
| 81            |
+-----+
```

 Note

使用 `SHOW STATS` 命令也可以统计点和边。

最后更新: May 4, 2023

4.6.4 GO

GO 从给定起始点开始遍历图。 GO 语句采用的路径类型是 `walk`，即遍历时点和边都可以重复。

openCypher 兼容性

本文操作仅适用于原生 nGQL。

语法

```
GO [[<M> TO] <N> {STEP|STEPS} ] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
YIELD [DISTINCT] <return_list>
[{ SAMPLE <sample_list> | <limit_by_list_clause> }]
[| GROUP BY {<col_name> | expression} | <position>] YIELD <col_name>
[| ORDER BY <expression> [{ASC | DESC}]]
[| LIMIT [<offset>,] <number_rows>];

<vertex_list> ::= 
    <vid> [, <vid> ...]
    
<edge_type_list> ::= 
    <edge_type> [, <edge_type> ...]
    | *
```

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

- <N> {STEP|STEPS}：指定跳数。如果没有指定跳数，默认值 N 为 1。如果 N 为 0，NebulaGraph 不会检索任何边。
- M TO N {STEP|STEPS}：遍历 M-N 跳的边。如果 M 为 0，输出结果和 N 为 1 相同，即 GO 0 TO 2 和 GO 1 TO 2 是相同的。
- <vertex_list>：用逗号分隔的点 ID 列表，或特殊的引用符 \$-.id。详情参见[管道符](#)。
- <edge_type_list>：遍历的 Edge type 列表。
- REVERSELY | BIDIRECT：默认情况下检索的是 <vertex_list> 的出边（正向），REVERSELY 表示反向，即检索入边；BIDIRECT 为双向，即检索正向和反向，通过返回 <edge_type>._type 字段判断方向，其正数为正向，负数为反向。
- WHERE <conditions>：指定遍历的过滤条件。用户可以在起始点、目的点和边使用 WHERE 子句，还可以结合 AND、OR、NOT、XOR 一起使用。详情参见[WHERE](#)。

Note

- 遍历多个 Edge type 时，WHERE 子句有一些限制。例如不支持 WHERE edge1.prop1 > edge2.prop2。
- GO 语句执行时先遍历所有的点，然后再根据过滤器条件进行过滤。

- YIELD [DISTINCT] <return_list>：定义需要返回的输出。<return_list> 建议使用[Schema 相关函数](#)，当前支持 src(edge)、dst(edge)、type(edge) 等，暂不支持嵌套函数。详情参见[YIELD](#)。
- SAMPLE <sample_list>：用于在结果集中取样。详情参见[SAMPLE](#)。
- <limit_by_list_clause>：用于在遍历过程中逐步限制输出数量。详情参见[LIMIT](#)。
- GROUP BY：根据指定属性的值将输出分组。详情参见[GROUP BY](#)。分组后需要再次使用 YIELD 定义需要返回的输出。
- ORDER BY：指定输出结果的排序规则。详情参见[ORDER BY](#)。

Note

没有指定排序规则时，输出结果的顺序不是固定的。

- LIMIT [<offset>,] <number_rows>：限制输出结果的行数。详情参见[LIMIT](#)。

示例

```
# 返回 player102 所属队伍。  
nebula> GO FROM "player102" OVER serve YIELD dst(edge);  
+-----+  
| dst(EDGE) |  
+-----+  
| "team203" |  
| "team204" |  
+-----+
```

```
# 返回距离 player102 两跳的朋友。  
nebula> GO 2 STEPS FROM "player102" OVER follow YIELD dst(edge);  
+-----+  
| dst(EDGE) |  
+-----+  
| "player101" |  
| "player125" |  
| "player100" |  
| "player102" |  
| "player125" |  
+-----+
```

```
# 添加过滤条件。  
nebula> GO FROM "player100", "player102" OVER serve \  
      WHERE properties(edge).start_year > 1995 \  
      \
```

```

YIELD DISTINCT properties($$).name AS team_name, properties(edge).start_year AS start_year, properties($^).name AS player_name;

+-----+-----+
| team_name | start_year | player_name |
+-----+-----+
| "Spurs" | 1997 | "Tim Duncan" |
| "Trail Blazers" | 2006 | "LaMarcus Aldridge" |
| "Spurs" | 2015 | "LaMarcus Aldridge" |
+-----+-----+

```

```

# 遍历多个 Edge type。属性没有值时，会显示`NULL`。
nebula> GO FROM "player100" OVER follow, serve \
    YIELD properties(edge).degree, properties(edge).start_year;
+-----+-----+
| properties(EDGE).degree | properties(EDGE).start_year |
+-----+-----+
| 95 | __NULL__ |
| 95 | __NULL__ |
| __NULL__ | 1997 |
+-----+-----+

```

```

# 返回 player100 入方向的邻居点。
nebula> GO FROM "player100" OVER follow REVERSELY \
    YIELD src(edge) AS destination;
+-----+
| destination |
+-----+
| "player101" |
| "player102" |
...

```

```

# 该 MATCH 查询与上一个 GO 查询具有相同的语义。
nebula> MATCH (v)->[e:follow]-(v2) WHERE id(v) == 'player100' \
    RETURN id(v2) AS destination;
+-----+
| destination |
+-----+
| "player101" |
| "player102" |
...

```

```

# 查询 player100 的朋友和朋友所属队伍。
nebula> GO FROM "player100" OVER follow REVERSELY \
    YIELD src(edge) AS id | \
    GO FROM $-.id OVER serve \
    WHERE properties($').age > 20 \
    YIELD properties($^).name AS FriendOf, properties($$).name AS Team;
+-----+-----+
| FriendOf | Team |
+-----+-----+
| "Boris Diaw" | "Spurs" |
| "Boris Diaw" | "Jazz" |
| "Boris Diaw" | "Suns" |
...

```

```

# 该 MATCH 查询与上一个 GO 查询具有相同的语义。
nebula> MATCH (v)->[e:follow]->(v2)->[e2:serve]-(v3) \
    WHERE id(v) == 'player100' \
    RETURN v2.player.name AS FriendOf, v3.team.name AS Team;
+-----+-----+
| FriendOf | Team |
+-----+-----+
| "Boris Diaw" | "Spurs" |
| "Boris Diaw" | "Jazz" |
| "Boris Diaw" | "Suns" |
...

```

```

# 查询 player100 1~2 跳内的朋友。
nebula> GO 1 TO 2 STEPS FROM "player100" OVER follow \
    YIELD dst(edge) AS destination;
+-----+
| destination |
+-----+
| "player101" |
| "player125" |
...

```

```

# 该 MATCH 查询与上一个 GO 查询具有相同的语义。
nebula> MATCH (v) -[e:follow*1..2]->(v2) \
    WHERE id(v) == "player100" \
    RETURN id(v2) AS destination;
+-----+
| destination |
+-----+
| "player100" |
| "player102" |
...

```

```

# 根据年龄段分组。
nebula> GO 2 STEPS FROM "player100" OVER follow \

```

```

    YIELD src(edge) AS src, dst(edge) AS dst, properties($$).age AS age \
    | GROUP BY $-.dst \
    YIELD $-.dst AS dst, collect_set($-.src) AS src, collect($-.age) AS age;
+-----+-----+-----+
| dst | src | age |
+-----+-----+-----+
| "player125" | ["player101"] | [41] |
| "player100" | ["player125", "player101"] | [42, 42] |
| "player102" | ["player101"] | [33] |
+-----+-----+-----+

```

```

# 分组并限制输出结果的行数。
nebula> $a = GO FROM "player100" OVER follow YIELD src(edge) AS src, dst(edge) AS dst; \
GO 2 STEPS FROM $a.dst OVER follow \
YIELD $a.src AS src, $a.dst, src(edge), dst(edge) \
| ORDER BY $-.src | OFFSET 1 LIMIT 2;
+-----+-----+-----+-----+
| src | $a.dst | src(EDGE) | dst(EDGE) |
+-----+-----+-----+-----+
| "player100" | "player125" | "player100" | "player101" |
| "player100" | "player101" | "player100" | "player125" |
+-----+-----+-----+-----+

```

```

# 在多个边上通过 IS NOT EMPTY 进行判断。
nebula> GO FROM "player100" OVER follow WHERE properties($$).name IS NOT EMPTY YIELD dst(edge);
+-----+
| dst(EDGE) |
+-----+
| "player125" |
| "player101" |
+-----+

```

最后更新: May 4, 2023

4.6.5 FETCH

FETCH 可以获取指定点或边的属性值。

openCypher 兼容性

本文操作仅适用于原生 nGQL。

获取点的属性值

语法

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
<vid> [, vid ...]
YIELD [DISTINCT] <return_list> [AS <alias>];
```

参数	说明
tag_name	Tag 名称。
*	表示当前图空间中的所有 Tag。
vid	点 ID。
YIELD	定义需要返回的输出。详情请参见 YIELD 。
AS	设置别名。

基于 **TAG** 获取点的属性值

在 FETCH 语句中指定 Tag 获取对应点的属性值。

```
nebula> FETCH PROP ON player "player100" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 42, name: "Tim Duncan"} |
+-----+
```

获取点的指定属性值

使用 YIELD 子句指定返回的属性。

```
nebula> FETCH PROP ON player "player100" \
    YIELD properties(vertex).name AS name;
+-----+
| name      |
+-----+
| "Tim Duncan" |
+-----+
```

获取多个点的属性值

指定多个点 ID 获取多个点的属性值，点之间用英文逗号（,）分隔。

```
nebula> FETCH PROP ON player "player101", "player102", "player103" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 33, name: "LaMarcus Aldridge"} |
| {age: 36, name: "Tony Parker"} |
| {age: 32, name: "Rudy Gay"} |
+-----+
```

基于多个 **TAG** 获取点的属性值

在 FETCH 语句中指定多个 Tag 获取属性值。Tag 之间用英文逗号（,）分隔。

```
# 创建新 Tag t1。
nebula> CREATE TAG IF NOT EXISTS t1(a string, b int);
```

```
# 为点 player100 添加 Tag t1。
nebula> INSERT VERTEX t1(a, b) VALUES "player100":("Hello", 100);

# 基于 Tag player 和 t1 获取点 player100 上的属性值。
nebula> FETCH PROP ON player, t1 "player100" YIELD vertex AS v;
+-----+
| v
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"} :t1{a: "Hello", b: 100}) |
+-----+
```

用户可以在 `FETCH` 语句中组合多个 `Tag` 和多个点。

```
nebula> FETCH PROP ON player, t1 "player100", "player103" YIELD vertex AS v;
+-----+
| v
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"} :t1{a: "Hello", b: 100}) |
| ("player103" :player{age: 32, name: "Rudy Gay"}) |
+-----+
```

在所有标签中获取点的属性值

在 `FETCH` 语句中使用 `*` 获取当前图空间所有标签里，点的属性值。

```
nebula> FETCH PROP ON * "player100", "player106", "team200" YIELD vertex AS v;
+-----+
| v
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"} :t1{a: "Hello", b: 100}) |
| ("player106" :player{age: 25, name: "Kyle Anderson"}) |
| ("team200" :team{name: "Warriors"}) |
+-----+
```

获取边的属性值

语法

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

参数	说明
<code>edge_type</code>	Edge type 名称。
<code>src_vid</code>	起始点 ID，表示边的起点。
<code>dst_vid</code>	目的点 ID，表示边的终点。
<code>rank</code>	边的 rank。可选参数，默认值为 0。起始点、目的点、Edge type 和 rank 可以唯一确定一条边。
<code>YIELD</code>	定义需要返回的输出。详情请参见 <code>YIELD</code> 。

获取边的所有属性值

```
# 获取连接 player100 和 team204 的边 serve 的所有属性值。
nebula> FETCH PROP ON serve "player100" -> "team204" YIELD properties(edge);
+-----+
| properties(EDGE) |
+-----+
| {end_year: 2016, start_year: 1997} |
+-----+
```

获取边的指定属性值

使用 `YIELD` 子句指定返回的属性。

```
nebula> FETCH PROP ON serve "player100" -> "team204" \
    YIELD properties(edge).start_year;
+-----+
| properties(EDGE).start_year |
+-----+
| 1997 |
+-----+
```

获取多条边的属性值

指定多个边模式 (`<src_vid> -> <dst_vid>[@<rank>]`) 获取多个边的属性值。模式之间用英文逗号 (,) 分隔。

```
nebula> FETCH PROP ON serve "player100" -> "team204", "player133" -> "team202" YIELD edge AS e;
+-----+
| e
+-----+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] |
| [:serve "player133"->"team202" @0 {end_year: 2011, start_year: 2002}] |
+-----+
```

基于 **RANK** 获取属性值

如果有多条边，起始点、目的点和 Edge type 都相同，可以通过指定 rank 获取正确的边属性值。

```
# 插入不同属性值、不同 rank 的边。
nebula> insert edge serve(start_year,end_year) \
    values "player100"->"team204"@1:(1998, 2017);

nebula> insert edge serve(start_year,end_year) \
    values "player100"->"team204"@2:(1990, 2018);

# 默认返回 rank 为 0 的边。
nebula> FETCH PROP ON serve "player100" -> "team204" YIELD edge AS e;
+-----+
| e
+-----+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] |
+-----+

# 要获取 rank 不为 0 的边，请在 FETCH 语句中设置 rank。
nebula> FETCH PROP ON serve "player100" -> "team204"@1 YIELD edge AS e;
+-----+
| e
+-----+
| [:serve "player100"->"team204" @1 {end_year: 2017, start_year: 1998}] |
+-----+
```

复合语句中使用 **FETCH**

将 **FETCH** 与原生 nGQL 结合使用是一种常见的方式，例如和 **GO** 一起。

```
# 返回从点 player101 开始的 follow 边的 degree 值。
nebula> GO FROM "player101" OVER follow \
    YIELD src(edge) AS s, dst(edge) AS d \
    | FETCH PROP ON follow $-.s -> $-.d \
    YIELD properties(edge).degree;
+-----+
| properties(EDGE).degree |
+-----+
| 95
| 90
| 95
+-----+
```

用户也可以通过自定义变量构建类似的查询。

```
nebula> $var = GO FROM "player101" OVER follow \
    YIELD src(edge) AS s, dst(edge) AS d; \
    FETCH PROP ON follow $var.s -> $var.d \
    YIELD properties(edge).degree;
+-----+
| properties(EDGE).degree |
+-----+
| 95
| 90
| 95
+-----+
```

更多复合语句的详情，请参见[复合查询（子句结构）](#)。

最后更新: May 4, 2023

4.6.6 SHOW

SHOW CHARSET

SHOW CHARSET 语句显示当前的字符集。

目前可用的字符集为 utf8 和 utf8mb4。默认字符集为 utf8。NebulaGraph 扩展 utf8 支持四字节字符，因此 utf8 和 utf8mb4 是等价的。

语法

```
SHOW CHARSET;
```

示例

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

参数

说明

Charset

字符集名称。

Description

字符集说明。

Default collation

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: May 4, 2023

SHOW COLLATION

SHOW COLLATION 语句显示当前的排序规则。

目前可用的排序规则为 `utf8_bin` 和 `utf8mb4_bin`。

- 当字符集为 `utf8`，默认排序规则为 `utf8_bin`。
- 当字符集为 `utf8mb4`，默认排序规则为 `utf8mb4_bin`。

语法

```
SHOW COLLATION;
```

示例

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

参数	说明
<code>Collation</code>	排序规则名称。
<code>Charset</code>	与排序规则关联的字符集名称。

最后更新: May 4, 2023

SHOW CREATE SPACE

SHOW CREATE SPACE 语句显示指定图空间的创建语句。

图空间的更多详细信息, 请参见 [CREATE SPACE](#)。

语法

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

示例

```
nebula> SHOW CREATE SPACE basketballplayer;
+-----+-----+
| Space | Create Space |
+-----+-----+
| "basketballplayer" | "CREATE SPACE `basketballplayer` (partition_num = 10, replica_factor = 1, charset = utf8, collate = utf8_bin, vid_type = FIXED_STRING(32))" |
+-----+-----+
```

最后更新: May 4, 2023

SHOW CREATE TAG/EDGE

`SHOW CREATE TAG` 语句显示指定 Tag 的基本信息。Tag 的更多详细信息, 请参见 [CREATE TAG](#)。

`SHOW CREATE EDGE` 语句显示指定 Edge type 的基本信息。Edge type 的更多详细信息, 请参见 [CREATE EDGE](#)。

语法

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

示例

```
nebula> SHOW CREATE TAG player;
+-----+-----+
| Tag   | Create Tag
+-----+-----+
| "player" | "CREATE TAG `player` (
|           |   `name` string NULL,
|           |   `age` int64 NULL
|           | ) ttl_duration = 0, ttl_col = """
+-----+-----+

nebula> SHOW CREATE EDGE follow;
+-----+-----+
| Edge  | Create Edge
+-----+-----+
| "follow" | "CREATE EDGE `follow` (
|           |   `degree` int64 NULL
|           | ) ttl_duration = 0, ttl_col = """"
+-----+-----+
```

最后更新: May 4, 2023

SHOW HOSTS

SHOW HOSTS 语句可以显示集群信息，包括端口、状态、leader、分片、版本等信息，或者指定显示 Graph、Storage、Meta 服务主机信息。

语法

```
SHOW HOSTS [GRAPH | STORAGE | META];
```



对于使用源码安装的NebulaGraph，执行添加了服务名的命令后，输出的信息中不显示版本信息。

示例

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution | Version |
+-----+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 8 | "docs:5, basketballplayer:3" | "docs:5, basketballplayer:3" | "3.4.1" |
| "storaged1" | 9779 | "ONLINE" | 9 | "basketballplayer:4, docs:5" | "docs:5, basketballplayer:4" | "3.4.1" |
| "storaged2" | 9779 | "ONLINE" | 8 | "basketballplayer:3, docs:5" | "docs:5, basketballplayer:3" | "3.4.1" |
+-----+-----+-----+-----+-----+-----+

nebula> SHOW HOSTS GRAPH;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Role | Git Info Sha | Version |
+-----+-----+-----+-----+-----+
| "graphd" | 9669 | "ONLINE" | "GRAPH" | "3ba41bd" | "3.4.1" |
| "graphd1" | 9669 | "ONLINE" | "GRAPH" | "3ba41bd" | "3.4.1" |
| "graphd2" | 9669 | "ONLINE" | "GRAPH" | "3ba41bd" | "3.4.1" |
+-----+-----+-----+-----+-----+

nebula> SHOW HOSTS STORAGE;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Role | Git Info Sha | Version |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd" | "3.4.1" |
| "storaged1" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd" | "3.4.1" |
| "storaged2" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd" | "3.4.1" |
+-----+-----+-----+-----+-----+

nebula> SHOW HOSTS META;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Role | Git Info Sha | Version |
+-----+-----+-----+-----+-----+
| "metad0" | 9559 | "ONLINE" | "META" | "3ba41bd" | "3.4.1" |
| "metad1" | 9559 | "ONLINE" | "META" | "3ba41bd" | "3.4.1" |
| "metad2" | 9559 | "ONLINE" | "META" | "3ba41bd" | "3.4.1" |
+-----+-----+-----+-----+-----+
```

最后更新: May 4, 2023

SHOW INDEX STATUS

SHOW INDEX STATUS 语句显示重建原生索引的作业状态，以便确定重建索引是否成功。

语法

```
SHOW {TAG | EDGE} INDEX STATUS;
```

示例

```
nebula> SHOW TAG INDEX STATUS;
+-----+-----+
| Name | Index Status |
+-----+-----+
| "date1_index" | "FINISHED" |
| "basketballplayer_all_tag_indexes" | "FINISHED" |
| "any_shape_geo_index" | "FINISHED" |
+-----+-----+

nebula> SHOW EDGE INDEX STATUS;
+-----+-----+
| Name | Index Status |
+-----+-----+
| "follow_index" | "FINISHED" |
+-----+-----+
```

相关文档

- [管理作业](#)
- [REBUILD NATIVE INDEX](#)

最后更新: May 4, 2023

SHOW INDEXES

SHOW INDEXES 语句可以列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。

语法

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

示例

```
nebula> SHOW TAG INDEXES;
+-----+-----+-----+
| Index Name | By Tag | Columns |
+-----+-----+-----+
| "player_index_0" | "player" | []      |
| "player_index_1" | "player" | ["name"] |
+-----+-----+-----+

nebula> SHOW EDGE INDEXES;
+-----+-----+-----+
| Index Name | By Edge | Columns |
+-----+-----+-----+
| "follow_index" | "follow" | []      |
+-----+-----+-----+
```



NebulaGraph 2.0.1 中， SHOW TAG/EDGE INDEXES 语句仅返回 Names。

最后更新: May 4, 2023

SHOW PARTS

SHOW PARTS 语句显示图空间中指定分片或所有分片的信息。

语法

```
SHOW PARTS [<part_id>];
```

示例

```
nebula> SHOW PARTS;
+-----+-----+-----+
| Partition ID | Leader | Peers | Losts |
+-----+-----+-----+
| 1 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
| 2 | "192.168.2.2:9779" | "192.168.2.2:9779" | "" |
| 3 | "192.168.2.3:9779" | "192.168.2.3:9779" | "" |
| 4 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
| 5 | "192.168.2.2:9779" | "192.168.2.2:9779" | "" |
| 6 | "192.168.2.3:9779" | "192.168.2.3:9779" | "" |
| 7 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
| 8 | "192.168.2.2:9779" | "192.168.2.2:9779" | "" |
| 9 | "192.168.2.3:9779" | "192.168.2.3:9779" | "" |
| 10 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
+-----+-----+-----+
nebula> SHOW PARTS 1;
+-----+-----+-----+
| Partition ID | Leader | Peers | Losts |
+-----+-----+-----+
| 1 | "192.168.2.1:9779" | "192.168.2.1:9779" | "" |
+-----+-----+-----+
```

返回结果的说明如下:

参数	说明
Partition ID	存储分片的 ID
Leader	分片对应的 Raft leader 副本的信息，包括 IP 地址与服务端口
Peers	分片对应的所有副本（leader 与 follower）的信息，包括 IP 地址与服务端口
Losts	分片对应的处于离线状态的副本信息，包括 IP 地址和服务端口

最后更新: May 4, 2023

SHOW ROLES

SHOW ROLES 语句显示分配给用户的角色信息。

根据登录的用户角色，返回的结果也有所不同：

- 如果登录的用户角色是 GOD，或者有权访问该图空间的 ADMIN，则返回该图空间内除 GOD 之外的所有用户角色信息。
- 如果登录的用户角色是有权访问该图空间 DBA、USER 或 GUEST，则返回自身的角色信息。
- 如果登录的用户角色没有权限访问该图空间，则返回权限错误。

关于角色的详情请参见[内置角色权限](#)。

语法

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

示例

```
nebula> SHOW ROLES in basketballplayer;
+-----+-----+
| Account | Role Type |
+-----+-----+
| "user1" | "ADMIN"   |
+-----+-----+
```

最后更新: May 4, 2023

SHOW SNAPSHOTS

SHOW SNAPSHOTS 语句显示所有快照信息。

快照的使用方式请参见[管理快照](#)。

角色要求

只有 GOD 角色的用户（即 root）才能执行 SHOW SNAPSHOTS 语句。

语法

```
SHOW SNAPSHOTS;
```

示例

```
nebula> SHOW SNAPSHOTS;
+-----+-----+
| Name      | Status | Hosts
+-----+-----+
| "SNAPSHOT_2020_12_16_11_13_55" | "VALID" | "storaged0:9779, storaged1:9779, storaged2:9779"
| "SNAPSHOT_2020_12_16_11_14_10"  | "VALID" | "storaged0:9779, storaged1:9779, storaged2:9779"
+-----+-----+
```

最后更新: May 4, 2023

SHOW SPACES

SHOW SPACES 语句显示现存的图空间。

如何创建图空间，请参见 [CREATE SPACE](#)。

语法

```
SHOW SPACES;
```

示例

```
nebula> SHOW SPACES;
+-----+
| Name |
+-----+
| "docs" |
| "basketballplayer" |
+-----+
```

最后更新: May 4, 2023

SHOW STATS

SHOW STATS 语句显示最近一次 SUBMIT JOB STATS 作业收集的图空间统计信息。

图空间统计信息包含：

- 点的总数
- 边的总数
- 每个 Tag 关联的点的总数
- 每个 Edge type 关联的边的总数



SHOW STATS 返回的不是实时数据。因为返回的数据是最近一次 **SUBMIT JOB STATS** 作业收集的数据。返回的数据可能包含 **TTL** 过期数据，该过期数据会在下次执行 **Compaction 操作** 时被删除并不纳入统计。

前提条件

在需要查看统计信息的图空间中执行 **SUBMIT JOB STATS**。详情请参见 **SUBMIT JOB STATS**。



SHOW STATS 的结果取决于最后一次执行的 **SUBMIT JOB STATS**。如果发生过新的写入或者更改，必须再次执行 **SUBMIT JOB STATS**，否则统计数据有错误。

语法

```
SHOW STATS;
```

示例

```
# 选择图空间。
nebula> USE basketballplayer;

# 执行 SUBMIT JOB STATS。
nebula> SUBMIT JOB STATS;
+-----+
| New Job Id |
+-----+
| 98          |
+-----+

# 确认作业执行成功。
nebula> SHOW JOB 98;
+-----+-----+-----+-----+-----+-----+
| Job Id(TaskId) | Command(Dest) | Status | Start Time | Stop Time | Error Code |
+-----+-----+-----+-----+-----+-----+
| 98           | "STATS"      | "FINISHED" | 2021-11-01T09:33:21.000000 | 2021-11-01T09:33:21.000000 | "SUCCEEDED" |
| 0            | "storaged2"   | "FINISHED" | 2021-11-01T09:33:21.000000 | 2021-11-01T09:33:21.000000 | "SUCCEEDED" |
| 1            | "storaged0"   | "FINISHED" | 2021-11-01T09:33:21.000000 | 2021-11-01T09:33:21.000000 | "SUCCEEDED" |
| 2            | "storaged1"   | "FINISHED" | 2021-11-01T09:33:21.000000 | 2021-11-01T09:33:21.000000 | "SUCCEEDED" |
| "Total:3"    | "Succeeded:3" | "Failed:0"  | "In Progress:0"           | ""           | ""           |
+-----+-----+-----+-----+-----+-----+

# 显示图空间统计信息。
nebula> SHOW STATS;
+-----+-----+
| Type | Name | Count |
+-----+-----+
| "Tag" | "player" | 51 |
| "Tag" | "team" | 30 |
| "Edge" | "follow" | 81 |
| "Edge" | "serve" | 152 |
| "Space" | "vertices" | 81 |
| "Space" | "edges" | 233 |
+-----+-----+
```

最后更新: May 4, 2023

SHOW TAGS/EDGES

SHOW TAGS 语句显示当前图空间内的所有 Tag。

SHOW EDGES 语句显示当前图空间内的所有 Edge type。

语法

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

示例

```
nebula> SHOW TAGS;
+-----+
| Name   |
+-----+
| "player" |
| "star"   |
| "team"   |
+-----+

nebula> SHOW EDGES;
+-----+
| Name   |
+-----+
| "follow" |
| "serve"  |
+-----+
```

最后更新: May 4, 2023

SHOW USERS

SHOW USERS 语句显示用户信息。

角色要求

只有 GOD 角色的用户（即 root）才能执行 SHOW USERS 语句。

语法

```
SHOW USERS;
```

示例

```
nebula> SHOW USERS;
+-----+-----+
| Account | IP Whitelist |
+-----+-----+
| "root" | ""           |
| "user1" | ""           |
| "user2" | "192.168.10.10" |
+-----+-----+
```

最后更新: May 4, 2023

SHOW SESSIONS

登录NebulaGraph数据库时，会创建对应会话，用户可以查询会话信息。

注意事项

- 执行 `exit` 退出登录时，客户端会调用 API `release`，释放会话并清除会话信息。如果没有正常退出，且没有在配置文件 `nebula-graphd.conf` 设置空闲会话超时时间（`session_idle_timeout_secs`），会话不会自动释放。对于未自动释放的会话，需要手动删除指定会话，详情参见[终止会话](#)。
- `SHOW SESSIONS` 查询所有 Graph 服务上的会话信息。
- `SHOW LOCAL SESSIONS` 从当前连接的 Graph 服务获取会话信息，不会查询其他 Graph 服务上的会话信息。
- `SHOW SESSION <Session_Id>` 查询指定 Session ID 的会话信息。

语法

```
SHOW [LOCAL] SESSIONS;
SHOW SESSION <Session_Id>;
```

示例

```
nebula> SHOW SESSIONS;
+-----+-----+-----+-----+-----+-----+-----+
| SessionId | UserName | SpaceName | CreateTime | UpdateTime | GraphAddr | Timezone | ClientIp |
+-----+-----+-----+-----+-----+-----+-----+
| 1651220858102296 | "root" | "basketballplayer" | 2022-04-29T08:27:38.102296 | 2022-04-29T08:50:46.282921 | "127.0.0.1:9669" | 0 | "127.0.0.1" |
| 1651199330300991 | "root" | "basketballplayer" | 2022-04-29T02:28:50.300991 | 2022-04-29T08:16:28.339038 | "127.0.0.1:9669" | 0 | "127.0.0.1" |
| 165111289947744 | "root" | "basketballplayer" | 2022-04-28T02:28:19.847744 | 2022-04-28T08:17:44.470210 | "127.0.0.1:9669" | 0 | "127.0.0.1" |
| 1651041092662100 | "root" | "basketballplayer" | 2022-04-27T06:31:32.662100 | 2022-04-27T07:01:25.200978 | "127.0.0.1:9669" | 0 | "127.0.0.1" |
| 1650959429593975 | "root" | "basketballplayer" | 2022-04-26T07:50:29.593975 | 2022-04-26T07:51:47.184810 | "127.0.0.1:9669" | 0 | "127.0.0.1" |
| 1650958897679595 | "root" | "" | 2022-04-26T07:41:37.679595 | 2022-04-26T07:41:37.683802 | "127.0.0.1:9669" | 0 | "127.0.0.1" |
+-----+-----+-----+-----+-----+-----+-----+
nebula> SHOW SESSION 1635254859271703;
+-----+-----+-----+-----+-----+-----+
| SessionId | UserName | SpaceName | CreateTime | UpdateTime | GraphAddr | Timezone | ClientIp |
+-----+-----+-----+-----+-----+-----+
| 1651220858102296 | "root" | "basketballplayer" | 2022-04-29T08:27:38.102296 | 2022-04-29T08:50:54.254384 | "127.0.0.1:9669" | 0 | "127.0.0.1" |
+-----+-----+-----+-----+-----+-----+
```

参数	说明
<code>SessionId</code>	会话 ID，唯一标识一个会话。
<code>UserName</code>	会话的登录用户名称。
<code>SpaceName</code>	用户当前所使用的图空间。刚登录时为空（ <code>""</code> ）。
<code>CreateTime</code>	会话的创建时间，即用户认证登录的时间。时区为配置文件中 <code>timezone_name</code> 指定的时区。
<code>UpdateTime</code>	用户有执行操作时，会更新此时间。时区为配置文件中 <code>timezone_name</code> 指定的时区。
<code>GraphAddr</code>	会话的 Graph 服务地址和端口。
<code>Timezone</code>	保留参数，暂无意义。
<code>ClientIp</code>	会话的客户端 IP 地址。

最后更新: May 4, 2023

SHOW QUERIES

SHOW QUERIES 语句可以查看当前 Session 中正在执行的查询请求信息。



如果需要终止查询, 请参见[终止查询](#)。

注意事项

- SHOW LOCAL QUERIES 从本地缓存获取当前 Session 中查询的状态, 几乎没有延迟。
- SHOW QUERIES 从 Meta 服务获取所有 Session 中的查询信息。这些信息会根据参数 session_reclaim_interval_secs 定义的周期同步到 Meta 服务, 因此在客户端获取到的信息可能属于上个同步周期。

语法

```
SHOW [LOCAL] QUERIES;
```

示例

```
nebula> SHOW LOCAL QUERIES;
+-----+-----+-----+-----+-----+-----+-----+-----+
| SessionID | ExecutionPlanID | User | Host | StartTime | DurationInUsec | Status | Query |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1625463842921750 | 46 | "root" | "192.168.x.x":9669 | 2021-07-05T05:44:19.502903 | 0 | "RUNNING" | "SHOW LOCAL QUERIES;" |
+-----+-----+-----+-----+-----+-----+-----+-----+

nebula> SHOW QUERIES;
+-----+-----+-----+-----+-----+-----+-----+-----+
| SessionID | ExecutionPlanID | User | Host | StartTime | DurationInUsec | Status | Query |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1625456037718757 | 54 | "user1" | "192.168.x.x":9669 | 2021-07-05T05:51:08.691318 | 1504502 | "RUNNING" | "MATCH p=(v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
+-----+-----+-----+-----+-----+-----+-----+-----+

# 返回耗时 TOP 10 的查询。
nebula> SHOW QUERIES | ORDER BY $-.DurationInUsec DESC | LIMIT 10;
+-----+-----+-----+-----+-----+-----+-----+-----+
| SessionID | ExecutionPlanID | User | Host | StartTime | DurationInUsec | Status | Query |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1625471375320831 | 98 | "user2" | "192.168.x.x":9669 | 2021-07-05T07:50:24.461779 | 2608176 | "RUNNING" | "MATCH (v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
| 1625456037718757 | 99 | "user1" | "192.168.x.x":9669 | 2021-07-05T07:50:24.910616 | 2159333 | "RUNNING" | "MATCH (v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

参数说明如下。

参数	说明
SessionID	会话 ID。
ExecutionPlanID	执行计划 ID。
User	执行查询的用户名。
Host	用户连接的服务器地址和端口。
StartTime	执行查询的开始时间。
DurationInUsec	执行查询的持续时长。单位: 微秒。
Status	查询的当前状态。
Query	查询语句。

最后更新: May 4, 2023

SHOW META LEADER

SHOW META LEADER 语句显示当前 Meta 集群的 leader 信息。

关于 Meta 服务的详细说明请参见 [Meta 服务](#)。

语法

```
SHOW META LEADER;
```

示例

```
nebula> SHOW META LEADER;
+-----+-----+
| Meta Leader | secs from last heart beat |
+-----+-----+
| "127.0.0.1:9559" | 3 |
+-----+-----+
```

参数

说明

Meta Leader

Meta 集群的 leader 信息，包括 leader 所在服务器的 IP 地址和端口。

secs from last heart beat

距离上次心跳的时间间隔。单位：秒。

最后更新: May 4, 2023

4.7 子句和选项

4.7.1 GROUP BY

GROUP BY 子句可以用于聚合数据。

openCypher 兼容性

本文操作仅适用于原生 nGQL。

用户也可以使用 openCypher 方式的 `count()` 函数聚合数据。

```
nebula> MATCH (v:player)<-[:follow]-(:player) RETURN v.player.name AS Name, count(*) as cnt ORDER BY cnt DESC;
+-----+-----+
| Name | cnt |
+-----+-----+
| "Tim Duncan" | 10 |
| "LeBron James" | 6 |
| "Tony Parker" | 5 |
| "Chris Paul" | 4 |
| "Manu Ginobili" | 4 |
+-----+-----+
...
```

语法

GROUP BY 子句可以聚合相同值的行，然后进行计数、排序和计算等操作。

GROUP BY 子句可以在管道符 (|) 之后和 YIELD 子句之前使用。

```
| GROUP BY <var> YIELD <var>, <aggregation_function(var)>
```

aggregation_function() 函数支持 `avg()`、`sum()`、`max()`、`min()`、`count()`、`collect()`、`std()`。

示例

```
# 查找所有连接到 player100 的点，并根据他们的姓名进行分组，返回姓名的出现次数。
nebula> GO FROM "player100" OVER follow BIDIRECT \
    YIELD properties($$).name as Name \
    | GROUP BY $-.Name \
    YIELD $-.Name as Player, count(*) AS Name_Count;
+-----+-----+
| Player | Name_Count |
+-----+-----+
| "Shaquille O'Neal" | 1 |
| "Tiago Splitter" | 1 |
| "Manu Ginobili" | 2 |
| "Boris Diaw" | 1 |
| "LaMarcus Aldridge" | 1 |
| "Tony Parker" | 2 |
| "Marco Belinelli" | 1 |
| "Dejounte Murray" | 1 |
| "Danny Green" | 1 |
| "Aron Baynes" | 1 |
+-----+-----+
```

```
# 查找所有连接到 player100 的点，并根据起始点进行分组，返回 degree 的总和。
nebula> GO FROM "player100" OVER follow \
    YIELD src(edge) AS player, properties(edge).degree AS degree \
    | GROUP BY $-.player \
    YIELD sum($-.degree);
+-----+
| sum($-.degree) |
+-----+
| 190 |
+-----+
```

`sum()` 函数详情请参见[内置数学函数](#)。

隐式分组

在上述 nGQL 语句中明确写出 GROUP BY 并起到分组字段作用的用法称为 GROUP BY 显示用法；而在 openCypher 语句中 GROUP BY 的用法是隐式的，即在语句中不用写出 GROUP BY 也可起到分组字段的作用。nGQL 语句中显示地 GROUP BY 用法与 openCypher 语句中的隐式地 GROUP BY 用法相同，并且 nGQL 语句兼容 openCypher 的用法，即也支持隐式地使用 GROUP BY。有关 GROUP BY 的隐式用法，请参见[how-to-make-group-by-in-a-cypher-query](#)。

例如：查询 34 岁以上的球员中完全重叠服役的区间。

```
nebula> LOOKUP ON player WHERE player.age > 34 YIELD id(vertex) AS v | \
    GO FROM $-.v OVER serve YIELD serve.start_year AS start_year, serve.end_year AS end_year | \
    YIELD $-.start_year, $-.end_year, count(*) AS count | \
    ORDER BY $-.count DESC | LIMIT 5;
+-----+-----+-----+
| $-.start_year | $-.end_year | count |
+-----+-----+-----+
| 2018          | 2019          | 3   |
| 1998          | 2004          | 2   |
| 2012          | 2013          | 2   |
| 2007          | 2012          | 2   |
| 2010          | 2011          | 2   |
+-----+-----+-----+
```

最后更新: May 4, 2023

4.7.2 LIMIT

`LIMIT` 子句限制输出结果的行数。`LIMIT` 在原生 nGQL 语句和 openCypher 兼容语句中的用法有所不同。

- 在原生 nGQL 语句中，一般需要在 `LIMIT` 子句前使用管道符，可以直接在 `LIMIT` 语句后设置或者省略偏移量参数。
- 在 openCypher 兼容语句中，不允许在 `LIMIT` 子句前使用管道符，可以使用 `SKIP` 指明偏移量。

Note

在原生 nGQL 或 openCypher 方式中使用 `LIMIT` 时，使用 `ORDER BY` 子句限制输出顺序非常重要，否则会输出一个不可预知的子集。

原生 nGQL 语句中的 LIMIT

在原生 nGQL 中，`LIMIT` 有通用语法和 GO 语句中的专属语法。

原生 nGQL 中的通用 LIMIT 语法

原生 nGQL 中的通用 `LIMIT` 语法与 SQL 中的 `LIMIT` 原理相同。`LIMIT` 子句接收一个或两个参数，参数的值必须是非负整数，且必须用在管道符之后。语法和说明如下：

```
... | LIMIT [<offset>[,] <number_rows>;
```

参数	说明
<code>offset</code>	偏移量，即定义从哪一行开始返回。索引从 0 开始。默认值为 0，表示从第一行开始返回。
<code>number_rows</code>	返回的总行数。

示例：

```
# 从结果中返回最前面的 3 行数据。
nebula> LOOKUP ON player YIELD id(vertex)|\
    LIMIT 3;
+-----+
| id(VERTEX) |
+-----+
| "player100" |
| "player101" |
| "player102" |
+-----+  
  
# 从排序后结果中返回第 2 行开始的 3 行数据。
nebula> GO FROM "player100" OVER follow REVERSELY \
    YIELD properties($$.name AS Friend, properties($$).age AS Age \
    | ORDER BY $-.Age, $-.Friend \
    | LIMIT 1, 3;
+-----+-----+
| Friend | Age |
+-----+-----+
| "Danny Green" | 31 |
| "Aron Baynes" | 32 |
| "Marco Belinelli" | 32 |
+-----+-----+
```

GO 语句中的 LIMIT

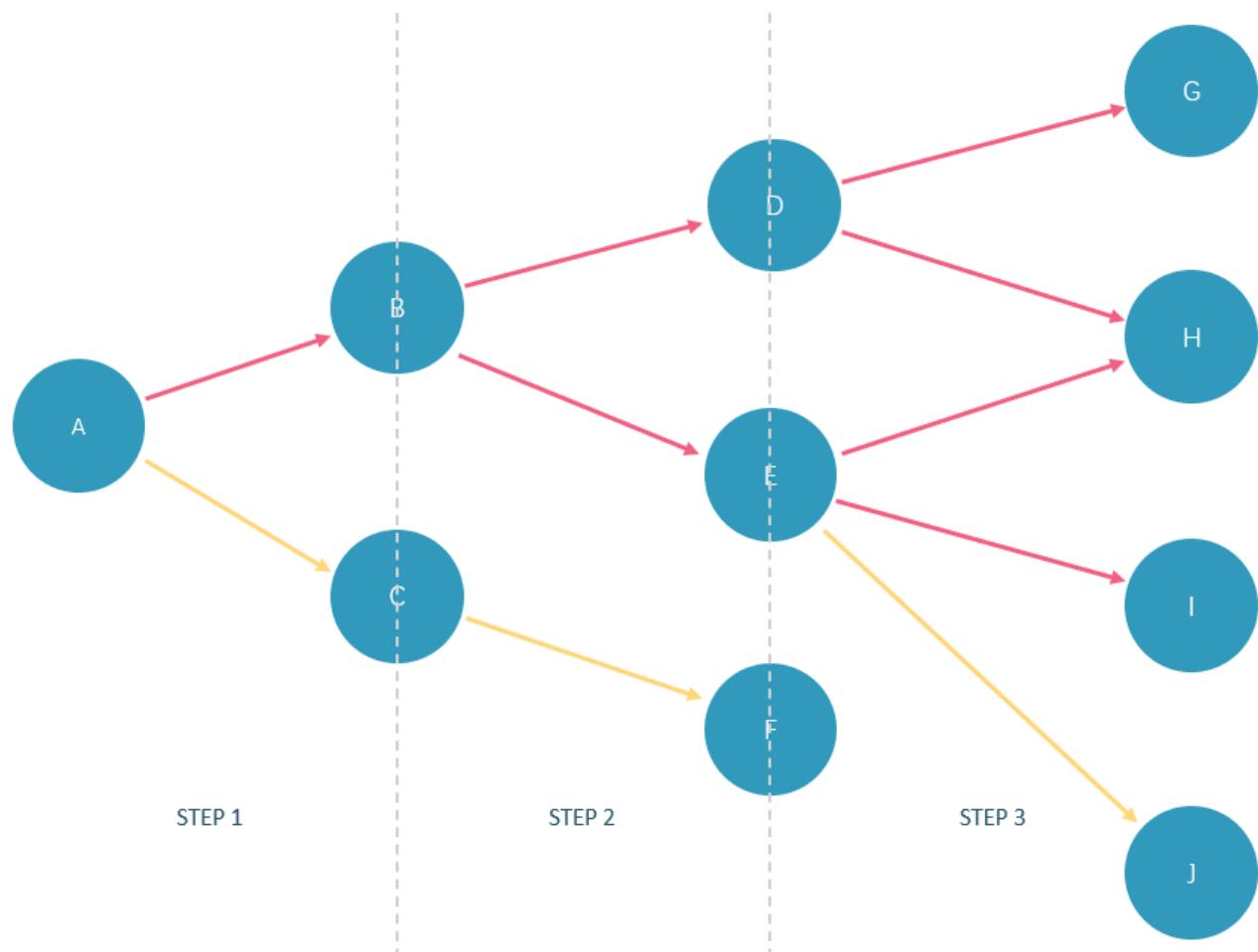
GO 语句中的 `LIMIT` 除了支持原生 nGQL 中的通用语法外，还支持根据边限制输出结果数量。

语法：

```
<go_statement> LIMIT <limit_list>;
```

`limit_list` 是一个列表，列表中的元素必须为自然数，且元素数量必须与 GO 语句中的 STEPS 的最大数相同。下文以 `GO 1 TO 3 STEPS FROM "A" OVER * LIMIT <limit_list>` 为例详细介绍 LIMIT 的这种用法。

- 列表 `limit_list` 必须包含 3 个自然数元素，例如 `GO 1 TO 3 STEPS FROM "A" OVER * LIMIT [1,2,4]`。
- `LIMIT [1,2,4]` 中的 1 表示系统在第一步时自动选择 1 条边继续遍历，2 表示在第二步时选择 2 条边继续遍历，4 表示在第三步时选择 4 条边继续遍历。
- 因为 `GO 1 TO 3 STEPS` 表示返回第一到第三步的所有遍历结果，因此下图中所有红色边和它们的原点与目的点都会被这条 GO 语句匹配上，而黄色边表示 GO 语句遍历时没有选择的路径。如果不是 `GO 1 TO 3 STEPS` 而是 `GO 3 STEPS`，则只会匹配上第三步的红色边和它们两端的点。



在 basketballplayer 数据集中的执行示例如下：

```
nebula> GO 3 STEPS FROM "player100" \
OVER * \
YIELD properties($$).name AS NAME, properties($$).age AS Age \
LIMIT [3,3,3];
+-----+-----+
| NAME | Age |
+-----+-----+
| "Tony Parker" | 36 |
| "Manu Ginobili" | 41 |
| "Spurs" | _NULL_ |
+-----+-----+

nebula> GO 3 STEPS FROM "player102" OVER * BIDIRECT \
YIELD dst(edge) \
LIMIT [rand32(5),rand32(5),rand32(5)];
+-----+
| dst(EDGE) |
+-----+
| "player100" |
| "player100" |
+-----+
```

openCypher 兼容语句中的 LIMIT

在 MATCH 等 openCypher 兼容语句中使用 LIMIT 不需要加管道符。语法和说明如下：

```
... [SKIP <offset>] [LIMIT <number_rows>];
```

参数	说明
offset	偏移量，即定义从哪一行开始返回。索引从 0 开始。默认值为 0，表示从第一行开始返回。
number_rows	返回的总行数量。

offset 和 number_rows 可以使用表达式，但是表达式的结果必须是非负整数。

Note

两个整数组成的分数表达式会自动向下取整。例如 8/6 向下取整为 1。

单独使用 LIMIT

LIMIT 可以单独使用，返回指定数量的结果。

```
nebula> MATCH (v:player) RETURN v.player.name AS Name, v.player.age AS Age \
    ORDER BY Age LIMIT 5;
+-----+-----+
| Name | Age |
+-----+-----+
| "Luka Doncic" | 20 |
| "Ben Simmons" | 22 |
| "Kristaps Porzingis" | 23 |
| "Giannis Antetokounmpo" | 24 |
| "Kyle Anderson" | 25 |
+-----+-----+
```

单独使用 SKIP

SKIP 可以单独使用，用于设置偏移量，返回指定位置之后的数据。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.player.name AS Name, v2.player.age AS Age \
    ORDER BY Age DESC SKIP 1;
+-----+-----+
| Name | Age |
+-----+-----+
| "Manu Ginobili" | 41 |
| "Tony Parker" | 36 |
+-----+-----+

nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.player.name AS Name, v2.player.age AS Age \
    ORDER BY Age DESC SKIP 1+1;
+-----+-----+
| Name | Age |
+-----+-----+
| "Tony Parker" | 36 |
+-----+-----+
```

同时使用 SKIP 与 LIMIT

同时使用 SKIP 与 LIMIT 可以返回从指定位置开始的指定数量的数据。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.player.name AS Name, v2.player.age AS Age \
    ORDER BY Age DESC SKIP 1 LIMIT 1;
+-----+-----+
| Name | Age |
+-----+-----+
| "Manu Ginobili" | 41 |
+-----+-----+
```

最后更新: May 4, 2023

4.7.3 SAMPLE

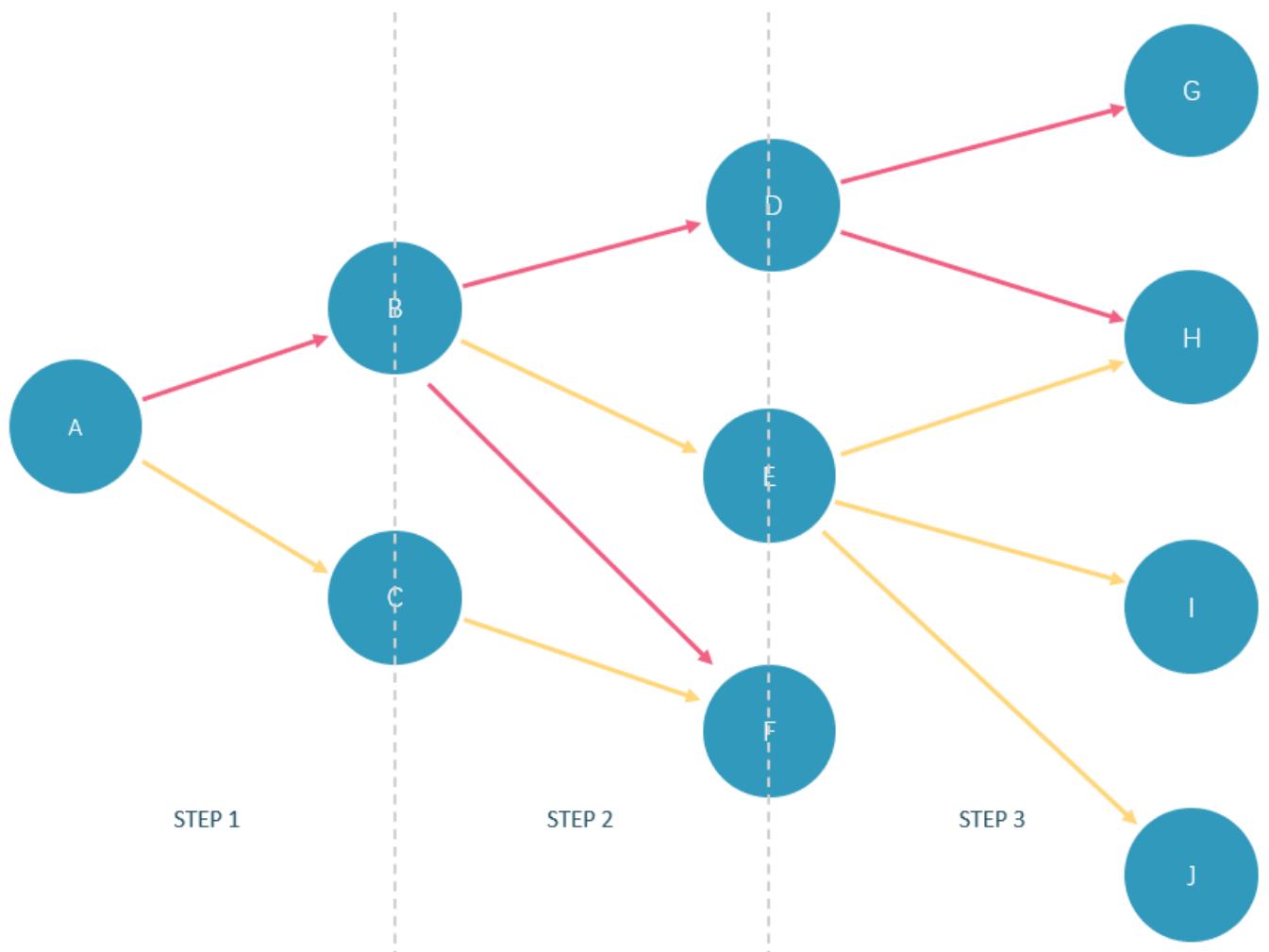
SAMPLE 子句用于在结果集中均匀取样并返回指定数量的数据。

SAMPLE 仅能在 GO 语句中使用，语法如下：

```
<go_statement> SAMPLE <sample_list>;
```

sample_list 是一个列表，列表中的元素必须为自然数，且元素数量必须与 GO 语句中的 STEPS 的最大数相同。下文以 GO 1 TO 3 STEPS FROM "A" OVER * SAMPLE [1,2,4] 为例详细介绍 SAMPLE 的用法。

- 列表 sample_list 必须包含 3 个自然数元素，例如 GO 1 TO 3 STEPS FROM "A" OVER * SAMPLE [1,2,4]。
- SAMPLE [1,2,4] 中的 1 表示系统在第一步时自动选择 1 条边继续遍历，2 表示在第二步时选择 2 条边继续遍历，4 表示在第三步时选择 4 条边继续遍历。如果某一步没有匹配的边或者匹配到的边数量小于指定数量，则按实际数量返回。
- 因为 GO 1 TO 3 STEPS 表示返回第一到第三步的所有遍历结果，因此下图中所有红色边和它们的原点与目的点都会被这条 GO 语句匹配上，而黄色边表示 GO 语句遍历时没有选择的路径。如果不是 GO 1 TO 3 STEPS 而是 GO 3 STEPS，则只会匹配上第三步的红色边和它们两端的点。



在 basketballplayer 数据集中的执行示例如下：

```
nebula> GO 3 STEPS FROM "player100" \
OVER * \
YIELD properties($$).name AS NAME, properties($$).age AS Age \
SAMPLE [1,2,3];
+-----+-----+
| NAME | Age |
+-----+-----+
| "Tony Parker" | 36 |
| "Manu Ginobili" | 41 |
| "Spurs" | __NULL__ |
```

```
+-----+-----+
nebula> GO 1 TO 3 STEPS FROM "player100" \
      OVER * \
      YIELD properties($$).name AS NAME, properties($$).age AS Age \
      SAMPLE [2,2,2];
+-----+-----+
| NAME          | Age   |
+-----+-----+
| "Manu Ginobili" | 41    |
| "Spurs"        | __NULL__ |
| "Tim Duncan"   | 42    |
| "Spurs"        | __NULL__ |
| "Manu Ginobili" | 41    |
| "Spurs"        | __NULL__ |
+-----+-----+
```

最后更新: May 4, 2023

4.7.4 ORDER BY

ORDER BY 子句指定输出结果的排序规则。

- 在原生 nGQL 中，必须在 YIELD 子句之后使用管道符 (|) 和 ORDER BY 子句。
- 在 openCypher 方式中，不允许使用管道符。在 RETURN 子句之后使用 ORDER BY 子句。

排序规则分为如下两种：

- ASC (默认)：升序。
- DESC：降序。

原生 nGQL 语法

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

Compatibility

原生 nGQL 语法中， ORDER BY 命令后必须使用引用符 \$-.。但在 2.5.0 之前的版本中不需要。

示例

```
nebula> FETCH PROP ON pLayer "player100", "player101", "player102", "player103" \
    YIELD properties(vertex).age AS age, properties(vertex).name AS name \
    | ORDER BY $-.age ASC, $-.name DESC;
+-----+
| age | name
+-----+
| 32 | "Rudy Gay"
| 33 | "LaMarcus Aldridge"
| 36 | "Tony Parker"
| 42 | "Tim Duncan"
+-----+
nebula> $var = GO FROM "player100" OVER follow \
    YIELD dst(edge) AS dst; \
    ORDER BY $var.dst DESC;
+-----+
| dst
+-----+
| "player125"
| "player101"
+-----+
```

OpenCypher 方式语法

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

示例

```
nebula> MATCH (v:player) RETURN v.player.name AS Name, v.player.age AS Age \
    ORDER BY Name DESC;
+-----+
| Name | Age
+-----+
| "Yao Ming" | 38
| "Vince Carter" | 42
| "Tracy McGrady" | 39
| "Tony Parker" | 36
| "Tim Duncan" | 42
+-----+
...
# 首先以年龄排序，如果年龄相同，再以姓名排序。
nebula> MATCH (v:player) RETURN v.player.age AS Age, v.player.name AS Name \
    ORDER BY Age DESC, Name ASC;
+-----+
| Age | Name
+-----+
| 47 | "Shaquille O'Neal"
```

46	"Grant Hill"
45	"Jason Kidd"
45	"Steve Nash"
...	

NULL 值的排序

升序排列时，会在输出的最后列出 **NULL** 值，降序排列时，会在输出的开头列出 **NULL** 值。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.player.name AS Name, v2.player.age AS Age \
    ORDER BY Age;
+-----+-----+
| Name | Age |
+-----+-----+
| "Tony Parker" | 36 |
| "Manu Ginobili" | 41 |
| __NULL__ | __NULL__ |
+-----+-----+

nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
    RETURN v2.player.name AS Name, v2.player.age AS Age \
    ORDER BY Age DESC;
+-----+-----+
| Name | Age |
+-----+-----+
| __NULL__ | __NULL__ |
| "Manu Ginobili" | 41 |
| "Tony Parker" | 36 |
+-----+-----+
```

最后更新: May 4, 2023

4.7.5 RETURN

`RETURN` 子句定义了 nGQL 查询的输出结果。如果需要返回多个字段，用英文逗号 (,) 分隔。

`RETURN` 可以引导子句或语句：

- `RETURN` 子句可以用于 nGQL 中的 `openCypher` 方式语句中，例如 `MATCH` 或 `UNWIND`。
- `RETURN` 可以单独使用，输出表达式的结果。

openCypher 兼容性

本文操作仅适用于 nGQL 中的 `openCypher` 方式。关于原生 nGQL 如何定义输出结果，请参见 [YIELD](#)。

`RETURN` 不支持如下 `openCypher` 功能：

- 使用不在英文字母表中的字符作为变量名。例如：

```
MATCH (`点 1`:{player}) \
RETURN `点 1`;
```

- 设置一个模式，并返回该模式匹配的所有元素。例如：

```
MATCH (v:{player}) \
RETURN (v)-[e]-(v2);
```

历史版本兼容性

- 在 nGQL 1.x 中，`RETURN` 适用于原生 nGQL，语法为 `RETURN <var_ref> IF <var_ref> IS NOT NULL`。
- 从 nGQL 2.0 开始，`RETURN` 不适用于原生 nGQL。

Map 顺序说明

`RETURN` 返回 Map 时，Key 的顺序是未定义的。

```
nebula> RETURN {age: 32, name: "Marco Belinelli"};
+-----+
| {age:32,name:"Marco Belinelli"} |
+-----+
| {age: 32, name: "Marco Belinelli"} |
+-----+
nebula> RETURN {zage: 32, name: "Marco Belinelli"};
+-----+
| {zage:32,name:"Marco Belinelli"} |
+-----+
| {name: "Marco Belinelli", zage: 32} |
+-----+
```

返回点或边

使用 `RETURN {<vertex_name> | <edge_name>}` 返回点或边的所有信息。

```
// 返回点
nebula> MATCH (v:{player}) \
    RETURN v;
+-----+
| v |
+-----+
| ("player104" :player{age: 32, name: "Marco Belinelli"}) |
| ("player107" :player{age: 32, name: "Aron Baynes"}) |
| ("player116" :player{age: 34, name: "LeBron James"}) |
| ("player120" :player{age: 29, name: "James Harden"}) |
| ("player125" :player{age: 41, name: "Manu Ginobili"}) |
+-----+
...
```

```
// 返回边
nebula> MATCH (v:player)-[e]->() \
    RETURN e;
+-----+
| e |
+-----+
| [:follow "player104"->"player100" @0 {degree: 55}] |
| [:follow "player104"->"player101" @0 {degree: 50}] |
| [:follow "player104"->"player105" @0 {degree: 60}] |
| [:serve "player104"->"team209" @0 {end_year: 2009, start_year: 2007}] |
| [:serve "player104"->"team208" @0 {end_year: 2016, start_year: 2015}] |
+-----+
...
```

返回点 ID

使用 `id()` 函数返回点 ID。

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

返回 Tag

使用 `labels()` 函数返回点上的 Tag 列表。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN labels(v);
+-----+
| labels(v) |
+-----+
| ["player"] |
+-----+
```

返回列表 `labels(v)` 中的第 N 个元素，可以使用 `labels(v)[n-1]`。例如下面示例使用 `labels(v)[0]` 检索第一个元素。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN labels(v)[0];
+-----+
| labels(v)[0] |
+-----+
| "player" |
+-----+
```

返回属性

使用语法 `<vertex_name>.<tag_name>.<property_name>` 返回点的属性；使用语法 `<edge_name>.<property_name>` 返回边的属性。

```
// 返回点的属性
nebula> MATCH (v:player) \
    RETURN v.player.name, v.player.age \
    LIMIT 3;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
| "Danny Green" | 31 |
| "Tiago Splitter" | 34 |
| "David West" | 38 |
+-----+-----+
```

```
// 返回边的属性
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
    RETURN e.start_year, e.degree \
+-----+-----+
| e.start_year | e.degree |
+-----+-----+
| __NULL__ | 95 |
| __NULL__ | 95 |
| 1997 | __NULL__ |
+-----+-----+
```

使用 `properties()` 函数返回点或边的所有属性。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
    RETURN properties(v2);
+-----+
| properties(v2) |
+-----+
| {name: "Spurs"} |
| {age: 36, name: "Tony Parker"} |
| {age: 41, name: "Manu Ginobili"} |
+-----+
```

返回 Edge type

使用 `type()` 函数返回匹配的 Edge type。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e]->() \
    RETURN DISTINCT type(e);
+-----+
| type(e) |
+-----+
| "serve" |
| "follow" |
+-----+
```

返回路径

使用 `RETURN <path_name>` 返回匹配路径的所有信息。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-*[3]->() \
    RETURN p;
+-----+
| p |
+-----+
| <"player100" :player{age: 42, name: "Tim Duncan"}->[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})->[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})->[:serve@0 {end_year: 2019, start_year: 2015}]->("team204" :team{name: "Spurs"})-> |
| <"player100" :player{age: 42, name: "Tim Duncan"}->[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})->[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})->[:serve@0 {end_year: 2015, start_year: 2006}]->("team203" :team{name: "Trail Blazers"})-> |
| <"player100" :player{age: 42, name: "Tim Duncan"}->[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})->[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})->[:follow@0 {degree: 75}]->("player101" :player{age: 36, name: "Tony Parker"})-> |
+-----+
| ...
```

返回路径中的点

使用 `nodes()` 函数返回路径中的所有点。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
    RETURN nodes(p);
+-----+
| nodes(p) |
+-----+
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("team204" :team{name: "Spurs"})] |
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("player101" :player{age: 36, name: "Tony Parker"})] |
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("player125" :player{age: 41, name: "Manu Ginobili"})] |
+-----+
```

返回路径中的边

使用 `relationships()` 函数返回路径中的所有边。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
    RETURN relationships(p);
+-----+
| relationships(p) |
+-----+
| [[:serve "player100" -> "team204" @0 {end_year: 2016, start_year: 1997}]] |
| [[:follow "player100" -> "player101" @0 {degree: 95}]] |
| [[:follow "player100" -> "player125" @0 {degree: 95}]] |
+-----+
```

返回路径长度

使用 `length()` 函数检索路径的长度。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[*..2]->(v2) \
    RETURN p AS Paths, length(p) AS Length;
+-----+
+-----+
| Paths
| Length |
+-----+
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:serve@0 {end_year: 2016, start_year: 1997})->("team204" :team{name: "Spurs"})>
| | 1   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player101" :player{age: 36, name: "Tony Parker"})>
| | 1   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player125" :player{age: 41, name: "Manu Ginobili"})>
| | 1   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player101" :player{age: 36, name: "Tony Parker"})->(:serve@0 {end_year: 2018, start_year: 1999})->("team204" :team{name: "Spurs"})>
| | 2   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player101" :player{age: 36, name: "Tony Parker"})->(:serve@0 {end_year: 2019, start_year: 2018})->("team215" :team{name: "Hornets"})>
| | 2   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player101" :player{age: 36, name: "Tony Parker"})->(:follow@0 {degree: 95})->("player100" :player{age: 42, name: "Tim Duncan"})>
| | 2   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player101" :player{age: 36, name: "Tony Parker"})->(:follow@0 {degree: 90})->("player102" :player{age: 33, name: "LaMarcus Aldridge"})>
| | 2   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player101" :player{age: 36, name: "Tony Parker"})->(:follow@0 {degree: 95})->("player125" :player{age: 41, name: "Manu Ginobili"})>
| | 2   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player125" :player{age: 41, name: "Manu Ginobili"})->(:serve@0 {end_year: 2018, start_year: 2002})->("team204" :team{name: "Spurs"})>
| | 2   |
| <"player100" :player{age: 42, name: "Tim Duncan"}->(:follow@0 {degree: 95})->("player125" :player{age: 41, name: "Manu Ginobili"})->(:follow@0 {degree: 90})->("player100" :player{age: 42, name: "Tim Duncan"})>
| | 2   |
+-----+
```

返回所有元素

使用星号 (*) 返回匹配模式中的所有元素。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
    RETURN *;
+-----+
| v |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+-----+  
  
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
    RETURN *;
+-----+-----+-----+
| v   | e   | v2  |
+-----+-----+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:follow "player100"->"player101" @0 {degree: 95}] | ("player101" :player{age: 36, name: "Tony Parker"}) |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:follow "player100"->"player125" @0 {degree: 95}] | ("player125" :player{age: 41, name: "Manu Ginobili"}) |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] | ("team204" :team{name: "Spurs"}) |
+-----+-----+-----+
```

重命名字段

使用语法 AS <alias> 重命名输出结果中的字段。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[:serve]->(v2) \
    RETURN v2.team.name AS Team;
+-----+
| Team |
+-----+
| "Spurs" |
+-----+  
  
nebula> RETURN "Amber" AS Name;
+-----+
| Name |
+-----+
| "Amber" |
+-----+
```

返回不存在的属性

如果匹配的结果中，某个属性不存在，会返回 NULL。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
    RETURN v2.player.name, type(e), v2.player.age;
+-----+-----+-----+
| v2.player.name | type(e) | v2.player.age |
+-----+-----+-----+
```

```
+-----+-----+
| "Manu Ginobili" | "follow" | 41
| _NULL_         | "serve"  | _NULL_
| "Tony Parker"  | "follow" | 36
+-----+-----+
```

返回表达式结果

RETURN 语句可以返回字面量、函数或谓词等表达式的结果。

```
nebula> MATCH (v:player{name:"Tony Parker"})-->(v2:player) \
    RETURN DISTINCT v2.player.name, "Hello"+ graphs!, v2.player.age > 35;
+-----+-----+
| v2.player.name | ("Hello"+ graphs!) | (v2.player.age>35) |
+-----+-----+
| "LaMarcus Aldridge" | "Hello graphs!" | false
| "Tim Duncan" | "Hello graphs!" | true
| "Manu Ginobili" | "Hello graphs!" | true
+-----+-----+

nebula> RETURN 1+1;
+-----+
| (1+1) |
+-----+
| 2 |
+-----+

nebula> RETURN 1- -1;
+-----+
| (1--(1)) |
+-----+
| 2 |
+-----+

nebula> RETURN 3 > 1;
+-----+
| (3>1) |
+-----+
| true |
+-----+

nebula> RETURN 1+1, rand32(1, 5);
+-----+-----+
| (1+1) | rand32(1,5) |
+-----+-----+
| 2     | 1      |
+-----+-----+
```

返回唯一字段

使用 **DISTINCT** 可以删除结果集中的重复字段。

```
# 未使用 DISTINCT。
nebula> MATCH (v:player{name:"Tony Parker"})--(v2:player) \
    RETURN v2.player.name, v2.player.age;
+-----+-----+
| v2.name | v2.age |
+-----+-----+
| "Tim Duncan" | 42
| "LaMarcus Aldridge" | 33
| "Marco Belinelli" | 32
| "Boris Diaw" | 36
| "Dejounte Murray" | 29
| "Tim Duncan" | 42
| "LaMarcus Aldridge" | 33
| "Manu Ginobili" | 41
+-----+-----+

# 使用 DISTINCT。
nebula> MATCH (v:player{name:"Tony Parker"})--(v2:player) \
    RETURN DISTINCT v2.player.name, v2.player.age;
+-----+-----+
| v2.name | v2.age |
+-----+-----+
| "Tim Duncan" | 42
| "LaMarcus Aldridge" | 33
| "Marco Belinelli" | 32
| "Boris Diaw" | 36
| "Dejounte Murray" | 29
| "Manu Ginobili" | 41
+-----+-----+
```

4.7.6 TTL

TTL (Time To Live) 指定属性的存活时间，超时后，该属性就会过期。

openCypher 兼容性

本文操作仅适用于原生 nGQL。

注意事项

- 不能修改带有 TTL 选项的属性的 Schema。
- TTL 和 INDEX 共存问题：
 - 如果一个 Tag/Edge type 的其中一属性已有 INDEX，则不能为其设置 TTL，也不能为该 Tag 的其他属性设置 TTL。
 - 如果已有 TTL，可以再添加 INDEX。

TTL 选项

nGQL 支持的 TTL 选项如下。

选项	说明
<code>ttl_col</code>	指定要设置存活时间的属性。属性的数据类型必须是 <code>int</code> 或者 <code>timestamp</code> 。
<code>ttl_duration</code>	指定时间戳差值，默认单位：秒。时间戳差值必须为 64 位非负整数。属性值和时间戳差值之和如果小于当前时间戳，属性就会过期。如果 <code>ttl_duration</code> 为 0，属性永不过期。 可在配置文件 <code>nebula-storaged.conf</code> （默认路径 <code>/usr/local/nightly/etc/</code> ）中设置 <code>ttl_use_ms</code> 为 <code>true</code> 将默认单位设为毫秒。

Caution

- 在设置 `ttl_use_ms` 为 `true` 前，请确保没有为属性设置 TTL，否则会因为过期时间缩短，导致数据被错误地删除。
- 在设置 `ttl_use_ms` 为 `true` 后，即设置 `ttl_duration` 的默认单位为毫秒后，`ttl_col` 的默认单位仍然为秒，它的数据类型必须是 `int`，并且需要手动转换属性值为毫秒。例如设置 `ttl_col` 为 `a`，则需要将 `a` 的值转换为毫秒，如当 `a` 的值为 `now()`，则需要将 `a` 的值设置为 `now() * 1000`。

属性过期

Caution

- 当为一个 Tag 或 Edge type 的属性设置 TTL 并该属性的值为 `NULL` 时，TTL 功能不会生效，即该属性永不过期。
- 如果为一个 Tag 或 Edge type 新增默认值为 `now()` 的属性并且该属性设置了 TTL，该 Tag 或 Edge type 相关的历史数据不会过期，因为历史数据的该属性值为当前时间戳。

点属性过期

点属性过期有如下影响：

- 如果一个点仅有一个 Tag，点上的一个属性过期，点也会过期。
- 如果一个点有多个 Tag，点上的一个属性过期，和该属性相同 Tag 的其他属性也会过期，但是点不会过期，点上其他 Tag 的属性保持不变。

边属性过期

因为一条边仅有一个 Edge type，边上的一个属性过期，边也会过期。

过期处理

属性过期后，对应的过期数据仍然存储在硬盘上，但是查询时会过滤过期数据。

NebulaGraph自动删除过期数据后，会在下一次 **Compaction** 过程中回收硬盘空间。

Note

如果**关闭 TTL 选项**，上一次 **Compaction** 之后的过期数据将可以被查询到。

使用 TTL 选项

TAG 或 EDGE TYPE 已存在

如果 Tag 和 Edge type 已经创建，请使用 ALTER 语句更新 Tag 或 Edge type。

```
# 创建 Tag。
nebula> CREATE TAG IF NOT EXISTS t1 (a timestamp);

# ALTER 修改 Tag，添加 TTL 选项。
nebula> ALTER TAG t1 TTL_COL = "a", TTL_DURATION = 5;

# 插入点，插入后 5 秒过期。
nebula> INSERT VERTEX t1(a) VALUES "101":(now());
```

TAG 或 EDGE TYPE 不存在

创建 Tag 或 Edge type 时可以同时设置 TTL 选项。详情请参见 **CREATE TAG** 和 **CREATE EDGE**。

```
# 创建 Tag 并设置 TTL 选项。
nebula> CREATE TAG IF NOT EXISTS t2(a int, b int, c string) TTL_DURATION= 100, TTL_COL = "a";

# 插入点。过期时间戳为 1648197238 (1648197138 + 100)。
nebula> INSERT VERTEX t2(a, b, c) VALUES "102":(1648197138, 30, "Hello");
```

删除存活时间

删除存活时间可以使用如下几种方法：

- 删除设置存活时间的属性。

```
nebula> ALTER TAG t1 DROP (a);
```

- 设置 `ttl_col` 为空字符串。

```
nebula> ALTER TAG t1 TTL_COL = "";
```

- 设置 `ttl_duration` 为 0。本操作可以保留 TTL 选项，属性永不过期，且属性的 Schema 无法修改。

```
nebula> ALTER TAG t1 TTL_DURATION = 0;
```

最后更新: May 4, 2023

4.7.7 WHERE

`WHERE` 子句可以根据条件过滤输出结果。

`WHERE` 子句通常用于如下查询：

- 原生 nGQL，例如 `GO` 和 `LOOKUP` 语句。
- `openCypher` 方式，例如 `MATCH` 和 `WITH` 语句。

openCypher 兼容性

过滤 `Rank` 是原生 nGQL 功能。如需在 `openCypher` 兼容语句中直接获取 `Rank` 值，可以使用 `rank()` 函数，例如

```
MATCH (:player)-[e:follow]->() RETURN rank(e);
```

基础用法



下文示例中的 `$$`、`$^` 等是引用符号，详情请参见[引用符](#)。

用布尔运算符定义条件

在 `WHERE` 子句中使用布尔运算符 `NOT`、`AND`、`OR` 和 `XOR` 定义条件。关于运算符的优先级，请参见[运算符优先级](#)。

```
nebula> MATCH (v:player) \
    WHERE v.player.name == "Tim Duncan" \
    XOR (v.player.age < 30 AND v.player.name == "Yao Ming") \
    OR NOT (v.player.name == "Yao Ming" OR v.player.name == "Tim Duncan") \
    RETURN v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
| "Danny Green" | 31      |
| "Tiago Splitter" | 34      |
| "David West" | 38      |
...
...
```

```
nebula> GO FROM "player100" \
    OVER follow \
    WHERE properties(edge).degree > 90 \
    OR properties($$).age != 33 \
    AND properties($$).name != "Tony Parker" \
    YIELD properties($$);
+-----+
| properties($$) |
+-----+
| {age: 41, name: "Manu Ginobili"} |
+-----+
```

过滤属性

在 WHERE 子句中使用点或边的属性定义条件。

- 过滤点属性:

```
nebula> MATCH (v:player)-[e]-(v2) \
    WHERE v2.player.age < 25 \
    RETURN v2.player.name, v2.player.age;
+-----+-----+
| v2.player.name | v2.player.age |
+-----+-----+
| "Ben Simmons" | 22 |
| "Luka Doncic" | 20 |
| "Kristaps Porzingis" | 23 |
+-----+-----+
```

```
nebula> GO FROM "player100" OVER follow \
    WHERE $^.player.age >= 42 \
    YIELD dst(edge);
+-----+
| dst(EDGE) |
+-----+
| "player101" |
| "player125" |
+-----+
```

- 过滤边属性:

```
nebula> MATCH (v:player)-[e]->() \
    WHERE e.start_year < 2000 \
    RETURN DISTINCT v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
| "Tony Parker" | 36 |
| "Tim Duncan" | 42 |
| "Grant Hill" | 46 |
...
```

```
nebula> GO FROM "player100" OVER follow \
    WHERE follow.degree > 90 \
    YIELD dst(edge);
+-----+
| dst(EDGE) |
+-----+
| "player101" |
| "player125" |
+-----+
```

过滤动态计算属性

```
nebula> MATCH (v:player) \
    WHERE v[toLowerCase("AGE")] < 21 \
    RETURN v.player.name, v.player.age;
+-----+-----+
| v.name | v.age |
+-----+-----+
| "Luka Doncic" | 20 |
+-----+-----+
```

过滤现存属性

```
nebula> MATCH (v:player) \
    WHERE exists(v.player.age) \
    RETURN v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
| "Danny Green" | 31 |
| "Tiago Splitter" | 34 |
| "David West" | 38 |
...
```

过滤 RANK

在 nGQL 中, 如果多个边拥有相同的起始点、目的点和属性, 则它们的唯一区别是 rank 值。在 WHERE 子句中可以使用 rank 过滤边。

```
# 创建测试数据。
nebula> CREATE SPACE IF NOT EXISTS test (vid_type=FIXED_STRING(30));
nebula> USE test;
```

```
nebula> CREATE EDGE IF NOT EXISTS e1(p1 int);
nebula> CREATE TAG IF NOT EXISTS 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);
```

通过 rank 过滤边，查找 rank 大于 2 的边。

```
nebula> GO FROM "1" \
    OVER e1 \
    WHERE rank(edge) > 2 \
    YIELD src(edge), dst(edge), rank(edge) AS Rank, properties(edge).p1 | \
    ORDER BY $-.Rank DESC;
+-----+-----+-----+
| src(EDGE) | dst(EDGE) | Rank | properties(EDGE).p1 |
+-----+-----+-----+
| "1" | "2" | 6 | 16 |
| "1" | "2" | 5 | 15 |
| "1" | "2" | 4 | 14 |
| "1" | "2" | 3 | 13 |
+-----+-----+-----+
```

通过 rank 过滤边，查找 rank 值等于 0 的 follow 边。

```
nebula> MATCH (v)-[e:follow]->() \
    WHERE rank(e)==0 \
    RETURN *;
+-----+
| v | e |
+-----+
| ("player142" :player{age: 29, name: "Klay Thompson"}) | [:follow "player142"-->"player117" @0 {degree: 90}] |
| ("player139" :player{age: 34, name: "Marc Gasol"}) | [:follow "player139"-->"player138" @0 {degree: 99}] |
| ("player108" :player{age: 36, name: "Boris Diaw"}) | [:follow "player108"-->"player100" @0 {degree: 80}] |
| ("player108" :player{age: 36, name: "Boris Diaw"}) | [:follow "player108"-->"player101" @0 {degree: 80}] |
...
...
```

过滤 PATTERN

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(t) \
    WHERE (v)-[e]->(t:team) \
    RETURN (v)-->();
+-----+
| (v)-->() = (v)--
>()
|
+-----+
| [<"player100" :player{age: 42, name: "Tim Duncan"}]-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})>] |
+-----+
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(t) \
    WHERE NOT (v)-[e]->(t:team) \
    RETURN (v)-->();
+-----+
| (v)-->() = (v)--
>()
|
+-----+
| [<"player100" :player{age: 42, name: "Tim Duncan"}]-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})>] |
+-----+
| [<"player100" :player{age: 42, name: "Tim Duncan"}]-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})>] |
+-----+
|
```

过滤字符串

在 WHERE 子句中使用 STARTS WITH、ENDS WITH 或 CONTAINS 可以匹配字符串的特定部分。匹配时区分大小写。

STARTS WITH

STARTS WITH 会从字符串的起始位置开始匹配。

```
# 查询姓名以 T 开头的 player 信息。
nebula> MATCH (v:player) \
```

```
WHERE v.player.name STARTS WITH "T" \
RETURN v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
| "Tony Parker" | 36      |
| "Tiago Splitter" | 34      |
| "Tim Duncan" | 42      |
| "Tracy McGrady" | 39      |
+-----+-----+
```

如果使用小写 t (STARTS WITH "t") , 会返回空集, 因为数据库中没有以小写 t 开头的姓名。

```
nebula> MATCH (v:player) \
    WHERE v.player.name STARTS WITH "t" \
    RETURN v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
Empty set (time spent 5080/6474 us)
```

ENDS WITH

ENDS WITH 会从字符串的结束位置开始匹配。

```
nebula> MATCH (v:player) \
    WHERE v.player.name ENDS WITH "r" \
    RETURN v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
| "Tony Parker" | 36      |
| "Tiago Splitter" | 34      |
| "Vince Carter" | 42      |
+-----+-----+
```

CONTAINS

CONTAINS 会检查关键字是否匹配字符串的某一部分。

```
nebula> MATCH (v:player) \
    WHERE v.player.name CONTAINS "Pa" \
    RETURN v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
| "Paul George" | 28      |
| "Tony Parker" | 36      |
| "Paul Gasol" | 38      |
| "Chris Paul" | 33      |
+-----+-----+
```

结合 NOT 使用

用户可以结合布尔运算符 NOT 一起使用, 否定字符串匹配条件。

```
nebula> MATCH (v:player) \
    WHERE NOT v.player.name ENDS WITH "R" \
    RETURN v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
| "Danny Green" | 31      |
| "Tiago Splitter" | 34      |
| "David West" | 38      |
| "Russell Westbrook" | 30      |
...
```

过滤列表

匹配列表中的值

使用 IN 运算符检查某个值是否在指定列表中。

```
nebula> MATCH (v:player) \
    WHERE v.player.age IN range(20,25) \
    RETURN v.player.name, v.player.age;
+-----+-----+
| v.player.name | v.player.age |
+-----+-----+
```

```
+-----+-----+
| "Ben Simmons" | 22 |
| "Giannis Antetokounmpo" | 24 |
| "Kyle Anderson" | 25 |
| "Joel Embiid" | 25 |
| "Kristaps Porzingis" | 23 |
| "Luka Doncic" | 20 |
+-----+-----+
```

```
nebula> LOOKUP ON player \
    WHERE player.age IN [25,28] \
    YIELD properties(vertex).name, properties(vertex).age;
+-----+-----+
| properties(VERTEX).name | properties(VERTEX).age |
+-----+-----+
| "Kyle Anderson" | 25 |
| "Damian Lillard" | 28 |
| "Joel Embiid" | 25 |
| "Paul George" | 28 |
| "Ricky Rubio" | 28 |
+-----+-----+
```

结合 NOT 使用

```
nebula> MATCH (v:player) \
    WHERE v.player.age NOT IN range(20,25) \
    RETURN v.player.name AS Name, v.player.age AS Age \
    ORDER BY Age;
+-----+-----+
| Name | Age |
+-----+-----+
| "Kyrie Irving" | 26 |
| "Cory Joseph" | 27 |
| "Damian Lillard" | 28 |
| "Paul George" | 28 |
| "Ricky Rubio" | 28 |
...
...
```

最后更新: May 4, 2023

4.7.8 YIELD

YIELD 定义 nGQL 查询的输出结果。

YIELD 可以引导子句或语句:

- **YIELD** 子句用于原生 nGQL 语句中，例如 **GO**、**FETCH** 或 **LOOKUP**，必须通过 **YIELD** 子句定义返回结果。
- **YIELD** 语句可以在独立查询或复合查询中使用。

openCypher 兼容性

本文操作仅适用于原生 nGQL。关于 openCypher 方式如何定义输出结果，请参见 [RETURN](#)。

YIELD 在 nGQL 和 openCypher 中有不同的函数:

- 在 openCypher 中，**YIELD** 用于在 **CALL[...YIELD]** 子句中指定过程调用的输出。

Note

nGQL 不支持 **CALL[...YIELD]**。

- 在 nGQL 中，**YIELD** 和 openCypher 中的 **RETURN** 类似。

Note

下文示例中的 **\$\$**、**\$-** 等是引用符号，详情请参见[引用符](#)。

YIELD 子句

语法

```
YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...];
```

参数	说明
DISTINCT	聚合输出结果，返回去重后的结果集。
col	要返回的字段。如果没有为字段设置别名，返回结果中的列名为 col 。
alias	col 的别名。使用关键字 AS 进行设置，设置后返回结果中的列名为该别名。

使用 YIELD 子句

- **GO** 语句中使用 **YIELD** :

```
nebula> GO FROM "player100" OVER follow \
    YIELD properties($$).name AS Friend, properties($$).age AS Age;
+-----+-----+
| Friend | Age |
+-----+-----+
| "Tony Parker" | 36 |
```

```
| "Manu Ginobili" | 41 |
+-----+-----+
```

- **FETCH** 语句中使用 **YIELD** :

```
nebula> FETCH PROP ON player "player100" \
          YIELD properties(vertex).name;
+-----+
| properties(VERTEX).name |
+-----+
| "Tim Duncan"           |
+-----+
```

- **LOOKUP** 语句中使用 **YIELD** :

```
nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
          YIELD properties(vertex).name, properties(vertex).age;
+-----+
| properties(VERTEX).name | properties(VERTEX).age |
+-----+
| "Tony Parker"          | 36                  |
+-----+
```

YIELD 语句

语法

```
YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...]
[WHERE <conditions>];
```

参数	说明
DISTINCT	聚合输出结果，返回去重后的结果集。
col	要按返回的字段。如果没有为字段设置别名，返回结果中的列名为 col。
alias	col 的别名。使用关键字 AS 进行设置，设置后返回结果中的列名为该别名。
conditions	在 WHERE 子句中设置的过滤条件。详情请参见 WHERE 。

复合查询中使用 YIELD 语句

在复合查询中，YIELD 语句可以接收、过滤、修改之前语句的结果集，然后输出。

```
# 查找 player100 关注的 player，并计算他们的平均年龄。
nebula> GO FROM "player100" OVER follow \
          YIELD dst(edge) AS ID \
          | FETCH PROP ON player $-.ID \
          YIELD properties(vertex).age AS Age \
          | YIELD AVG($-.Age) as Avg_age, count(*)as Num_friends;
+-----+
| Avg_age | Num_friends |
+-----+
| 38.5    | 2              |
+-----+
```

```
# 查找 player101 关注的 player，返回 degree 大于 90 的 player。
nebula> $var1 = GO FROM "player101" OVER follow \
          YIELD properties(edge).degree AS Degree, dst(edge) as ID; \
          YIELD $var1.ID AS ID WHERE $var1.Degree > 90;
+-----+
| ID   |
+-----+
| "player100" |
| "player125" |
+-----+
```

```
# 查找 player 中年龄大于 30 且小于 32 的点，返回去掉重复属性的值。
nebula> LOOKUP ON player \
          WHERE player.age < 32 and player.age >30 \
          YIELD DISTINCT properties(vertex).age as v;
+-----+
| v   |
+-----+
| 31  |
+-----+
```

独立使用 **YIELD** 语句

YIELD 可以计算表达式并返回结果。

```
nebula> YIELD rand32(1, 6);
+-----+
| rand32(1,6) |
+-----+
| 3           |
+-----+

nebula> YIELD "Hel" + "\tlo" AS string1, ", World!" AS string2;
+-----+-----+
| string1   | string2   |
+-----+-----+
| "Hel      lo" | ", World!" |
+-----+-----+

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

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

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

最后更新: May 4, 2023

4.7.9 WITH

`WITH` 子句可以获取并处理查询前半部分的结果，并将处理结果作为输入传递给查询的后半部分。

openCypher 兼容性

本文操作仅适用于 `openCypher` 方式。



在原生 `nGQL` 中，有与 `WITH` 类似的管道符，但它们的工作方式不同。不要在 `openCypher` 方式中使用管道符，也不要在原生 `nGQL` 中使用 `WITH` 子句。

组成复合查询

使用 `WITH` 子句可以组合语句，将一条语句的输出转换为另一条语句的输入。

示例 1

- 匹配一个路径。
- 通过 `nodes()` 函数将路径上的所有点输出到一个列表。
- 将列表拆分为行。
- 去重后返回点的信息。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})--() \
    WITH nodes(p) AS n \
    UNWIND n AS n1 \
    RETURN DISTINCT n1;
+-----+
| n1 |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
| ("player101" :player{age: 36, name: "Tony Parker"}) |
| ("team204" :team{name: "Spurs"}) |
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
| ("player125" :player{age: 41, name: "Manu Ginobili"}) |
| ("player104" :player{age: 32, name: "Marco Belinelli"}) |
| ("player144" :player{age: 47, name: "Shaquille O'Neal"}) |
| ("player105" :player{age: 31, name: "Danny Green"}) |
| ("player113" :player{age: 29, name: "Dejounte Murray"}) |
| ("player107" :player{age: 32, name: "Aron Baynes"}) |
| ("player109" :player{age: 34, name: "Tiago Splitter"}) |
| ("player108" :player{age: 36, name: "Boris Diaw"}) |
+-----+
```

示例 2

- 匹配点 ID 为 `player100` 的点。
- 通过 `labels()` 函数将点的所有 Tag 输出到一个列表。
- 将列表拆分为行。
- 返回结果。

```
nebula> MATCH (v) \
    WHERE id(v)=="player100" \
    WITH labels(v) AS tags_unf \
    UNWIND tags_unf AS tags_f \
    RETURN tags_f;
+-----+
| tags_f |
+-----+
| "player" |
+-----+
```

过滤聚合查询

`WITH` 可以在聚合查询中作为过滤器使用。

```
nebula> MATCH (v:player)-->(v2:player) \
    WITH DISTINCT v2 AS v2, v2.player.age AS Age \
    ORDER BY Age \
    WHERE Age<25 \
    RETURN v2.player.name AS Name, Age;
+-----+-----+
| Name | Age |
+-----+-----+
| "Luka Doncic" | 20 |
| "Ben Simmons" | 22 |
| "Kristaps Porzingis" | 23 |
+-----+-----+
```

collect() 之前处理输出

在 `collect()` 函数将输出结果转换为列表之前，可以使用 `WITH` 子句排序和限制输出结果。

```
nebula> MATCH (v:player) \
    WITH v.player.name AS Name \
    ORDER BY Name DESC \
    LIMIT 3 \
    RETURN collect(Name);
+-----+
| collect(Name) |
+-----+
| ["Yao Ming", "Vince Carter", "Tracy McGrady"] |
+-----+
```

结合 `RETURN` 语句使用

在 `WITH` 子句中设置别名，并通过 `RETURN` 子句输出结果。

```
nebula> WITH [1, 2, 3] AS `list` RETURN 3 IN `list` AS r;
+---+
| r |
+---+
| true |
+---+
nebula> WITH 4 AS one, 3 AS two RETURN one > two AS result;
+---+
| result |
+---+
| true |
+---+
```

最后更新: May 4, 2023

4.7.10 UNWIND

`UNWIND` 语句可以将列表拆分为单独的行，列表中的每个元素为一行。

`UNWIND` 可以作为单独语句或语句中的子句使用。

UNWIND 语句

语法

```
UNWIND <list> AS <alias> <RETURN clause>;
```

示例

- 拆分列表。

```
nebula> UNWIND [1,2,3] AS n RETURN n;
+---+
| n |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

UNWIND 子句

语法

- 原生 nGQL 语句中使用 `UNWIND` 子句。



在原生 nGQL 语句中使用 `UNWIND` 子句时，需要用在管道符 `|` 之后，并使用 `$-` 引用管道符之前的变量。如果 `UNWIND` 后使用语句或子句，需要使用管道符 `|` 并且使用 `$-` 引用管道符之前的变量。

```
<statement> | UNWIND $-.<var> AS <alias> <|> <clause>;
```

- openCypher 语句中使用 `UNWIND` 子句。

```
<statement> UNWIND <list> AS <alias> <RETURN clause>;
```

示例

- 在 `UNWIND` 子句中使用 `WITH DISTINCT` 可以将列表中的重复项忽略，返回去重后的结果。



原生 nGQL 语句不支持 `WITH DISTINCT`。

```
// 拆分列表'[1,1,2,2,3,3]'，删除重复行，排序行，将行转换为列表。
nebula> WITH [1,1,2,2,3,3] AS n \
    UNWIND n AS r \
    WITH DISTINCT r AS r \
    ORDER BY r \
    RETURN collect(r);
+-----+
| collect(r) |
+-----+
```

```
| [1, 2, 3] |
+-----+
```

- MATCH 语句中使用 UNWIND。

```
// 将匹配路径上的顶点输出到列表中，拆分列表，删除重复行，将行转换为列表。
nebula> MATCH p=(v:player{name:"Tim Duncan"})--(v2) \
    WITH nodes(p) AS n \
    UNWIND n AS r \
    WITH DISTINCT r AS r \
    RETURN collect(r);
+-----+
| collect(r) |
+-----+
| [{"player100": {"player": {"age": 42, "name": "Tim Duncan"}, "player": {"age": 36, "name": "Tony Parker"}}, {"team204": {"team": {"name": "Spurs"}, "player": {"age": 33, "name": "LaMarcus Aldridge"}}, {"player125": {"player": {"age": 41, "name": "Manu Ginobili"}, "player": {"age": 32, "name": "Marco Belinelli"}}, {"player144": {"player": {"age": 47, "name": "Shaquille O'Neal"}, "player": {"age": 31, "name": "Danny Green"}}, {"player113": {"player": {"age": 29, "name": "Dejounte Murray"}, "player": {"age": 32, "name": "Aron Baynes"}}, {"player109": {"player": {"age": 34, "name": "Tiago Splitter"}, "player": {"age": 36, "name": "Boris Diaw"}}}] |
+-----+
```

- GO 语句中使用 UNWIND。

```
// 在点列表中查询点关联的边。
nebula> YIELD ['player101', 'player100'] AS a | UNWIND $-.a AS b | GO FROM $.b OVER follow YIELD edge AS e;
+-----+
| e |
+-----+
| [:follow "player101"->"player100" @0 {degree: 95}] |
| [:follow "player101"->"player102" @0 {degree: 90}] |
| [:follow "player101"->"player125" @0 {degree: 95}] |
| [:follow "player100"->"player101" @0 {degree: 95}] |
| [:follow "player100"->"player125" @0 {degree: 95}] |
+-----+
```

- LOOKUP 语句中使用 UNWIND。

```
// 查询年龄大于 46 岁球员的所有属性，去掉重复属性，并将结果转换为行。
nebula> LOOKUP ON player \
    WHERE player.age > 46 \
    YIELD DISTINCT keys(vertex) as p | UNWIND $-.p as a | YIELD $-.a AS a;
+-----+
| a |
+-----+
| "age" |
| "name" |
+-----+
```

- FETCH 语句中使用 UNWIND。

```
// 查询 player101 点的所有 Tag，并将结果转换为行。
nebula> CREATE TAG hero(like string, height int);
    INSERT VERTEX hero(like, height) VALUES "player101":("deep", 182);
    FETCH PROP ON * "player101" \
    YIELD tags(vertex) as t | UNWIND $-.t as a | YIELD $-.a AS a;
+-----+
| a |
+-----+
| "hero" |
| "player" |
+-----+
```

- GET SUBGRAPH 语句中使用 UNWIND 。

```
// 查询从点 player100 开始、0~2 跳、serve 类型的出边和入边的子图，并将结果转换为行。
nebula> GET SUBGRAPH 2 STEPS FROM "player100" BOTH serve \
    YIELD edges as e | UNWIND $-.e as a | YIELD $-.a AS a;
+-----+
| a |
+-----+
| [:serve "player100"->"team204" @0 {}] |
| [:serve "player101"->"team204" @0 {}] |
| [:serve "player102"->"team204" @0 {}] |
| [:serve "player103"->"team204" @0 {}] |
| [:serve "player105"->"team204" @0 {}] |
| [:serve "player106"->"team204" @0 {}] |
| [:serve "player107"->"team204" @0 {}] |
| [:serve "player108"->"team204" @0 {}] |
| [:serve "player109"->"team204" @0 {}] |
| [:serve "player110"->"team204" @0 {}] |
| [:serve "player111"->"team204" @0 {}] |
| [:serve "player112"->"team204" @0 {}] |
| [:serve "player113"->"team204" @0 {}] |
| [:serve "player114"->"team204" @0 {}] |
| [:serve "player125"->"team204" @0 {}] |
| [:serve "player138"->"team204" @0 {}] |
| [:serve "player104"->"team204" @20132015 {}] |
| [:serve "player104"->"team204" @20182019 {}] |
+-----+
```

- FIND PATH 语句中使用 UNWIND 。

```
// 找出 player101 到 team204 延 serve 类型边的最短路径上的所有点，并将结果转换为行。
nebula> FIND SHORTEST PATH FROM "player101" TO "team204" OVER serve \
    YIELD path as p | YIELD nodes($-.p) AS nodes | UNWIND $-.nodes AS a | YIELD $-.a AS a;
+-----+
| a |
+-----+
| ("player101") |
| ("team204") |
+-----+
```

最后更新: May 4, 2023

4.8 图空间语句

4.8.1 CREATE SPACE

图空间是NebulaGraph中彼此隔离的图数据集合，与 MySQL 中的 database 概念类似。CREATE SPACE 语句可以创建一个新的图空间，或者克隆现有图空间的 Schema。

前提条件

只有 God 角色的用户可以执行 CREATE SPACE 语句。详情请参见[身份验证](#)。

语法

创建图空间

```
CREATE SPACE [IF NOT EXISTS] <graph_space_name> (
    [partition_num = <partition_number>],
    [replica_factor = <replica_number>],
    vid_type = {FIXED_STRING(<N>) | INT[64]}
)
[COMMENT = '<comment>'];
```

参数	说明
IF NOT EXISTS	检测待创建的图空间是否存在，只有不存在时，才会创建图空间。仅检测图空间的名称，不会检测具体属性。
<graph_space_name>	1、在NebulaGraph实例中唯一标识一个图空间。 2、图空间名称设置后无法被修改。 3、不能以数字开头；支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等，但是不包括除下划线外的特殊字符；使用特殊字符、保留关键字或数字开头时，需要用反引号 (`) 包围且不能使用英文句号 (.)。详情参见 关键字和保留字 。 注意：如果以中文为图空间命名，报 SyntaxError 错误时，需使用反引号 (`) 包围中文字符。
partition_num	指定图空间的分片数量。建议设置为集群中硬盘数量的 20 倍（HDD 硬盘建议为 2 倍）。例如集群中有 3 个硬盘，建议设置 60 个分片。默认值为 100。
replica_factor	指定每个分片的副本数量。建议在生产环境中设置为 3，在测试环境中设置为 1。由于需要基于多数表决，副本数量必须是奇数。默认值为 1。
vid_type	必选参数。指定点 ID 的数据类型。可选值为 FIXED_STRING(<N>) 和 INT64。INT 等同于 INT64。 FIXED_STRING(<N>) 表示数据类型为定长字符串，长度为 N 字节，超出长度会报错。例如，UTF-8 中，一个中文字符的长度为三个字节，如果设置 N 为 12，那么 vid_type 为最多 4 个中文字符。 INT64 表示数据类型为整数。
COMMENT	图空间的描述。最大为 256 字节。默认无描述。

Caution

- 如果将副本数设置为 1，用户将无法使用 **BALANCE** 命令为 NebulaGraph 的存储服务平衡负载或扩容。
- VID 类型变更与长度限制：
- 在 NebulaGraph 1.x 中，VID 的类型只能为 **INT64**，不支持字符型；在 NebulaGraph 2.x 中，VID 的类型支持 **INT64** 和 **FIXED_STRING(<N>)**。请在创建图空间时指定 VID 类型，使用 **INSERT** 语句时也需要保持一致，否则会报错 VID 类型不匹配 **Wrong vertex id type: 1001**。
- VID 最大长度必须为 N，不可任意长度；超过该长度也会报错 **The VID must be a 64-bit integer or a string fitting space vertex id length limit.**。
- 如果出现 **Host not enough!** 的报错，直接原因是线上 **Storage Host** 的数量少于即将创建的图空间的 **replica_factor** 的数量。此时，可用 **SHOW HOSTS** 指令查看判断，出现的情况有：
 - 在集群是单 **Storage Host** 的情况下试图创建多副本（**replica_factor**）图空间，这时候只能选择创建单副本数的图空间，或者扩容 **Storage Host** 之后再创建图空间。
 - 新创建的集群里 **Storage Host** 已经被服务发现，但是尚未执行 **ADD HOSTS** 将其激活，这时候需要通过 **Console** 连接，执行 **SHOW HOSTS** 获取被发现了的 **Storage Host**，然后执行相应的 **ADD HOSTS** 激活，待有足够的 **Online Storage Host** 之后再尝试创建图空间。
 - 有部分 **Storage Host** 处在非 **Online** 状态，需要进行进一步排查。

↑ 版本兼容性

2.5.0 之前的 2.x 版本中，**vid_type** 不是必选参数，默認為 **FIXED_STRING(8)**。

Note

`graph_space_name, partition_num, replica_factor, vid_type, comment` 設置后就无法改变。除非 **DROP SPACE**，并重新 **CREATE SPACE**。

克隆图空间

```
CREATE SPACE [IF NOT EXISTS] <new_graph_space_name> AS <old_graph_space_name>;
```

参数	说明
<code>IF NOT EXISTS</code>	检测待创建的图空间是否存在，只有不存在时，才会克隆图空间。仅检测图空间的名称，不会检测具体属性。
<code><new_graph_space_name></code>	目标图空间名称。该图空间必须未创建。图空间名称以英文字母开头，支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等，但是不包括除下划线外的特殊字符。使用保留关键字时，需要用反引号 (`) 包围，详情参见 关键字和保留字 。创建时会克隆 <code><old_graph_space_name></code> 图空间的 Schema，包括图空间本身参数（分片数量、副本数量等）、Tag、Edge type 和 原生索引。
<code><old_graph_space_name></code>	原始图空间名称。该图空间必须已存在。

示例

```
# 仅指定 VID 类型，其他选项使用默认值。
nebula> CREATE SPACE IF NOT EXISTS my_space_1 (vid_type=FIXED_STRING(30));

# 指定分片数量、副本数量和 VID 类型。
nebula> CREATE SPACE IF NOT EXISTS my_space_2 (partition_num=15, replica_factor=1, vid_type=FIXED_STRING(30));

# 指定分片数量、副本数量和 VID 类型，并添加描述。
nebula> CREATE SPACE IF NOT EXISTS my_space_3 (partition_num=15, replica_factor=1, vid_type=FIXED_STRING(30)) comment="测试图空间";

# 克隆图空间。
nebula> CREATE SPACE IF NOT EXISTS my_space_4 as my_space_3;
nebula> SHOW CREATE SPACE my_space_4;
+-----+-----+
| Space       | Create Space
+-----+-----+
| "my_space_4" | "CREATE SPACE `my_space_4` (partition_num = 15, replica_factor = 1, charset = utf8, collate = utf8_bin, vid_type = FIXED_STRING(30)) comment = '测试图空间'" |
+-----+-----+
```

Caution

立刻尝试使用刚创建的图空间可能会失败。因为创建是异步实现的。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。如果需要修改心跳间隔，请为[所有配置文件](#)修改参数 `heartbeat_interval_secs`。但过短的心跳周期（<5 秒）可能会导致分布式系统中的机器误判对端失联。

检查分片分布情况

在大型集群中，由于启动时间不同，分片的分布可能不均衡。用户可以执行如下命令检查分片的分布情况：

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution | Version |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 8 | "basketballplayer:3, test:5" | "basketballplayer:10, test:10" | "3.1.0" |
| "storaged1" | 9779 | "ONLINE" | 9 | "basketballplayer:4, test:5" | "basketballplayer:10, test:10" | "3.1.0" |
| "storaged2" | 9779 | "ONLINE" | 3 | "basketballplayer:3" | "basketballplayer:10, test:10" | "3.1.0" |
+-----+-----+-----+-----+-----+
```

如果需要均衡负载，请执行如下命令：

```
nebula> BALANCE LEADER;
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution | Version |
+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 7 | "basketballplayer:3, test:4" | "basketballplayer:10, test:10" | "3.4.1" |
| "storaged1" | 9779 | "ONLINE" | 7 | "basketballplayer:4, test:3" | "basketballplayer:10, test:10" | "3.4.1" |
| "storaged2" | 9779 | "ONLINE" | 6 | "basketballplayer:3, test:3" | "basketballplayer:10, test:10" | "3.4.1" |
+-----+-----+-----+-----+-----+
```

最后更新: May 4, 2023

4.8.2 USE

USE 语句可以指定一个图空间，或切换到另一个图空间，将其作为后续查询的工作空间。

前提条件

执行 USE 语句指定图空间时，需要当前登录的用户拥有指定图空间的权限，否则会报错。

语法

```
USE <graph_space_name>;
```

示例

```
# 创建示例空间。  
nebula> CREATE SPACE IF NOT EXISTS space1 (vid_type=FIXED_STRING(30));  
nebula> CREATE SPACE IF NOT EXISTS space2 (vid_type=FIXED_STRING(30));  
  
# 指定图空间 space1 作为工作空间。  
nebula> USE space1;  
  
# 切换到图空间 space2。检索 space2 时，无法从 space1 读取任何数据，检索的点和边与 space1 无关。  
nebula> USE space2;
```



不能在一条语句中同时操作两个图空间。

与 Fabric Cypher 不同，NebulaGraph 的图空间彼此之间是完全隔离的，将一个图空间作为工作空间后，用户无法访问其他空间。使用新图空间的唯一方法是通过 USE 语句切换。而在 Fabric Cypher 中可以在一条语句中（USE + CALL 语法）使用两个图空间。

最后更新: May 4, 2023

4.8.3 SHOW SPACES

SHOW SPACES 语句可以列出NebulaGraph示例中的所有图空间。

语法

```
SHOW SPACES;
```

示例

```
nebula> SHOW SPACES;
+-----+
| Name      |
+-----+
| "cba"    |
| "basketballplayer" |
+-----+
```

创建图空间请参见 [CREATE SPACE](#)。

最后更新: May 4, 2023

4.8.4 DESCRIBE SPACE

DESCRIBE SPACE 语句可以显示指定图空间的信息。

语法

你可以用 DESC 作为 DESCRIBE 的缩写。

```
DESC[DESCRIBE] SPACE <graph_space_name>;
```

示例

```
nebula> DESCRIBE SPACE basketballplayer;
+-----+-----+-----+-----+-----+
| ID | Name          | Partition Number | Replica Factor | Charset | Collate   | Vid Type      | Comment |
+-----+-----+-----+-----+-----+
| 1  | "basketballplayer" | 10            | 1              | "utf8"    | "utf8_bin" | "FIXED_STRING(32)" |
+-----+-----+-----+-----+-----+
```

最后更新: May 4, 2023

4.8.5 Macro Rendering Error

UndefinedError: 'comm' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 21, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'comm' is undefined
```

最后更新: May 4, 2023

4.8.6 DROP SPACE

DROP SPACE 语句用于删除指定图空间以及其中的所有信息。



DROP SPACE 是否删除图空间对应的硬盘数据由 Storage 配置参数 `auto_remove_invalid_space` 决定。`auto_remove_invalid_space` 的默认值为 `true`，表示会删除数据。如需在删除逻辑图空间时保留硬盘数据，将 `auto_remove_invalid_space` 的值修改为 `false`。详情参见 [Storage 服务配置](#)。



执行 DROP SPACE 后，即使快照中存在该图空间的数据，该图空间的数据也无法恢复。但是如果 `auto_remove_invalid_space` 的值为 `false` 时，可联系商务人员恢复数据。

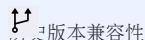
前提条件

只有 God 角色的用户可以执行 DROP SPACE 语句。详情请参见[身份验证](#)。

语法

```
DROP SPACE [IF EXISTS] <graph_space_name>;
```

`IF EXISTS` 关键字可以检测待删除的图空间是否存在，只有存在时，才会删除图空间。



在NebulaGraph 3.1.0 版本前，DROP SPACE 语句不会删除硬盘上对应图空间的目录和文件。



请谨慎执行删除图空间操作。

FAQ

问：执行 DROP SPACE 语句删除图空间后，为什么磁盘的大小没变化？

答：如果使用 3.1.0 之前版本的NebulaGraph，DROP SPACE 语句仅删除指定的逻辑图空间，不会删除硬盘上对应图空间的目录和文件。如需删除硬盘上的数据，需手动删除相应文件的路径，文件路径为 `<nebula_graph_install_path>/data/storage/nebula/<space_id>`。其中 `<space_id>` 可以通过 `DESCRIBE SPACE {space_name}` 查看。

最后更新: May 4, 2023

4.9 Tag 语句

4.9.1 CREATE TAG

`CREATE TAG` 语句可以通过指定名称创建一个 Tag。

OpenCypher 兼容性

nGQL 中的 Tag 和 openCypher 中的 Label 相似，但又有所不同，例如它们的创建方式。

- openCypher 中的 Label 需要在 `CREATE` 语句中与点一起创建。
- nGQL 中的 Tag 需要使用 `CREATE TAG` 语句独立创建。Tag 更像是 MySQL 中的表。

前提条件

执行 `CREATE TAG` 语句需要当前登录的用户拥有指定图空间的 [创建 Tag 权限](#)，否则会报错。

语法

创建 Tag 前，需要先用 `USE` 语句指定工作空间。

```
CREATE TAG [IF NOT EXISTS] <tag_name>
(
    <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
    [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
)
[TTL_DURATION = <ttx_duration>]
[TTL_COL = <prop_name>]
[COMMENT = '<comment>'];
```

参数	说明
<code>IF NOT EXISTS</code>	检测待创建的 Tag 是否存在，只有不存在时，才会创建 Tag。仅检测 Tag 的名称，不会检测具体属性。
<code><tag_name></code>	1、每个图空间内的 Tag 必须是唯一的。 2、Tag 名称设置后无法修改。 3、不能以数字开头；支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等，但是不包括除下划线外的特殊字符；使用特殊字符、保留关键字或数字开头时，需要用反引号 (`) 包围且不能使用英文句号 (.)。详情参见 关键字和保留字 。 注意：如果以中文命名 Tag，报 <code>SyntaxError</code> 错误时，需使用反引号 (`) 包围中文字符。
<code><prop_name></code>	属性名称。每个 Tag 中的属性名称必须唯一。属性的命名规则与 Tag 相同。
<code><data_type></code>	属性的数据类型，目前支持 数值 、 布尔 、 字符串 以及 日期与时间 。
<code>NULL \ NOT NULL</code>	指定属性值是否支持为 <code>NULL</code> 。默认值为 <code>NULL</code> 。当指定属性值为 <code>NOT NULL</code> 时，必需指定属性的默认值，也就是 <code>DEFAULT</code> 的值。
<code>DEFAULT</code>	指定属性的默认值。默认值可以是一个文字值或 NebulaGraph 支持的表达式。如果插入点时没有指定某个属性的值，则使用默认值。
<code>COMMENT</code>	对单个属性或 Tag 的描述。最大为 256 字节。默认无描述。
<code>TTL_DURATION</code>	指定时间戳差值，单位：秒。时间戳差值必须为 64 位非负整数。属性值和时间戳差值之和如果小于当前时间戳，属性就会过期。默认值为 0，表示属性永不过期。
<code>TTL_COL</code>	指定要设置存活时间的属性。属性的数据类型必须是 <code>int</code> 或者 <code>timestamp</code> 。一个 Tag 只能指定一个字段为 <code>TTL_COL</code> 。更多 TTL 的信息请参见 TTL 。

示例

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

# 创建没有属性的 Tag。
nebula> CREATE TAG IF NOT EXISTS no_property();

# 创建包含默认值的 Tag。
nebula> CREATE TAG IF NOT EXISTS player_with_default(name string, age int DEFAULT 20);

# 对字段 create_time 设置 TTL 为 100 秒。
nebula> CREATE TAG IF NOT EXISTS woman(name string, age int, \
    married bool, salary double, create_time timestamp) \
    TTL_DURATION = 100, TTL_COL = "create_time";
```

创建 Tag 说明

尝试使用新创建的 Tag 可能会失败，因为创建是异步实现的。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

如果需要修改心跳间隔，请为[所有配置文件](#)修改参数 heartbeat_interval_secs。

最后更新: May 4, 2023

4.9.2 DROP TAG

`DROP TAG` 语句可以删除当前工作空间内所有点上的指定 `Tag`。

点可以有一个或多个 `Tag`。

- 如果某个点只有一个 `Tag`, 删除这个 `Tag` 后, 用户就无法访问这个点, 下次 `Compaction` 操作时会删除该点, 但与该点相邻的边仍然存在——这会造成悬挂边。
- 如果某个点有多个 `Tag`, 删除其中一个 `Tag`, 仍然可以访问这个点, 但是无法访问已删除 `Tag` 所定义的所有属性。

删除 `Tag` 操作仅删除 `Schema` 数据, 硬盘上的文件或目录不会立刻删除, 而是在下一次 `Compaction` 操作时删除。

↑ Compatibility

NebulaGraph 3.4.1 中默认不支持插入无 `Tag` 的点。如需使用无 `Tag` 的点, 在集群内所有 `Graph` 服务的配置文件 (`nebula-graphd.conf`) 中新增 `--graph_use_vertex_key=true`; 在所有 `Storage` 服务的配置文件 (`nebula-storaged.conf`) 中新增 `--use_vertex_key=true`。

前提条件

- 登录的用户必须拥有对应权限才能执行 `DROP TAG` 语句。详情请参见[内置角色权限](#)。
- 确保 `Tag` 不包含任何索引, 否则 `DROP TAG` 时会报冲突错误 [ERROR (-1005)]: `Conflict!`。删除索引请参见 [drop index](#)。

语法

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

- `IF EXISTS`: 检测待删除的 `Tag` 是否存在, 只有存在时, 才会删除 `Tag`。
- `tag_name`: 指定要删除的 `Tag` 名称。一次只能删除一个 `Tag`。

示例

```
nebula> CREATE TAG IF NOT EXISTS test(p1 string, p2 int);
nebula> DROP TAG test;
```

最后更新: May 4, 2023

4.9.3 ALTER TAG

ALTER TAG 语句可以修改 Tag 的结构。例如增删属性、修改数据类型，也可以为属性设置、修改 TTL (Time-To-Live)。

注意事项

- 登录的用户必须拥有对应权限才能执行 ALTER TAG 语句。详情请参见[内置角色权限](#)。
- 确保要修改的属性不包含索引，否则 ALTER TAG 时会报冲突错误 [ERROR (-1005)]: Conflict!。删除索引请参见 [drop index](#)。
- 确保新增的属性名不与已存在或被删除的属性名同名，否则新增属性会失败。

语法

```
ALTER TAG <tag_name>
  <alter_definition> [[, <alter_definition>] ...]
  [<ttl_definition> [, <ttl_definition>] ... ]
  [COMMENT = '<comment>'];

<alter_definition>:
| ADD   (<prop_name> <data_type> [<NULL | NOT NULL>] [<DEFAULT ><default_value>] [<COMMENT '<comment>'>])
| DROP  (<prop_name>)
| CHANGE (<prop_name> <data_type> [<NULL | NOT NULL>] [<DEFAULT ><default_value>] [<COMMENT '<comment>'>])

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

- <tag_name>**：指定要修改的 Tag 名称。一次只能修改一个 Tag。请确保要修改的 Tag 在当前工作空间中存在，否则会报错。
- 可以在一个 ALTER TAG 语句中使用多个 ADD、DROP 和 CHANGE 子句，子句之间用英文逗号 (,) 分隔。
- 当使用 ADD 或 CHANGE 指定属性值为 NOT NULL 时，必需为该属性指定默认值，即定义 DEFAULT 的值。
- 当使用 CHANGE 修改属性的数据类型时：
 - 仅允许修改 FIXED_STRING 和 INT 类型的长度为更大的长度，不允许减少长度。
 - 仅允许修改 FIXED_STRING 类型为 STRING 类型、修改 FLOAT 类型为 DOUBLE 类型。

示例

```
nebula> CREATE TAG IF NOT EXISTS t1 (p1 string, p2 int);
nebula> ALTER TAG t1 ADD (p3 int32, p4 fixed_string(10));
nebula> ALTER TAG t1 TTL_DURATION = 2, TTL_COL = "p2";
nebula> ALTER TAG t1 COMMENT = 'test1';
nebula> ALTER TAG t1 ADD (p5 double NOT NULL DEFAULT 0.4 COMMENT 'p5') COMMENT='test2';
// 将 TAG t1 的 p3 属性类型从 INT32 改为 INT64, p4 属性类型从 FIXED_STRING(10) 改为 STRING。
nebula> ALTER TAG t1 CHANGE (p3 int64, p4 string);
```

修改 Tag 说明

尝试使用刚修改的 Tag 可能会失败，因为修改是异步实现的。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

如果需要修改心跳间隔，请为[所有配置文件](#)修改参数 heartbeat_interval_secs。

最后更新: May 4, 2023

4.9.4 SHOW TAGS

SHOW TAGS 语句显示当前图空间内的所有 Tag 名称。

执行 SHOW TAGS 语句不需要任何权限，但是返回结果由登录的用户权限决定。

语法

```
SHOW TAGS;
```

示例

```
nebula> SHOW TAGS;
+-----+
| Name      |
+-----+
| "player"  |
| "team"    |
+-----+
```

最后更新: May 4, 2023

4.9.5 DESCRIBE TAG

DESCRIBE TAG 显示指定 Tag 的详细信息，例如字段名称、数据类型等。

前提条件

登录的用户必须拥有对应权限才能执行 DESCRIBE TAG 语句。详情请参见[内置角色权限](#)。

语法

```
DESC[RIBE] TAG <tag_name>;
```

DESCRIBE 可以缩写为 DESC。

示例

```
nebula> DESCRIBE TAG player;
+-----+-----+-----+-----+
| Field | Type   | Null | Default | Comment |
+-----+-----+-----+-----+
| "name" | "string" | "YES" |          |
| "age"  | "int64"  | "YES" |          |
+-----+-----+-----+-----+
```

最后更新: May 4, 2023

4.9.6 DELETE TAG

`DELETE TAG` 语句可以删除指定点上的指定 Tag。

前提条件

登录的用户必须拥有对应权限才能执行 `DELETE TAG` 语句。详情请参见[内置角色权限](#)。

语法

```
DELETE TAG <tag_name_list> FROM <VID>;
```

- `tag_name_list`：指定 Tag 名称。多个 Tag 用英文逗号 (,) 分隔，也可以用 * 表示所有 Tag。
- `VID`：指定要删除 Tag 的点 ID。

示例

```
nebula> CREATE TAG IF NOT EXISTS test1(p1 string, p2 int);
nebula> CREATE TAG IF NOT EXISTS test2(p3 string, p4 int);
nebula> INSERT VERTEX test1(p1, p2),test2(p3, p4) VALUES "test":("123", 1, "456", 2);
nebula> FETCH PROP ON * "test" YIELD vertex AS v;
+-----+
| v |
+-----+
| ("test" :test1{p1: "123", p2: 1} :test2{p3: "456", p4: 2}) |
+-----+
```



```
nebula> DELETE TAG test1 FROM "test";
nebula> FETCH PROP ON * "test" YIELD vertex AS v;
+-----+
| v |
+-----+
| ("test" :test2{p3: "456", p4: 2}) |
+-----+
```



```
nebula> DELETE TAG * FROM "test";
nebula> FETCH PROP ON * "test" YIELD vertex AS v;
+---+
| v |
+---+
+---+
```

Compatibility

- 在 openCypher 中，可以使用 `REMOVE v:LABEL` 语句来移除该点 `v` 的 `LABEL`。
- 相同语意，但不同语法。在 nGQL 中使用 `DELETE TAG`。

最后更新: May 4, 2023

4.9.7 增加和删除标签

在 openCypher 中，有增加标签（`SET Label`）和移除标签（`REMOVE Label`）的功能，可以用于加速查询或者标记过程。

在 NebulaGraph 中，可以通过 Tag 变相实现相同操作，创建 Tag 并将 Tag 插入到已有的点上，就可以根据 Tag 名称快速查找点，也可以通过 `DELETE TAG` 删除某些点上不再需要的 Tag。

示例

例如在 `basketballplayer` 数据集中，部分篮球运动员同时也是球队股东，可以为股东 Tag `shareholder` 创建索引，方便快速查找。如果不再是股东，可以通过 `DELETE TAG` 语句删除相应运动员的股东 Tag。

```
//创建股东 Tag 和索引
nebula> CREATE TAG IF NOT EXISTS shareholder();
nebula> CREATE TAG INDEX IF NOT EXISTS shareholder_tag on shareholder();

//为点添加 Tag
nebula> INSERT VERTEX shareholder() VALUES "player100":();
nebula> INSERT VERTEX shareholder() VALUES "player101":();

//快速查询所有股东
nebula> MATCH (v:shareholder) RETURN v;
+-----+
| v |
+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"} :shareholder{}) |
| ("player101" :player{age: 36, name: "Tony Parker"} :shareholder{}) |
+-----+
nebula> LOOKUP ON shareholder YIELD id(vertex);
+-----+
| id(VERTEX) |
+-----+
| "player100" |
| "player101" |
+-----+

//如果 player100 不再是股东
nebula> DELETE TAG shareholder FROM "player100";
nebula> LOOKUP ON shareholder YIELD id(vertex);
+-----+
| id(VERTEX) |
+-----+
| "player101" |
+-----+
```



如果插入测试数据后才创建索引，请用 `REBUILD TAG INDEX <index_name_list>`; 语句重建索引。

最后更新: May 4, 2023

4.10 Edge type 语句

4.10.1 CREATE EDGE

`CREATE EDGE` 语句可以通过指定名称创建一个 Edge type。

OpenCypher 兼容性

nGQL 中的 Edge type 和 openCypher 中的关系类型相似，但又有所不同，例如它们的创建方式。

- openCypher 中的关系类型需要在 `CREATE` 语句中与点一起创建。
- nGQL 中的 Edge type 需要使用 `CREATE EDGE` 语句独立创建。Edge type 更像是 MySQL 中的表。

前提条件

执行 `CREATE EDGE` 语句需要当前登录的用户拥有指定图空间的 **创建 Edge type 权限**，否则会报错。

语法

创建 Edge type 前，需要先用 `USE` 语句指定工作空间。

```
CREATE EDGE [IF NOT EXISTS] <edge_type_name>
(
  <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
  [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
)
[TTL_DURATION = <ttx_duration>]
[TTL_COL = <prop_name>]
[COMMENT = '<comment>'];
```

参数	说明
<code>IF NOT EXISTS</code>	检测待创建的 Edge type 是否存在，只有不存在时，才会创建 Edge type。仅检测 Edge type 的名称，不会检测具体属性。
<code><edge_type_name></code>	1、每个图空间内的 Edge type 必须是唯一的。 2、Edge type 名称设置后无法修改。 3、不能以数字开头；支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等，但是不包括除下划线外的特殊字符；使用特殊字符、保留关键字或数字开头时，需要用反引号 (`) 包围且不能使用英文句号 (.)。详情参见 关键字和保留字 。 注意：如果以中文命名 Edge type，报 SyntaxError 错误时，需使用反引号 (`) 包围中文字符。
<code><prop_name></code>	属性名称。每个 Edge type 中的属性名称必须唯一。属性的命名规则与 Edge type 相同。
<code><data_type></code>	属性的数据类型，目前支持 数值 、 布尔 、 字符串 以及 日期与时间 。
<code>NULL NOT NULL</code>	指定属性值是否支持为 <code>NULL</code> 。默认值为 <code>NULL</code> 。当指定属性值为 <code>NOT NULL</code> 时，必需指定属性的默认值，也就是 <code>DEFAULT</code> 的值。
<code>DEFAULT</code>	指定属性的默认值。默认值可以是一个文字值或 NebulaGraph 支持的表达式。如果插入边时没有指定某个属性的值，则使用默认值。
<code>COMMENT</code>	对单个属性或 Edge type 的描述。最大为 256 字节。默认无描述。
<code>TTL_DURATION</code>	指定时间戳差值，单位：秒。时间戳差值必须为 64 位非负整数。属性值和时间戳差值之和如果小于当前时间戳，属性就会过期。默认值为 0，表示属性永不过期。
<code>TTL_COL</code>	指定要设置存活时间的属性。属性的数据类型必须是 <code>int</code> 或者 <code>timestamp</code> 。一个 Edge type 只能指定一个字段为 <code>TTL_COL</code> 。更多 TTL 的信息请参见 TTL 。

示例

```
nebula> CREATE EDGE IF NOT EXISTS follow(degree int);
```

```
# 创建没有属性的 Edge type。  
nebula> CREATE EDGE IF NOT EXISTS no_property();  
  
# 创建包含默认值的 Edge type。  
nebula> CREATE EDGE IF NOT EXISTS follow_with_default(degree int DEFAULT 20);  
  
# 对字段 p2 设置 TTL 为 100 秒。  
nebula> CREATE EDGE IF NOT EXISTS e1(p1 string, p2 int, p3 timestamp) \  
TTL_DURATION = 100, TTL_COL = "p2";
```

最后更新: May 4, 2023

4.10.2 DROP EDGE

DROP EDGE 语句可以删除当前工作空间内的指定 Edge type。

一个边只能有一个 Edge type，删除这个 Edge type 后，用户就无法访问这个边，下次 Compaction 操作时会删除该边。

删除 Edge type 操作仅删除 Schema 数据，硬盘上的文件或目录不会立刻删除，而是在下一次 Compaction 操作时删除。

前提条件

- 登录的用户必须拥有对应权限才能执行 DROP EDGE 语句。详情请参见[内置角色权限](#)。
- 确保 Edge type 不包含任何索引，否则 DROP EDGE 时会报冲突错误 [ERROR (-1005)]: Conflict!。删除索引请参见 [drop index](#)。

语法

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

- IF EXISTS：检测待删除的 Edge type 是否存在，只有存在时，才会删除 Edge type。
- edge_type_name：指定要删除的 Edge type 名称。一次只能删除一个 Edge type。

示例

```
nebula> CREATE EDGE IF NOT EXISTS e1(p1 string, p2 int);
nebula> DROP EDGE e1;
```

最后更新: May 4, 2023

4.10.3 ALTER EDGE

ALTER EDGE 语句可以修改 Edge type 的结构。例如增删属性、修改数据类型，也可以为属性设置、修改 TTL (Time-To-Live)。

注意事项

- 登录的用户必须拥有对应权限才能执行 ALTER EDGE 语句。详情请参见[内置角色权限](#)。
- 确保要修改的属性不包含索引，否则 ALTER EDGE 时会报冲突错误 [ERROR (-1005)]: Conflict!。删除索引请参见 [drop index](#)。
- 确保新增的属性名不与已存在或被删除的属性名同名，否则新增属性会失败。
- 允许增加 FIXED_STRING 和 INT 类型的长度。
- 允许 FIXED_STRING 类型转换为 STRING 类型、FLOAT 类型转换为 DOUBLE 类型。

语法

```
ALTER EDGE <edge_type_name>
  <alter_definition> [, alter_definition] ...
  [ttl_definition [, ttl_definition] ...]
  [COMMENT = '<comment>'];

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

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

- edge_type_name**：指定要修改的 Edge type 名称。一次只能修改一个 Edge type。请确保要修改的 Edge type 在当前工作空间中存在，否则会报错。
- 可以在一个 ALTER EDGE 语句中使用多个 ADD、DROP 和 CHANGE 子句，子句之间用英文逗号（,）分隔。
- 当使用 ADD 或 CHANGE 指定属性值为 NOT NULL 时，必需为该属性指定默认值，即定义 DEFAULT 的值。

示例

```
nebula> CREATE EDGE IF NOT EXISTS e1(p1 string, p2 int);
nebula> ALTER EDGE e1 ADD (p3 int, p4 string);
nebula> ALTER EDGE e1 TTL_DURATION = 2, TTL_COL = "p2";
nebula> ALTER EDGE e1 COMMENT = 'edge1';
```

修改 Edge type 说明

尝试使用刚修改的 Edge type 可能会失败，因为修改是异步实现的。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

如果需要修改心跳间隔，请为[所有配置文件](#)修改参数 heartbeat_interval_secs。

最后更新: May 4, 2023

4.10.4 SHOW EDGES

SHOW EDGES 语句显示当前图空间内的所有 Edge type 名称。

执行 SHOW EDGES 语句不需要任何权限，但是返回结果由登录的用户权限 决定。

语法

```
SHOW EDGES;
```

示例

```
nebula> SHOW EDGES;
+-----+
| Name   |
+-----+
| "follow" |
| "serve"  |
+-----+
```

最后更新: May 4, 2023

4.10.5 DESCRIBE EDGE

DESCRIBE EDGE 显示指定 Edge type 的详细信息，例如字段名称、数据类型等。

前提条件

登录的用户必须拥有对应权限才能执行 DESCRIBE EDGE 语句。详情请参见[内置角色权限](#)。

语法

```
DESC[RIBE] EDGE <edge_type_name>
```

DESCRIBE 可以缩写为 DESC。

示例

```
nebula> DESCRIBE EDGE follow;
+-----+-----+-----+-----+
| Field | Type  | Null | Default | Comment |
+-----+-----+-----+-----+
| "degree" | "int64" | "YES" |          |
+-----+-----+-----+-----+
```

最后更新: May 4, 2023

4.11 点语句

4.11.1 INSERT VERTEX

`INSERT VERTEX` 语句可以在NebulaGraph实例的指定图空间中插入一个或多个点。

前提条件

执行 `INSERT VERTEX` 语句需要当前登录的用户拥有指定图空间的[插入点权限](#)，否则会报错。

语法

```
INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...]
VALUES VID: ([prop_value_list])

tag_props:
  tag_name ([prop_name_list])

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

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

- `IF NOT EXISTS`：检测待插入的 VID 是否存在，只有不存在时，才会插入，如果已经存在，不会进行修改。



- `IF NOT EXISTS` 仅检测 VID + Tag 的值是否相同，不会检测属性值。
- `IF NOT EXISTS` 会先读取一次数据是否存在，因此对性能会有明显影响。

- `tag_name`：点关联的 Tag（点类型）。Tag 的创建，详情请参见 [CREATE TAG](#)。



NebulaGraph 3.4.1 中默认不支持插入无 Tag 的点。如需使用无 Tag 的点，在集群内所有 Graph 服务的配置文件（`nebula-graphd.conf`）中新增 `--graph_use_vertex_key=true`；在所有 Storage 服务的配置文件（`nebula-storaged.conf`）中新增 `--use_vertex_key=true`。插入无 Tag 点的命令示例如下：

- `property_name`：需要设置的属性名称。
- `vid`：点 ID。在NebulaGraph 3.4.1 中支持字符串和整数，需要在创建图空间时设置，详情请参见 [CREATE SPACE](#)。
- `property_value`：根据 `prop_name_list` 填写属性值。如果没有填写属性值，而 Tag 中对应的属性设置为 `NOT NULL`，会返回错误。详情请参见 [CREATE TAG](#)。



`INSERT VERTEX` 与 `openCypher` 中 `CREATE` 的语意不同：

- `INSERT VERTEX` 语意更接近于 NoSQL(key-value) 方式的 `INSERT` 语意，或者 SQL 中的 `UPSERT (UPDATE OR INSERT)`。
- 相同 VID 和 TAG 的情况下，如果没有使用 `IF NOT EXISTS`，新写入的数据会覆盖旧数据，不存在时会新写入。
- 相同 VID 但不同 TAG 的情况下，不同 TAG 对应的记录不会相互覆盖，不存在会新写入。

参考以下示例。

示例

```
# 插入不包含属性的点。
nebula> CREATE TAG IF NOT EXISTS t1();
nebula> INSERT VERTEX t1() VALUES "10":();

nebula> CREATE TAG IF NOT EXISTS t2 (name string, age int);
nebula> INSERT VERTEX t2 (name, age) VALUES "11":("n1", 12);

# 创建失败，因为"a13"不是 int 类型。
nebula> INSERT VERTEX t2 (name, age) VALUES "12":("n1", "a13");

# 一次插入 2 个点。
nebula> INSERT VERTEX t2 (name, age) VALUES "13":("n3", 12), "14":("n4", 8);

nebula> CREATE TAG IF NOT EXISTS t3(p1 int);
nebula> CREATE TAG IF NOT EXISTS t4(p2 string);

# 一次插入两个 Tag 的属性到同一个点。
nebula> INSERT VERTEX t3 (p1), t4(p2) VALUES "21": (321, "hello");
```

一个点可以多次插入属性值，以最后一次为准。

```
# 多次插入属性值。
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);
nebula> FETCH PROP ON t2 "11" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 15, name: "n4"} |
+-----+

nebula> CREATE TAG IF NOT EXISTS t5(p1 fixed_string(5) NOT NULL, p2 int, p3 int DEFAULT NULL);
nebula> INSERT VERTEX t5(p1, p2, p3) VALUES "001":("Abe", 2, 3);

# 插入失败，因为属性 p1 不能为 NULL。
nebula> INSERT VERTEX t5(p1, p2, p3) VALUES "002":(NULL, 4, 5);
[ERROR (-1009)]: SemanticError: No schema found for `t5'

# 属性 p3 为默认值 NULL。
nebula> INSERT VERTEX t5(p1, p2) VALUES "003":("cd", 5);
nebula> FETCH PROP ON t5 "003" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {p1: "cd", p2: 5, p3: __NULL__} |
+-----+

# 属性 p1 最大长度为 5，因此会被截断。
nebula> INSERT VERTEX t5(p1, p2) VALUES "004":("shalalalala", 4);
nebula> FETCH PROP ON t5 "004" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {p1: "shala", p2: 4, p3: __NULL__} |
+-----+
```

使用 IF NOT EXISTS 插入已存在的点时，不会进行修改。

```
# 插入点 1。
nebula> INSERT VERTEX t2 (name, age) VALUES "1":("n2", 13);

# 使用 IF NOT EXISTS 修改点 1，因为点 1 已存在，不会进行修改。
nebula> INSERT VERTEX IF NOT EXISTS t2 (name, age) VALUES "1":("n3", 14);
nebula> FETCH PROP ON t2 "1" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 13, name: "n2"} |
+-----+
```

最后更新: May 4, 2023

4.11.2 DELETE VERTEX

`DELETE VERTEX` 语句可以删除点，但是默认不删除该点关联的出边和入边。

Compatibility

NebulaGraph 2.x 默认删除点及关联该点的出边和入，NebulaGraph 3.4.1 默认只删除点，不删除该点关联的出边和入边，此时将默认存在悬挂边。

`DELETE VERTEX` 语句一次可以删除一个或多个点。用户可以结合管道符一起使用，详情请参见[管道符](#)。

Note

- `DELETE VERTEX` 是直接删除点，不删除关联的边。
- `DELETE TAG` 是删除指定点上的指定 Tag。

语法

```
DELETE VERTEX <vid> [ , <vid> ... ] [WITH EDGE];
```

- `WITH EDGE`：删除该点关联的出边和入边。

示例

```
# 删除 VID 为 `team1` 的点，不删除该点关联的出边和入边。
nebula> DELETE VERTEX "team1";

# 删除 VID 为 `team1` 的点，并删除该点关联的出边和入边。
nebula> DELETE VERTEX "team1" WITH EDGE;

# 结合管道符，删除符合条件的点。
nebula> GO FROM "player100" OVER serve WHERE properties(edge).start_year == "2021" YIELD dst(edge) AS id | DELETE VERTEX $-.id;
```

删除过程

NebulaGraph找到目标点并删除，该目标点的所有邻边（出边和入边）将成为悬挂边。

Caution

- 不支持原子性删除，如果发生错误请重试，避免出现部分删除的情况。否则会导致悬挂边。
- 删除超级节点耗时较多，为避免删除完成前连接超时，可以调整 `nebula-graphd.conf` 中的参数 `--storage_client_timeout_ms` 延长超时时间。

悬挂边视频

- [NebulaGraph 的悬挂边小科普](#) (2 分 28 秒)

最后更新: May 4, 2023

4.11.3 UPDATE VERTEX

`UPDATE VERTEX` 语句可以修改点上 Tag 的属性值。

NebulaGraph 支持 CAS (compare and set) 操作。

Note

一次只能修改一个 Tag。

语法

```
UPDATE VERTEX ON <tag_name> <vid>
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

参数	是否必须	说明	示例
ON <tag_name>	是	指定点的 Tag。要修改的属性必须在这个 Tag 内。	ON player
<vid>	是	指定要修改的点 ID。	"player100"
SET <update_prop>	是	指定如何修改属性值。	SET age = age +1
WHEN <condition>	否	指定过滤条件。如果 <condition> 结果为 false， SET 子句不会生效。	WHEN name == "Tim"
YIELD <output>	否	指定语句的输出格式。	YIELD name AS Name

示例

```
// 查看点"player101"的属性。
nebula> FETCH PROP ON player "player101" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 36, name: "Tony Parker"} |
+-----+

// 修改属性 age 的值，并返回 name 和新的 age。
nebula> UPDATE VERTEX ON player "player101" \
    SET age = age + 2 \
    WHEN name == "Tony Parker" \
    YIELD name AS Name, age AS Age;
+-----+
| Name      | Age   |
+-----+-----+
| "Tony Parker" | 38 |
+-----+-----+
```

最后更新: May 4, 2023

4.11.4 UPSERT VERTEX

UPSERT VERTEX 语句结合 UPDATE 和 INSERT，如果点存在，会修改点的属性值；如果点不存在，会插入新的点。



UPSERT VERTEX 一次只能修改一个 Tag。

UPSERT VERTEX 性能远低于 INSERT，因为 UPSERT 是一组分片级别的读取、修改、写入操作。



并发 UPSERT 同一个 TAG 或 EDGE TYPE 会报错。

语法

```
UPSERT VERTEX ON <tag> <vid>
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

参数	是否必须	说明	示例
ON <tag>	是	指定点的 Tag。要修改的属性必须在这个 Tag 内。	ON player
<vid>	是	指定要修改或插入的点 ID。	"player100"
SET <update_prop>	是	指定如何修改属性值。	SET age = age +1
WHEN <condition>	否	指定过滤条件。	WHEN name == "Tim"
YIELD <output>	否	指定语句的输出格式。	YIELD name AS Name

插入不存在的点

如果点不存在，无论 WHEN 子句的条件是否满足，都会插入点，同时执行 SET 子句，因此新插入的点的属性值取决于：

- SET 子句。
- 属性是否有默认值。

例如：

- 要插入的点包含基于 Tag player 的属性 name 和 age。
- SET 子句指定 age=30。

不同情况下的属性值如下表。

是否满足 WHEN 子句条件	属性是否有默认值	name 属性值	age 属性值
是	是	默认值	30
是	否	NULL	30
否	是	默认值	30
否	否	NULL	30

示例如下：

```
// 查看三个点是否存在，结果“Empty set”表示顶点不存在。
nebula> FETCH PROP ON * "player666", "player667", "player668" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| Empty set |

nebula> UPSERT VERTEX ON player "player666" \
    SET age = 30 \
    WHEN name == "Joe" \
    YIELD name AS Name, age AS Age;
+-----+
| Name      | Age      |
+-----+
| __NULL__  | 30      |
+-----+

nebula> UPSERT VERTEX ON player "player666" \
    SET age = 31 \
    WHEN name == "Joe" \
    YIELD name AS Name, age AS Age;
+-----+
| Name      | Age      |
+-----+
| __NULL__  | 30      |
+-----+

nebula> UPSERT VERTEX ON player "player667" \
    SET age = 31 \
    YIELD name AS Name, age AS Age;
+-----+
| Name      | Age      |
+-----+
| __NULL__  | 31      |
+-----+

nebula> UPSERT VERTEX ON player "player668" \
    SET name = "Amber", age = age + 1 \
    YIELD name AS Name, age AS Age;
+-----+
| Name      | Age      |
+-----+
| "Amber"   | __NULL__ |
+-----+
```

上面最后一个示例中，因为 `age` 没有默认值，插入点时，`age` 默认值为 `NULL`，执行 `age = age + 1` 后仍为 `NULL`。如果 `age` 有默认值，则 `age = age + 1` 可以正常执行，例如：

```
nebula> CREATE TAG IF NOT EXISTS player_with_default(name string, age int DEFAULT 20);
Execution succeeded

nebula> UPSERT VERTEX ON player_with_default "player101" \
    SET age = age + 1 \
    YIELD name AS Name, age AS Age;
+-----+
| Name      | Age      |
+-----+
| __NULL__  | 21      |
+-----+
```

修改存在的点

如果点存在，且满足 `WHEN` 子句的条件，就会修改点的属性值。

```
nebula> FETCH PROP ON player "player101" YIELD properties(vertex);
+-----+
| properties(VERTEX) |
+-----+
| {age: 36, name: "Tony Parker"} |
+-----+

nebula> UPSERT VERTEX ON player "player101" \
    SET age = age + 2 \
    WHEN name == "Tony Parker" \
    YIELD name AS Name, age AS Age;
+-----+
| Name      | Age      |
+-----+
| "Tony Parker" | 38      |
+-----+
```

如果点存在，但是不满足 `WHEN` 子句的条件，修改不会生效。

```
nebula> FETCH PROP ON player "player101" YIELD properties(vertex);
+-----+
| properties(VERTEX)      |
+-----+
| {age: 38, name: "Tony Parker"} |
+-----+  
  
nebula> UPSERT VERTEX ON player "player101" \
    SET age = age + 2 \
    WHEN name == "Someone else" \
    YIELD name AS Name, age AS Age;
+-----+-----+
| Name      | Age   |
+-----+-----+
| "Tony Parker" | 38 |
+-----+-----+
```

最后更新: May 4, 2023

4.12 边语句

4.12.1 INSERT EDGE

`INSERT EDGE` 语句可以在NebulaGraph实例的指定图空间中插入一条或多条边。边是有方向的，从起始点 (`src_vid`) 到目的点 (`dst_vid`)。

`INSERT EDGE` 的执行方式为覆盖式插入。如果已有 Edge type、起点、终点、rank 都相同的边，则覆盖原边。

语法

```
INSERT EDGE [IF NOT EXISTS] <edge_type> ( <prop_name_list> ) VALUES
<src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list>
[, <src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> ), ...];

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

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

- `IF NOT EXISTS`：用户可以使用 `IF NOT EXISTS` 关键字检测待插入的边是否存在，只有不存在时，才会插入。

Note

- `IF NOT EXISTS` 仅检测<边的类型、起始点、目的点和 `rank`>是否存在，不会检测属性值是否重合。
- `IF NOT EXISTS` 会先读取一次数据是否存在，因此对性能会有明显影响。

- `<edge_type>`：边关联的 Edge type，只能指定一个 Edge type。Edge type 必须提前创建，详情请参见 [CREATE EDGE](#)。
- `<prop_name_list>`：需要设置的属性名称列表。
- `src_vid`：起始点 ID，表示边的起点。
- `dst_vid`：目的点 ID，表示边的终点。
- `rank`：可选项。边的 rank 值。数据类型为 `int`。默认值为 0。

↑ openCypher 兼容性

openCypher 中没有 `rank` 的概念。

- `<prop_value_list>`：根据 `prop_name_list` 填写属性值。如果属性值和 Edge type 中的数据类型不匹配，会返回错误。如果没有填写属性值，而 Edge type 中对应的属性设置为 `NOT NULL`，也会返回错误。详情请参见 [CREATE EDGE](#)。

示例

```
# 插入不包含属性的边。
nebula> CREATE EDGE IF NOT EXISTS e1();
nebula> INSERT EDGE e1 () VALUES "10"->"11":();

# 插入 rank 为 1 的边。
nebula> INSERT EDGE e1 () VALUES "10"->"11"@1:();

nebula> CREATE EDGE IF NOT EXISTS e2 (name string, age int);
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", 1);

# 一次插入 2 条边。
nebula> INSERT EDGE e2 (name, age) VALUES \
"12"->"13":("n1", 1), "13"->"14":("n2", 2);
```

```
# 创建失败，因为“a13”不是 int 类型。
nebula> INSERT EDGE e2 (name, age) VALUES "11"-">"13":("n1", "a13");
```

一条边可以多次插入属性值，以最后一次为准。

```
# 多次插入属性值。
nebula> INSERT EDGE e2 (name, age) VALUES "11"-">"13":("n1", 12);
nebula> INSERT EDGE e2 (name, age) VALUES "11"-">"13":("n1", 13);
nebula> INSERT EDGE e2 (name, age) VALUES "11"-">"13":("n1", 14);
nebula> FETCH PROP ON e2 "11"-">"13" YIELD edge AS e;
+-----+
| e
+-----+
| [:e2 "11"-">"13" @0 {age: 14, name: "n1"}] |
+-----+
```

使用 IF NOT EXISTS 插入已存在的边时，不会进行修改。

```
# 插入边。
nebula> INSERT EDGE e2 (name, age) VALUES "14"-">"15">@1:(“n1”, 12);
# 使用 IF NOT EXISTS 修改边，因为边已存在，不会进行修改。
nebula> INSERT EDGE IF NOT EXISTS e2 (name, age) VALUES "14"-">"15">@1:(“n2”, 13);
nebula> FETCH PROP ON e2 "14"-">"15"@1 YIELD edge AS e;
+-----+
| e
+-----+
| [:e2 "14"-">"15" @1 {age: 12, name: "n1"}] |
+-----+
```

Note

- NebulaGraph 3.4.1 允许存在悬挂边（Dangling edge）。因此可以在起点或者终点存在前，先写边；此时就可以通过 `<edgetype>._src` 或 `<edgetype>._dst` 获取到（尚未写入的）点 VID（不建议这样使用）。
- 目前还不能保证操作的原子性，如果失败请重试，否则会发生部分写入。此时读取该数据的行为是未定义的。例如写入操作涉及到多个机器时，可能会出现插入单个边的正反向边只写入成功一个，或者插入多个边时只写入成功一部分，此时会返回报错，请重新执行命令。
- 并发写入同一条边会报 `edge conflict` 错误，可稍后重试。
- 边的 `INSERT` 速度大约是点的 `INSERT` 速度一半。原因是 `INSERT` 边会对应 `storaged` 的两个 `INSERT`，`INSERT` 点对应 `storaged` 的一个 `INSERT`。

最后更新: May 4, 2023

4.12.2 DELETE EDGE

`DELETE EDGE` 语句可以删除边。一次可以删除一条或多条边。用户可以结合管道符一起使用，详情请参见[管道符](#)。

如果需要删除一个点的所有出边，请删除这个点。详情请参见 [DELETE VERTEX](#)。

语法

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



如果不指定 `rank`，则仅仅删除 `rank` 为 0 的边。需要删除所有的 `rank`，见如下示例。

示例

```
nebula> DELETE EDGE serve "player100" -> "team204"@0;  
  
# 结合管道符，删除两点之间同类型的所有rank的边。  
nebula> GO FROM "player100" OVER follow \  
    WHERE dst(edge) == "player101" \  
    YIELD src(edge) AS src, dst(edge) AS dst, rank(edge) AS rank \  
    | DELETE EDGE follow $-.src -> $-.dst @ $-.rank;
```

最后更新: May 4, 2023

4.12.3 UPDATE EDGE

`UPDATE EDGE` 语句可以修改边上 Edge type 的属性。

NebulaGraph 支持 CAS (compare and swap) 操作。

语法

```
UPDATE EDGE ON <edge_type>
<src_vid> -> <dst_vid> [<rank>]
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

参数	是否必须	说明	示例
ON <edge_type>	是	指定 Edge type。要修改的属性必须在这个 Edge type 内。	ON serve
<src_vid>	是	指定边的起始点 ID。	"player100"
<dst_vid>	是	指定边的目的点 ID。	"team204"
<rank>	否	指定边的 rank 值。	10
SET <update_prop>	是	指定如何修改属性值。	SET start_year = start_year +1
WHEN <condition>	否	指定过滤条件。如果 <condition> 结果为 false , SET 子句不会生效。	WHEN end_year < 2010
YIELD <output>	否	指定语句的输出格式。	YIELD start_year AS Start_Year

示例

```
// 用 GO 语句查看边的属性值。
nebula> GO FROM "player100" \
    OVER serve \
    YIELD properties(edge).start_year, properties(edge).end_year;
+-----+-----+
| properties(EDGE).start_year | properties(EDGE).end_year |
+-----+-----+
| 1997 | 2016 |
+-----+-----+

// 修改属性 start_year 的值，并返回 end_year 和新的 start_year。
nebula> UPDATE EDGE ON serve "player100" -> "team204" @0 \
    SET start_year = start_year + 1 \
    WHEN end_year > 2010 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| 1998 | 2016 |
+-----+-----+
```

最后更新: May 4, 2023

4.12.4 UPSERT EDGE

`UPSERT EDGE` 语句结合 `UPDATE` 和 `INSERT`，如果边存在，会更新边的属性；如果边不存在，会插入新的边。

`UPSERT EDGE` 性能远低于 `INSERT`，因为 `UPSERT` 是一组分片级别的读取、修改、写入操作。



并发 `UPSERT` 同一个 TAG 或 EDGE TYPE 会报错。

语法

```
UPSERT EDGE ON <edge_type>
<src_vid> -> <dst_vid> [<rank>]
SET <update_prop>
[WHEN <condition>]
[YIELD <properties>]
```

参数	是否必须	说明	示例
<code>ON <edge_type></code>	是	指定 Edge type。要修改的属性必须在这个 Edge type 内。	<code>ON serve</code>
<code><src_vid></code>	是	指定边的起始点 ID。	<code>"player100"</code>
<code><dst_vid></code>	是	指定边的目的点 ID。	<code>"team204"</code>
<code><rank></code>	否	指定边的 <code>rank</code> 值。数据类型为 <code>int</code> 。	<code>10</code>
<code>SET <update_prop></code>	是	指定如何修改属性值。	<code>SET start_year = start_year +1</code>
<code>WHEN <condition></code>	否	指定过滤条件。	<code>WHEN end_year < 2010</code>
<code>YIELD <output></code>	否	指定语句的输出格式。	<code>YIELD start_year AS Start_Year</code>

插入不存在的边

如果边不存在，无论 `WHEN` 子句的条件是否满足，都会插入边，同时执行 `SET` 子句，因此新插入的边的属性值取决于：

- `SET` 子句。
- 属性是否有默认值。

例如：

- 要插入的边包含基于 Edge type `serve` 的属性 `start_year` 和 `end_year`。
- `SET` 子句指定 `end_year = 2021`。

不同情况下的属性值如下表。

是否满足 <code>WHEN</code> 子句条件	属性是否有默认值	<code>start_year</code> 属性值	<code>end_year</code> 属性值
是	是	默认值	<code>2021</code>
是	否	<code>NULL</code>	<code>2021</code>
否	是	默认值	<code>2021</code>
否	否	<code>NULL</code>	<code>2021</code>

示例如下：

```
// 查看如下三个点是否有 serve 类型的出边，结果“Empty set”表示没有 serve 类型的出边。
nebula> GO FROM "player666", "player667", "player668" \
    OVER serve \
    YIELD properties(edge).start_year, properties(edge).end_year;
+-----+-----+
| properties(EDGE).start_year | properties(EDGE).end_year |
+-----+-----+
+-----+-----+
Empty set

nebula> UPSERT EDGE on serve \
    "player666" -> "team200" @0 \
    SET end_year = 2021 \
    WHEN end_year == 2010 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| _NULL_ | 2021 |
+-----+-----+

nebula> UPSERT EDGE on serve \
    "player666" -> "team200" @0 \
    SET end_year = 2022 \
    WHEN end_year == 2010 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| _NULL_ | 2021 |
+-----+-----+

nebula> UPSERT EDGE on serve \
    "player667" -> "team200" @0 \
    SET end_year = 2022 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| _NULL_ | 2022 |
+-----+-----+

nebula> UPSERT EDGE on serve \
    "player668" -> "team200" @0 \
    SET start_year = 2000, end_year = end_year + 1 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| 2000 | _NULL_ |
+-----+-----+
```

上面最后一个示例中，因为 `end_year` 没有默认值，插入边时，`end_year` 默认值为 `NULL`，执行 `end_year = end_year + 1` 后仍为 `NULL`。如果 `end_year` 有默认值，则 `end_year = end_year + 1` 可以正常执行，例如：

```
nebula> CREATE EDGE IF NOT EXISTS serve_with_default(start_year int, end_year int DEFAULT 2010);
Execution succeeded

nebula> UPSERT EDGE on serve_with_default \
    "player668" -> "team200" \
    SET end_year = end_year + 1 \
    YIELD start_year, end_year;
+-----+-----+
| start_year | end_year |
+-----+-----+
| _NULL_ | 2011 |
+-----+-----+
```

修改存在的边

如果边存在，且满足 `WHEN` 子句的条件，就会修改边的属性值。

```
nebula> MATCH (v:player{name:"Ben Simmons"})-[e:serve]-(v2) \
    RETURN e;
+-----+
| e |
+-----+
| [:serve "player149" ->"team219" @0 {end_year: 2019, start_year: 2016}] |
+-----+

nebula> UPSERT EDGE on serve \
    "player149" -> "team219" \
    SET end_year = end_year + 1 \
    WHEN start_year == 2016 \
    YIELD start_year, end_year;
```

start_year	end_year
2016	2020

如果边存在，但是不满足 WHEN 子句的条件，修改不会生效。

```
nebula> MATCH (v:player{name:"Ben Simmons"})-[e:serve]-(v2) \
    RETURN e;
+-----+
| e |
+-----+
| [:serve "player149"->"team219" @0 {end_year: 2020, start_year: 2016}] |
+-----+  
  
nebula> UPSERT EDGE on serve \
    "player149" -> "team219" \
    SET end_year = end_year + 1 \
    WHEN start_year != 2016 \
    YIELD start_year, end_year;
+-----+
| start_year | end_year |
+-----+
| 2016       | 2020      |
+-----+
```

最后更新: May 4, 2023

4.13 原生索引

4.13.1 索引介绍

为了提高查询性能，NebulaGraph 支持为点的 Tag 或 Tag 的某个属性，边的 Edge type 或 Edge type 的某个属性创建索引。索引可以基于指定的 Tag、Edge type、属性查询数据，但是索引本身不存储数据，而是存储数据的位置。

NebulaGraph 支持两种类型索引：原生索引和全文索引。

使用说明

- 索引可以提高查询性能，但是会降低写入性能。
- 索引是执行 LOOKUP 语句时用于定位到数据的前置条件，如果没有索引，执行 LOOKUP 语句会报错。
- 使用索引时，NebulaGraph 会自动选择最优的索引。
- 具有高选择度的索引，即索引列中不同值的记录数与总记录数的比值较高（例如身份证号的比值为 1）可显著提升查询性能；而对于低选择度的索引（例如国家），查询性能可能不会带来显著提升。

原生索引

原生索引可以基于指定的属性查询数据，有如下特点：

- 包括 Tag 索引和 Edge type 索引。
- 必须手动重建索引（REBUILD INDEX）。
- 支持创建同一个 Tag 或 Edge type 的多个属性的索引（复合索引），但是不能跨 Tag 或 Edge type。

原生索引操作

- CREATE INDEX
- SHOW CREATE INDEX
- SHOW INDEXES
- DESCRIBE INDEX
- REBUILD INDEX
- SHOW INDEX STATUS
- DROP INDEX
- LOOKUP
- MATCH

全文索引

全文索引是基于 Elasticsearch 来实现的，用于对字符串属性进行前缀搜索、通配符搜索、正则表达式搜索和模糊搜索，有如下特点：

- 只允许创建一个属性的索引。
- 不支持逻辑操作，例如 AND、OR、NOT。

Note

如果需要进行整个字符串的匹配，请使用原生索引。

没有 **NULL** 值索引

不支持对值为 **NULL** 的属性创建索引。

没有唯一索引

在 Cypher 中，可以通过 `Constraints` 实现属性值的唯一性限制。在 MySQL 中，可以建立唯一索引来限制某字段只有唯一值。在 nGQL 中没有属性的唯一索引（用户自行保证属性值的唯一性）。

数字、日期和时间类型的范围查询

原生索引还支持对数字、日期和时间类型的属性进行范围查询，不支持其他属性类型的范围查询。

最后更新: May 4, 2023

4.13.2 CREATE INDEX

前提条件

创建索引之前，请确保相关的 Tag 或 Edge type 已经创建。如何创建 Tag 和 Edge type，请参见 [CREATE TAG](#) 和 [CREATE EDGE](#)。

如何创建全文索引，请参见[部署全文索引](#)。

使用索引必读

索引的概念和使用限制都较为复杂。在使用索引前，请务必阅读以下内容。

`CREATE INDEX` 语句用于对 Tag、EdgeType 或其属性创建原生索引。通常分别称为“Tag 索引”、“Edge type 索引”和“属性索引”。

- Tag 索引和 Edge type 索引应用于和 Tag、Edge type 自身相关的查询，例如用 `LOOKUP` 查找有 Tag `player` 的所有点。
- “属性索引”应用于基于属性的查询，例如基于属性 `age` 找到 `age == 19` 的所有的点。

如果已经为 Tag `T` 的属性 `A` 建立过属性索引 `i_TA` (`T` 的索引为 `i_T`)，索引之间的可替代关系如下（Edge type 索引同理）：

- 查询引擎可以使用 `i_TA` 来替代 `i_T`。
- 在 `MATCH`、`LOOKUP` 语句中 `i_T` 可以替代 `i_TA` 查找属性。

版本兼容性

在此前的版本中，`LOOKUP` 语句中的 Tag 或 Edge type 索引不可替代属性索引用于属性查找。

使用替代索引进行查询虽然能获得相同的结果，但查询性能会根据选择的索引有所区别。

Caution

不要任意在生产环境中使用索引，除非很清楚使用索引对业务的影响。索引会导致写性能大幅下降。

长索引会降低 Storage 服务的扫描性能，以及占用更多内存。建议将索引长度设置为和要被索引的最长字符串相同。索引长度最长为 256。

操作步骤

如果必须使用索引，通常按照如下步骤：

1. 初次导入数据至NebulaGraph。
2. 创建索引。
3. [重建索引](#)。
4. 使用 `LOOKUP` 或 `MATCH` 语句查询数据。不需要（也无法）指定使用哪个索引，NebulaGraph会自动计算。

Note

如果先创建索引再导入数据，会因为写性能的下降导致导入速度极慢。

日常增量写入时保持 `--disable_auto_compaction = false`。

新创建的索引并不会立刻生效。创建新的索引并尝试立刻使用（例如 `LOOKUP` 或者 `REBUILD INDEX`）通常会失败（报错 `can't find xxx in the space`）。因为创建步骤是异步实现的，NebulaGraph要在下一个心跳周期才能完成索引的创建。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。如果需要修改心跳间隔，请为[所有配置文件](#)修改参数 `heartbeat_interval_secs`。

Danger

创建索引，或者删除并再次创建同名索引后，必须 REBUILD INDEX。否则无法在 MATCH 和 LOOKUP 语句中返回这些数据。

语法

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

参数	说明
TAG EDGE	指定要创建的索引类型。
IF NOT EXISTS	检测待创建的索引是否存在，只有不存在时，才会创建索引。
<index_name>	索引名。索引名在一个图空间中必须是唯一的。推荐的命名方式为 i_tagName_propName 。 索引名称不能以数字开头，支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等，但是不包括除下划线外的特殊字符。使用特殊字符、保留关键字或数字开头时，需要用反引号 (`) 包围，详情参见 关键字和保留字 。 注意：如果以中文为索引命名，报 SyntaxError 错误时，需使用反引号 (`) 包围中文字符。
<tag_name> <edge_name>	指定索引关联的 Tag 或 Edge 名称。
<prop_name_list>	为变长字符串属性创建索引时，必须用 prop_name(length) 指定索引长度；为 Tag 或 Edge type 本身创建索引时，忽略 <prop_name_list>。
COMMENT	索引的描述。最大为 256 字节。默认无描述。

创建 Tag/Edge type 索引

```
nebula> CREATE TAG INDEX player_index on player();
nebula> CREATE EDGE INDEX follow_index on follow();
```

为 Tag 或 Edge type 创建索引后，用户可以使用 LOOKUP 语句查找 带有该 Tag 的所有点的 VID，或者 所有该类型的边 的 对应起始点 VID、目的点 VID、以及 rank 。详情请参见 [LOOKUP](#) 。

创建单属性索引

```
nebula> CREATE TAG INDEX IF NOT EXISTS player_index_0 on player(name(10));
```

上述示例是为所有包含 Tag player 的点创建属性 name 的索引，索引长度为 10。即只使用属性 name 的前 10 个字符来创建索引。

```
# 变长字符串需要指定索引长度。
nebula> CREATE TAG IF NOT EXISTS var_string(p1 string);
nebula> CREATE TAG INDEX IF NOT EXISTS var ON var_string(p1(10));

# 定长字符串不需要指定索引长度。
nebula> CREATE TAG IF NOT EXISTS fix_string(p1 FIXED_STRING(10));
nebula> CREATE TAG INDEX IF NOT EXISTS fix ON fix_string(p1);

nebula> CREATE EDGE INDEX IF NOT EXISTS follow_index_0 on follow(degree);
```

创建复合属性索引

复合属性索引用于查找一个 Tag （或者 Edge type）中的多个属性（的组合）。

```
nebula> CREATE TAG INDEX IF NOT EXISTS player_index_1 on player(name(10), age);
```

 **Caution**

不支持跨 Tag 或 Edge type 创建复合索引。

 **Note**

使用复合属性索引时，遵循“最左匹配原则”，必须从复合属性索引的最左侧开始匹配。

最后更新: May 4, 2023

4.13.3 SHOW INDEXES

SHOW INDEXES 语句可以列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。

语法

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

示例

```
nebula> SHOW TAG INDEXES;
+-----+-----+-----+
| Index Name | By Tag | Columns |
+-----+-----+-----+
| "fix"      | "fix_string" | ["p1"]   |
| "player_index_0" | "player" | ["name"] |
| "player_index_1" | "player" | ["name", "age"] |
| "var"       | "var_string" | ["p1"]   |
+-----+-----+-----+

nebula> SHOW EDGE INDEXES;
+-----+-----+-----+
| Index Name | By Edge | Columns |
+-----+-----+-----+
| "follow_index" | "follow" | []     |
+-----+-----+-----+
```

最后更新: May 4, 2023

4.13.4 SHOW CREATE INDEX

`SHOW CREATE INDEX` 展示创建 Tag 或者 Edge type 时使用的 nGQL 语句，其中包含索引的详细信息，例如其关联的属性。

语法

```
SHOW CREATE {TAG | EDGE} INDEX <index_name>;
```

示例

用户可以先运行 `SHOW TAG INDEXES` 查看有哪些 Tag 索引，然后用 `SHOW CREATE TAG INDEX` 查看指定索引的创建信息。

```
nebula> SHOW TAG INDEXES;
+-----+-----+
| Index Name | By Tag | Columns |
+-----+-----+
| "player_index_0" | "player" | [] |
| "player_index_1" | "player" | ["name"] |
+-----+-----+

nebula> SHOW CREATE TAG INDEX player_index_1;
+-----+
| Tag Index Name | Create Tag Index |
+-----+
| "player_index_1" | "CREATE TAG INDEX `player_index_1` ON `player` ( |
|                   |   `name`(20) |
|                   | )"
+-----+
```

Edge type 索引可以用类似的方法查询：

```
nebula> SHOW EDGE INDEXES;
+-----+-----+
| Index Name | By Edge | Columns |
+-----+-----+
| "follow_index" | "follow" | []
+-----+-----+

nebula> SHOW CREATE EDGE INDEX follow_index;
+-----+
| Edge Index Name | Create Edge Index |
+-----+
| "follow_index" | "CREATE EDGE INDEX `follow_index` ON `follow` ( |
|                   | )"
+-----+
```

最后更新: May 4, 2023

4.13.5 DESCRIBE INDEX

DESCRIBE INDEX 语句可以查看指定索引的信息，包括索引的属性名称（Field）和数据类型（Type）。

语法

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

示例

```
nebula> DESCRIBE TAG INDEX player_index_0;
+-----+-----+
| Field | Type   |
+-----+-----+
| "name" | "fixed_string(30)" |
+-----+-----+

nebula> DESCRIBE TAG INDEX player_index_1;
+-----+-----+
| Field | Type   |
+-----+-----+
| "name" | "fixed_string(10)" |
| "age"  | "int64" |
+-----+-----+
```

最后更新: May 4, 2023

4.13.6 REBUILD INDEX

Danger

- 索引功能不会自动对其创建之前已存在的存量数据生效——在索引重建完成之前，无法基于该索引使用 `LOOKUP` 和 `MATCH` 语句查询到存量数据。
- 索引的重建未完成时，依赖索引的查询仅能使用部分索引，因此不能获得准确结果。

请在创建索引后，选择合适的时间为存量数据重建索引。使用索引的详情请参见 [CREATE INDEX](#)。

Performance

通过修改配置文件中的 `rebuild_index_part_rate_limit` 和 `rebuild_index_batch_size` 两个参数，可优化重建索引的速度，另外，更大参数可能会导致更高的内存和网络占用，详情请参见 [Storage服务配置](#)。

语法

```
REBUILD {TAG | EDGE} INDEX [<index_name_list>];

<index_name_list> ::=  
[index_name [, index_name] ...]
```

- 可以一次重建多个索引，索引名称之间用英文逗号（,）分隔。如果没有指定索引名称，将会重建所有索引。
- 重建完成后，用户可以使用命令 `SHOW {TAG | EDGE} INDEX STATUS` 检查索引是否重建完成。详情请参见 [SHOW INDEX STATUS](#)。

示例

```
nebula> CREATE TAG IF NOT EXISTS person(name string, age int, gender string, email string);
nebula> CREATE TAG INDEX IF NOT EXISTS single_person_index ON person(name(10));

# 重建索引，返回任务 ID。
nebula> REBUILD TAG INDEX single_person_index;
+-----+
| New Job Id |
+-----+
| 31          |
+-----+

# 查看索引状态。
nebula> SHOW TAG INDEX STATUS;
+-----+-----+
| Name      | Index Status |
+-----+-----+
| "single_person_index" | "FINISHED"   |
+-----+-----+

# 也可以使用 SHOW JOB <job_id> 查看重建索引的任务状态。
nebula> SHOW JOB 31;
+-----+-----+-----+-----+-----+
| Job Id(TaskId) | Command(Dest) | Status    | Start Time        | Stop Time       | Error Code |
+-----+-----+-----+-----+-----+
| 31           | "REBUILD_TAG_INDEX" | "FINISHED" | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:24.000 | "SUCCEEDED" |
| 0            | "storaged1"        | "FINISHED" | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 | "SUCCEEDED" |
| 1            | "storaged2"        | "FINISHED" | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 | "SUCCEEDED" |
| 2            | "storaged0"        | "FINISHED" | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 | "SUCCEEDED" |
| "Total:3"    | "Succeeded:3"      | "Failed:0" | "In Progress:0"      | ""               | ""           |
+-----+-----+-----+-----+-----+
```

NebulaGraph创建一个任务去重建索引，因此可以根据返回的任务 ID，通过 `SHOW JOB <job_id>` 语句查看任务状态。详情请参见 [SHOW JOB](#)。

最后更新: May 4, 2023

4.13.7 SHOW INDEX STATUS

SHOW INDEX STATUS 语句可以查看索引名称和对应作业的状态。

索引状态包括:

- QUEUE : 队列中
- RUNNING : 执行中
- FINISHED : 已完成
- FAILED : 失败
- STOPPED : 停止
- INVALID : 失效



如何创建索引请参见 [CREATE INDEX](#)。

语法

```
SHOW {TAG | EDGE} INDEX STATUS;
```

示例

```
nebula> SHOW TAG INDEX STATUS;
+-----+-----+
| Name | Index Status |
+-----+-----+
| "player_index_0" | "FINISHED" |
| "player_index_1" | "FINISHED" |
+-----+-----+
```

最后更新: May 4, 2023

4.13.8 DROP INDEX

DROP INDEX 语句可以删除当前图空间中已存在的索引。

前提条件

执行 DROP INDEX 语句需要当前登录的用户拥有指定图空间的 DROP TAG INDEX 和 DROP EDGE INDEX 权限，否则会报错。

语法

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

IF EXISTS：检测待删除的索引是否存在，只有存在时，才会删除索引。

示例

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

最后更新: May 4, 2023

4.14 全文索引

4.14.1 全文索引限制



Caution

本文介绍全文索引的限制，请在使用全文索引前仔细阅读。

全文索引有如下 14 条限制：

- 全文索引当前仅支持 `LOOKUP` 语句。
- 全文索引名称必须以 `nebula_` 开头，只能包含数字、小写字母、下划线，字符串长度不能超过 256 字节。
- 如果 Tag/Edge type 上存在全文索引，无法删除或修改 Tag/Edge type。
- 属性的类型必须为 `STRING` 或 `FIXED_STRING`。
- 全文索引不支持多个 Tag/Edge type 的搜索。
- 不支持排序全文搜索的返回结果，而是按照数据插入的顺序返回。
- 全文索引不支持搜索属性值为 `NULL` 的属性。
- 不支持修改 Elasticsearch 中的索引，只能删除重建。
- 不支持管道符。
- `WHERE` 子句只能用单个条件进行全文搜索。
- 确保同时启动了 Elasticsearch 集群和 NebulaGraph，否则可能导致 Elasticsearch 集群写入的数据不完整。
- 从写入 NebulaGraph，到写入 `listener`，再到写入 Elasticsearch 并创建索引可能需要一段时间。如果访问全文索引时返回未找到索引，可等待索引生效（但是，该等待时间未知，也无返回码检查）。
- 使用 K8s 方式部署的 NebulaGraph 集群不支持全文索引。

最后更新: May 4, 2023

4.14.2 部署全文索引

NebulaGraph的全文索引是基于 [Elasticsearch](#) 实现，这意味着用户可以使用 Elasticsearch 全文查询语言来检索想要的内容。全文索引由内置的进程管理，当 listener 集群和 Elasticsearch 集群部署后，内置的进程只能为数据类型为定长字符串或变长字符串的属性创建全文索引。

注意事项

使用全文索引前，请确认已经了解全文索引的[使用限制](#)。

部署 Elasticsearch 集群

部署 Elasticsearch 集群请参见 [Kubernetes 安装 Elasticsearch](#) 或[单机安装 Elasticsearch](#)。目前仅支持 7.x 版本的 Elasticsearch。

Compatibility

NebulaGraph 3.4 及以后，不需要额外创建模板。

Caution

- 创建全文索引时，索引名称需要以 `nebula_` 开头。

用户可以配置 Elasticsearch 来满足业务需求，如果需要定制 Elasticsearch，请参见 [Elasticsearch 官方文档](#)。

登录文本搜索客户端

部署 Elasticsearch 集群之后，可以使用 `SIGN IN` 语句登录 Elasticsearch 客户端。必须使用 Elasticsearch 配置文件中的 IP 地址和端口才能正常连接，同时登录多个客户端，请在多个 `elastic_ip:port` 之间用英文逗号 (,) 分隔。

语法

```
SIGN IN TEXT SERVICE (<elastic_ip:port>, {HTTP | HTTPS} [<username>, <password>]) [, (<elastic_ip:port>, ...)];
```

示例

```
nebula> SIGN IN TEXT SERVICE (127.0.0.1:9200, HTTP);
```

Note

Elasticsearch 默认没有用户名和密码，如果设置了用户名和密码，请在 `SIGN IN` 语句中指定。

Caution

Elasticsearch 客户端只能登录一次，如有修改，需要 `SIGN OUT` 后重新 `SIGN IN`，且客户端对全局生效，多个图空间共享相同的 Elasticsearch 客户端。

显示文本搜索客户端

`SHOW TEXT SEARCH CLIENTS` 语句可以列出文本搜索客户端。

语法

```
SHOW TEXT SEARCH CLIENTS;
```

示例

```
nebula> SHOW TEXT SEARCH CLIENTS;
+-----+-----+
| Host | Port |
+-----+-----+
| "127.0.0.1" | 9200 |
+-----+-----+
```

退出文本搜索客户端

SIGN OUT TEXT SERVICE 语句可以退出所有文本搜索客户端。

语法

```
SIGN OUT TEXT SERVICE;
```

示例

```
nebula> SIGN OUT TEXT SERVICE;
```

最后更新: May 4, 2023

4.14.3 部署 Raft listener

全文索引的数据是异步写入 Elasticsearch 集群的。流程是通过 Storage 服务的 Raft listener（简称 listener）这个单独部署的进程，从 Storage 服务读取数据，然后将它们写入 Elasticsearch 集群。

前提条件

- 已经了解全文索引的使用限制。
- 已经部署 NebulaGraph 集群。
- 完成部署 Elasticsearch 集群。
- 准备一台或者多台额外的服务器，来部署 Raft listener。

注意事项

- 请保证 NebulaGraph 各组件（Metad、Storage、Graphd、listener）有相同的版本。
- 只能为一个图空间“一次性添加所有的 listener 机器”。尝试向已经存在有 listener 的图空间再添加新 listener 会失败。因此，需在一个命令语句里完整地添加全部的 listener。

部署流程

第一步：安装 STORAGE 服务

listener 进程与 storaged 进程使用相同的二进制文件，但是二者配置文件不同，进程使用端口也不同，可以在所有需要部署 listener 的服务器上都安装 NebulaGraph，但是仅使用 Storage 服务。详情请参见[使用 RPM 或 DEB 安装包安装 NebulaGraph](#)。

第二步：准备 LISTENER 的配置文件

用户必须在需要部署 listener 的机器上准备对应的配置文件，文件名称必须为 `nebula-storaged-listener.conf`，并保存在安装路径下的 `etc` 目录内。用户可以参考提供的[模板](#)。注意去掉文件后缀 `.production`。

大部分配置与 [Storage 服务](#) 的配置文件相同，本文仅介绍差异部分。

名称	预设值	说明
<code>daemonize</code>	<code>true</code>	是否启动守护进程。
<code>pid_file</code>	<code>pids/nebula-storaged-listener.pid</code>	记录进程 ID 的文件。
<code>meta_server_addrs</code>	-	全部 Meta 服务的 IP 地址和端口。多个 Meta 服务用英文逗号（,）分隔。
<code>local_ip</code>	-	listener 服务的本地 IP 地址。
<code>port</code>	-	listener 服务的 RPC 守护进程监听端口。
<code>heartbeat_interval_secs</code>	10	Meta 服务的心跳间隔。单位：秒（s）。
<code>listener_path</code>	<code>data/listener</code>	listener 的 WAL 目录。只允许使用一个目录。
<code>data_path</code>	<code>data</code>	出于兼容性考虑，可以忽略此参数。填充一个默认值 <code>data</code> 。
<code>part_man_type</code>	<code>memory</code>	部件管理器类型，可选值为 <code>memory</code> 和 <code>meta</code> 。
<code>rocksdb_batch_size</code>	4096	批处理操作的默认保留字节。
<code>rocksdb_block_cache</code>	4	BlockBasedTable 的默认块缓存大小。单位：兆字节（MB）。
<code>engine_type</code>	<code>rocksdb</code>	存储引擎类型，例如 <code>rocksdb</code> 、 <code>memory</code> 等。
<code>part_type</code>	<code>simple</code>	部件类型，例如 <code>simple</code> 、 <code>consensus</code> 等。

Note

在配置文件中请使用真实的 (listener 机器) IP 地址替换 127.0.0.1。

第三步：启动 LISTENER

执行如下命令启动启动 listener:

```
./bin/nebula-storaged --flagfile <listener_config_path>/nebula-storaged-listener.conf
```

listener_config_path 是存放 listener 配置文件的路径。

第四步：添加 LISTENER 到NEBULAGRAPH集群

用命令行连接到NebulaGraph，然后执行 **USE <space>** 进入需要创建全文索引的图空间。然后执行如下命令添加 listener:

```
ADD LISTENER ELASTICSEARCH <listener_ip:port> [,<listener_ip:port>, ...]
```

Warning

listener 必须使用真实的 IP 地址。

请在一个语句里完整地添加所有 listener。例如:

```
nebula> ADD LISTENER ELASTICSEARCH 192.168.8.5:9789,192.168.8.6:9789;
```

查看 listener

执行 SHOW LISTENER 语句可以列出所有的 listener。

示例

```
nebula> SHOW LISTENER;
+-----+-----+-----+-----+
| PartId | Type      | Host           | Status   |
+-----+-----+-----+-----+
| 1     | "ELASTICSEARCH" | "[192.168.8.5:46780]" | "ONLINE" |
| 2     | "ELASTICSEARCH" | "[192.168.8.5:46780]" | "ONLINE" |
| 3     | "ELASTICSEARCH" | "[192.168.8.5:46780]" | "ONLINE" |
+-----+-----+-----+-----+
```

删除 listener

执行 REMOVE LISTENER ELASTICSEARCH 语句可以删除图空间的所有 listener。

示例

```
nebula> REMOVE LISTENER ELASTICSEARCH;
```

最后更新: May 4, 2023

4.14.4 全文搜索

全文搜索是基于全文索引对值为字符串类型的属性进行前缀搜索、通配符搜索、正则表达式搜索和模糊搜索。

在 `LOOKUP` 语句中，使用 `WHERE` 子句指定字符串的搜索条件。

前提条件

请确保已经部署全文索引。详情请参见[部署全文索引](#)和[部署 listener](#)。

注意事项

使用全文索引前，请确认已经了解全文索引的[使用限制](#)。

自然语言全文搜索

自然语言搜索将搜索的字符串解释为自然人类语言中的短语。搜索区分大小写，且默认是对字符串进行前缀匹配。例如，有三个点属于标签 `player`，标签 `player` 含有属性 `name`，这三个点的 `name` 分别为 `Kevin Durant`、`Tim Duncan` 和 `David Beckham`。现在已经建立好有关 `player.name` 的全文索引，在用全文索引前缀搜索语句 `LOOKUP ON player WHERE PREFIX(player.name,"D")`；查询时，只有 `David Beckham` 会被查询到。

语法

创建全文索引

```
CREATE FULLTEXT {TAG | EDGE} INDEX <index_name> ON {<tag_name> | <edge_name>} ([<prop_name>]);
```

显示全文索引

```
SHOW FULLTEXT INDEXES;
```

重建全文索引

```
REBUILD FULLTEXT INDEX;
```



数据量大时，重建全文索引速度较慢，可以修改 `Storage` 服务的配置文件（`nebula-storaged.conf`）中 `snapshot_send_files=false`。

删除全文索引

```
DROP FULLTEXT INDEX <index_name>;
```

使用查询选项

```
LOOKUP ON {<tag> | <edge_type>} WHERE <expression> [YIELD <return_list>];  
<expression> ::=  
PREFIX | WILDCARD | REGEXP | FUZZY
```

```
<return_list>
  <prop_name> [AS <prop_alias>] [, <prop_name> [AS <prop_alias>] ...]
```

- PREFIX(schema_name.prop_name, prefix_string, row_limit, timeout)
- WILDCARD(schema_name.prop_name, wildcard_string, row_limit, timeout)
- REGEXP(schema_name.prop_name, regexp_string, row_limit, timeout)
- FUZZY(schema_name.prop_name, fuzzy_string, fuzziness, operator, row_limit, timeout)
 - fuzziness：可选项。允许匹配的最大编辑距离。默认值为 AUTO。查看其他可选值和更多信息，请参见 [Elasticsearch 官方文档](#)。
 - operator：可选项。解释文本的布尔逻辑。可选值为 OR（默认）和 and。
- row_limit：可选项。指定要返回的行数。默认值为 100。
- timeout：可选项。指定超时时间。单位：毫秒（ms）。默认值为 200。

示例

```
//创建图空间。
nebula> CREATE SPACE IF NOT EXISTS basketballplayer (partition_num=3,replica_factor=1, vid_type=fixed_string(30));

//登录文本搜索客户端。
nebula> SIGN IN TEXT SERVICE (127.0.0.1:9200, HTTP);

//检查是否成功登录。
nebula> SHOW TEXT SEARCH CLIENTS;

//切换图空间。
nebula> USE basketballplayer;

//添加 Listener 到NebulaGraph集群。
nebula> ADD LISTENER ELASTICSEARCH 192.168.8.5:9789;

//检查是否成功添加 Listener，当状态为 Online 时表示成功添加。
nebula> SHOW LISTENER;

//创建 Tag。
nebula> CREATE TAG IF NOT EXISTS player(name string, age int);

//创建全文索引，索引名称需要以`nebula_`开头。
nebula> CREATE FULLTEXT TAG INDEX nebula_index_1 ON player(name);

//重建全文索引。
nebula> REBUILD FULLTEXT INDEX;

//查看全文索引。
nebula> SHOW FULLTEXT INDEXES;
+-----+-----+-----+-----+
| Name | Schema Type | Schema Name | Fields |
+-----+-----+-----+-----+
| "nebula_index_1" | "Tag" | "player" | "name" |
+-----+-----+-----+-----+

//插入测试数据。
nebula> INSERT VERTEX player(name, age) VALUES \
  "Russell Westbrook": ("Russell Westbrook", 30), \
  "Chris Paul": ("Chris Paul", 33), \
  "Boris Diaw": ("Boris Diaw", 36), \
  "David West": ("David West", 38), \
  "Danny Green": ("Danny Green", 31), \
  "Tim Duncan": ("Tim Duncan", 42), \
  "James Harden": ("James Harden", 29), \
  "Tony Parker": ("Tony Parker", 36), \
  "Aron Baynes": ("Aron Baynes", 32), \
  "Ben Simmons": ("Ben Simmons", 22), \
  "Blake Griffin": ("Blake Griffin", 30);

//测试查询
nebula> LOOKUP ON player WHERE PREFIX(player.name, "B") YIELD id(vertex);
+-----+
| id(VERTEX) |
+-----+
| "Boris Diaw" |
| "Ben Simmons" |
| "Blake Griffin" |
+-----+

nebula> LOOKUP ON player WHERE WILDCARD(player.name, "*ri*") YIELD player.name, player.age;
+-----+-----+
| name | age |
+-----+-----+
| "Chris Paul" | 33 |
```

```
| "Boris Diaw" | 36 |
| "Blake Griffin" | 30 |
+-----+
nebula> LOOKUP ON player WHERE WILDCARD(player.name, "*ri*") YIELD player.name, player.age | YIELD count(*);
+-----+
| count(*) |
+-----+
| 3 |
+-----+
nebula> LOOKUP ON player WHERE REGEXP(player.name, "R.*") YIELD player.name, player.age;
+-----+
| name | age |
+-----+
| "Russell Westbrook" | 30 |
+-----+
nebula> LOOKUP ON player WHERE REGEXP(player.name, ".") YIELD id(vertex);
+-----+
| id(VERTEX) |
+-----+
| "Danny Green" |
| "David West" |
| "Russell Westbrook" |
+-----+
...
nebula> LOOKUP ON player WHERE FUZZY(player.name, "Tim Dunncan", AUTO, OR) YIELD player.name;
+-----+
| name |
+-----+
| "Tim Duncan" |
+-----+
//删除全文索引。
nebula> DROP FULLTEXT INDEX nebula_index_1;
```

最后更新: May 4, 2023

4.15 子图和路径

4.15.1 GET SUBGRAPH

`GET SUBGRAPH` 语句查询并返回一个通过从指定点出发对图进行游走而生成的子图。在 `GET SUBGRAPH` 语句中，用户可以指定游走的步数以及游走所经过的边的类型或方向。

语法

```
GET SUBGRAPH [WITH PROP] [<step_count> {STEP|STEPS}] FROM {<vid>, <vid>...}
[ {IN | OUT | BOTH} <edge_type>, <edge_type>...]
[WHERE <expression> [AND <expression> ...]]
YIELD [VERTICES AS <vertex_alias>] [, EDGES AS <edge_alias>];
```

- `WITH PROP`：展示属性。不添加本参数则隐藏属性。
- `step_count`：指定从起始点开始的跳数，返回从 0 到 `step_count` 跳的子图。必须是非负整数。默认值为 1。
- `vid`：指定起始点 ID。
- `edge_type`：指定 Edge type。可以用 IN、OUT 和 BOTH 来指定起始点上该 Edge type 的方向。默认为 BOTH。
- `WHERE`：指定遍历的过滤条件，可以结合布尔运算符 AND 使用。
- `YIELD`：定义需要返回的输出。可以仅返回点或边。必须设置别名。

Note

`GET SUBGRAPH` 语句检索的路径类型为 `trail`，即检索的路径只有点可以重复，边不可以重复。详情请参见[路径](#)。

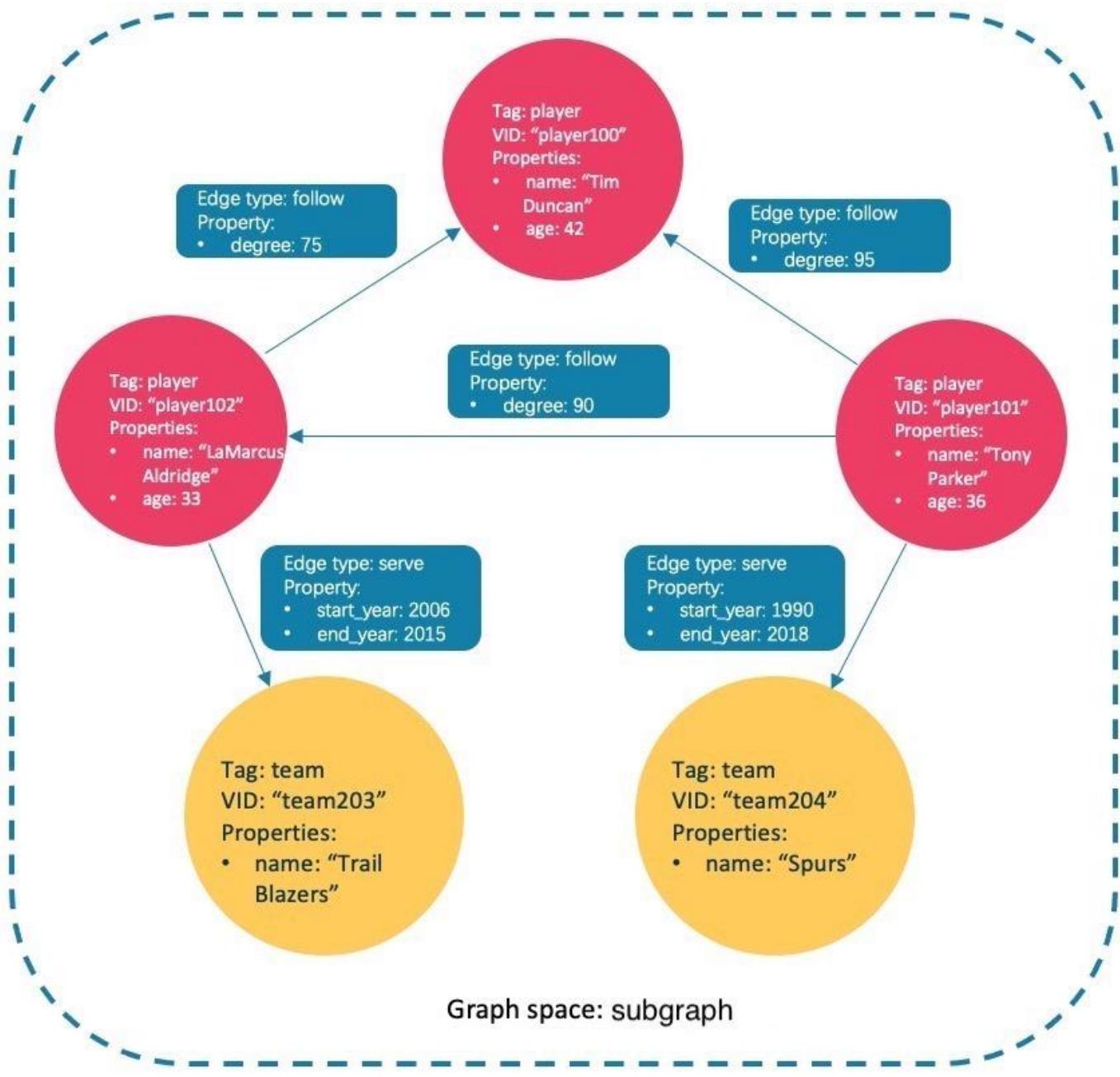
WHERE 语句限制

在 `GET SUBGRAPH` 语句中使用 `WHERE` 子句，注意以下限制：

- 仅支持 AND 运算符。
- 仅支持过滤目的点，点的格式为 `$.tagName.propName`。
- 支持过滤边，边的格式为 `edge_type.propName`。
- 支持数学函数、聚合函数、字符串函数、日期时间函数、列表函数中的通用函数和类型转化函数。
- 不支持聚合函数、Schema 相关函数、条件表达式函数、谓词函数、geo 函数和自定义函数，列表函数中除通用函数以外的函数。

示例

以下面的示例图进行演示。



插入测试数据:

```

nebula> CREATE SPACE IF NOT EXISTS subgraph(partition_num=15, replica_factor=1, vid_type=fixed_string(30));
nebula> USE subgraph;
nebula> CREATE TAG IF NOT EXISTS player(name string, age int);
nebula> CREATE TAG IF NOT EXISTS team(name string);
nebula> CREATE EDGE IF NOT EXISTS follow(degree int);
nebula> CREATE EDGE IF NOT EXISTS serve(start_year int, end_year int);
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42);
nebula> INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36);
nebula> INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);
nebula> INSERT VERTEX team(name) VALUES "team203":("Trail Blazers"), "team204":("Spurs");
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player100":(95);
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player102":(90);

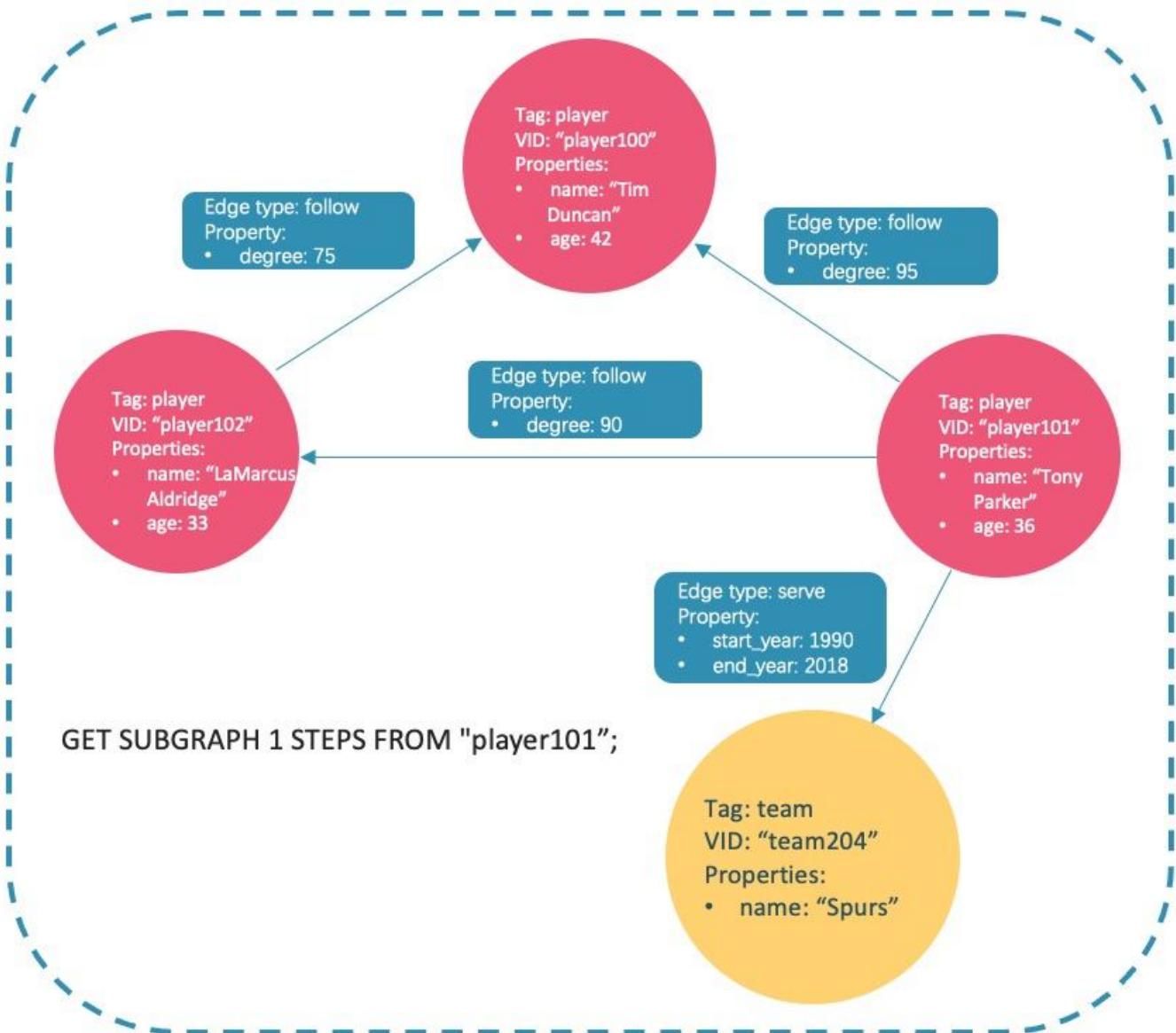
```

```
nebula> INSERT EDGE follow(degree) VALUES "player102" -> "player100":(75);
nebula> INSERT EDGE serve(start_year, end_year) VALUES "player101" -> "team204":(1999, 2018), "player102" -> "team203":(2006, 2015);
```

- 查询从点 player101 开始、0~1 跳、所有 Edge type 的子图。

```
nebula> GET SUBGRAPH 1 STEPS FROM "player101" YIELD VERTICES AS nodes, EDGES AS relationships;
+-----+
| nodes | | relationships |
+-----+
| [{"player101": "player{0}"}] | [[{"serve": "player101->team204": @0}, {"follow": "player101->player100": @0}, {"follow": "player101->player102": @0}], [{"team204": "team{0}"}, {"player100": "player{0}"}, {"player102": "player{0}"}] | [{"follow": "player102->player100": @0}]
+-----+
```

返回的子图如下。



- 查询从点 player101 开始、0~1 跳、follow 类型的入边的子图。

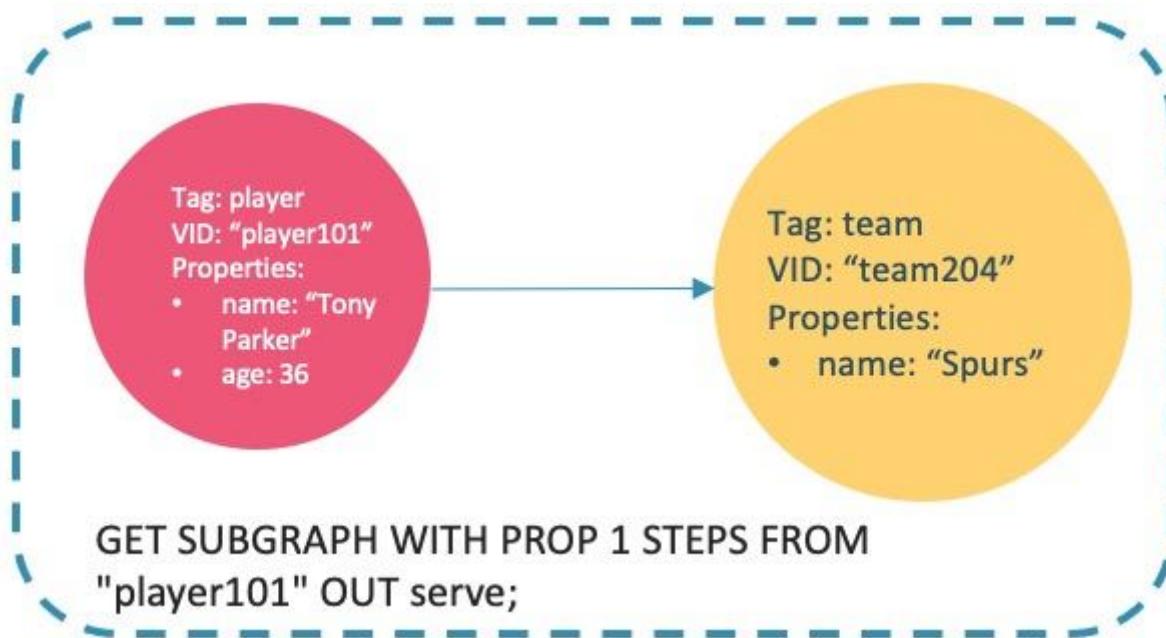
```
nebula> GET SUBGRAPH 1 STEPS FROM "player101" IN follow YIELD VERTICES AS nodes, EDGES AS relationships;
+-----+-----+
| nodes | relationships |
+-----+-----+
| [{"player101":player{}}] | [] |
+-----+-----+
```

因为 player101 没有 follow 类型的入边。所以仅返回点 player101。

- 查询从点 player101 开始、0~1 跳、serve 类型的出边的子图，同时展示边的属性。

```
nebula> GET SUBGRAPH WITH PROP 1 STEPS FROM "player101" OUT serve YIELD VERTICES AS nodes, EDGES AS relationships;
+-----+-----+
| nodes | relationships |
+-----+-----+
| [{"player101":player{age: 36, name: "Tony Parker"}}, [{"serve": "player101->team204 @0 {end_year: 2018, start_year: 1999}"}]] |
| [{"team204":team{name: "Spurs"}}, []] |
+-----+-----+
```

返回的子图如下。



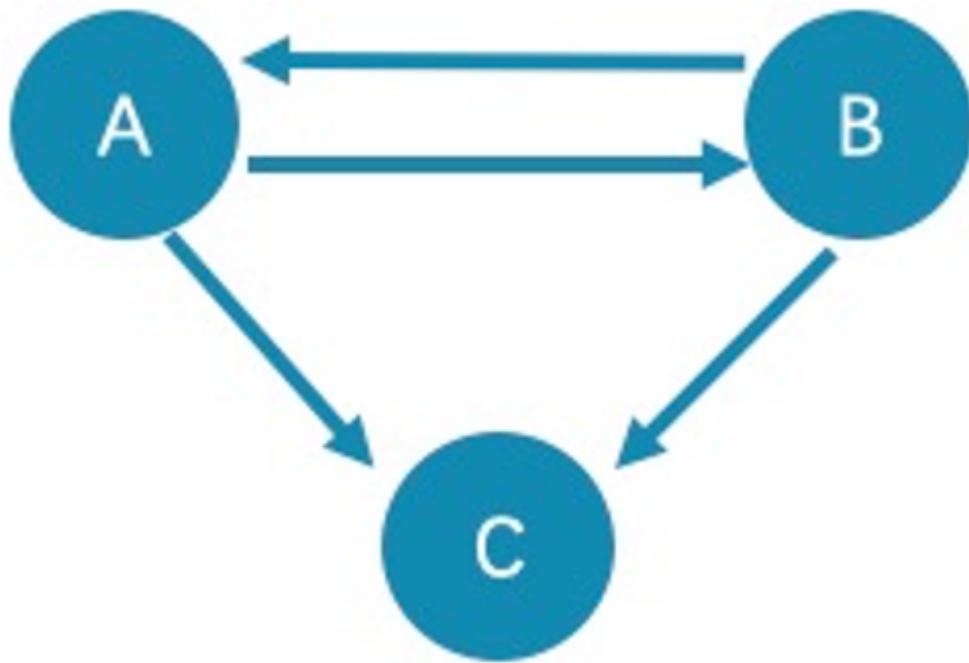
- 查询从点 player101 开始、0~2 跳、follow 类型边 degree 大于 90，年龄大于 30 的子图，同时展示边的属性。

```
nebula> GET SUBGRAPH WITH PROP 2 STEPS FROM "player101" \
    WHERE follow.degree > 90 AND $$.player.age > 30 \
    YIELD VERTICES AS nodes, EDGES AS relationships;
+-----+-----+
| nodes | relationships |
+-----+-----+
| [{"player101":player{age: 36, name: "Tony Parker"}}, [{"follow": "player101->player100 @0 {degree: 95}"}], [{"player100":player{age: 42, name: "Tim Duncan"}}, []]] |
+-----+-----+
```

FAQ

为什么返回结果中会出现超出 **STEP_COUNT** 跳数之外的关系？

为了展示子图的完整性，会在满足条件的所有点上额外查询一跳。例如下图。



- 用 `GET SUBGRAPH 1 STEPS FROM "A";` 查询的满足结果的路径是 `A->B`、`B->A` 和 `A->C`，为了子图的完整性，会在满足结果的点上额外查询一跳，即 `B->C`。
- 用 `GET SUBGRAPH 1 STEPS FROM "A" IN follow;` 查询的满足结果的路径是 `B->A`，在满足结果的点上额外查询一跳，即 `A->B`。

如果只是查询满足条件的路径或点，建议使用 `MATCH` 或 `GO` 语句。例如：

```
nebula> MATCH p=(v:player) -- (v2) WHERE id(v)=="A" RETURN p;
nebula> GO 1 STEPS FROM "A" OVER follow YIELD src(edge),dst(edge);
```

为什么返回结果中会出现低于 `STEP_COUNT` 跳数的关系？

查询到没有多余子图数据时会停止查询，且不会返回空值。

```
nebula> GET SUBGRAPH 100 STEPS FROM "player101" OUT follow YIELD VERTICES AS nodes, EDGES AS relationships;
+-----+-----+-----+
| nodes | relationships |
+-----+-----+-----+
| [{"player101": "player{[]}"}, {"player100": "player{[]}"}, {"player102": "player{[]}"}] | [[{"follow": "player101->player100": 0, "follow": "player101->player102": 0}]] |
| [{"player100": "player{[]}"}, {"player102": "player{[]}"}, {"player101": "player{[]}"}] | [[{"follow": "player102->player100": 0}]] |
+-----+-----+-----+
```

最后更新: May 4, 2023

4.15.2 FIND PATH

FIND PATH 语句查找指定起始点和目的点之间的路径。



用户可在配置文件 `nebula-graphd.conf` 中添加 `num_operator_threads` 参数提高 FIND PATH 的查询性能。`num_operator_threads` 的取值为 2 ~ 10，该值不能超过 Graph 服务所在机器的 CPU 核心个数，建议设置为 Graph 服务所在机器的 CPU 核心个数。关于配置文件的详细信息，参见 [Graph 服务配置](#)。

语法

```
FIND { SHORTEST | ALL | NOLOOP } PATH [WITH PROP] FROM <vertex_id_list> TO <vertex_id_list>
OVER <edge_type_list> [REVERSELY | BIDIRECT]
[<WHERE clause>] [UPTO <N> {STEP|STEPS}]
YIELD path as <alias>
[| ORDER BY $-.path] [| LIMIT <M>];

<vertex_id_list> ::= 
    [vertex_id [, vertex_id] ...]
```

- `SHORTEST`：查找最短路径。
- `ALL`：查找所有路径。
- `NOLOOP`：查找非循环路径。
- `WITH PROP`：展示点和边的属性。不添加本参数则隐藏属性。
- `<vertex_id_list>`：点 ID 列表。多个点用英文逗号 (,) 分隔。支持 \$- 和 \$var。
- `<edge_type_list>`：Edge type 列表。多个 Edge type 用英文逗号 (,) 分隔。* 表示所有 Edge type。
- `REVERSELY | BIDIRECT`：REVERSELY 表示反向，BIDIRECT 表示双向。
- `<WHERE clause>`：可以使用 WHERE 子句过滤边属性。
- `<N>`：路径的最大跳数。默认值为 5。
- `<M>`：指定返回的最大行数。



FIND PATH 语句检索的路径类型为 `trail`，即检索的路径只有点可以重复，边不可以重复。详情请参见[路径](#)。

限制

- 指定起始点和目的点的列表后，会返回起始点和目的点所有组合的路径。
- 搜索所有路径时可能会出现循环。
- 使用 WHERE 子句时只能过滤边属性，暂不支持过滤点属性，且不支持函数。
- graphd 是单进程查询，会占用很多内存。

示例

返回的路径格式类似于 `(<vertex_id>) -[:<edge_type_name>@<rank>]->(<vertex_id>)`。

```
nebula> FIND SHORTEST PATH FROM "player102" TO "team204" OVER * YIELD path AS p;
+-----+
| p   |
+-----+
```

```

| <("player102")-[:serve@0 {}]->("team204")> |
+-----+-----+-----+-----+-----+-----+-----+-----+
| p
+-----+-----+-----+-----+-----+-----+-----+-----+
| <"team204" :team{name: "spurs"}>-[:serve@0 {end_year: 1916, start_year: 1997}]-("player100" :player{age: 42, name: "Tim Duncan"})> |
+-----+-----+-----+-----+-----+-----+-----+-----+
nebula> FIND SHORTEST PATH WITH PROP FROM "team204" TO "player100" OVER * REVERSELY YIELD path AS p;
+-----+-----+-----+-----+-----+-----+-----+-----+
| p
+-----+-----+-----+-----+-----+-----+-----+-----+
| <("player100")-[:serve@0 {}]->("team204")>
| <("player100")-[:follow@0 {}]->("player125")-[:serve@0 {}]->("team204")>
| <("player100")-[:follow@0 {}]->("player101")-[:serve@0 {}]->("team204")>
| ...
+-----+-----+-----+-----+-----+-----+-----+-----+
nebula> FIND ALL PATH FROM "player100" TO "team204" OVER * WHERE follow.degree is EMPTY or follow.degree >=0 YIELD path AS p;
+-----+-----+-----+-----+-----+-----+-----+-----+
| p
+-----+-----+-----+-----+-----+-----+-----+-----+
| <("player100")-[:serve@0 {}]->("team204")>
| <("player100")-[:follow@0 {}]->("player125")-[:serve@0 {}]->("team204")>
| <("player100")-[:follow@0 {}]->("player101")-[:serve@0 {}]->("team204")>
| ...
+-----+-----+-----+-----+-----+-----+-----+-----+
nebula> FIND NOLOOP PATH FROM "player100" TO "team204" OVER * YIELD path AS p;
+-----+-----+-----+-----+-----+-----+-----+-----+
| p
+-----+-----+-----+-----+-----+-----+-----+-----+
| <("player100")-[:serve@0 {}]->("team204")>
| <("player100")-[:follow@0 {}]->("player125")-[:serve@0 {}]->("team204")>
| <("player100")-[:follow@0 {}]->("player101")-[:serve@0 {}]->("team204")>
| <("player100")-[:follow@0 {}]->("player101")-[:follow@0 {}]->("player125")-[:serve@0 {}]->("team204")>
| <("player100")-[:follow@0 {}]->("player101")-[:follow@0 {}]->("player102")-[:serve@0 {}]->("team204")>
| ...
+-----+-----+-----+-----+-----+-----+-----+-----+

```

FAQ

是否支持 **WHERE** 子句，以实现图遍历过程中的条件过滤？

支持使用 **WHERE** 子句过滤，但只能过滤边属性，不支持过滤点属性。

如示例中的 `WHERE follow.degree is EMPTY or follow.degree >= 0`。

最后更新: May 4, 2023

4.16 查询调优与终止

4.16.1 EXPLAIN 和 PROFILE

EXPLAIN 语句输出 nGQL 语句的执行计划，但不会执行 nGQL 语句。

PROFILE 语句执行 nGQL 语句，然后输出执行计划和执行概要。用户可以根据执行计划和执行概要优化查询性能。

执行计划

执行计划由NebulaGraph查询引擎中的执行计划器决定。

执行计划器将解析后的 nGQL 语句处理为 action 。action 是最小的执行单元。典型的 action 包括获取指定点的所有邻居、获取边的属性、根据条件过滤点或边等。每个 action 都被分配给一个 operator 。

例如 SHOW TAGS 语句分为两个 action ， operator 为 Start 和 ShowTags 。更复杂的 GO 语句可能会被处理成 10 个以上的 action 。

语法

- EXPLAIN

```
EXPLAIN [format= {"row" | "dot" | "tck"}] <your_nGQL_statement>;
```

- PROFILE

```
PROFILE [format= {"row" | "dot" | "tck"}] <your_nGQL_statement>;
```

输出格式

EXPLAIN 或 PROFILE 语句的输出有三种格式： row （默认） 、 dot 和 tck 。用户可以使用 format 选项修改输出格式。

row 格式

row 格式将返回信息输出到一个表格中。

• EXPLAIN

```
nebula> EXPLAIN format="row" SHOW TAGS;
Execution succeeded (time spent 327/892 us)

Execution Plan

-----+-----+-----+-----+
| id | name      | dependencies | profiling data | operator info
-----+-----+-----+-----+
| 1  | ShowTags_0 | 0          |               | outputVar: [{"colNames": [], "name": "\_ShowTags_1", "type": "DATASET"}] |
|     |           |             |               | inputVar:                                |
-----+-----+-----+-----+
| 0  | Start     |           |               | outputVar: [{"colNames": [], "name": "\_Start_0", "type": "DATASET"}] |
-----+-----+-----+-----+
```

• PROFILE

```
nebula> PROFILE format="row" SHOW TAGS;
+-----+
| Name   |
+-----+
| player |
| team   |
+-----+
Got 2 rows (time spent 2038/2728 us)

Execution Plan

-----+-----+-----+-----+
| id | name      | dependencies | profiling data | operator info
-----+-----+-----+-----+
| 1  | ShowTags_0 | 0          | ver: 0, rows: 1, execTime: 42us, totalTime: 1177us | outputVar: [{"colNames": [], "name": "\_ShowTags_1", "type": "DATASET"}] |
|     |           |             |               | inputVar:                                |
-----+-----+-----+-----+
| 0  | Start     |           | ver: 0, rows: 0, execTime: 1us, totalTime: 57us  | outputVar: [{"colNames": [], "name": "\_Start_0", "type": "DATASET"}] |
-----+-----+-----+-----+
```

参数

说明

id	operator 的 ID。
name	operator 的名称。
dependencies	当前 operator 所依赖的 operator 的 ID。
profiling data	执行概要文件内容。ver 表示 operator 的版本；rows 表示 operator 输出结果的行数；execTime 表示执行 action 的时间；totalTime 表示执行 action 的时间、系统调度时间、排队时间的总和。
operator info	operator 的详细信息。

dot 格式

dot 格式将返回 DOT 语言的信息，然后用户可以使用 Graphviz 生成计划图。

Note

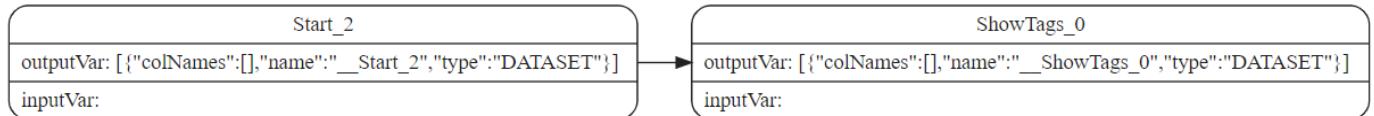
Graphviz 是一款开源可视化图工具，可以绘制 DOT 语言脚本描述的图。Graphviz 提供一个在线工具，可以预览 DOT 语言文件，并将它们导出为 SVG 或 JSON 等其他格式。详情请参见 [Graphviz Online](#)。

```
nebula> EXPLAIN format="dot" SHOW TAGS;
Execution succeeded (time spent 161/665 us)
Execution Plan

graph TD
    Start((Start)) --> ShowTags0[ShowTags_0]
    ShowTags0 --> Start2[Start_2]
    Start2 --> ShowTags0
```

```
}
```

将上述示例的 DOT 语言转换为 Graphviz 图，如下所示。



tck 格式

tck 格式类似于表格，但是没有边框和行之间的间的分割线。用户可以将结果用在单元测试的测试用例中。关于 tck 格式的测试用例，参见 [TCK case](#)。

- EXPLAIN

```
nebula> EXPLAIN format="tck" FETCH PROP ON player "player_1","player_2","player_3" YIELD properties(vertex).name as name, properties(vertex).age as age;
Execution succeeded (time spent 261μs/613.718μs)
```

Execution Plan (optimize time 28 us)

id	name	dependencies	profiling data	operator info
2	Project	1		
1	GetVertices	0		
0	Start			

Wed, 22 Mar 2023 23:15:52 CST

- PROFILE

```
nebula> PROFILE format="tck" FETCH PROP ON player "player_1","player_2","player_3" YIELD properties(vertex).name as name, properties(vertex).age as age;
```

name	age
"Piter Park"	24
"aaa"	24
"ccc"	24

Got 3 rows (time spent 1.474ms/2.19677ms)

Execution Plan (optimize time 41 us)

id	name	dependencies	profiling data	operator info
2	Project	1	{"rows":3,"version":0}	
1	GetVertices	0	{"resp[0]":{"exec":"232(us)","host":"127.0.0.1:9779","total":"758(us)"}, "rows":3, "total_rpc":"875(us)", "version":0}	
0	Start		{"rows":0,"version":0}	

Wed, 22 Mar 2023 23:16:13 CST

最后更新: May 4, 2023

4.16.2 终止查询

KILL QUERY 命令可以终止正在执行的查询，常用于终止慢查询。

Note

仅 God 角色权限可以终止任意查询，其他角色只能终止自己的查询。

语法

```
KILL QUERY (session=<session_id>, plan=<plan_id>);
```

- `session_id`：会话 ID。
- `plan_id`：执行计划 ID。

会话 ID 和执行计划 ID 可以唯一确定一个查询。二者可以通过 **SHOW QUERIES** 语句获取。

示例

在一个会话中执行命令终止另一个会话中的查询：

```
nebula> KILL QUERY(SESSION=1625553545984255,PLAN=163);
```

另一个会话中的查询会终止，并返回如下信息：

```
[ERROR (-1005)]: ExecutionPlanId[1001] does not exist in current Session.
```

最后更新: May 4, 2023

4.16.3 终止会话 (KILL SESSION)

`KILL SESSION` 语句用于关闭未登出的会话 (Session)。

Note

- 只有 `root` 用户可以终止会话。
- 执行 `KILL SESSION` 命令后，所有的 Graph 服务同步最新的会话信息需要等待 `2 * session_reclaim_interval_secs` 秒，默认等待 120 秒。

语法

`KILL SESSION` 语句支持终止单个和多个会话，语法如下：

- 终止单个会话

```
KILL {SESSION|SESSIONS} <SessionId>
```

- {SESSION|SESSIONS}：支持 `SESSION` 和 `SESSIONS` 的写法。
- <SessionId>：指会话 Session 的 ID。可执行 **SHOW SESSIONS** 命令查看会话 ID。

- 终止多个会话

```
SHOW SESSIONS
| YIELD $-.SessionId AS sid [WHERE <filter_clause>]
| KILL {SESSION|SESSIONS} $-.sid
```

Note

`KILL SESSION` 语句支持管道操作，即将 `SHOW SESSIONS` 语句与 `KILL SESSION` 语句结合使用，以终止多个会话。

- [`WHERE <filter_clause>`]：
- 可选项，使用 `WHERE` 子句过滤会话；`<filter_expression>` 指滤过表达式，例如 `WHERE $-.CreateTime < datetime("2022-12-14T18:00:00")`。如果不加改选项，则关闭所有当前会话。
- `WHERE` 子句中支持的过滤项有：`SessionId`、`UserName`、`SpaceName`、`CreateTime`、`UpdateTime`、`GraphAddr`、`Timezone`、`ClientIp`。可以执行 **SHOW SESSIONS** 命令查看这些过滤项的含义。
- {SESSION|SESSIONS}：支持 `SESSION` 和 `SESSIONS` 的写法。

Caution

请谨慎使用过滤条件以防误删会话。

示例

- 终止单个会话。

```
nebula> KILL SESSION 1672887983842984
```

- 终止多个会话。
- 终止创建时间小于 2023-01-05T18:00:00 的所有会话。

```
nebula> SHOW SESSIONS | YIELD $-.SessionId AS sid WHERE $-.CreateTime < datetime("2023-01-05T18:00:00") | KILL SESSIONS $-.sid
```

- 终止创建时间最早的两个会话。

```
nebula> SHOW SESSIONS | YIELD $-.SessionId AS sid, $-.CreateTime as CreateTime | ORDER BY $-.CreateTime ASC | LIMIT 2 | KILL SESSIONS $-.sid
```

- 终止用户名为 session_user1 创建的所有会话。

```
nebula> SHOW SESSIONS | YIELD $-.SessionId as sid WHERE $-.UserName == "session_user1" | KILL SESSIONS $-.sid
```

- 终止所有会话

```
nebula> SHOW SESSIONS | YIELD $-.SessionId as sid | KILL SESSION $-.sid
```

// 或者

```
nebula> SHOW SESSIONS | KILL SESSIONS $-.SessionId
```

Caution

终止所有会话时，当前会话也会被终止。请谨慎使用。

最后更新: May 4, 2023

4.17 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 9, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

5. 安装部署

5.1 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 19, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

5.2 编译与安装

5.2.1 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 87, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

5.2.2 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 3, in top-level template code
  File "docs-2.0/reuse/source_install-nebula-graph-by-rpm-or-deb.md", line 8, in top-level template code
    {{ ent.ent_begin }}
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

5.2.3 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 76, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

5.2.4 使用 Docker Compose 部署NebulaGraph

使用 Docker Compose 可以基于准备好的配置文件快速部署NebulaGraph服务，仅建议在测试NebulaGraph功能时使用该方式。

前提条件

- 主机上安装如下应用程序。

应用程序	推荐版本	官方安装参考
Docker	最新版本	Install Docker Engine
Docker Compose	最新版本	Install Docker Compose
Git	最新版本	Download Git

- 如果使用非 root 用户部署NebulaGraph，请授权该用户 Docker 相关的权限。详细信息，请参见 [Manage Docker as a non-root user](#)。
- 启动主机上的 Docker 服务。
- 如果已经通过 Docker Compose 在主机上部署了另一个版本的NebulaGraph，为避免兼容性问题，需要删除目录 `nebula-docker-compose/data`。

部署NebulaGraph

- 通过 Git 克隆 nebula-docker-compose 仓库的 3.4.0 分支到主机。

Danger

master 分支包含最新的未测试代码。请不要在生产环境使用此版本。

```
$ git clone -b release-3.4 https://github.com/vesoft-inc/nebula-docker-compose.git
```

Note

Docker Compose 的 x.y 版本对齐内核的 x.y 版本，对于内核 z 版本，Docker Compose 不会发布对应的 z 版本，但是会拉取 z 版本的内核镜像。

- 切换至目录 `nebula-docker-compose`。

```
$ cd nebula-docker-compose/
```

- 执行如下命令启动NebulaGraph服务。

Note

- 如果长期未内核更新镜像，请先更新 [NebulaGraph镜像](#)和 [NebulaGraph Console 镜像](#)。
- 执行命令后的返回结果因安装目录不同而不同。

```
[nebula-docker-compose]$ docker-compose up -d
Creating nebuladockercompose_metad0_1 ... done
Creating nebuladockercompose_metad2_1 ... done
Creating nebuladockercompose_metad1_1 ... done
Creating nebuladockercompose_graphd2_1 ... done
Creating nebuladockercompose_graphd_1 ... done
Creating nebuladockercompose_graphd1_1 ... done
Creating nebuladockercompose_storaged0_1 ... done
Creating nebuladockercompose_storaged2_1 ... done
Creating nebuladockercompose_storaged1_1 ... done
```

↑ Compatibility

从 3.1 版本开始，Docker-compose 会自动启动 NebulaGraph Console 镜像的容器，并将 Storage 主机增加至集群中（即 ADD HOSTS 命令）。

Note

上述服务的更多信息，请参见[架构总览](#)。

连接NebulaGraph

连接NebulaGraph有两种方式：

- 在容器外通过 Nebula Console 连接。因为容器的配置文件中将 Graph 服务的外部映射端口也固定为 9669，因此可以直接通过默认端口连接。
详情参见[连接NebulaGraph](#)。
- 登录安装了 NebulaGraph Console 的容器，然后再连接 Graph 服务。本小节介绍这种方式。

1. 使用 docker-compose ps 命令查看 NebulaGraph Console 容器名称。

```
$ docker-compose ps
      Name           Command     State    Ports
-----+-----+-----+-----+
nebuladockercompose_console_1   sh -c sleep 3 &&      Up
                                nebula-co ...
.....
```

2. 进入 NebulaGraph Console 容器中。

```
$ docker exec -it nebuladockercompose_console_1 /bin/sh
/ #
```

3. 通过 NebulaGraph Console 连接NebulaGraph。

```
/ # ./usr/local/bin/nebula-console -u <user_name> -p <password> --address=graphd --port=9669
```

Note

默认情况下，身份认证功能是关闭的，只能使用已存在的用户名（默认为 root）和任意密码登录。如果想使用身份认证，请参见[身份认证](#)。

4. 查看集群状态。

```
nebula> SHOW HOSTS;
+-----+-----+-----+-----+-----+-----+
| Host | Port | Status | Leader count | Leader distribution | Partition distribution | Version |
+-----+-----+-----+-----+-----+-----+
| "storaged0" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" | "3.4.1" |
| "storaged1" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" | "3.4.1" |
| "storaged2" | 9779 | "ONLINE" | 0 | "No valid partition" | "No valid partition" | "3.4.1" |
+-----+-----+-----+-----+-----+-----+
```

执行两次 exit 可以退出容器。

查看NebulaGraph服务的状态和端口

执行命令 docker-compose ps 可以列出NebulaGraph服务的状态和端口。

Note

NebulaGraph默认使用 9669 端口为客户端提供服务，如果需要修改端口，请修改目录 nebula-docker-compose 内的文件 docker-compose.yaml，然后重启 NebulaGraph 服务。

```
$ docker-compose ps
nebuladockercompose_console_1    sh -c sleep 3 &&      Up
                                 nebula-co ...
nebuladockercompose_graphd1_1   /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49174->19669/tcp,:::49174->19669/tcp, 0.0.0.0:49171->19670/tcp,:::49171->19670/tcp, 0.0.0.0:49177->9669/
tcp,:::49177->9669/tcp
nebuladockercompose_graphd2_1   /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49175->19669/tcp,:::49175->19669/tcp, 0.0.0.0:49172->19670/tcp,:::49172->19670/tcp, 0.0.0.0:49178->9669/
tcp,:::49178->9669/tcp
nebuladockercompose_graphd_1    /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49180->19669/tcp,:::49180->19669/tcp, 0.0.0.0:49179->19670/tcp,:::49179->19670/tcp, 0.0.0.0:9669->9669/
tcp,:::9669->9669/tcp
nebuladockercompose_metad0_1    /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49157->19559/tcp,:::49157->19559/tcp, 0.0.0.0:49154->19560/tcp,:::49154->19560/tcp, 0.0.0.0:49160->9559/
tcp,:::49160->9559/tcp, 9560/tcp
nebuladockercompose_metad1_1    /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49156->19559/tcp,:::49156->19559/tcp, 0.0.0.0:49153->19560/tcp,:::49153->19560/tcp, 0.0.0.0:49159->9559/
tcp,:::49159->9559/tcp, 9560/tcp
nebuladockercompose_metad2_1    /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49158->19559/tcp,:::49158->19559/tcp, 0.0.0.0:49155->19560/tcp,:::49155->19560/tcp, 0.0.0.0:49161->9559/
tcp,:::49161->9559/tcp, 9560/tcp
nebuladockercompose_storaged0_1 /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49166->19779/tcp,:::49166->19779/tcp, 0.0.0.0:49163->19780/tcp,:::49163->19780/tcp, 9777/tcp, 9778/tcp, 0.
0.0.0:49169->9779/tcp,:::49169->9779/tcp, 9780/tcp
nebuladockercompose_storaged1_1 /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49165->19779/tcp,:::49165->19779/tcp, 0.0.0.0:49162->19780/tcp,:::49162->19780/tcp, 9777/tcp, 9778/tcp, 0.
0.0.0:49168->9779/tcp,:::49168->9779/tcp, 9780/tcp
nebuladockercompose_storaged2_1 /usr/local/nebula/bin/nebu ... Up   0.0.0.0:49167->19779/tcp,:::49167->19779/tcp, 0.0.0.0:49164->19780/tcp,:::49164->19780/tcp, 9777/tcp, 9778/tcp, 0.
0.0.0:49170->9779/tcp,:::49170->9779/tcp, 9780/tcp
```

如果服务有异常，用户可以先确认异常的容器名称（例如 nebuladockercompose_graphd2_1），

然后执行 docker ps 查看对应的 CONTAINER ID（示例为 2a6c56c405f5）。

```
[nebula-docker-compose]$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS NAMES
PORTS
2a6c56c405f vesoft/nebula-graphd:nightly "/usr/local/nebula/b..." 36 minutes ago Up 36 minutes (healthy) 0.0.0.0:49230->9669/tcp, 0.0.0.0:49229->19669/tcp, 0.0.0.0:49228->19670/
tcp,nebuladockercompose_graphd2_1
7042e0a8e83d vesoft/nebula-storaged:nightly "./bin/nebula-storag..." 36 minutes ago Up 36 minutes (healthy) 9777-9778/tcp, 9780/tcp, 0.0.0.0:49227->9779/tcp, 0.0.0.0:49226->19779/
tcp, 0.0.0.0:49225->19780/tcp nebuladockercompose_storaged2_1
18e3ea63ad65 vesoft/nebula-storaged:nightly "./bin/nebula-storag..." 36 minutes ago Up 36 minutes (healthy) 9777-9778/tcp, 9780/tcp, 0.0.0.0:49219->9779/tcp, 0.0.0.0:49218->19779/
tcp, 0.0.0.0:49217->19780/tcp nebuladockercompose_storaged0_1
4dcabfe8677a vesoft/nebula-graphd:nightly "/usr/local/nebula/b..." 36 minutes ago Up 36 minutes (healthy) 0.0.0.0:49224->9669/tcp, 0.0.0.0:49223->19669/tcp, 0.0.0.0:49222->19670/
tcp,nebuladockercompose_graphd1_1
a74054c6ae25 vesoft/nebula-graphd:nightly "/usr/local/nebula/b..." 36 minutes ago Up 36 minutes (healthy) 0.0.0.0:9669->9669/tcp, 0.0.0.0:49221->19669/tcp, 0.0.0.0:49220->19670/
tcp,nebuladockercompose_graphd_1
880025a3858c vesoft/nebula-storaged:nightly "./bin/nebula-storag..." 36 minutes ago Up 36 minutes (healthy) 9777-9778/tcp, 9780/tcp, 0.0.0.0:49216->9779/tcp, 0.0.0.0:49215->19779/
tcp, 0.0.0.0:49214->19780/tcp nebuladockercompose_storaged1_1
45736a32a23a vesoft/nebula-metad:nightly "./bin/nebula-metad ..." 36 minutes ago Up 36 minutes (healthy) 9560/tcp, 0.0.0.0:49213->9559/tcp, 0.0.0.0:49212->19559/tcp, 0.
0.0.0:49211->19560/tcp nebuladockercompose_metad0_1
3b2c90eb073e vesoft/nebula-metad:nightly "./bin/nebula-metad ..." 36 minutes ago Up 36 minutes (healthy) 9560/tcp, 0.0.0.0:49207->9559/tcp, 0.0.0.0:49206->19559/tcp, 0.
0.0.0:49205->19560/tcp nebuladockercompose_metad2_1
7bb31b7a5b3f vesoft/nebula-metad:nightly "./bin/nebula-metad ..." 36 minutes ago Up 36 minutes (healthy) 9560/tcp, 0.0.0.0:49210->9559/tcp, 0.0.0.0:49209->19559/tcp, 0.
0.0.0:49208->19560/tcp nebuladockercompose_metad1_1
```

最后登录容器排查问题

```
[nebula-docker-compose]$ docker exec -it 2a6c56c405f bash
[root@2a6c56c405f nebula]#
```

查看NebulaGraph服务的数据和日志

NebulaGraph的所有数据和日志都持久化存储在 nebula-docker-compose/data 和 nebula-docker-compose/logs 目录中。

目录的结构如下：

```
nebula-docker-compose/
|-- docker-compose.yaml
|   |-- data
|   |   |-- meta0
|   |   |-- meta1
|   |   |-- meta2
|   |   |-- storage0
|   |   |-- storage1
|   |   |-- storage2
|   |-- logs
|       |-- graph
|       |-- graph1
|       |-- graph2
|       |-- meta0
|       |-- meta1
|       |-- meta2
```

```
└── storage0
    └── storage1
        └── storage2
```

停止NebulaGraph服务

用户可以执行如下命令停止NebulaGraph服务:

```
$ docker-compose down
```

如果返回如下信息，表示已经成功停止服务。

```
Stopping nebuladockercompose_console_1 ... done
Stopping nebuladockercompose_graphd1_1 ... done
Stopping nebuladockercompose_graphd_1 ... done
Stopping nebuladockercompose_graphd2_1 ... done
Stopping nebuladockercompose_storaged1_1 ... done
Stopping nebuladockercompose_storaged0_1 ... done
Stopping nebuladockercompose_storaged2_1 ... done
Stopping nebuladockercompose_metad2_1 ... done
Stopping nebuladockercompose_metad0_1 ... done
Stopping nebuladockercompose_metad1_1 ... done
Removing nebuladockercompose_console_1 ... done
Removing nebuladockercompose_graphd1_1 ... done
Removing nebuladockercompose_graphd_1 ... done
Removing nebuladockercompose_graphd2_1 ... done
Removing nebuladockercompose_storaged1_1 ... done
Removing nebuladockercompose_storaged0_1 ... done
Removing nebuladockercompose_storaged2_1 ... done
Removing nebuladockercompose_metad2_1 ... done
Removing nebuladockercompose_metad0_1 ... done
Removing nebuladockercompose_metad1_1 ... done
Removing network nebuladockercompose_nebula-net
```



命令 `docker-compose down -v` 的参数 `-v` 将会删除所有本地的数据。如果使用的是 `nightly` 版本，并且有一些兼容性问题，请尝试这个命令。

修改配置

Docker Compose 部署的NebulaGraph，配置文件位置为 `nebula-docker-compose/docker-compose.yaml`，修改该文件内的配置并重启服务即可使新配置生效。

具体的配置说明请参见[配置管理](#)。

常见问题

如何固定 `DOCKER` 映射到外部的端口？

在目录 `nebula-docker-compose` 内修改文件 `docker-compose.yaml`，将对应服务的 `ports` 设置为固定映射，例如:

```
graphd:
  image: vesoft/nebula-graphd:release-3.4
  ...
  ports:
    - 9669:9669
    - 19669
    - 19670
```

`9669:9669` 表示内部的 `9669` 映射到外部的端口也是 `9669`，下方的 `19669` 表示内部的 `19669` 映射到外部的端口是随机的。

如何升级/更新NEBULAGRAPH服务的 `DOCKER` 镜像？

1. 在文件 `nebula-docker-compose/docker-compose.yaml` 中，找到所有服务的 `image` 并修改其值为相应的镜像版本。
2. 在目录 `nebula-docker-compose` 内执行命令 `docker-compose pull`，更新 Graph 服务、Storage 服务、Meta 服务和 NebulaGraph Console 的镜像。
3. 执行命令 `docker-compose up -d` 启动NebulaGraph服务。
4. 通过 NebulaGraph Console 连接NebulaGraph后，分别执行命令 `SHOW HOSTS GRAPH`、`SHOW HOSTS STORAGE`、`SHOW HOSTS META` 查看各服务版本。

执行命令 `DOCKER-COMPOSE PULL` 报错 `ERROR: TOOMANYREQUESTS`

可能遇到如下错误:

```
ERROR: toomanyrequests: You have reached your pull rate limit. You may increase the limit by authenticating and upgrading: https://www.docker.com/increase-rate-limit
```

以上错误表示已达到 Docker Hub 的速率限制。解决方案请参见 [Understanding Docker Hub Rate Limiting](#)。

如何更新 NEBULAGRAPH CONSOLE?

`docker-compose pull` 会同时更新NebulaGraph服务和 NebulaGraph Console。

最后更新: May 4, 2023

5.2.5 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 34, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

5.2.6 Macro Syntax Error

*Line 6 in Markdown file: **unexpected '}'***

```
- {{dashboard_ent.name}}
```

最后更新: May 4, 2023

5.3 存算合并版NebulaGraph

存算合并版NebulaGraph将存储服务（Meta 和 Storage）和计算服务（Graph）合并至一个进程，用于部署在单台机器上。本文介绍存算合并版NebulaGraph的使用场景、安装步骤等。



存算合并版NebulaGraph不用于生产环境。

5.3.1 背景信息

传统的NebulaGraph架构由 3 个服务构成，每个服务都有可执行的二进制文件和对应的进程，进程之间通过 RPC 协议进行调用。而在存算合并版NebulaGraph中，NebulaGraph中 3 个服务对应的 3 个进程被合为 1 个进程。

关于NebulaGraph的更多信息，参见[架构总览](#)。

5.3.2 使用场景

数据规模小，可用性需求不大的场景。例如，受限于机器数量的测试环境或者仅用于验证功能的场景。

5.3.3 使用限制

- 仅支持单副本服务。
- 不支持高可用和可靠性。

5.3.4 环境准备

关于安装存算合并版NebulaGraph所需的环境，参见[编译NebulaGraph源码要求](#)。

5.3.5 安装步骤

目前仅支持使用源码安装存算合并版NebulaGraph。其安装步骤与多进程的NebulaGraph步骤类似，用户只需在使用 **CMake** 生成 **makefile** 文件步骤的命令中添加 `-DENABLE_STANDALONE_VERSION=on`。示例如下：

```
cmake -DCMAKE_INSTALL_PREFIX=/usr/local/nebula -DENABLE_TESTING=OFF -DENABLE_STANDALONE_VERSION=on -DCMAKE_BUILD_TYPE=Release ..
```

有关具体的安装步骤，参见[使用源码安装](#)。

用户完成存算合并版NebulaGraph后，可以参见[连接服务](#)连接NebulaGraph。

5.3.6 配置文件

存算合并版NebulaGraph的配置文件的路径默认为 `/usr/local/nebula/etc`。

用户可执行 `sudo cat nebula-standalone.conf.default` 查看配置文件内容。配置文件参数和描述和多进程的NebulaGraph大体一致，除以下参数外：

参数	预设值	说明
<code>meta_port</code>	9559	Meta 服务的端口号。
<code>storage_port</code>	9779	Storage 服务的端口号。
<code>meta_data_path</code>	<code>data/meta</code>	Meta 数据存储路径。

用户可以执行命令查看配置项列表与说明。具体操作，请参见[配置管理](#)。

最后更新: May 4, 2023

5.4 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 3, in top-level template code
  File "docs-2.0/reuse/source_manage-service.md", line 3, in top-level template code
    {{ ent.ent_begin }}
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

5.5 连接NebulaGraph服务

本文介绍如何使用原生命令行客户端 Nebula Console 连接 NebulaGraph。



首次连接到NebulaGraph后，必须先[注册 Storage 服务](#)，才能正常查询数据。

NebulaGraph支持多种类型的客户端，包括命令行客户端、可视化界面客户端和流行编程语言客户端。详情参见[客户端列表](#)。

5.5.1 前提条件

- NebulaGraph服务已启动。
- 运行 Nebula Console 的机器和运行NebulaGraph的服务器网络互通。
- Nebula Console 的版本兼容NebulaGraph的版本。



版本相同的 Nebula Console 和NebulaGraph兼容程度最高，版本不同的 Nebula Console 连接NebulaGraph时，可能会有兼容问题，或者无法连接并报错 `incompatible version between client and server`。

5.5.2 操作步骤

1. 在 Nebula Console [下载页面](#)，确认需要的版本，单击 **Assets**。



建议选择最新版本。

2. 在 **Assets** 区域找到机器运行所需的二进制文件，下载文件到机器上。
3. (可选) 为方便使用，重命名文件为 `nebula-console`。



在 Windows 系统中，请重命名为 `nebula-console.exe`。

4. 在运行 Nebula Console 的机器上执行如下命令，为用户授予 `nebula-console` 文件的执行权限。



Windows 系统请跳过此步骤。

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

5. 在命令行界面中，切换工作目录至 `nebula-console` 文件所在目录。

6. 执行如下命令连接NebulaGraph。

- Linux 或 macOS

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

- Windows

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

参数说明如下。

参数	说明
-h/-help	显示帮助菜单。
-addr/-address	设置要连接的 Graph 服务的 IP 地址。默认地址为 127.0.0.1。
-P/-port	设置要连接的 Graph 服务的端口。默认端口为 9669。
-u/-user	设置NebulaGraph账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 root）。
-p/-password	设置用户名对应的密码。未启用身份认证时，密码可以填写任意字符。
-t/-timeout	设置整数类型的连接超时时间。单位为毫秒， 默认值为 120。
-e/-eval	设置字符串类型的 nGQL 语句。连接成功后会执行一次该语句并返回结果，然后自动断开连接。
-f/-file	设置存储 nGQL 语句的文件的路径。连接成功后会执行该文件内的 nGQL 语句并返回结果，执行完毕后自动断开连接。
-enable_ssl	连接NebulaGraph时使用 SSL 加密。
-ssl_root_ca_path	指定 CA 证书的存储路径。
-ssl_cert_path	指定 CRT 证书的存储路径。
-ssl_private_key_path	指定私钥文件的存储路径。

更多参数参见[项目仓库](#)。

最后更新: May 4, 2023

5.6 管理 Storage 主机

从 3.0.0 版本开始，在配置文件中添加的 Storage 主机无法直接读写，配置文件的作用仅仅是将 Storage 主机注册至 Meta 服务中。必须使用 ADD HOSTS 命令后，才能正常读写 Storage 主机。

5.6.1 增加 Storage 主机

向集群中增加 Storage 主机。

```
ADD HOSTS <ip>:<port> [<ip>:<port> ...];  
ADD HOSTS "<hostname>":<port> ["<hostname>":<port> ...];
```



- 增加 Storage 主机在下一个心跳周期之后才能生效，为确保数据同步，请等待 2 个心跳周期（20 秒），然后执行 SHOW HOSTS 查看是否在线。
- IP地址和端口请和配置文件中的设置保持一致，例如单机部署的默认为 127.0.0.1:9779。
- 使用域名时，需要用引号包裹，例如 ADD HOSTS "foo-bar":9779。
- 确保新增的 Storage 主机没有被其他集群使用过，否则会导致添加 Storage 节点失败。

5.6.2 删除 Storage 主机

从集群中删除 Storage 主机。



无法直接删除正在使用的 Storage 主机，需要先删除关联的图空间，才能删除 Storage 主机。

```
DROP HOSTS <ip>:<port> [<ip>:<port> ...];  
DROP HOSTS "<hostname>":<port> ["<hostname>":<port> ...];
```

最后更新: May 4, 2023

5.7 卸载NebulaGraph

本文介绍如何卸载NebulaGraph。



如果需要重新部署NebulaGraph，请务必完全卸载后再重新部署，否则可能会出现问题，包括 Meta 不一致等。

5.7.1 前提条件

停止NebulaGraph服务。详情参见[管理NebulaGraph服务](#)。

5.7.2 步骤 1: 删除数据和元数据文件

如果在配置文件内修改了数据文件的路径，可能会导致安装路径和数据文件保存路径不一致，因此需要查看配置文件，确认数据文件保存路径，然后手动删除数据文件目录。



如果是集群架构，需要删除所有 Storage 和 Meta 服务节点的数据文件。

1. 检查 Storage 服务的 `disk` 配置。例如：

```
##### Disk #####
# Root data path. Split by comma. e.g. --data_path=/disk1/path1/,/disk2/path2/
# One path per Rocksdb instance.
--data_path=/nebula/data/storage
```

2. 检查 `metad` 服务的配置文件，找到对应元数据目录。

3. 删除以上数据和元数据目录。

5.7.3 步骤 2: 卸载安装目录



删除整个安装目录，包括 `cluster.id` 文件。

安装路径为参数 `--prefix` 指定的路径。默认路径为 `/usr/local/nebula`。

卸载编译安装的NebulaGraph

找到NebulaGraph的安装目录，删除整个安装目录。

卸载 RPM 包安装的NebulaGraph

1. 使用如下命令查看NebulaGraph版本。

```
$ rpm -qa | grep "nebula"
```

返回类似如下结果。

```
nebula-graph-3.4.1-1.x86_64
```

2. 使用如下命令卸载NebulaGraph。

```
sudo rpm -e <nebula_version>
```

例如:

```
sudo rpm -e nebula-graph-3.4.1-1.x86_64
```

3. 删除安装目录。

卸载 DEB 包安装的NebulaGraph

1. 使用如下命令查看NebulaGraph版本。

```
$ dpkg -l | grep "nebula"
```

返回类似如下结果。

```
ii  nebula-graph  3.4.1  amd64      NebulaGraph Package built using CMake
```

2. 使用如下命令卸载NebulaGraph。

```
sudo dpkg -r <nebula_version>
```

例如:

```
sudo dpkg -r nebula-graph
```

3. 删除安装目录。

卸载 Docker Compose 部署的NebulaGraph

1. 在目录 nebula-docker-compose 内执行如下命令停止NebulaGraph服务。

```
docker-compose down -v
```

2. 删除目录 nebula-docker-compose。

最后更新: May 4, 2023

6. 配置与日志

6.1 配置

6.1.1 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
File "<template>", line 5, in top-level template code
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

6.1.2 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 32, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

6.1.3 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 80, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

6.1.4 Macro Rendering Error

UndefinedError: 'comm' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 83, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'comm' is undefined
```

最后更新: May 4, 2023

6.1.5 Linux 内核配置

本文介绍与NebulaGraph相关的 Linux 内核配置，并介绍如何修改配置。

资源控制

ULIMIT 注意事项

命令 `ulimit` 用于为当前 shell 会话设置资源阈值，注意事项如下：

- `ulimit` 所做的更改仅对当前会话或子进程生效。
- 资源的阈值（软阈值）不能超过硬阈值。
- 普通用户不能使用命令调整硬阈值，即使使用 `sudo` 也不能调整。
- 修改系统级别或调整硬性阈值，请编辑文件 `/etc/security/limits.conf`。这种方式需要重新登录才生效。

ULIMIT -C

`ulimit -c` 用于限制 `core` 文件的大小，建议设置为 `unlimited`，命令如下：

```
ulimit -c unlimited
```

ULIMIT -N

`ulimit -n` 用于限制打开文件的数量，建议设置为超过 10 万，例如：

```
ulimit -n 130000
```

内存

VM.SWAPPINESS

`vm.swappiness` 是触发虚拟内存（swap）的空闲内存百分比。值越大，使用 swap 的可能性就越大，建议设置为 0，表示首先删除页缓存。需要注意的是，0 表示尽量不使用 swap。

VM.MIN_FREE_KBYTES

`vm.min_free_kbytes` 用于设置 Linux 内核保留的最小空闲千字节数。如果系统内存足够，建议设置较大值。例如物理内存为 128 GB，可以将 `vm.min_free_kbytes` 设置为 5 GB。如果值太小，会导致系统无法申请足够的连续物理内存。

VM.MAX_MAP_COUNT

`vm.max_map_count` 用于限制单个进程的 VMA（虚拟内存区域）数量。默认值为 65530，对于绝大多数应用程序来说已经足够。如果应用程序因为内存消耗过大而报错，请增大本参数的值。

VM.DIRTY_*

`vm.dirty_*` 是一系列控制系统脏数据缓存的参数。对于写密集型场景，用户可以根据需要进行调整（吞吐量优先或延迟优先），建议使用系统默认值。

TRANSPARENT HUGE PAGE

为了降低延迟，用户必须关闭 THP（transparent huge page）。命令如下：

```
root# echo never > /sys/kernel/mm/transparent_hugepage/enabled
root# echo never > /sys/kernel/mm/transparent_hugepage/defrag
root# swapoff -a && swapon -a
```

为了防止系统重启后该配置失效，可以在GRUB配置文件或 `/etc/rc.local` 中添加相关配置，使系统启动时自动关闭 THP。

网络

NET.IPV4.TCP_SLOW_START_AFTER_IDLE

`net.ipv4.tcp_slow_start_after_idle` 默认值为 1，会导致闲置一段时间后拥塞窗口超时，建议设置为 0，尤其适合大带宽高延迟场景。

NET.CORE.SOMAXCONN

`net.core.somaxconn` 用于限制 `socket` 监听的连接队列数量。默认值为 128。对于有大量突发连接的场景，建议设置为不低于 1024。

NET.IPV4.TCP_MAX_SYN_BACKLOG

`net.ipv4.tcp_max_syn_backlog` 用于限制处于 `SYN_RECV` (半连接) 状态的 TCP 连接数量。默认值为 128。对于有大量突发连接的场景，建议设置为不低于 1024。

NET.CORE.NETDEV_MAX_BACKLOG

`net.core.netdev_max_backlog` 用于限制队列中数据包的数量。默认值为 1000，建议设置为 10000 以上，尤其是万兆网卡。

NET.IPV4.TCP_KEEPALIVE_*

`net.ipv4.tcp_keepalive_*` 是一系列保持 TCP 连接存活的参数。对于使用四层透明负载均衡的应用程序，如果空闲连接异常断开，请增大 `tcp_keepalive_time` 和 `tcp_keepalive_intvl` 的值。

NET.IPV4.TCP_WMEM/RMEM

TCP 套接字发送/接收缓冲池的最小、最大、默认空间。对于大连接，建议设置为 带宽 (GB) *往返时延 (ms)。

SCHEDULER

对于 SSD 设备，建议将 `scheduler` 设置为 `noop` 或者 `none`，路径为 `/sys/block/DEV_NAME/queue/scheduler`。

其他参数

KERNEL.CORE_PATTERN

建议设置为 `core`，并且将 `kernel.core_uses_pid` 设置为 1。

修改参数

SYSCTL 命令

- `sysctl <conf_name>`

查看当前参数值。

- `sysctl -w <conf_name>=<value>`

临时修改参数值，立即生效，重启后恢复原值。

- `sysctl -p [<file_path>]`

从指定配置文件里加载 Linux 系统参数，默认从 `/etc/sysctl.conf` 加载。

PRLIMIT

命令 `prlimit` 可以获取和设置进程资源的限制，结合 `sudo` 可以修改硬阈值，例如，`prlimit --nofile=140000 --pid=$$` 调整当前进程允许的打开文件的最大数量为 140000，立即生效，此命令仅支持 RedHat 7u 或更高版本。

最后更新: May 4, 2023

6.2 日志

6.2.1 运行日志

运行日志通常提供给 DBA 或开发人员查看，当系统出现故障，DBA 或开发人员可以根据运行日志定位问题。

NebulaGraph默认使用 `glog` 打印运行日志，使用 `gflags` 控制日志级别，并在运行时通过 HTTP 接口动态修改日志级别，方便跟踪问题。

运行日志目录

运行日志的默认目录为 `/usr/local/nebula/logs/`。

如果在NebulaGraph运行过程中删除运行日志目录，日志不会继续打印，但是不会影响业务。重启服务可以恢复正常。

配置说明

- `minLogLevel`：最小日志级别，即不会记录低于这个级别的日志。可选值为 0（INFO）、1（WARNING）、2（ERROR）、3（FATAL）。建议在调试时设置为 0，生产环境中设置为 1。如果设置为 4，NebulaGraph不会记录任何日志。
- `v`：日志详细级别，值越大，日志记录越详细。可选值为 0、1、2、3。

Meta 服务、Graph 服务和 Storage 服务的日志级别可以在各自的配置文件中查看，默认路径为 `/usr/local/nebula/etc/`。

查看运行日志级别

使用如下命令查看当前所有的 `gflags` 参数（包括日志参数）：

```
$ curl <ws_ip>:<ws_port>/flags
```

参数	说明
<code>ws_ip</code>	HTTP 服务的 IP 地址，可以在配置文件中查看。默认值为 <code>127.0.0.1</code> 。
<code>ws_port</code>	HTTP 服务的端口，可以在配置文件中查看。默认值分别为 <code>19559</code> （Meta）、 <code>19669</code> （Graph） <code>19779</code> （Storage）。

示例如下：

- 查看 Meta 服务当前的最小日志级别：

```
$ curl 127.0.0.1:19559/flags | grep 'minLogLevel'
```

- 查看 Storage 服务当前的日志详细级别：

```
$ curl 127.0.0.1:19779/flags | grep -w 'v'
```

修改运行日志级别

使用如下命令修改运行日志级别：

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"<key>:<value>[,<key>:<value>]}' "<ws_ip>:<ws_port>/flags"
```

参数	说明
key	待修改的运行日志类型，可选值请参见 配置说明 。
value	运行日志级别，可选值请参见 配置说明 。
ws_ip	HTTP 服务的 IP 地址，可以在配置文件中查看。默认值为 127.0.0.1。
ws_port	HTTP 服务的端口，可以在配置文件中查看。默认值分别为 19559（Meta）、19669（Graph）19779（Storage）。

示例如下：

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"minLogLevel":0,"v":3}' "127.0.0.1:19779	flags" # storaged
$ curl -X PUT -H "Content-Type: application/json" -d '{"minLogLevel":0,"v":3}' "127.0.0.1:19669	flags" # graphd
$ curl -X PUT -H "Content-Type: application/json" -d '{"minLogLevel":0,"v":3}' "127.0.0.1:19559	flags" # metad
```

如果在NebulaGraph运行时修改了运行日志级别，重启服务后会恢复为配置文件中设置的级别，如果需要永久修改，请修改[配置文件](#)。

RocksDB 运行日志

RocksDB 的运行日志通常在 `/usr/local/nebula/data/storage/nebula/$id/data/LOG`，其中 `$id` 为实例号。该日志通常用于调试 RocksDB 参数。

最后更新: May 4, 2023

7. 监控

7.1 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 109, in top-level template code
  File "docs-2.0/reuse/source-monitoring-metrics.md", line 154, in top-level template code
    {{ent.ent_begin}}
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

7.2 RocksDB 统计数据

NebulaGraph 使用 RocksDB 作为底层存储，本文介绍如何收集和展示 NebulaGraph 的 RocksDB 统计信息。

7.2.1 启用 RocksDB

RocksDB 统计功能默认关闭，启动 RocksDB 统计功能，你需要：

1. 修改 `nebula-storaged.conf` 文件中 `--enable_rocksdb_statistics` 参数为 `true`。配置默认文件目录为 `/use/local/nebula/etc`。
2. 重启服务使修改生效。

7.2.2 获取 RocksDB 统计信息

用户可以使用存储服务中的内置 HTTP 服务来获取以下类型的统计信息，且支持返回 JSON 格式的结果：

- 所有统计信息。
- 指定条目的信息。

7.2.3 示例

使用以下命令获取所有 RocksDB 统计信息：

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats"
```

例如：

```
curl -L "http://172.28.2.1:19779/rocksdb_stats"
rocksdb.blobdb.blob.file.bytes.read=0
rocksdb.blobdb.blob.file.bytes.written=0
rocksdb.blobdb.blob.file.bytes.synced=0
...
```

使用以下命令获取部分 RocksDB 统计信息：

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats?stats=${stats_name}"
```

例如使用以下语句获取 `rocksdb.bytes.read` 和 `rocksdb.block.cache.add` 的信息。

```
curl -L "http://172.28.2.1:19779/rocksdb_stats?stats=rocksdb.bytes.read,rocksdb.block.cache.add"
rocksdb.block.cache.add=14
rocksdb.bytes.read=1632
```

使用以下命令获取部分 JSON 格式的 RocksDB 统计信息：

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats?stats=${stats_name}&format=json"
```

例如使用以下语句获取 `rocksdb.bytes.read` 和 `rocksdb.block.cache.add` 的统计信息并返回 JSON 的格式数据。

```
curl -L "http://172.28.2.1:19779/rocksdb_stats?stats=rocksdb.bytes.read,rocksdb.block.cache.add&format=json"
[
  {
    "rocksdb.block.cache.add": 1
  },
  {
    "rocksdb.bytes.read": 160
  }
]
```

8. 数据安全

8.1 验证和授权

8.1.1 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
File "<template>", line 32, in top-level template code
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

8.1.2 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 27, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

8.1.3 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 50, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

8.2 SSL 加密

NebulaGraph 支持在客户端、Graph 服务、Meta 服务和 Storage 服务之间进行 SSL 加密传输，本文介绍如何设置 SSL 加密。

8.2.1 注意事项

开启 SSL 加密会轻微影响性能。

8.2.2 参数介绍

参数	默认值	说明
<code>cert_path</code>	-	PEM 证书的路径。
<code>key_path</code>	-	密钥证书的路径。
<code>password_path</code>	-	密码文件证书的路径。
<code>ca_path</code>	-	受信任 CA 证书文件的路径。
<code>enable_ssl</code>	<code>false</code>	是否开启 SSL 加密。
<code>enable_graph_ssl</code>	<code>false</code>	是否仅在 Graph 服务上开启 SSL 加密。
<code>enable_meta_ssl</code>	<code>false</code>	是否仅在 Meta 服务上开启 SSL 加密。

8.2.3 证书模式

为了使用 SSL 加密，必须有 SSL 证书。NebulaGraph 支持两种证书模式：

- 自签名证书模式

需要自行制作签名证书。需要根据加密策略，在对应的配置文件内设置 `cert_path`、`key_path` 和 `password_path`。

- CA 签名证书模式

需要在认证机构（Certificate Authority）申请签名证书。需要根据加密策略，在对应的配置文件内设置 `cert_path`、`key_path` 和 `ca_path`。

8.2.4 加密策略

NebulaGraph 支持三种加密策略。加密涉及的具体进程请参见[详细说明](#)。

- 对客户端、Graph 服务、Meta 服务和 Storage 服务之间的传输数据加密。

需要修改 `nebula-graphd.conf`、`nebula-metad.conf` 和 `nebula-storaged.conf` 配置文件，设置 `enable_ssl = true`。

- 对客户端和 Graph 服务之间的传输数据加密。

适用于集群设置在同一个机房内，仅对外开放 Graph 服务的端口。因为其他服务可以通过内部网络通信，无需加密。需要修改 `nebula-graphd.conf` 配置文件，设置 `enable_graph_ssl = true`。

- 对集群中 Meta 服务相关的传输数据加密。

适用于向 meta 服务传输需保密的信息。需要修改 `nebula-graphd.conf`、`nebula-metad.conf` 和 `nebula-storaged.conf` 配置文件，设置 `enable_meta_ssl = true`。

8.2.5 使用方式

1. 确认证书模式和加密策略。
2. 在对应的配置文件内增加证书配置和策略配置。

例如使用自签名证书，并对客户端、Graph 服务、Meta 服务和 Storage 服务之间的数据传输进行加密。需要对三个配置文件都进行如下设置：

```
--cert_path=xxxxxx  
--key_path=xxxxxx  
--password_path=xxxxxx  
--enable_ssl=true
```

3. 客户端设置安全套接字并添加受信任的 CA。示例代码请参见 [nebula-test-run.py](#)。

最后更新: May 4, 2023

9. 备份与恢复

9.1 管理快照

NebulaGraph提供快照（snapshot）功能，用于保存集群当前时间点的数据状态，当出现数据丢失或误操作时，可以通过快照恢复数据。

9.1.1 前提条件

NebulaGraph的**身份认证**功能默认是关闭的，此时任何用户都能使用快照功能。

如果身份认证开启，仅 God 角色用户可以使用快照功能。关于角色说明，请参见[内置角色权限](#)。

9.1.2 注意事项

- 系统结构发生变化后，建议立刻创建快照，例如在 add host、drop host、create space、drop space、balance 等操作之后。
- 不支持自动回收创建失败的快照垃圾文件，需要手动删除。
- 不支持指定快照保存路径，默认路径为 /usr/local/nebula/data。

9.1.3 快照路径

NebulaGraph创建的快照以目录的形式存储，例如 SNAPSHOT_2021_03_09_08_43_12，后缀 2021_03_09_08_43_12 根据创建时间（UTC）自动生成。

创建快照时，快照目录会自动在 leader Meta 服务器和所有 Storage 服务器的目录 checkpoints 内创建。

为了快速定位快照所在路径，可以使用 Linux 命令 find。例如：

```
$ find |grep 'SNAPSHOT_2021_03_09_08_43_12'
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_09_08_43_12
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_09_08_43_12/data
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_09_08_43_12/data/000081.sst
...
```

9.1.4 创建快照

命令 CREATE SNAPSHOT 可以创建集群当前时间点的快照。只支持创建所有图空间的快照，不支持创建指定图空间的快照。



如果快照创建失败，请[删除快照](#)重新创建。

```
nebula> CREATE SNAPSHOT;
```

9.1.5 查看快照

命令 SHOW SNAPSHOTS 可以查看集群中的所有快照。

```
nebula> SHOW SNAPSHOTS;
+-----+-----+-----+
| Name | Status | Hosts |
+-----+-----+-----+
| "SNAPSHOT_2021_03_09_08_43_12" | "VALID" | "127.0.0.1:9779" |
| "SNAPSHOT_2021_03_09_09_10_52" | "VALID" | "127.0.0.1:9779" |
+-----+-----+-----+
```

参数说明如下：

参数	说明
Name	快照名称，前缀为 <code>SNAPSHOT</code> ，表示该文件为快照文件，后缀为快照创建的时间点（UTC 时间）。
Status	快照状态。 <code>VALID</code> 表示快照有效， <code>INVALID</code> 表示快照无效。
Hosts	创建快照时所有 <code>Storage</code> 服务器的 IP 地址和端口。

9.1.6 删除快照

命令 `DROP SNAPSHOT` 可以删除指定的快照，语法为：

```
DROP SNAPSHOT <snapshot_name>;
```

示例如下：

```
nebula> DROP SNAPSHOT SNAPSHOT_2021_03_09_08_43_12;
nebula> SHOW SNAPSHOTS;
+-----+-----+-----+
| Name          | Status | Hosts      |
+-----+-----+-----+
| "SNAPSHOT_2021_03_09_10_52" | "VALID" | "127.0.0.1:9779" |
+-----+-----+-----+
```

9.1.7 恢复快照



恢复快照数据时，如果快照中备份的图空间被删除（执行了 `DROP SPACE` 操作），那么该图空间数据将无法恢复。

当前未提供恢复快照命令，需要手动拷贝快照文件到对应的文件夹内，也可以通过 `shell` 脚本进行操作。实现逻辑如下：

1. 创建快照后，会在 `leader Meta` 服务器和所有 `Storage` 服务器的安装目录内生成 `checkpoints` 目录，保存创建的快照。以本文为例，当存在 2 个图空间时，创建的快照分别保存在 `/usr/local/nebula/data/meta/nebula/0/checkpoints`、`/usr/local/nebula/data/storage/nebula/3/checkpoints` 和 `/usr/local/nebula/data/storage/nebula/4/checkpoints` 中。

```
$ ls /usr/local/nebula/data/meta/nebula/0/checkpoints/
SNAPSHOT_2021_03_09_10_52
$ ls /usr/local/nebula/data/storage/nebula/3/checkpoints/
SNAPSHOT_2021_03_09_10_52
$ ls /usr/local/nebula/data/storage/nebula/4/checkpoints/
SNAPSHOT_2021_03_09_10_52
```

2. 当数据丢失需要通过快照恢复时，用户可以找到合适的时间点快照，将内部的文件夹 `data` 和 `wal` 分别拷贝到各自的上级目录（和 `checkpoints` 平级），覆盖之前的 `data` 和 `wal`，然后重启集群即可。



需要同时覆盖所有 `Meta` 节点的 `data` 和 `wal` 目录，因为存在重启集群后发生 `Meta` 重新选举 `leader` 的情况，如果不覆盖所有 `Meta` 节点，新的 `leader` 使用的还是最新的 `Meta` 数据，导致恢复失败。

最后更新: May 4, 2023

10. 同步与迁移

10.1 BALANCE

BALANCE 语句可以让NebulaGraph的 Storage 服务实现负载均衡。更多 BALANCE 语句示例和 Storage 负载均衡, 请参见 [Storage 负载均衡](#)。

BALANCE 语法说明如下。

语法	说明
<code>BALANCE LEADER</code>	启动任务均衡分布所有图空间中的 leader。该命令会返回任务 ID（job_id）。

查看、停止、重启任务, 请参见[作业管理](#)。

最后更新: May 4, 2023

11. 最佳实践

11.1 Compaction

本文介绍 Compaction 的相关信息。

NebulaGraph 中，Compaction 是最重要的后台操作，对性能有极其重要的影响。

Compaction 操作会读取硬盘上的数据，然后重组数据结构和索引，然后再写回硬盘，可以成倍提升读取性能。将大量数据写入 NebulaGraph 后，为了提高读取性能，需要手动触发 Compaction 操作（全量 Compaction）。

Note

Compaction 操作会长时间占用硬盘的 IO，建议在业务低峰期（例如凌晨）执行该操作。

NebulaGraph 有两种类型的 Compaction 操作：自动 Compaction 和全量 Compaction。

11.1.1 自动 Compaction

自动 Compaction 是在系统读取数据、写入数据或系统重启时自动触发 Compaction 操作，提升短时间内的读取性能。默认情况下，自动 Compaction 是开启状态，可能在业务高峰期触发，导致意外抢占 IO 影响业务。

11.1.2 全量 Compaction

全量 Compaction 可以对图空间进行大规模后台操作，例如合并文件、删除 TTL 过期数据等，该操作需要手动发起。使用如下语句执行全量 Compaction 操作：

Note

建议在业务低峰期（例如凌晨）执行该操作，避免大量占用硬盘 IO 影响业务。

```
nebula> USE <your_graph_space>;  
nebula> SUBMIT JOB COMPACT;
```

上述命令会返回作业的 ID，用户可以使用如下命令查看 Compaction 状态：

```
nebula> SHOW JOB <job_id>;
```

11.1.3 操作建议

为保证 NebulaGraph 的性能，请参考如下操作建议：

- 数据导入完成后，请执行 `SUBMIT JOB COMPACT`。
- 业务低峰期（例如凌晨）执行 `SUBMIT JOB COMPACT`。
- 为控制 Compaction 的写入速率，请在配置文件 `nebula-storaged.conf` 中设置如下参数（注：此参数限制全部写入，包括正常写入和 Compaction）：

```
# 写入速度限制为 20MB/S。  
--rocksdb_rate_limit=20 (in MB/s)
```

11.1.4 FAQ

Compaction 相关的日志在哪?

默认情况下，`/usr/local/nebula/data/storage/nebula/{1}/data/` 目录下的文件名为 `LOG` 文件，或者类似 `LOG.old.1625797988509303`，找到如下的部分。

** Compaction Stats [default] **																			
Level	Files	Size	Score	Read(GB)	Rn(GB)	Rnp1(GB)	Write(GB)	Wnew(GB)	Moved(GB)	W-Amp	Rd(MB/s)	Wr(MB/s)	Comp(sec)	CompMergeCPU(sec)	Comp(cnt)	Avg(sec)	KeyIn	KeyDrop	
L0	2/0	2.46 KB	0.5	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.53	0.51	2	0.264	0	0	
Sum	2/0	2.46 KB	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.53	0.51	2	0.264	0	0	
Int	0/0	0.00 KB	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0	0.000	0	0	

如果当前的 `L0` 文件数量较多，对读性能影响较大，可以触发 `compaction`。

可以同时在多个图空间执行全量 **Compaction** 操作吗?

可以，但是此时的硬盘 IO 会很高，可能会影响效率。

全量 **Compaction** 操作会耗费多长时间?

如果已经设置读写速率限制，例如 `rocksdb_rate_limit` 限制为 20MB/S 时，用户可以通过 硬盘使用量/`rocksdb_rate_limit` 预估需要耗费的时间。如果没有设置读写速率限制，根据经验，速率大约为 50MB/S。

可以动态调整 `rocksdb_rate_limit` 吗?

不可以。

全量 **Compaction** 操作开始后可以停止吗?

不可以停止，必须等待操作完成。这是 RocksDB 的限制。

最后更新: May 4, 2023

11.2 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 9, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

11.3 图建模设计

本文介绍在NebulaGraph项目中成功应用的一些图建模和系统设计的通用建议。



本文建议是通用的，在特定领域有例外，请结合实际业务情况进行图建模。

11.3.1 以性能为第一目标进行建模

目前NebulaGraph没有完美的建模方法，如何建模取决于想从数据中挖掘的内容。分析数据并根据业务模型创建方便直观的数据模型，测试模型并优化，逐渐适应业务。为了更好的性能，用户可以多次更改或重新设计模型。

设计和评估最重要的查询语句

在测试环节中，通常会验证各种各样的查询语句，以全面评估系统能力。但在大多数生产场景下，每个集群被频繁调用的查询语句的类型并不会太多；根据 20-80 原则，针对重要的查询语句进行建模优化。

避免全量扫描

通过属性索引或者 VID 来先定位到某个（些）点或者边，然后开始图遍历；对于有些查询，它们只给定了一个子图或者路径的（正则）模式，但无法通过属性索引或者 VID 定位到遍历起始的点边，而期望找到库中全部满足该模式的子图，这样的查询是通过全量扫描实现的，这样的性能会很差。NebulaGraph没有实现对于子图或者路径的图结构的索引。

Tag 与 Edge type 之间没有绑定关系

任何 Tag 可以与任何 Edge type 相关联，完全交由应用程序控制。不需要在NebulaGraph中预先定义，也没有命令获取哪些 Tag 与哪些 Edge type 相关联。

Tag/Edge type 预先定义了一组属性

建立 Tag（或者 Edge type）时，需要指定对应的属性。通常称为 Schema。

区分“经常改变的部分”和“不经常改变的部分”

改变指的是业务模型和数据模型上的改变（元信息），不是数据自身的改变。

一些图数据库产品是 schema-free 的设计，所以在数据模型上，不论是图拓扑结构还是属性，都可以非常自由。属性可以建模转变为图拓扑，反之亦然。这类系统通常对于图拓扑的访问有特别的优化。

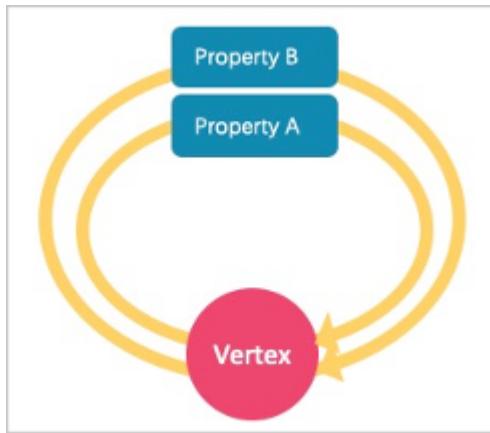
而NebulaGraph 3.4.1 是强 Schema 的（行存型）系统，这意味着业务数据模型中的部分是“不应该经常改变的”，例如属性 Schema 应该避免改变。类似于 MySQL 中 ALTER TABLE 是应该尽量避免的操作。

而点及邻边可以非常低成本的增删，因此可以将业务模型中“经常改变的部分”建模成点或边（关系），而不是属性 Schema。

例如，在一个业务模型中，人的属性是相对固定的，例如“年龄”，“性别”，“姓名”。而“通信好友”，“出入场所”，“交易账号”，“登录设备”等是相对容易改变的。前者适合建模为属性，后者适合建模为点或边。

自环

NebulaGraph是强 Schema 类型系统，使用 ALTER TAG 的开销很大，而且也不支持List类型属性，当用户需要为点添加一些临时属性或者List类型的属性时，可以先创建包含所需属性的边类型，然后为点插入一条或多条指向自身的边。查询时只需要查询指向自己的边属性。如下图所示。



示例：

如果要检索点的临时属性，请从自环边中获取。例如：

```
//创建边类型并插入自环属性。
nebula> CREATE EDGE IF NOT EXISTS temp(tmp int);
nebula> INSERT EDGE temp(tmp) VALUES "player100"->"player100"@1:(1);
nebula> INSERT EDGE temp(tmp) VALUES "player100"->"player100"@2:(2);
nebula> INSERT EDGE temp(tmp) VALUES "player100"->"player100"@3:(3);

//插入数据后，可以通过查询语句查询，例如：
nebula> GO FROM "player100" OVER temp YIELD properties(edge).tmp;
+-----+
| properties(EDGE).tmp |
+-----+
| 1 |
| 2 |
| 3 |
+-----+

//如果需要返回结果为List，可以通过函数实现，例如：
nebula> MATCH (v1:player)-[e:temp]->() return collect(e.tmp);
+-----+
| collect(e.tmp) |
+-----+
| [1, 2, 3] |
+-----+
```

对于自环的操作没有封装任何的语法糖，操作方式与普通的边无异。

- NebulaGraph自环小科普（2分54秒）



悬挂边

悬挂边 (Dangling edge) 是指一条起点或者终点不存在于数据库中的边。

在NebulaGraph 3.4.1中，有两种情况可能会出现悬挂边。

第一种情况：在起点和终点插入之前，用 **INSERT EDGE** 语句插入一条边。

第二种情况：使用 **DELETE VERTEX** 语句删除点的时候，没有使用 **WITH EDGE** 选项。此时系统默认不删除该点关联的出边和入边，这些边将变成悬挂边。

NebulaGraph 3.4.1 的数据模型中，由于设计允许图中存在“悬挂边”；没有 openCypher 中的 MERGE 语句。对于悬挂边的保证完全依赖应用层面。用户可以使用 **GO** 和 **LOOKUP** 语句查询到悬挂边，但无法使用 **MATCH** 语句查询到悬挂边。

示例：

```
// 插入起点为"11"，终点为"13"并且都不存在于数据库中的悬挂边
nebula> CREATE EDGE IF NOT EXISTS e1 (name string, age int);
nebula> INSERT EDGE e1 (name, age) VALUES "11"->"13":("n1", 1);

// 使用 GO 语句查询
nebula> GO FROM "11" over e1 YIELD properties(edge);
+-----+
| properties(EDGE) |
+-----+
```

```
+-----+
| {age: 1, name: "n1"} |
+-----+
// 使用 LOOKUP 语句查询
nebula> LOOKUP ON e1 YIELD EDGE AS r;
+-----+
| r |
+-----+
| [:e2 "11"~>"13" @0 {age: 1, name: "n1"}] |
+-----+
// 使用 MATCH 查询, 不能查询到悬挂边
nebula> MATCH ()-[e:e1]->() RETURN e;
+---+
| e |
+---+
+---+
Empty set (time spent 3153/3573 us)
```

- NebulaGraph的悬挂边小科普 (2 分 28 秒)



广度优先大于深度优先

- NebulaGraph基于图拓扑结构进行深度图遍历的性能较低，广度优先遍历以及获取属性的性能较好。例如，模型 a 包括姓名、年龄、眼睛颜色三种属性，建议创建一个 Tag person，然后为它添加姓名、年龄、眼睛颜色的属性。如果创建一个包含眼睛颜色的 Tag 和一个 Edge type has，然后创建一个边用来表示人拥有的眼睛颜色，这种建模方法会降低遍历性能。
- “通过边属性获取边”的性能与“通过点属性获取点”的性能是接近的。在一些数据库中，会建议将边上的属性重新建模为中间节点的属性：例如 (src)-[edge {P1, P2}]->(dst)，edge 上有属性 P1, P2，会建议建模为 (src)-[edge1]->(i_node {P1, P2})-[edge2]->(dst)。在NebulaGraph 3.4.1 中可以直接使用 (src)-[edge {P1, P2}]->(dst)，减少遍历深度有助于性能。

边的方向

查询时，如果需要使用边的逆向查询，可以用如下语法：

```
(dst)<-[edge]-(src) 或者 GO FROM dst REVERSELY ;
```

如果不关心边的方向，可以使用如下语法：

```
(src)-[edge]-(dst) 或者 GO FROM src BIDIRECT ;
```

因此，通常同一条边没有必要反向再冗余插入一次。

合理设置 Tag 属性

在图建模中，请将一组类似的平级属性放入同一个 Tag，即按不同概念进行分组。

正确使用索引

使用属性索引可以通过属性查找到 VID。但是索引会导致写性能大幅下降，只有在根据点或边的属性定位点或边时才使用索引。

合理设计 VID

参考[点 VID 一节](#)。

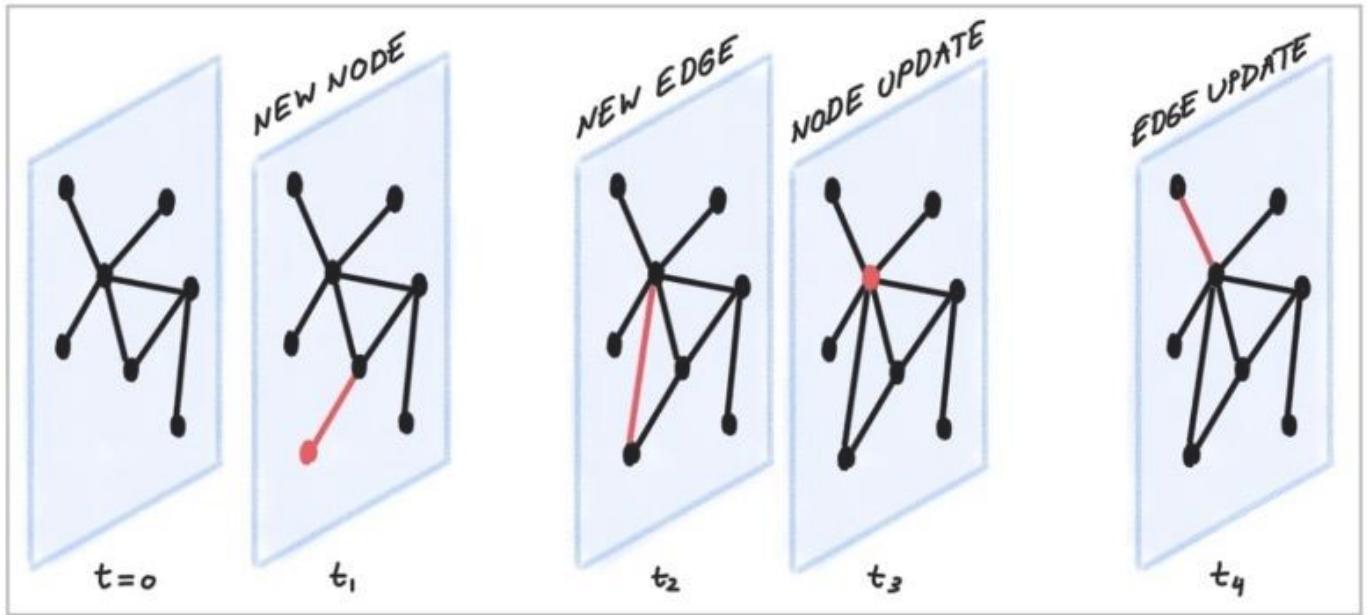
长文本

为边创建属性时请勿使用长文本：这些属性会被**存储 2 份**，导致写入放大问题（write amplification）。此时建议将长文本放在 HBase/ES 中，将其地址存放在NebulaGraph中。

11.3.2 关于支持动态图 (时序图)

在某些场景下，图需要同时带有时序信息，以描述整个图的结构随着时间变化的情况 [1](#)。

NebulaGraph 3.4.1 的边可以使用 Rank 字段存放时间信息 (int64)，但是点上没有字段可以存放时间信息（存放在属性会被新写入覆盖）。一个折中的办法是在点上设计自己指向自己的自环，并将时间信息放置在自环的 Rank 上。



11.3.3 一些免费的建模工具

[arrows.app](#)

1. https://blog.twitter.com/engineering/en_us/topics/insights/2021/temporal-graph-networks ↪

最后更新: May 4, 2023

11.4 系统设计建议

11.4.1 选择 QPS 优先或时延优先

- NebulaGraph 3.4.1 更擅长处理（互联网式的）有大量并发的小请求。也即：虽然全图很大（万亿点边），但是每个请求要访问到的子图本身并不大（几百万个点边）——单个请求时延不大；但这类请求的并发数量特别多——QPS 大。
- 但对于一些交互分析型的场景，并发请求的数量不多，而每个请求要访问的子图本身特别大（亿以上）。为降低时延，可以在应用程序中将一个大的请求，拆分为多个小请求，并发发送给多个 graphd。这样可以降低单个大请求的时延，降低单个 graphd 的内存占用。另外，也可以使用[图计算功能 NebulaGraph Algorithm](#)。

11.4.2 数据传输与优化

- 读写平衡。NebulaGraph适合读写平衡性的在线场景，也即 OLTP 型的“并发的发生写入与读取”；而非数仓 OLAP 型的“一次写入多次读取”。
- 选择不同的写入方式。大批量的数据写入可以使用 sst 加载的方式；小批量的写入使用 INSERT 语句。
- 选择合适的时间运行 COMPACTION 和 BALANCE，来分别优化数据格式和存储分布。
- NebulaGraph 3.4.1 不支持关系型数据库意义上的事务和隔离性，更接近 NoSQL。

11.4.3 查询预热与数据预热

应用端进行预热：

- Graphd 不支持预编译查询及相应生成查询计划，也不支持缓存之前的查询结果；
- Storaged 不支持预热数据，只有 RocksDB 自身的 LSM-tree 和 BloomFilter 会启动时加载到内存中。
- 点和边被访问过后，会各自缓存在 Storaged 的两种 (LRU) Cache 中。

最后更新: May 4, 2023

11.5 执行计划

NebulaGraph 3.4.1 实现了基于规则的执行计划。用户无法改变执行计划，无法进行查询的预编译（及相应的计划缓存），无法通过指定索引来加速查询。

要查看执行计划及执行概要，请参考 [EXPLAIN](#) 和 [PROFILE](#)。

最后更新: May 4, 2023

11.6 超级顶点（稠密点）处理

11.6.1 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 NebulaGraph 3.4.1 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。



NebulaGraph 中没有专用的字段来记录每个点的出度和入度，也没有内置任务来进行统计，因此无法预知哪些点会是超级节点。一个折中的办法是使用 Spark 周期性地计算和统计。

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”。

NebulaGraph 3.4.1 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NebulaGraph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. **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 Edge type 拆分更细的 Edge type，如 depart_ceair, depart_csaic 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。

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，都转变为对于 A1、A2、A3 的三次单独操作。

最后更新: May 4, 2023

11.7 启用 AutoFDO

AutoFDO 可以对优化过的程序进行性能分析，并使用性能信息来指导编译器再次优化程序。本文将帮助您为NebulaGraph启用AutoFDO。

关于 AutoFDO 的更多信息，请参见 [AutoFDO Wiki](#)。

11.7.1 准备资源

安装依赖

- 安装 perf

```
sudo apt-get update
sudo apt-get install -y linux-tools-common \
linux-tools-generic \
linux-tools-'uname -r'
```

- 安装 autofdo tool

```
sudo apt-get update
sudo apt-get install -y autofdo
```

或者你可以从[源代码](#)编译 **autofdo tool**。

编译**NebulaGraph**二进制文件

关于如何从源码编译NebulaGraph，请参考[使用源码安装NebulaGraph](#)。

在配置步骤中，将 `CMAKE_BUILD_TYPE=Release` 替换为 `CMAKE_BUILD_TYPE=RelWithDebInfo`：

```
$ cmake -DCMAKE_INSTALL_PREFIX=/usr/local/nebula -DENABLE_TESTING=OFF -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

11.7.2 准备测试数据

在测试环境中，我们使用 **NebulaGraph Bench** 来准备测试数据，并通过运行 **FindShortestPath**、**Go1Step**、**Go2Step**、**Go3Step**、**InsertPersonScenario** 这5个场景脚本来收集性能数据。



可以在生产环境中使用 **TopN** 查询来收集性能数据，在你的环境中可以提高更多性能。

11.7.3 准备性能数据

收集 **AutoFDO** 工具的性能数据

1. 测试数据准备完成后，收集不同场景的性能数据。首先获取 `storaged`、`graphd`、`metad` 的 pid。

```
$ nebula.service status all
[INFO] nebula-metad: Running as 305422, Listening on 9559
[INFO] nebula-graphd: Running as 305516, Listening on 9669
[INFO] nebula-storaged: Running as 305707, Listening on 9779
```

2. 为 `nebula-graphd` 和 `nebula-storaged` 启动 `perf record`。

```
perf record -p 305516,305707 -b -e br_inst_retired.near_taken:pp -o ~/FindShortestPath.data
```

Note

因为与 nebula-graphd 和 nebula-storaged 相比，nebula-metad 的贡献率很小。为了减少工作量，我们没有收集 nebula-metad 的性能数据。

3. 启动 **FindShortestPath** 场景的基准测试。

```
cd NebulaGraph-Bench
python3 run.py stress run -s benchmark -scenario find_path.FindShortestPath -a localhost:9669 --args=' -u 100 -i 100000'
```

4. 测试完成后，按 **Ctrl + C** 结束 perf record。

5. 重复上述步骤为 **Go1Step**、**Go2Step**、**Go3Step**、**InsertPersonScenario** 这4个场景收集性能数据。

创建 Gcov 文件

```
create_gcov --binary=$NEBULA_HOME/bin/nebula-storaged \
--profile=~/FindShortestPath.data \
--gcov=~/FindShortestPath-storaged.gcov \
-gcov_version=1

create_gcov --binary=$NEBULA_HOME/bin/nebula-graphd \
--profile=~/FindShortestPath.data \
--gcov=~/FindShortestPath-graphd.gcov \
-gcov_version=1
```

按照上面 **FindShortestPath** 的例子，为 **Go1Step**、**Go2Step**、**Go3Step**、**InsertPersonScenario** 这4个场景也创建 Gcov 文件。

合并性能数据

```
profile_merger ~/FindShortestPath-graphd.gcov \
~/FindShortestPath-storaged.gcov \
~/go1step-storaged.gcov \
~/go1step-graphd.gcov \
~/go2step-storaged.gcov \
~/go2step-graphd.gcov \
~/go3step-storaged.gcov \
~/go3step-graphd.gcov \
~/go3step-master-graphd.gcov \
~/InsertPersonScenario-storaged.gcov \
~/InsertPersonScenario-graphd.gcov
```

合并后的配置文件名称为 `fbdata.afdo`。

11.7.4 使用合并的性能数据文件重新编译NebulaGraph二进制文件

使用编译选项 `-fauto-profile` 重新编译NebulaGraph二进制文件。

```
diff -git a/cmake/nebula/GeneralCompilerConfig.cmake b/cmake/nebula/GeneralCompilerConfig.cmake
@@ -20,6 +20,8 @@ add_compile_options(-Wshadow)
 add_compile_options(-Wnon-virtual-dtor)
 add_compile_options(-Woverloaded-virtual)
 add_compile_options(-Wignored-qualifiers)
+add_compile_options(-fauto-profile=~/fbdata.afdo)
```

Note

当你使用多个 `fbdata.afdo` 多次编译时，请在重新编译之前执行 `make clean` 操作，因为只是修改 `fbdata.afdo` 不会触发重新编译。

11.7.5 性能测试结果

软硬件环境

类型	环境
CPU Processor#	2
Sockets	2
NUMA	2
CPU Type	Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz
Cores per Processor	40C80T
Cache	L1 data: 48KB L1 i: 32KB L2: 1.25MB per physical core L3: shared 60MB per processor
Memory	Micron DDR4 3200MT/s 16GB16Micron DDR4 3200MT/s 16GB16
SSD Disk	INTEL SSDPE2KE016T8
SSD R/W Sequential	3200 MB/s (read) / 2100 MB/s(write)
Nebula Version	master with commit id 51d84a4ed7d2a032a337e3b996c927e3bc5d1415
Kernel	4.18.0-408.el8.x86_64

测试结果

场景	Average Latency(LiB)	Default Binary	Optimized Binary with AutoFDO	P95 Latency (LiB)	Default Binary	Optimized Binary with AutoFDO
FindShortestPath	1	8072.52	7260.10	1	22102.00	19108.00
	2	8034.32	7218.59	2	22060.85	19006.00
	3	8079.27	7257.24	3	22147.00	19053.00
	4	8087.66	7221.39	4	22143.00	19050.00
	5	8044.77	7239.85	5	22181.00	19055.00
	STDDEVP	20.57	17.34	STDDEVP	41.41	32.36
	Mean	8063.71	7239.43	Mean	22126.77	19054.40
	STDDEVP/ Mean	0.26%	0.24%	STDDEVP/ Mean	0.19%	0.17%
	Opt/Default	100.00%	10.22%	Opt/ Default	100.00%	13.89%
Go1Step	1	422.53	418.37	1	838.00	850.00
	2	432.37	402.44	2	866.00	815.00
	3	437.45	407.98	3	874.00	836.00
	4	429.16	408.38	4	858.00	838.00
	5	446.38	411.32	5	901.00	837.00
	STDDEVP	8.02	5.20	STDDEVP	20.63	11.30
	Mean	433.58	409.70	Mean	867.40	835.20
	STDDEVP/ Mean	1.85%	1.27%	STDDEVP/ Mean	2.38%	1.35%
	Opt/Default	100.00%	5.51%	Opt/ Default	100.00%	3.71%
Go2Step	1	2989.93	2824.29	1	10202.00	9656.95
	2	2957.22	2834.55	2	10129.00	9632.40
	3	2962.74	2818.62	3	10168.40	9624.70
	4	2992.39	2817.27	4	10285.10	9647.50
	5	2934.85	2834.91	5	10025.00	9699.65
	STDDEVP	21.53	7.57	STDDEVP	85.62	26.25
	Mean	2967.43	2825.93	Mean	10161.90	9652.24
	STDDEVP/ Mean	0.73%	0.27%	STDDEVP/ Mean	0.84%	0.27%
	Opt/Default	100.00%	4.77%	Opt/ Default	100.00%	5.02%
Go3Step	1	93551.97	89406.96	1	371359.55	345433.50
	2	92418.43	89977.25	2	368868.00	352375.20
	3	92587.67	90339.25	3	365390.15	356198.55

场景	Average Latency(LiB)	Default Binary	Optimized Binary with AutoFDO	P95 Latency (LiB)	Default Binary	Optimized Binary with AutoFDO
4	93371.64	92458.95	4	373578.15	365177.75	
5	94046.05	89943.44	5	373392.25	352576.00	
STDDEV_P	609.07	1059.54	STDDEV_P	3077.38	6437.52	
Mean	93195.15	90425.17	Mean	370517.62	354352.20	
STDDEV_P/Mean	0.65%	1.17%	STDDEV_P/Mean	0.83%	1.82%	
Opt/Default	100.00%	2.97%	Opt/Default	100.00%	4.36%	
<i>InsertPerson</i>	1	2022.86	1937.36	1	2689.00	2633.45
2	1966.05	1935.41	2	2620.45	2555.00	
3	1985.25	1953.58	3	2546.00	2593.00	
4	2026.73	1887.28	4	2564.00	2394.00	
5	2007.55	1964.41	5	2676.00	2581.00	
STDDEV_P	23.02	26.42	STDDEV_P	57.45	82.62	
Mean	2001.69	1935.61	Mean	2619.09	2551.29	
STDDEV_P/Mean	1.15%	1.37%	STDDEV_P/Mean	2.19%	3.24%	
Opt/Default	100.00%	3.30%	Opt/Default	100.00%	2.59%	

最后更新: May 4, 2023

11.8 实践案例

NebulaGraph在各行各业都有应用，本文介绍部分实践案例。更多实践分享内容请参见[博客](#)。

11.8.1 业务场景

- 案例
- 经验
- 三方评测

11.8.2 内核

- MATCH 中变长 Pattern 的实现
- 如何向NebulaGraph增加一个测试用例
- 基于 BDD 理论的NebulaGraph集成测试框架重构（上）
- 基于 BDD 理论的NebulaGraph集成测试框架重构（下）
- 解析NebulaGraph子图设计及实践
- 基于全文搜索引擎的文本搜索
- 实操 | LDBC 数据导入及 nGQL 实践

11.8.3 周边工具

- 基于NebulaGraphImporter 批量导入工具性能验证方案总结
 - 详解NebulaGraph 3.0 性能报告
 - NebulaGraph 支持 JDBC 协议
 - Nebula·利器 | Norm 知乎开源的 ORM 工具
 - 基于NebulaGraph的 Betweenness Centrality 算法
 - 无依赖单机尝鲜 NebulaGraph Exchange 的 SST 导入
 - logrotate 在NebulaGraph的日志滚动实践
- # # 视频
- Nebula 高性能图 schema 设计 by 青藤云安全 (51 分 30 秒)
- |
- 同花顺图数据库选型：消息面、基本面、技术面 (21 分 53 秒)
- |
- Nebula 在 Akulaku 智能风控的实践 (40 分 03 秒)
- |
- 从零到一：如何使用NebulaGraph构建一个企业股权图谱系统 (09 分 34 秒)
- |
- 美团图数据库平台建设及业务实践（上） (14 分 36 秒)

- 美团图数据库平台建设及业务实践（下）（21 分 33 秒）
- 信息图谱在携程酒店的应用（39 分 06 秒）
- OPPO 图平台建设（43 分 09 秒）
- NebulaGraph在网易游戏业务中的实践（47 分 40 秒）
- BIGO 数据管理与应用实践（53 分 47 秒）
- NebulaGraph保险反欺诈解决方案 Demo 分享（39 分 54 秒）
- NebulaGraph人际关系查询之疫情防控场景 Demo 分享（1 时 01 分 55 秒）
- NebulaGraph在携程金融风控的应用（29 分 06 秒）

最后更新: May 4, 2023

12. 客户端

12.1 客户端介绍

NebulaGraph提供多种类型客户端，便于用户连接、管理NebulaGraph图数据库。

- **NebulaGraph Console**: 原生 CLI 客户端
- **NebulaGraph CPP**: C++ 客户端
- **NebulaGraph Java**: Java 客户端
- **NebulaGraph Python**: Python 客户端
- **NebulaGraph Go**: Go 客户端



目前仅 NebulaGraph Java 客户端支持线程安全（thread-safe）。



以下客户端工具也可用于连接和管理NebulaGraph。他们由非常酷的社区用户提供和维护，欢迎大家参与测试和贡献。

- **NebulaGraph PHP**
- **NebulaGraph Node**
- **NebulaGraph .net**
- **NebulaGraph JDBC**
- **NebulaGraph Carina (Python ORM)**
- **NORM (Golang ORM)**
- **Graph-Ocean (Java ORM)**
- **NebulaGraph Ngbatis (MyBatis 风格的 Java ORM)**

最后更新: May 4, 2023

12.2 NebulaGraph Console

NebulaGraph Console 是NebulaGraph的原生命令行客户端，用于连接NebulaGraph集群并执行查询，同时支持管理参数、导出命令的执行结果、导入测试数据集等功能。

12.2.1 获取 NebulaGraph Console

NebulaGraph Console 的获取方式如下：

- 从 [GitHub 发布页](#) 下载二进制文件。
- 编译源码获取二进制文件。编译方法参见 [Install from source code](#)。

12.2.2 功能说明

连接NebulaGraph

运行二进制文件 `nebula-console` 连接NebulaGraph的命令语法如下：

```
<path_of_console> -addr <ip> -port <port> -u <username> -p <password>
```

`path_of_console` 是 NebulaGraph Console 二进制文件的存储路径。

常用参数的说明如下。

参数	说明
<code>-h/-help</code>	显示帮助菜单。
<code>-addr/-address</code>	设置要连接的 Graph 服务的 IP 地址。默认地址为 127.0.0.1。
<code>-P/-port</code>	设置要连接的 Graph 服务的端口。默认端口为 9669。
<code>-u/-user</code>	设置NebulaGraph账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 <code>root</code> ）。
<code>-p/-password</code>	设置用户名对应的密码。未启用身份认证时，密码可以填写任意字符。
<code>-t/-timeout</code>	设置整数类型的连接超时时间。单位为毫秒，默认值为 120。
<code>-e/-eval</code>	设置字符串类型的 nGQL 语句。连接成功后会执行一次该语句并返回结果，然后自动断开连接。
<code>-f/-file</code>	设置存储 nGQL 语句的文件的路径。连接成功后会执行该文件内的 nGQL 语句并返回结果，执行完毕后自动断开连接。
<code>-enable_ssl</code>	连接NebulaGraph时使用 SSL 加密。
<code>-ssl_root_ca_path</code>	指定 CA 证书的存储路径。
<code>-ssl_cert_path</code>	指定 CRT 证书的存储路径。
<code>-ssl_private_key_path</code>	指定私钥文件的存储路径。

更多参数参见[项目仓库](#)。

例如，要连接到部署在 192.168.10.8 上的 Graph 服务，运行以下命令。

```
./nebula-console -addr 192.168.10.8 -port 9669 -u Joe -p Joespassword
```

管理参数

NebulaGraph Console 可以保存参数，用于参数化查询。

Note

- VID 不支持参数化查询。
- SAMPLE 子句中不支持参数化查询。
- 会话释放后，参数不会保留。

- 保存参数命令如下：

```
nebula> :param <param_name> => <param_value>;
```

示例：

```
nebula> :param p1 => "Tim Duncan";
nebula> MATCH (v:player{name:$p1})-[:follow]->(n) RETURN v,n;
+-----+-----+
| v | n |
+-----+-----+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | ("player125" :player{age: 41, name: "Manu Ginobili"}) |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | ("player101" :player{age: 36, name: "Tony Parker"}) |
+-----+-----+
```



```
nebula> :param p2 => {"a":3,"b":false,"c":"Tim Duncan"};
nebula> RETURN $p2.b AS b;
+-----+
| b |
+-----+
| false |
+-----+
```

- 查看当前保存的所有参数，命令如下：

```
nebula> :params;
```

- 查看指定参数，命令如下：

```
nebula> :params <param_name>;
```

- 删除指定参数，命令如下：

```
nebula> :param <param_name> =>;
```

导出执行结果

导出命令执行的返回结果，可以保存为 CSV 文件、 DOT 文件或者 Profile/Explain 结果。

Note

- 文件保存在当前工作目录中，即 Linux 命令 `pwd` 显示的目录。
- 命令只对下一条查询语句生效。
- DOT 文件的内容可以复制后在 [GraphvizOnline](#) 网页中粘贴，生成可视化的执行计划图。
- 导出 CSV 文件命令如下：

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

- 导出 DOT 文件命令如下：

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

示例：

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

- 导出 PROFILE/EXPLAIN 结果到文件命令如下：

```
nebula> :profile <file_name>;
```

或者

```
nebula> :explain <file_name>;
```

Note

相比于 Studio 中的截图、CSV 文件，因为保有更多信息量和拥有更好的可读性，经由此命令输出的文本文件内容是首推的在 GitHub issue、论坛中报告执行计划、图查询调优的方式。

示例：

```
nebula> :profile profile.log
nebula> PROFILE GO FROM "player102" OVER serve YIELD dst(edge);
nebula> :profile profile.dot
nebula> PROFILE FORMAT="dot" GO FROM "player102" OVER serve YIELD dst(edge);
nebula> :explain explain.log
nebula> EXPLAIN GO FROM "player102" OVER serve YIELD dst(edge);
```

加载测试数据集

测试数据集名称为 `basketballplayer`，详细 Schema 信息和数据信息请使用相关 SHOW 命令查看。

加载测试数据集命令如下：

```
nebula> :play basketballplayer;
```

重复执行语句

重复执行下一个命令 N 次，然后打印平均执行时间。命令如下：

```
nebula> :repeat N;
```

示例：

```

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

Fri, 20 Aug 2021 06:36:05 UTC

+-----+
| dst(EDGE) |
+-----+
| "player101" |
| "player125" |
+-----+
Got 2 rows (time spent 583/849 us)

Fri, 20 Aug 2021 06:36:05 UTC

+-----+
| dst(EDGE) |
+-----+
| "player101" |
| "player125" |
+-----+
Got 2 rows (time spent 496/671 us)

Fri, 20 Aug 2021 06:36:05 UTC

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

```

睡眠

睡眠 N 秒。常用于修改 Schema 的操作中，因为修改 Schema 是异步实现的，需要在下一个心跳周期才同步数据。命令如下：

```
nebula> :sleep N;
```

断开连接

用户可以使用 :EXIT 或者 :QUIT 从NebulaGraph断开连接。为方便使用，NebulaGraph Console 支持使用不带冒号（:）的小写命令，例如 quit。

示例：

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

最后更新: May 4, 2023

12.3 NebulaGraph CPP

NebulaGraph CPP 是一款 C++ 语言的客户端，可以连接、管理NebulaGraph图数据库。

12.3.1 使用限制

请确保已安装 C++ 且 GCC 版本为 4.8 及以上。

12.3.2 版本对照表

NebulaGraph版本	NebulaGraph CPP 版本
3.3.0	3.3.0
3.1.0 ~ 3.2.x	3.0.2
3.0.0	3.0.0
2.6.x	2.5.0
2.5.x	2.5.0
2.0.x	2.0.0

12.3.3 安装 NebulaGraph CPP

本文介绍通过编译方式安装 NebulaGraph CPP。

前提条件

- 准备正确的编译环境，详情请参见[软硬件要求和安装三方库依赖包](#)。
- 确保已安装 C++ 且 GCC 版本为: {10.1.0 | 9.3.0 | 9.2.0 | 9.1.0 | 8.3.0 | 7.5.0 | 7.1.0}。详情请参见 [gcc_preset_versions 参数](#)。

安装步骤

1. 克隆 NebulaGraph CPP 源码到机器。

- (推荐) 如果需要安装指定版本的 NebulaGraph CPP，请使用选项 `--branch` 指定分支。例如安装 v3.4.0发布版本，请执行如下命令:

```
$ git clone --branch release-3.4 https://github.com/vesoft-inc/nebula-cpp.git
```

- 如果需要安装日常开发版本，请执行如下命令下载 `master` 分支的源码:

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

2. 进入目录 `nebula-cpp`。

```
$ cd nebula-cpp
```

3. 创建目录 `build` 并进入该目录。

```
$ mkdir build && cd build
```

4. 使用 CMake 生成 `makefile` 文件。

Note

默认安装路径为 `/usr/local/nebula`，如果需要修改路径，请在下方命令内增加参数 `-DCMAKE_INSTALL_PREFIX=<installation_path>`。

```
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

Note

如果 `g++` 不支持 `C++11`，请添加选项 `-fno-strict-aliasing`。

5. 编译 NebulaGraph CPP。

为了适当地加快编译速度，可以使用选项 `-j` 并行编译。并行数量 `N` 建议为 $\lfloor \min(\text{CPU} \text{核数}, \frac{\text{内存 (GB)}}{2}) \rfloor$ 。

```
$ make -j{N}
```

6. 安装 NebulaGraph CPP。

```
$ sudo make install
```

7. 更新动态链接库。

```
$ sudo ldconfig
```

12.3.4 使用方法

将 CPP 文件编译为可执行文件即可。接下来以 `SessionExample.cpp` 为例，介绍如何操作。

1. 使用示例代码创建 `SessionExample.cpp` 文件。

2. 编译文件，命令如下：

```
$ LIBRARY_PATH=<library_folder_path>:$LIBRARY_PATH g++ -std=c++11 SessionExample.cpp -I<include_folder_path> -lnebula_graph_client -o session_example
```

- `library_folder_path`：NebulaGraph 动态库文件存储路径，默认为 `/usr/local/nebula/lib64`。

- `include_folder_path`：NebulaGraph 头文件存储路径，默认为 `/usr/local/nebula/include`。

示例：

```
$ LIBRARY_PATH=/usr/local/nebula/lib64:$LIBRARY_PATH g++ -std=c++11 SessionExample.cpp -I/usr/local/nebula/include -lnebula_graph_client -o session_example
```

12.3.5 核心代码

NebulaGraph CPP 客户端提供 `Session Pool` 和 `Connection Pool` 两种方式连接NebulaGraph。使用 `Connection Pool` 需要用户自行管理 `Session` 实例。

• Session Pool

详细示例请参见 [SessionPoolExample](#)。

• Connection Pool

详细示例请参见 [SessionExample](#)。

12.4 NebulaGraph Java

NebulaGraph Java 是一款 Java 语言的客户端，可以连接、管理NebulaGraph图数据库。

12.4.1 前提条件

已安装 Java，版本为 8.0 及以上。

12.4.2 版本对照表

NebulaGraph版本	NebulaGraph Java 版本
3.3.0	3.3.0
3.0.0 ~ 3.2.0	3.0.0
2.6.x	2.6.1
2.0.x	2.0.0
2.0.0-rc1	2.0.0-rc1

12.4.3 下载 NebulaGraph Java

- （推荐）如果需要使用指定版本的 NebulaGraph Java，请使用选项 `--branch` 指定分支。例如使用 v3.4.0发布版本，请执行如下命令：

```
$ git clone --branch release-3.4 https://github.com/vesoft-inc/nebula-java.git
```

- 如果需要安装日常开发版本，请执行如下命令下载 `master` 分支的源码：

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

12.4.4 使用方法

Note

建议一个线程使用一个会话，如果多个线程使用同一个会话，会降低效率。

使用 IDEA 等工具导入 Maven 项目，请在 `pom.xml` 中添加如下依赖：

Note

3.0.0-SNAPSHOT 为日常研发版本，可能存在未知问题，建议使用 `release` 版本号替换 3.0.0-SNAPSHOT。

```
<dependency>
  <groupId>com.vesoft</groupId>
  <artifactId>client</artifactId>
  <version>3.0.0-SNAPSHOT</version>
</dependency>
```

如果无法下载日常研发版本的依赖，请在 `pom.xml` 中添加如下内容（`release` 版本不需要添加）：

```
<repositories>
  <repository>
    <id>snapshots</id>
    <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
```

```
</repository>
</repositories>
```

如果没有 Maven 管理项目, 请手动[下载 JAR 包](#)进行安装。

核心代码

NebulaGraph Java 客户端提供 Connection Pool 和 Session Pool 两种使用方式, 使用 Connection Pool 需要用户自行管理 Session 实例。

- Session Pool

详细示例请参见 [GraphSessionPoolExample](#)。

- Connection Pool

详细示例请参见 [GraphClientExample](#)。

12.4.5 Java 相关库

以下由非常酷的社区用户提供和维护, 欢迎大家参与测试和贡献。

- [java-jdbc](#)
- [java-orm](#)
- [java-springboot demo](#)
- [ngbatis](#)

最后更新: May 4, 2023

12.5 NebulaGraph Python

NebulaGraph Python 是一款 Python 语言的客户端，可以连接、管理NebulaGraph图数据库。

12.5.1 前提条件

已安装 Python，版本为 3.6 及以上。

12.5.2 版本对照表

NebulaGraph版本	NebulaGraph Python 版本
3.3.0	3.3.0
3.1.0 ~ 3.2.x	3.1.0
3.0.0 ~ 3.0.2	3.0.0
2.6.x	2.6.0
2.0.x	2.0.0
2.0.0-rc1	2.0.0rc1

12.5.3 安装 NebulaGraph Python

pip 安装

```
$ pip install nebula3-python==<version>
```

克隆源码安装

1. 克隆 NebulaGraph Python 源码到机器。

- （推荐）如果需要安装指定版本的 NebulaGraph Python，请使用选项 `--branch` 指定分支。例如安装 v3.4.0发布版本，请执行如下命令：

```
$ git clone --branch release-3.4 https://github.com/vesoft-inc/nebula-python.git
```

- 如果需要安装日常开发版本，请执行如下命令下载 `master` 分支的源码：

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

2. 进入目录 `nebula-python`。

```
$ cd nebula-python
```

3. 执行如下命令安装。

```
$ pip install .
```

12.5.4 核心代码

NebulaGraph Python 客户端提供 Connection Pool 和 Session Pool 两种使用方式，使用 Connection Pool 需要用户自行管理 Session 实例。

- Session Pool

详细示例请参见 [SessinPoolExample.py](#)。

使用限制请参见 [Example of using session pool](#)。

- Connection Pool

详细示例请参见 [Example](#)。

最后更新: May 4, 2023

12.6 NebulaGraph Go

NebulaGraph Go 是一款 Go 语言的客户端，可以连接、管理NebulaGraph图数据库。

12.6.1 前提条件

已安装 Go，版本为 1.13 及以上。

12.6.2 版本对照表

NebulaGraph版本	NebulaGraph Go 版本
3.3.0	3.3.0
3.2.x	3.2.0
3.1.0	3.1.0
3.0.0 ~ 3.0.2	3.0.0
2.6.x	2.6.0
2.0.x	2.0.0-GA

12.6.3 下载 NebulaGraph Go

- (推荐) 如果需要下载指定版本的 NebulaGraph Go，请使用选项 `--branch` 指定分支。例如安装 v3.4.0发布版本，请执行如下命令：

```
$ git clone --branch release-3.4 https://github.com/vesoft-inc/nebula-go.git
```

- 如果需要安装日常开发版本，请执行如下命令下载 `master` 分支的源码：

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

12.6.4 安装或更新

安装或更新的命令如下：

```
$ go get -u -v github.com/vesoft-inc/nebula-go@<tag>
```

`tag`：指定分支。例如 `master` 或 `release-3.4`。

12.6.5 核心代码

NebulaGraph GO 客户端提供 Connection Pool 和 Session Pool 两种使用方式，使用 Connection Pool 需要用户自行管理 Session 实例。

- Session Pool

详细示例请参见 [session_pool_example.go](#)。

使用限制请参见 [Usage example](#)。

- Connection Pool

详细示例请参见 [graph_client_basic_example](#) 和 [graph_client_goroutines_example](#)。

13. NebulaGraph Cloud

13.1 什么是 NebulaGraph Cloud

NebulaGraph Cloud（简称 Cloud）是一套集成了NebulaGraph和数据服务的云上服务，支持一键部署NebulaGraph和相关可视化产品。用户可以在几分钟内创建一个图数据库，并快速扩展计算、存储等资源。

13.1.1 产品功能

- 即买即用。用户可以根据自己的业务发展随时创建或停止NebulaGraph实例，业务提供充分的灵活性。
- 集成可视化图数据库管理工具 Nebula Dashboard、NebulaGraph Explorer。可以使用它们导入图数据集、执行 nGQL 语句查询、探索图数据、监控数据等。即使没有图数据库操作经验，也可以快速成为图专家。
- 用户可直接使用 Nebula Console 等客户端连接到云上数据库，而不需要先访问云平台。
- 权限控制。用户可以为指定图空间增加不同角色权限的数据库用户，保证业务数据安全。
- 官方支持。用户可以快速创建工单，咨询使用过程中遇到的问题。

13.1.2 产品优点

- 灵活与弹性：Cloud 最大的优势在于灵活与弹性，无需自购软硬件即可快速（5-10分钟）搭建NebulaGraph。
- 高可用性：Cloud 上创建的NebulaGraph集群采用 Raft 协议，结合云厂商提供的同城多机房实现数据冗余备份，保证云计算基础框架的高可用性、数据的可靠性以及云服务器的高可用性。
- 操作易用：Cloud 提供 Web 页面对NebulaGraph集群进行在线管理，同时提供配套的可视化软件，降低用户使用门槛，快速上手。

13.1.3 视频

用户也可以观看视频了解 NebulaGraph Cloud。

[NebulaGraph Cloud 阿里云版介绍](#) (05 分 37 秒)

最后更新: May 4, 2023

13.2 Nebula Graph Cloud 阿里云版

13.2.1 创建服务实例

在使用 NebulaGraph Cloud 阿里云版服务前，要先在阿里云控制台创建服务实例。

账号与权限

- 准备[阿里云账号](#)。
- 如果使用 RAM 用户创建实例，需为其添加以下权限：
- AliyunECSFullAccess
- AliyunVPCFullAccess
- AliyunROSFulAccess
- AliyunCloudMonitorFullAccess
- AliyunComputeNestUserFullAccess

资源与费用

NebulaGraph Cloud 阿里云版支持免费试用和付费使用，二者的详细说明如下表。

项目	免费试用版	付费版
云资源归属	阿里云官方账号	用户账号
云资源费用	阿里云承担	用户承担
云资源使用时长	30 天（不可续期）	用户创建实例时选择
NebulaGraph许可证费用	试用期内免费	用户承担
NebulaGraph服务许可证有效期	30 天	用户创建实例时选择
许可证过期后数据是否保留	否（因云资源会同时到期）	是
云资源到期后数据是否保留	否	否
数据盘快照费用	阿里云承担	用户承担（可关闭功能）

数据备份

在创建服务实例时，系统会默认开启数据盘自动快照备份，用于周期性备份NebulaGraph数据。

创建付费版服务实例前需[开通快照](#)。免费试用实例使用的是阿里云官方账号下的资源，已开通快照，无需用户手动开通。

默认的快照备份策略为每天 00:00 创建快照。

创建快照会产生费用，详情参见[快照计费](#)。

如需删除快照、修改或删除自动快照策略，参见[快照文档](#)。

套餐版本

NebulaGraph Cloud 阿里云版支持如下套餐版本。

套餐版本	说明
基础版	将所有NebulaGraph服务节点部署在 1 台 ECS 服务器上。
标准版	将 Graph 服务和 Storage 服务分别部署在不同的 ECS 服务器上，每个服务都是单节点（1 台 ECS）。将 NebulaGraph Explorer 等生态工具混合部署在 1 台 ECS 服务器上。
高可用版	将 Graph 服务和 Storage 服务分别部署在不同的 ECS 服务器上，每个服务都包含 3 节点（3 台 ECS）。将 NebulaGraph Explorer 等生态工具混合部署在 1 台 ECS 服务器上。

付费版和免费试用版服务实例支持的套餐版本有所不同，详情参见创建实例页面。

创建付费版服务实例

1. 登录[阿里云控制台](#)。
2. 打开云市场的 [NebulaGraph产品页](#)。
3. 选择套餐版本和购买时长（即NebulaGraph服务的许可证有效期），并单击立即购买。
4. 在创建服务实例页，保持选择模板处的选择不变。如需切换部署架构，可改变选中的选项，重新选择模板。

Note

改变模板会改变之前选择的套餐版本，软件费用（NebulaGraph许可证费用）和创建服务实例需要的资源也会改变。

- 5.（可选）设置服务实例名称。默认值为服务实例 ID。
6. 选择要创建服务实例的地域。
7. 在付费模式设置区域，指定 ECS 服务器的付费方式。默认为按量付费。
 - 按量付费：按照计费周期计费，在每个结算周期生成账单并从账户中扣除相应费用。详情参见[按量付费](#)。
 - 包年包月：先付费后使用。详情参见[包年包月](#)。选择包年包月模式需要指定购买时长周期和购买时长。购买时长周期当前仅支持 **Month**，即按月购买。
8. 在 **NebulaGraph** 配置区域，完成数据盘和 ECS 服务器密码设置。
9. 在基础设施配置区域，完成以下设置。
 - a. 选择专有网络 **VPC 实例 ID**。
如果下拉列表为空，先单击其右侧的新建专有网络，完成专有网络创建。
 - b. 选择交换机可用区。
 - c. 选择业务网络交换机的实例 **ID**。
如果下拉列表为空，先单击其右侧的新建交换机，完成交换机创建。
10. 完成权限确认，并选中我同意授权服务商（杭州悦数科技有限公司）获取上述权限以提供代运维服务。
11. 在页面底部，单击下一步：确认订单。
12. 在服务条款区域，勾选我已阅读并同意《商品在线协议》《云市场平台服务协议》《计算巢服务协议》。
13. 在页面底部，单击立即支付。
14. 在支付页面，选择支付方式，之后单击页面底部的支付。
15. 在支付完成页面，单击管理控制台。
16. 在已购买的服务页面，单击要创建的实例右侧的立即进入计算巢部署或查看部署详情。

Note

如果已购买的服务列表中未显示要创建的实例，刷新页面。

17. 在服务实例详情页查看实例的状态，确保状态为部署中。

部署的平均耗时为 10 分钟。完成后实例的状态变为已部署。

创建免费版试用版服务实例

Danger

免费试用实例不可在生产环境中使用。

1. 登录阿里云计算巢[推荐服务](#)页面，搜索 **NebulaGraph**。

2. 在 **NebulaGraph**集群版卡片上，单击免费试用。

3. 系统会为首次创建NebulaGraph服务实例的用户弹出申请对话框。在申请权限对话框中填写申请人信息。带有红色星号 (*) 的为必填项。

Note

申请通过后需从第 1 步重新开始。

4. (可选) 设置服务实例名称。默认值为服务实例 ID。

5. 选择要创建实例的地域。

6. 在 **NebulaGraph**配置区域，完成数据盘和 ECS 服务器密码设置。

7. 在基础设施配置区域，选择交换机可用区。

8. 完成权限确认，并选中我同意授权服务商（杭州悦数科技有限公司）获取上述权限以提供代运维服务。

9. 在页面底部，单击下一步：确认订单。

10. 在服务条款区域，勾选我已阅读并同意《计算巢服务协议》。

11. 在页面底部，单击开始免费试用。

12. 在提交成功页面，单击去列表查看。

13. 在实例列表中查看目标实例的状态，确保状态为部署中。

部署的平均耗时为 10 分钟。完成后实例的状态变为已部署。

常见问题

Q: 服务实例的状态显示为部署失败怎么处理？

1. 如果使用 RAM 账号创建的实例，确认为该账号授予了本文前提条件中指定的权限。

2. 如果权限符合要求，[删除](#)创建失败的实例，尝试重新创建。

3. 如果仍然创建失败，到 [NebulaGraph论坛](#)寻求帮助。

最后更新: May 4, 2023

13.2.2 使用服务实例

创建好服务实例后，即可查看服务信息，并开始使用NebulaGraph服务。

查看服务实例

在计算巢控制台可以查看服务实例的详细信息，详情参见[查看服务实例](#)。

查看连接信息

连接服务前，需要在阿里云计算巢的实例详情中查看各服务的连接地址。查看方式如下：

1. 登录[服务实例管理](#)页面。



如果要寻找的是免费试用版服务实例，需先在服务实例管理页面单击试用服务标签。

2. 在实例列表中，单击目标实例的服务实例**ID**，或其右侧操作列的详情。

3. 在概览标签页的基本信息区域，可以查看 Graph 服务、Storage 服务，以及NebulaGraph Explorer、Dashboard 等周边工具的连接信息。

连接NebulaGraph

NebulaGraph Cloud 阿里云版提供多种连接方式。

使用客户端直连NEBULAGRAPH

获取到连接地址后，即可使用NebulaGraph的各类客户端连接服务。

连接方式：

直连 Graph 服务的私网或公网 IP 地址。详细连接方式参见[客户端文档](#)。

如需通过公网连接NebulaGraph数据库：

1. 到 Graph 服务部署的 ECS 实例详情中找到其公网 IP 地址。
- 2.（仅付费版需执行该步骤）在NebulaGraph服务实例加入的安全组中，[添加安全组规则](#)，允许从指定的公网 IP 地址访问 TCP 9669 端口。



免费试用版默认允许从公网访问 Graph 服务的 9669 端口。

3. 使用 Graph 服务的公网 IP 地址和 9669 端口访问NebulaGraph。



建议仅通过私网 IP 连接，提高网络和数据的安全性。

使用 EXPLORER 连接NEBULAGRAPH

使用图探索工具可以快速连接NebulaGraph，与图数据进行可视化交互。

连接方式:

1. 在浏览器中使用 **Explorer** 服务访问地址打开 Nebula Explorer。
2. 使用 Explorer 连接NebulaGraph。详情参见[连接数据库](#)。

监控NebulaGraph

在 NebulaGraph Cloud 阿里云版中部署 Dashboard 后，可以使用 Dashboard 实时监控NebulaGraph服务的状态。详情参见[Dashboard 文档](#)。

最后更新: May 4, 2023

13.2.3 扩缩容服务

NebulaGraph Cloud 阿里云版的弹性扩缩容功能支持根据需求扩缩容服务节点，灵活调整集群规模。

使用限制

- 弹性扩缩容功能支持 Storage 扩容，不支持 Storage 缩容。
- 使用弹性扩缩容功能缩容 Graph 服务时，需先使用集群中 NebulaGraph Dashboard 服务的扩缩容功能删除不再需要的 Graph 服务节点。扩容 Graph 服务时无此限制。

操作步骤

- 登录阿里云计算巢的[服务实例管理页面](#)。
- 在服务实例列表中，单击目标实例的 ID 或其右侧操作列的详情。

Note

免费试用版服务实例列表在试用服务标签页中。

- 在服务实例详情页面，单击运维管理标签。
- 在应用运维区域，单击弹性扩缩容。
- 在弹性扩缩容对话框，完成如下操作。
 - 确认将要调整的资源类型为 **ESS**。
 - 选择伸缩组 **ID**。
 - 如需扩缩容 Storage 服务，选择名称以 **ng-storage** 开头的伸缩组。
 - 如需扩缩容 Graph 服务，选择名称以 **ng-graph** 开头的伸缩组。
 - 选择伸缩规则的调整方式。
 - 如果选择扩容或缩容，下方的伸缩组 **ECS** 实例调整的数量指定的是要增加或减少的节点数量。
 - 如果选择调整至指定数量，下方的伸缩组 **ECS** 实例调整的数量指定的是节点调整之后剩余的数量，而不是增加或减少的数量。
 - 输入伸缩组 **ECS** 实例调整的数量。
 - 单击下一步。
 - 检查任务信息，确认无误后单击创建。
- 等候数分钟，直到运维管理标签页中的执行状态显示成功。
- 单击概览标签页，查看服务资源区域的 **ECS** 实例数量，如果符合预期的调整效果，则扩缩容完成。

最后更新: May 4, 2023

13.2.4 删除服务实例

如果不再需要某个实例，可手动删除该实例。

删除方式参见[删除服务实例](#)。

最后更新: May 4, 2023

14. NebulaGraph Dashboard (社区版)

14.1 什么是 NebulaGraph Dashboard (社区版)

NebulaGraph Dashboard (简称 Dashboard) 是一款用于监控NebulaGraph集群中机器和服务状态的可视化工具。本文主要介绍社区版 Dashboard，企业版详情参见[什么是 NebulaGraph Dashboard \(企业版\)](#)。



企业版增加了可视化创建集群、批量导入集群、快速扩缩容等功能，点击[定价](#)查看更多。

14.1.1 产品功能

- 监控集群中所有机器的状态，包括 CPU、内存、负载、磁盘和流量。
- 监控集群中所有服务的信息，包括服务 IP 地址、版本和监控指标（例如查询数量、查询延迟、心跳延迟等）。
- 监控集群本身的信息，包括集群的服务信息、分区信息、配置和长时任务。
- 支持全局调整监控数据的页面更新频率。

14.1.2 适用场景

如果有以下任一需求，都可以使用 Dashboard：

- 需要方便快捷地监测关键指标，集中呈现业务的多个重点信息，保证业务正常运行。
- 需要多维度（例如时间段、聚合规则、指标）监控集群。
- 故障发生后，需要复盘问题，确认故障发生时间、异常现象。

14.1.3 注意事项

监控数据默认保留 14 天，即只能查询最近 14 天内任意时间段的监控数据。



监控服务由 [prometheus](#) 提供，更新频率和保留时间等都可以自行修改。详情请参见 [prometheus 官方文档](#)。

14.1.4 版本兼容性

NebulaGraph的版本和 Dashboard 社区版的版本对应关系如下。

NebulaGraph版本	Dashboard 版本
3.4.0 ~ 3.4.1	3.4.0、3.2.0
3.3.0	3.2.0
2.5.0 ~ 3.2.0	3.1.0
2.5.x ~ 3.1.0	1.1.1
2.0.1 ~ 2.5.1	1.0.2
2.0.1 ~ 2.5.1	1.0.1

14.1.5 更新说明

Release

14.1.6 视频

NebulaGraph Dashboard Demo 介绍 (v3.0 版本) (2 分 57 秒)



最后更新: May 4, 2023

14.2 部署 Dashboard 社区版

本文将介绍如何通过 TAR 包安装部署 NebulaGraph Dashboard。

下载和编译 Dashboard 的最新源码，参见 [GitHub NebulaGraph dashboard](#) 页面的说明。

14.2.1 前提条件

在部署 Dashboard 之前，用户需要确认以下信息：

- NebulaGraph服务已经部署并启动。详细信息参考 [NebulaGraph安装部署](#)。
- 确保以下端口未被使用：
 - 9200
 - 9100
 - 9090
 - 8090
 - 7003
- 待监控的机器上已经安装 node-exporter。安装方法请参见 [Prometheus 官方文档](#)。

14.2.2 操作步骤

1. 下载 TAR 包 [nebula-dashboard-3.4.0.x86_64.tar.gz](#)。
2. 执行命令 `tar -xvf nebula-dashboard-3.4.0.x86_64.tar.gz` 解压缩。
3. 进入解压缩的 `nebula-dashboard` 文件夹，并修改配置文件 `config.yaml`。

配置文件内主要包含 4 种依赖服务的配置和集群的配置。依赖服务的说明如下。

服务名称	默认端口号	说明
nebula-http-gateway	8090	为集群服务提供 HTTP 接口，执行 nGQL 语句与NebulaGraph进行交互。
nebula-stats-exporter	9200	收集集群的性能指标，包括服务 IP 地址、版本和监控指标（例如查询数量、查询延迟、心跳延迟等）。
node-exporter	9100	收集集群中机器的资源信息，包括 CPU、内存、负载、磁盘和流量。
prometheus	9090	存储监控数据的时间序列数据库。

配置文件说明如下。

```
port: 7003 # Web 服务端口。
gateway:
  ip: hostIP # 部署 Dashboard 的机器 IP。
  port: 8090
  https: false # 是否为 HTTPS 端口。
  runmode: dev # 程序运行模式，包括 dev、test、prod。一般用于区分不同运行环境。
stats-exporter:
  ip: hostIP # 部署 Dashboard 的机器 IP。
  nebulaPort: 9200
  https: false # 是否为 HTTPS 端口。
node-exporter:
  - ip: nebulaHostIP_1 # 部署NebulaGraph的机器 IP。
    port: 9100
    https: false # 是否为 HTTPS 端口。
  # - ip: nebulaHostIP_2
  #   port: 9100
  #   https: false
prometheus:
  ip: hostIP # 部署 Dashboard 的机器 IP。
  prometheusPort: 9090
  https: false # 是否为 HTTPS 端口。
```

```

scrape_interval: 5s # 收集监控数据的间隔时间。默认为 1 分钟。
evaluation_interval: 5s # 告警规则扫描时间间隔。默认为 1 分钟。
# 集群节点信息
nebula-cluster:
  name: 'default' # 集群名称
  metad:
    - name: metad0
      endpointIP: nebulaMetadIP # 部署 Meta 服务的机器 IP。
      port: 9559
      endpointPort: 19559
    # - name: metad1
    #   endpointIP: nebulaMetadIP
    #   port: 9559
    #   endpointPort: 19559
  graphd:
    - name: graphd0
      endpointIP: GraphdIP # 部署 Graph 服务的机器 IP。
      port: 9669
      endpointPort: 19669
    # - name: graphd1
    #   endpointIP: GraphdIP
    #   port: 9669
    #   endpointPort: 19669
  storaged:
    - name: storaged0
      endpointIP: StoragedIP # 部署 Storage 服务的机器 IP。
      port: 9779
      endpointPort: 19779
    # - name: storaged1
    #   endpointIP: StoragedIP
    #   port: 9779
    #   endpointPort: 19779

```

4. 执行 `./dashboard.service start all` 一键启动服务。

容器部署

如果使用容器部署 Dashboard，同样是修改配置文件 `config.yaml`，修改完成后，执行 `docker-compose up -d` 即可启动容器。



如果修改了 `config.yaml` 内的端口号，`docker-compose.yaml` 里的端口号也需要保持一致。

执行 `docker-compose stop` 命令停止容器部署的 Dashboard。

14.2.3 管理 Dashboard 服务

Dashboard 使用脚本 `dashboard.service` 管理服务，包括启动、重启、停止和查看状态。

```
sudo <dashboard_path>/dashboard.service
[-v] [-h]
<start|restart|stop|status> <prometheus|webserver|exporter|gateway|all>
```

参数	说明
dashboard_path	Dashboard 安装路径。
-v	显示详细调试信息。
-h	显示帮助信息。
start	启动服务。
restart	重启服务。
stop	停止服务。
status	查看服务状态。
prometheus	管理 prometheus 服务。
webserver	管理 webserver 服务。
exporter	管理 exporter 服务。
gateway	管理 gateway 服务。
all	管理所有服务。

Note

查看 Dashboard 版本可以使用命令 `./dashboard.service -version`。

14.2.4 后续操作

连接 Dashboard

最后更新: May 4, 2023

14.3 连接 Dashboard

Dashboard 部署完成后，可以通过浏览器登录使用 Dashboard。

14.3.1 前提条件

- Dashboard 相关服务已经启动。详情请参见部署 Dashboard。
- 建议使用 Chrome 89 及以上的版本的 Chrome 浏览器，否则可能有兼容问题。

14.3.2 操作步骤

1. 确认 Dashboard 服务所在机器的 IP 地址，在浏览器中输入 <IP>:7003 打开登录页面。
2. 输入 NebulaGraph 数据库的账号和密码。
 - 如果 NebulaGraph 已经启用身份验证，用户可以使用已创建的账号连接 Dashboard。
 - 如果 NebulaGraph 未启用身份验证，用户只能使用默认用户 root 和任意密码连接 Dashboard。

有关如何启用身份验证请参见[身份验证](#)。
3. 单击登录。

最后更新: May 4, 2023

14.4 Dashboard 页面介绍

Dashboard 页面主要分为机器、服务、管理三个部分，本文将详细介绍这些界面。

14.4.1 页面概览

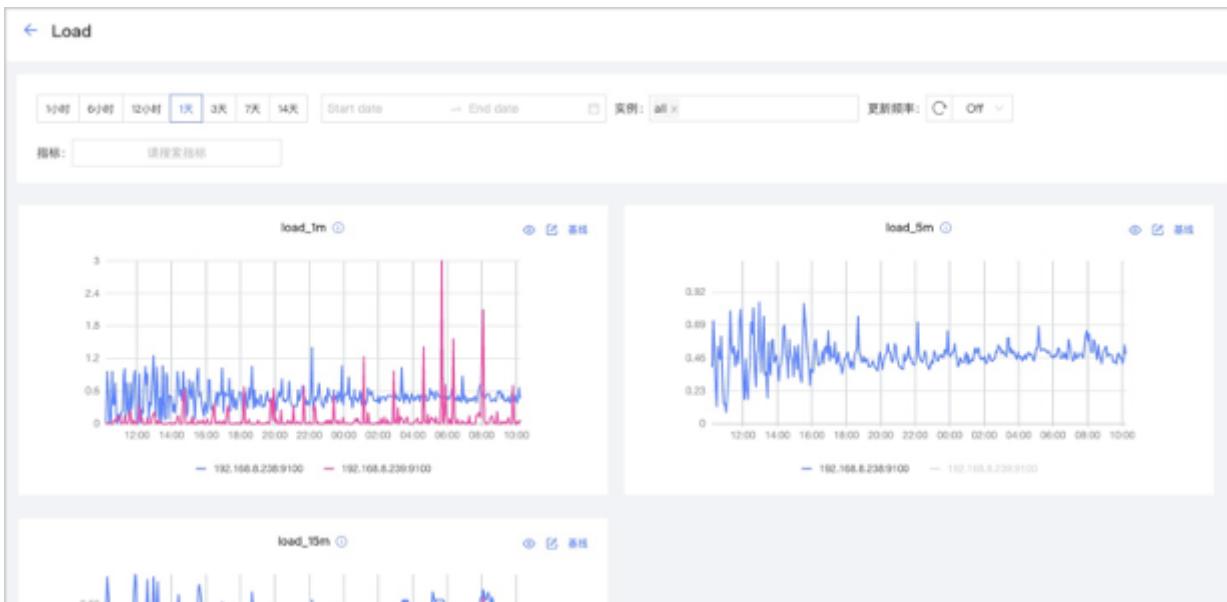


14.4.2 机器页面介绍

单击机器->概览进入机器概览页面。

用户可快速查看 CPU、Memory、Load、Disk 和 Network In/Out 变化情况。

- 默认最多可选择 14 天的监控数据进行查看，支持选择时间段，也可以快捷选择最近 1 小时、6 小时、12 小时、1 天、3 天、7 天和 14 天。
- 默认勾选显示所有实例的监控数据，可以在实例框内调整。
- 页面的监控数据默认不自动更新，可以调整更新频率让页面自动更新，也可以单击  按钮手动更新。
- 如果需要设置基线，作为参考标准线，可以单击模块右上角的  按钮。
- 如果需要查看某一项更详细的监控指标，可以单击模块右上角的  按钮，在示例中选择 Load 查看详情信息，如下图。



- 可以设置监控时间段、实例、更新频率和基线。
- 可以搜索和勾选指标。监控指标详情请参见[监控指标说明](#)。
- 可以暂时隐藏不需要查看的节点。
- 可以单击  按钮查看指标详情。

14.4.3 服务页面介绍

单击服务->概览进入服务概览页面。

用户可快速查看 Graph、Meta、Storage 服务的信息，右上角显示正常服务和异常服务的数量。

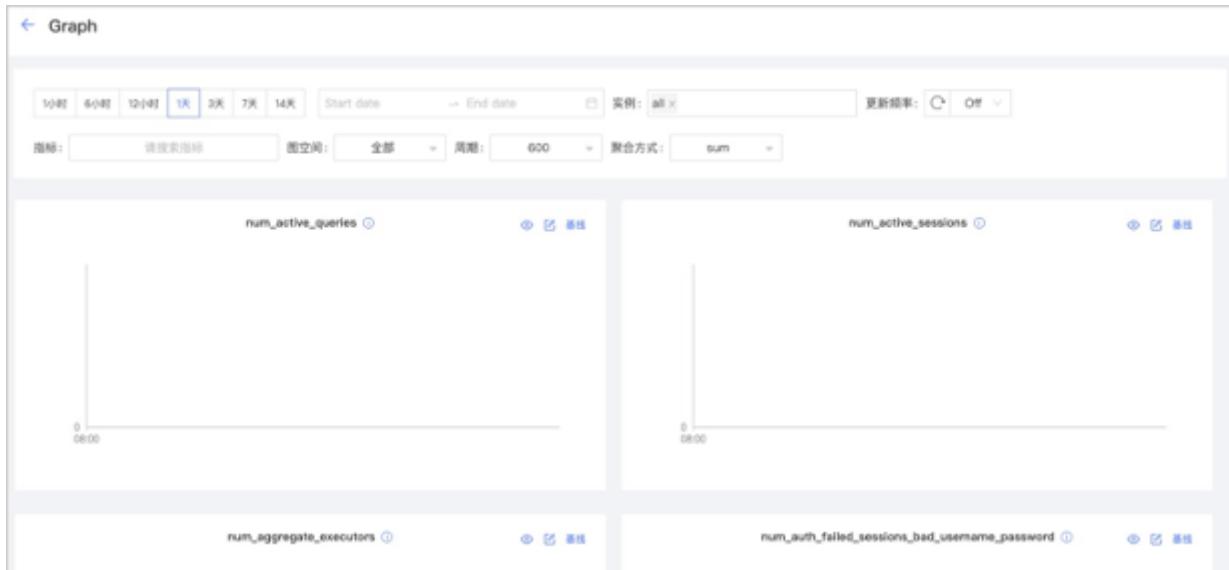
Note

服务监控页仅支持每种服务设置两个监控指标，可以单击模块内的设置按钮进行调整。

- 默认最多可选择 14 天的监控数据进行查看，支持选择时间段，也可以快捷选择最近 1 小时、6 小时、12 小时、1 天、3 天、7 天和 14 天。
- 默认勾选显示所有实例的监控数据，可以在实例框内调整。
- 页面的监控数据默认不自动更新，可以调整更新频率让页面自动更新，也可以单击  按钮手动更新。
- 可以查看集群内所有服务的状态。
-



如果需要查看某一项更详细的监控指标，可以单击模块右上角的  按钮，在示例中选择 Graph 查看详情信息，如下图。



- 可以设置监控时间段、实例、更新频率、周期、聚合方式和基线。

可以搜索和勾选指标。监控指标详情请参见[监控指标说明](#)。

可以暂时隐藏不需要查看的节点。

-



可以单击  按钮查看指标详情。

- Graph 服务支持一系列基于图空间的监控指标。详情参见下文图空间监控。

图空间监控

Caution

使用图空间指标前，用户需要在 Graph 服务中，设置 `enable_space_level_metrics` 为 `true`。具体操作，参见[Graph 服务配置](#)。

↑ 监控指标不兼容性

如果图空间的名称包括特殊字符，可能会有图空间监控指标数据不显示的问题。

服务监控页面还可以监控图空间级别指标。只有当图空间指标的行为被触发后，用户才能指定图空间查看对应的图空间指标的信息。图空间的监控指标记录不同图空间的数据。目前，只有 Graph 服务支持基于图空间的监控指标。

Dashboard 支持的图空间指标，参见[图空间级别监控指标](#)。

查询条件 X

周期: ▼

指标: ▼ (i)

图空间: ▼

聚合方式: ▼

基线:

取消 确认

14.4.4 管理页面介绍

信息总览

用户可以在信息总览页面查看NebulaGraph内核相关信息，包括 Storage 服务信息、Storage Leader 分布、NebulaGraph各个服务的版本信息及所在节点信息、分片的分布情况及详细信息。

集群管理 / 信息总览

[← 信息总览](#)

Storage Leader 分布



服务	Leader 数量	Leader 分布
192.168.8.129	0	No valid partition

[Balance Leader](#) [详情](#)

[版本](#) [详情](#)

Graph Service	
服务	版本
192.168.8.129:9669	3.1.0-ent

Storage Service	
服务	版本
192.168.8.129:9779	3.1.0-ent

Meta Service	
服务	版本
192.168.8.129:9559	3.1.0-ent

[请选择图空间](#)

Partition 分布

服务	分片数
	No Data
	No Data

[分片信息](#)

STORAGE LEADER 分布

显示 Leader 数量及 Leader 的分布。

- 单击右上角的 **Balance Leader** 按钮可以快速在NebulaGraph集群中均衡分布 Leader。关于 Leader 的详细信息，参见 [Storage 服务](#)。
- 单击右上角的详情，查看 Leader 分布的详细信息。

版本

显示NebulaGraph所有服务版本及服务地址。单击右上角的详情，查看更多信息。

服务信息

展示 Storage 服务的基本信息。参数说明如下：

参数	说明
Host	主机地址
Port	主机端口号
Status	主机状态
Git Info Sha	版本 Commit ID
Leader Count	Leader 总数
Partition Distribution	分片分布
Leader Distribution	Leader 分布

单击右上角的详情，查看更多信息。

PARTITION 分布

左上方选择指定图空间，查看图空间的分片分布情况。显示所有 Storage 服务的 IP 地址、端口，及对应 Storage 服务中的分片数量。

单击右上角的详情，查看更多信息。

分片信息

显示分片信息。用户需要在左上角选择图空间，查看分片信息。参数说明如下：

参数	说明
Partition ID	分片序号。
Leader	分片的 leader 副本的 IP 地址和端口。
Peers	分片所有副本的 IP 地址和端口。
Losts	分片的故障副本的 IP 地址和端口。

单击右上角的详情，查看更多信息，通过右上角的输入框，输入分片 ID，筛选展示的数据。

配置

展示服务的配置信息。暂不支持在线修改配置。

14.4.5 其他

在页面左下角，还可以进行如下操作：

- 退出登录
- 切换中英文
- 查看当前 Dashboard 版本
- 查看帮助信息
- 折叠侧边栏

14.5 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 86, in top-level template code
  File "docs-2.0/reuse/source-monitoring-metrics.md", line 154, in top-level template code
    {{ent.ent_begin}}
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

15. NebulaGraph Importer

15.1 NebulaGraph Importer

NebulaGraph Importer（简称 Importer）是一款 **NebulaGraph** 的 CSV 文件单机导入工具。Importer 可以读取本地的 CSV 文件，然后导入数据至NebulaGraph图数据库中。

15.1.1 适用场景

Importer 适用于将本地 CSV 文件的内容导入至NebulaGraph中。

15.1.2 优势

- 轻量快捷：不需要复杂环境即可使用，快速导入数据。
- 灵活筛选：通过配置文件可以实现对 CSV 文件数据的灵活筛选。

15.1.3 更新说明

[Release notes](#)

15.1.4 前提条件

在使用 NebulaGraph Importer 之前，请确保：

- 已部署NebulaGraph服务。目前有三种部署方式：
 - Docker Compose 部署
 - RPM/DEB 包安装
 - 源码编译安装
- NebulaGraph 中已创建 Schema，包括图空间、Tag 和 Edge type，或者通过参数 `clientSettings.postStart.commands` 设置。
- 运行 Importer 的机器已部署 Golang 环境。详情请参见 [Golang 环境搭建](#)。

15.1.5 操作步骤

配置 yaml 文件并准备好待导入的 CSV 文件，即可使用本工具向NebulaGraph批量写入数据。

下载二进制包运行

1. 在[Release](#)页面下载二进制包，并添加执行权限。
2. 启动服务。

```
$ ./<binary_package_name> --config <yaml_config_file_path>
```

源码编译运行

1. 克隆仓库。

```
$ git clone -b release-3.4 https://github.com/vesoft-inc/nebula-importer.git
```

Note

请使用正确的分支。 NebulaGraph 2.x 和 3.x 的 rpc 协议不同。

2. 进入目录 `nebula-importer`。

```
$ cd nebula-importer
```

3. 编译源码。

```
$ make build
```

4. 启动服务。

```
$ ./nebula-importer --config <yaml_config_file_path>
```

Note

`yaml` 配置文件说明请参见下文的配置文件说明。

无网络编译方式

如果服务器不能联网，建议在能联网的机器上将源码和各种依赖打包上传到对应的服务器上编译即可，操作步骤如下：

1. 克隆仓库。

```
$ git clone -b release-3.4 https://github.com/vesoft-inc/nebula-importer.git
```

2. 使用如下的命令下载并打包依赖的源码。

```
$ cd nebula-importer
$ go mod vendor
$ cd .. && tar -zcvf nebula-importer.tar.gz nebula-importer
```

3. 将压缩包上传到不能联网的服务器上。

4. 解压并编译。

```
$ tar -zxfv nebula-importer.tar.gz
$ cd nebula-importer
$ go build -mod vendor cmd/importer.go
```

Docker 方式运行

使用 Docker 可以不必在本地安装 Go 语言环境，只需要拉取 NebulaGraph Importer 的镜像，并将本地配置文件和 CSV 数据文件挂载到容器中。命令如下：

```
$ docker run --rm -ti \
--network=host \
-v <config_file>:<config_file> \
-v <csv_data_dir>:<csv_data_dir> \
vesoft/nebula-importer:<version>
--config <config_file>
```

- `<config_file>`：本地 `yaml` 配置文件的绝对路径。
- `<csv_data_dir>`：本地 CSV 数据文件的绝对路径。
- `<version>`：NebulaGraph 3.x 请填写 `v3`。

Note

建议使用相对路径。如果使用本地绝对路径，请检查路径映射到 Docker 中的路径。

15.1.6 配置文件说明

NebulaGraph Importer 通过 `nebula-importer/examples/v2/example.yaml` 配置文件来描述待导入文件信息、NebulaGraph服务器信息等。用户可以参考示例配置文件：[无表头配置/有表头配置](#)。下文将分类介绍配置文件内的字段。

Note

如果用户下载的是二进制包，请手动创建配置文件。

基本配置

示例配置如下：

```
version: v2
description: example
removeTempFiles: false
```

参数	默认值	是否必须	说明
<code>version</code>	v2	是	配置文件的版本。
<code>description</code>	example	否	配置文件的描述。
<code>removeTempFiles</code>	false	否	是否删除临时生成的日志和错误数据文件。

客户端配置

客户端配置存储客户端连接NebulaGraph相关的配置。

示例配置如下：

```
clientSettings:
  retry: 3
  concurrency: 10
  channelBufferSize: 128
  space: test
  connection:
    user: user
    password: password
    address: 192.168.11.13:9669,192.168.11.14:9669
  # # 只有 local_config 是 false 的情况下，才可以通过 UPDATE CONFIGS 更新配置
  # postStart:
  #   commands: |
  #     UPDATE CONFIGS storage:wal_ttl=3600;
  #     UPDATE CONFIGS storage:rocksdb_column_family_options = { disable_auto_compactions = true };
  #     afterPeriod: 8s
  # preStop:
  #   commands: |
```

```
# UPDATE CONFIGS storage:wal_ttl=86400;
# UPDATE CONFIGS storage:rocksdb_column_family_options = { disable_auto_compactions = false };
```

参数	默认值	是否必须	说明
clientSettings.retry	3	否	nGQL 语句执行失败的重试次数。
clientSettings.concurrency	10	否	NebulaGraph客户端并发数。
clientSettings.channelBufferSize	128	否	每个NebulaGraph客户端的缓存队列大小。
clientSettings.space	-	是	指定数据要导入的NebulaGraph图空间。不要同时导入多个空间，以免影响性能。
clientSettings.connection.user	-	是	NebulaGraph的用户名。
clientSettings.connection.password	-	是	NebulaGraph用户名对应的密码。
clientSettings.connection.address	-	是	所有 Graph 服务的地址和端口。
clientSettings.postStart.commands	-	否	配置连接NebulaGraph服务器之后，在插入数据之前执行的一些操作。
clientSettings.postStart.afterPeriod	-	否	执行上述 commands 命令后到执行插入数据命令之间的间隔，例如 8s。
clientSettings.preStop.commands	-	否	配置断开NebulaGraph服务器连接之前执行的一些操作。

文件配置

文件配置存储数据文件和日志的相关配置，以及 Schema 的具体信息。

文件和日志配置

示例配置如下：

```
workingDir: ./data/
logPath: ./err/test.log
files:
  - path: ./student.csv
    failDataPath: ./err/student
    batchSize: 128
    limit: 10
    inOrder: false
    type: csv
    CSV:
      withHeader: false
      withLabel: false
```

```
delimiter: ","
lazyQuotes: false
```

参数	默认值	是否必须	说明
workingDir	-	否	在多个目录包含具有相同文件结构的数据，使用此参数在多个目录之间切换。例如，下面配置的 path 和 failDataPath 的值会自动更改为 ./data/student 和 ./data/err/student。参数可以是绝对的或相对的。
logPath	-	否	导入过程中的错误等日志信息输出的文件路径。
files.path	-	是	数据文件的存放路径，如果使用相对路径，则会将路径和当前配置文件的目录拼接。可以使用星号 (*) 进行模糊匹配，导入多个名称相似的文件，但是文件的结构需要相同。
files.failDataPath	-	是	插入失败的数据文件存放路径，以便后面补写数据。
files.batchSize	128	否	单批次插入数据的语句数量。
files.limit	-	否	读取数据的行数限制。
files.inOrder	-	否	是否按顺序在文件中插入数据行。如果为 false，可以避免数据倾斜导致的导入速率降低。
files.type	-	是	文件类型。
files.csv.withHeader	false	是	是否有表头。详情请参见 关于 CSV 文件表头 。
files.csv.withLabel	false	是	是否有 LABEL。详情请参见 有表头配置说明 。
files.csv.delimiter	"","	是	指定 csv 文件的分隔符。只支持一个字符的字符串分隔符。使用特殊字符做分隔符时需要进行转义。例如当分隔符为十六进制 0x03 即 Ctrl+C 时，转义的写法为: "\x03" 或 "\u0003"。关于 yaml 格式特殊字符转义的细节请参见 更多 。
files.csv.lazyQuotes	false	否	LazyQuotes 设置为真时，一个引号可能会出现在非引号字段中，一个非双引号可能会出现在引号字段中。

SCHEMA 配置

Schema 配置描述当前数据文件的 Meta 信息，Schema 的类型分为点和边两类，可以同时配置多个点或边。

- 点配置

示例配置如下：

```
schema:
  type: vertex
  vertex:
    vid:
      index: 1
      concatItems: # "c1{index0}c2{index1}2"
      - "c1"
      - 0
      - c2
      - 1
      - "2"
    function: hash
    prefix: abc
  tags:
    - name: student
      props:
        - name: age
          type: int
          index: 2
        - name: name
          type: string
          index: 1
        - name: gender
          type: string
          defaultValue: "male"
        - name: phone
          type: string
          nullable: true
        - name: email
          type: string
```

```

  nullable: true
  nullValue: "__NULL__"
- name: address
  type: string
  nullable: true
  alternativeIndices:
    - 7
    - 8

```

参数	默认值	是否必须	说明
files.schema.type	-	是	Schema 的类型, 可选值为 vertex 和 edge 。
files.schema.vertex.vid.index	-	否	点 ID 对应 CSV 文件中列的序号。
files.schema.vertex.vid.concatItem	-	否	用于连接两个或多个数组, 连接项可以是 string 、 int 或 index 参数将不生效。
files.schema.vertex.vid.function	-	否	生成 VID 的函数。目前, 我们只支持 hash 函数。
files.schema.vertex.vid.type	-	否	点 ID 的数据类型, 可选值为 int 和 string 。
files.schema.vertex.vid.prefix	-	否	给原始vid 添加的前缀, 当同时指定了 function 时, 生
files.schema.vertex.tags.name	-	是	Tag 名称。
files.schema.vertex.tags.props.name	-	是	Tag 属性名称, 必须和NebulaGraph中的 Tag 属性一致。
files.schema.vertex.tags.props.type	-	是	属性数据类型, 支持 bool 、 int 、 float 、 double 、 geography(point) 、 geography(linestring) 和 geography(polygon) 。
files.schema.vertex.tags.props.index	-	否	属性对应 CSV 文件中列的序号。
files.schema.vertex.tags.props.nullable	false	否	属性是否可以为 NULL , 可选 true 或者 false 。
files.schema.vertex.tags.props.nullValue	""	否	nullable 设置为 true 时, 属性的值与 nullValue 相等则将该属性忽略。
files.schema.vertex.tags.props.alternativeIndices	-	否	当 nullable 为 false 时忽略。该属性根据索引顺序从 csv
files.schema.vertex.tags.props.defaultValue	-	否	当 nullable 为 false 时忽略。根据 index 和 alternativeIndices 获取的所有值为 nullValue 时设置默认值。

Note

CSV 文件中列的序号从 0 开始, 即第一列的序号为 0, 第二列的序号为 1。

- 边配置

示例配置如下:

```

schema:
  type: edge
  edge:
    name: follow
    srcID:
      index: 0
      function: hash
    dstID:
      index: 1
      function:
    rank:
      index: 2
    props:
      - name: grade

```

```
type: int
index: 3
```

参数	默认值	是否必须	说明
files.schema.type	-	是	Schema 的类型, 可选值为 vertex 和 edge。
files.schema.edge.name	-	是	Edge type 名称。
files.schema.edge.srcVID.index	-	否	边的起始点 ID 对应 CSV 文件中列的序号。
files.schema.edge.srcVID.function	-	否	生成 VID 的函数。目前, 我们只支持 hash 函数。
files.schema.edge.dstVID.index	-	否	边的目的点 ID 对应 CSV 文件中列的序号。
files.schema.edge.dstVID.function	-	否	生成 VID 的函数。目前, 我们只支持 hash 函数。
files.schema.edge.rank.index	-	否	边的 rank 值对应 CSV 文件中列的序号。
files.schema.edge.props.name	-	是	Edge type 属性名称, 必须和NebulaGraph中的 Edge type 属性一致。
files.schema.edge.props.type	-	是	属性类型, 支持 bool、int、float、double、timestamp、string 和 geo。
files.schema.edge.props.index	-	否	属性对应 CSV 文件中列的序号。

15.1.7 关于 CSV 文件表头 (header)

Importer 根据 CSV 文件有无表头, 需要对配置文件进行不同的设置, 相关示例和说明请参见:

- [无表头配置说明](#)
- [有表头配置说明](#)

15.1.8 视频

- [数据库导入工具——NebulaGraph Importer 简介 \(3 分 09 秒\)](#)

最后更新: May 4, 2023

15.2 有表头配置说明

对于有表头（header）的 CSV 文件，需要在配置文件里设置 `withHeader` 为 `true`，表示 CSV 文件中第一行为表头，表头内容具有特殊含义。



如果 CSV 文件中含有 `header`，Importer 就会按照 `header` 来解析每行数据的 Schema，并忽略 yaml 文件中的点或边设置。

15.2.1 示例文件

有表头的 CSV 文件示例如下：

- 点示例

`student_with_header.csv` 的示例数据：

```
:VID(string),student.name:string,student.age:int,student.gender:string
student100,Monica,16,female
student101,Mike,18,male
student102,Jane,17,female
```

第一列为点 ID，后面三列为属性 `name`、`age` 和 `gender`。

- 边示例

`follow_with_header.csv` 的示例数据：

```
:SRC_VID(string),:DST_VID(string),:RANK,follow.degree:double
student100,student101,0,92.5
student101,student100,1,85.6
student101,student102,2,93.2
student100,student102,1,96.2
```

前两列的数据分别为起始点 ID 和目的点 ID，第三列为 `rank`，第四列为属性 `degree`。

15.2.2 表头格式说明

表头通过一些关键词定义起始点、目的点、`rank` 以及一些特殊功能，说明如下：

- `:VID`（必填）：点 ID。需要用 `:VID(type)` 形式设置数据类型，例如 `:VID(string)` 或 `:VID(int)`。
- `:SRC_VID`（必填）：边的起始点 ID。需要用 `:SRC_VID(type)` 形式设置数据类型。
- `:DST_VID`（必填）：边的目的点 ID。需要用 `:DST_VID(type)` 形式设置数据类型。
- `:RANK`（可选）：边的 `rank` 值。
- `:IGNORE`（可选）：插入数据时忽略这一列。
- `:LABEL`（可选）：表示对该行进行插入（+）或删除（-）操作。必须为第一列。例如：

```
:LABEL,
+,
```



除了 `:LABEL` 列之外的所有列都可以按任何顺序排序，因此针对较大的 CSV 文件，用户可以灵活地设置 `header` 来选择需要的列。

对于 Tag 或 Edge type 的属性，格式为 <tag_name/edge_name>.〈prop_name〉:〈prop_type〉，说明如下：

- <tag_name/edge_name>：Tag 或者 Edge type 的名称。
- 〈prop_name〉：属性名称。
- 〈prop_type〉：属性类型。支持 bool、int、float、double、timestamp 和 string，默认为 string。

例如 student.name:string 、 follow.degree:double 。

15.2.3 配置示例

```
version: v2
description: example
# 是否删除临时生成的日志和错误数据文件。
removeTempFiles: false
clientSettings:
  # nGQL 语句执行失败的重试次数。
  retry: 3
  # NebulaGraph客户端并发数。
  concurrency: 10
  # 每个NebulaGraph客户端的缓存队列大小。
  channelBufferSize: 128
  # 指定数据要导入的NebulaGraph图空间。
  space: student
  # 连接信息。
  connection:
    user: root
    password: nebula
    address: 192.168.11.13:9669
postStart:
  # 配置连接NebulaGraph服务器之后，在插入数据之前执行的一些操作。
  commands: |
    DROP SPACE IF EXISTS student;
    CREATE SPACE IF NOT EXISTS student(partition_num=5, replica_factor=1, vid_type=FIXED_STRING(20));
    USE student;
    CREATE TAG student(name string, age int, gender string);
    CREATE EDGE follow(degree int);

  # 执行上述命令后到执行插入数据命令之间的间隔。
  afterPeriod: 15s
preStop:
  # 配置断开NebulaGraph服务器连接之前执行的一些操作。
  commands: |
# 错误等日志信息输出的文件路径。
logPath: ./err/test.log
# CSV 文件相关设置。
files:
  # 数据文件的存放路径，如果使用相对路径，则会将路径和当前配置文件的目录拼接。本示例第一个数据文件为点的数据。
  - path: ./student_with_header.csv
  # 插入失败的数据文件存放路径，以便后面补写数据。
  failDataPath: ./err/studenterr
  # 单批次插入数据的语句数量。
  batchSize: 10
  # 读取数据的行数限制。
  limit: 10
  # 是否按顺序在文件中插入数据行。如果为 false，可以避免数据倾斜导致的导入速率降低。
  inOrder: true
  # 文件类型，当前仅支持 csv。
  type: csv
  csv:
    # 是否有表头。
    withHeader: true
    # 是否有 LABEL。
    withLabel: false
    # 指定 csv 文件的分隔符。只支持一个字符的字符串分隔符。
    delimiter: ","

```

```
schema:  
  # Schema 的类型，可选值为 vertex 和 edge。  
  type: vertex  
  
  # 本示例第二个数据文件为边的数据。  
  - path: ./follow_with_header.csv  
    failDataPath: ./err/followerr  
    batchSize: 10  
    limit: 10  
    inOrder: true  
    type: csv  
    CSV:  
      withHeader: true  
      withLabel: false  
    schema:  
      # Schema 的类型为 edge。  
      type: edge  
      edge:  
        # Edge type 名称。  
        name: follow  
  
        # 是否包含 rank。  
        withRanking: true
```

Note

点 ID 的数据类型需要和 clientSettings.postStart.commands 中的创建图空间语句的数据类型一致。

最后更新: May 4, 2023

15.3 无表头配置说明

对于无表头（header）的 CSV 文件，需要在配置文件里设置 `withHeader` 为 `false`，表示 CSV 文件中只含有数据（不含第一行表头），同时可能还需要设置数据类型、对应的列等。

15.3.1 示例文件

无表头的 CSV 文件示例如下：

- 点示例

`student_without_header.csv` 的示例数据：

```
student100,Monica,16,female
student101,Mike,18,male
student102,Jane,17,female
```

第一列为点 ID，后面三列为属性 `name`、`age` 和 `gender`。

- 边示例

`follow_without_header.csv` 的示例数据：

```
student100,student101,0,92.5
student101,student100,1,85.6
student101,student102,2,93.2
student100,student102,1,96.2
```

前两列的数据分别为起始点 ID 和目的点 ID，第三列为 `rank`，第四列为属性 `degree`。

15.3.2 配置示例

```
version: v2
description: example
# 是否删除临时生成的日志和错误数据文件。
removeTempFiles: false
clientSettings:
  # nGQL 语句执行失败的重试次数。
  retry: 3
  # NebulaGraph客户端并发数。
  concurrency: 10
  # 每个NebulaGraph客户端的缓存队列大小。
  channelBufferSize: 128
  # 指定数据要导入的NebulaGraph图空间。
  space: student
  # 连接信息。
  connection:
    user: root
    password: nebula
    address: 192.168.11.13:9669
postStart:
  # 配置连接NebulaGraph服务器之后，在插入数据之前执行的一些操作。
  commands: |
    DROP SPACE IF EXISTS student;
    CREATE SPACE IF NOT EXISTS student(partition_num=5, replica_factor=1, vid_type=FIXED_STRING(20));
    USE student;
    CREATE TAG student(name string, age int, gender string);
    CREATE EDGE follow(degree int);
  # 执行上述命令后到执行插入数据命令之间的间隔。
  afterPeriod: 15s
preStop:
  # 配置断开NebulaGraph服务器连接之前执行的一些操作。
  commands: |
# 错误等日志信息输出的文件路径。
```

```

logPath: ./err/test.log

# CSV 文件相关设置。
files:
  # 数据文件的存放路径，如果使用相对路径，则会将路径和当前配置文件的目录拼接。本示例第一个数据文件为点的数据。
  - path: ./student_without_header.csv

  # 插入失败的数据文件存放路径，以便后面补写数据。
  failDataPath: ./err/studenterr

  # 单批次插入数据的语句数量。
  batchSize: 10

  # 读取数据的行数限制。
  limit: 10

  # 是否按顺序在文件中插入数据行。如果为 false，可以避免数据倾斜导致的导入速率降低。
  inOrder: true

  # 文件类型，当前仅支持 csv。
  type: csv

csv:
  # 是否有表头。
  withHeader: false

  # 是否有 LABEL。
  withLabel: false

  # 指定 csv 文件的分隔符。只支持一个字符的字符串分隔符。
  delimiter: ","

schema:
  # Schema 的类型，可选值为 vertex 和 edge。
  type: vertex

vertex:
  # 点 ID 设置。
  vid:
    # 点 ID 对应 CSV 文件中列的序号。CSV 文件中列的序号从 0 开始。
    index: 0

    # 点 ID 的数据类型，可选值为 int 和 string，分别对应NebulaGraph中的 INT64 和 FIXED_STRING。
    type: string

  # Tag 设置。
  tags:
    # Tag 名称。
    - name: student

    # Tag 内的属性设置。
    props:
      # 属性名称。
      - name: name

      # 属性数据类型。
      type: string

      # 属性对应 CSV 文件中列的序号。
      index: 1

      - name: age
        type: int
        index: 2
      - name: gender
        type: string
        index: 3

  # 本示例第二个数据文件为边的数据。
  - path: ./follow_without_header.csv
  failDataPath: ./err/followerr
  batchSize: 10
  limit: 10
  inOrder: true
  type: csv
  csv:
    withHeader: false
    withLabel: false
  schema:
    # Schema 的类型为 edge。
    type: edge
    edge:
      # Edge type 名称。
      name: follow

      # 是否包含 rank。
      withRanking: true

      # 起始点 ID 设置。
      srcVID:
        # 数据类型。
        type: string

```

```
# 起始点 ID 对应 CSV 文件中列的序号。  
index: 0  
  
# 目的点 ID 设置。  
dstVID:  
  type: string  
  index: 1  
  
# rank 设置。  
rank:  
  # rank 值对应 CSV 文件中列的序号。如果没有设置 index，请务必在第三列设置 rank 的值。之后的列依次设置各属性。  
  index: 2  
  
# Edge type 内的属性设置。  
props:  
  # 属性名称。  
  - name: degree  
  
  # 属性数据类型。  
  type: double  
  
  # 属性对应 CSV 文件中列的序号。  
  index: 3
```

Note

- CSV 文件中列的序号从 0 开始，即第一列的序号为 0，第二列的序号为 1。
- 点 ID 的数据类型需要和 `clientSettings.postStart.commands` 中的创建图空间语句的数据类型一致。
- 如果没有设置 `index` 字段指定列的序号，CSV 文件必须遵守如下规则：
 - 在点数据文件中，第一列必须为点 ID，后面的列为属性，且需要和配置文件内的顺序一一对应。
 - 在边数据文件中，第一列必须为起始点 ID，第二列必须为目的点 ID，如果 `withRanking` 为 `true`，第三列必须为 `rank` 值，后面的列为属性，且需要和配置文件内的顺序一一对应。

最后更新: May 4, 2023

16. NebulaGraph Exchange

16.1 认识 NebulaGraph Exchange

16.1.1 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
File "<template>", line 117, in top-level template code
File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

16.1.2 使用限制

本文描述 Exchange 3.4.0 的一些使用限制。

使用环境

Exchange 3.x 支持以下操作系统:

- CentOS 7
- macOS

软件依赖

为保证 Exchange 正常工作, 请确认机器上已经安装如下软件:

- Java 1.8 版本
- Scala 2.10.7、2.11.12 或 2.12.10 版本
- Apache Spark。使用 Exchange 从不同数据源导出数据对 Spark 版本的要求如下:

Note

使用 Exchange 时, 需根据 Spark 版本选择相应的 JAR 文件。例如, 当 Spark 版本为 2.4 时, 选择 `nebula-exchange_spark_2.4-3.4.0.jar`。

数据源	Spark 2.2	Spark 2.4	Spark 3
CSV 文件	支持	支持	支持
JSON 文件	支持	支持	支持
ORC 文件	支持	支持	支持
Parquet 文件	支持	支持	支持
HBase	支持	支持	支持
MySQL	支持	支持	支持
PostgreSQL	支持	支持	支持
Oracle	支持	支持	支持
ClickHouse	支持	支持	支持
Neo4j	不支持	支持	不支持
Hive	支持	支持	支持
MaxCompute	不支持	支持	不支持
Pulsar	不支持	支持	未测试
Kafka	不支持	支持	未测试
NebulaGraph	不支持	支持	不支持

在以下使用场景, 还需要部署 Hadoop Distributed File System (HDFS):

- 迁移 HDFS 的数据
- 生成 SST 文件

最后更新: May 4, 2023

16.2 获取 NebulaGraph Exchange

本文介绍如何获取 NebulaGraph Exchange 的 JAR 文件。

16.2.1 直接下载 JAR 文件

社区版 Exchange 的 JAR 文件可以直接[下载](#)。

要下载企业版 Exchange，需先[获取NebulaGraph企业版套餐](#)。

16.2.2 编译源代码获取 JAR 文件

社区版 Exchange 的 JAR 文件还可以通过编译源代码获取。下文介绍如何编译 Exchange 源代码。



企业版 Exchange 仅能在NebulaGraph企业版套餐中获取。

前提条件

- 安装 [Maven](#)。
- 根据数据源安装需要的 [Spark](#) 版本，从各数据源导出数据支持的 [Spark](#) 版本参见[软件依赖](#)。

操作步骤

- 在根目录克隆仓库 `nebula-exchange`。

```
git clone -b release-3.4 https://github.com/vesoft-inc/nebula-exchange.git
```

- 切换到目录 `nebula-exchange`。

```
cd nebula-exchange
```

- 根据 Exchange 使用环境中的 [Spark](#) 版本打包 Exchange。

- [Spark 2.2:](#)

```
mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true \
-pl nebula-exchange_spark_2.2 -am -Pscala-2.11 -Pspark-2.2
```

- [Spark 2.4:](#)

```
mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true \
-pl nebula-exchange_spark_2.4 -am -Pscala-2.11 -Pspark-2.4
```

- [Spark 3.0:](#)

```
mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true \
-pl nebula-exchange_spark_3.0 -am -Pscala-2.12 -Pspark-3.0
```

编译成功后，可以在 `nebula-exchange_spark_x.x/target/` 目录里找到 `nebula-exchange_spark_x.x-release-3.4.jar` 文件。`x.x` 代表 [Spark](#) 版本，例如 `2.4`。



JAR 文件版本号会因 NebulaGraph Java Client 的发布版本而变化。用户可以在 [Releases 页面](#)查看最新版本。

迁移数据时，用户可以参考配置文件 `target/classes/application.conf`。

下载依赖包失败

如果编译时下载依赖包失败：

- 检查网络设置，确认网络正常。
- 修改 Maven 安装目录下 `libexec/conf/settings.xml` 文件的 `mirror` 部分：

```
<mirror>
<id>alimaven</id>
<mirrorOf>central</mirrorOf>
<name>aliyun maven</name>
<url>http://maven.aliyun.com/nexus/content/repositories/central/</url>
</mirror>
```

最后更新: May 4, 2023

16.3 参数说明

16.3.1 导入命令参数

完成配置文件修改后，可以运行以下命令将指定来源的数据导入NebulaGraph数据库。

- 首次导入

```
<spark_install_path>/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-2.x.y.jar_path> -c <application.conf_path>
```

- 导入 reload 文件

如果首次导入时有一些数据导入失败，会将导入失败的数据存入 `reload` 文件，可以用参数 `-r` 尝试导入 `reload` 文件。

```
<spark_install_path>/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-2.x.y.jar_path> -c <application.conf_path> -r "<reload_file_path>"
```



JAR 文件版本号以实际编译得到的 JAR 文件名称为准。



如果使用 `yarn-cluster` 模式提交任务，请参考如下示例，尤其是示例中的两个 `--conf`。

```
$SPARK_HOME/bin/spark-submit --master yarn-cluster \
--class com.vesoft.nebula.exchange.Exchange \
--files application.conf \
--conf spark.driver.extraClassPath=./ \
--conf spark.executor.extraClassPath=./ \
nebula-exchange-3.4.0.jar \
-c application.conf
```

下表列出了命令的相关参数。

参数	是否必需	默认值	说明
<code>--class</code>	是	无	指定驱动的主类。
<code>--master</code>	是	无	指定 Spark 集群中 master 进程的 URL。详情请参见 master-urls 。
<code>-c / --config</code>	是	无	指定配置文件的路径。
<code>-h / --hive</code>	否	<code>false</code>	添加这个参数表示支持从 Hive 中导入数据。
<code>-D / --dry</code>	否	<code>false</code>	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 <code>tags</code> 和 <code>edges</code> 的配置项是否正确。正式导入数据时不能添加这个参数。
<code>-r / --reload</code>	否	无	指定需要重新加载的 <code>reload</code> 文件路径。

更多 Spark 的参数配置说明请参见 [Spark Configuration](#)。

最后更新: May 4, 2023

16.3.2 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 219, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

16.4 使用 NebulaGraph Exchange

16.4.1 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 或本地的 CSV 文件数据导入NebulaGraph。

如果要向NebulaGraph导入本地 CSV 文件, 请参见 [NebulaGraph Importer](#)。

数据集

本文以 [basketballplayer](#) 数据集为例。

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
- NebulaGraph: 3.4.1。使用 [Docker Compose](#) 部署。

前提条件

开始导入数据之前, 用户需要确认以下信息:

- 已经[安装部署NebulaGraph](#) 并获取如下信息:
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息, 包括 Tag 和 Edge type 的名称、属性等。
- 如果文件存储在 HDFS 上, 需要确认 Hadoop 服务运行正常。
- 如果文件存储在本地且NebulaGraph是集群架构, 需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析 CSV 文件中的数据, 按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 使用NebulaGraphConsole 创建一个图空间 **basketballplayer**, 并创建一个 Schema, 如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息, 请参见[快速开始](#)。

步骤 2: 处理 CSV 文件

确认以下信息:

- 处理 CSV 文件以满足 Schema 的要求。



Exchange 支持上传有表头或者无表头的 CSV 文件。

- 获取 CSV 文件存储路径。

步骤 3: 修改配置文件

编译 Exchange 后, 复制 target/classes/application.conf 文件设置 CSV 数据源相关的配置。在本示例中, 复制的文件名为 csv_application.conf。各个配置项的详细说明请参见[配置说明](#)。

```
{
# Spark 相关配置
spark: {
  app: {
    name: NebulaGraph Exchange 3.4.0
  }
  driver: {
    cores: 1
    maxResultSize: 1G
  }
  executor: {
    memory: 1G
  }
}
cores: {
```

```

        max: 16
    }

}

# NebulaGraph 相关配置
nebula: {
    address: {
        # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
        # 如果有多台服务器，地址之间用英文逗号 (,) 分隔。
        # 格式："ip1:port","ip2:port","ip3:port"
        graph: ["127.0.0.1:9669"]
        #任意一个 Meta 服务的地址。
        meta: ["127.0.0.1:9559"]
    }

    # 指定拥有 NebulaGraph 写权限的用户名和密码。
    user: root
    pswd: nebula

    # 指定图空间名称。
    space: basketballplayer
    connection: {
        timeout: 3000
        retry: 3
    }
    execution: {
        retry: 3
    }
    error: {
        max: 32
        output: /tmp/errors
    }
    rate: {
        limit: 1024
        timeout: 1000
    }
}

# 处理点
tags: [
    # 设置 Tag player 相关信息。
    {
        # 指定 NebulaGraph 中定义的 Tag 名称。
        name: player
        type: {
            # 指定数据源，使用 CSV。
            source: csv

            # 指定如何将点数据导入 NebulaGraph: Client 或 SST。
            sink: client
        }

        # 指定 CSV 文件的路径。
        # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
        # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.csv"。
        path: "hdfs://192.168.*.*:9000/data/vertex_player.csv"

        # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
        # 如果 CSV 文件有表头，则使用实际的列名。
        fields: [_c1, _c2]

        # 指定 NebulaGraph 中定义的属性名称。
        # fields 与 nebula.fields 的顺序必须一一对应。
        nebula.fields: [age, name]

        # 指定一个列作为 VID 的源。
        # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
        # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
        vertex: {
            field: _c0
            # policy:hash
        }

        # 指定的分隔符。默认值为英文逗号 (,)。
        separator: ","

        # 如果 CSV 文件有表头，请将 header 设置为 true。
        # 如果 CSV 文件没有表头，请将 header 设置为 false。默认值为 false。
        header: false

        # 指定单批次写入 NebulaGraph 的最大点数量。
        batch: 256
    }

    # 设置 Tag team 相关信息。
    {
        # 指定 NebulaGraph 中定义的 Tag 名称。
        name: team
        type: {
            # 指定数据源，使用 CSV。

```

```

source: csv

# 指定如何将点数据导入NebulaGraph : Client 或 SST。
sink: client
}

# 指定 CSV 文件的路径。
# 如果文件存储在 HDFS 上, 用双引号括起路径, 以 hdfs://开头, 例如"dfs://ip:port/xx/xx"。
# 如果文件存储在本地, 用双引号括起路径, 以 file://开头, 例如"file:///tmp/xx.csv"。
path: "hdfs://192.168.*.*:9000/data/vertex_team.csv"

# 如果 CSV 文件没有表头, 使用 [_c0, _c1, _c2, ..., _cn] 表示其表头, 并将列指示为属性值的源。
# 如果 CSV 文件有表头, 则使用实际的列名。
fields: [_c1]

# 指定 NebulaGraph 中定义的属性名称。
# fields 与 nebula.fields 的顺序必须一一对应。
nebula.fields: [name]

# 指定一个列作为 VID 的源。
# vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
# 目前, NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
vertex: {
    field: _c0
    # policy:hash
}

# 指定的分隔符。默认值为英文逗号 (,)。
separator: ","

# 如果 CSV 文件有表头, 请将 header 设置为 true。
# 如果 CSV 文件没有表头, 请将 header 设置为 false。默认值为 false。
header: false

# 指定单批次写入 NebulaGraph 的最大点数量。
batch: 256

# 指定 Spark 分片数量。
partition: 32
}

# 如果需要添加更多点, 请参考前面的配置进行添加。
]

# 处理边
edges: [
    # 设置 Edge type follow 相关信息。
    {
        # 指定 NebulaGraph 中定义的 Edge type 名称。
        name: follow
        type: {
            # 指定数据源, 使用 CSV。
            source: csv

            # 指定如何将点数据导入NebulaGraph : Client 或 SST。
            sink: client
        }

        # 指定 CSV 文件的路径。
        # 如果文件存储在 HDFS 上, 用双引号括起路径, 以 hdfs://开头, 例如"dfs://ip:port/xx/xx"。
        # 如果文件存储在本地, 用双引号括起路径, 例如"file:///tmp/xx.csv"。
        path: "hdfs://192.168.*.*:9000/data/edge_follow.csv"

        # 如果 CSV 文件没有表头, 使用 [_c0, _c1, _c2, ..., _cn] 表示其表头, 并将列指示为属性值的源。
        # 如果 CSV 文件有表头, 则使用实际的列名。
        fields: [_c2]

        # 指定 NebulaGraph 中定义的属性名称。
        # fields 与 nebula.fields 的顺序必须一一对应。
        nebula.fields: [degree]

        # 指定一个列作为起始点和目的点的源。
        # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
        # 目前, NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
        source: {
            field: _c0
        }
        target: {
            field: _c1
        }

        # 指定的分隔符。默认值为英文逗号 (,)。
        separator: ","
    }

    # 指定一个列作为 rank 的源 (可选)。
    #ranking: rank

    # 如果 CSV 文件有表头, 请将 header 设置为 true。
    # 如果 CSV 文件没有表头, 请将 header 设置为 false。默认值为 false。
    header: false

    # 指定单批次写入 NebulaGraph 的最大边数量。
    batch: 256
]

```

```

# 指定 Spark 分片数量。
partition: 32
}

# 设置 Edge type serve 相关信息。
{
  # 指定 NebulaGraph 中定义的 Edge type 名称。
  name: serve
  type: {
    # 指定数据源，使用 CSV。
    source: csv

    # 指定如何将点数据导入NebulaGraph : Client 或 SST。
    sink: client
  }

  # 指定 CSV 文件的路径。
  # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"dfs://ip:port/xx/xx"。
  # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.csv"。
  path: "hdfs://192.168.*.*:9000/data/edge_serve.csv"

  # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
  # 如果 CSV 文件有表头，则使用实际的列名。
  fields: [_c2,_c3]

  # 指定NebulaGraph中定义的属性名称。
  # fields 与 nebula.fields 的顺序必须一一对应。
  nebula.fields: [start_year, end_year]

  # 指定一个列作为起始点和目的点的源。
  # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
  # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
  source: {
    field: _c0
  }
  target: {
    field: _c1
  }

  # 指定的分隔符。默认值为英文逗号 (,) 。
  separator: ","

  # 指定一个列作为 rank 的源（可选）。
  #ranking: _c5

  # 如果 CSV 文件有表头，请将 header 设置为 true。
  # 如果 CSV 文件没有表头，请将 header 设置为 false。默认值为 false。
  header: false

  # 指定单批次写入 NebulaGraph 的最大边数量。
  batch: 256

  # 指定 Spark 分片数量。
  partition: 32
}

]

# 如果需要添加更多边，请参考前面的配置进行添加。
}

```

步骤 4: 向NEBULAGRAPH导入数据

运行如下命令将 CSV 文件数据导入到NebulaGraph中。关于参数的说明，请参见[导入命令参数](#)。

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <csv_application.conf_path>
```



JAR 包有两种获取方式：[自行编译](#)或者从 maven 仓库下载。

示例:

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/csv_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 5: （可选）验证数据

用户可以在NebulaGraph客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 6: (如有) 在NEBULAGRAPH中重建索引

导入数据后, 用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.2 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 或本地的 JSON 文件数据导入 NebulaGraph。

数据集

本文以 basketballplayer 数据集为例。部分示例数据如下：

- player

```
{"id": "player100", "age": 42, "name": "Tim Duncan"}  
{"id": "player101", "age": 36, "name": "Tony Parker"}  
{"id": "player102", "age": 33, "name": "LaMarcus Aldridge"}  
{"id": "player103", "age": 32, "name": "Rudy Gay"}  
...
```

- team

```
{"id": "team200", "name": "Warriors"}  
{"id": "team201", "name": "Nuggets"}  
...
```

- follow

```
{"src": "player100", "dst": "player101", "degree": 95}  
{"src": "player101", "dst": "player102", "degree": 90}  
...
```

- serve

```
{"src": "player100", "dst": "team204", "start_year": 1997, "end_year": 2016}  
{"src": "player101", "dst": "team204", "start_year": 1999, "end_year": 2018}  
...
```

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
- NebulaGraph： 3.4.1。使用 [Docker Compose 部署](#)。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 如果文件存储在 HDFS 上，需要确认 Hadoop 服务运行正常。
- 如果文件存储在本地且NebulaGraph是集群架构，需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析文件中的数据，按以下步骤在NebulaGraph中创建 Schema：

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 使用 NebulaGraph Console 创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2: 处理 JSON 文件

确认以下信息：

- 处理 JSON 文件以满足 Schema 的要求。
- 获取 JSON 文件存储路径。

步骤 3. 修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 JSON 数据源相关的配置。在本示例中，复制的文件名为 json_application.conf 。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    executor: {
      memory:1G
    }
  }

  cores: {
    max: 16
  }
}

# NebulaGraph 相关配置
nebula: {
  address: {
    # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
    # 如果有多台服务器，地址之间用英文逗号 (,) 分隔。
    # 格式："ip1:port","ip2:port","ip3:port"
    graph: ["127.0.0.1:9669"]
    #任意一个 Meta 服务的地址。
    #如果您将 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
    meta: ["127.0.0.1:9559"]
  }
}

# 指定拥有 NebulaGraph 写权限的用户名和密码。
user: root
pswd: nebula

# 指定图空间名称。
space: basketballplayer
connection: {
  timeout: 3000
  retry: 3
}
execution: {
  retry: 3
}
error: {
  max: 32
  output: /tmp/errors
}
rate: {
  limit: 1024
  timeout: 1000
}
}

# 处理点
tags: [
  # 设置 Tag player 相关信息。
  {
    # 指定 NebulaGraph 中定义的 Tag 名称。
    name: player
    type: {
      # 指定数据源，使用 JSON。
      source: json
    }
    # 指定如何将点数据导入NebulaGraph：Client 或 SST。
    sink: client
  }
]

# 指定 JSON 文件的路径。
# 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
# 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.json"。
path: "hdfs://192.168.*.*:9000/data/vertex_player.json"

# 在 fields 里指定 JSON 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
# 如果需要指定多个值，用英文逗号 (,) 隔开。
fields: [age,name]

# 指定 NebulaGraph 中定义的属性名称。
# fields 与 nebula.fields 的顺序必须一一对应。
nebula.fields: [age, name]

# 指定一个列作为 VID 的源。
# vertex 的值必须与 JSON 文件中的字段保持一致。
# 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
vertex: {
  field:id
}
```

```

# 指定单批次写入 NebulaGraph 的最大点数量。
batch: 256

# 指定 Spark 分片数量。
partition: 32
}

# 设置 Tag team 相关信息。
{
    # 指定 NebulaGraph 中定义的 Tag 名称。
    name: team
    type: {
        # 指定数据源，使用 JSON。
        source: json

        # 指定如何将点数据导入NebulaGraph：Client 或 SST。
        sink: client
    }

    # 指定 JSON 文件的路径。
    # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
    # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.json"。
    path: "hdfs://192.168.*.*:9000/data/vertex_team.json"

    # 在 fields 里指定 JSON 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
    # 如果需要指定多个值，用英文逗号 (,) 隔开。
    fields: [name]

    # 指定 NebulaGraph 中定义的属性名称。
    # fields 与 nebula.fields 的顺序必须一一对应。
    nebula.fields: [name]

    # 指定一个列作为 VID 的源。
    # vertex 的值必须与 JSON 文件中的字段保持一致。
    # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
    vertex: {
        field:id
    }

    # 指定单批次写入 NebulaGraph 的最大点数量。
    batch: 256

    # 指定 Spark 分片数量。
    partition: 32
}

# 如果需要添加更多点，请参考前面的配置进行添加。
]

# 处理边
edges: [
    # 设置 Edge type follow 相关信息。
    {
        # 指定 NebulaGraph 中定义的 Edge type 名称。
        name: follow
        type: {
            # 指定数据源，使用 JSON。
            source: json

            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }

        # 指定 JSON 文件的路径。
        # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
        # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.json"。
        path: "hdfs://192.168.*.*:9000/data/edge_follow.json"

        # 在 fields 里指定 JSON 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
        # 如果需要指定多个值，用英文逗号 (,) 隔开。
        fields: [degree]

        # 指定 NebulaGraph 中定义的属性名称。
        # fields 与 nebula.fields 的顺序必须一一对应。
        nebula.fields: [degree]

        # 指定一个列作为起始点和目的点的源。
        # vertex 的值必须与 JSON 文件中的字段保持一致。
        # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
        source: {
            field: src
        }

        target: {
            field: dst
        }

        # 指定一个列作为 rank 的源（可选）。
        #ranking: rank

        # 指定单批次写入 NebulaGraph 的最大边数量。
        batch: 256

        # 指定 Spark 分片数量。
        partition: 32
    }
]

```

```

}

# 设置 Edge type serve 相关信息。
{
  # 指定 NebulaGraph 中定义的 Edge type 名称。
  name: serve
  type: {
    # 指定数据源，使用 JSON。
    source: json
  }

  # 指定如何将点数据导入NebulaGraph：Client 或 SST。
  sink: client
}

# 指定 JSON 文件的路径。
# 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
# 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.json"。
path: "hdfs://192.168.*.*:9000/data/edge_serve.json"

# 在 fields 里指定 JSON 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
# 如果需要指定多个值，用英文逗号 (,) 隔开。
fields: [start_year,end_year]

# 指定 NebulaGraph 中定义的属性名称。
# fields 与 nebula.fields 的顺序必须一一对应。
nebula.fields: [start_year, end_year]

# 指定一个列作为起始点和目的点的源。
# vertex 的值必须与 JSON 文件中的字段保持一致。
# 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
source: {
  field: src
}
target: {
  field: dst
}

# 指定一个列作为 rank 的源（可选）。
#ranking: _c5

# 指定单批次写入 NebulaGraph 的最大边数量。
batch: 256

# 指定 Spark 分片数量。
partition: 32
}

]

# 如果需要添加更多边，请参考前面的配置进行添加。
}
}

```

步骤 4: 向NEBULAGRAPH导入数据

运行如下命令将 JSON 文件数据导入到NebulaGraph中。关于参数的说明，请参见[导入命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <json_application.conf_path>
```

Note

JAR 包有两种获取方式：[自行编译](#)或者从 maven 仓库下载。

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/json_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 5: （可选）验证数据

用户可以在NebulaGraph客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 6: （如有）在NEBULAGRAPH中重建索引

导入数据后，用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.3 导入 ORC 文件数据

本文以一个示例说明如何使用 Exchange 将存储在 HDFS 或本地的 ORC 文件数据导入 NebulaGraph。

如果要向NebulaGraph导入本地 ORC 文件, 请参见 [NebulaGraph Importer](#)。

数据集

本文以 [basketballplayer](#) 数据集为例。

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
- NebulaGraph: 3.4.1。使用 [Docker Compose](#) 部署。

前提条件

开始导入数据之前, 用户需要确认以下信息:

- 已经[安装部署NebulaGraph](#) 并获取如下信息:
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息, 包括 Tag 和 Edge type 的名称、属性等。
- 如果文件存储在 HDFS 上, 需要确认 Hadoop 服务运行正常。
- 如果文件存储在本地且NebulaGraph是集群架构, 需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析 ORC 文件中的数据, 按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 使用 NebulaGraph Console 创建一个图空间 **basketballplayer**, 并创建一个 Schema, 如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息, 请参见[快速开始](#)。

步骤 2: 处理 ORC 文件

确认以下信息:

- 处理 ORC 文件以满足 Schema 的要求。
- 获取 ORC 文件存储路径。

步骤 3: 修改配置文件

编译 Exchange 后, 复制 target/classes/application.conf 文件设置 ORC 数据源相关的配置。在本示例中, 复制的文件名为 orc_application.conf。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    executor: {
      memory:1G
    }
  }

  cores: {
    max: 16
  }
}

# NebulaGraph相关配置
nebula: {
  address: {
    # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
    # 如果有多台服务器, 地址之间用英文逗号 (,) 分隔。
  }
}
```

```

# 格式："ip1:port","ip2:port","ip3:port"
graph: ["127.0.0.1:9669"]
#任意一个 Meta 服务的地址。
#如果您的NebulaGraph在虚拟网络中，如k8s，请配置 Leader Meta的地址。
meta: ["127.0.0.1:9559"]
}

# 指定拥有NebulaGraph写权限的用户名和密码。
user: root
pswd: nebulas

# 指定图空间名称。
space: basketballplayer
connection: {
    timeout: 3000
    retry: 3
}
execution: {
    retry: 3
}
error: {
    max: 32
    output: /tmp/errors
}
rate: {
    limit: 1024
    timeout: 1000
}
}

# 处理点
tags: [
    # 设置 Tag player 相关信息。
    {
        # 指定NebulaGraph中定义的 Tag 名称。
        name: player
        type: {
            # 指定数据源，使用 ORC。
            source: orc
        }
        # 指定如何将点数据导入NebulaGraph：Client 或 SST。
        sink: client
    }
]
# 指定 ORC 文件的路径。
# 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
# 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.orc"。
path: "hdfs://192.168.*.*:9000/data/vertex_player.orc"

# 在 fields 里指定 ORC 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
# 如果需要指定多个值，用英文逗号 (,) 隔开。
fields: [age,name]

# 指定NebulaGraph中定义的属性名称。
# fields 与 nebula.fields 的顺序必须一一对应。
nebula.fields: [age, name]

# 指定一个列作为 VID 的源。
# vertex 的值必须与 ORC 文件中的字段保持一致。
# 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
vertex: {
    field:id
}

# 指定单批次写入NebulaGraph的最大点数量。
batch: 256

# 指定 Spark 分片数量。
partition: 32
}

# 设置 Tag team 相关信息。
{
    # 指定NebulaGraph中定义的 Tag 名称。
    name: team
    type: {
        # 指定数据源，使用 ORC。
        source: orc
    }
    # 指定如何将点数据导入NebulaGraph：Client 或 SST。
    sink: client
}

# 指定 ORC 文件的路径。
# 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
# 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.orc"。
path: "hdfs://192.168.*.*:9000/data/vertex_team.orc"

# 在 fields 里指定 ORC 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
# 如果需要指定多个值，用英文逗号 (,) 隔开。
fields: [name]

# 指定NebulaGraph中定义的属性名称。
# fields 与 nebula.fields 的顺序必须一一对应。

```

```

nebula.fields: [name]

# 指定一个列作为 VID 的源。
# vertex 的值必须与 ORC 文件中的字段保持一致。
# 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
vertex: {
    field:id
}

# 指定单批次写入NebulaGraph的最大点数量。
batch: 256

# 指定 Spark 分片数量。
partition: 32
}

# 如果需要添加更多点，请参考前面的配置进行添加。
]

# 处理边
edges: [
    # 设置 Edge type follow 相关信息。
    {
        # 指定NebulaGraph中定义的 Edge type 名称。
        name: follow
        type: {
            # 指定数据源，使用 ORC。
            source: orc

            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }
    }

    # 指定 ORC 文件的路径。
    # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
    # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.orc"。
    path: "hdfs://192.168.*.*:9000/data/edge_follow.orc"

    # 在 fields 里指定 ORC 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
    # 如果需要指定多个值，用英文逗号 (,) 隔开。
    fields: [degree]

    # 指定NebulaGraph中定义的属性名称。
    # fields 与 nebula.fields 的顺序必须一一对应。
    nebula.fields: [degree]

    # 指定一个列作为起始点和目的点的源。
    # vertex 的值必须与 ORC 文件中的字段保持一致。
    # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
    source: {
        field: src
    }

    target: {
        field: dst
    }

    # 指定一个列作为 rank 的源（可选）。
    #ranking: rank

    # 指定单批次写入NebulaGraph的最大边数量。
    batch: 256

    # 指定 Spark 分片数量。
    partition: 32
}

# 设置 Edge type serve 相关信息。
{
    # 指定NebulaGraph中定义的 Edge type 名称。
    name: serve
    type: {
        # 指定数据源，使用 ORC。
        source: orc

        # 指定如何将点数据导入NebulaGraph：Client 或 SST。
        sink: client
    }

    # 指定 ORC 文件的路径。
    # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
    # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.orc"。
    path: "hdfs://192.168.*.*:9000/data/edge_serve.orc"

    # 在 fields 里指定 ORC 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
    # 如果需要指定多个值，用英文逗号 (,) 隔开。
    fields: [start_year,end_year]

    # 指定NebulaGraph中定义的属性名称。
    # fields 与 nebula.fields 的顺序必须一一对应。
    nebula.fields: [start_year, end_year]

    # 指定一个列作为起始点和目的点的源。
    # vertex 的值必须与 ORC 文件中的字段保持一致。
    # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
    source: {
}

```

```

        field: src
    }
    target: {
        field: dst
    }

    # 指定一个列作为 rank 的源 (可选)。
    #ranking: _c5

    # 指定单批次写入NebulaGraph的最大边数量。
    batch: 256

    # 指定 Spark 分片数量。
    partition: 32
}

]

# 如果需要添加更多边, 请参考前面的配置进行添加。
}

```

步骤 4: 向NEBULAGRAPH导入数据

运行如下命令将 ORC 文件数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <orc_application.conf_path>
```

Note

JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/orc_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`, 确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 5: (可选) 验证数据

用户可以在NebulaGraph客户端 (例如 NebulaGraph Studio) 中执行查询语句, 确认数据是否已导入。例如:

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 6: (如有) 在NEBULAGRAPH中重建索引

导入数据后, 用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.4 导入 Parquet 文件数据

本文以一个示例说明如何使用 Exchange 将存储在 HDFS 或本地的 Parquet 文件数据导入 NebulaGraph。

如果要向 NebulaGraph 导入本地 Parquet 文件, 请参见 [NebulaGraph Importer](#)。

数据集

本文以 [basketballplayer](#) 数据集为例。

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
- NebulaGraph: 3.4.1。使用 [Docker Compose](#) 部署。

前提条件

开始导入数据之前, 用户需要确认以下信息:

- 已经[安装部署NebulaGraph](#) 并获取如下信息:
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息, 包括 Tag 和 Edge type 的名称、属性等。
- 如果文件存储在 HDFS 上, 需要确认 Hadoop 服务运行正常。
- 如果文件存储在本地且NebulaGraph是集群架构, 需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析 Parquet 文件中的数据, 按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 使用 NebulaGraph Console 创建一个图空间 **basketballplayer**, 并创建一个 Schema, 如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息, 请参见[快速开始](#)。

步骤 2: 处理 PARQUET 文件

确认以下信息:

- 处理 Parquet 文件以满足 Schema 的要求。
- 获取 Parquet 文件存储路径。

步骤 3: 修改配置文件

编译 Exchange 后, 复制 target/classes/application.conf 文件设置 Parquet 数据源相关的配置。在本示例中, 复制的文件名为 **parquet_application.conf**。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    executor: {
      memory:1G
    }
  }

  cores: {
    max: 16
  }
}

#NebulaGraph相关配置
nebula: {
  address: {
    # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
    # 如果有多台服务器, 地址之间用英文逗号 (,) 分隔。
  }
}
```

```

# 格式："ip1:port","ip2:port","ip3:port"
graph: ["127.0.0.1:9669"]
#任意一个 Meta 服务的地址。
#如果您的NebulaGraph在虚拟网络中，如k8s，请配置 Leader Meta的地址。
meta: ["127.0.0.1:9559"]
}

# 指定拥有NebulaGraph写权限的用户名和密码。
user: root
pswd: nebula

# 指定图空间名称。
space: basketballplayer
connection: {
    timeout: 3000
    retry: 3
}
execution: {
    retry: 3
}
error: {
    max: 32
    output: /tmp/errors
}
rate: {
    limit: 1024
    timeout: 1000
}
}

# 处理点
tags: [
    # 设置 Tag player 相关信息。
    {
        # 指定NebulaGraph中定义的 Tag 名称。
        name: player
        type: {
            # 指定数据源，使用 Parquet。
            source: parquet
        }
        # 指定如何将点数据导入NebulaGraph：Client 或 SST。
        sink: client
    }
]
# 指定 Parquet 文件的路径。
# 如果文件存储在 HDFS 上，用双引号括起路径，以 "hdfs://ip:port/xx/xx"。
# 如果文件存储在本地，用双引号括起路径，以 "file:///tmp/xx.csv"。
path: "hdfs://192.168.11.139000/data/vertex_player.parquet"

# 在 fields 里指定 Parquet 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
# 如果需要指定多个值，用英文逗号 (,) 隔开。
fields: [age,name]

# 指定NebulaGraph中定义的属性名称。
# fields 与 nebula.fields 的顺序必须一一对应。
nebula.fields: [age, name]

# 指定一个列作为 VID 的源。
# vertex 的值必须与 Parquet 文件中的字段保持一致。
# 目前，NebulaGraph 3.4.1 只支持字符串或整数类型的 VID。
vertex: {
    field:id
}

# 指定单批次写入NebulaGraph的最大点数量。
batch: 256

# 指定 Spark 分片数量。
partition: 32
}

# 设置 Tag team 相关信息。
{
    # 指定NebulaGraph中定义的 Tag 名称。
    name: team
    type: {
        # 指定数据源，使用 Parquet。
        source: parquet
    }
    # 指定如何将点数据导入NebulaGraph：Client 或 SST。
    sink: client
}

# 指定 Parquet 文件的路径。
# 如果文件存储在 HDFS 上，用双引号括起路径，以 "hdfs://ip:port/xx/xx"。
# 如果文件存储在本地，用双引号括起路径，以 "file:///tmp/xx.csv"。
path: "hdfs://192.168.11.13:9000/data/vertex_team.parquet"

# 在 fields 里指定 Parquet 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
# 如果需要指定多个值，用英文逗号 (,) 隔开。
fields: [name]

# 指定NebulaGraph中定义的属性名称。
# fields 与 nebula.fields 的顺序必须一一对应。

```

```

nebula.fields: [name]

# 指定一个列作为 VID 的源。
# vertex 的值必须与 Parquet 文件中的字段保持一致。
# 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
vertex: {
    field:id
}

# 指定单批次写入NebulaGraph的最大点数量。
batch: 256

# 指定 Spark 分片数量。
partition: 32
}

# 如果需要添加更多点，请参考前面的配置进行添加。
]

# 处理边
edges: [
    # 设置 Edge type follow 相关信息。
    {
        # 指定NebulaGraph中定义的 Edge type 名称。
        name: follow
        type: {
            # 指定数据源，使用 Parquet。
            source: parquet

            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }

        # 指定 Parquet 文件的路径。
        # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
        # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.csv"。
        path: "hdfs://192.168.11.13:9000/data/edge_follow.parquet"

        # 在 fields 里指定 Parquet 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
        # 如果需要指定多个值，用英文逗号 (,) 隔开。
        fields: [degree]

        # 指定NebulaGraph中定义的属性名称。
        # fields 与 nebula.fields 的顺序必须一一对应。
        nebula.fields: [degree]

        # 指定一个列作为起始点和目的点的源。
        # vertex 的值必须与 Parquet 文件中的字段保持一致。
        # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
        source: {
            field: src
        }

        target: {
            field: dst
        }

        # 指定一个列作为 rank 的源（可选）。
        #ranking: rank

        # 指定单批次写入NebulaGraph的最大边数量。
        batch: 256

        # 指定 Spark 分片数量。
        partition: 32
    }

    # 设置 Edge type serve 相关信息。
    {
        # 指定NebulaGraph中定义的 Edge type 名称。
        name: serve
        type: {
            # 指定数据源，使用 Parquet。
            source: parquet

            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }

        # 指定 Parquet 文件的路径。
        # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
        # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.csv"。
        path: "hdfs://192.168.11.13:9000/data/edge_serve.parquet"

        # 在 fields 里指定 Parquet 文件中 key 名称，其对应的 value 会作为NebulaGraph中指定属性的数据源。
        # 如果需要指定多个值，用英文逗号 (,) 隔开。
        fields: [start_year,end_year]

        # 指定NebulaGraph中定义的属性名称。
        # fields 与 nebula.fields 的顺序必须一一对应。
        nebula.fields: [start_year, end_year]

        # 指定一个列作为起始点和目的点的源。
        # vertex 的值必须与 Parquet 文件中的字段保持一致。
        # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
        source: {
    }
}

```

```

        field: src
    }
    target: {
        field: dst
    }

    # 指定一个列作为 rank 的源 (可选)。
    #ranking: _c5

    # 指定单批次写入NebulaGraph的最大边数量。
    batch: 256

    # 指定 Spark 分片数量。
    partition: 32
}

]

# 如果需要添加更多边, 请参考前面的配置进行添加。
}

```

步骤 4: 向NEBULAGRAPH导入数据

运行如下命令将 Parquet 文件数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <parquet_application.conf_path>
```

Note

JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/parquet_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`, 确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 5: (可选) 验证数据

用户可以在NebulaGraph客户端 (例如 NebulaGraph Studio) 中执行查询语句, 确认数据是否已导入。例如:

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 6: (如有) 在NEBULAGRAPH中重建索引

导入数据后, 用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.5 导入 HBase 数据

本文以一个示例说明如何使用 Exchange 将存储在 HBase 上的数据导入 NebulaGraph。

数据集

本文以 **basketballplayer** 数据集为例。

在本示例中，该数据集已经存入 HBase 中，以 `player`、`team`、`follow` 和 `serve` 四个表存储了所有点和边的信息。以下为各个表的部分数据。

```
hbase(main):002:0> scan "player"
ROW                                         COLUMN+CELL
player100                                     column=cf:age, timestamp=1618881347530, value=42
player100                                     column=cf:name, timestamp=1618881354604, value=Tim Duncan
player101                                     column=cf:age, timestamp=1618881369124, value=36
player101                                     column=cf:name, timestamp=1618881379102, value=Tony Parker
player102                                     column=cf:age, timestamp=1618881386987, value=33
player102                                     column=cf:name, timestamp=1618881393370, value=LaMarcus Aldridge
player103                                     column=cf:age, timestamp=1618881402002, value=32
player103                                     column=cf:name, timestamp=1618881407882, value=Rudy Gay
...
hbase(main):003:0> scan "team"
ROW                                         COLUMN+CELL
team200                                      column=cf:name, timestamp=1618881445563, value=Warriors
team201                                      column=cf:name, timestamp=1618881453636, value=Nuggets
...
hbase(main):004:0> scan "follow"
ROW                                         COLUMN+CELL
player100                                     column=cf:degree, timestamp=1618881804853, value=95
player100                                     column=cf:dst_player, timestamp=1618881791522, value=player101
player101                                     column=cf:degree, timestamp=1618881824685, value=90
player101                                     column=cf:dst_player, timestamp=1618881816042, value=player102
...
hbase(main):005:0> scan "serve"
ROW                                         COLUMN+CELL
player100                                     column=cf:end_year, timestamp=1618881899333, value=2016
player100                                     column=cf:start_year, timestamp=1618881890117, value=1997
player100                                     column=cf:teamid, timestamp=1618881875739, value=team204
...
```

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
- HBase: 2.2.7
- NebulaGraph: 3.4.1。使用 [Docker Compose 部署](#)。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经安装并开启 Hadoop 服务。

操作步骤

步骤 1：在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema：

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 在NebulaGraph中创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
  (partition_num = 10, \
  replica_factor = 1, \
  vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2：修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 HBase 数据源相关的配置。在本示例中，复制的文件名为 `hbase_application.conf`。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
  }
}
```

```

}
cores: {
  max: 16
}
}

# NebulaGraph 相关配置
nebula: {
  address: {
    # 以下为 NebulaGraph 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:["127.0.0.1:9669"]
    #任意一个 Meta 服务的地址。
    meta:["127.0.0.1:9559"]
  }
  # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
  user: root
  pswd: nebulapass
  # 填写 NebulaGraph 中需要写入数据的图空间名称。
  space: basketballplayer
  connection: {
    timeout: 3000
    retry: 3
  }
  execution: {
    retry: 3
  }
  error: {
    max: 32
    output: /tmp/errors
  }
  rate: {
    limit: 1024
    timeout: 1000
  }
}
# 处理点
tags: [
  # 设置 Tag player 相关信息。
  # 如果需要将 rowkey 设置为数据源，请填写“rowkey”，列族内的列请填写实际列名。
  {
    # NebulaGraph 中对应的 Tag 名称。
    name: player
    type: {
      # 指定数据源文件格式，设置为 HBase。
      source: hbase
      # 指定如何将点数据导入NebulaGraph：Client 或 SST。
      sink: client
    }
    host:192.168.*.*
    port:2181
    table:"player"
    columnFamily:"cf"
    # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
    # 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: [age,name]
    nebula.fields: [age,name]
    # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
    # 例如 rowkey 作为 VID 的来源，请填写“rowkey”。
    vertex:{ field:rowkey
  }
  # 单批次写入 NebulaGraph 的数据条数。
  batch: 256
  # Spark 分区数量
  partition: 32
}
# 设置 Tag team 相关信息。
{
  name: team
  type: {
    source: hbase
    sink: client
  }
  host:192.168.*.*
  port:2181
  table:"team"
  columnFamily:"cf"
  fields: [name]
  nebula.fields: [name]
  vertex:{ field:rowkey
  }
  batch: 256
  partition: 32
}

```

```

]
# 处理边数据
edges: [
  # 设置 Edge type follow 相关信息
  {
    # NebulaGraph 中对应的 Edge type 名称。
    name: follow

    type: {
      # 指定数据源文件格式, 设置为 HBase。
      source: hbase

      # 指定边数据导入 NebulaGraph 的方式,
      # 指定如何将点数据导入NebulaGraph: Client 或 SST。
      sink: client
    }

    host:192.168.*.*
    port:2181
    table:"follow"
    columnFamily:"cf"

    # 在 fields 里指定 follow 表中的列名称, 其对应的 value 会作为 NebulaGraph 中指定属性。
    # 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: [degree]
    nebula.fields: [degree]

    # 在 source 里, 将 follow 表中某一列作为边的起始点数据源。示例使用 rowkey。
    # 在 target 里, 将 follow 表中某一列作为边的目的点数据源。示例使用列 dst_player。
    source:{
      field:rowkey
    }

    target:{ 
      field:dst_player
    }

    # 指定一个列作为 rank 的源 (可选)。
    #ranking: rank

    # 单批次写入 NebulaGraph 的数据条数。
    batch: 256

    # Spark 分区数量
    partition: 32
  }

  # 设置 Edge type serve 相关信息
  {
    name: serve
    type: {
      source: hbase
      sink: client
    }

    host:192.168.*.*
    port:2181
    table:"serve"
    columnFamily:"cf"

    fields: [start_year,end_year]
    nebula.fields: [start_year,end_year]
    source:{
      field:rowkey
    }

    target:{ 
      field:teamid
    }

    # 指定一个列作为 rank 的源 (可选)。
    #ranking: rank

    batch: 256
    partition: 32
  }
]
}

```

步骤 3: 向NEBULAGRAPH导入数据

运行如下命令将 HBase 数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <hbase_application.conf_path>
```

Note

JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/hbase_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4: (可选) 验证数据

用户可以在NebulaGraph客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如:

```
 LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5: (如有) 在NEBULAGRAPH中重建索引

导入数据后，用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.6 导入 MySQL/PostgreSQL 数据

本文以一个示例说明如何使用 Exchange 将存储在 MySQL 上的数据导入 NebulaGraph，也适用于从 PostgreSQL 导出数据到 NebulaGraph。

数据集

本文以 **basketballplayer** 数据集为例。

在本示例中，该数据集已经存入 MySQL 中名为 basketball 的数据库中，以 player、team、follow 和 serve 四个表存储了所有点和边的信息。以下为各个表的结构。

```
mysql> desc player;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| playerid | varchar(30) | YES | NULL |   |
| age | int | YES | NULL |   |
| name | varchar(30) | YES | NULL |   |
+-----+-----+-----+-----+-----+

mysql> desc team;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| teamid | varchar(30) | YES | NULL |   |
| name | varchar(30) | YES | NULL |   |
+-----+-----+-----+-----+-----+

mysql> desc follow;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| src_player | varchar(30) | YES | NULL |   |
| dst_player | varchar(30) | YES | NULL |   |
| degree | int | YES | NULL |   |
+-----+-----+-----+-----+-----+

mysql> desc serve;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| playerid | varchar(30) | YES | NULL |   |
| teamid | varchar(30) | YES | NULL |   |
| start_year | int | YES | NULL |   |
| end_year | int | YES | NULL |   |
+-----+-----+-----+-----+-----+
```

环境配置

本文示例在 Mac OS 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
- MySQL： 8.0.23
- NebulaGraph： 3.4.1。使用 **Docker Compose** 部署。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经安装并开启 Hadoop 服务。

操作步骤

步骤 1：在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema：

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 在NebulaGraph中创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
  (partition_num = 10, \
  replica_factor = 1, \
  vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2：修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 MySQL 数据源相关的配置。在本示例中，复制的文件名为 mysql_application.conf。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
  }
}
```

```

}
cores: {
  max: 16
}
}

# NebulaGraph 相关配置
nebula: {
  address: {
    # 以下为 NebulaGraph 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:["127.0.0.1:9669"]
    #任意一个 Meta 服务的地址。
    meta:["127.0.0.1:9559"]
  }
  # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
  user: root
  pswd: nebula
  # 填写 NebulaGraph 中需要写入数据的图空间名称。
  space: basketballplayer
  connection: {
    timeout: 3000
    retry: 3
  }
  execution: {
    retry: 3
  }
  error: {
    max: 32
    output: /tmp/errors
  }
  rate: {
    limit: 1024
    timeout: 1000
  }
}
# 处理点
tags: [
  # 设置 Tag player 相关信息。
  {
    # NebulaGraph 中对应的 Tag 名称。
    name: player
    type: {
      # 指定数据源文件格式，设置为 MySQL。
      source: mysql
      # 指定如何将点数据导入NebulaGraph：Client 或 SST。
      sink: client
    }
    host:192.168.*.*
    port:3306
    database:"basketball"
    table:"player"
    user:"test"
    password:"123456"
    sentence:"select playerid, age, name from player order by playerid"

    # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
    # 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: [age,name]
    nebula.fields: [age,name]

    # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
    vertex: {
      field:playerid
    }

    # 单批次写入 NebulaGraph 的数据条数。
    batch: 256

    # Spark 分区数量
    partition: 32
  }
  # 设置 Tag team 相关信息。
  {
    name: team
    type: {
      source: mysql
      sink: client
    }
    host:192.168.*.*
    port:3306
    database:"basketball"
    table:"team"
    user:"test"
    password:"123456"
    sentence:"select teamid, name from team order by teamid"

    fields: [name]
    nebula.fields: [name]
  }
}

```

```

vertex: {
    field: teamid
}
batch: 256
partition: 32
}

]

# 处理边数据
edges: [
    # 设置 Edge type follow 相关信息
    {
        # NebulaGraph 中对应的 Edge type 名称。
        name: follow

        type: {
            # 指定数据源文件格式，设置为 MySQL。
            source: mysql

            # 指定边数据导入 NebulaGraph 的方式，
            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }

        host:192.168.*.*
        port:3306
        database:"basketball"
        table:"follow"
        user:"test"
        password:"123456"
        sentence:"select src_player,dst_player,degree from follow order by src_player"

        # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
        # 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: [degree]
        nebula.fields: [degree]

        # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
        # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
        source: {
            field: src_player
        }

        target: {
            field: dst_player
        }

        # 指定一个列作为 rank 的源（可选）。
        #ranking: rank

        # 单批次写入 NebulaGraph 的数据条数。
        batch: 256

        # Spark 分区数量
        partition: 32
    }
]

# 设置 Edge type serve 相关信息
{
    name: serve
    type: {
        source: mysql
        sink: client
    }

    host:192.168.*.*
    port:3306
    database:"basketball"
    table:"serve"
    user:"test"
    password:"123456"
    sentence:"select playerid,teamid,start_year,end_year from serve order by playerid"
    fields: [start_year,end_year]
    nebula.fields: [start_year,end_year]
    source: {
        field: playerid
    }
    target: {
        field: teamid
    }

    # 指定一个列作为 rank 的源（可选）。
    #ranking: rank

    batch: 256
    partition: 32
}
]
}

```

步骤 3: 向**NEBULAGRAPH**导入数据

运行如下命令将 MySQL 数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <mysql_application.conf_path>
```



JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/mysql_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`, 确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4: (可选) 验证数据

用户可以在NebulaGraph客户端 (例如 NebulaGraph Studio) 中执行查询语句, 确认数据是否已导入。例如:

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5: (如有) 在**NEBULAGRAPH**中重建索引

导入数据后, 用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.7 导入 Oracle 数据

本文以一个示例说明如何使用 Exchange 将存储在 Oracle 上的数据导入 NebulaGraph。

数据集

本文以 **basketballplayer** 数据集为例。

在本示例中，该数据集已经存入 Oracle 中名为 `basketball` 的数据库中，以 `player`、`team`、`follow` 和 `serve` 四个表存储了所有点和边的信息。以下为各个表的结构。

```
oracle> desc player;
+-----+-----+-----+
| Column | Null | Type |
+-----+-----+-----+
| PLAYERID | - | VARCHAR2(30) |
| NAME | - | VARCHAR2(30) |
| AGE | - | NUMBER |
+-----+-----+-----+

oracle> desc team;
+-----+-----+-----+
| Column | Null | Type |
+-----+-----+-----+
| TEAMID | - | VARCHAR2(30) |
| NAME | - | VARCHAR2(30) |
+-----+-----+-----+

oracle> desc follow;
+-----+-----+-----+
| Column | Null | Type |
+-----+-----+-----+
| SRC_PLAYER | - | VARCHAR2(30) |
| DST_PLAYER | - | VARCHAR2(30) |
| DEGREE | - | NUMBER |
+-----+-----+-----+

oracle> desc serve;
+-----+-----+-----+
| Column | Null | Type |
+-----+-----+-----+
| PLAYERID | - | VARCHAR2(30) |
| TEAMID | - | VARCHAR2(30) |
| START_YEAR | - | NUMBER |
| END_YEAR | - | NUMBER |
+-----+-----+-----+
```

环境配置

本文示例在 Mac OS 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
- NebulaGraph：3.4.1。使用 [Docker Compose](#) 部署。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建的 Schema 信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经安装并开启 Hadoop 服务。

操作步骤

步骤 1：在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema：

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 在NebulaGraph中创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
  (partition_num = 10, \
  replica_factor = 1, \
  vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2：修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 Oracle 数据源相关的配置。在本示例中，复制的文件名为 oracle_application.conf。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
  }
}
```

```

}
cores: {
  max: 16
}
}

# NebulaGraph 相关配置
nebula: {
  address: {
    # 以下为 NebulaGraph 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: ["127.0.0.1:9669"]
    #任意一个 Meta 服务的地址。
    meta: ["127.0.0.1:9559"]
  }
  # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
  user: root
  pswd: nebula
  # 填写 NebulaGraph 中需要写入数据的图空间名称。
  space: basketballplayer
  connection: {
    timeout: 3000
    retry: 3
  }
  execution: {
    retry: 3
  }
  error: {
    max: 32
    output: /tmp/errors
  }
  rate: {
    limit: 1024
    timeout: 1000
  }
}
# 处理点
tags: [
{
  # NebulaGraph 中对应的 Tag 名称。
  name: player
  type: {
    # 指定数据源文件格式，设置为 Oracle。
    source: oracle
    # 指定如何将点数据导入NebulaGraph：Client 或 SST。
    sink: client
  }
}

url:"jdbc:oracle:thin:@host:1521:db"
driver: "oracle.jdbc.driver.OracleDriver"
user: "root"
password: "123456"
table: "basketball.player"
sentence: "select playerid, name, age from player"

# 在 fields 里指定 player 表中的列名称，其对应的 value 会作为NebulaGraph中指定属性。
# 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: [age,name]
nebula.fields: [age,name]

# 指定表中某一列数据为NebulaGraph中点 VID 的来源。
vertex: {
  field:playerid
}

# 单批次写入NebulaGraph的数据条数。
batch: 256

# Spark 分区数量
partition: 32
}
# 设置 Tag team 相关信息。
{
  name: team
  type: {
    source: oracle
    sink: client
  }
}

url:"jdbc:oracle:thin:@host:1521:db"
driver: "oracle.jdbc.driver.OracleDriver"
user: "root"
password: "123456"
table: "basketball.team"
sentence: "select teamid, name from team"

fields: [name]
nebula.fields: [name]
vertex: {
  field: teamid
}

```

```

        }
        batch: 256
        partition: 32
    }

]

# 处理边数据
edges: [
    # 设置 Edge type follow 相关信息
    {
        # NebulaGraph中对应的 Edge type 名称。
        name: follow

        type: {
            # 指定数据源文件格式, 设置为 Oracle。
            source: oracle

            # 指定边数据导入 NebulaGraph 的方式,
            # 指定如何将点数据导入NebulaGraph : Client 或 SST。
            sink: client
        }

        url:"jdbc:oracle:thin:@host:1521:db"
        driver: "oracle.jdbc.driver.OracleDriver"
        user: "root"
        password: "123456"
        table: "basketball.follow"
        sentence: "select src_player, dst_player, degree from follow"

        # 在 fields 里指定 follow 表中的列名称, 其对应的 value 会作为NebulaGraph中指定属性。
        # 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: [degree]
        nebula.fields: [degree]

        # 在 source 里, 将 follow 表中某一列作为边的起始点数据源。
        # 在 target 里, 将 follow 表中某一列作为边的目的点数据源。
        source: {
            field: src_player
        }

        target: {
            field: dst_player
        }

        # 指定一个列作为 rank 的源 (可选)。
        #ranking: rank

        # 单批次写入NebulaGraph的数据条数。
        batch: 256

        # Spark 分区数量
        partition: 32
    }

    # 设置 Edge type serve 相关信息
    {
        name: serve
        type: {
            source: oracle
            sink: client
        }

        url:"jdbc:oracle:thin:@host:1521:db"
        driver: "oracle.jdbc.driver.OracleDriver"
        user: "root"
        password: "123456"
        table: "basketball.serve"
        sentence: "select playerid, teamid, start_year, end_year from serve"

        fields: [start_year,end_year]
        nebula.fields: [start_year,end_year]
        source: {
            field: playerid
        }

        target: {
            field: teamid
        }

        # 指定一个列作为 rank 的源 (可选)。
        #ranking: rank

        batch: 256
        partition: 32
    }
]
}

```

步骤 3: 向NEBULAGRAPH导入数据

运行如下命令将 Oracle 数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <oracle_application.conf_path>
```

Note

JAR 包有两种获取方式: [自行编译](#)或者从 [maven](#) 仓库下载。

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/oracle_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4: (可选) 验证数据

用户可以在NebulaGraph客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如:

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5: (如有) 在NEBULAGRAPH中重建索引

导入数据后，用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.8 导入 ClickHouse 数据

本文以一个示例说明如何使用 Exchange 将存储在 ClickHouse 上的数据导入 NebulaGraph。

数据集

本文以 **basketballplayer** 数据集为例。

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
- ClickHouse: docker 部署 yandex/clickhouse-server tag: latest(2021.07.01)
- NebulaGraph: 3.4.1。使用 **Docker Compose** 部署。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经[安装部署NebulaGraph](#) 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经安装并开启 Hadoop 服务。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 在NebulaGraph中创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2: 修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 ClickHouse 数据源相关的配置。在本示例中，复制的文件名为 `clickhouse_application.conf`。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: Nebula Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  # NebulaGraph相关配置
  nebula: {
    address: {
      # 以下为 NebulaGraph 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: ["127.0.0.1:9669"]

      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta: ["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
  }
}
```

```

}
execution: {
  retry: 3
}
error: {
  max: 32
  output: /tmp/errors
}
rate: {
  limit: 1024
  timeout: 1000
}
}

# 处理点
tags: [
  # 设置 Tag player 相关信息。
  {
    name: player
    type: {
      # 指定数据源文件格式，设置为 ClickHouse。
      source: clickhouse
      # 指定如何将点数据导入NebulaGraph：Client 或 SST。
      sink: client
    }
  }

  # ClickHouse 的 JDBC URL
  url:"jdbc:clickhouse://192.168.*.*:8123/basketballplayer"

  user:"user"
  password:"123456"

  # ClickHouse 分区数
  numPartition:"5"

  table:"player"
  sentence:"select * from player"

  # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为NebulaGraph中指定属性。
  # 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: [name,age]
  nebula.fields: [name,age]

  # 指定表中某一列数据为NebulaGraph中点 VID 的来源。
  vertex: {
    field:playerid
    # policy:hash
  }

  # 单批次写入NebulaGraph的数据条数。
  batch: 256

  # Spark 分区数量
  partition: 32
}

# 设置 Tag team 相关信息。
{
  name: team
  type: {
    source: clickhouse
    sink: client
  }
  url:"jdbc:clickhouse://192.168.*.*:8123/basketballplayer"
  user:"user"
  password:"123456"
  numPartition:"5"
  table:"team"
  sentence:"select * from team"
  fields: [name]
  nebula.fields: [name]
  vertex: {
    field:teamid
  }
  batch: 256
  partition: 32
}

]

# 处理边数据
edges: [
  # 设置 Edge type follow 相关信息
  {
    # NebulaGraph中对应的 Edge type 名称。
    name: follow

    type: {
      # 指定数据源文件格式，设置为 ClickHouse。
      source: clickhouse

      # 指定边数据导入NebulaGraph的方式，
      # 指定如何将点数据导入NebulaGraph：Client 或 SST。
      sink: client
    }
  }
]

```

```

# ClickHouse 的 JDBC URL
url:"jdbc:clickhouse://192.168.*.*:8123/basketballplayer"

user:"user"
password:"123456"

# ClickHouse 分区数
numPartition:"5"

table:"follow"
sentence:"select * from follow"

# 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为NebulaGraph中指定属性。
# 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: [degree]
nebula.fields: [degree]

# 在 source 里，将 follow 表中某一列作为边的起始点数据源。
source: {
  field:src_player
}

# 在 target 里，将 follow 表中某一列作为边的目的点数据源。
target: {
  field:dst_player
}

# 指定一个列作为 rank 的源（可选）。
#ranking: rank

# 单批次写入NebulaGraph的数据条数。
batch: 256

# Spark 分区数量
partition: 32
}

# 设置 Edge type serve 相关信息
{
  name: serve
  type: {
    source: clickhouse
    sink: client
  }
  url:"jdbc:clickhouse://192.168.*.*:8123/basketballplayer"
  user:"user"
  password:"123456"
  numPartition:"5"
  sentence:"select * from serve"
  fields: [start_year,end_year]
  nebula.fields: [start_year,end_year]
  source: {
    field:playerid
  }
  target: {
    field:teamid
  }

  # 指定一个列作为 rank 的源（可选）。
  #ranking: rank

  batch: 256
  partition: 32
}
]
}

```

步骤 3: 向NEBULAGRAPH导入数据

运行如下命令将 ClickHouse 数据导入到NebulaGraph中。关于参数的说明，请参见[导入命令参数](#)。

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <clickhouse_application.conf_path>
```



JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes	clickhouse_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4: (可选) 验证数据

用户可以在NebulaGraph客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5: (如有) 在NEBULAGRAPH中重建索引

导入数据后，用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.9 导入 Neo4j 数据

本文以一个示例说明如何使用 Exchange 将存储在 Neo4j 的数据导入 NebulaGraph。

实现方法

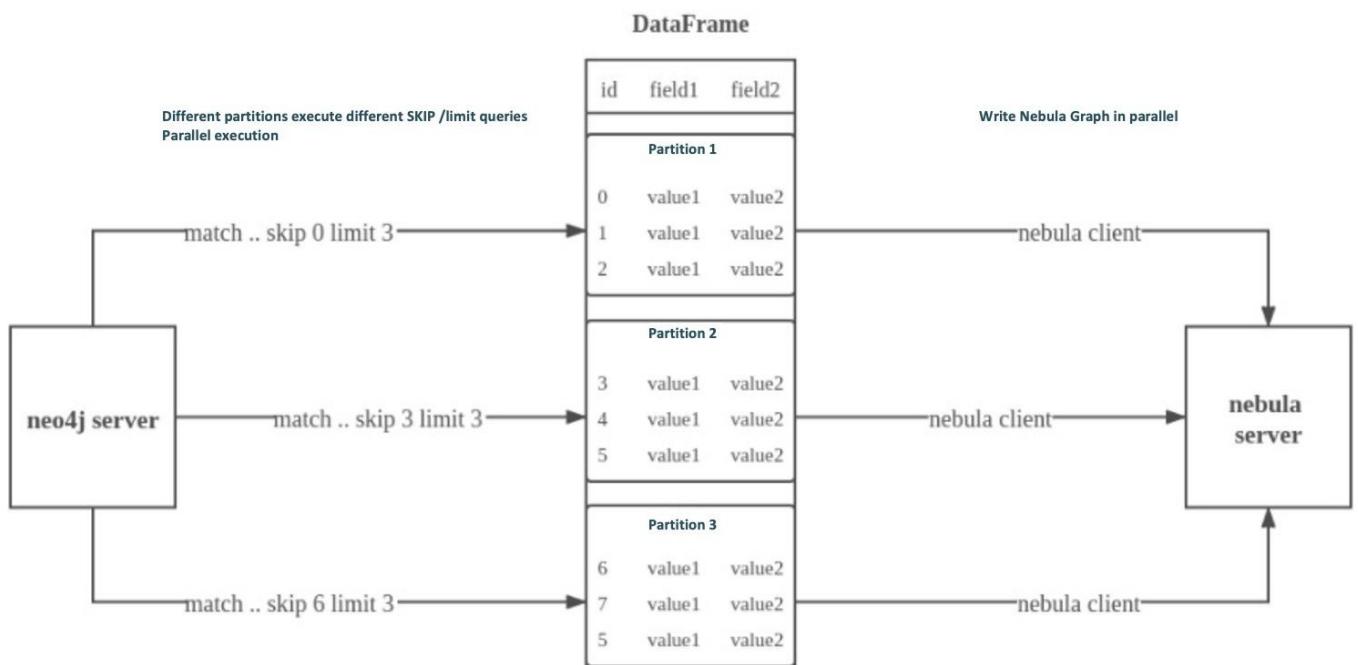
Exchange 使用 **Neo4j Driver 4.0.1** 实现对 Neo4j 数据的读取。执行批量导出之前，用户需要在配置文件中写入针对标签（label）和关系类型（Relationship Type）自动执行的 Cypher 语句，以及 Spark 分区数，提高数据导出性能。

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。如果处于续传状态，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NebulaGraph数据库中。

整个过程如下图所示。



数据集

本文以 **basketballplayer** 数据集为例。

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: 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
- NebulaGraph: 3.4.1。使用 [Docker Compose 部署](#)。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经[安装部署NebulaGraph](#) 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 使用 NebulaGraph Console 创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[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 数据库中为相应属性创建索引。详细信息，参考 [Neo4j 用户手册](#)。

步骤 3: 修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置数据源相关的配置。在本示例中，复制的文件名为 neo4j_application.conf。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }

    driver: {
      cores: 1
      maxResultSize: 1G
    }

    executor: {
      memory:1G
    }

    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address: {
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
    user: root
  }
}
```

```

pswd: nebula
space: basketballplayer

connection: {
    timeout: 3000
    retry: 3
}

execution: {
    retry: 3
}

error: {
    max: 32
    output: /tmp/errors
}

rate: {
    limit: 1024
    timeout: 1000
}
}

# 处理点
tags: [
    # 设置 Tag player 相关信息。
    {
        name: player
        type: {
            source: neo4j
            sink: client
        }
        server: "bolt://192.168.*.*:7687"
        user: neo4j
        password:neo4j
        database:neo4j
        exec: "match (n:player) return n.id as id, n.age as age, n.name as name"
        fields: [age,name]
        nebula.fields: [age,name]
        vertex: {
            field:id
        }
        partition: 10
        batch: 1000
        check_point_path: /tmp/test
    }
    # 设置 Tag team 相关信息。
    {
        name: team
        type: {
            source: neo4j
            sink: client
        }
        server: "bolt://192.168.*.*:7687"
        user: neo4j
        password:neo4j
        # bolt 3 does not support `select database`, please do not config database
        # database:neo4j
        exec: "match (n:team) return n.id as id,n.name as name order by id(n)"
        fields: [name]
        nebula.fields: [name]
        vertex: {
            field:id
        }
        partition: 10
        batch: 1000
        check_point_path: /tmp/test
    }
]

# 处理边数据
edges: [
    # 设置 Edge type follow 相关信息
    {
        name: follow
        type: {
            source: neo4j
            sink: client
        }
        server: "bolt://192.168.*.*:7687"
        user: neo4j
        password:neo4j
        database:neo4j
        exec: "match (a:player)-[r:follow]-(b:player) return a.id as src, b.id as dst, r.degree as degree order by id(r)"
        fields: [degree]
        nebula.fields: [degree]
        source: {
            field: src
        }
        target: {
            field: dst
        }
        #ranking: rank
    }
]

```

```

partition: 10
batch: 1000
check_point_path: /tmp/test
}
# 设置 Edge type serve 相关信息
{
  name: serve
  type: {
    source: neo4j
    sink: client
  }
  server: "bolt://192.168.*.*:7687"
  user: neo4j
  password:neo4j
  # bolt 3 does not support 'select database', please do not config database
  #database:neo4j
  exec: "match (a:player)-[r:serve]->(b:team) return a.id as src, b.id as dst, r.start_year as start_year, r.end_year as end_year order by id(r)"
  fields: [start_year,end_year]
  nebula.fields: [start_year,end_year]
  source: {
    field: src
  }
  target: {
    field: dst
  }
  #ranking: rank
  partition: 10
  batch: 1000
  check_point_path: /tmp/test
}
]
}

```

exec 配置说明

在配置 tags.exec 或者 edges.exec 参数时，需要填写 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，用户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，用户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

NebulaGraph 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 NebulaGraph 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 NebulaGraph 中，其它的则会被覆盖掉。由于数据导入过程是并发地往 NebulaGraph 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: 向NEBULAGRAPH导入数据

运行如下命令将文件数据导入到NebulaGraph中。关于参数的说明，请参见[导入命令参数](#)。

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <neo4j_application.conf_path>
```



JAR 包有两种获取方式：[自行编译](#)或者从 maven 仓库下载。

示例:

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/neo4j_application.conf
```

用户可以在返回信息中搜索 batchSuccess.<tag_name/edge_name>，确认成功的数量。例如 batchSuccess.follow: 300。

步骤 5: (可选) 验证数据

用户可以在NebulaGraph客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 **SHOW STATS** 查看统计数据。

步骤 6: (如有) 在**NEBULAGRAPH**中重建索引

导入数据后，用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.10 导入 Hive 数据

本文以一个示例说明如何使用 Exchange 将存储在 Hive 上的数据导入 NebulaGraph。

数据集

本文以 **basketballplayer** 数据集为例。

在本示例中，该数据集已经存入 Hive 中名为 basketball 的数据库中，以 player、team、follow 和 serve 四个表存储了所有点和边的信息。以下为各个表的结构。

```
scala> spark.sql("describe basketball.player").show
+-----+-----+
| col_name|data_type|comment|
+-----+-----+
| playerid| string| null|
| age| bigint| null|
| name| string| null|
+-----+-----+


scala> spark.sql("describe basketball.team").show
+-----+-----+
| col_name|data_type|comment|
+-----+-----+
| teamid| string| null|
| name| string| null|
+-----+-----+


scala> spark.sql("describe basketball.follow").show
+-----+-----+
| col_name|data_type|comment|
+-----+-----+
| src_player| string| null|
| dst_player| string| null|
| degree| bigint| null|
+-----+-----+


scala> spark.sql("describe basketball.serve").show
+-----+-----+
| col_name|data_type|comment|
+-----+-----+
| playerid| string| null|
| teamid| string| null|
| start_year| bigint| null|
| end_year| bigint| null|
+-----+-----+
```

说明：Hive 的数据类型 **bigint** 与 NebulaGraph 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
- NebulaGraph: 3.4.1。使用 **Docker Compose** 部署。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 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）。

操作步骤

步骤 1：在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema：

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 在NebulaGraph中创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
  (partition_num = 10, \
  replica_factor = 1, \
  vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[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 playerid, age, name from basketball.player").show
scala> sql("select teamid, name from basketball.team").show
scala> sql("select src_player, dst_player, degree from basketball.follow").show
scala> sql("select playerid, teamid, start_year, end_year from basketball.serve").show
```

以下为表 basketball.player 中读出的结果。

playerid	age	name
----------	-----	------

```
+-----+-----+
|player100| 42|      Tim Duncan|
|player101| 36|      Tony Parker|
|player102| 33|LaMarcus Aldridge|
|player103| 32|      Rudy Gay|
|player104| 32| Marco Belinelli|
+-----+-----+
...
```

步骤 3: 修改配置文件

编译 Exchange 后, 复制 target/classes/application.conf 文件设置 Hive 数据源相关的配置。在本示例中, 复制的文件名为 `hive_application.conf`。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  # 如果 Spark 和 Hive 部署在不同集群, 才需要配置连接 Hive 的参数, 否则请忽略这些配置。
  #hive: {
  #  waredir: "hdfs://NAMENODE_IP:9000/apps/svr/hive-xxx/warehouse/"
  #  connectionURL: "jdbc:mysql://your_ip:3306/hive_spark?characterEncoding=UTF-8"
  #  connectionDriverName: "com.mysql.jdbc.Driver"
  #  connectionUserName: "user"
  #  connectionPassword: "password"
  #}

  # NebulaGraph 相关配置
  nebula: {
    address: {
      # 以下为 NebulaGraph 的 Graph 服务和所有 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: ["127.0.0.1:9699"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中, 如k8s, 请配置 Leader Meta的地址。
      meta: ["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebulua
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # NebulaGraph 中对应的 Tag 名称。
      name: player
      type: {
        # 指定数据源文件格式, 设置为 hive。
        source: hive
        # 指定如何将点数据导入NebulaGraph : Client 或 SST。
        sink: client
      }
    }
    # 设置读取数据库 basketball 中 player 表数据的 SQL 语句
    exec: "select playerid, age, name from basketball.player"

    # 在 fields 里指定 player 表中的列名称, 其对应的 value 会作为 NebulaGraph 中指定属性。
    # 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: [age,name]
    nebula.fields: [age,name]
  ]
}
```

```

# 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
vertex: {
    field: playerid
}

# 单批次写入 NebulaGraph 的最大数据条数。
batch: 256

# Spark 分区数量
partition: 32
}

# 设置 Tag team 相关信息。
{
    name: team
    type: {
        source: hive
        sink: client
    }
    exec: "select teamid, name from basketball.team"
    fields: [name]
    nebula.fields: [name]
    vertex: {
        field: teamid
    }
    batch: 256
    partition: 32
}

]

# 处理边数据
edges: [
    # 设置 Edge type follow 相关信息
    {
        # NebulaGraph 中对应的 Edge type 名称。
        name: follow
        type: {
            # 指定数据源文件格式，设置为 hive。
            source: hive
            # 指定边数据导入 NebulaGraph 的方式。
            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }
        # 设置读取数据库 basketball 中 follow 表数据的 SQL 语句。
        exec: "select src_player, dst_player, degree from basketball.follow"
        # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
        # 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: [degree]
        nebula.fields: [degree]
        # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
        # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
        source: {
            field: src_player
        }
        target: {
            field: dst_player
        }
        # 指定一个列作为 rank 的源（可选）。
        #ranking: rank
        # 单批次写入 NebulaGraph 的最大数据条数。
        batch: 256
        # Spark 分区数量
        partition: 32
    }

    # 设置 Edge type serve 相关信息
    {
        name: serve
        type: {
            source: hive
            sink: client
        }
        exec: "select playerid, teamid, start_year, end_year from basketball.serve"
        fields: [start_year,end_year]
        nebula.fields: [start_year,end_year]
        source: {
            field: playerid
        }
        target: {
            field: teamid
        }
        # 指定一个列作为 rank 的源（可选）。
    }
]

```

```
#ranking: rank
batch: 256
partition: 32
}
}
```

步骤 4: 向NEBULAGRAPH导入数据

运行如下命令将 Hive 数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <hive_application.conf_path> -h
```



JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/hive_application.conf -h
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`, 确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 5: (可选) 验证数据

用户可以在NebulaGraph客户端 (例如 NebulaGraph Studio) 中执行查询语句, 确认数据是否已导入。例如:

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 6: (如有) 在NEBULAGRAPH中重建索引

导入数据后, 用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.11 导入 MaxCompute 数据

本文以一个示例说明如何使用 Exchange 将存储在 MaxCompute 上的数据导入 NebulaGraph。

数据集

本文以 **basketballplayer** 数据集为例。

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
- MaxCompute: 阿里云官方版本
- NebulaGraph: 3.4.1。使用 **Docker Compose** 部署。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经[安装部署NebulaGraph](#) 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经安装并开启 Hadoop 服务。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 在NebulaGraph中创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2: 修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 MaxCompute 数据源相关的配置。在本示例中，复制的文件名为 `maxcompute_application.conf`。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address: {
      # 以下为 NebulaGraph 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: ["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta: ["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
  }
}
```

```

execution: {
    retry: 3
}
error: {
    max: 32
    output: /tmp/errors
}
rate: {
    limit: 1024
    timeout: 1000
}
}

# 处理点
tags: [
    # 设置 Tag player 相关信息
    {
        name: player
        type: {
            # 指定数据源文件格式, 设置为 MaxCompute。
            source: maxcompute
            # 指定如何将点数据导入NebulaGraph: Client 或 SST。
            sink: client
        }
    }
]

# MaxCompute 的表名
table:player

# MaxCompute 的项目名
project:project

# MaxCompute 服务的 odpsUrl 和 tunnelUrl,
# 地址可在 https://help.aliyun.com/document\_detail/34951.html 查看。
odpsUrl:"http://service.cn-hangzhou.maxcompute.aliyun.com/api"
tunnelUrl:"http://dt.cn-hangzhou.maxcompute.aliyun.com"

# MaxCompute 服务的 accessKeyId 和 accessKeySecret。
accessKeyId:xxx
accessKeySecret:xxx

# MaxCompute 表的分区描述, 该配置可选。
partitionSpec:"dt='partition1'"

# MaxCompute 的 Spark 连接器在读取 MaxCompute 数据时使用的分区数。默认为1, 该配置可选。
numPartitions:100

# 请确保 SQL 语句中的表名和上方 table 的值相同, 该配置可选。
sentence:"select id, name, age, playerid from player where id < 10"

# 在 fields 里指定 player 表中的列名称, 其对应的 value 会作为 NebulaGraph 中指定属性。
# 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:[name, age]
nebula.fields:[name, age]

# 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
vertex:{
    field: playerid
}

# 单批次写入 NebulaGraph 的数据条数。
batch: 256

# Spark 分区数量
partition: 32
}

# 设置 Tag team 相关信息。
{
    name: team
    type: {
        source: maxcompute
        sink: client
    }
    table:team
    project:project
    odpsUrl:"http://service.cn-hangzhou.maxcompute.aliyun.com/api"
    tunnelUrl:"http://dt.cn-hangzhou.maxcompute.aliyun.com"
    accessKeyId:xxx
    accessKeySecret:xxx
    partitionSpec:"dt='partition1'"
    sentence:"select id, name, teamid from team where id < 10"
    fields:[name]
    nebula.fields:[name]
    vertex:{
        field: teamid
    }
    batch: 256
    partition: 32
}
]

# 处理边数据
edges: [
    # 设置 Edge type follow 相关信息
]

```

```
{
  # NebulaGraph 中对应的 Edge type 名称。
  name: follow

  type:{
    # 指定数据源文件格式，设置为 MaxCompute。
    source:maxcompute

    # 指定边数据导入 NebulaGraph 的方式，
    # 指定如何将点数据导入NebulaGraph：Client 或 SST。
    sink:client
  }

  # MaxCompute 的表名
  table:follow

  # MaxCompute 的项目名
  project:project

  # MaxCompute 服务的 opdsUrl 和 tunnelUrl,
  # 地址可在 https://help.aliyun.com/document_detail/34951.html 查看。
  opdsUrl:"http://service.cn-hangzhou.maxcompute.aliyun.com/api"
  tunnelUrl:"http://dt.cn-hangzhou.maxcompute.aliyun.com"

  # MaxCompute 服务的 accessKeyId 和 accessKeySecret。
  accessKeyId:xxx
  accessKeySecret:xxx

  # MaxCompute 表的分区描述，该配置可选。
  partitionSpec:"dt='partition1'"

  # 请确保 SQL 语句中的表名和上方 table 的值相同，该配置可选。
  sentence:"select * from follow"

  # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
  # 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:[degree]
  nebula.fields:[degree]

  # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
  source:{
    field: src_player
  }

  # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
  target:{
    field: dst_player
  }

  # 指定一个列作为 rank 的源（可选）。
  #ranking: rank

  # Spark 分区数量
  partition:10

  # 单批次写入 NebulaGraph 的数据条数。
  batch:10
}

# 设置 Edge type serve 相关信息
{
  name: serve
  type:{
    source:maxcompute
    sink:client
  }
  table:serve
  project:project
  opdsUrl:"http://service.cn-hangzhou.maxcompute.aliyun.com/api"
  tunnelUrl:"http://dt.cn-hangzhou.maxcompute.aliyun.com"
  accessKeyId:xxx
  accessKeySecret:xxx
  partitionSpec:"dt='partition1'"
  sentence:"select * from serve"
  fields:[start_year,end_year]
  nebula.fields:[start_year,end_year]
  source:{
    field: playerid
  }
  target:{
    field: teamid
  }

  # 指定一个列作为 rank 的源（可选）。
  #ranking: rank

  partition:10
  batch:10
}
]
```

步骤 3: 向**NEBULAGRAPH**导入数据

运行如下命令将 MaxCompute 数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <maxcompute_application.conf_path>
```



JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/maxcompute_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`, 确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4: (可选) 验证数据

用户可以在NebulaGraph客户端 (例如 NebulaGraph Studio) 中执行查询语句, 确认数据是否已导入。例如:

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5: (如有) 在**NEBULAGRAPH**中重建索引

导入数据后, 用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.12 导入 Pulsar 数据

本文简单说明如何使用 Exchange 将存储在 Pulsar 上的数据导入 NebulaGraph。

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
- NebulaGraph：3.4.1。使用 [Docker Compose 部署](#)。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经[安装部署NebulaGraph](#) 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经安装并开启 Pulsar 服务。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 在NebulaGraph中创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2: 修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 Pulsar 数据源相关的配置。在本示例中，复制的文件名为 pulsar_application.conf。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  #NebulaGraph相关配置
  nebula: {
    address: {
      # 以下为NebulaGraph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: ["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的NebulaGraph在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta: ["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有NebulaGraph相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 填写NebulaGraph中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
  }
}
```

```

execution: {
    retry: 3
}
error: {
    max: 32
    output: /tmp/errors
}
rate: {
    limit: 1024
    timeout: 1000
}
}

# 处理点
tags: [
    # 设置 Tag player 相关信息。
    {
        #NebulaGraph中对应的 Tag 名称。
        name: player
        type: {
            # 指定数据源文件格式，设置为 Pulsar。
            source: pulsar
            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }
        # Pulsar 服务器地址。
        service: "pulsar://127.0.0.1:6650"
        # 连接 pulsar 的 admin.url。
        admin: "http://127.0.0.1:8081"
        # Pulsar 的选项，可以从 topic、topics 和 topicsPattern 选择一个进行配置。
        options: {
            topics: "topic1,topic2"
        }
    }

    # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为NebulaGraph中指定属性。
    # 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: [age,name]
    nebula.fields: [age,name]

    # 指定表中某一列数据为NebulaGraph中点 VID 的来源。
    vertex: {
        field:playerid
    }

    # 单批次写入NebulaGraph的数据条数。
    batch: 10

    # Spark 分区数量
    partition: 10
    # 读取消息的间隔。单位：秒。
    interval.seconds: 10
}

# 设置 Tag team 相关信息。
{
    name: team
    type: {
        source: pulsar
        sink: client
    }
    service: "pulsar://127.0.0.1:6650"
    admin: "http://127.0.0.1:8081"
    options: {
        topics: "topic1,topic2"
    }
    fields: [name]
    nebula.fields: [name]
    vertex: {
        field:teamid
    }
    batch: 10
    partition: 10
    interval.seconds: 10
}

]

# 处理边数据
edges: [
    # 设置 Edge type follow 相关信息
    {
        #NebulaGraph中对应的 Edge type 名称。
        name: follow

        type: {
            # 指定数据源文件格式，设置为 Pulsar。
            source: pulsar
            # 指定边数据导入NebulaGraph的方式。
            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }

        # Pulsar 服务器地址。
        service: "pulsar://127.0.0.1:6650"
    }
]

```

```

# 连接 pulsar 的 admin.url。
admin: "http://127.0.0.1:8081"
# Pulsar 的选项，可以从 topic、topics 和 topicsPattern 选择一个进行配置。
options: {
    topics: "topic1,topic2"
}

# 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为NebulaGraph中指定属性。
# 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: [degree]
nebula.fields: [degree]

# 在 source 里，将 follow 表中某一列作为边的起始点数据源。
# 在 target 里，将 follow 表中某一列作为边的目的点数据源。
source:{
    field:src_player
}

target:{
    field:dst_player
}

# 指定一个列作为 rank 的源（可选）。
#ranking: rank

# 单批次写入NebulaGraph的数据条数。
batch: 10

# Spark 分区数量
partition: 10

# 读取消息的间隔。单位：秒。
interval.seconds: 10
}

# 设置 Edge type serve 相关信息
{
    name: serve
    type: {
        source: Pulsar
        sink: client
    }
    service: "pulsar://127.0.0.1:6650"
    admin: "http://127.0.0.1:8081"
    options: {
        topics: "topic1,topic2"
    }

    fields: [start_year,end_year]
    nebula.fields: [start_year,end_year]
    source:{
        field:playerid
    }

    target:{
        field:teamid
    }

    # 指定一个列作为 rank 的源（可选）。
    #ranking: rank

    batch: 10
    partition: 10
    interval.seconds: 10
}
]
}

```

步骤 3: 向NEBULAGRAPH导入数据

运行如下命令将 Pulsar 数据导入到NebulaGraph中。关于参数的说明，请参见[导入命令参数](#)。

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <pulsar_application.conf_path>
```



JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/pulsar_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4: （可选）验证数据

用户可以在NebulaGraph客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5: （如有）在NEBULAGRAPH中重建索引

导入数据后，用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.13 导入 Kafka 数据

本文简单说明如何使用 Exchange 将存储在 Kafka 上的数据导入 NebulaGraph。

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
- NebulaGraph：3.4.1。使用 [Docker Compose 部署](#)。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经[安装部署NebulaGraph](#) 并获取如下信息：
- Graph 服务和 Meta 服务的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经安装并开启 Kafka 服务。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析数据，按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 在NebulaGraph中创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2: 修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 Kafka 数据源相关的配置。在本示例中，复制的文件名为 kafka_application.conf。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address: {
      # 以下为 NebulaGraph 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: ["127.0.0.1:9699"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta: ["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
  }
}
```

```

execution: {
    retry: 3
}
error: {
    max: 32
    output: /tmp/errors
}
rate: {
    limit: 1024
    timeout: 1000
}
}

# 处理点
tags: [
    # 设置 Tag player 相关信息。
    {
        # NebulaGraph 中对应的 Tag 名称。
        name: player
        type: {
            # 指定数据源文件格式，设置为 Kafka。
            source: kafka
            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }
        # Kafka 服务器地址。
        service: "127.0.0.1:9092"
        # 消息类别。
        topic: "topic_name1"

        # 在 fields 里指定 Kafka value 中的字段名称，多个字段用英文逗号 (,) 隔开。Spark Structured Streaming 读取 Kafka 数据后会将其以 JSON 格式存储于 value 字段中，而这里的 fields 要配置 JSON 的 key 名。示例如下：
        fields: [personName, personAge]
        # 设置与 fields 中的 key 对应的 NebulaGraph 属性名，key 的 value 将保存为相应的属性值。下方设置会将 personName 的 value 保存到 NebulaGraph 中的 name 属性，personAge 的 value 则保存到 age 属性。
        nebula.fields: [name, age]

        # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
        # 这里的值 key 和上面的 key 重复，表示 key 既作为 VID，也作为属性 name。
        vertex: {
            field: personId
        }

        # 单批次写入 NebulaGraph 的数据条数。
        batch: 10

        # Spark 分区数量
        partition: 10
        # 读取消息的间隔。单位：秒。
        interval.seconds: 10
    }
}

# 设置 Tag team 相关信息。
{
    name: team
    type: {
        source: kafka
        sink: client
    }
    service: "127.0.0.1:9092"
    topic: "topic_name2"
    fields: [key]
    nebula.fields: [name]
    vertex: {
        field: teamId
    }
    batch: 10
    partition: 10
    interval.seconds: 10
}

]

# 处理边数据
edges: [
    # 设置 Edge type follow 相关信息
    {
        # NebulaGraph 中对应的 Edge type 名称。
        name: follow

        type: {
            # 指定数据源文件格式，设置为 Kafka。
            source: kafka
            # 指定边数据导入 NebulaGraph 的方式。
            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: client
        }

        # Kafka 服务器地址。
        service: "127.0.0.1:9092"
        # 消息类别。
        topic: "topic_name3"

        # 在 fields 里指定 Kafka value 中的字段名称，多个字段用英文逗号 (,) 隔开。Spark Structured Streaming 读取 Kafka 数据后会将其以 JSON 格式存储于 value 字段中，而这里的 fields 要配置 JSON 的 key 名。示例如下：
    }
]

```

```

fields: [degree]
# 设置与 fields 中的 key 对应的 NebulaGraph 属性名, key 的 value 将保存为相应的属性值。下方设置会将 degree 的 value 保存到 NebulaGraph 中的 degree 属性。
nebula.fields: [degree]

# 在 source 里, 将 topic 中某一列作为边的起始点数据源。
# 在 target 里, 将 topic 中某一列作为边的目的点数据源。
source:{
    field:srcPersonId
}

target:{
    field:dstPersonId
}

# 指定一个列作为 rank 的源 (可选)。
#ranking: rank

# 单批次写入 NebulaGraph 的数据条数。
batch: 10

# Spark 分区数量
partition: 10

# 读取消息的间隔。单位：秒。
interval.seconds: 10
}

# 设置 Edge type serve 相关信息
{
    name: serve
    type: {
        source: kafka
        sink: client
    }
    service: "127.0.0.1:9092"
    topic: "topic_name4"

    fields: [startYear,endYear]
    nebula.fields: [start_year,end_year]
    source:{
        field:personId
    }

    target:{
        field:teamId
    }

    # 指定一个列作为 rank 的源 (可选)。
    #ranking: rank

    batch: 10
    partition: 10
    interval.seconds: 10
}
]
}

```

步骤 3: 向NEBULAGRAPH导入数据

运行如下命令将 Kafka 数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <kafka_application.conf_path>
```



JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/kafka_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`, 确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4: (可选) 验证数据

用户可以在NebulaGraph客户端 (例如 NebulaGraph Studio) 中执行查询语句, 确认数据是否已导入。例如:

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5: (如有) 在**NEBULAGRAPH**中重建索引

导入数据后, 用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.14 导入通用 JDBC 数据

JDBC 数据是指用 JDBC 接口访问的各类数据库的数据的统称。本文以 MySQL 数据库为例说明如何使用 Exchange 将 JDBC 数据导入 NebulaGraph。

数据集

本文以 **basketballplayer** 数据集为例。

在本示例中，该数据集已经存入 MySQL 中名为 basketball 的数据库中，以 player、team、follow 和 serve 四个表存储了所有点和边的信息。以下为各个表的结构。

```
mysql> desc player;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| playerid | int | YES | NULL |  |  |
| age | int | YES | NULL |  |  |
| name | varchar(30) | YES | NULL |  |  |
+-----+-----+-----+-----+
mysql> desc team;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| teamid | int | YES | NULL |  |  |
| name | varchar(30) | YES | NULL |  |  |
+-----+-----+-----+-----+
mysql> desc follow;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| src_player | int | YES | NULL |  |  |
| dst_player | int | YES | NULL |  |  |
| degree | int | YES | NULL |  |  |
+-----+-----+-----+-----+
mysql> desc serve;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| playerid | int | YES | NULL |  |  |
| teamid | int | YES | NULL |  |  |
| start_year | int | YES | NULL |  |  |
| end_year | int | YES | NULL |  |  |
+-----+-----+-----+-----+
```

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
- NebulaGraph: 3.4.1。使用 [Docker Compose 部署](#)。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- 已经编译 Exchange。详情请参见[编译 Exchange](#)。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 了解NebulaGraph中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 如果文件存储在 HDFS 上，需要确认 Hadoop 服务运行正常。
- 如果文件存储在本地且NebulaGraph是集群架构，需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 1：在NEBULAGRAPH中创建 SCHEMA

分析文件中的数据，按以下步骤在NebulaGraph中创建 Schema：

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 使用 NebulaGraph Console 创建一个图空间 **basketballplayer**，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见[快速开始](#)。

步骤 2：修改配置文件

编译 Exchange 后，复制 target/classes/application.conf 文件设置 JDBC 数据源相关的配置。在本示例中，复制的文件名为 `jdbc_application.conf`。各个配置项的详细说明请参见[配置说明](#)。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.4.0
    }
  }
}
```

```

driver: {
  cores: 1
  maxResultSize: 1G
}
executor: {
  memory:1G
}

cores: {
  max: 16
}
}

# NebulaGraph 相关配置
nebula: {
  address: {
    # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
    # 如果有多台服务器，地址之间用英文逗号 (,) 分隔。
    # 格式："ip1:port","ip2:port","ip3:port"
    graph: ["127.0.0.1:9669"]
    #任意一个 Meta 服务的地址。
    #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
    meta: ["127.0.0.1:9559"]
  }
}

# 指定拥有 NebulaGraph 写权限的用户名和密码。
user: root
pswd: nebulab

# 指定图空间名称。
space: basketballplayer
connection: {
  timeout: 3000
  retry: 3
}
execution: {
  retry: 3
}
error: {
  max: 32
  output: /tmp/errors
}
rate: {
  limit: 1024
  timeout: 1000
}
}

# 处理点
tags: [
  # 设置 Tag player 相关信息。
  {
    # 指定 NebulaGraph 中定义的 Tag 名称。
    name: player
    type: {
      # 指定数据源，使用 JDBC。
      source: jdbc
      # 指定如何将点数据导入NebulaGraph：Client 或 SST。
      sink: client
    }
  }
]

# JDBC 数据源的 URL。示例为 MySQL 数据库。
url:"jdbc:mysql://127.0.0.1:3306/basketball?useUnicode=true&characterEncoding=utf-8"

# JDBC 驱动。
driver:"com.mysql.cj.jdbc.Driver"

# 数据库用户名和密码。
user:root
password:"12345"

table:player
sentence:"select playerid, age, name from player order by playerid"

# (可选) 多连接读取参数 参见 https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html
partitionColumn:playerid # 可选。数值类型必须为数字、日期或时间戳。
lowerBound:1 # 可选
upperBound:5 # 可选
numPartitions:5 # 可选

fetchSize:2 # 每次请求数据库要读取的行数。

# 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
# 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: [age,name]
nebula.fields: [age,name]

# 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
vertex: {
  field:playerid
}

```

```

# 单批次写入 NebulaGraph 的数据条数。
batch: 256

# Spark 分区数量
partition: 32
}

# 设置 Tag team 相关信息。
{
  name: team
  type: {
    source: jdbc
    sink: client
  }
}

url:"jdbc:mysql://127.0.0.1:3306/basketball?useUnicode=true&characterEncoding=utf-8"
driver:"com.mysql.cj.jdbc.Driver"
user:root
password:"12345"
table:team
sentence:"select teamid, name from team order by teamid"
partitionColumn:teamid
lowerBound:1
upperBound:5
numPartitions:5
fetchSize:2

fields: [name]
nebula.fields: [name]
vertex: {
  field: teamid
}
batch: 256
partition: 32
}

]

# 处理边数据
edges: [
  # 设置 Edge type follow 相关信息
  {
    # NebulaGraph 中对应的 Edge type 名称。
    name: follow

    type: {
      # 指定数据源文件格式，设置为 JDBC。
      source: jdbc

      # 指定边数据导入 NebulaGraph 的方式,
      # 指定如何将点数据导入NebulaGraph: Client 或 SST。
      sink: client
    }

    url:"jdbc:mysql://127.0.0.1:3306/basketball?useUnicode=true&characterEncoding=utf-8"
    driver:"com.mysql.cj.jdbc.Driver"
    user:root
    password:"12345"
    table:follow
    sentence:"select src_player,dst_player,degree from follow order by src_player"
    partitionColumn:src_player
    lowerBound:1
    upperBound:5
    numPartitions:5
    fetchSize:2

    # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
    # 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: [degree]
    nebula.fields: [degree]

    # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
    # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
    source: {
      field: src_player
    }

    target: {
      field: dst_player
    }

    # 指定一个列作为 rank 的源（可选）。
    #ranking: rank

    # 单批次写入 NebulaGraph 的数据条数。
    batch: 256

    # Spark 分区数量
    partition: 32
  }

  # 设置 Edge type serve 相关信息
]

```

```
{
  name: serve
  type: {
    source: jdbc
    sink: client
  }
  url:"jdbc:mysql://127.0.0.1:3306/basketball?useUnicode=true&characterEncoding=utf-8"
  driver:"com.mysql.cj.jdbc.Driver"
  user:root
  password:"12345"
  table:serve
  sentence:"select playerid,teamid,start_year,end_year from serve order by playerid"
  partitionColumn:playerid
  lowerBound:1
  upperBound:5
  numPartitions:5
  batchSize:2

  fields: [start_year,end_year]
  nebula.fields: [start_year,end_year]
  source: {
    field: playerid
  }
  target: {
    field: teamid
  }

  # 指定一个列作为 rank 的源（可选）。
  #ranking: rank

  batch: 256
  partition: 32
}
]
```

步骤 4: 向NEBULAGRAPH导入数据

运行如下命令将 JDBC 数据导入到NebulaGraph中。关于参数的说明, 请参见[导入命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <jdbc_application.conf_path>
```

Note

JAR 包有两种获取方式: [自行编译](#)或者从 maven 仓库下载。

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/jdbc_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`, 确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 5: (可选) 验证数据

用户可以在NebulaGraph客户端 (例如 NebulaGraph Studio) 中执行查询语句, 确认数据是否已导入。例如:

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 6: (如有) 在NEBULAGRAPH中重建索引

导入数据后, 用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.4.15 导入 SST 文件数据

本文以一个示例说明如何将数据源的数据生成 SST（Sorted String Table）文件并保存在 HDFS 上，然后导入NebulaGraph，示例数据源是 CSV 文件。

注意事项

- 仅 Linux 系统支持导入 SST 文件。
- 不支持属性的 Default 值。
- 企业版 Exchange 3.4.0 不支持基于 **GEOGRAPHY** 类型的数据生成 SST 文件。

背景信息

Exchange 支持两种数据导入模式：

- 直接将数据源的数据通过 **nGQL** 语句的形式导入NebulaGraph。
- 将数据源的数据生成 SST 文件，然后借助 Console 将 SST 文件导入NebulaGraph。

下文将介绍生成 SST 文件并用其导入数据的适用场景、实现方法、前提条件、操作步骤等内容。

适用场景

- 适合在线业务，因为生成时几乎不会影响业务（只是读取 Schema），导入速度快。



导入期间（大约 10 秒）会阻塞对应空间的写操作，并且之后数小时内可能有历史数据整理，建议在业务低高峰期进行导入。

- 适合数据源数据量较大的场景，导入速度快。

实现方法

NebulaGraph底层使用 RocksDB 作为键值型存储引擎。RocksDB 是基于硬盘的存储引擎，提供了一系列 API 用于创建及导入 SST 格式的文件，有助于快速导入海量数据。

SST 文件是一个内部包含了任意长度的有序键值对集合的文件，用于高效地存储大量键值型数据。生成 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：

- Reader 从数据源中读取数据。
- sstProcessor 根据NebulaGraph的 Schema 信息生成 SST 文件，然后上传至 HDFS。SST 文件的格式请参见[数据存储格式](#)。
- sstWriter 打开一个文件并插入数据。生成 SST 文件时，Key 必须按照顺序写入。
- 生成 SST 文件之后，RocksDB 通过 `IngestExternalFile()` 方法将 SST 文件导入到NebulaGraph中。例如：

```
IngestExternalFileOptions info;
# 导入两个 SST 文件
Status s = db_->IngestExternalFile({"~/home/usr/file1.sst", "~/home/usr/file2.sst"}, info);
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。

数据集

本文以 **basketballplayer** 数据集为例。

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
- NebulaGraph: 3.4.1。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经[安装部署 NebulaGraph 3.4.1](#) 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有NebulaGraph写权限的用户名和密码。
- Meta 服务配置文件中的 `--ws_storage_http_port` 和 Storage 服务配置文件中的 `--ws_http_port` 一致。例如都为 19779。
- Graph 服务配置文件中的 `--ws_meta_http_port` 和 Meta 服务配置文件中的 `--ws_http_port` 一致。例如都为 19559。
- Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经[编译 Exchange](#)，或者直接[下载](#)编译完成的 jar 文件。本示例中使用 Exchange 3.4.0。
- 已经安装 Spark。
- 已经安装 JDK 1.8 或以上版本，并配置环境变量 `JAVA_HOME`。
- 确认 Hadoop 服务在所有部署 Storage 服务的机器上运行正常。

Note

- 如果需要生成其他数据源的 SST 文件，请参见相应数据源的文档，查看前提条件部分。
- 如果只需要生成 SST 文件，不需要在部署 Storage 服务的机器上安装 Hadoop 服务。
- 如需在 INGEST（数据导入）结束后自动移除 SST 文件，在 Storage 服务配置文件中增加 `--move_files=true`，该配置会让NebulaGraph在 INGEST 后将 SST 文件移动（mv）到 data 目录下。`--move_files` 的默认值为 `false`，此时NebulaGraph会复制（cp）SST 文件而不是移动。

操作步骤

步骤 1: 在NEBULAGRAPH中创建 SCHEMA

分析 CSV 文件中的数据, 按以下步骤在NebulaGraph中创建 Schema:

- 确认 Schema 要素。NebulaGraph中的 Schema 要素如下表所示。

要素	名称	属性
Tag	player	name string, age int
Tag	team	name string
Edge Type	follow	degree int
Edge Type	serve	start_year int, end_year int

- 使用 NebulaGraph Console 创建一个图空间 **basketballplayer**, 并创建一个 Schema, 如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
(partition_num = 10, \
replica_factor = 1, \
vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息, 请参见[快速开始](#)。

步骤 2: 处理 CSV 文件

确认以下信息:

- 处理 CSV 文件以满足 Schema 的要求。



可以使用有表头或者无表头的 CSV 文件。

- 获取 CSV 文件存储路径。

步骤 3: 修改配置文件

编译 Exchange 后, 复制 target/classes/application.conf 文件设置相关配置。在本示例中, 复制的文件名为 sst_application.conf。各个配置项的详细说明请参见[配置说明](#)。

```
{
# Spark 相关配置
spark: {
  app: {
    name: NebulaGraph Exchange 3.4.0
  }
}

master:local

driver: {
  cores: 1
  maxResultSize: 1G
}

executor: {
```

```

        memory:1G
    }

cores:{
    max: 16
}
}

#NebulaGraph相关配置
nebula: {
    address:{ 
        graph:["127.0.0.1:9669"]
        #任意一个 Meta 服务的地址。
        #如果您的NebulaGraph在虚拟网络中，如k8s，请配置 Leader Meta的地址。
        meta:["127.0.0.1:9559"]
    }
    user: root
    pswd: nebula
    space: basketballplayer

# SST 文件相关配置
path:{ 
    # 本地临时存放生成的 SST 文件的目录
    local:"/tmp"

    # SST 文件在 HDFS 的存储路径
    remote:"/sst"

    # HDFS 的 NameNode 地址
    hdfs.namenode: "hdfs://*.*.*:9000"
}

# 客户端连接参数
connection: {
    # socket 连接、执行的超时时间，单位：毫秒。
    timeout: 30000
}

error: {
    # 最大失败数，超过后会退出应用程序。
    max: 32
    # 失败的导入作业将记录在输出路径中。
    output: /tmp/errors
}

# 使用Google Guava RateLimiter 来限制发送到NebulaGraph的请求。
rate: {
    # RateLimiter 的稳定吞吐量。
    limit: 1024

    # 从 RateLimiter 获取允许的超时时间，单位：毫秒
    timeout: 1000
}

# 处理点
tags: [
    # 设置 Tag player 相关信息。
    {
        # 指定NebulaGraph中定义的 Tag 名称。
        name: player
        type: {
            # 指定数据源，使用 CSV。
            source: csv

            # 指定如何将点数据导入NebulaGraph：Client 或 SST。
            sink: sst
        }

        # 指定 CSV 文件的路径。
        # 文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"dfs://ip:port/xx/xx.csv"。
        path: "hdfs://*.*.*:9000/dataset/vertex_player.csv"

        # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
        # 如果 CSV 文件有表头，则使用实际的列名。
        fields: [_c1, _c2]

        # 指定NebulaGraph中定义的属性名称。
        # fields 与 nebula.fields 的顺序必须一一对应。
        nebula.fields: [age, name]

        # 指定一个列作为 VID 的源。
        # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
        # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
        vertex: {
            field:_c0
        }

        # 指定的分隔符。默认值为英文逗号 (,) 。
        separator: ","

        # 如果 CSV 文件有表头，请将 header 设置为 true。
        # 如果 CSV 文件没有表头，请将 header 设置为 false。默认值为 false。
        header: false
    }
}

```

```

# 指定单批次写入NebulaGraph的最大点数量。
batch: 256

# 指定 Spark 分片数量。
partition: 32

# 生成 SST 文件时是否要基于NebulaGraph中图空间的 partition 进行数据重分区。
repartitionWithNebula: false
}

# 设置 Tag team 相关信息。
{
  # 指定NebulaGraph中定义的 Tag 名称。
  name: team
  type: {
    # 指定数据源，使用 CSV。
    source: csv

    # 指定如何将点数据导入NebulaGraph：Client 或 SST。
    sink: sst
  }

  # 指定 CSV 文件的路径。
  # 文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"dfs://ip:port/xx/xx.csv"。
  path: "hdfs://*.*.*:9000/dataset/vertex_team.csv"

  # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
  # 如果 CSV 文件有表头，则使用实际的列名。
  fields: [_c1]

  # 指定NebulaGraph中定义的属性名称。
  # fields 与 nebula.fields 的顺序必须一一对应。
  nebula.fields: [name]

  # 指定一个列作为 VID 的源。
  # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
  # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
  vertex: {
    field: _c0
  }

  # 指定的分隔符。默认值为英文逗号 (,)。
  separator: ","

  # 如果 CSV 文件有表头，请将 header 设置为 true。
  # 如果 CSV 文件没有表头，请将 header 设置为 false。默认值为 false。
  header: false

  # 指定单批次写入NebulaGraph的最大点数量。
  batch: 256

  # 指定 Spark 分片数量。
  partition: 32

  # 生成 SST 文件时是否要基于NebulaGraph中图空间的 partition 进行数据重分区。
  repartitionWithNebula: false
}

# 如果需要添加更多点，请参考前面的配置进行添加。
]
# 处理边
edges: [
  # 设置 Edge type follow 相关信息。
  {
    # 指定NebulaGraph中定义的 Edge type 名称。
    name: follow
    type: {
      # 指定数据源，使用 CSV。
      source: csv

      # 指定如何将点数据导入NebulaGraph：Client 或 SST。
      sink: sst
    }

    # 指定 CSV 文件的路径。
    # 文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"dfs://ip:port/xx/xx.csv"。
    path: "hdfs://*.*.*:9000/dataset/edge_follow.csv"

    # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
    # 如果 CSV 文件有表头，则使用实际的列名。
    fields: [_c2]

    # 指定NebulaGraph中定义的属性名称。
    # fields 与 nebula.fields 的顺序必须一一对应。
    nebula.fields: [degree]

    # 指定一个列作为起始点和目的点的源。
    # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
    # 目前，NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
    source: {
      field: _c0
    }

    target: {
  
```

```

        field: _c1
    }

    # 指定的分隔符。默认值为英文逗号 (,)。
    separator: ","

    # 指定一个列作为 rank 的源 (可选)。

    #ranking: rank

    # 如果 CSV 文件有表头, 请将 header 设置为 true。
    # 如果 CSV 文件没有表头, 请将 header 设置为 false。默认值为 false。
    header: false

    # 指定单批次写入NebulaGraph的最大边数量。
    batch: 256

    # 指定 Spark 分片数量。
    partition: 32

    # 生成 SST 文件时是否要基于NebulaGraph中图空间的 partition 进行数据重分区。
    repartitionWithNebula: false
}

# 设置 Edge type serve 相关信息。
{
    # 指定NebulaGraph中定义的 Edge type 名称。
    name: serve
    type: {
        # 指定数据源, 使用 CSV。
        source: csv

        # 指定如何将点数据导入NebulaGraph : Client 或 SST。
        sink: sst
    }

    # 指定 CSV 文件的路径。
    # 文件存储在 HDFS 上, 用双引号括起路径, 以 hdfs://开头, 例如"dfs://ip:port/xx/xx.csv"。
    path: "hdfs://*.*.*:9000/dataset/edge_serve.csv"

    # 如果 CSV 文件没有表头, 使用 [_c0, _c1, _c2, ..., _cn] 表示其表头, 并将列指示为属性值的源。
    # 如果 CSV 文件有表头, 则使用实际的列名。
    fields: [_c2,_c3]

    # 指定NebulaGraph中定义的属性名称。
    # fields 与 nebula.fields 的顺序必须一一对应。
    nebula.fields: [start_year, end_year]

    # 指定一个列作为起始点和目的点的源。
    # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
    # 目前, NebulaGraph 3.4.1仅支持字符串或整数类型的 VID。
    source: {
        field: _c0
    }
    target: {
        field: _c1
    }

    # 指定的分隔符。默认值为英文逗号 (,)。
    separator: ","

    # 指定一个列作为 rank 的源 (可选)。
    #ranking: _c5

    # 如果 CSV 文件有表头, 请将 header 设置为 true。
    # 如果 CSV 文件没有表头, 请将 header 设置为 false。默认值为 false。
    header: false

    # 指定单批次写入NebulaGraph的最大边数量。
    batch: 256

    # 指定 Spark 分片数量。
    partition: 32

    # 生成 SST 文件时是否要基于NebulaGraph中图空间的 partition 进行数据重分区。
    repartitionWithNebula: false
}

]
# 如果需要添加更多边, 请参考前面的配置进行添加。
}

```

步骤 4: 生成 SST 文件

运行如下命令将 CSV 源文件生成为 SST 文件。关于参数的说明, 请参见[命令参数](#)。

```
`${SPARK_HOME}/bin/spark-submit --master "local" --conf spark.sql.shuffle.partitions=<shuffle_concurrency> --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-3.4.0.jar_path> -c <sst_application.conf_path>
```

Note

生成 SST 文件时，会涉及到 Spark 的 shuffle 操作，请注意在提交命令中增加 spark.sql.shuffle.partitions 的配置。

Note

JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
 ${SPARK_HOME}/bin/spark-submit --master "local" --conf spark.sql.shuffle.partitions=200 --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange-3.4.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/sst_application.conf
```

任务执行完成后，可以在 HDFS 上的 /sst 目录（nebula.path.remote 参数指定）内查看到生成的 SST 文件。

Note

如果对 Schema 有修改操作，例如重建图空间、修改 Tag、修改 Edge type 等，需要重新生成 SST 文件，因为 SST 文件会验证 Space ID、Tag ID、Edge ID 等信息。

步骤 5：导入 SST 文件

Note

导入前请确认以下信息：

- 确认所有部署 Storage 服务的机器上都已部署 Hadoop 服务，并配置 HADOOP_HOME 和 JAVA_HOME。
- Meta 服务配置文件中的 --ws_storage_http_port（如果没有，请手动添加）和 Storage 服务配置文件中的 --ws_http_port 一致。例如都为 19779。
- Graph 服务配置文件中的 --ws_meta_http_port（如果没有，请手动添加）和 Meta 服务配置文件中的 --ws_http_port 一致。例如都为 19559。

使用客户端工具连接NebulaGraph，按如下操作导入 SST 文件：

1. 执行命令选择之前创建的图空间。

```
nebula> USE basketballplayer;
```

2. 执行命令下载 SST 文件：

```
nebula> SUBMIT JOB DOWNLOAD HDFS "hdfs://<hadoop_address>:<hadoop_port>/<sst_file_path>";
```

示例：

```
nebula> SUBMIT JOB DOWNLOAD HDFS "hdfs://*.**.*:9000/sst";
```

3. 执行命令导入 SST 文件：

```
nebula> SUBMIT JOB INGEST;
```

Note

- 如果需要重新下载，请在NebulaGraph安装路径内的 data/storage/nebula 目录内，将对应 Space ID 目录内的 download 文件夹删除，然后重新下载 SST 文件。如果图空间是多副本，保存副本的所有机器都需要删除 download 文件夹。
- 如果导入时出现问题需要重新导入，重新执行 SUBMIT JOB INGEST；即可。

步骤 6: (可选) 验证数据

用户可以在NebulaGraph客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 **SHOW STATS** 查看统计数据。

步骤 7: (如有) 在**NEBULAGRAPH**中重建索引

导入数据后，用户可以在NebulaGraph中重新创建并重建索引。详情请参见[索引介绍](#)。

最后更新: May 4, 2023

16.5 Exchange 常见问题

16.5.1 编译问题

Q: 部分非 central 仓库的包下载失败，报错 `Could not resolve dependencies for project xxx`

请检查 Maven 安装目录下 `libexec/conf/settings.xml` 文件的 `mirror` 部分：

```
<mirror>
  <id>alimaven</id>
  <mirrorOf>central</mirrorOf>
  <name>aliyun maven</name>
  <url>http://maven.aliyun.com/nexus/content/repositories/central/</url>
</mirror>
```

检查 `mirrorOf` 的值是否配置为 `*`，如果为 `*`，请修改为 `central` 或 `*,!SparkPackagesRepo,!bintray-streamnative-maven`。

原因：Exchange 的 `pom.xml` 中有两个依赖包不在 Maven 的 `central` 仓库中，`pom.xml` 配置了这两个依赖所在的仓库地址。如果 `maven` 中配置的镜像地址对应的 `mirrorOf` 值为 `*`，那么所有依赖都会在 `central` 仓库下载，导致下载失败。

Q: 编译 Exchange 时无法下载 `SNAPSHOT` 包

现象：编译时提示 `Could not find artifact com.vesoft:client:jar:xxx-SNAPSHOT`。

原因：本地 `maven` 没有配置用于下载 `SNAPSHOT` 的仓库。`maven` 中默认的 `central` 仓库用于存放正式发布版本，而不是开发版本（`SNAPSHOT`）。

解决方案：在 `maven` 的 `setting.xml` 文件的 `profiles` 作用域内中增加以下配置：

```
<profile>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <repositories>
    <repository>
      <id>snapshots</id>
      <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>
  </repositories>
</profile>
```

16.5.2 执行问题

Q: 报错 `java.lang.ClassNotFoundException: com.vesoft.nebula.exchange.Exchange`

在 Yarn-Cluster 模式下提交任务，请参考如下命令，尤其是示例中的两个 `--conf`：

```
$SPARK_HOME/bin/spark-submit --class com.vesoft.nebula.exchange.Exchange \
--master yarn-cluster \
--files application.conf \
--conf spark.driver.extraClassPath=./ \
--conf spark.executor.extraClassPath=./ \
nebula-exchange-3.0.0.jar \
-c application.conf
```

Q: 报错 `method name xxx not found`

一般是端口配置错误，需检查 Meta 服务、Graph 服务、Storage 服务的端口配置。

Q: 报 `NoSuchMethod`、`MethodNotFound` 错误（`Exception in thread "main" java.lang.NoSuchMethodError` 等）

绝大多数是因为 JAR 包冲突和版本冲突导致的报错，请检查报错服务的版本，与 Exchange 中使用的版本进行对比，检查是否一致，尤其是 Spark 版本、Scala 版本、Hive 版本。

Q: Exchange 导入 Hive 数据时报错 Exception in thread "main" org.apache.spark.sql.AnalysisException: Table or view not found

检查提交 exchange 任务的命令中是否遗漏参数 -h，检查 table 和 database 是否正确，在 spark-sql 中执行用户配置的 exec 语句，验证 exec 语句的正确性。

Q: 运行时报错 com.facebook.thrift.protocol.TProtocolException: Expected protocol id xxx

请检查NebulaGraph服务端口配置是否正确。

- 如果是源码、RPM 或 DEB 安装，请配置各个服务的配置文件中 --port 对应的端口号。

- 如果是 docker 安装，请配置 docker 映射出来的端口号，查看方式如下：

在 nebula-docker-compose 目录下执行 docker-compose ps，例如：

Name	Command	State	Ports
nebula-docker-compose_graphd_1	/usr/local/nebula/bin/nebu ...	Up (healthy)	0.0.0.0:33205->19669/tcp, 0.0.0.0:33204->19670/tcp, 0.0.0.0:9669->9669/tcp
nebula-docker-compose_metad0_1	./bin/nebula-metad --flagf ...	Up (healthy)	0.0.0.0:33165->19559/tcp, 0.0.0.0:33162->19560/tcp, 0.0.0.0:33167->9559/tcp, 9560/tcp
nebula-docker-compose_metad1_1	./bin/nebula-metad --flagf ...	Up (healthy)	0.0.0.0:33166->19559/tcp, 0.0.0.0:33163->19560/tcp, 0.0.0.0:33168->9559/tcp, 9560/tcp
nebula-docker-compose_metad2_1	./bin/nebula-metad --flagf ...	Up (healthy)	0.0.0.0:33161->19559/tcp, 0.0.0.0:33160->19560/tcp, 0.0.0.0:33164->9559/tcp, 9560/tcp
nebula-docker-compose_storaged0_1	./bin/nebula-storaged --fl ...	Up (healthy)	0.0.0.0:33180->19779/tcp, 0.0.0.0:33178->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:33183->9779/tcp, 9780/tcp
nebula-docker-compose_storaged1_1	./bin/nebula-storaged --fl ...	Up (healthy)	0.0.0.0:33175->19779/tcp, 0.0.0.0:33172->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:33177->9779/tcp, 9780/tcp
nebula-docker-compose_storaged2_1	./bin/nebula-storaged --fl ...	Up (healthy)	0.0.0.0:33184->19779/tcp, 0.0.0.0:33181->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:33185->9779/tcp, 9780/tcp

查看 Ports 列，查找 docker 映射的端口号，例如：

- Graph 服务可用的端口号是 9669。
- Meta 服务可用的端口号有 33167、33168、33164。
- Storage 服务可用的端口号有 33183、33177、33185。

Q: 运行时报错 Exception in thread "main" com.facebook.thrift.protocol.TProtocolException: The field 'code' has been assigned the invalid value -4

检查 Exchange 版本与NebulaGraph版本是否匹配，详细信息可参考[使用限制](#)。

Q: 将 Hive 中的数据导入NebulaGraph时出现乱码如何解决？

如果 Hive 中数据的属性值包含中文字符，可能出现该情况。解决方案是在导入命令中的 JAR 包路径前加上以下选项：

```
--conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8
--conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8
```

即：

```
<spark_install_path>/bin/spark-submit --master "local" \
--conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8 \
--conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8 \
--class com.vesoft.nebula.exchange.Exchange \
<nebula-exchange-3.x.y.jar_path> -c <application.conf_path>
```

如果是在 YARN 中，则用以下命令：

```
<spark_install_path>/bin/spark-submit \
--class com.vesoft.nebula.exchange.Exchange \
--master yarn-cluster \
--files <application.conf_path> \
--conf spark.driver.extraClassPath= ./ \
--conf spark.executor.extraClassPath= ./ \
--conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8 \
--conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8 \
<nebula-exchange-3.x.y.jar_path> \
-c application.conf
```

Q: Hive 数据导入时提示 schema 版本不一致

Spark 日志提示 `Hive Schema version 1.2.0 does not match metastore's schema version 2.1.0 Metastore is not upgraded or corrupt` 的原因是 Hive 环境中配置的 metastore schema 版本和 Spark 使用的 metastore 版本不一致。

解决方法:

1. 将 Hive 环境中存储 Hive metastore 信息的 MySQL version 信息更新为 Spark 中使用的 metastore 版本。

假设 Hive 在 MySQL 中存储 metastore 的数据库是 `hive`，需要按如下方式修改 `hive.VERSION` 表中的 `version` 字段：

```
update hive.VERSION set SCHEMA_VERSION="2.1.0" where VER_ID=1
```

2. 在 Hive 环境的 `hive-site.xml` 文件中增加如下配置：

```
<property>
<name>hive.metastore.schema.verification</name>
<value>false</value>
</property>
```

3. 重启 Hive。

Q: 生成 SST 时提示 `org.rocksdb.RocksDBException: While open a file for appending: /path/sst/1-xxx.sst: No such file or directory`

排查方法:

1. 检查 `/path` 是否存在，如没有或者路径设置错误，创建或修正路径。
2. 检查 Spark 在每台机器上的当前用户对 `/path` 是否有操作权限，如没有，添加权限。

16.5.3 配置问题

Q: 哪些配置项影响导入性能？

- `batch`: 每次发送给NebulaGraph服务的 nGQL 语句中包含的数据条数。
- `partition`: Spark 数据的分区数，表示数据导入的并发数。
- `nebula.rate`: 向NebulaGraph发送请求前先去令牌桶获取令牌。
 - `limit`: 表示令牌桶的大小。
 - `timeout`: 表示获取令牌的超时时间。

根据机器性能可适当调整这四项参数的值。如果在导入过程中，Storage 服务的 `leader` 变更，可以适当调小这四项参数的值，降低导入速度。

16.5.4 其他问题

Q: Exchange 支持哪些版本的NebulaGraph？

请参见 Exchange 的[使用限制](#)。

Q: 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NebulaGraph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NebulaGraph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NebulaGraph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`）。

Q: Exchange 传输数据的性能如何?

Exchange 的性能测试数据和测试方法参见 [NebulaGraph Exchange test result](#)。

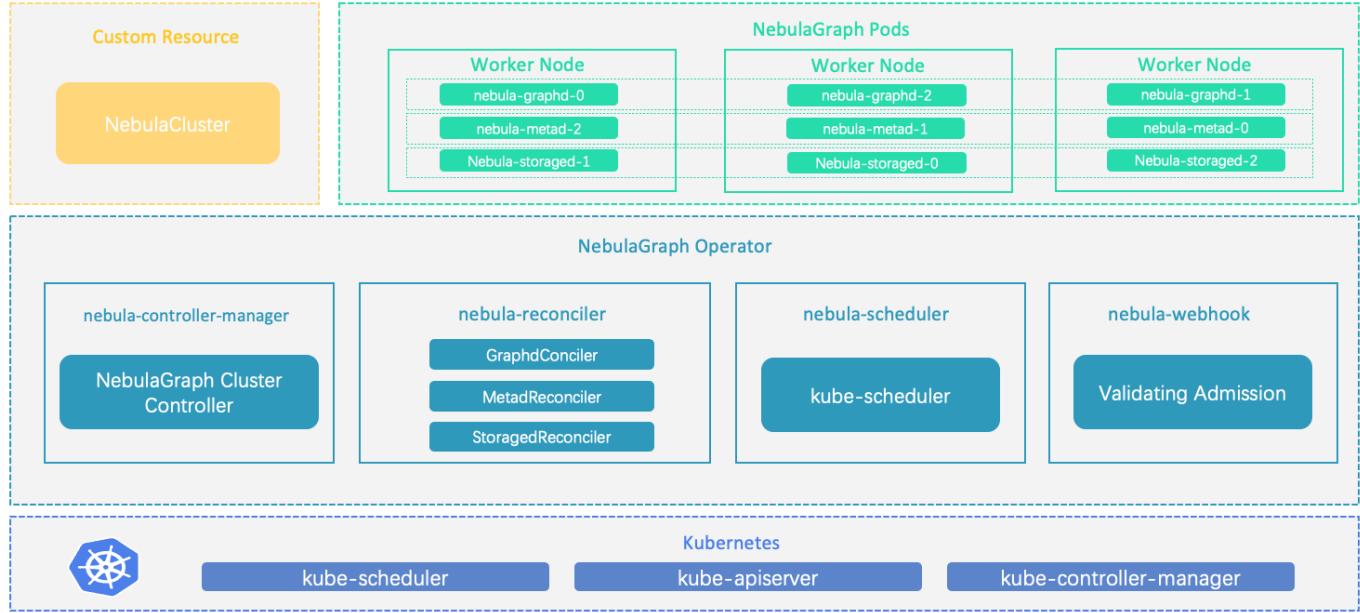
最后更新: May 4, 2023

17. NebulaGraph Operator

17.1 什么是 NebulaGraph Operator

17.1.1 基本概念

NebulaGraph Operator 是用于在 **Kubernetes** 系统上自动化部署和运维 **NebulaGraph** 集群的工具。依托于 **Kubernetes** 扩展机制，NebulaGraph 将其运维领域的知识全面注入至 **Kubernetes** 系统中，让 NebulaGraph 成为真正的云原生图数据库。



17.1.2 工作原理

对于 **Kubernetes** 系统内不存在的资源类型，用户可以通过添加自定义 API 对象的方式注册，常见的方法是使用 **CustomResourceDefinition (CRD)**。

NebulaGraph Operator 将 NebulaGraph 集群的部署管理抽象为 CRD。通过结合多个内置的 API 对象，包括 StatefulSet、Service 和 ConfigMap，NebulaGraph 集群的日常管理和维护被编码为一个控制循环。在 **Kubernetes** 系统内，每一种内置资源对象，都运行着一个特定的控制循环，将它的实际状态通过事先规定好的编排动作，逐步调整为最终的期望状态。当一个 CR 实例被提交时，NebulaGraph Operator 会根据控制流程驱动数据库集群进入最终状态。

17.1.3 功能介绍

NebulaGraph Operator 已具备的功能如下：

- 集群创建和卸载：NebulaGraph Operator 简化了用户部署和卸载集群的过程。用户只需提供对应的 CR 文件，NebulaGraph Operator 即可快速创建或者删除一个对应的 NebulaGraph 集群。更多信息参考[使用 Kubectl 部署 NebulaGraph 集群](#)或者[使用 Helm 部署 NebulaGraph 集群](#)。
- 集群扩容和缩容：通过在控制循环中调用 NebulaGraph 原生提供的扩缩容接口，NebulaGraph 封装 NebulaGraph Operator 实现了扩缩容的逻辑，用户可以通过 YAML 配置进行简单的扩缩容，且保证数据的稳定性。更多信息参考[使用 Kubeclt 扩缩容集群](#)或[使用 Helm 扩缩容集群](#)。
- 集群升级：支持升级 3.0.0 版的 NebulaGraph 集群至 3.4.0 版。
- 备份和恢复：支持备份 NebulaGraph 集群的数据至兼容 S3 协议的存储服务中并支持恢复数据至集群中。更多信息参考[使用 NebulaGraph Operator 备份和恢复数据](#)。
- 故障自愈：NebulaGraph Operator 调用 NebulaGraph 集群提供的接口，动态地感知服务状态。一旦发现异常，NebulaGraph Operator 自动进行容错处理。更多信息参考[故障自愈](#)。
- 均衡调度：基于调度器扩展接口，NebulaGraph Operator 提供的调度器可以将应用 Pods 均匀地分布在 NebulaGraph 集群中。

17.1.4 使用限制

版本限制

NebulaGraph Operator 不支持 v1.x 版本的 NebulaGraph，其与 NebulaGraph 版本的对应关系如下：

NebulaGraph 版本	NebulaGraph Operator 版本
3.0.0 ~ 3.4.1	1.3.0、1.4.0 ~ 1.4.2
3.0.0 ~ 3.3.x	1.0.0、1.1.0、1.2.0
2.5.x ~ 2.6.x	0.9.0
2.5.x	0.8.0

版本兼容性

- 1.x 版本的 NebulaGraph Operator 不兼容 3.x 以下版本的 NebulaGraph。
- 由于 0.9.0 版本的 NebulaGraph Operator 的日志盘和数据盘分开存储，因此用 0.9.0 版的 NebulaGraph Operator 管理通过 0.8.0 版本创建的 2.5.x 版本的 NebulaGraph 集群会导致兼容性问题。用户可以备份 2.5.x 版本的 NebulaGraph 集群，然后使用 0.9.0 版本的 Operator 创建 2.6.x 版本集群。

功能限制

NebulaGraph Operator 的扩缩容功能仅对企业版的 NebulaGraph 集群开放，不支持使用 NebulaGraph Operator 扩缩容社区版的 NebulaGraph 集群。

17.1.5 更新说明

Release

最后更新: May 4, 2023

17.2 使用流程

使用 NebulaGraph Operator 访问NebulaGraph集群服务的流程如下:

1. 安装 NebulaGraph Operator。
2. 创建NebulaGraph集群。
具体步骤参考[使用 Kubectl 部署NebulaGraph集群](#)或者[使用 Helm 部署NebulaGraph集群](#)。
3. 连接NebulaGraph服务。

最后更新: May 4, 2023

17.3 Macro Rendering Error

UndefinedError: 'operation' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 78, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'operation' is undefined
```

最后更新: May 4, 2023

17.4 部署 NebulaGraph

17.4.1 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 12, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

17.4.2 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 12, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

17.5 通过 Nebular Operator 连接NebulaGraph

使用 NebulaGraph Operator 创建NebulaGraph 集群后，用户可在NebulaGraph集群内部访问NebulaGraph，也可在集群外访问NebulaGraph。如果用户在集群内部，可以通过访问集群内的虚拟 IP 地址（ClusterIP）访问数据库。而如果用户在集群外部，则需要使用集群节点的公共 IP 地址或者通过部署 Nginx Ingress 控制器来访问NebulaGraph。IP 地址类型可以在创建集群的配置文件中，通过 `spec.graphd.service` 指定。关于 Service 的更多信息，参考[Service](#)。

17.5.1 前提条件

使用 NebulaGraph Operator 创建NebulaGraph集群。具体步骤参考[使用 Kubectl 部署NebulaGraph集群](#) 或者[使用 Helm 部署NebulaGraph集群](#)。

17.5.2 通过 NodePort 在NebulaGraph集群外部连接NebulaGraph

用户可创建 `NodePort` 类型的 `Service`，通过集群任一节点 IP 和暴露的节点端口，从集群外部访问集群内部的服务。用户也可以使用云厂商（例如 Azure、AWS 等）提供的负载均衡服务，设置 `Service` 的类型为 `LoadBalancer`，通过云厂商提供的负载均衡器的公网 IP 和端口，从集群外部访问集群内部的服务。

`NodePort` 类型的 `Service` 通过标签选择器 `spec.selector` 将前端的请求转发到带有标签 `app.kubernetes.io/cluster: <cluster-name>`、`app.kubernetes.io/component: graphd` 的 `Graphd` pod 中。

当根据集群[示例模板](#)，其中 `spec.graphd.service.type=NodePort`，创建 NebulaGraph 集群后，NebulaGraph Operator 会自动在同一命名空间下，创建名为 `<cluster-name>-graphd-svc`、类型为 `NodePort` 的 `Service`。通过任一节点 IP 和暴露的节点端口，可直接连接 NebulaGraph 数据库（参见下文第 4 步）。用户也可以根据自己的需求，创建自定义的 `Service`。

操作步骤如下：

1. 创建名为 graphd-nodeport-service.yaml 的文件。示例内容如下：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: graphd
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  name: nebula-graph-svc-nodeport
  namespace: default
spec:
  externalTrafficPolicy: Local
  ports:
    - name: thrift
      port: 9669
      protocol: TCP
      targetPort: 9669
    - name: http
      port: 19669
      protocol: TCP
      targetPort: 19669
  selector:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: graphd
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  type: NodePort # 设置 Service 类型为 NodePort。
```

- NebulaGraph默认使用 9669 端口为客户端提供服务。19669 为 Graph 服务的 HTTP 端口号。
- targetPort 的值为映射至 Pod 的端口，可自定义。

2. 执行以下命令使 Service 服务在集群中生效。

```
kubectl create -f graphd-nodeport-service.yaml
```

3. 查看 Service 中NebulaGraph映射至集群节点的端口。

```
kubectl get services -l app.kubernetes.io/cluster=<nebula> #<nebula>为变量值，请用实际集群名称替换。
```

返回：

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nebula-graph-svc-nodeport	NodePort	10.107.153.129	<none>	9669:32236/TCP,19669:31674/TCP,19670:31057/TCP	24h
...					

NodePort 类型的 Service 中，映射至集群节点的端口为 32236。

4. 使用节点 IP 和上述映射的节点端口连接 NebulaGraph。

```
kubectl run -ti --image vesoft/nebula-console:v3.4.0 --restart=Never -- <nebula_console_name> -addr <node_ip> -port <node_port> -u <username> -p <password>
```

示例如下：

```
kubectl run -ti --image vesoft/nebula-console:v3.4.0 --restart=Never -- nebula-console -addr 192.168.8.24 -port 32236 -u root -p vesoft
If you don't see a command prompt, try pressing enter.

(root@nebula) [(none)]>
```

- --image：为连接NebulaGraph的工具 NebulaGraph Console 的镜像。
- <nebula-console>：自定义的 Pod 名称。本示例为 nebula-console。
- -addr：NebulaGraph集群中任一节点 IP 地址。本示例为 192.168.8.24。
- -port：NebulaGraph映射至节点的端口。本示例为 32236。
- -u：NebulaGraph账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 root）。
- -p：用户名对应的密码。未启用身份认证时，密码可以填写任意字符。

17.5.3 在NebulaGraph集群内连接NebulaGraph

用户也可以创建 ClusterIP 类型的 Service，为集群内的其他 Pod 提供访问NebulaGraph的入口。通过该 Service 的 IP 和数据库 Graph 服务的端口号（9669），可连接NebulaGraph。更多信息，请参考 [ClusterIP](#)。

1. 创建名为 graphd-clusterip-service.yaml 的文件。示例内容如下：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: graphd
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  name: nebula-graph-svc
  namespace: default
spec:
  externalTrafficPolicy: Local
  ports:
    - name: thrift
      port: 9669
      protocol: TCP
      targetPort: 9669
    - name: http
      port: 19669
      protocol: TCP
      targetPort: 19669
  selector:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: graphd
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  type: ClusterIP # 设置 Service 类型为 ClusterIP。
```

- NebulaGraph默认使用 9669 端口为客户端提供服务。19669 为 Graph 服务的 HTTP 端口号。
- targetPort 的值为映射至 Pod 的端口，可自定义。

2. 执行以下命令使 Service 服务在集群中生效。

```
kubectl create -f graphd-clusterip-service.yaml
```

3. 查看 Service，命令如下：

```
$ kubectl get service -l app.kubernetes.io/cluster=<nebula> #<nebula>为变量值，请用实际集群名称替换。
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)          AGE
nebula-graph-svc   ClusterIP  10.98.213.34 <none>        9669/TCP,19669/TCP,19670/TCP   23h
...
```

4. 使用上述<cluster-name>-graphd-svc Service 的 IP 连接NebulaGraph:

```
kubectl run -ti --image vesoft/nebula-console:v3.4.0 --restart=Never -- <nebula_console_name> -addr <cluster_ip> -port <service_port> -u <username> -p <password>
```

示例：

```
kubectl run -ti --image vesoft/nebula-console:v3.4.0 --restart=Never -- nebula-console -addr 10.98.213.34 -port 9669 -u root -p vesoft
```

- --image：为连接NebulaGraph的工具 NebulaGraph Console 的镜像。
- <nebula-console>：自定义的 Pod 名称。
- -addr：连接 Graphd 服务的 IP 地址，即 ClusterIP 类型的 Service IP 地址。
- -port：连接 Graphd 服务的端口。默认端口为 9669。
- -u：NebulaGraph账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 root）。
- -p：用户名对应的密码。未启用身份认证时，密码可以填写任意字符。

如果返回以下内容，说明成功连接数据库：

```
If you don't see a command prompt, try pressing enter.
(root@nebula) [(none)]>
```

用户还可以使用完全限定域名（**FQDN**）连接数据库，域名格式为 <cluster-name>-graphd.<cluster-namespace>.svc.<CLUSTER_DOMAIN>，`CLUSTER_DOMAIN` 的默认值为 `cluster.local`。

```
kubectl run -ti --image vesoft/nebula-console:v3.4.0 --restart=Never -- <nebula_console_name> -addr <cluster_name>-graphd-svc.default.svc.cluster.local -port <service_port> -u <username> -p <password>
```

- `<service_port>` 为 Graph 服务默认的端口 9669。

17.5.4 通过 Ingress 在NebulaGraph集群外部连接NebulaGraph

当集群中有多个 Pod 时，为每个 Pod 分别提供服务会变得非常困难和繁琐，而使用 Ingress 可以轻松解决这个问题。Ingress 可以将流量路由到集群内部的多个 Pod。

Nginx Ingress 是 Kubernetes 中的一个 Ingress 控制器（Controller），是对 Kubernetes Ingress 资源的一个实现，通过 Watch 机制感知 Kubernetes 集群中的 Ingress 资源。它将这些 Ingress 规则转换为 Nginx 配置并启动一个 Nginx 实例来处理流量。

用户可以通过 HostNetwork 和 DaemonSet 组合的模式使用 Nginx Ingress 从集群外部连接NebulaGraph集群。

由于使用 HostNetwork，Nginx Ingress 的 Pod 就不能被调度在同一个节点上。为了避免监听端口冲突，可以事先选择一些节点并将其标记为边缘节点，专门用于部署 Nginx Ingress。然后 Nginx Ingress 以 DaemonSet 模式部署在这些节点上。

由于 Ingress 不支持 TCP 或 UDP 服务，为此 `nginx-ingress-controller` 使用 `--tcp-services-configmap` 和 `--udp-services-configmap` 参数指向一个 ConfigMap，该 ConfigMap 中的键指需要使用的外部端口，值指要公开的服务的格式，值的格式为 `<命名空间/服务名称>:<服务端口>`。

例如指向名为 `tcp-services` 的 ConfigMap 的配置如下：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: tcp-services
  namespace: nginx-ingress
data:
  9769: "default/nebula-graphd-svc:9669"
```

操作步骤如下：

1. 创建名为 nginx-ingress-daemonset-hostnetwork.yaml 的文件。

单击 [nginx-ingress-daemonset-hostnetwork.yaml](#) 查看完整的 YAML 示例内容。



上述 YAML 中的资源对象均使用 nginx-ingress 命名空间。用户可执行 `kubectl create namespace nginx-ingress` 创建命名空间，或者自定义其他命名空间。

2. 为任一节点（本示例使用的节点名为 worker2，IP 为 192.168.8.160）打上标签，以运行上述 YAML 文件中名为 nginx-ingress-controller 的 DaemonSet。

```
kubectl label node worker2 nginx-ingress=true
```

3. 执行以下命令使 Nginx Ingress 在集群中生效。

```
kubectl create -f nginx-ingress-daemonset-hostnetwork.yaml
```

返回：

```
configmap/nginx-ingress-controller created
configmap/tcp-services created
serviceaccount/nginx-ingress created
serviceaccount/nginx-ingress-backend created
clusterrole.rbac.authorization.k8s.io/nginx-ingress created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress created
role.rbac.authorization.k8s.io/nginx-ingress created
rolebinding.rbac.authorization.k8s.io/nginx-ingress created
service/nginx-ingress-controller-metrics created
service/nginx-ingress-default-backend created
service/nginx-ingress-proxy-tcp created
daemonset.apps/nginx-ingress-controller created
```

成功部署 Nginx Ingress 后，由于 Nginx Ingress 中配置的网络类型为 hostNetwork，因此用户可通过部署了 Nginx Ingress 的节点的 IP（192.168.8.160）和外部端口（9769）访问NebulaGraph服务。

4. 执行以下命令部署连接NebulaGraph服务的 Console 并通过宿主机 IP（本示例为 192.168.8.160）和上述配置的外部端口访问NebulaGraph服务。

```
kubectl run -ti --image vesoft/nebula-console:v3.4.0 --restart=Never -- <nebula_console_name> -addr <host_ip> -port <external_port> -u <username> -p <password>
```

示例：

```
kubectl run -ti --image vesoft/nebula-console:v3.4.0 --restart=Never -- nebula-console -addr 192.168.8.160 -port 9769 -u root -p vesoft
```

- `--image`：为连接NebulaGraph的工具 NebulaGraph Console 的镜像。
- `<nebula-console>`：自定义的 Pod 名称。本示例为 `nebula-console`。
- `-addr`：部署 Nginx Ingress 的节点 IP，本示例为 192.168.8.160。
- `-port`：外网访问使用的端口。本示例设置为 9769。
- `-u`：NebulaGraph账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 `root`）。
- `-p`：用户名对应的密码。未启用身份认证时，密码可以填写任意字符。

如果返回以下内容，说明成功连接数据库：

```
If you don't see a command prompt, try pressing enter.  
(root@nebula) [(none)]>
```

最后更新: May 4, 2023

17.6 配置 NebulaGraph

17.6.1 自定义NebulaGraph集群的配置参数

NebulaGraph集群中 Meta、Storage、Graph 服务都有各自的配置，其在用户创建的NebulaGraph集群实例的 YAML 文件中被定义为 config。 config 中的设置会被映射并加载到对应服务的 ConfigMap 中。各个服务在启动时会挂载 ConfigMap 中的配置到 /usr/local/nebula/etc/ 目录下。

Note

暂不支持通过 Helm 自定义NebulaGraph集群的配置参数。

config 结构如下：

```
Config map[string]string `json:"config,omitempty"`
```

前提条件

已使用 K8s 创建一个集群。具体步骤，参见[使用 Kubectl 创建NebulaGraph集群](#)。

操作步骤

以下示例使用名为 nebula 的集群、名为 nebula_cluster.yaml 的 YAML 配置文件，说明如何为集群的 Graph 服务配置 config：

1. 执行以下命令进入 nebula 集群的编辑页面。

```
kubectl edit nebulaclusters.apps.nebula-graph.io nebula
```

2. 在 YAML 文件的 spec.graphd.config 配置项中，添加 enable_authorize 和 auth_type。

```
apiVersion: apps.nebula-graph.io/v1alpha1
kind: NebulaCluster
metadata:
  name: nebula
  namespace: default
spec:
  graphd:
    resources:
      requests:
        cpu: "500m"
        memory: "500Mi"
      limits:
        cpu: "1"
        memory: "1Gi"
    replicas: 1
    image: vesoft/nebula-graphd
    version: v3.4.1
    storageClaim:
      resources:
        requests:
          storage: 2Gi
        storageClassName: fast-disks
    config: //为 Graph 服务自定义参数。
      "enable_authorize": "true"
      "auth_type": "password"
...
```

如果需要为 Meta 服务和 Storage 服务配置 config，则在 spec.metad.config 和 spec.storaged.config 中添加对应的配置项。

3. 执行 kubectl apply -f nebula_cluster.yaml 使上述更新生效。

在自定义参数 enable_authorize 和 auth_type 后，Graph 服务对应的 ConfigMap（nebula-graphd）中的配置将被覆盖。

在线修改集群配置

在线修改集群配置主要通过调用 HTTP 接口对集群的配置参数进行修改，而无需重启集群 Pod。

需要注意的是，只有当 config 中的配置项均为可在运行时动态修改的参数时，才能触发在线修改集群配置操作。如果 config 中的配置项包含不可动态修改的参数，那么集群配置将通过重启 Pod 的方式进行更新。

关于各个服务可动态修改的参数，分别参见 [Meta 服务配置参数](#)、[Storage 服务配置参数](#)、[Graph 服务配置参数](#)的配置参数表格中是否支持运行时动态修改一列。

更多信息

有关 Meta、Storage、Graph 服务的配置参数的详细介绍，参见 [Meta 服务配置参数](#)、[Storage 服务配置参数](#)、[Graph 服务配置参数](#)。

最后更新: May 4, 2023

17.6.2 回收 PV

NebulaGraph Operator 使用持久化卷 PV (Persistent Volume) 和持久化卷声明 PVC (Persistent Volume Claim) 来存储持久化数据。如果用户不小心删除了一个NebulaGraph集群，默认 PV 和 PVC 对象及其数据仍可保留，以确保数据安全。

用户也可以在集群实例的配置文件中通过设置参数 `spec.enablePVReclaim` 为 `true` 来定义在删除集群后自动删除 PVC 以释放数据。至于在删除 PVC 后是否删除 PV，用户需要自定义 PV 的回收策略。参见 [StorageClass 中设置 reclaimPolicy](#) 和 [PV Reclaiming](#) 了解 PV 回收策略。

前提条件

已使用 K8s 创建一个集群。具体步骤，参见[使用 Kubectl 创建NebulaGraph集群](#)。

操作步骤

以下示例使用名为 nebula 的集群、名为 nebula_cluster.yaml 的 YAML 配置文件，说明如何设置 enablePVReclaim：

1. 执行以下命令进入 nebula 集群的编辑页面。

```
kubectl edit nebulaclusters.apps.nebula-graph.io nebula
```

2. 在 YAML 文件的 spec 配置项中，添加 enablePVReclaim 并设置其值为 true。

```
apiVersion: apps.nebula-graph.io/v1alpha1
kind: NebulaCluster
metadata:
  name: nebula
spec:
  enablePVReclaim: true //设置其值为 true。
  graphd:
    image: vesoft/nebula-graphd
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
        storageClassName: fast-disks
    replicas: 1
  resources:
    limits:
      cpu: "1"
      memory: 1Gi
    requests:
      cpu: 500m
      memory: 500Mi
    version: v3.4.1
  imagePullPolicy: IfNotPresent
  metadata:
    dataVolumeClaim:
      resources:
        requests:
          storage: 2Gi
        storageClassName: fast-disks
    image: vesoft/nebula-metad
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
        storageClassName: fast-disks
    replicas: 1
  resources:
    limits:
      cpu: "1"
      memory: 1Gi
    requests:
      cpu: 500m
      memory: 500Mi
    version: v3.4.1
  nodeSelector:
    nebula: cloud
  reference:
    name: statefulsets.apps
    version: v1
  schedulerName: default-scheduler
  storaged:
    dataVolumeClaims:
      - resources:
          requests:
            storage: 2Gi
          storageClassName: fast-disks
      - resources:
          requests:
            storage: 2Gi
          storageClassName: fast-disks
    image: vesoft/nebula-storaged
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
        storageClassName: fast-disks
    replicas: 3
  resources:
    limits:
      cpu: "1"
      memory: 1Gi
    requests:
      cpu: 500m
      memory: 500Mi
    version: v3.4.1
...

```

3. 执行 kubectl apply -f nebula_cluster.yaml 使上述更新生效。

当集群删除后，系统将会自动删除 PVC 以回收存储资源。是否删除 PV，取决于 PV 的回收策略。

最后更新: May 4, 2023

17.6.3 管理集群日志

NebulaGraph 集群各服务（graphd、metad、storaged）在运行期间会生成运行日志，日志默认存放在各个服务容器的 /usr/local/nebula/logs 目录下。

查看运行日志

如果您需要查看NebulaGraph集群的运行日志，可以通过 kubectl logs 命令查看。

例如，查看 Storage 服务的运行日志：

```
// 查看 Storage 服务 Pod 的名称 (nebula-storaged-0)。
$ kubectl get pods -l app.kubernetes.io/component=storaged
NAME          READY   STATUS    RESTARTS   AGE
nebula-storaged-0   1/1     Running   0          45h
...
// 进入 Storage 服务所在容器 storaged。
$ kubectl exec -it nebula-storaged-0 -c storaged -- /bin/bash

// 查看 Storage 服务的运行日志。
$ cd /usr/local/nebula/logs
```

清理日志

集群服务在运行期间生成的运行日志会占用磁盘空间，为避免占用过多磁盘空间，Operator 使用 sidecar 容器定期清理和归档日志。

为了方便日志的采集和管理，每个NebulaGraph服务都会部署一个 sidecar 容器，负责收集该服务容器产生的日志，并将其发送到指定的日志磁盘中。sidecar 容器使用 logrotate 工具自动清理和归档日志。

在集群实例的 YAML 配置文件中，可以通过 spec.logRotate 字段配置日志轮转以自动对日志进行清理和归档。默认情况下，日志轮转功能是关闭的。开启日志轮转功能示例如下：

```
...
spec:
  graphd:
    config:
      # 是否在日志文件名中包含时间戳，true 表示包含，false 表示不包含。
      "timestamp_in_logfile_name": "false"
  metad:
    config:
      "timestamp_in_logfile_name": "false"
  storaged:
    config:
      "timestamp_in_logfile_name": "false"
  logRotate: # 日志轮转配置
    # 日志文件在被删除前会被轮转的次数。默认值为 5，0 表示删除前不会被轮转。
    rotate: 5
    # 仅当日志文件增长超过定义的字节大小时才会轮转日志文件。默认值为 200M。
    size: "200M"
```

收集日志

如果不想挂载额外的日志磁盘备份日志文件，或者想通过诸如 fluent-bit 之类的服务收集日志并将其发送到日志中心，可以配置日志至标准错误输出。Operator 使用 glog 工具将日志记录到标准错误输出。



目前 Operator 仅收集标准错误日志。

在集群实例的 YAML 配置文件中，可以在各个服务下的 config 和 env 字段中配置日志记录到标准错误输出。

```
...
spec:
  graphd:
    config:
      # 是否将标准错误重定向到单独的输出文件。默认值为 false，表示不重定向。
      redirect_stdout: "false"
      # 日志内容的严重程度级别：INFO、WARNING、ERROR 和 FATAL。取值分别为 0、1、2 和 3。
```

```
    stderrthreshold: "0"
env:
- name: GLOG_logtostderr # 日志写入标准错误而不是文件。
  value: "1" # 1 表示写入标准错误，0 表示写入文件中。
image: vesoft/nebula-graphd
replicas: 1
resources:
  requests:
    cpu: 500m
    memory: 500Mi
service:
  externalTrafficPolicy: Local
  type: NodePort
  version: v3.4.1
metad:
  config:
    redirect_stdout: "false"
    stderrthreshold: "0"
  dataVolumeClaim:
    resources:
      requests:
        storage: 1Gi
    storageClassName: ebs-sc
env:
- name: GLOG_logtostderr
  value: "1"
image: vesoft/nebula-metad
...
...
```

最后更新: May 4, 2023

17.7 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 16, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

17.8 NebulaGraph集群的滚动更新策略

NebulaGraph集群使用分布式架构将数据分成多个逻辑分片，这些分片通常均分在不同的节点上。分布式系统中，同一份数据通常会有多个副本。为了保证多个副本数据的一致性，NebulaGraph集群使用 Raft 协议实现了多分片副本同步。Raft 协议中，每个分片都会选举出一个 Leader 副本，Leader 副本负责处理写请求，Follower 副本负责处理读请求。

通过 Operator 创建的NebulaGraph集群在滚动更新过程中，一个存储节点会暂时停止提供服务以进行更新。关于滚动更新的概述，参见[执行滚动更新](#)。如果 Leader 副本所在的节点停止提供服务，会导致该分片的读写不可用。为了避免这种情况，Operator 会在NebulaGraph集群滚动更新过程中，默认将 Leader 副本迁移到其他未受影响节点上。这样，当一个存储节点处于更新状态时，其他节点上的 Leader 副本能够继续处理客户端请求，以保证集群的读写可用性。

一个存储节点上的所有 Leader 副本全部迁移到其他节点的这个过程可能会持续较长时间。为了更好地控制滚动更新的时间，Operator 提供了一个名为 `enableForceUpdate` 参数。当确定没有外部访问流量时，可将该参数设置为 `true`，这样，Leader 副本将不会被迁移到其他节点上，从而加快滚动更新的速度。

17.8.1 滚动更新触发条件

Operator 会在以下情况下触发NebulaGraph集群的滚动更新：

- NebulaGraph 集群的版本发生变化。
- NebulaGraph 集群的配置发生变化。

17.8.2 配置滚动更新策略

在创建集群实例的 YAML 文件中，添加 `spec.storaged.enableForceUpdate` 参数，设置为 `true` 或 `false`，以控制滚动更新的速度。

当 `enableForceUpdate` 为 `true` 时，表示不迁移分片 Leader 副本，从而加快滚动更新的速度；反之，表示迁移分片 Leader 副本，以保证集群的读写可用性。默认值为 `false`。



设置 `enableForceUpdate` 为 `true` 时，确保没有流量进入集群进行读写操作。因为该设置会强制重建集群 Pod，此过程会发生数据丢失或客户端请求失败的情况。

配置示例：

```
...  
spec:  
...  
  storaged:  
    enableForceUpdate: true // 设置为 true 时，加快滚动更新的速度。  
...  
...
```

最后更新: May 4, 2023

17.9 故障自愈

NebulaGraph Operator 调用NebulaGraph集群提供的接口，动态地感知服务是否正常运行。当NebulaGraph集群中某一组件停止运行时，NebulaGraph Operator 会自动地进行容错处理。本文通过删除NebulaGraph集群中 1 个 Storage 服务 Pod，模拟集群故障为例，说明 Nebula Operator 如何进行故障自愈。

17.9.1 前提条件

安装 NebulaGraph Operator

17.9.2 操作步骤

1. 创建NebulaGraph集群。具体步骤参考[使用 Kubectl 部署NebulaGraph集群](#)或者[使用 Helm 部署NebulaGraph集群](#)。

2. 待所有 Pods 都处于 Running 状态时，模拟故障，删除名为 <cluster_name>-storaged-2 Pod。

```
kubectl delete pod <cluster-name>-storaged-2 --now
```

<cluster_name> 为NebulaGraph集群的名称。

3. NebulaGraph Operator 自动创建名为 <cluster-name>-storaged-2 的 Pod，以修复故障。

执行 kubectl get pods 查看 <cluster-name>-storaged-2 Pod 的创建状态。

```
...
nebula-cluster-storaged-1      1/1    Running   0          5d23h
nebula-cluster-storaged-2      0/1    ContainerCreating 0          1s
...
```

```
...
nebula-cluster-storaged-1      1/1    Running   0          5d23h
nebula-cluster-storaged-2      1/1    Running   0          4m2s
...
```

当 <cluster-name>-storaged-2 的状态由 ContainerCreating 变为 Running 时，说明自愈成功。

最后更新: May 4, 2023

17.10 常见问题

17.10.1 NebulaGraph Operator 支持 v1.x 版本的NebulaGraph吗?

不支持，因为 v1.x 版本的NebulaGraph不支持 DNS，而 NebulaGraph Operator 需要使用 DNS。

17.10.2 使用本地存储是否可以保证集群稳定性?

无法保证。使用本地存储意味着 Pod 被绑定到一个特定的节点，NebulaGraph Operator 目前不支持在绑定的节点发生故障时进行故障转移。

17.10.3 扩缩容集群时，如何确保稳定性?

建议提前备份数据，以便故障发生时回滚数据。

17.10.4 Operator 文档中的 replica 和NebulaGraph内核文档中的 replica 是一样的吗?

二者是不同的概念。Operator 文档中的 replica 是 K8s 中的 Pod 副本，而内核文档中的 replica 是NebulaGraph中的分片副本。

最后更新: May 4, 2023

18. 图计算

18.1 算法简介

图计算可以检测图结构，例如图中社区的检测、图的划分等，也可以揭示各个点之间关联关系的内在特征，例如点的中心性、相似性等。本文介绍相关算法和参数。

Note

本文仅介绍NebulaGraph Analytics的参数，NebulaGraph Algorithm 的参数请先参见对应的[算法文件](#)。

Note

执行图计算时不仅需要设置算法的参数，对数据源也有要求。数据源需要包含起点和终点。PageRank、DegreeWithTime、SSSP、APSP、LPA、HANP、Louvain 算法还需要包含权重（weight）。

- 如果数据源来自 HDFS，需要指定 CSV 文件，包含 src 和 dst 列，部分算法还需要包含 weight 列。
- 如果数据源来自NebulaGraph，需要指定边类型，该类型的边提供 src 和 dst 列，部分算法还需要指定边类型的某个属性作为 weight 列。

18.1.1 节点重要度算法

PageRank

PageRank（页面排序）算法根据点之间的关系（边）计算点的相关性和重要性，通常使用在搜索引擎页面排名中。如果一个网页被很多其他网页链接，说明这个网页比较重要（PageRank 值较高）；如果一个 PageRank 值很高的网页链接到其他网页，那么被链接到的网页的 PageRank 值会提高。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
ITERATIONS	10	最大迭代次数。
IS_DIRECTED	true	是否考虑边的方向。如果设置为 false，系统会自动添加反向边。
EPS	0.0001	收敛精度，两轮迭代的结果差值小于这个值，结束迭代。
DAMPING	0.85	阻尼系数，访问页面后的跳转概率。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点 ID。
VALUE	double	点的 PageRank 值。

KCore

KCore 算法用于计算出没有小于 K 度的点组成的子图，通常使用在社区发现、金融风控等场景。其计算结果是判断点重要性最常用的参考值之一，体现了点的传播能力。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
TYPE	vertex	计算类型。取值: <code>vertex</code> 、 <code>subgraph</code> 。 <code>vertex</code> 表示为每个点计算核心度, <code>subgraph</code> 表示计算邻居。
KMIN	1	范围计算时设置 K 的最小值。仅在 <code>TYPE = subgraph</code> 时生效。
KMAX	1000000	范围计算时设置 K 的最大值。仅在 <code>TYPE = subgraph</code> 时生效。

- TYPE=vertex 时的输出参数

参数	类型	说明
VID	创建图空间时 <code>vid_type</code> 决定	点 ID。
VALUE	int	输出点的核心度。

- TYPE=subgraph 时的输出参数

参数	类型	说明
VID	创建图空间时 <code>vid_type</code> 决定	点 ID。
VALUE	与 VID 类型相同	输出点的邻居。

DegreeCentrality (NStepDegree)

DegreeCentrality (度中心性) 算法用于查找图中的流行点。度中心性测量来自点的传入或传出（或两者）关系的数量，具体取决于关系投影的方向。一个点的度越大就意味着这个点的度中心性越高，该点在网络中就越重要。



NebulaGraph Analytics仅粗略估算度中心性。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
STEP	3	计算度数。 -1 表示无穷大。
BITS	6	用于基数估计的 hyperloglog 位宽。
TYPE	both	计算的边的方向。取值: in、out、both。

- TYPE=both 时的输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点 ID。
BOTH_DEGREE	int	输出点的双向度中心性。
OUT_DEGREE	int	输出点的出方向度中心性。
IN_DEGREE	int	输出点的入方向度中心性。

- TYPE=out 时的输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点 ID。
OUT_DEGREE	int	输出点的出方向度中心性。

- TYPE=in 时的输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点 ID。
IN_DEGREE	int	输出点的入方向度中心性。

DegreeWithTime

DegreeWithTime 算法是基于边的时间范围统计邻居，查找出图中的流行点。



仅NebulaGraph Analytics支持该算法。

参数说明如下。

- 传入参数

参数	默认值	说明
TYPE	both	计算的边的方向。取值: <code>in</code> 、 <code>out</code> 、 <code>both</code> 。
BEGIN_TIME	-	起始时间。格式为 <code>yyyy-MM-dd HH:mm:ss.SSS</code> 。
END_TIME	-	结束时间。格式为 <code>yyyy-MM-dd HH:mm:ss.SSS</code> 。

- TYPE=both 时的输出参数

参数	类型	说明
VID	创建图空间时 <code>vid_type</code> 决定	点 ID。
BOTH_DEGREE	int	输出点的双向流行度。
OUT_DEGREE	int	输出点的出方向流行度。
IN_DEGREE	int	输出点的入方向流行度。

- TYPE=out 时的输出参数

参数	类型	说明
VID	创建图空间时 <code>vid_type</code> 决定	点 ID。
OUT_DEGREE	int	输出点的出方向流行度。

- TYPE=in 时的输出参数

参数	类型	说明
VID	创建图空间时 <code>vid_type</code> 决定	点 ID。
IN_DEGREE	int	输出点的入方向流行度。

BetweennessCentrality

BetweennessCentrality（介数中心性）算法是一种检测点对图中信息流的影响量的方法，用于查找从图的一部分到另一部分时作为桥梁的点。每个点都会根据通过该点的最短路径的数量获得一个分数，即介数中心性分数。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
ITERATIONS	10	最大迭代次数。
IS_DIRECTED	true	是否考虑边的方向。如果设置为 false，系统会自动添加反向边。
CHOSEN	-1	选取的点ID，-1表示随机选。
CONSTANT	2	系数。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点 ID。
VALUE	double	点的介数中心性分数。

ClosenessCentrality

ClosenessCentrality（紧密中心性）算法用于计算一个点到所有其他可达点的最短距离的平均值的倒数。值越大，点在图中的位置越靠近中心，也可以用来衡量信息从该点传输到其他点的时间长短。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
IS_DIRECTED	true	是否考虑边的方向。如果设置为 false，系统会自动添加反向边。
NUM_SAMPLES	10	采样的点数量。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点 ID。
VALUE	double	点的紧密中心性分数。

18.1.2 路径算法

APSP

APSP（全图最短路径）算法用于寻找图中两点之间的所有最短路径。



仅NebulaGraph Analytics支持该算法。

参数说明如下。

- 输出参数

参数	类型	说明
VID1	创建图空间时 vid_type 决定	起点的 ID。
VID2	创建图空间时 vid_type 决定	终点的 ID。
DISTANCE	double	输出 VID1 到 VID2 的距离。

SSSP

SSSP（单源最短路径）算法用于计算给定的一个点（起始点）出发到其余各点的最短路径长度。通常使用在网络路由、路径设计等场景。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
ROOT	-	起始点的 VID。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点 ID。
DISTANCE	double	输出 ROOT 到 VID 的距离。

BFS

BFS（广度优先遍历）算法是一种基础的图遍历算法，它给定一个起始点，以递增的跳数访问其他点，即先遍历点的所有相邻点，再往相邻点的相邻点延伸。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
IS_DIRECTED	true	是否考虑边的方向。如果设置为 false，系统会自动添加反向边。
ROOT	-	起始点的 VID。

- 输出参数

参数	类型	说明
ROOT	创建图空间时 vid_type 决定	起始点的 ID。
VISITED	int	输出 ROOT 访问过的点数量。

ShortestPath

ShortestPath（最短路径）算法用于寻找图中起点和终点任意两两之间的最短路径，适用于路径设计、网络规划等场景。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
src	"100"	起点。多个值用逗号分隔。
dst	"200"	终点。多个值用逗号分隔。

- 输出参数

参数	类型	说明
VALUE	list	返回最短路径中的点。格式为 src, vid1,vid2...dst 。同时有多个最短路径时，只返回一条。

18.1.3 社区发现算法**LPA**

LPA（标签传播）算法是一种基于图的半监督学习方法，其基本思路是用已标记点的标签信息去预测未标记点的标签信息。利用样本间的关系建图，点包括已标注和未标注数据，其边表示两个点的相似度，点的标签按相似度传递给其他点。标签数据就像是一个源头，可以对无标签数据进行标注，点的相似度越大，标签越容易传播。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
ITERATIONS	10	最大迭代次数。
IS_DIRECTED	true	是否考虑边的方向。如果设置为 false，系统会自动添加反向边。
IS_CALC_MODULARITY	false	是否计算模块度。
IS_OUTPUT_MODULARITY	false	是否计算并输出模块度。设置为 true 时，默认输出到文件的第三列，也可以通过选项 --nebula_output_props 和 --nebula_output_types 输出到NebulaGraph中。使用NebulaGraph Explorer时暂不支持输出到NebulaGraph中。
IS_STAT_COMMUNITY	false	是否统计社区的数量。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点的 ID。
LABEL	与 VID 类型相同	输出标签相同的点的 ID。

HANP

HANP（Hop Attenuation & Node Preference）算法是LPA算法的优化算法，考虑了标签的其他信息，例如度的信息、距离信息等，同时在传播时引入了衰减系数，防止过渡传播。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
ITERATIONS	10	最大迭代次数。
IS_DIRECTED	true	是否考虑边的方向。如果设置为 false，系统会自动添加反向边。
PREFERENCE	1.0	对邻居节点度的偏向性。 $m > 0$ 表示偏向节点度高的邻居， $m < 0$ 表示偏向节点度低的邻居， $m = 0$ 表示不考虑邻居节点度。
HOP_ATT	0.1	衰减因子。取值范围 $0 \sim 1$ 。值越大衰减的越快，可以传递的次数越少。
IS_OUTPUT_MODULARITY	false	是否计算并输出模块度。设置为 true 时，默认输出到文件的第三列，也可以通过选项 --nebula_output_props 和 --nebula_output_types 输出到NebulaGraph中。使用NebulaGraph Explorer时暂不支持输出到NebulaGraph中。
IS_STAT_COMMUNITY	false	是否统计社区的数量。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点的 ID。
LABEL	与 VID 类型相同	输出标签相同的点的 ID。

ConnectedComponent

ConnectedComponent（联通分量）算法用于计算出图中的一个子图，当中所有节点都相互连接。考虑路径方向的为强联通分量（strongly connected component），不考虑路径方向的为弱联通分量（weakly connected component）。

 Note

NebulaGraph Analytics仅支持弱联通分量。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
IS_DIRECTED	true	是否考虑边的方向。如果设置为 false，系统会自动添加反向边。
IS_CALC_MODULARITY	false	是否计算模块度。
IS_OUTPUT_MODULARITY	false	是否计算并输出模块度。设置为 true 时，默认输出到文件的第三列，也可以通过选项 --nebula_output_props 和 --nebula_output_types 输出到NebulaGraph中。使用NebulaGraph Explorer时暂不支持输出到NebulaGraph中。
IS_STAT_COMMUNITY	false	是否统计社区的数量。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点的 ID。
LABEL	与 VID 类型相同	输出标签相同的点的 ID。

Louvain

Louvain 算法是基于模块度的社区发现算法，该算法在效率和效果上都表现较好，并且能够发现层次性的社区结构，其优化目标是最大化整个社区网络的模块度。模块度用于区分社区内和社区间链路密度的差异，是衡量每个点划分社区的好坏。通常情况下，一个优秀的分群方法将会使得社区内部的模块度高于社区与社区之间。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
IS_DIRECTED	true	是否考虑边的方向。如果设置为 false，系统会自动添加反向边。
OUTER_ITERATION	20	第一阶段最大迭代次数。
INNER_ITERATION	10	第二阶段最大迭代次数。
IS_CALC_MODULARITY	false	是否计算模块度。
IS_OUTPUT_MODULARITY	false	是否计算并输出模块度。设置为 true 时，默认输出到文件的第三列，也可以通过选项 --nebula_output_props 和 --nebula_output_types 输出到NebulaGraph中。使用NebulaGraph Explorer时暂不支持输出到NebulaGraph中。
IS_STAT_COMMUNITY	false	是否统计社区的数量。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点的 ID。
LABEL	与 VID 类型相同	输出标签相同的点的 ID。

InfoMap

InfoMap 算法使用双层编码方式将有向图进行社区分类。不同社区内部节点的编码复用，可以大幅缩短描述的信息长度。在实现方式上，该算法包含了 PageRank 算法，用于将随机游走转变为随机冲浪。

Note

仅NebulaGraph Analytics支持该算法。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
pagerank_iter	10	内部 PageRank 算法的最大迭代次数。
pagerank_threshold	0.0001	内部 PageRank 算法的收敛精度。
teleport_prob	0.15	穿越概率。
inner_iter	3	内层迭代次数。
outer_iter	2	外层迭代次数。
comm_info_num	100	输出的社群数量。

- 输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点的 ID。
LABEL	与 VID 类型相同	输出标签相同的点的 ID。

18.1.4 图特征算法**TriangleCount**

TriangleCount（三角计数）算法用于统计图中三角形个数。三角形越多，代表图中节点关联程度越高，组织关系越严密。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
OPT	3	计算类型。取值: 1 (统计整个图)、2 (通过每个点统计)、3 (列出所有三角形)。
REMOVED_DUPLICATION_EDGE	true	是否排除重复边。
REMOVED_SELF_EDGE	true	是否排除自环边。

- OPT=1 时的输出参数

参数	类型	说明
COUNT	int	输出全图的三角形数量。

- OPT=2 时的输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	输出每个点的 ID。
COUNT	int	输出每个点的三角形数量。

- OPT=3 时的输出参数

参数	类型	说明
VID1	与 VID 类型相同	输出构成三角形的点 A 的 ID。
VID2	与 VID 类型相同	输出构成三角形的点 B 的 ID。
VID3	与 VID 类型相同	输出构成三角形的点 C 的 ID。

Node2Vec

Node2Vec 算法在 DeepWalk 的基础上提出了更加合理的图特征学习方法，提出了用于网络中可伸缩特征学习的半监督算法，使用 SGD 优化一个自定义的基于图的目标函数，该方法可以最大化的在 D 维特征空间保留节点的网络领域信息；在随机游走的基础上设计了一种二阶随机游走的过程，相当于对 DeepWalk 算法的一种扩展，它保留了邻居节点的图特征。适用于节点功能相似性比较、节点结构相似性比较、社团聚类等场景。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
is_weighted	false	随机游走是否有偏差。
p	1.0	随机游走的后向偏差。
q	0.5	随机游走的前向偏差。
epoch	1	迭代次数。
step	10	每次迭代的步数。
rate	0.02	随机游走的比例。

- 输出参数

输出多个列，同一列中的点是有关联的。

Tree_stat

Tree_stat 算法用于统计图空间中指定根节点的子图的宽度或深度。



仅NebulaGraph Analytics支持该算法。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
root	100	根节点的 VID。
stat	width,depth	统计宽度或深度。多个值用逗号分隔。

- 输出参数

参数	类型	说明
VALUE	list	返回一行统计数据，格式与参数 stat 一致。

HyperANF

HyperANF 算法用于评估图中任意两点的平均距离。

Note

仅NebulaGraph Analytics支持该算法。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
bits	6	HyperLogLog 计数器的 bit 位长度，取值范围：6~16。

- 输出参数

参数	类型	说明
VALUE	double	平均距离。

18.1.5 聚类算法

ClusteringCoefficient

ClusteringCoefficient（聚集系数）算法用于计算图中节点的聚集程度。在各类反映真实世界的网络结构，特别是社交网络结构中，各个点之间倾向于形成密度相对较高的网络群，也就是说，相对于在两个点之间随机连接而得到的网络，真实世界网络的聚集系数更高。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
TYPE	local	聚集类型。取值： <code>local</code> （为每个点计算聚集系数） <code>global</code> （为全图计算聚集系数）。
REMOVED_DUPLICATION_EDGE	true	是否排除重复边。
REMOVED_SELF_EDGE	true	是否排除自环边。

- TYPE=local 时的输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点的 ID。
VALUE	double	输出每个点的聚集系数。

- TYPE=global 时的输出参数

参数	类型	说明
VID	创建图空间时 vid_type 决定	点的 ID。
VALUE	double	输出全图的聚集系数。只有一行数据。

18.1.6 相似度算法

Jaccard

Jaccard（杰卡德相似度）算法用于计算两个点（或集合）的相似程度，预测他们之间的关系。适用于社交网上的好友推荐、关系预测等场景。

参数说明如下。

- NebulaGraph Analytics

- 传入参数

参数	默认值	说明
IDS1	-	若干个 VID 构成的集合A。多个 VID 之间用英文逗号 (,) 隔开。不可为空。
IDS2	-	若干个 VID 构成的集合B。多个 VID 之间用英文逗号 (,) 隔开。可以为空，为空时表示所有点。
REMOVED_SELF_EDGE	true	是否排除自环边。

- 输出参数

参数	类型	说明
VID1	创建图空间时 vid_type 决定	第一个点的 ID。
VID2	创建图空间时 vid_type 决定	第二个点的 ID。
VALUE	double	VID1 和 VID2 的相似度。

最后更新: May 4, 2023

18.2 NebulaGraph Algorithm

NebulaGraph Algorithm（简称 Algorithm）是一款基于 **GraphX** 的 **Spark** 应用程序，通过提交 **Spark** 任务的形式使用完整的算法工具对 NebulaGraph 数据库中的数据执行图计算，也可以通过编程形式调用 lib 库下的算法针对 **DataFrame** 执行图计算。

18.2.1 版本兼容性

NebulaGraph Algorithm 版本和 NebulaGraph 内核的版本对应关系如下。

NebulaGraph 版本	NebulaGraph Algorithm 版本
nightly	3.0-SNAPSHOT
3.0.0 ~ 3.4.x	3.x.0
2.6.x	2.6.x
2.5.0 ~ 2.5.1	2.5.0
2.0.0 ~ 2.0.1	2.1.0

18.2.2 前提条件

在使用 Algorithm 之前，用户需要确认以下信息：

- NebulaGraph 服务已经部署并启动。详细信息，参考 [NebulaGraph 安装部署](#)。
- Spark 版本为 2.4.x。
- Scala 版本为 2.11。
- （可选）如果用户需要在 Github 中克隆最新的 Algorithm，并自行编译打包，可以选择安装 [Maven](#)。

18.2.3 使用限制

- 对于非整数的 String 类型数据，推荐使用调用算法接口的方式，可以使用 SparkSQL 的 dense_rank 函数进行编码，将 String 类型转换为 Long 类型。
- 图计算会输出点的数据集，算法结果会以 DataFrame 形式作为点的属性存储。用户可以根据业务需求，自行对算法结果做进一步操作，例如统计、筛选。

Compatibility

Algorithm v3.1.0 版本之前，直接提交算法包时，点 ID 的数据必须为整数，即点 ID 可以是 INT 类型，或者是 String 类型但数据本身为整数。

18.2.4 支持算法

NebulaGraph Algorithm 支持的图计算算法如下。

算法名	说明	应用场景	属性名称	属性数据类型
PageRank	页面排序	网页排序、重点节点挖掘	pagerank	double/string
Louvain	鲁汶	社团挖掘、层次化聚类	louvain	int/string
KCore	K 核	社区发现、金融风控	kcore	int/string
LabelPropagation	标签传播	资讯传播、广告推荐、社区发现	lpa	int/string
Hanp	标签传播进阶版	社区发现、推荐	hanp	int/string
ConnectedComponent	弱联通分量	社区发现、孤岛发现	cc	int/string
StronglyConnectedComponent	强联通分量	社区发现	scc	int/string
ShortestPath	最短路径	路径规划、网络规划	shortestpath	string
TriangleCount	三角形计数	网络结构分析	trianglecount	int/string
GraphTriangleCount	全图三角形计数	网络结构及紧密程度分析	count	int
BetweennessCentrality	中介中心性	关键节点挖掘、节点影响力计算	betweenness	double/string
ClosenessCentrality	紧密中心性	关键节点挖掘、节点影响力计算	closeness	double/string
DegreeStatic	度统计	图结构分析	degree,inDegree,outDegree	int/string
ClusteringCoefficient	聚集系数	推荐、电信诈骗分析	clustercoefficient	double/string
Jaccard	杰卡德相似度计算	相似度计算、推荐	jaccard	string
BFS	广度优先遍历	层序遍历、最短路径规划	bfs	string
DFS	深度优先遍历	层序遍历、最短路径规划	dfs	string
Node2Vec	-	图分类	node2vec	string

Note

如果需要将算法结果写入到NebulaGraph中，请确保对应图空间中的 Tag 有和上表对应的属性名称和数据类型。

18.2.5 实现方法

NebulaGraph Algorithm 实现图计算的流程如下：

- 利用 NebulaGraph Spark Connector 从NebulaGraph中读取图数据为 DataFrame。
- 将 DataFrame 转换为 GraphX 的图。
- 调用 GraphX 提供的图算法（例如 PageRank）或者自行实现的算法（例如 Louvain 社区发现）。

详细的实现方法可以参见相关 [Scala 文件](#)。

18.2.6 获取 NebulaGraph Algorithm

编译打包

- 克隆仓库 `nebula-algorithm`。

```
$ git clone -b v3.0.0 https://github.com/vesoft-inc/nebula-algorithm.git
```

- 进入目录 `nebula-algorithm`。

```
$ cd nebula-algorithm
```

- 编译打包。

```
$ mvn clean package -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

编译完成后，在目录 `nebula-algorithm/target` 下生成类似文件 `nebula-algorithm-3.x.x.jar`。

Maven 远程仓库下载

[下载地址](#)

18.2.7 使用方法

调用算法接口（推荐）

lib 库中提供了 10 种常用图计算算法，用户可以通过编程调用的形式调用算法。

- 在文件 `pom.xml` 中添加依赖。

```
<dependency>
  <groupId>com.vesoft</groupId>
  <artifactId>nebula-algorithm</artifactId>
  <version>3.0.0</version>
</dependency>
```

- 传入参数调用算法（以 PageRank 为例）。更多调用示例请参见[示例](#)。



执行算法的 DataFrame 默认第一列是起始点，第二列是目的点，第三列是边权重（非NebulaGraph中的 Rank）。

```
val prConfig = new PRConfig(5, 1.0)
val louvainResult = PageRankAlgo.apply(spark, data, prConfig, false)
```

如果用户的节点 ID 是 String 类型，可以参考 PageRank 的[示例](#)。示例中进行了 ID 转换，将 String 类型编码为 Long 类型，并在算法结果中将 Long 类型 ID 解码为原始的 String 类型。

直接提交算法包

1. 设置配置文件。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: LPA
      # Spark 分片数量
      partitionNum:100
    }
    master:local
  }

  data: {
    # 数据源, 可选值为 nebula、csv、json。
    source: nebula
    # 数据落库, 即图计算的结果写入的目标, 可选值为 nebula、csv、json。
    sink: nebula
    # 算法是否需要权重。
    hasWeight: false
  }

  # NebulaGraph相关配置
  nebula: {
    # 数据源。NebulaGraph作为图计算的数据源时, nebula.read 的配置才生效。
    read: {
      # 所有 Meta 服务的 IP 地址和端口, 多个地址用英文逗号 (,) 分隔。格式 :"ip1:port1,ip2:port2"。
      # 使用 docker-compose 部署, 端口需要填写 docker-compose 映射到外部的端口
      # 可以用'docker-compose ps`查看
      metaAddress: "192.168.*.10:9559"
      # NebulaGraph图空间名称
      space: basketballplayer
      # NebulaGraph Edge type, 多个 labels 时, 多个边的数据将合并。
      labels: ["serve"]
      # NebulaGraph 每个 Edge type 的属性名称, 此属性将作为算法的权重列, 请确保和 Edge type 对应。
      weightCols: ["start_year"]
    }

    # 数据落库。图计算结果落库到NebulaGraph时, nebula.write 的配置才生效。
    write: {
      # Graph 服务的 IP 地址和端口, 多个地址用英文逗号 (,) 分隔。格式 :"ip1:port1,ip2:port2"。
      # 使用 docker-compose 部署, 端口需要填写 docker-compose 映射到外部的端口
      # 可以用'docker-compose ps`查看
      graphAddress: "192.168.*.11:9669"
      # 所有 Meta 服务的 IP 地址和端口, 多个地址用英文逗号 (,) 分隔。格式 :"ip1:port1,ip2:port2"。
      # 使用 docker-compose 部署, 端口需要填写 docker-compose 映射到外部的端口
      # 可以用'docker-compose ps`查看
      metaAddress: "192.168.*.12:9559"
      user:root
      pswd:nebula
      # 在提交图计算任务之前需要自行创建图空间及 Tag
      # NebulaGraph图空间名称
      space:nb
      # NebulaGraph Tag 名称, 图计算结果会写入该 Tag。Tag 中的属性名称固定如下:
      # PageRank : pagerank
      # Louvain : louvain
      # ConnectedComponent : cc
      # StronglyConnectedComponent : scc
      # LabelPropagation : lpa
      # ShortestPath : shortestpath
      # DegreeStatic : degree、inDegree、outDegree
      # KCore : kcore
      # TriangleCount : trianglecount
      # BetweennessCentrality : betweenness
      tag:pagerank
    }
  }

  local: {
    # 数据源。图计算的数据源为 csv 文件或 json 文件时, local.read 的配置才生效。
    read: {
      filePath: "hdfs://127.0.0.1:9000/edge/work_for.csv"
      # 如果 CSV 文件没有表头, 使用 [_c0, _c1, _c2, ..., _cn] 表示其表头, 有表头或者是 json 文件时, 直接使用表头名称即可。
      # 起始点 ID 列的表头。
      srcId:"_c0"
      # 目的点 ID 列的表头。
      dstId:"_c1"
      # 权重列的表头
      weight: "_c2"
      # csv 文件是否有表头
      header: false
      # csv 文件的分隔符
      delimiter:","
    }

    # 数据落库。图计算结果落库到 csv 文件或 text 文件时, local.write 的配置才生效。
    write: {
      resultPath:/tmp/
    }
  }
}
```

```

algorithm: {
  # 需要执行的算法, 可选值为:
  # pagerank、louvain、connectedcomponent、labelpropagation、shortestpaths、
  # degreestatic、kcore、stronglyconnectedcomponent、trianglecount、
  # betweenness、graphtriangleCount。
  executeAlgo: pagerank

  # PageRank 参数
  pagerank: {
    maxIter: 10
    resetProb: 0.15
  }

  # Louvain 参数
  louvain: {
    maxIter: 20
    internalIter: 10
    tol: 0.5
  }

  # ...
}
}

```

Note

当配置为 `sink: nebula` 的时候, 意味着算法运算结果将被写回NebulaGraph集群, 这对写回到的 TAG 中的属性名有隐含的约定。详情参考本文支持算法部分。

2. 提交图计算任务。

```
`${SPARK_HOME}/bin/spark-submit --master <mode> --class com.vesoft.nebula.algorithm.Main <nebula-algorithm-3.0.0.jar_path> -p <application.conf_path>
```

示例:

```
`${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.algorithm.Main /root/nebula-algorithm/target/nebula-algorithm-3.0-SNAPSHOT.jar -p /root/nebula-algorithm/src/main/resources/application.conf
```

18.2.8 视频

- 图计算工具——NebulaGraph Algorithm 介绍 (2 分 36 秒)

最后更新: May 4, 2023

18.3 NebulaGraph Analytics License

License 是为用户提供的软件授权证书。用户在部署NebulaGraph Analytics时，需要设置 License 才能正常启动NebulaGraph Analytics。本文介绍NebulaGraph Analytics的 License 相关信息。

18.3.1 注意事项

- 没有设置 License 时，NebulaGraph Analytics无法启动。
- 请勿修改 License，否则会导致 License 失效。
- License 快过期时，请[联系我们](#)申请续期。
- License 的过期缓冲为 14 天。
- 过期前 30 天和过期当天，服务启动时会打印日志进行提醒。
- 过期后仍可继续使用 14 天。
- 过期 14 天后，服务无法启动，并会打印日志进行提醒。

18.3.2 获取方式

[联系我们](#)申请NebulaGraph Analytics License。



填写 [NebulaGraph Analytics试用申请](#)可获取 30 天免费试用的 License。

18.3.3 NebulaGraph Analytics License 说明

NebulaGraph Analytics License 是一个名为 `nebula.license` 的文件，内容示例如下：

```
-----License Content Start-----
{
  "vendor": "vesoft",
  "organization": "vesoft",
  "issuedate": "2022-11-01T16:00:00.000Z",
  "expirationDate": "2023-11-01T15:59:59.000Z",
  "product": "nebula_graph_analytics",
  "version": "3.0.0",
  "licenseType": "enterprise",
  "gracePeriod": 14,
  "analytics": {
    "nodes": 3,
    "vcpu": 3
  }
}
-----License Content End-----

-----License Key Start-----
Rrjip5c+xxxxxxxxxxxxk5Yg==
-----License Key End-----
```

License 文件包含生效时间、过期时间等信息。说明如下。

参数	说明
<code>vendor</code>	发放渠道。
<code>organization</code>	用户名称。
<code>issuedDate</code>	License 生效时间。
<code>expirationDate</code>	License 过期时间。
<code>product</code>	产品类型。NebulaGraph Analytics的产品类型为 <code>nebula_graph_analytics</code> 。
<code>version</code>	版本支持的信息。
<code>licenseType</code>	License 类型（预留参数）。包括 <code>enterprise</code> 、 <code>saml_bussiness</code> 、 <code>pro</code> 、 <code>individual</code> 。
<code>gracePeriod</code>	证书过期后可继续使用服务的缓冲时间（单位天），超过缓冲期后停止服务。试用版的 License 过期后无缓冲期，默认值为 0。
<code>nodes</code>	集群中NebulaGraph Analytics服务的数量限制。
<code>vcpu</code>	集群中NebulaGraph Analytics服务的总线程数量限制。
<code>clusterCode</code>	用户的硬件信息，也是集群的唯一标识码。试用版的 License 中无此参数。

18.3.4 使用方式

关于如何使用NebulaGraph Analytics License，参见[NebulaGraph Analytics](#)。

18.3.5 续期操作

按照以下操作续期NebulaGraph Analytics License。

1. 联系我们申请新的NebulaGraph Analytics License 文件 `nebula.license`。
2. 在NebulaGraph Analytics的安装目录下（例如，`/usr/local/nebula-analytics/scripts/`），使用新的 License 文件替换旧的 License 文件。

Note

NebulaGraph Analytics License 过期后，用户无法正常使用NebulaGraph Analytics。为了保证业务正常运行，请及时更新 License。

最后更新: May 4, 2023

19. NebulaGraph Spark Connector

NebulaGraph Spark Connector 是一个 Spark 连接器，提供通过 Spark 标准形式读写NebulaGraph数据的能力。NebulaGraph Spark Connector 由 Reader 和 Writer 两部分组成。

- Reader

提供一个 Spark SQL 接口，用户可以使用该接口编程读取NebulaGraph图数据，单次读取一个点或 Edge type 的数据，并将读取的结果组装成 Spark 的 DataFrame。

- Writer

提供一个 Spark SQL 接口，用户可以使用该接口编程将 DataFrame 格式的数据逐条或批量写入NebulaGraph。

更多使用说明请参见 [NebulaGraph Spark Connector](#)。

19.1 版本兼容性

NebulaGraph Spark Connector、NebulaGraph内核版本和 Spark 版本对应关系如下。

Spark Connector 版本	NebulaGraph 版本	Spark 版本
nebula-spark-connector_3.0-3.0-SNAPSHOT.jar	nightly	3.x
nebula-spark-connector_2.2-3.0-SNAPSHOT.jar	nightly	2.2.x
nebula-spark-connector-3.0-SNAPSHOT.jar	nightly	2.4.x
nebula-spark-connector_2.2-3.4.0.jar	3.x	2.2.x
nebula-spark-connector-3.4.0.jar	3.x	2.4.x
nebula-spark-connector_2.2-3.3.0.jar	3.x	2.2.x
nebula-spark-connector-3.3.0.jar	3.x	2.4.x
nebula-spark-connector-3.0.0.jar	3.x	2.4.x
nebula-spark-connector-2.6.1.jar	2.6.0, 2.6.1	2.4.x
nebula-spark-connector-2.6.0.jar	2.6.0, 2.6.1	2.4.x
nebula-spark-connector-2.5.1.jar	2.5.0, 2.5.1	2.4.x
nebula-spark-connector-2.5.0.jar	2.5.0, 2.5.1	2.4.x
nebula-spark-connector-2.1.0.jar	2.0.0, 2.0.1	2.4.x
nebula-spark-connector-2.0.1.jar	2.0.0, 2.0.1	2.4.x
nebula-spark-connector-2.0.0.jar	2.0.0, 2.0.1	2.4.x

19.2 适用场景

NebulaGraph Spark Connector 适用于以下场景：

- 在不同的NebulaGraph集群之间迁移数据。
- 在同一个NebulaGraph集群内不同图空间之间迁移数据。
- NebulaGraph与其他数据源之间迁移数据。
- 结合 [NebulaGraph Algorithm](#) 进行图计算。

19.3 特性

NebulaGraph Spark Connector 3.4.0版本特性如下：

- 提供多种连接配置项，如超时时间、连接重试次数、执行重试次数等。
- 提供多种数据配置项，如写入数据时设置对应列为点 ID、起始点 ID、目的点 ID 或属性。
- Reader 支持无属性读取和全属性读取。
- Reader 支持将NebulaGraph数据读取成 Graphx 的 VertexRDD 和 EdgeRDD，支持非 Long 型点 ID。
- 统一了 SparkSQL 的扩展数据源，统一采用 DataSourceV2 进行NebulaGraph数据扩展。
- 支持 insert、update 和 delete 三种写入模式。insert 模式会插入（覆盖）数据，update 模式仅会更新已存在的数据，delete 模式只删除数据。
- 支持与NebulaGraph之间的 SSL 加密连接。

19.4 更新说明

[Release notes](#)

19.5 获取 NebulaGraph Spark Connector

19.5.1 编译打包

1. 克隆仓库 `nebula-spark-connector`。

```
$ git clone -b release-3.4 https://github.com/vesoft-inc/nebula-spark-connector.git
```

2. 进入目录 `nebula-spark-connector`。

3. 编译打包。不同版本的 Spark 命令略有不同。

Note

需已安装对应版本 Spark。

- Spark 2.4

```
$ mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true -pl nebula-spark-connector -am -Pscala-2.11 -Pspark-2.4
```

- Spark 2.2

```
$ mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true -pl nebula-spark-connector_2.2 -am -Pscala-2.11 -Pspark-2.2
```

- Spark 3.x

```
$ mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true -pl nebula-spark-connector_3.0 -am -Pscala-2.12 -Pspark-3.0
```

编译完成后，在目录的文件夹 target 下生成类似文件 `nebula-spark-connector-3.4.0-SHAPSHOT.jar`。

19.5.2 Maven 远程仓库下载

[下载地址](#)

19.6 使用方法

使用 NebulaGraph Spark Connector 读写NebulaGraph时，只需要编写以下代码即可实现。

```
# 从NebulaGraph读取点边数据。
spark.read.nebula().loadVerticesToDF()
spark.read.nebula().loadEdgesToDF()

# 将 dataframe 数据作为点和边写入NebulaGraph中。
dataframe.write.nebula().writeVertices()
dataframe.write.nebula().writeEdges()
```

`nebula()` 接收两个配置参数，包括连接配置和读写配置。

19.6.1 从NebulaGraph读取数据

```
val config = NebulaConnectionConfig
  .builder()
  .withMetaAddress("127.0.0.1:9559")
  .withConectionRetry(2)
  .withExecuteRetry(2)
  .withTimeout(6000)
  .build()

val nebulaReadVertexConfig: ReadNebulaConfig = ReadNebulaConfig
  .builder()
  .withSpace("test")
  .withLabel("person")
  .withNoColumn(false)
  .withReturnCols(List("birthday"))
  .withLimit(10)
  .withPartitionNum(10)
  .build()

val vertex = spark.read.nebula(config, nebulaReadVertexConfig).loadVerticesToDF()

val nebulaReadEdgeConfig: ReadNebulaConfig = ReadNebulaConfig
  .builder()
  .withSpace("test")
  .withLabel("knows")
  .withNoColumn(false)
  .withReturnCols(List("degree"))
  .withLimit(10)
  .withPartitionNum(10)
```

```
.build()
val edge = spark.read.nebula(config, nebulaReadEdgeConfig).LoadEdgesToDF()
```

- NebulaConnectionConfig 是连接NebulaGraph的配置，说明如下。

参数	是否必须	说明
withMetaAddress	是	所有 Meta 服务的地址，多个地址用英文逗号（,）隔开，格式为 ip1:port1,ip2:port2,...。读取数据不需要配置 withGraphAddress。
withConnectionRetry	否	NebulaGraph Java Client 连接NebulaGraph的重试次数。默认值为 1。
withExecuteRetry	否	NebulaGraph Java Client 执行查询语句的重试次数。默认值为 1。
withTimeout	否	NebulaGraph Java Client 请求响应的超时时间。默认值为 6000，单位：毫秒（ms）。

- ReadNebulaConfig 是读取NebulaGraph数据的配置，说明如下。

参数	是否必须	说明
withSpace	是	NebulaGraph图空间名称。
withLabel	是	NebulaGraph图空间内的 Tag 或 Edge type 名称。
withNoColumn	否	是否不读取属性。默认值为 false，表示读取属性。取值为 true 时，表示不读取属性，此时 withReturnCols 配置无效。
withReturnCols	否	配置要读取的点或边的属性集。格式为 List(property1,property2,...)，默认值为 List()，表示读取全部属性。
withLimit	否	配置 NebulaGraph Java Storage Client 一次从服务端读取的数据行数。默认值为 1000。
withPartitionNum	否	配置读取NebulaGraph数据时 Spark 的分区数。默认值为 100。该值的配置最好不超过图空间的分片数量（partition_num）。

19.6.2 向NebulaGraph写入数据



DataFrame 中的列会自动作为属性写入NebulaGraph。

```
val config = NebulaConnectionConfig
.builder()
.withMetaAddress("127.0.0.1:9559")
.withGraphAddress("127.0.0.1:9669")
.withConenctionRetry(2)
.build()

val nebulaWriteVertexConfig: WriteNebulaVertexConfig = WriteNebulaVertexConfig
.builder()
.withSpace("test")
.withTag("person")
.withVidField("id")
.withVidPolicy("hash")
.withVidAsProp(true)
.withUser("root")
.withPasswd("nebula")
.withBatch(512)
.build()
df.write.nebula(config, nebulaWriteVertexConfig).writeVertices()

val nebulaWriteEdgeConfig: WriteNebulaEdgeConfig = WriteNebulaEdgeConfig
.builder()
.withSpace("test")
.withEdge("friend")
.withSrcIdField("src")
.withSrcPolicy(null)
.withDstIdField("dst")
.withDstPolicy(null)
.withRankField("degree")
.withSrcAsProperty(true)
.withDstAsProperty(true)
.withRankAsProperty(true)
.withUser("root")
```

```
.withPassword("nebula")
.withBatch(512)
.build()
df.write.nebula(config, nebulaWriteEdgeConfig).writeEdges()
```

默认写入模式为 `insert`，可以通过 `withWriteMode` 配置修改为 `update` 或 `delete`：

```
val config = NebulaConnectionConfig
.builder()
.withMetaAddress("127.0.0.1:9559")
.withGraphAddress("127.0.0.1:9669")
.build()
val nebulaWriteVertexConfig = WriteNebulaVertexConfig
.builder()
.withSpace("test")
.withTag("person")
.withVidField("id")
.withVidAsProp(true)
.withBatch(512)
.withWriteMode(WriteMode.UPDATE)
```

```
.build()
df.write.nebula(config, nebulaWriteVertexConfig).writeVertices()
```

- `NebulaConnectionConfig` 是连接NebulaGraph的配置，说明如下。

参数	是否必须	说明
<code>withMetaAddress</code>	是	所有 Meta 服务的地址，多个地址用英文逗号 (,) 隔开，格式为 ip1:port1,ip2:port2,...。
<code>withGraphAddress</code>	是	Graph 服务的地址，多个地址用英文逗号 (,) 隔开，格式为 ip1:port1,ip2:port2,...。
<code>withConnectionRetry</code>	否	NebulaGraph Java Client 连接NebulaGraph的重试次数。默认值为 1。

- `WriteNebulaVertexConfig` 是写入点的配置，说明如下。

参数	是否必须	说明
<code>withSpace</code>	是	NebulaGraph图空间名称。
<code>withTag</code>	是	写入点时需要关联的 Tag 名称。
<code>withVidField</code>	是	DataFrame 中作为点 ID 的列。
<code>withVidPolicy</code>	否	写入点 ID 时，采用的映射函数，NebulaGraph仅支持 HASH。默认不做映射。
<code>withVidAsProp</code>	否	DataFrame 中作为点 ID 的列是否也作为属性写入。默认值为 false。如果配置为 true，请确保 Tag 中有和 VidField 相同的属性名。
<code>withUser</code>	否	NebulaGraph用户名。若未开启身份验证，无需配置用户名和密码。
<code>withPasswd</code>	否	NebulaGraph用户名对应的密码。
<code>withBatch</code>	是	一次写入的数据行数，默认值为 512。当 <code>withWriteMode</code> 为 update 时，该参数的最大值为 512。
<code>withWriteMode</code>	否	写入模式。可选值为 insert、update 和 delete。默认为 insert。
<code>withDeleteEdge</code>	否	删除点时是否删除该点关联的边。默认为 false。当 <code>withWriteMode</code> 为 delete 时生效。

- `WriteNebulaEdgeConfig` 是写入边的配置，说明如下。

参数	是否必须	说明
<code>withSpace</code>	是	NebulaGraph图空间名称。
<code>withEdge</code>	是	写入边时需要关联的 Edge type 名称。
<code>withSrcIdField</code>	是	DataFrame 中作为起始点的列。
<code>withSrcPolicy</code>	否	写入起始点时，采用的映射函数，NebulaGraph仅支持 HASH。默认不做映射。
<code>withDstIdField</code>	是	DataFrame 中作为目的点的列。
<code>withDstPolicy</code>	否	写入目的点时，采用的映射函数，NebulaGraph仅支持 HASH。默认不做映射。
<code>withRankField</code>	否	DataFrame 中作为 rank 的列。默认不写入 rank。
<code>withSrcAsProperty</code>	否	DataFrame 中作为起始点的列是否也作为属性写入。默认值为 false。如果配置为 true，请确保 Edge type 中有和 <code>SrcIdField</code> 相同的属性名。
<code>withDstAsProperty</code>	否	DataFrame 中作为目的点的列是否也作为属性写入。默认值为 false。如果配置为 true，请确保 Edge type 中有和 <code>DstIdField</code> 相同的属性名。
<code>withRankAsProperty</code>	否	DataFrame 中作为 rank 的列是否也作为属性写入。默认值为 false。如果配置为 true，请确保 Edge type 中有和 <code>RankField</code> 相同的属性名。
<code>withUser</code>	否	NebulaGraph用户名。若未开启 身份验证 ，无需配置用户名和密码。
<code>withPasswd</code>	否	NebulaGraph用户名对应的密码。
<code>withBatch</code>	是	一次写入的数据行数，默认值为 512。当 <code>withWriteMode</code> 为 update 时，该参数的最大值为 512。
<code>withWriteMode</code>	否	写入模式。可选值为 <code>insert</code> 、 <code>update</code> 和 <code>delete</code> 。默认为 <code>insert</code> 。

19.6.3 示例代码

详细的使用方式参见 [示例代码](#)。

最后更新: May 4, 2023

20. NebulaGraph Flink Connector

NebulaGraph Flink Connector 是一款帮助 Flink 用户快速访问NebulaGraph的连接器，支持从NebulaGraph图数据库中读取数据，或者将其他外部数据源读取的数据写入NebulaGraph图数据库。

更多使用说明请参见 [NebulaGraph Flink Connector](#)。

20.1 适用场景

NebulaGraph Flink Connector 适用于以下场景:

- 在不同的NebulaGraph集群之间迁移数据。
- 在同一个NebulaGraph集群内不同图空间之间迁移数据。
- NebulaGraph与其他数据源之间迁移数据。

20.2 更新说明

[Release notes](#)

最后更新: May 4, 2023

21. NebulaGraph Bench

NebulaGraph Bench 是一款利用 LDBC 数据集对NebulaGraph进行性能测试的工具。

21.1 适用场景

- 生成测试数据并导入NebulaGraph。
- 对NebulaGraph集群进行性能测试。

21.2 更新说明

Release

详细使用说明请参见 [NebulaGraph Bench](#)。

最后更新: May 4, 2023

22. 附录

22.1 Macro Syntax Error

Line 183 in Markdown file: **unexpected '**

```
| 数据监控和运维 | [{{dashboard_ent.name}}](https://docs.nebula-graph.com.cn/{{nebula.release}}/nebula-dashboard-ent/1.what-is-dashboard-ent/) 和 [Nebula Dashboard 社区版](https://docs.nebula-graph.com.cn/{{nebula.release}}/nebula-dashboard/1.what-is-dashboard/) | [{{dashboard_ent.name}}](https://www.bilibili.com/video/BV1qQ4y1k7gb?)  
|
```

最后更新: May 4, 2023

22.2 Macro Rendering Error

UndefinedError: 'ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 342, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'ent' is undefined
```

最后更新: May 4, 2023

22.3 Macro Rendering Error

UndefinedError: 'comm' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 34, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'comm' is undefined
```

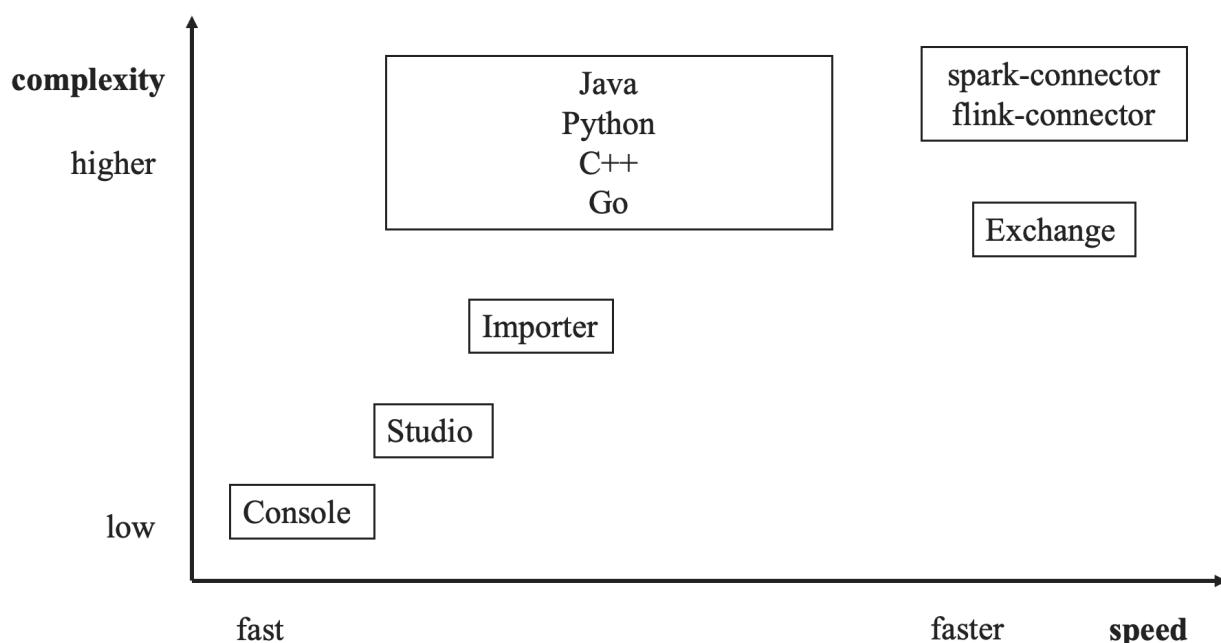
最后更新: May 4, 2023

22.4 导入工具选择

有多种方式可以写入NebulaGraph 3.4.1:

- 使用 **命令行 -f 的方式** 导入: 可以导入少量准备好的 nGQL 文件, 适合少量手工测试数据准备;
- 使用 **studio 导入**: 可以用过浏览器导入本机多个 csv 文件, 格式有限制;
- 使用 **importer 导入**: 导入单机多个 csv 文件, 大小没有限制, 格式灵活; 数据量十亿级以内;
- 使用 **Exchange 导入**: 从 Neo4j, Hive, MySQL 等多种源分布式导入, 需要有 Spark 集群; 数据量十亿级以上
- 使用 **Spark-connector/Flink-connector** 导入: 有相应组件 (Spark/Flink), 撰写少量代码;
- 使用 **C++/GO/Java/Python SDK**: 编写程序的方式导入, 需要有一定编程和调优能力。

下图给出了几种方式的定位:



最后更新: May 4, 2023

22.5 如何贡献代码和文档

22.5.1 开始之前

GitHub 或社区提交问题

欢迎为项目贡献任何代码或文档，但是建议先在 [GitHub](#) 或[社区](#)上提交一个问题，和大家共同讨论。

签署贡献者许可协议（[CLA](#)）

什么是 [CLA](#)？

签署协议链接: [vesoft inc. Contributor License Agreement](#)

单击按钮 **Sign in with GitHub to agree** 签署协议。

如果有任何问题，请发送邮件至 info@vesoft.com。

22.5.2 修改单篇文档

NebulaGraph文档以 Markdown 语言编写。单击文档标题右侧的铅笔图标即可提交修改建议。

该方法仅适用于修改单篇文档。

22.5.3 批量修改或新增文件

该方法适用于贡献代码、批量修改多篇文档或者新增文档。

Step 1: 通过 [GitHub fork](#) 仓库

NebulaGraph项目有很多[仓库](#)，以 [NebulaGraph仓库](#)为例:

1. 访问 github.com/vesoft-inc/nebula。
2. 在右上角单击按钮 **Fork**，然后单击用户名，即可 fork 出NebulaGraph仓库。

Step 2: 将分支克隆到本地

1. 定义本地工作目录。

```
# 定义工作目录。
working_dir=$HOME/Workspace
```

2. 将 user 设置为 GitHub 的用户名。

```
user={GitHub 用户名}
```

3. 克隆代码。

```
mkdir -p $working_dir
cd $working_dir
git clone https://github.com/$user/nebula.git
# 或 : git clone git@github.com:$user/nebula.git

cd $working_dir/nebula
git remote add upstream https://github.com/vesoft-inc/nebula.git
# 或 : git remote add upstream git@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
```

4. (可选) 定义 pre-commit hook。

请将NebulaGraph的 pre-commit hook 连接到 .git 目录。

hook 将检查 commit, 包括格式、构建、文档生成等。

```
cd $working_dir/nebula/.git/hooks
ln -s $working_dir/nebula/.linters/cpp/hooks/pre-commit.sh .
```

pre-commit hook 有时候可能无法正常执行，用户必须手动执行。

```
cd $working_dir/nebula/.git/hooks
chmod +x pre-commit
```

Step 3: 分支

1. 更新本地主分支。

```
cd $working_dir/nebula
git fetch upstream
git checkout master
git rebase upstream/master
```

2. 从主分支创建并切换分支:

```
git checkout -b myfeature
```

Note

由于一个 PR 通常包含多个 commits, 最终合入 upstream/master 分支时, 我们会将这些 commits 挤压 (squash) 成一个 commit 进行合并。因此强烈建议创建一个独立的分支进行更改, 这样在合入时才容易被挤压。合并后, 这个分支可以被丢弃。如果未创建单独的分支, 而是直接将 commits 提交至 origin/master, 在合入时, 可能会出现问题。若未创建单独的分支 (或是 origin/master 合并了其他的分支等), 导致 origin/master 和 upstream/master 不一致时, 用户可以使用 hard reset 强制两者进行一致。例如:

```
git fetch upstream
git checkout master
git reset --hard upstream/master
git push --force origin master
```

Step 4: 开发

- 代码风格

NebulaGraph采用 `cppcheck` 来确保代码符合 Google 的代码风格指南。检查器将在提交代码之前执行。

- 单元测试要求
请为新功能或 Bug 修复添加单元测试。
- 构建代码时开启单元测试
详情请参见[使用源码安装NebulaGraph](#)。

Note

请确保已设置 `-DENABLE_TESTING = ON` 启用构建单元测试。

- 运行所有单元测试

在 `nebula` 根目录执行如下命令:

```
cd nebula/build
ctest -j$(nproc)
```

Step 5: 保持分支同步

```
# 当处于 myfeature 分支时。
git fetch upstream
git rebase upstream/master
```

在其他贡献者将 PR 合并到基础分支之后，用户需要更新 `head` 分支。

Step 6: Commit

提交代码更改:

```
git commit -a
```

用户可以使用命令 `--amend` 重新编辑之前的代码。

Step 7: Push

需要审核或离线备份代码时，可以将本地仓库创建的分支 `push` 到 GitHub 的远程仓库。

```
git push origin myfeature
```

Step 8: 创建 pull request

1. 访问 `fork` 出的仓库 [https://github.com/\\$user/nebula](https://github.com/$user/nebula) (替换此处的用户名 \$user)。
2. 单击 `myfeature` 分支旁的按钮 `Compare & pull request`。

Step 9: 代码审查

`pull request` 创建后，至少需要两人审查。审查人员将进行彻底的代码审查，以确保变更满足存储库的贡献准则和其他质量标准。

22.5.4 添加测试用例

添加测试用例的方法参见 [How to add test cases](#)。

22.5.5 捐赠项目

Step 1: 确认项目捐赠

通过邮件、微信、Slack 等方式联络NebulaGraph官方人员，确认捐赠项目一事。项目将被捐赠至 **NebulaGraph Contrib** 组织下。

- 邮件地址: info@vesoft.com
- 微信: NebulaGraphbot
- Slack: [Join Slack](#)

Step 2: 获取项目接收人信息

由NebulaGraph官方人员给出NebulaGraph Contrib 的项目接收者 ID。

Step 3: 捐赠项目

由您将项目转移至本次捐赠的项目接受人，并由项目接收者将该项目转移至 **NebulaGraph Contrib** 组织下。捐赠后，您将以 **Maintain** 角色继续主导社区项目的发展。

GitHub 上转移仓库的操作，请参见 [Transferring a repository owned by your user account](#)。

最后更新: May 4, 2023

22.6 Macro Rendering Error

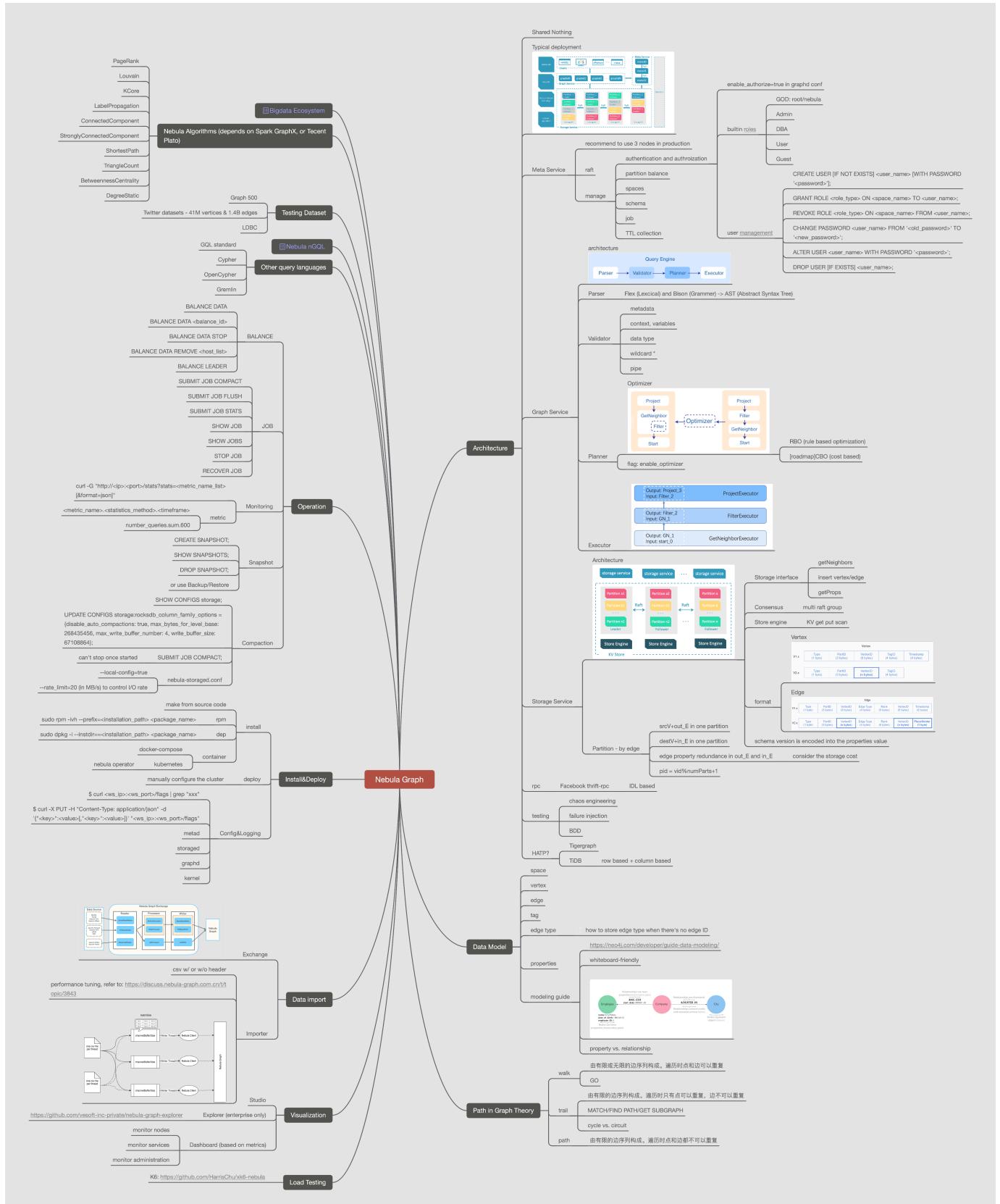
UndefinedError: 'comm' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 480, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1301, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 936, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 3, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/jinja2/environment.py", line 485, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'comm' is undefined
```

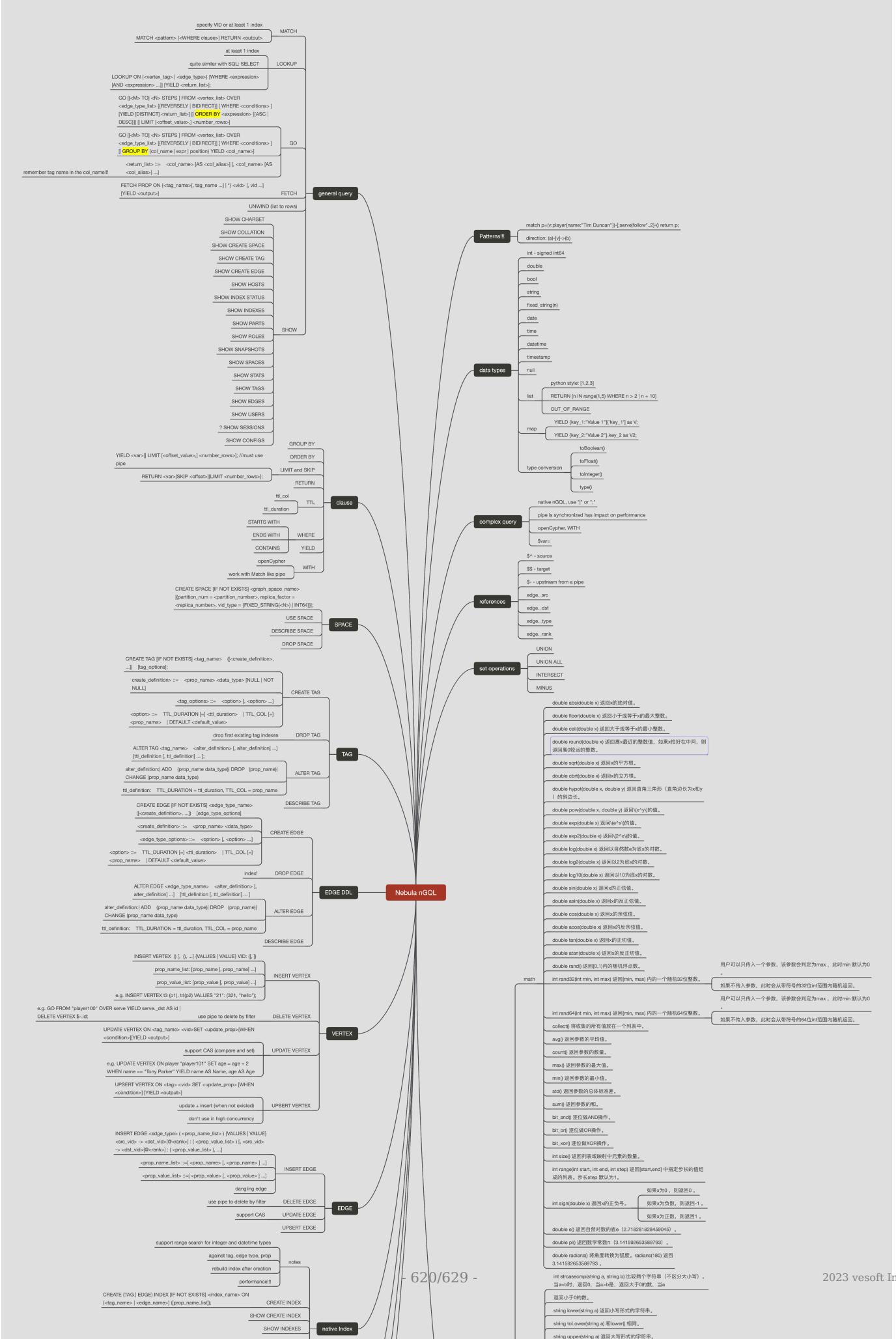
最后更新: May 4, 2023

22.7 思维导图

以下给出NebulaGraph结构框架的思维导图，用户可以[点击](#)并查看大图。



以下给出 nGQL 的思维导图，用户可以[点击](#) 并查看大图。



最后更新: May 4, 2023

22.8 错误码

NebulaGraph运行出现问题时，会返回错误码。本文介绍错误码的详细信息。

 Note

- 如果出现错误但没有返回错误码，或错误码描述不清，请在[论坛](#)或[GitHub](#) 反馈。
- 返回 0 表示执行成功。

错误名称	错误码	说明
E_DISCONNECTED	-1	连接断开
E_FAIL_TO_CONNECT	-2	无法建立连接
E_RPC_FAILURE	-3	RPC 失败
E_LEADER_CHANGED	-4	Raft leader 变更
E_SPACE_NOT_FOUND	-5	图空间不存在
E_TAG_NOT_FOUND	-6	Tag 不存在
E_EDGE_NOT_FOUND	-7	Edge type 不存在
E_INDEX_NOT_FOUND	-8	索引不存在
E_EDGE_PROP_NOT_FOUND	-9	边属性不存在
E_TAG_PROP_NOT_FOUND	-10	Tag 属性不存在
E_ROLE_NOT_FOUND	-11	当前角色不存在
E_CONFIG_NOT_FOUND	-12	当前配置不存在
E_MACHINE_NOT_FOUND	-13	当前主机不存在
E_LISTENER_NOT_FOUND	-15	listener 不存在
E_PART_NOT_FOUND	-16	当前分区不存在
E_KEY_NOT_FOUND	-17	key 不存在
E_USER_NOT_FOUND	-18	用户不存在
E_STATS_NOT_FOUND	-19	统计信息不存在
E_SERVICE_NOT_FOUND	-20	没有找到当前服务
E_DRAINER_NOT_FOUND	-21	drainer 不存在
E_DRAINER_CLIENT_NOT_FOUND	-22	drainer 客户端不存在
E_PART_STOPPED	-23	当前 partition 已经被停止
E_BACKUP_FAILED	-24	备份失败
E_BACKUP_EMPTY_TABLE	-25	备份的表为空
E_BACKUP_TABLE_FAILED	-26	备份表失败
E_PARTIAL_RESULT	-27	multiget 无法获得所有数据
E_REBUILD_INDEX_FAILED	-28	重建索引失败
E_INVALID_PASSWORD	-29	密码无效
E_FAILED_GET_ABS_PATH	-30	无法获得绝对路径
E_BAD_USERNAME_PASSWORD	-1001	身份验证失败
E_SESSION_INVALID	-1002	无效会话
E_SESSION_TIMEOUT	-1003	会话超时
E_SYNTAX_ERROR	-1004	语法错误
E_EXECUTION_ERROR	-1005	执行错误
E_STATEMENT_EMPTY	-1006	语句为空

错误名称	错误码	说明
E_BAD_PERMISSION	-1008	权限不足
E_SEMANTIC_ERROR	-1009	语义错误
E_TOO_MANY_CONNECTIONS	-1010	超出最大连接数
E_PARTIAL_SUCCEEDED	-1011	访问存储失败（仅有部分请求成功）
E_NO_HOSTS	-2001	主机不存在
E_EXISTED	-2002	主机已经存在
E_INVALID_HOST	-2003	无效主机
E_UNSUPPORTED	-2004	当前命令、语句、功能不支持
E_NOT_DROP	-2005	不允许删除
E_CONFIG_IMMUTABLE	-2007	配置项不能改变
E_CONFLICT	-2008	参数与 meta 数据冲突
E_INVALID_PARM	-2009	无效的参数
E_WRONGCLUSTER	-2010	错误的集群
E_ZONE_NOT_ENOUGH	-2011	listener 冲突
E_ZONE_IS_EMPTY	-2012	主机不存在
E_SCHEMA_NAME_EXISTS	-2013	Schema 名字已存在
E RELATED INDEX EXISTS	-2014	与 Tag 或 Edge Type 相关的索引存在，不能被删除
E RELATED SPACE EXISTS	-2015	仍有图空间在主机上，不能被删除
E_STORE_FAILURE	-2021	存储数据失败
E_STORE_SEGMENT_ILLEGAL	-2022	存储段非法
E_BAD_BALANCE_PLAN	-2023	无效的数据均衡计划
E_BALANCED	-2024	集群已经处于数据均衡状态
E_NO_RUNNING_BALANCE_PLAN	-2025	没有正在运行的数据均衡计划
E_NO_VALID_HOST	-2026	缺少有效的主机
E_CORRUPTED_BALANCE_PLAN	-2027	已经损坏的数据均衡计划
E_IMPROPER_ROLE	-2030	回收用户角色失败
E_INVALID_PARTITION_NUM	-2031	无效的分区数量
E_INVALID_REPLICA_FACTOR	-2032	无效的副本因子
E_INVALID_CHARSET	-2033	无效的字符集
E_INVALID_COLLATE	-2034	无效的字符排序规则
E_CHARSET_COLLATE_NOT_MATCH	-2035	字符集和字符排序规则不匹配
E_SNAPSHOT_FAILURE	-2040	生成快照失败
E_BLOCK_WRITE_FAILURE	-2041	写入块数据失败
E_ADD_JOB_FAILURE	-2044	增加新的任务失败
E_STOP_JOB_FAILURE	-2045	停止任务失败

错误名称	错误码	说明
E_SAVE_JOB_FAILURE	-2046	保存任务信息失败
E_BALANCER_FAILURE	-2047	数据均衡失败
E_JOB_NOT_FINISHED	-2048	当前任务还没有完成
E_TASK_REPORT_OUT_DATE	-2049	任务报表失效
E_JOB_NOT_IN_SPACE	-2050	当前任务不在图空间内
E_JOB_NEED_RECOVER	-2051	当前任务需要恢复
E_JOB_ALREADY_FINISH	-2052	任务已经失败或完成
E_JOB_SUBMITTED	-2053	任务默认状态
E_JOB_NOT_STOPPABLE	-2054	给定任务不支持停止
E_JOB_HAS_NO_TARGET_STORAGE	-2055	leader 分布未上报，因此无法将任务发送到存储
E_INVALID_JOB	-2065	无效的任务
E_BACKUP_BUILDING_INDEX	-2066	备份终止（正在创建索引）
E_BACKUP_SPACE_NOT_FOUND	-2067	备份时图空间不存在
E_RESTORE_FAILURE	-2068	备份恢复失败
E_SESSION_NOT_FOUND	-2069	会话不存在
E_LIST_CLUSTER_FAILURE	-2070	获取集群信息失败
E_LIST_CLUSTER_GET_ABS_PATH_FAILURE	-2071	获取集群信息时无法获取绝对路径
E_LIST_CLUSTER_NO_AGENT_FAILURE	-2072	获取集群信息时无法获得 agent
E_QUERY_NOT_FOUND	-2073	query 未找到
E_AGENT_HB_FAILURE	-2074	agent 没有汇报心跳
E_GRAPH_MEMORY_EXCEEDED	-2600	Graph 内存超出
E_CONSENSUS_ERROR	-3001	选举时无法达成共识
E_KEY_HAS_EXISTS	-3002	key 已经存在
E_DATA_TYPE_MISMATCH	-3003	数据类型不匹配
E_INVALID_FIELD_VALUE	-3004	无效的字段值
E_INVALID_OPERATION	-3005	无效的操作
E_NOT_NULLABLE	-3006	当前值不允许为空
E_FIELD_UNSET	-3007	字段非空或者没有默认值时，字段值必须设置
E_OUT_OF_RANGE	-3008	取值超出了当前类型的范围
E_DATA_CONFLICT_ERROR	-3010	数据冲突
E_WRITE_STALLED	-3011	写入被延迟
E_IMPROPER_DATA_TYPE	-3021	不正确的数据类型
E_INVALID_SPACEVIDLEN	-3022	VID 长度无效
E_INVALID_FILTER	-3031	无效的过滤器
E_INVALID_UPDATER	-3032	无效的字段更新

错误名称	错误码	说明
E_INVALID_STORE	-3033	无效的 KV 存储
E_INVALID_PEER	-3034	peer 无效
E_RETRY_EXHAUSTED	-3035	重试次数耗光
E_TRANSFER_LEADER_FAILED	-3036	leader 转换失败
E_INVALID_STAT_TYPE	-3037	无效的统计类型
E_INVALID_VID	-3038	VID 无效
E_LOAD_META_FAILED	-3040	加载元信息失败
E_FAILED_TO_CHECKPOINT	-3041	生成 checkpoint 失败
E_CHECKPOINT_BLOCKED	-3042	生成 checkpoint 被阻塞
E_FILTER_OUT	-3043	数据被过滤
E_INVALID_DATA	-3044	无效的数据
E_MUTATE_EDGE_CONFLICT	-3045	并发写入同一条边发生冲突
E_MUTATE_TAG_CONFLICT	-3046	并发写入同一个点发生冲突
E_OUTDATED_LOCK	-3047	锁已经失效
E_INVALID_TASK_PARA	-3051	无效的任务参数
E_USER_CANCEL	-3052	用户取消了任务
E_TASK_EXECUTION_FAILED	-3053	任务执行失败
E_PLAN_IS_KILLED	-3060	执行计划被清除
E_NO_TERM	-3070	收到请求时心跳流程未完成
E_OUTDATED_TERM	-3071	收到旧 leader 的过时心跳（已选举出新的 leader）
E_WRITE_WRITE_CONFLICT	-3073	并发写入时与后到的请求发生冲突
E_RAFT_UNKNOWN_PART	-3500	未知的分区
E_RAFT_LOG_GAP	-3501	raft 日志落后
E_RAFT_LOG_STALE	-3502	raft 日志过期
E_RAFT_TERM_OUT_OF_DATE	-3503	心跳信息已经过期
E_RAFT_UNKNOWN_APPEND_LOG	-3504	未知的追加日志
E_RAFT_WAITING_SNAPSHOT	-3511	等待快照完成
E_RAFT_SENDING_SNAPSHOT	-3512	发送快照过程出错
E_RAFT_INVALID_PEER	-3513	无效的接收端
E_RAFT_NOT_READY	-3514	Raft 没有启动
E_RAFT_STOPPED	-3515	Raft 已经停止
E_RAFT_BAD_ROLE	-3516	错误的角色
E_RAFT_WAL_FAIL	-3521	写入 WAL 失败
E_RAFT_HOST_STOPPED	-3522	主机已经停止
E_RAFT_TOO_MANY_REQUESTS	-3523	请求数量过多

错误名称	错误码	说明
E_RAFT_PERSIST_SNAPSHOT_FAILED	-3524	持久化快照失败
E_RAFT_RPC_EXCEPTION	-3525	RPC 异常
E_RAFT_NO_WAL_FOUND	-3526	没有发现 WAL 日志
E_RAFT_HOST_PAUSED	-3527	主机暂停
E_RAFT_WRITE_BLOCKED	-3528	写入被堵塞
E_RAFT_BUFFER_OVERFLOW	-3529	缓存溢出
E_RAFT_ATOMIC_OP_FAILED	-3530	原子操作失败
E_LEADERLEASE_FAILED	-3531	leader 租约过期
E_RAFT_CAUGHT_UP	-3532	Raft 已经同步数据
E_STORAGE_MEMORY_EXCEEDED	-3600	Storage 内存超出
E_LOG_GAP	-4001	drainer 日志落后
E_LOG_STALE	-4002	drainer 日志过期
E_INVALID_DRAINER_STORE	-4003	drainer 数据存储无效
E_SPACE_MISMATCH	-4004	图空间不匹配
E_PART_MISMATCH	-4005	分区不匹配
E_DATA_CONFLICT	-4006	数据冲突
E_REQ_CONFLICT	-4007	请求冲突
E_DATA_ILLEGAL	-4008	数据非法
E_CACHE_CONFIG_ERROR	-5001	缓存配置错误
E_NOT_ENOUGH_SPACE	-5002	空间不足
E_CACHE_MISS	-5003	没有命中缓存
E_POOL_NOT_FOUND	-5005	写缓存失败
E_NODE_NUMBER_EXCEED_LIMIT	-7001	机器节点数超出限制
E_PARSING_LICENSE_FAILURE	-7002	解析证书失败
E_UNKNOWN	-8000	未知错误

最后更新: May 4, 2023



<https://docs.nebula-graph.com.cn/master>