NebulaGraph Database 手册

v3.6.0

NebulaGraph

Table of contents

1. 欢迎阅读 NebulaGraph 3.6.0 文档

Note

本文档更新时间2024-3-4，GitHub commit 1a43aed。该版本主色系为"桑色"，色号为 #55295B。

1.1 快速开始

- 快速开始
- 部署要求
- nGQL 命令汇总
- FAQ
- 生态工具
- Academy 课程
- 在线体验

1.2 最新发布

- NebulaGraph 3.6.0
- NebulaGraph Dashboard Community
- NebulaGraph Studio

1.3 其他资料

- 学习路径
- 引用 NebulaGraph
- 论坛
- 主页
- 系列视频
- 英文文档

1.4 图例说明

Note

额外的信息或者操作相关的提醒等。

Caution

可能会产生不良影响，例如导致性能下降或引发已知的小问题。

Warning

可能导致严重后果，例如数据丢失、系统崩溃。

Danger

可能导致极其严重的后果，例如系统损坏、信息泄露。

Compatibility

nGQL 与 openCypher 的兼容性或 nGQL 当前版本与历史版本的兼容性。

Enterpriseonly

描述社区版和企业版的差异。

1.5 修改文档中的错误

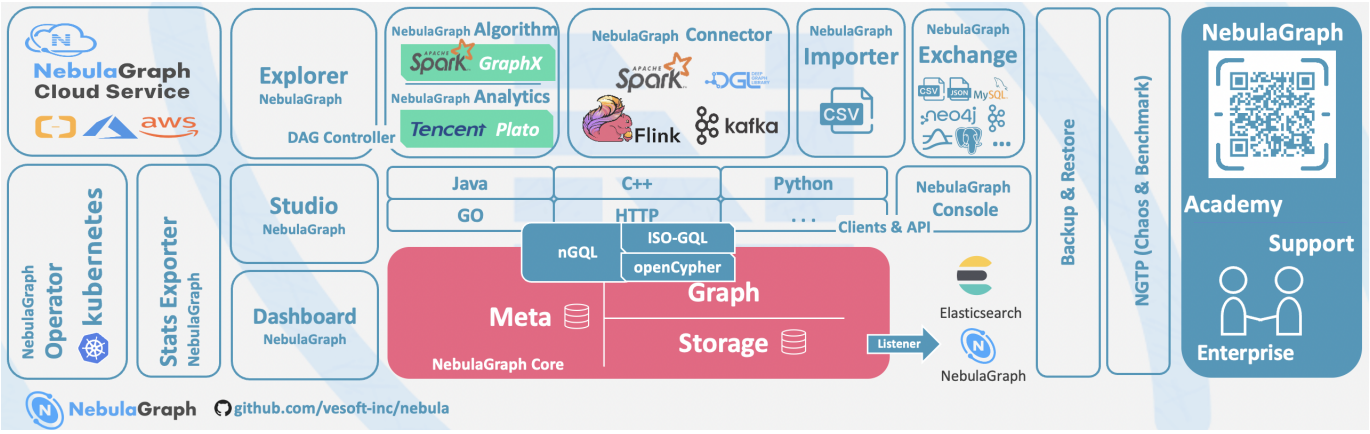
NebulaGraph 文档以 Markdown 语言编写。单击文档标题右上侧的铅笔图标即可提交修改建议。

最后更新: March 4, 2024

2. 简介

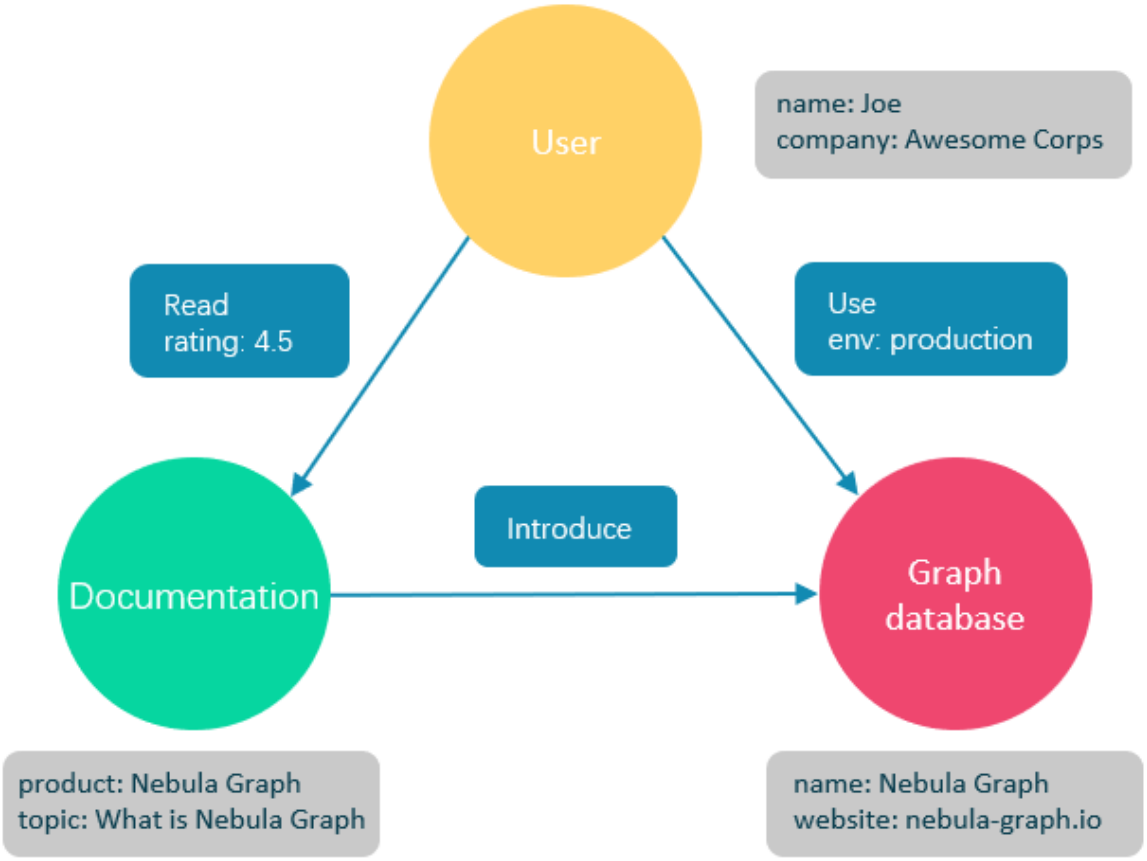
2.1 什么是 NebulaGraph

NebulaGraph 是一款开源的、分布式的、易扩展的原生图数据库，能够承载包含数千亿个点和数万亿条边的超大规模数据集，并且提供毫秒级查询。



2.1.1 什么是图数据库

图数据库是专门存储庞大的图形网络并从中检索信息的数据库。它可以将图中的数据高效存储为点（Vertex）和边（Edge），还可以将属性（Property）附加到点和边上。



图数据库适合存储大多数从现实抽象出的数据类型。世界上几乎所有领域的事物都有内在联系，像关系型数据库这样的建模系统会提取实体之间的关系，并将关系单独存储到表和列中，而实体的类型和属性存储在其他列甚至其他表中，这使得数据管理费时费力。

NebulaGraph 作为一个典型的图数据库，可以将丰富的关系通过边及其类型和属性自然地呈现。

2.1.2 NebulaGraph 的优势

开源

NebulaGraph 是在 Apache 2.0 条款下开发的。越来越多的人，如数据库开发人员、数据科学家、安全专家、算法工程师，都参与到 NebulaGraph 的设计和开发中来，欢迎访问 NebulaGraph GitHub 主页参与开源项目。

高性能

基于图数据库的特性使用 C++ 编写的 NebulaGraph，可以提供毫秒级查询。众多数据库中，NebulaGraph 在图数据服务领域展现了卓越的性能，数据规模越大，NebulaGraph 优势就越大。详情请参见 NebulaGraph benchmarking 页面。

易扩展

NebulaGraph 采用 shared-nothing 架构，支持在不停止数据库服务的情况下扩缩容。

易开发

NebulaGraph 提供 Java、Python、C++ 和 Go 等流行编程语言的客户端，更多客户端仍在开发中。详情请参见 NebulaGraph clients。

高可靠访问控制

NebulaGraph 支持严格的角色访问控制和 LDAP（Lightweight Directory Access Protocol）等外部认证服务，能够有效提高数据安全性。详情请参见验证和授权。

生态多样化

NebulaGraph 开放了越来越多的原生工具，例如 NebulaGraph Studio、NebulaGraph Console、NebulaGraph Exchange 等，更多工具可以查看生态工具概览。

此外，NebulaGraph 还具备与 Spark、Flink、HBase 等产品整合的能力，在这个充满挑战与机遇的时代，大大增强了自身的竞争力。

兼容 **openCypher** 查询语言

NebulaGraph 查询语言，简称为 nGQL，是一种声明性的、部分兼容 openCypher 的文本查询语言，易于理解和使用。详细语法请参见 nGQL 指南。

面向未来硬件，读写平衡

闪存型设备有着极高的性能，并且价格快速下降，NebulaGraph 是一个面向 SSD 设计的产品，相比于基于 HDD + 大内存的产品，更适合面向未来的硬件趋势，也更容易做到读写平衡。

灵活数据建模

用户可以轻松地在 NebulaGraph 中建立数据模型，不必将数据强制转换为关系表。而且可以自由增加、更新和删除属性。详情请参见数据模型。

广受欢迎

腾讯、美团、京东、快手、360 等科技巨头都在使用 NebulaGraph 。详情请参见 NebulaGraph 官网。

2.1.3 适用场景

NebulaGraph 可用于各种基于图的业务场景。为节约转换各类数据到关系型数据库的时间，以及避免复杂查询，建议使用 NebulaGraph 。

欺诈检测

金融机构必须仔细研究大量的交易信息，才能检测出潜在的金融欺诈行为，并了解某个欺诈行为和设备的内在关联。这种场景可以通过图来建模，然后借助 NebulaGraph，可以很容易地检测出诈骗团伙或其他复杂诈骗行为。

实时推荐

NebulaGraph 能够及时处理访问者产生的实时信息，并且精准推送文章、视频、产品和服务。

知识图谱

自然语言可以转化为知识图谱，存储在 NebulaGraph 中。用自然语言组织的问题可以通过智能问答系统中的语义解析器进行解析并重新组织，然后从知识图谱中检索出问题的可能答案，提供给提问人。

社交网络

人际关系信息是典型的图数据，NebulaGraph 可以轻松处理数十亿人和数万亿人际关系的社交网络信息，并在海量并发的情况下，提供快速的好友推荐和工作岗位查询。

2.1.4 视频

用户也可以通过视频了解什么是图数据。

- NebulaGraph 介绍视频（01 分 39 秒）

2.1.5 主题演讲

查看演讲快速了解图数据库概况。

2.1.6 相关链接

- 官方网站
- 文档首页
- 博客首页
- 论坛
- GitHub

最后更新: March 4, 2024

2.2 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。

2.2.1 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。

2.2.2 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。

Compatibility

由于 NebulaGraph 3.6.0 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。

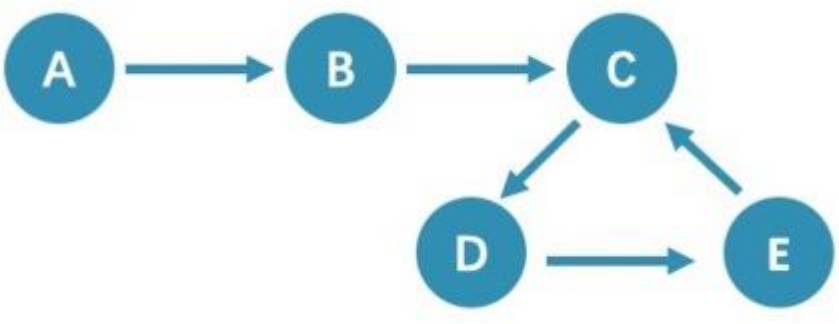
最后更新: March 4, 2024

2.3 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。

本文以下图为例进行简单介绍。



2.3.1 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。

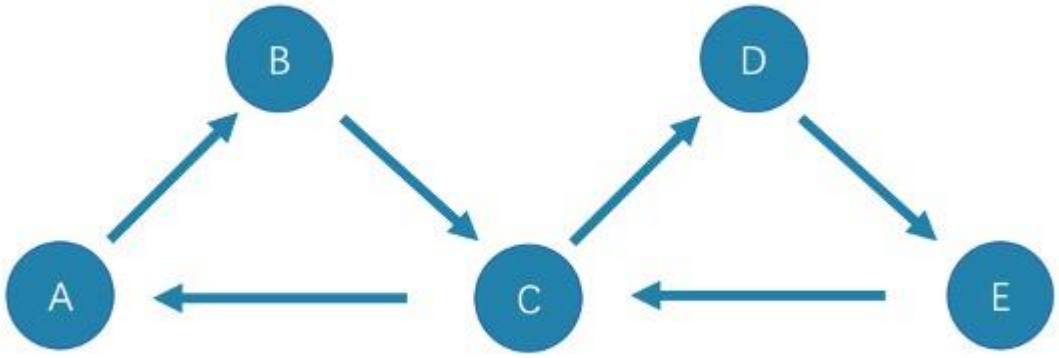
2.3.2 trail

`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->B->C->A` 或 `C->D->E->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->B->C->D->E->C->A` 。

2.3.3 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->B->C->D->E` 。

2.3.4 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: March 4, 2024

2.4 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。

2.4.1 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。

2.4.2 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（`|`）。

2.4.3 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，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。

2.4.4 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。

2.4.5 "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: March 4, 2024

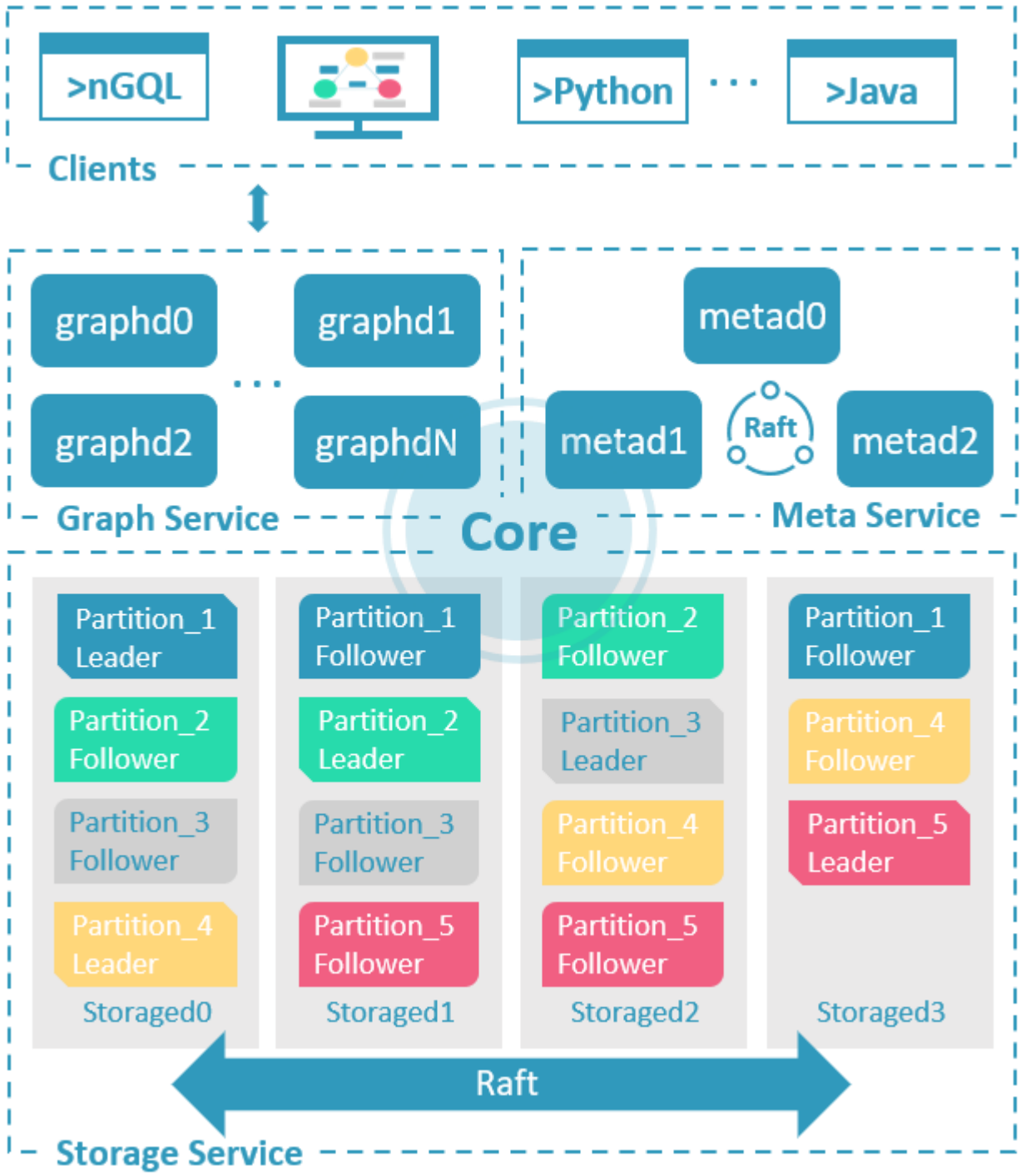
2.5 服务架构

2.5.1 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。

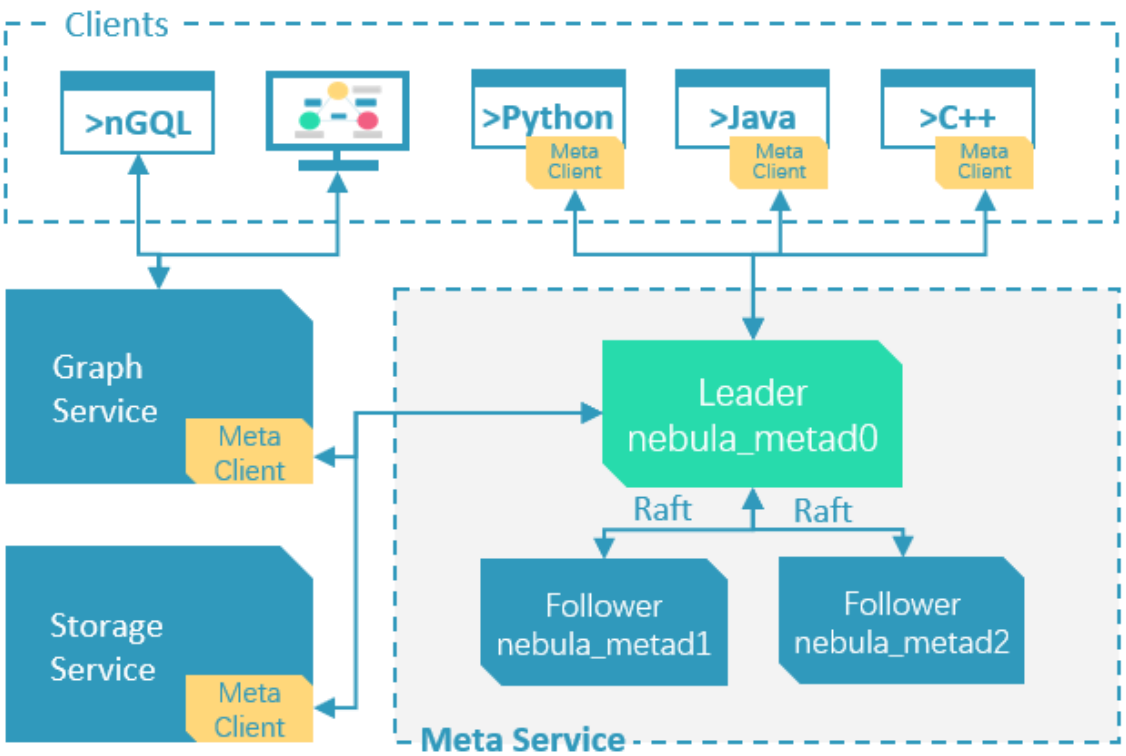
Graph 服务和 Storage 服务的详细说明，请参见 Graph 服务和 Storage 服务。

最后更新: March 4, 2024

2.5.2 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

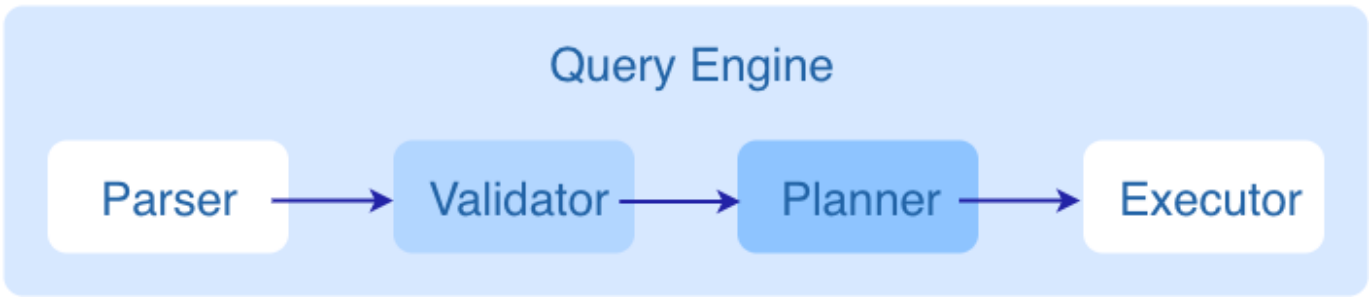
Meta 服务中的作业管理模块负责作业的创建、排队、查询和删除。

最后更新: March 4, 2024

2.5.3 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



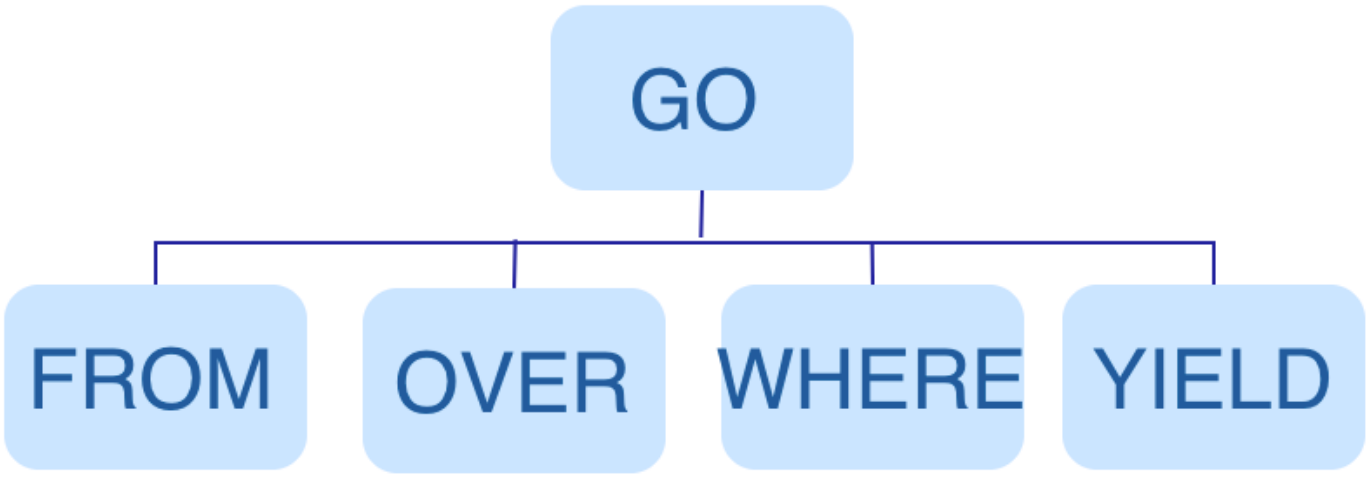
查询请求发送到 Graph 服务后，会由如下模块依次处理：

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。

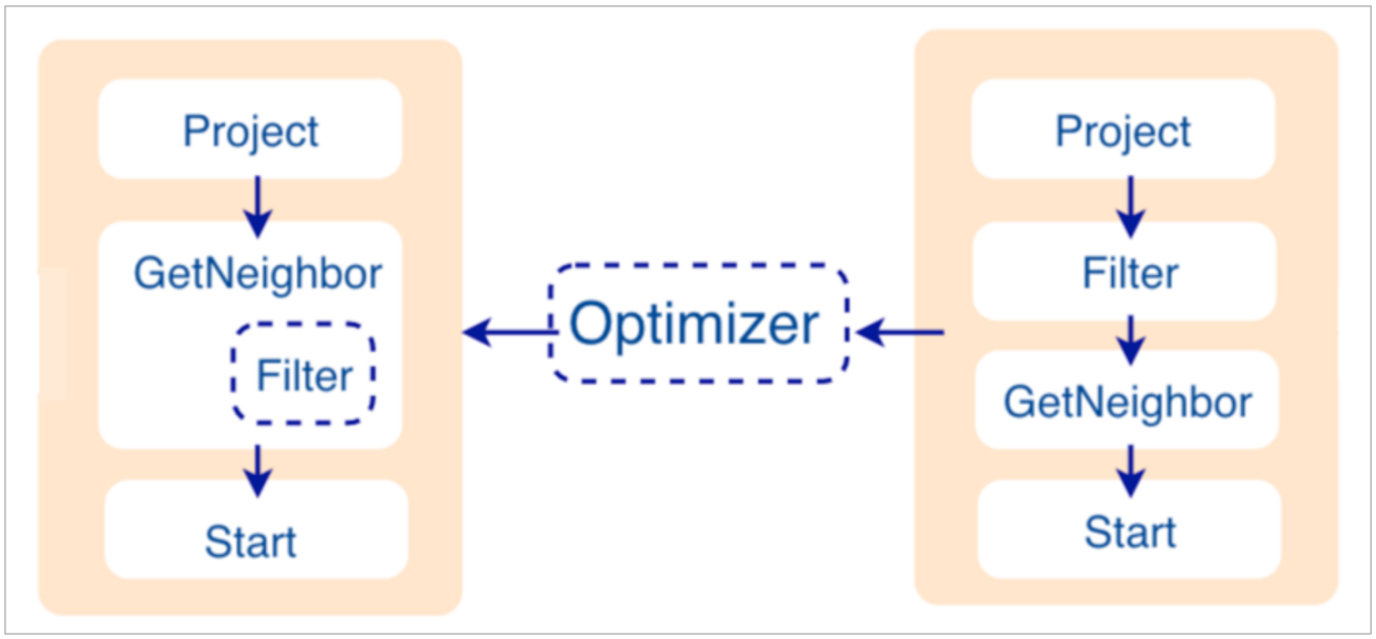
  例如语句 `GO FROM "Tim" OVER like YIELD dst(edge) AS ID | GO FROM $-.ID OVER serve YIELD dst(edge)`，Validator 模块会校验 `$-.ID` 在管道符左侧是否已经定义。

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

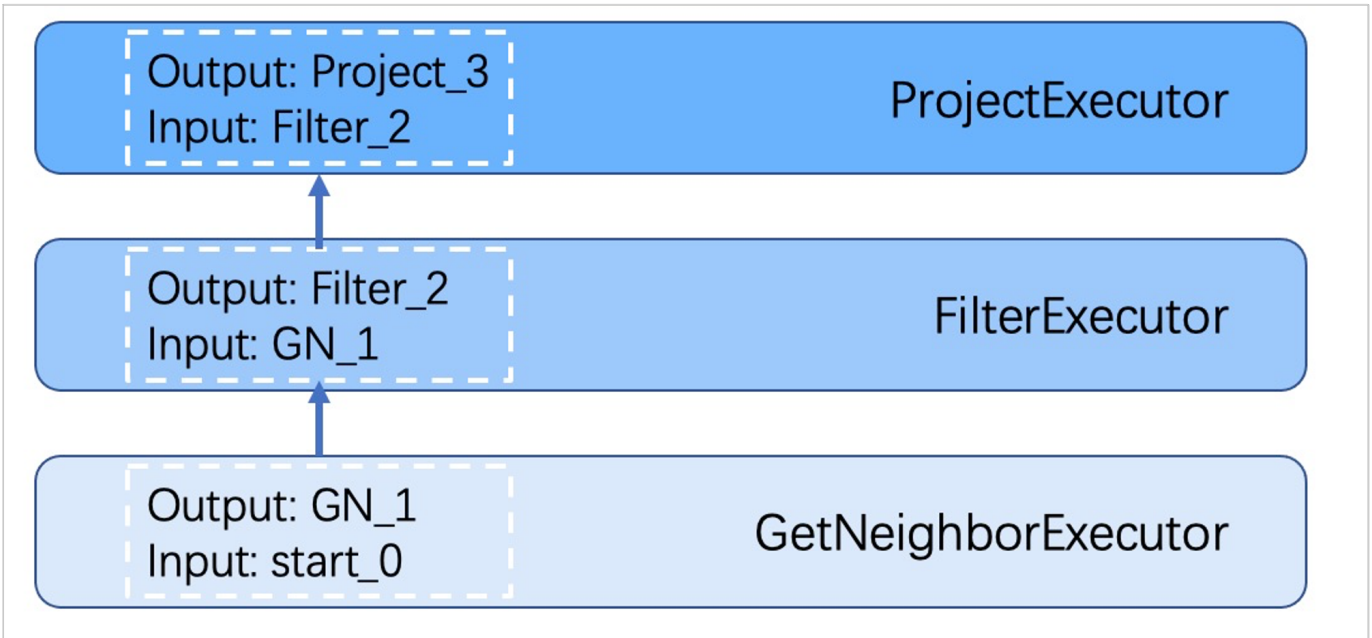
  Planner 模块目前的优化方式是 RBO（rule-based optimization），即预定义优化规则，然后对 Validator 模块生成的默认执行计划进行优化。新的优化规则 CBO（cost-based optimization）正在开发中。优化代码存储在仓库 `nebula-graph` 的目录 `src/optimizer/` 内。

  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: March 4, 2024

2.5.4 Storage 服务

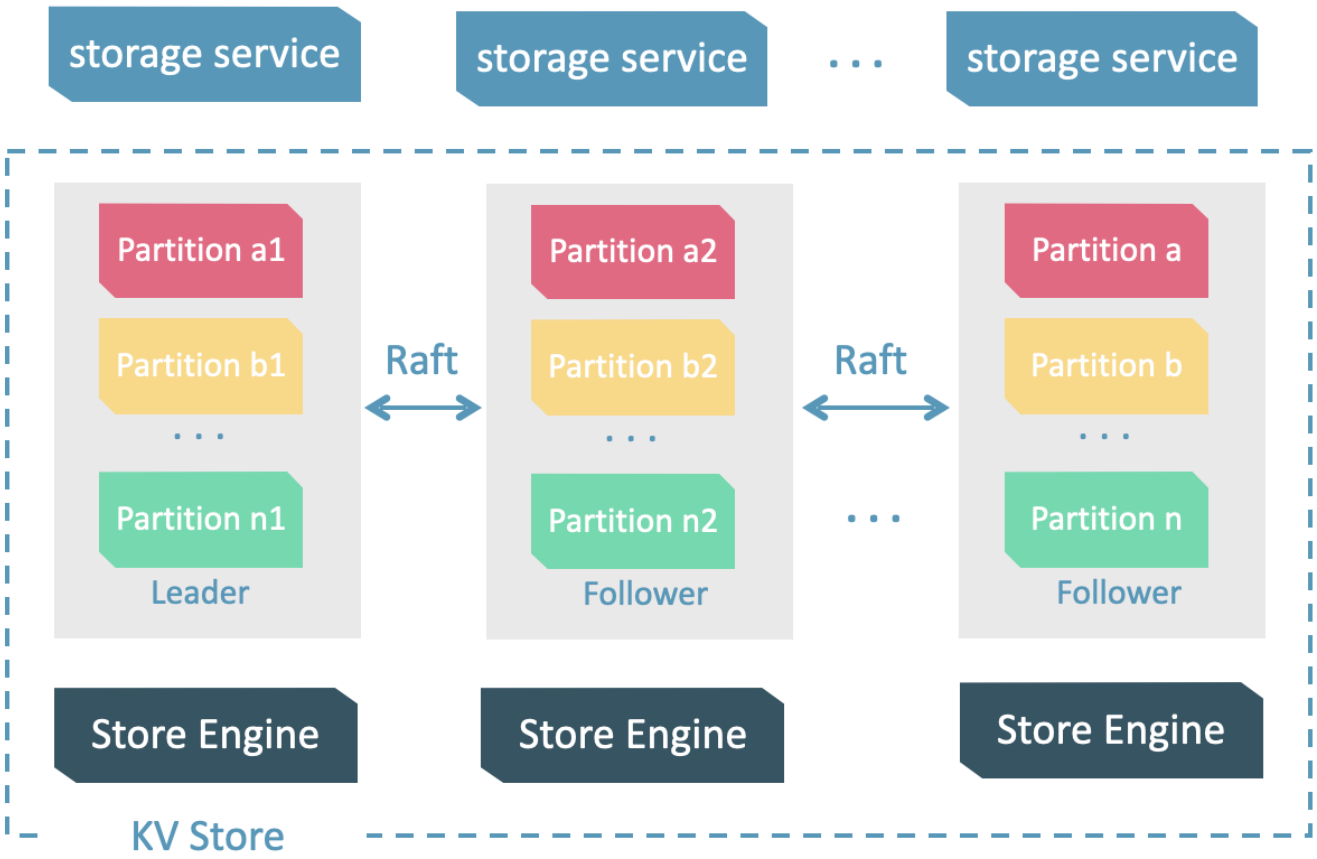
NebulaGraph 的存储包含两个部分，一个是 Meta 相关的存储，称为 Meta 服务，在前文已有介绍。

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

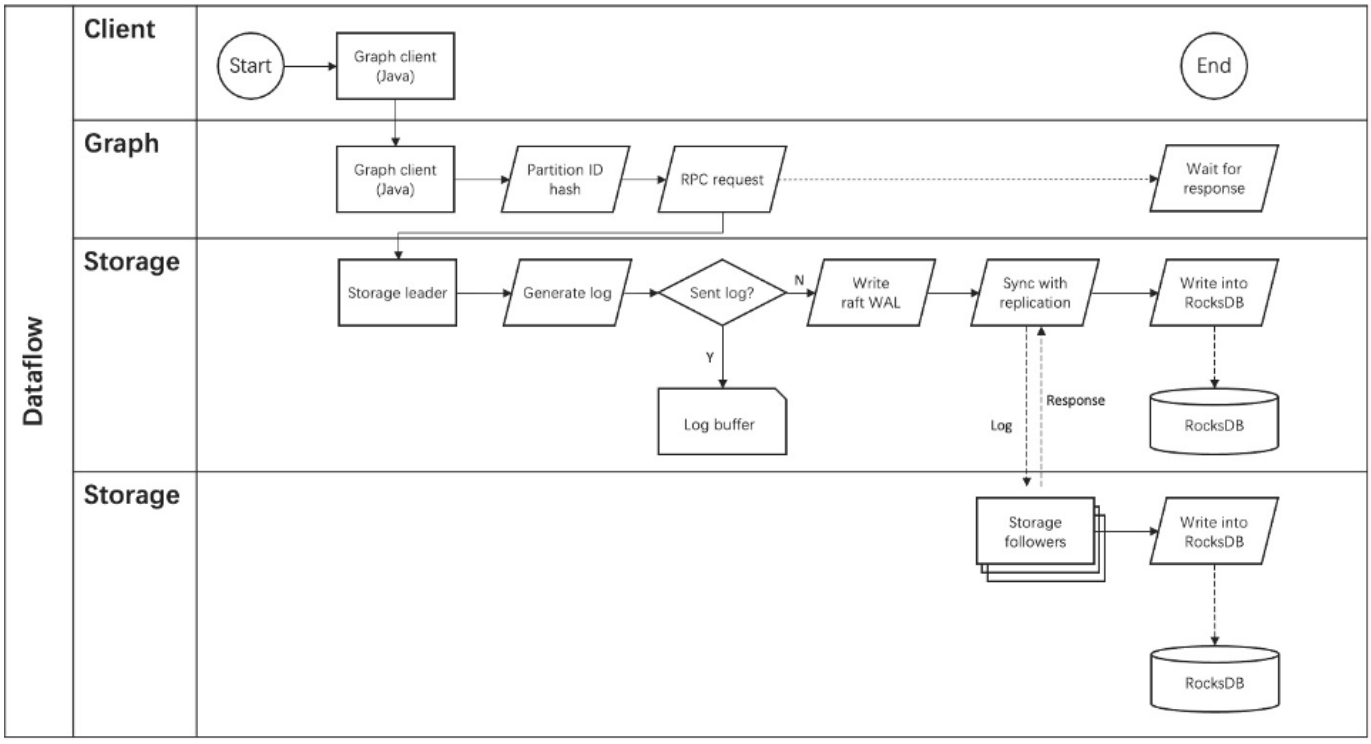
  Storage 服务的中间层，实现了 Multi Group Raft，保证强一致性和高可用性。

- 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。

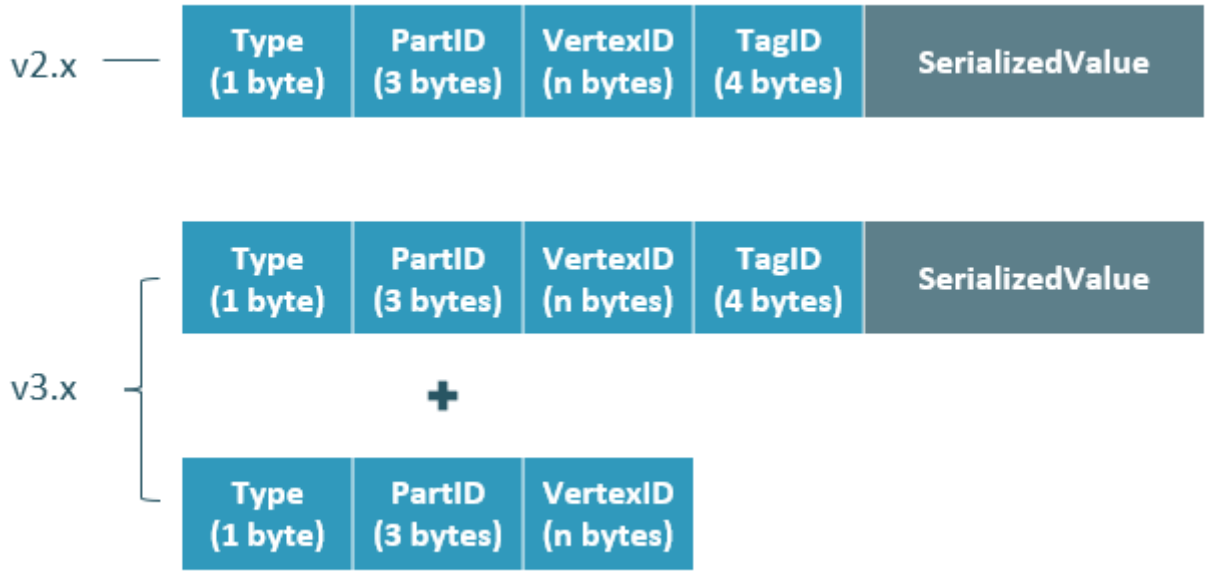
- 定制预写日志（WAL），每个分片都有自己的 WAL。
- 支持多个图空间，不同图空间相互隔离，每个图空间可以设置自己的分片数和副本数。

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 `fixed_string` 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。

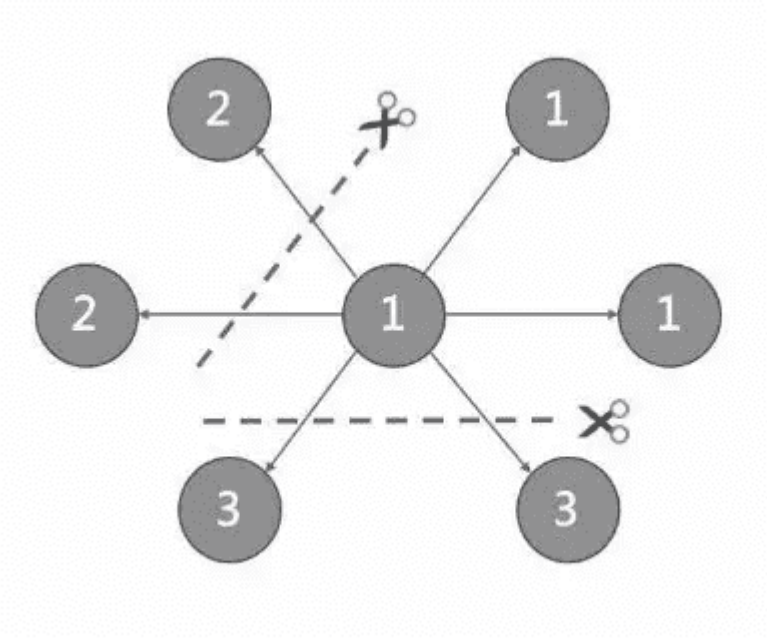
属性说明

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。由于定长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。

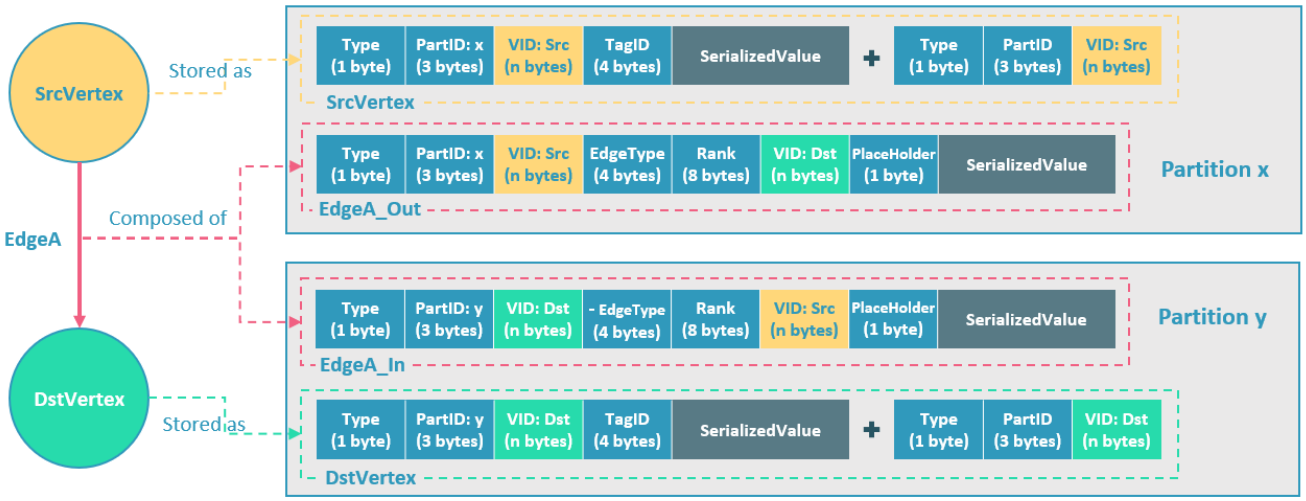
数据分片

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，是序列化的边属性，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 CREATE SPACE 语句中的 `partition_num` 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，其自身就是 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: March 4, 2024

3. 快速入门

3.1 基于 Docker 快速部署

NebulaGraph 提供了基于 Docker 的快速部署方式，可以在几分钟内完成部署。

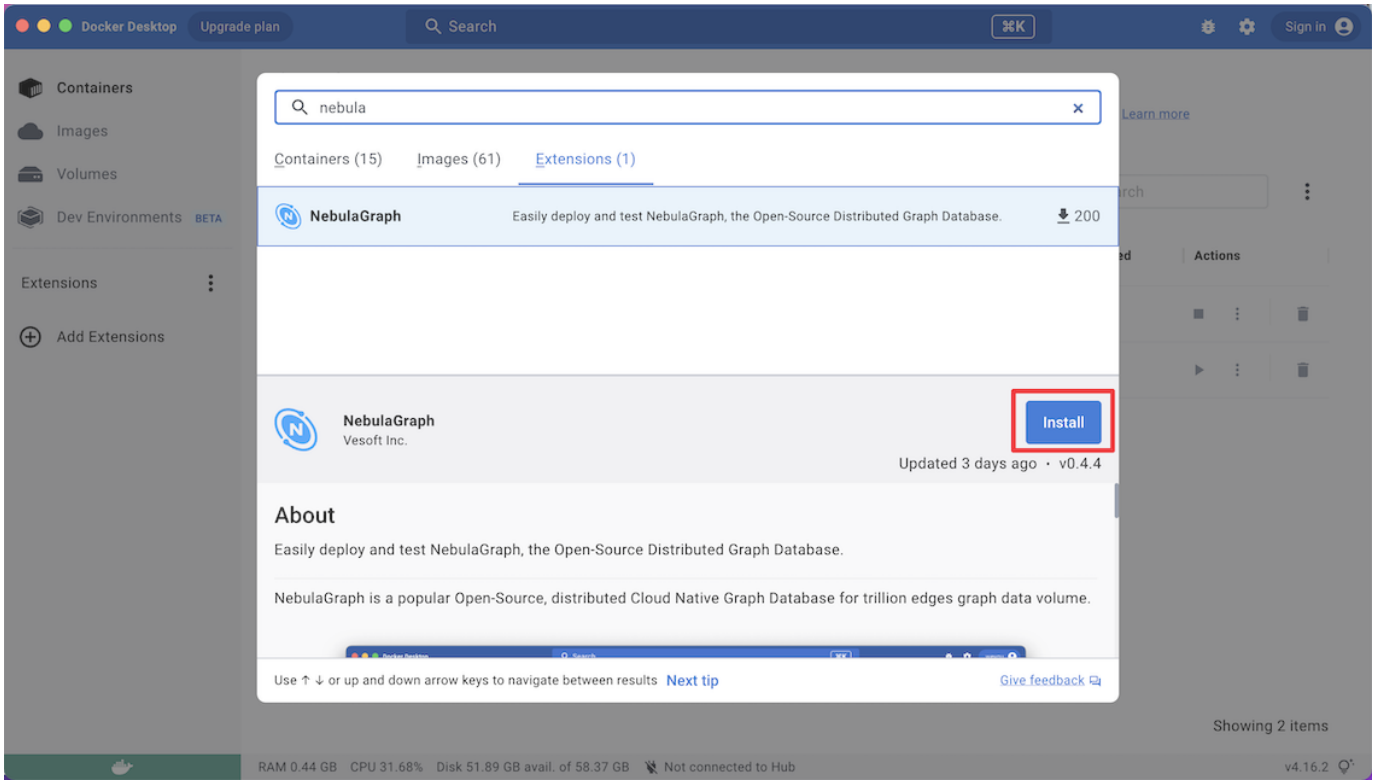
使用 **Docker Desktop**　　使用 **Docker Compose**

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。