**GridGraph图数据库用户手册**

（版本号：v1.2.0）

**二零二一年三月**

**目录**

[1. GridGraph概述](#_Toc2051541005_WPSOffice_Level1) [1](#_Toc2051541005_WPSOffice_Level1)

[2. GridGraph基础](#_Toc663727759_WPSOffice_Level1) [6](#_Toc663727759_WPSOffice_Level1)

[3. 查询、遍历及图计算](#_Toc176266200_WPSOffice_Level1) [15](#_Toc176266200_WPSOffice_Level1)

[4. GridGraph进阶](#_Toc1666398907_WPSOffice_Level1) [26](#_Toc1666398907_WPSOffice_Level1)

[5. 引用](#_Toc1814389920_WPSOffice_Level1) [34](#_Toc1814389920_WPSOffice_Level1)

# GridGraph概述

## 综述

## 在计算机科学中，图数据库（英语：graph database，GDB）是一个使用图结构进行语义查询的数据库，它使用节点、边和属性来表示和存储数据。

## GridGraph是一款通用的图数据库，提供图遍历、图计算分析环境，同时支持OLTP及OLAP图数据分析。GridGraph内建多种图遍历、图计算算法，适用于多种应用场景，如社交网络、金融欺诈检测、实时推荐引擎、知识图谱、工业领域等。

## GridGraph实现了Apache TinkerPop3框架，兼容Gremlin查询语言，业务应用代码可通过Java、Groovy、Python等多种语言连接到GridGraph图数据库以访问其数据。

## GridGraph的优势及特性

## GridGraph支持批量导入数据，支持多个图实例运行，支持多用户并发访问。用户可在命令行、编程的客户端等环境下通过Gremlin查询语言快速得到图查询遍历结果。

GridGraph具有如下优势：

* GridGraph是一个通用的图数据库系统，提供一体化的运行平台，同时支持图遍历和图计算；
* GridGraph是一个高效的图数据库系统，其核心计算层基于其优化的数据管理及计算算法，存储层基于其原生的存储设计，为业务应用提供了毫秒级的图数据查询能力；
* GridGraph是一个便利的图数据库系统，支持业界使用广泛的Apache Gremlin图查询语言，提供多种客户端连接方式，支持各种应用进行图查询计算。

GridGraph提供以下特性：

* 遵循属性图的数据模型；
* 对点或边的数据格式、数据类型等支持以模式(Schema)来进行元数据描述，方便应用的业务建模；
* 提供实时的图遍历查询；
* 支持Gremlin通用图遍历语言；
* 提供语言无关的接口方式，支持客户端多种语言编程；
* 支持基于csv及关系型数据库数据的批量导入；
* 提供优化的本地原生图数据存储能力，提高了数据的存储效率和访问速度。

## 快速开始

本节将使用一个名为modern的图实例进行演示，此图的结构及数据如下所示：

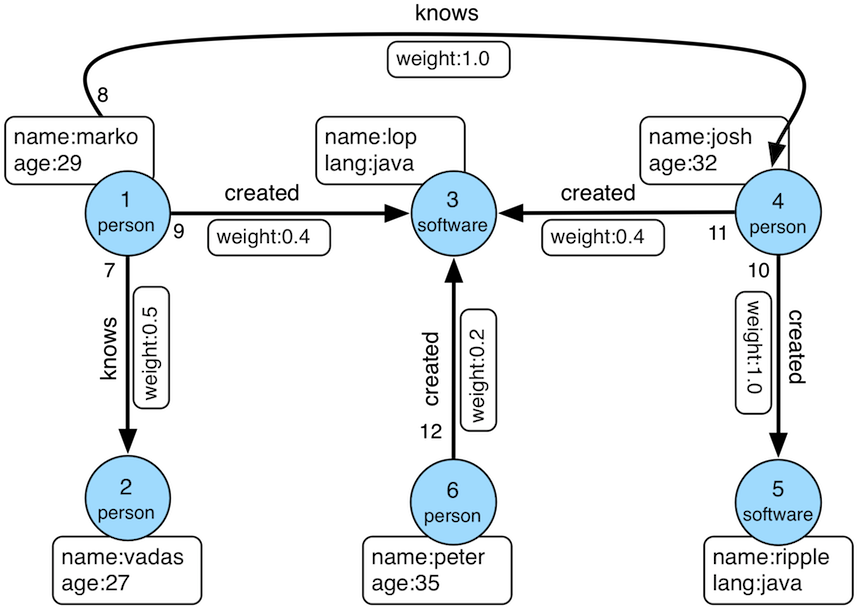


图１.1 名为“modern”的演示用图实例

下面将演示如何在GridGraph上来定义这个“modern”图的元数据结构、加入数据并进行查询。演示过程假定GridGraph服务器已经安装成功，Gremlin Console命令行客户端程序已经安装成功并可以成功连接到GridGraph服务器。具体服务器端及客户端安装过程请参见“GridGraph基础”部分内容。

## 为modern图准备元数据

在Gremlin Console命令行程序连接到GridGraph服务器后，在Console命令行里运行下面的脚本来创建modern图元数据结构：

gremlin> graph = GridGraphFactory.createGraph('modern')

==>gridgraph[GridGraph: modern]

gremlin> graph.createPrimaryKey('name')

==>[name - name, type - STRING, unique - true, nullable - false, defaultValue - null]

gremlin> personSchema = graph.createSchema('person', SchemaType.VERTEX)

==>[[1] - person ]

gremlin> ageProperty = personSchema.createProperty('age', GridDataType.INTEGER, false, false, null)

==>[name - age, type - INTEGER, unique - false, nullable - false, defaultValue - null]

gremlin> softwareSchema = graph.createSchema('software', SchemaType.VERTEX)

==>[[2] - software ]

gremlin> softwareSchema.createProperty('lang', GridDataType.STRING, false, false, null)

==>[name - lang, type - STRING, unique - false, nullable - false, defaultValue - null]

gremlin> knowsSchema = graph.createSchema('knows', SchemaType.EDGE)

==>[[3] - knows ]

gremlin> knowsSchema.createProperty('weight', GridDataType.DOUBLE, false, false, null)

==>[name - weight, type - DOUBLE, unique - false, nullable - false, defaultValue - null]

gremlin> createdSchema = graph.createSchema('created', SchemaType.EDGE)

==>[[4] - created ]

gremlin> createdSchema.createProperty('weight', GridDataType.DOUBLE, false, false, null)

==>[name - weight, type - DOUBLE, unique - false, nullable - false, defaultValue - null]

## 加入点和边

对应于modern图结构，接下来可运行以下Gremlin语句来加入点、边：

gremlin> marko = graph.addVertex(T.label, "person", "name", "marko", "age", 29)

==>v[281474976710656]

gremlin> vadas = graph.addVertex(T.label, "person", "name", "vadas", "age", 27)

==>v[281474976710657]

gremlin> lop = graph.addVertex(T.label, "software", "name", "lop", "lang", "java")

==>v[562949953421312]

gremlin> josh = graph.addVertex(T.label, "person", "name", "josh", "age", 32)

==>v[281474976710658]

gremlin> ripple = graph.addVertex(T.label, "software", "name", "ripple", "lang", "java")

==>v[562949953421313]

gremlin> peter = graph.addVertex(T.label, "person", "name", "peter", "age", 35)

==>v[281474976710659]

gremlin> marko.addEdge("knows", vadas, "weight", 0.5d)

==>e[844424930131968][281474976710656-knows->281474976710657]

gremlin> marko.addEdge("knows", josh, "weight", 1.0d)

==>e[844424930131969][281474976710656-knows->281474976710658]

gremlin> marko.addEdge("created", lop, "weight", 0.4d)

==>e[1125899906842624][281474976710656-created->562949953421312]

gremlin> josh.addEdge("created", ripple, "weight", 1.0d)

==>e[1125899906842625][281474976710658-created->562949953421313]

gremlin> josh.addEdge("created", lop, "weight", 0.4d)

==>e[1125899906842626][281474976710658-created->562949953421312]

gremlin> peter.addEdge("created", lop, "weight", 0.2d)

==>e[1125899906842627][281474976710659-created->562949953421312]

gremlin> graph.tx().commit()

==>null

需要注意的是，数据操作需要使用“graph.tx().commit()”来提交事务及操作的数据。

## 遍历图数据

将数据添加到图数据库之后，就可以开始进行遍历和分析了。

先获得一个”g”对象，用来进行遍历访问：

gremlin> g=graph.traversal()

==>graphtraversalsource[gridgraph[GridGraph: modern], standard]

通过这个”g”对象可获取点或边，获取过程可提供过滤条件：

gremlin> g.V()

==>v[562949953421312]

==>v[562949953421313]

==>v[281474976710656]

==>v[281474976710657]

==>v[281474976710658]

==>v[281474976710659]

gremlin> g.E()

==>e[1125899906842624][281474976710656-created->562949953421312]

==>e[1125899906842625][281474976710658-created->562949953421313]

==>e[1125899906842626][281474976710658-created->562949953421312]

==>e[1125899906842627][281474976710659-created->562949953421312]

==>e[844424930131968][281474976710656-knows->281474976710657]

==>e[844424930131969][281474976710656-knows->281474976710658]

gremlin> g.V().has('name','marko')

==>v[281474976710656]

获取点后，可以通过这个点遍历到其关联的边及边的邻接点，并可以得到点的特定属性值：

gremlin> g.V().has('name','marko').outE('created')

==>e[1125899906842624][281474976710656-created->562949953421312]

gremlin> g.V().has('person','name','marko').outE('created').inV()

==>v[562949953421312]

gremlin> g.V().has('person','name','marko').out('created')

==>v[562949953421312]

gremlin> g.V().has('person','name','marko').out('created').values('name')

==>lop

# GridGraph基础

## 环境

部署GridGraph的环境要求如下：

内存：8G；

操作系统：64位Linux(Ubuntu或CentOS)；

Java版本：jdk1.8

## 安装

获取已发布的GridGraph软件包：gridgraph-server-1.2.0.tar.gz

选择需要安装的目录，对软件包进行解压，解压后如下图：

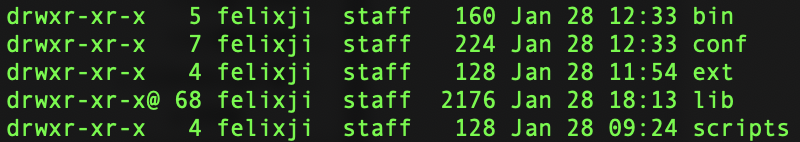


图2.1 GridGraph软件目录图

## 配置

配置文件在conf目录下，需要更新如下配置：

1）更新conf / gremlin-server.yaml，为服务启动配置端口，JVM heap。修改authentication.config.credentialsDb，此参数是存放GridGraph安全数据的根目录。如果不启用安全认证模式，可以移除相关的配置项（authentication和ssl）。

2）更新conf / gridgraph-config.properties，将gridgraph.directory更改为GridGraph数据文件夹的正确路径。确保文件夹存在并且授予写访问权限。全部配置项如下：

#图数据库数据文件的根目录

gridgraph.directory=/home/gdb

#关系型数据库文件的根目录

gridgraph.rdbms.directory=/home/rdb

#默认图，在服务启动时会自动打开。非必要项

gridgraph.defaultgraph=modern

#打开审计日志

gridgraph.enable.audit=true

#打开客户审计日志

griddgraph.enable.audit.client=true

#引入插件，如果有多个用英文逗号隔开。非必要项

gridgraph.classimports=com.gridgraph.api.plugin.test.TestHeapAPI

3）更新conf / log4j-server.properties，将log4j.appender.gridGraphLoggingFile.File目录更改为想要的日志文件路径。确保文件夹存在并且授予写访问权限。

4）更新conf / log4j-server.properties，将log4j.appender.errorLoggingFile.File目录更改为更改为想要的日志文件路径。确保文件夹存在并且授予写访问权限。

5）将license文件（名为”gridgraph.license”）放置到conf/目录下。

## 运行

安装配置完成后进入bin目录，即可运行GridGraph，命令如下：

1）./gremlin-server.sh start 启动

2）./gremlin-server.sh stop 停止

3）./gremlin-server.sh restart 重启

注意：需要确保端口并未占用。

在文件gremlin-server.sh 中可以配置java option，如修改heap大小等。

## console命令行

目前，可以使用控制台UI客户端（v3.4.6）连接到GridGraph服务器。可通过https://archive.apache.org/dist/tinkerpop/3.4.6/apache-tinkerpop-gremlin-console-3.4.6-bin.zip获得控制台的副本。

在解压完成后，创建一个yaml文件（例如，其名称为“ remote.gridgraph.yaml ”）并将其放在控制台的“ conf”文件夹下。并相应地更新主机和端口，以指向之前部署的GridGraph服务器。yaml文件的内容如下所示：

hosts: xx.xxx.xxx.xx

port: xxxx

serializer: { className:org.apache.tinkerpop.gremlin.driver.ser.GraphBinaryMessageSerializerV1, config: { serializeResultToString: true }}

#如果服务启用了安全认证，还需要配置如下配置项

username: admin

password: admin

connectionPool: {

enableSsl: false,

sslEnabledProtocols: [TLSv1.2] }

在控制台文件夹中，运行以下命令（以Linux OS作为运行控制台UI的示例环境。对于Windows，请使用bin / gremlin.bat）即可运行控制台：

./bin/gremlin.sh

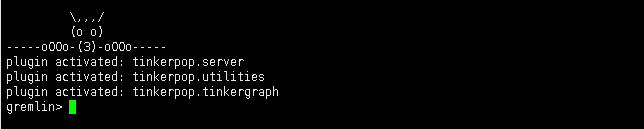


图2.2 控制台UI客户端

退出控制台UI客户端命令为，:exit（冒号+exit）

## Java客户端

通过引入Java代码，基于Java的客户端应用程序也可以连接到GridGraph服务器。

### Java版本要求

需支持1.8版本的Java

### 开发环境准备

需要以下maven依赖项才能使用Java连接到GridGraph。

<dependencies>

<dependency>

<groupId>com.gridgraph</groupId>

<artifactId>gridgraph</artifactId>

<version>1.2.0</version>

</dependency>

<dependency>

<groupId>org.apache.tinkerpop</groupId>

<artifactId>gremlin-driver</artifactId>

<version>3.4.6</version>

</dependency>

</dependencies>

建议您始终获取最新的“ com.gridgraph：gridgraph”，该文件可从GridGraph maven存储库中获得

### 创建连接

**服务端启用安全认证的连接方式：**

GridGraphSecureCluster cluster = new GridGraphSecureCluster("127.0.0.1", 8182, "admin", "admin");

Client client = cluster.connect(true);

**服务端未启用安全认证的连接方式：**

在Java客户端代码内部，需要进行配置以指定要连接的远程服务器：conf / gridgraph-remote.properties

gremlin.remote.remoteConnectionClass=org.apache.tinkerpop.gremlin.driver.remote.DriverRemoteConnection

# cluster file has the remote server configuration

gremlin.remote.driver.clusterFile=conf/remote-objects.yaml

上面的文件指的是yaml文件，我们需要在同一文件夹下使用此yaml（请记住将主机/端口更新为正确的远程服务器）

hosts: xxx.xxx.xxx.xx

port: xxxx

serializer: {

className: org.apache.tinkerpop.gremlin.driver.ser.GryoMessageSerializerV3d0,

config: {

ioRegistries: [

com.gridgraph.structure.GridIoRegistry

]

}

}

创建客户类示例：

提供了一个基类“ GridGraphBaseClient”，因此可以轻松创建客户端类。需要确保“ gridgraph-remote.properties”文件的路径应作为参数给出。如下：

public class GridGraphExampleClientApp extends GridGraphBaseClient {

......

}

使用上述基本客户端类，扩展的客户端代码可以通过调用以下命令连接到远程服务器：

super.connect()

### 使用

以下示例用于显示如何使用我们的客户端代码创建图形，为其创建元数据：

final StringBuilder sb = new StringBuilder();

sb.append("graph = GridGraphFactory.createGraph('TestGridGraph');");

sb.append("pk = graph.createPrimaryKey('name');");

sb.append("personSchema = graph.createSchema('person', SchemaType.VERTEX); ");

sb.append("sp1 = personSchema.createProperty('age', GridDataType.INTEGER, false, false, null);");

sb.append("softwareSchema = graph.createSchema('software', SchemaType.VERTEX);");

sb.append("softwareSchema.createProperty('lang', GridDataType.STRING, false, false, null);");

sb.append("knowsSchema = graph.createSchema('knows', SchemaType.EDGE);");

sb.append("knowsSchema.createProperty('weight', GridDataType.DOUBLE, false, false, null);");

sb.append("createdSchema = graph.createSchema('created', SchemaType.EDGE);");

sb.append("createdSchema.createProperty('weight', GridDataType.DOUBLE, false, false, null);");

sb.append("graph.schemas();");

this.client.submit(sb.toString())

请注意，在调用“ super.connect（）”之后，可以使用对象“ this.client”提交请求。使用上述Java代码，将在远程GridGraph服务器端创建一个具有指定名称'TestGridGraph'的图形。

基于字符串的请求和基于字节码的请求：

上面的示例Java代码（用于创建图形）是基于字符串的请求，表示将字符串作为整个脚本发送到服务器端。对于此类请求，将返回最后一条语句的结果。客户端请求的一件事是，最好在脚本/请求中指定结果尽可能清晰。考虑一下诸如“ graph.schemas（）”之类的请求，尽管它有望获得结果，但实际上它只是返回带有对空对象的引用的数组，因为客户端响应仅是“引用”响应。对于此特定示例，如果希望获得该图的模式名称列表，则最好使用以下代码替换上面的字符串请求的最后一行：

sb.append("graph.schemas().stream().map{s -> s.getName()};");

将这一行请求发送到服务器端后，可以使用以下示例客户端代码来提取结果：

ResultSet resultSet = client.submit(sb.toString());

// following code will get expected result

List<Object> resultList = resultSet.all().get().stream().map(r -> r.getObject()).collect(Collectors.toList());

// let's print it out by using Jackson, we'll get: Schemas ["software","person","created","knows"] have been created

ObjectMapper mapper = new ObjectMapper();

List<Object> resultList = resultSet.all().get().stream().map(r -> r.getObject()).collect(Collectors.toList());

LOGGER.info("Schemas {} have been created", mapper.writeValueAsString(resultList));

同样，以下代码可用于获取远程GridGraph服务器内的所有图实例：

final StringBuilder sb = new StringBuilder();

sb.append("GridGraphFactory.openGraph('" + graphName + "');");

sb.append("GridGraphManager.getInstance().getGraphNames()");

ResultSet resultSet = client.submit(sb.toString());

List<Object> resultList = resultSet.all().get().stream().map(r -> r.getObject()).collect(Collectors.toList());

ObjectMapper mapper = new ObjectMapper();

LOGGER.info("Following GridGraph instances are running on remote graph server: {}", mapper.writeValueAsString(resultList))

以上请求以字符串形式发送。遍历操作的另一种选择是使用基于字节码的请求。使用这种客户端编码样式，客户端代码可以获得一个Java对象，该对象是远程图遍历的引用，然后编写纯Java语句，而不是字符串/文本来构建请求。

图远程图遍历

使用Java基类，客户端代码可以通过指定所创建图的名称来打开遍历，获取可以被重用于进一步遍历操作的引用遍历对象：

private GraphTraversalSource g;

......

this.g = this.traversal(this.graphName);

......

遍历与分析

如添加点和边

Vertex marko = g.addV("person").property("name", "marko").property("age", 29).next();

Vertex vadas = g.addV("person").property("name", "vadas").property("age", 27).next();

g.V(marko.id()).as("a").V(vadas.id()).addE("knows").property("weight", 0.5d).from("a").next();

然后可以进行更多的遍历和分析操作：

// check if a specific vertex exists or not

final Optional<Map<Object, Object>> v = g.V().has("name", "testPerson").elementMap().tryNext();

if (v.isPresent()) {

LOGGER.info("Reading element and get the vertex {}", v.get().toString());

} else {

LOGGER.warn("testPerson not found");

}

// get property for a vertex

Map<Object, Object> vertexValueMap = this.g.V().has("name", "marko").valueMap().next();

LOGGER.info("Get node marko's properties {}", vertexValueMap);

Set<Object> keys = vertexValueMap.keySet();

for (Object key: keys) {

Object value = vertexValueMap.get(key);

if (value instanceof List) {

LOGGER.info("Value for property {} is {}", key, ((List)value).get(0));

}

}

## 服务器日志

根据上述配置内容，在自己配置的日志路径下，可查看日志。

# 查询、遍历及图计算

下面主要介绍通过使用java API、Gremlin语言，创建图、查询图以及调用图数据库底层提供的算法。

## 元数据、模式/Schema

### 使用JAVA API创建图样例函数

下图示例创建一个graphName图的样例过程：

1. 根据图名创建图；
2. 给当前创建的图建立主键；
3. 创建一个结点类型person的顶点；
4. 给person顶点添加一个年龄属性，类型为Interger
5. 再创建一个software类型的顶点;
6. 给software顶点添加一个lang属性，类型为String;
7. 然后创建一个knows的边类型；
8. 给kowns边创建一个属性weight,类型为DOUBLE……



图 3.1

通过脚本打开图：

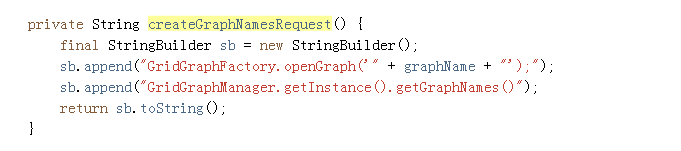


图 3.2

调用图 3.2和图 3.1方法示例：

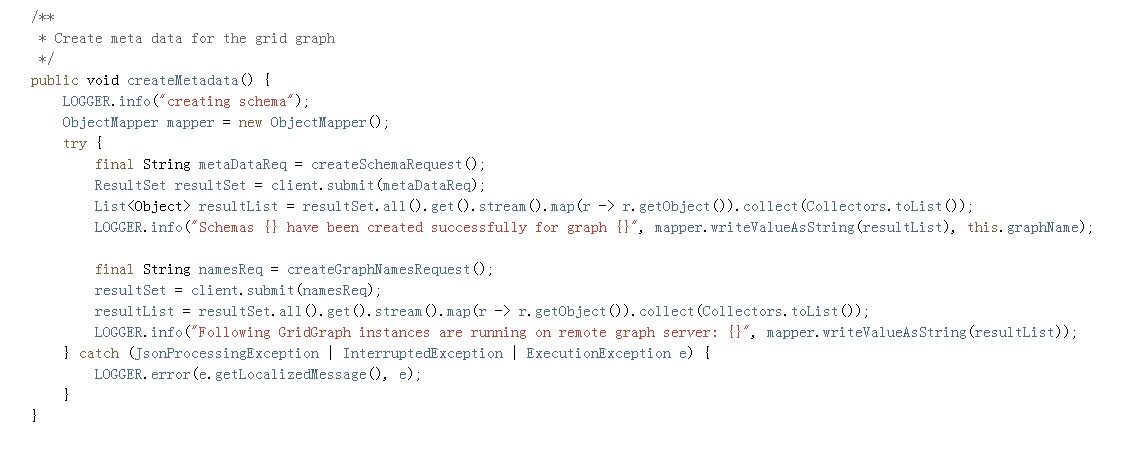


图 3.3

### 图的CRUD

给创建的图增加实例点和边数据：



图 3.4

更新顶点内容的属性样例，边类似：



图 3.5

查询图实例数据样例：

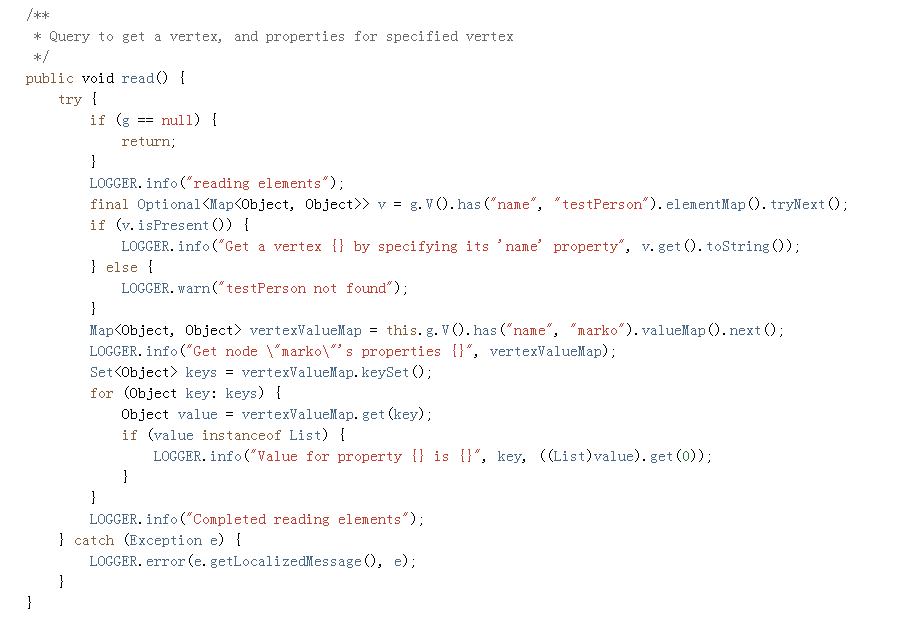


图 3.6

删除图数据样例：

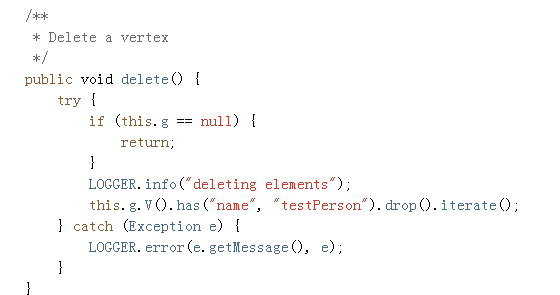


图 3.7

### 完整调用

下面展示对上面所有方法的一个完整调用过程，当前java类继承了GridGraphBaseClient类，定义了属性private GraphTraversalSource g和private String graphName;



图 3.8

## 数据类型

### Schema类型

在SchemaType里面定义了VERTEX、EDGE俩种类型

### 数据类型

在GridDataType里面有如下几种类型：

1. GridType.STRING
2. GridType.INTEGER
3. GridType.LONG
4. GridType.BOOLEAN
5. GridType.DATE
6. GridType.FLOAT
7. GridType.DOUBLE
8. GridType.LIST

## Gremlin查询语言及样例

Gridgraph图数据库上层集成了Gremlin查询语言框架，下面举例几个Gremlin语言样例，其它请参考相关官方文档。

1. V() :从图中读取顶点开始遍历
2. E():从图中读取边开始遍历
3. addV():遍历中增加顶点
4. addE()：遍历中增加边
5. hasNext():是否还有下一个结果
6. next():返回下一个结果
7. outE():出边
8. inE():入边
9. toList():返回所有结果为一个list
10. hasLabel:过滤标签
11. has():过滤属性和属性值
12. limit():限制个
13. bothE():双向边
14. values():获取值
15. count():获取数量
16. drop():删除

## 图计算

### 数据准备

下面脚本是创建测试图样例代码：



图 3.9

### 运行图计算算法

1. 运行pagerank算法
   1. 样例语言

graph.compute().program(PageRankVertexProgram.build().property("pageRank").create(graph)).submit().get().graph().traversal().V().order().by('pageRank', asc).elementMap('name', 'pageRank')

* 1. 结果

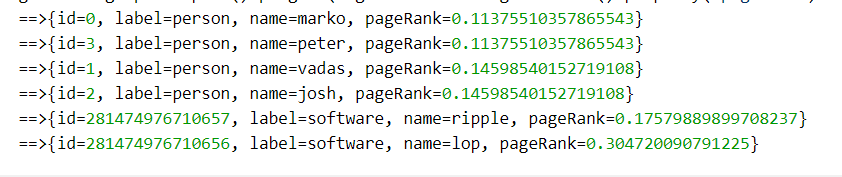


图 3.10

1. 运行带有顶点和边过滤条件的pagerank
   1. 样例语言

graph.compute().vertices(hasLabel("person")).edges(bothE("knows")).program(PageRankVertexProgram.build().property("pageRank").create(graph)).submit().get().graph().traversal().V().order().by('name', asc).elementMap('name', 'pageRank')

* 1. 结果

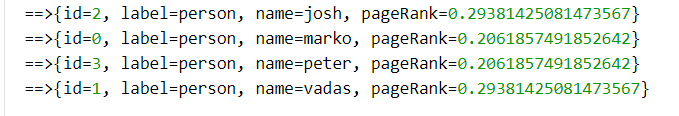


图 3.11

1. 运行shortest path
   1. 样例语言

graph.compute().program(ShortestPathVertexProgram.build().create()).submit().get().memory().get(ShortestPathVertexProgram.SHORTEST\_PATHS)

* 1. 结果



图 3.12

1. 运行PeerPressure
   1. 样例语言

graph.compute().program(PeerPressureVertexProgram.build().create(graph)).submit().get().graph().traversal().V().elementMap()

* 1. 结果

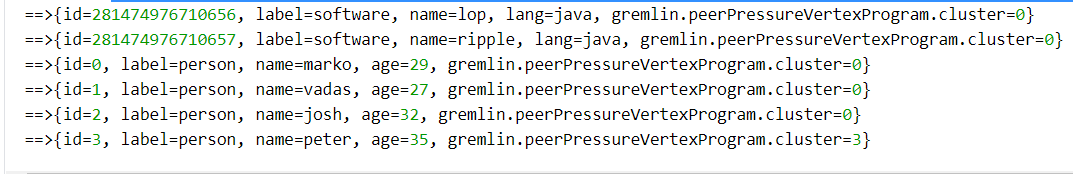


图 3.13

1. 在计算环境下运行遍历
   1. 样例语言

graph.traversal().withComputer().V().both().hasLabel('person').values('age').groupCount().next()

* 1. 结果

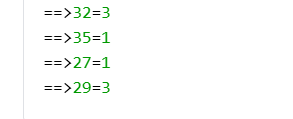


图 3.14

1. 运行OLAP聚类
   1. 样例语言

graph.compute().mapReduce(AggregationMapReduce.build().type(AggregationMapReduce.Type.COUNT).groupBy{v->v.label()}.create()).submit().get().memory().get('aggregationValue')

* 1. 结果

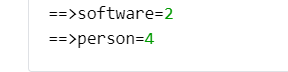


图 3.15

### 图计算任务相关操作

1. 提交多个图计算任务

graph.compute().vertices(hasLabel("person")).edges(bothE("knows")).program(PageRankVertexProgram.build().property("pageRank").create(graph)).submit().get().graph().traversal().V().order().by('pageRank', asc).elementMap('name', 'pageRank')

graph.compute().program(PageRankVertexProgram.build().property("pageRank").create(graph)).submit().get().graph().traversal().V().order().by('pageRank', asc).elementMap('name', 'pageRank')

1. 查询所有计算结果

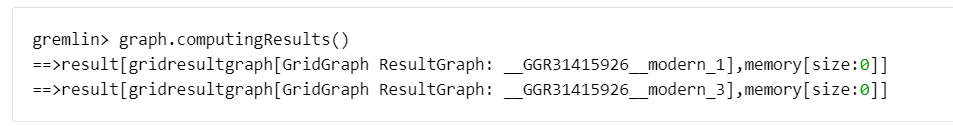


图 3.16

1. 获取相关结果的ID

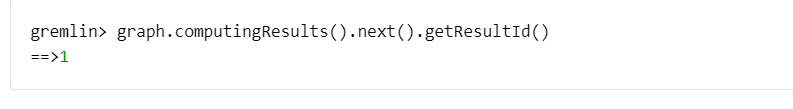


图 3.17

1. 通过Id获取结果内容



图 3.18

1. 遍历检索图计算结果

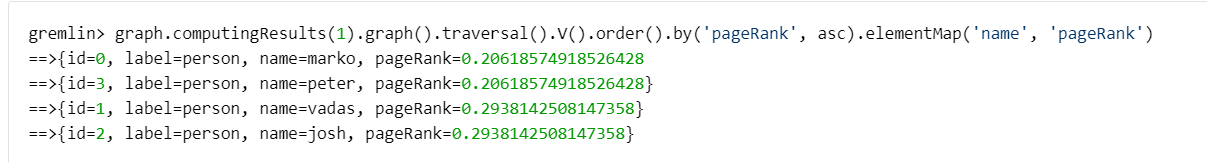


图 3.19

1. 通过Id删除对应的计算结果



图 3.20

1. 重新提交相同参数任务
   1. 语言

graph.compute().program(PageRankVertexProgram.build().property("pageRank").create(graph)).submit().get().graph().traversal().V().order().by('pageRank', asc).elementMap('name', 'pageRank')

* 1. 结果

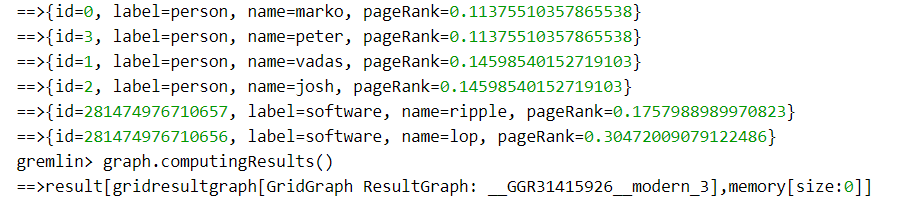


图 3.21

# GridGraph进阶

## 数据导入/bulkLoad

GridGraph数据导入有两种数据源csv文件，和关系数据库，支持数据源的编码方式为UTF-8。

## csv数据导入

源数据为标准化的csv格式，顶点数据每一列为顶点的数据的一个属性列，多种顶点类型可对应多个csv文件，以构建一个mordern图为例。

## 源文件准备

Person.csv

|  |  |
| --- | --- |
| |  | | --- | | marko,29  vadas,27  josh,32  peter,35 | |

Softeware.csv

|  |  |
| --- | --- |
| |  | | --- | | lop,java  ripple,java | |

第一个文件为对应为图里person节点，第二个文件对应为图

softwar节点。

边文件csv

knows.csv

|  |
| --- |
| marko,vadas,0.5  marko,josh,1.0 |

Created.csv

|  |  |
| --- | --- |
| |  | | --- | | marko,lop,0.4  josh,ripple,1.0  josh,lop,0.4  peter,lop,0.2 | |

第一个文件为know边，第二个文件为created边，边文件必须含有该边的出入顶点的标识信息。

## 配置文件

**图结构文件graphStruct.json该文件用来创建图模型**

|  |
| --- |
| {      "graphName": "modern",//图名称      "primaryKey": "name",//默认主键      "defaultSchema": "person",      "schemas": [ //图元素分别有顶点和边，为一个数组类型          {              "name": "person",//元素标签              "type": "vertex",//元素类型有vertex 顶点，edge边              "properties": [//顶点的属性列  ///"name": "name", //属性名  //"type": "STRING | INTEGER | LONG | BOOLEAN | DATE | FLOAT | DOUBLE | //LIST",//数据类型  // "subType": "STRING | INTEGER | LONG | BOOLEAN | FLOAT | DOUBLE", //如果属性声明LIST,  //"unique": **true**, //set to false if the type is LIST  //"nullable": **false**,  // "defaultValue": "abc"//Can be null if no default value                {"name": "name", "type": "STRING", "unique": **true**, "nullable": **false** },                  {"name": "age", "type": "INTEGER", "unique": **false**, "nullable": **true** }              ]          },          {              "name": "software",              "type": "vertex",              "properties": [                  {"name": "name", "type": "STRING", "unique": **true**, "nullable": **false** },                  {"name": "lang", "type": "STRING", "unique": **false**, "nullable": **true** }              ]          },          {              "name": "knows",              "type": "edge",              "properties": [                  {"name": "weight", "type": "DOUBLE", "unique": **false**, "nullable": **true** }              ]          },          {              "name": "created",              "type": "edge",              "properties": [                  {"name": "weight", "type": "DOUBLE", "unique": **false**, "nullable": **true** }              ]          }      ],    "indexs": [ //第二索引，图已经有一个主键索引了，如果不需要第二索引此处可为空（大部分情况下不需要）          {              "indexName": "name",              "propertyIndexs": [                  {                      "schemaName": "vert\_name",                      "propertyName": "name",                      "direction": "asc"                  }              ]          }      ]  } |

**导入映射关系**

|  |
| --- |
| {      "graphName": "modern",  "primaryey": "vert\_id",                                  //指定主键      "connection": {          "sourCsv": **true**,          "csvSpliteChar": ",",  //默认分隔符是英文逗号，默认quotechar是英文双引号          "csvFolderPath": "/home/user/sourcedata"     //csv 目录，把所有的顶点和边文件都存放该目录下      },        "vertexes": [{          "schemaName": "person",          "fileName": "person.csv",          "fieldMap" : {"vert\_id": 1, "prop1": 2, "prop2": 3}  //和csv文件的列序号对应关系例如vert\_id对应csv文件的第一列    },  {          "schemaName": "software",          "fileName": "software.csv",          "fieldMap" : {"vert\_id": 1, "prop1": 2, "prop2": 3}      }],        "edges": [{          "schemaName": "knows",          "fileName": "knows.csv",          "fromVertex": "person", //该边的源点标签          "toVertex": "person", //该边的目的点的标签          "fieldMap" : {"fromId": 1, "toId": 2}      }, {          "schemaName": "created",          "fileName": "created.csv",          "fromVertex": "person",          "toVertex": "software",          "fieldMap" : {"fromId": 1, "toId": 2}      }]  } |

## 执行命令

先创建图(具体版本号以release 版本为主)

|  |
| --- |
| java -Xms8G -Xmx12G -jar bulkLoad-x.x.x.jar -r /home/jack/modern -s /home/jack/modern/config/graphStruct.json |

导入数据(具体版本号以release 版本为主)

|  |
| --- |
| java -Xms8G -Xmx12G -jar bulkLoad-x.x.xar -r /home/jack/modern -f /home/jack/modern/config/bulkloadCsv.json |

导入数据jar包参数解析

|  |
| --- |
| -h  --help  打印帮助信息; -r  --rootGraph 数据库数据存放目录;  -s  --graphStuct  图结构文件，用来创建图;  -i  --increment 增量导入  -n  --schemaName  要导入的schema名称，如果没有的话所有schema都会导入.  -e  --throwEdgeError  指定参数，如果边的起点或者源点不存在会抛出错误异常，如果不指定改参数会跳过这些数据，并写入到日志  -f --fullDose 指定映射关系文件（bulkloadCsv.json）  -d --disableCacheVertexWhenLoadEdges 使用此参数关闭点缓存，如果任何类型的点的数量大于Integer.MAX\_VALUE，需要启用此参数  -c --createSchemas 为已有图创建新的schema |

## 关系库数据导入

关系数据库导入数据只需修改bulkLoadDb.json 文件即可

|  |
| --- |
| {      "graphName": "modern\_mysql",      "primaryKey": "name",      "connection": {          "driver": "com.mysql.cj.jdbc.Driver",          "url": "jdbc:mysql://localhost:3306/modern?useSSL=false", //填写关系库连接地址          "user": "root",          "password": "password"      },        "vertexes": [{          "schemaName":"person",          "sql":"select \* from person",          "fieldMap" : {"name": 1, "age": 2}      }, {          "schemaName":"software",          "sql":"select \* from software",          "fieldMap" : {"name": 1, "lang": 2}      }],      "edges": [{          "schemaName": "knows",          "sql": "select \* from knows",          "fromVertex": "person",          "toVertex": "person",          "fieldMap": {"fromId": 1, "toId": 2, "weight": 3}      },{          "schemaName": "created",          "sql": "select \* from created",          "fromVertex": "person",          "toVertex": "software",          "fieldMap": {"fromId": 1, "toId": 2, "weight": 3}      }]  } |

## 索引

系统支持在创建图的时候创建主键，代码调用方式 graph.createPrimaryKey(“name”)中间为创建主键的列名，系统会根据这个primary key 为每个顶点类型创建唯一索引。

也支持编写代码对指定节点类型，以及该节点的某一属性创建索引，具体方式为：

|  |
| --- |
| graph.createIndex(‘age-index’,’person’,’age’,IndexSortDirection.ASC);  //age-index 索引名称  // person 顶点类型  // age 顶点列名  //IndexSortDirection.ASC 索引方向  //删除索引  graph.dropIndex(‘age-index’) |

系统也支持编写代码对节点的度创建索引，并可以指定为某一种边的度、某几种边的度数的和、所有边类型的度数之和创建索引。创建索引的同时将为入度、出度、总度（出度＋入度）具体方式为：

|  |
| --- |
| // 为所有度数之和创建度的索引  graph.createDegreeIndex(‘degree-index1’, IndexSortDirection.ASC);  // 为”knows”边类型创建度的索引  graph.createDegreeIndex(‘degree-index2’, IndexSortDirection.ASC, “knows”);  // 为”knows”及”created”边类型创建度数的索引（这两种边的度的和）  graph.createDegreeIndex(‘degree-index3’, IndexSortDirection.ASC, “knows”, “created”);  //删除索引  graph.dropIndex(‘degree-index1’) |

## 事务

Gridgraph的每个尝试修改数据库信息的操作都会新建一个写事务该写事务在当前事务里信息可见，需在所有操作完成后进行commit提交。  
API体现为

|  |
| --- |
| g.getGraph().txn().commit(); |

## 服务器端插件

**GridGraph支持服务端插件开发 以下是代码样例**

|  |
| --- |
| package com.gridgraph.api.plugin.example;  import com.gridgraph.structure.GridGraph;  import com.gridgraph.structure.GridGraphFactory;  import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversalSource;  import org.apache.tinkerpop.gremlin.structure.Vertex;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class AlgorithmAPI {  private static final Logger LOGGER = LoggerFactory.getLogger(AlgorithmAPI.class);  public static ResponseData algorithm1(String graphName, String para1, Long para2) {  LOGGER.info("API call to drawMap for graph {}, para1={}, para2={}", graphName, para1, para2);  GridGraph graph = GridGraphFactory.openGraph(graphName);  GraphTraversalSource g = graph.traversal();  Vertex v = g.V(1).next();  LOGGER.info("Simulate to call graph tarversal to get v(1).property={}", v.property("name"));  return ResponseData.generateTestResponse();  }  public static ResponseData algorithm2(String graphName, String para1, Long para2, Long para3, Long para4, String para5) {  LOGGER.info("API call to drawMap for graph {}, para1={}, para2={}, " +  "para3={}, para4={}, para5={}", graphName, para1, para2, para3, para4, para5);  GridGraph graph = GridGraphFactory.openGraph(graphName);  GraphTraversalSource g = graph.traversal();  Vertex v = g.V(1).next();  LOGGER.info("Simulate to call graph tarversal to get v(1).property={}", v.property("name"));  return ResponseData.generateTestResponse();  }  } |

将该代码打包成jar包放入gridgraph-server-1.2.0/lib目录下（版本号以具体release版本为准），并配置conf目录下gridgraph-config.properties文件，在gridgraph.classimports=添加编写的自定义类(参考2.3配置),并重启服务。

## 通过http连接GridGraph服务器

http调用GridGraph服务器分为两种，一种是gremlin语言，一种是自定义插件接口分别为，调用都为post模式以curl为例

Gremlin接口为

|  |
| --- |
| Curl --request POST “ip:端口” --data‘{“gremlin”:“g.V().count”}’ |

自定义插件接口调用方式为：

|  |
| --- |
| Curl --request POST “ip:端口” --data‘{“gremlin”:“AlgorithmAPI.algorim1(‘TestGraph7’,’’param2’,18)”}’ |

# 引用

1. Apache TinkerPop Reference Documentation v 3.4.6: <https://tinkerpop.apache.org/docs/3.4.6/reference/>
2. 图数据库wiki: <https://zh.wikipedia.org/wiki/%E5%9B%BE%E6%95%B0%E6%8D%AE%E5%BA%93>