Gremlin语法分享

1、基本概念

1.1 TinkerPop和Gremlin

TinkerPop 是一个面向实时事务处理（OLAP）以及批量、分析型图分析（OLTP）的图计算框架，它是一个总称，包含若干子项目以及与核心TinkerPop Gremlin引擎集成的模块。 该框架还提供了Gremlin语言，这是一种图遍历语言，是其核心功能的一部分。

Gremlin是 Apache TinkerPop 框架下的图遍历语言。Gremlin是一种函数式数据流语言，可以使得用户使用简洁的方式表述复杂的属性图（property graph）的遍历或查询。每个Gremlin遍历由一系列步骤（可能存在嵌套）组成，每一步都在数据流（data stream）上执行一个原子操作。

Tinkerpop3 模型核心概念

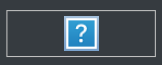
- Graph: 维护节点&边的集合，提供访问底层数据库功能，如事务功能
- Element: 维护属性集合，和一个字符串label，表明这个element种类
- Vertex: 继承自Element，维护了一组入度，出度的边集合
- Edge: 继承自Element，维护一组入度，出度vertex节点集合.
- Property: kv键值对
- VertexProperty: 节点的属性，有一组健值对kv，还有额外的properties 集合。同时也继承自element，必须有自己的id, label.
- Cardinality: 「single, list, set」 节点属性对应的value是单值，还是列表，或者set。

1.2 图数据库特征

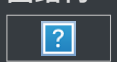
图数据库和关系型数据库的区别：

多种实体、关系混合存储

便于查找节点连接



图结构



1.3 分布式图切割

作为一种分布式图数据库，需要将数据切分存储到多台机器上。

1.3.1 边切割

JanusGraph
切割线穿过连接点的边，每个点只保存一次，切断的边会保存在多台机器上。可以节省存储空间，但对图进行基于边的计算时，对于一条两个顶点被分到不同机器上的边来说，跨机器通信传输数据，内网通信流量大。

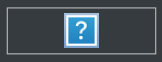
> **JanusGraph对于每条边都会被切断，该边在起始点目的点各存储一次，这样能快速找到对端的点。**

1.3.2 点切割

GraphX
每一条边只保存一次，邻居多的点会被分发到不同的机器上。可以大幅减少内网通信量，但是邻居多的点会复制到多台机器上，增加存储开销，引发数据同步问题。

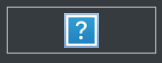
1.4 图存储模型

1.4.1 邻接矩阵



1.4.2 邻接表

1.4.3 原生处理能力



- 假如图数据库可存在免索引邻接（Index Free Adjacency）属性，那么就是具有原生处理能力.
- 免索引邻接主要是对用非原生图数据库里使用的全局索引而言的，也就是每个节点都会维护其相邻节点的引用，这样可以提供快速高效的图遍历性能。
- 拥有免索引邻接的图数据库引擎遍历整个图的复杂度为O(n)，而使用索引的复杂度则是O(nlogn)，但是在实际中，如果遇到要删除一个有大量连接的节点，这时的操作代价就会非常高昂.

1.5 GA相关业务

- 串并案分析，通过各种关联关系（涉案人、物、手段等）进行串并；
- 航班同机、同日、多目的地关联分析，查找嫌疑人及同伙
- 旅业同住关系分析，通过酒店与入住关系查找同住人
- 轨迹分析，通过航班、旅业、基站、QQ号IP地址等寻找轨迹

- 短信诈骗分析，通过电话及帐号关系进行关联及串并分析
- 盗抢案件分析，通过基站地点、话单时序发现上下游团伙
- 话单关系分析（发现团伙），话单时序分析（时间间隔及方向发现同伙）
- 频繁入住、频繁更换（酒店、SIM卡、手机）分析
- 洗钱分析，通过账单的关联关系找出资金流向，通过时序找出洗钱账户
- 类案分析，通过已抓获人员通讯录分析跨团伙未抓获嫌疑人
- 命案分析，通过对海量信息进行比对，快速生成否定库
- 社会网络分析，分析QQ、微博首发、转发关系，找到关键人员

1.6 常用算法

参考华为图引擎：[https://support.huaweicloud.com/usermanual-ges/ges_01_0031.html

1.6.1 中心性

衡量不同节点的重要程度

- PageRank
- Betweenness Centrality
- Closeness Centrality

1.6.2 路径查找

- Minimum Weight Spanning Tree
- All Pairs- and Single Source - Shortest Path
- A* Algorithm
- Yen’s K-Shortest Paths
- Random Walk

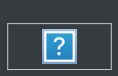
1.6.3 社区检测

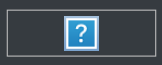
- Louvain
- Label Propagation
- (Weakly) Connected Components
- Strongly Connected Components
- Triangle Count / Clustering Coefficient

2、JanusGraph基本架构

2.1 技术点

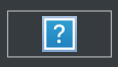
Hbase、ES、Tinkerpop/JanusGraph API、Gremlin



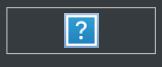


2.2 Hbase存储源码

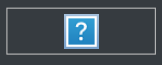
```
// hdfs路径查看
hadoop fs -ls
/hbase/data/default/yk_janus_20190122/cb988886dced15fe04f25f0761c5fdec/

// 源码设置：
package org.janusgraph.diskstorage.hbase;

public static BiMap<String, String> createShortCfMap(Configuration config)
{
    return ImmutableBiMap.<String, String>builder()
                .put(INDEXSTORE_NAME, "g")
                .put(INDEXSTORE_NAME + LOCK_STORE_SUFFIX, "h")
                .put(config.get(IDS_STORE_NAME), "i")
                .put(EDGESTORE_NAME, "e")
                .put(EDGESTORE_NAME + LOCK_STORE_SUFFIX, "f")
                .put(SYSTEM_PROPERTIES_STORE_NAME, "s")
                .put(SYSTEM_PROPERTIES_STORE_NAME + LOCK_STORE_SUFFIX, "t")
                .put(SYSTEM_MGMT_LOG_NAME, "m")
                .put(SYSTEM_TX_LOG_NAME, "l")
                .build();
  }

 // 混合索引存储：Storage graph indexes (Composite Index) data
Rowkey -> property values
Cell value->relationId、outVertexId、typeId、inVertexId
```



按照边切分结果：



节点的ID作为HBase的Rowkey，节点上的每一个属性和每一条边，作为该Rowkey的一个个独立的Cell。即**每一个属性、每一条边，都是一个个独立的KCV结构(Key-Column-Value)**。



2.3 事务

参考资料：https://zhuanlan.zhihu.com/p/50279477

> JanusGraph中的事务是自动开启（与MySQL类似），但不会自动commit或rollback的（与MySQL不同），所有的读写操作都在事务中执行。形如：
>
> juno = graph.addVertex() //Automatically opens a new transaction
> juno.property("name", "juno")
> graph.tx().commit() //Commits transaction
>
> 所以，如果一直执行插入或更新，那么会导致隐式开启的事务越来越大。如果一个图数据库实例被close()时，还有事务在执行中，那么该事务的状态是未定的（TinkPop语义），对于隐式开启的与该实例线程绑定的事务，jg会将其提交掉，如果是显式开启的非线程绑定的事务，则会被回滚掉。
> JanusGraph采用乐观事务模型，在提交时才会进行冲突检测，那么会导致在一个长事务中较早插入的具有唯一性约束的记录，在事务执行过程中被其他事务提交了。针对这种情况，可以在一个事务里面开启子事务，让子事务执行插入操作并先提交，这样能够提交长事务执行效率，避免无谓的失败。
>
> 多个事务间的隔离。JanusGraph中的事务类似于可重复读级别。事务开启后就会获取系统的状态（Snapshot？），在提交前不会更新，那么如果其他事务做了新的操作并提交，该事务是无法看到这些操作的。

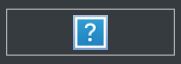
2.4 缓存

参考资料：https://zhuanlan.zhihu.com/p/50279477

> JanusGraph包含了事务层（Transaction-Level）、数据库层（Database Level）和后端存储（Storage Backend）缓存，其中第三种位于所配置的后端存储上，下面简单介绍下前2种。
>
> **事务层缓存**包括顶点缓存（Vertex Cache）和索引缓存（Index Cache）。
> 　　**顶点缓存**用于缓存顶点及其相连的边，其大小可通过**tx-cache-size** 设置，但被修改的vertex被pin在cache中无法替换，所以如果事务修改了很多vertex，那么cache大小会超过设置值。vertex用于加速事务对顶点的多次访问场景。
> 　　**索引缓存**用于缓存使用索引后端进行查询的结果，这样，后续使用相同sql进行查询的时候，就不需要在从索引后端跨网络获取数据，只需从内存中读取即可。最大大小为tx-cache-size 的一半。这个有点像query cache。
>
> **数据库层缓存**在事务间共享缓存数据。对于读多写少的场景，有较好的加速效果。默认关闭，可通过 **cache.db-cache=true**开启。**cache.db-cache-time**是影响该性能和查询行为最重要的参数，若仅有一个JanusGraph实例打开存储后端，或仅有该实例有修改存储后端数据，则可以将过期设置为0，表示不过期。
>
> 从cache.db-cache-time描述可发现，**JanusGraph各个实例之间并没有建立数据同步关系**，也就是说，一个实例修改了数据，另一个实例是不感知的。可以把JanusGraph理解为HBase等存储后端的客户端。客户端的缓存是本地缓存，服务器端数据被修改后，客户端的缓存当然不会感知。 **cache.db-cache-size** 设置数据库层缓存的大小，可以是其所属的JVM内存的百

> 分比，或者是具体的内存byte大小。过大的缓存会引起过多的GC，从而反过来影响性能，默认为50%。cache.db-cache-clean-wait表示本地事务修改了某数据后，等待多少时间将对应数据从缓存中失效掉，从存储后端重新获取。

2.5 CAP理论



一个分布式计算机系统无法同时满足以下三点（摘自Wikipedia）：

- **一致性（Consistency)**，所有节点访问同一份最新的数据副本
- **可用性（Availability)**，每次请求都能获取到非错的响应——但是不保证获取的数据为最新数据
- **分区容错性（Partition tolerance）**，以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

JanusGraph在CAP理论上的侧重，是要看底层存储的。如果底层是Cassandra，那么就是偏向于AP（Cassandra是最终一致性的）；如果底层是HBase，就是偏向于CP（强一致性）；BerkleyDB单机不存在这个问题。

2.6 BASE理论

BASE理论是对CAP理论的延伸，核心思想是即使无法做到强一致性（Strong Consistency，CAP的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。BASE是指**基本可用（Basically Available）**、**软状态（ Soft State）**、**最终一致性（ Eventual Consistency）**。

- **基本可用**是指分布式系统在出现故障的时候，允许损失部分可用性，即保证核心可用。
  电商大促时，为了应对访问量激增，部分用户可能会被引导到降级页面，服务层也可能只提供降级服务。这就是损失部分可用性的体现。
- **软状态**是指允许系统存在中间状态，而该中间状态不会影响系统整体可用性。分布式存储中一般一份数据至少会有三个副本，允许不同节点间副本同步的延时就是软状态的体现。mysql replication的异步复制也是一种体现。
- **最终一致性**是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。弱一致性和强一致性相反，最终一致性是弱一致性的一种特殊情况。

> **ACID**是传统数据库常用的设计理念，追求强一致性模型。BASE支持的是大型分布式系统，提出通过牺牲强一致性获得高可用性。
> ACID和BASE代表了两种截然相反的设计哲学。在分布式系统设计的场景中，系统组件对一致性要求是不同的，因此ACID和BASE又会结合使用。

2.7 常用优化

主要是配置hbase相关参数

云栖大会参考资料：Track2-5JanusGraph—Distributed graph database with HBase

> hbase.regionserver.thread.compaction.large/small
> hbase.hstore.flusher.count
> hbase.hregion.memstore.flush.sizeh
> base.hregion.memstore.block.multiplier
> hbase.hregion.percolumnfamilyflush.size.lower.bound
> hbase.regionserver.global.memstore.size
> hfile.block.cache.size
> hbase.regionserver.global.memstore.size.lower.limit
> （hbase.regionserver.global.memstore.lowerLimit）

3、Gremlin 基本语法

3.1 开图

3.1.1 Graph实现类

主要通过重写Graph内部Features接口，来定制Graph的特征，得到不同用途的Graph实现类。

- **EmptyGraph**：用于连接远程数据库实例。
  janusGraph可直接在配置文件中读取，不需要该方式。
  ```
  graph = EmptyGraph.instance()； //单例模式
  g = graph.traversal().withRemote('conf/remote-graph.properties')
  ```
- **StarGraph（Serializable）**：对应GraphSON的一行记录，仅表示一个节点以及它相关的边和属性。
  用于网络和磁盘序列化操作（导入、导出数据）。
- **TinkerGraph**：是TinkerPop3的参考实现，和janusGraph中的SandardJanusGraph地位一致。
- **HadoopGraph**：用于Hadoop、spark相关计算分析。
  1. 不支持节点、边和属性的增加和删除，事务，图相关参数的操作。
  2. 支持SparkGraphComputer和GiraphGraphComputer两种计算引擎。
  3. 支持三大类输入输出格式(Input/Output Formats)：
     Gryo、GraphSON、Script （org.apache.tinkerpop.gremlin.hadoop.structure.io包）
- **ComputerGraph**：tinkerpop官网无介绍，源码简介主要是针对MapReduce的图计算。

3.1.2 JanusGraph模式

常用配置项地址：https://docs.janusgraph.org/latest/config-ref.html

基本配置，执行 `vim  conf/test.properties` ：

```
gremlin.graph=org.janusgraph.core.JanusGraphFactory
storage.hbase.table = show
storage.backend=hbase
storage.hostname=192.168.84.161
cache.db-cache = true
cache.db-cache-clean-wait = 20
cache.db-cache-time = 180000
cache.db-cache-size = 0.5
storage.batch-loading=true

#es配置
index.search.backend=elasticsearch
index.search.hostname=192.168.84.161
index.search.client-only=true
```

开图语句，执行 `./bin/gremlin.sh` ：

```
graph = JanusGraphFactory.open('conf/test.properties')
g = graph.traversal();
```

3.1.3 HadoopGraph模式

常用配置项地址：http://tinkerpop.apache.org/docs/3.3.4/reference/#_properties_files

基本配置，执行 `vim  conf/hadoop-graph/read-hbase.properties` ：

```
gremlin.graph=org.apache.tinkerpop.gremlin.hadoop.structure.HadoopGraph
gremlin.hadoop.graphReader=org.janusgraph.hadoop.formats.hbase.HBaseInputFo
rmat
gremlin.hadoop.graphWriter=org.apache.tinkerpop.gremlin.hadoop.structure.io
.gryo.GryoOutputFormat
gremlin.hadoop.jarsInDistributedCache=true
gremlin.hadoop.inputLocation=none
gremlin.hadoop.outputLocation=output

# JanusGraph HBase InputFormat configuration
janusgraphmr.ioformat.conf.storage.backend=hbase
janusgraphmr.ioformat.conf.storage.hostname=localhost
janusgraphmr.ioformat.conf.storage.hbase.table=janusgraph
```

```
# SparkGraphComputer Configuration
spark.master=local[4]
spark.serializer=org.apache.spark.serializer.KryoSerializer
```

开图语句，执行 `./bin/gremlin.sh` ：

```
graph = GraphFactoy.open('conf/hadoop-graph/read-hbase.properties');
g = graph.traversal().withComputer(SparkGraphComputer.class);
```

3.2 建模

3.2.1 建模语句

边的label和属性的key不能有相同的命名，因为他们同属于relation types。

```
mgmt = graph.openManagement()

## 定义点和边的label:
mgmt.makeVertexLabel("person").make();
mgmt.makeEdgeLabel("knows").multiplicity(Multiplicity.SIMPLE).make();

## 定义属性:
mgmt.makePropertyKey("name").dataType(String.class).cardinality(Cardinality
.SINGLE).make()
mgmt.makePropertyKey("times").dataType(String.class).cardinality(Cardinalit
y.SINGLE).make()

## 定义索引:
property = mgmt.getPropertyKey("name");
mgmt.buildIndex("nameCompositeIndex",
Vertex.class).addKey(property).buildCompositeIndex(); #基于Hbase索引
mgmt.buildIndex(“nameMixedIndex”,Vertex.class).addKey(property).buildMixedI
ndex("search"); #基于ES的索引

##节点中心索引
eLabel = mgmt.getEdgeLabel(“knows”);
propertyVertexCentricIndexSortKeys =  mgmt.getPropertyKey(“times”);
mgmt.buildEdgeIndex(eLabel, “knowsVertexCentricIndex”, Direction.BOTH,
Order.decr,propertyVertexCentricIndexSortKeys);
```

```
mgmt.commit()
```

3.2.2 建模基本参数

3.2.2.1 Multiplicity Settings

- MULTI：　　一对节点间相同label的边可以有多条。
- SIMPLE：　　一对节点间相同label的边只能有一条。
- MANY2ONE：任意节点间相同label的边只允许一条出边，任意多条入边（每人一个母亲，一个母亲可以有多个孩子）。
- ONE2MANY：任意节点间相同label的边只允许一条入边，任意多条入边（一个比赛一个冠军，一人可多个冠军）。
- ONE2ONE：　任意节点间相同label的边只允许一条入边，一条入边（两人结婚married to）

3.2.2.2 数据类型

dataType：

String、Character、Boolean、Byte、Short、Integer、Long、Float、Double、
Date（java.util.Date）、Geoshape、UUID(java.util.UUID)

3.2.2.3 Cardinality Settings

SINGLE、LIST、SET。　（一个节点或边的一种属性可以有多少个值）

3.2.2.4 定义索引：

- **Composite Index**：基于Hbase索引的精确查询。（唯一限制unique()目前对java api插入无效）
- **Mixed Index**：基于ES的模糊、比较和地理位置等查询。
- **Vertex-centric Indexes**：加速点的关系查询速度，缓解超级节点的查询问题。（目前作用尚未测试出来，更新索引失败。）

3.2.2.5 其他功能

- **更新名称：JanusGraphSchemaElement为空接口**
  JanusGraphManagement.changeName(JanusGraphSchemaElement, String)；
- **shcema Constraints：显示绑定属性到指定label的点和边上。显示指定边和起始点的label**
  addProperties(VertexLabel, PropertyKey...);
  addProperties(EdgeLabel, PropertyKey...);
  addConnection(EdgeLabel, VertexLabel out, VertexLabel in);　　（pserson -[works]-> company）
- **Static Vertices：点在创建后不能更改**
  mgmt.makeVertexLabel('tweet').setStatic().make()
- **Edge and Vertex TTL:设置点、边和属性的存活时间**
  visits = mgmt.makeEdgeLabel('visits').make()

mgmt.setTTL(visits, Duration.ofDays(7))

- **Unidirected Edges：单一方向边，只能从出度遍历，起点删除时边不会自动删除。**
  mgmt.makeEdgeLabel('author').unidirected().make()
- **重新索引和删除索引的两种方式：JanusGraphManagement、MapReduce**
  详见：https://docs.janusgraph.org/latest/index-admin.html

3.2.2 建模结果查询

```
mgmt = graph.openManagement()

## 查询所有实体label:
mgmt.getVertexLabels();

## 查询所有关系或属性:

mgmt.getRelationTypes(RelationType.class); //关系和属性
mgmt.getRelationTypes(EdgeLabel.class); //关系
mgmt.getRelationTypes(PropertyKey.class); //属性

##查询所有索引信息:
mgmt.getGraphIndexes(Vertex.class); //实体
mgmt.getGraphIndexes(Edge.class); //关系
```

3.2.3 建模信息删除

```
// mgmt.containsVertexLabel("person")
vertexLabel = mgmt.getVertexLabel("person")
vertexLabel.remove()

// mgmt.containsPropertyKey("name")
propertyKey = mgmt.getPropertyKey("name");
propertyKey.remove()

//mgmt.containsEdgeLabel("knows")
edgeLabel = mgmt.getEdgeLabel("knows");
edgeLabel.remove();
```

3.3 数据导入

3.3.1 新增点

```
graph.addVertex(label, 'person','name','x')     // janusGraph不支持自定义id。
```

3.3.2 新增边

```
a = graph.addVertex(label,'person','name','a')
b = graph.addVertex(label,'person','name','b')
a.addEdge('knows', b, 'times','100')
graph.tx().commit()
```

3.3.3 删除数据

```
g = graph.traversal()
g.V('4184').drop()        //4184、2rv-6e0-2dx-39k 为janus自动生成的唯一id。
g.E('2rv-6e0-2dx-39k').drop()
graph.tx().commit()
```

3.4 遍历查询

Gremlin 语言包括三个基本的操作：

- map-step：对数据流中的对象进行转换；
- filter-step：对数据流中的对象就行过滤；
- sideEffect-step：对数据流进行计算统计；

采用TinkerGraph的样例图：

```
graph = TinkerFactory.createModern()
g = graph.traversal()
```

3.4.1 基础

1. `V()`：查询顶点，一般作为图查询的第1步。例，g.V()，g.V('v_id')，查询所有点和特定点；
2. `E()`：查询边，一般作为图查询的第1步；
3. `id()`：获取节点、边的id。例：g.V().id()，查询所有节点的id；
4. `valueMap()`：获取节点、边的属性，以Map的形式体现，`valueMap(true)`会返回数据中会包含节点的label和id；

3.4.2 遍历

3.4.2.1 顶点为基准

1. `out(label)`：根据指定的 Edge Label 来访问顶点的 OUT 方向邻接**点**（可以是零个 Edge Label，代表所有类型边；也可以一个或多个 Edge Label，代表任意给定 Edge Label 的边，下同）；
2. `in(label)`：根据指定的 Edge Label 来访问顶点的 IN 方向邻接**点**；
3. `both(label)`：根据指定的 Edge Label 来访问顶点的双向邻接**点**；
4. `outE(label)`： 根据指定的 Edge Label 来访问顶点的 OUT 方向邻接**边**；
5. `inE(label)`：根据指定的 Edge Label 来访问顶点的 IN 方向邻接**边**；
6. `bothE(label)`：根据指定的 Edge Label 来访问顶点的双向邻接**边**；

3.4.2.2 边为基准

1. `outV()`：访问边的出顶**点**，出顶点是指边的起始顶点；
2. `inV()`：访问边的入顶**点**，入顶点是指边的目标顶点，也就是箭头指向的顶点；
3. `bothV()`：访问边的双向顶**点**；
4. `otherV()`：访问边的伙伴顶**点**，即相对于基准顶点而言的另一端的顶点；

3.4.3 过滤

1. `has(key,value)`：通过属性的名字和值来过滤顶点或边；
2. `has(key,predicate)`：通过对指定属性用条件过滤顶点和边，例：`g.V().has('age', gt(20))`，可得到年龄大于20的顶点；
3. `hasLabel(labels…)`：通过 label 来过滤顶点或边，满足label列表中一个即可通过；
4. `hasId(ids…)`：通过 id 来过滤顶点或者边，满足id列表中的一个即可通过；
5. `hasNot(key)`：和 has(key) 相反；

3.4.4 路径

1. `path()`：获取当前遍历过的所有路径；
2. `simplePath()`：过滤掉路径中含有环路的对象，只保留路径中不含有环路的对象；
3. `cyclicPath()`：过滤掉路径中不含有环路的对象，只保留路径中含有环路的对象。

3.4.5 迭代

1. `repeat()`：指定要重复执行的语句；
2. `times()`： 指定要重复执行的次数，如执行3次；
3. `until()`：指定循环终止的条件，如一直找到某个名字的朋友为止；
4. `emit()`：指定循环语句的执行过程中收集数据的条件，每一步的结果只要符合条件则被收集，不指定条件时收集所有结果；
5. `loops()`：当前循环的次数，可用于控制最大循环次数等，如最多执行3次。

`repeat()` 和 `until()` 的位置不同，决定了不同的循环效果：

- `repeat()` + `until()`：等同 do-while；
- `until()` + `repeat()`：等同 while-do。

`repeat()` 和 `emit()` 的位置不同，决定了不同的循环效果：

- `repeat()` + `emit()`：先执行后收集；
- `emit()` + `repeat()`：表示先收集再执行。

3.4.6 分支

- `coalesce()` 接受任意数量的遍历器，按顺序执行，返回第一个能产生输出的结果**

```
g.V('8360').coalesce(outE('father'),outE('mother')).otherV().path().by('nam
e').by()
```

- `union()` 接受任意数量的遍历器，将所有遍历器结果合并

```
g.V('8360').union(outE('father'),outE('mother')).otherV().path().by('name')
.by()
```

- `optional()` 接受遍历器，当前遍历器无返回值时，返回上一步及之前的遍历器的结果。

```
g.V('8360').optional(outE('father111')).path()
```

3.4.7 逻辑

1. `is()`：可以接受一个对象（能判断相等）或一个判断语句（如：P.gt()、P.lt()、P.inside()等），当接受的是对象时，原遍历器中的元素**必须与对象相等**才会保留；当接受的是判断语句时，原遍历器中的元素满足判断才会保留，其实接受一个对象相当于P.eq()；
2. `and()`：可以接受任意数量的遍历器（traversal），原遍历器中的元素，只有在每个新遍历器中**都能生成至少一个输出的情况下**才会保留，相当于过滤器组合的与条件；
3. `or()`：可以接受任意数量的遍历器（traversal），原遍历器中的元素，只要在全部新遍历器中**能生成至少一个输出**的情况下就会保留，相当于过滤器组合的或条件；
4. `not()`：仅能接受一个遍历器（traversal），原遍历器中的元素，在新遍历器中能生成输出时会被移除，不能生成输出时则会保留，相当于**过滤器的非条件**。

3.4.8 终止步骤

1. hasNext()
2. next()
3. next(n)
4. tryNext():返回Optional类
5. toList()
6. toSet()：删除重复

7. toBulkSet()：将所有的元素放到一个能排序的List中返回，重复元素也会保留
8. fill(collection):将结果返回到指定集合
9. iterate()
10. explain()

Gremlin控制台会自动遍历返回的Iterator，java语言需要添加终止步骤。

```
gremlin > g.V(1).outE().group().by(label).by(inV().fold())

java:
public static Map<String,List<Vertex>> groupAround(GraphTraversalSource g,
long vertexId) {
     return (Map<String,List<Vertex>>(Map)
g.V(vertexId).outE().group().by(label).by(inV().fold()).next()
}

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

java:
public static void removeByName(GraphTraversalSource g, String name) {

         g.V().has("name", name).drop().iterate();
}
```

4、基于项目的gremlin设计和优化

4.1 主要思路

- 完成产品功能设计。主要分为三种查询结果：点，边，路径。
- 对于全图的即时查询，要求基于索引查询；内存子图无该要求。

在gremlin shell中查询不基于索引会提示以下信息：

```
WARN  org.janusgraph.graphdb.transaction.StandardJanusGraphTx  - Query
requires iterating over all vertices [()]. For better performance, use
indexes
```

- 合并查询，减少与gremlin-server交互次数。([a.toList，b.toList],union查询)
- 分批查询，groovy语言的可变参数255限制。一类实体发送一条请求，单类实体数量超过阈值后进行分批查询。

- 延迟查询，先获取节点和关系id，再获取属性。（因图面涉及到节点和关系的显示属性，不适合分步请求，故目前无法较好优化）
- 基于WebSocket建立持久连接，减少http交互次数,不存在groovy语言255参数限制。
- 兼容neo4j模式，查询点的语句后面追加valueMap(true)。

4.2 选择实体

4.2.1 精确查找：

根据节点类型和主键值查询节点

```
g.V().hasLabel('人员').has('人员_身份证
号',within('5001201964030615 34','5001171959042293 04')).valueMap(true)
```

注意：

- 对于输入的数据类型进行校验过滤（数值类型的合法性）；

4.2.2 条件查找：

指定条件（包括节点label、属性，节点邻接边的label、方向和属性）

```
g.V().has('人员_身份证号',textPrefix('5001')).hasLabel('人
员').path().map{it.get().get(0)}.dedup().valueMap(true)

g.V().and(has('人员_身份证号',textPrefix('5001')),has('人员_性别'
,'男')).hasLabel('人员').union(bothE('父子'),bothE('夫妻'),bothE('同
事'),bothE('姐弟'),inE('人-号码'),bothE('同住').has('同住_次
数',inside(1,5))).path().map{it.get().get(0)}.dedup().valueMap(true)

g.E().has('通话_通话次数' ,1).hasLabel('通话').bothV().hasLabel('号
码').dedup().valueMap(true)
```

注意：

- 过滤条件中没有边的条件，是可以筛选出孤立节点的，不能追加bothE()。
- 前两条的选取节点get(0),不能使用as、select替代map操作。因为在测试过程中g.V()后面接and或者or都会导致as丢失，无法查出结果。
- 使用EmptyGraph远程连接图时，map里面的lambda表达式不能使用，会出现以下报错，需要拆分成两步操作，先查path，再查点。

```
Caused by: java.lang.IllegalArgumentException: Class is not registered:
com.iflytek.janus.RemoteGraph02$$Lambda$39/45019084
```

4.2.3 文件导入：

和精确查找共用一个接口。

4.3 扩展关系

选中图面上的节点，进行关系扩展。可选条件：**扩展的度数，扩展关系的label、方向和属性过滤，目标节点的label和属性过滤**。

```
// 扩展度数为2时的gremlin语句（两条）

g.V().hasId(within('4096')).union(bothE('姐弟'),bothE('父女'),bothE('结
婚').has('结婚_结婚日期',lt(new SimpleDateFormat('yyyy-MM-dd
HH:mm:ss').parse('2019-08-14 12:00:00')))).otherV().hasLabel('人
员').simplePath().path()

g.V().hasId(within('4096')).union(bothE('姐弟'),bothE('父女'),bothE('结
婚').has('结婚_结婚日期',lt(new SimpleDateFormat('yyyy-MM-dd
HH:mm:ss').parse('2019-08-14 12:00:00')))).otherV().hasLabel('人
员').simplePath().union(bothE('姐弟'),bothE('父女'),bothE('结婚').has('结婚_结
婚日期',lt(new SimpleDateFormat('yyyy-MM-dd HH:mm:ss').parse('2019-08-14
12:00:00')))).otherV().hasLabel('人员').simplePath().path()
```

注意：

- 多度扩展时（度数为n），会把n以内的所有满足条件的路径返回。
- 扩展度数大于1时，可选择的关系label必须满足两端节点类型相同。
- 起始节点的id值4096为JanusGraph生成的唯一id，在选择实体阶段已经获取。
- 对于日期类型的比较要引入SimpleDateFormat类（gremlin server的脚本配置中添加）。

```
scriptEngines: {
  gremlin-groovy: {
    plugins: {
org.janusgraph.graphdb.tinkerpop.plugin.JanusGraphGremlinPlugin: {},

org.apache.tinkerpop.gremlin.server.jsr223.GremlinServerGremlinPlugin: {},

org.apache.tinkerpop.gremlin.tinkergraph.jsr223.TinkerGraphGremlinPlugin:
{},
               org.apache.tinkerpop.gremlin.jsr223.ImportGremlinPlugin:
{classImports: [java.lang.Math,java.text.SimpleDateFormat], methodImports:
[java.lang.Math#*]},
               org.apache.tinkerpop.gremlin.jsr223.ScriptFileGremlinPlugin:
{files: [scripts/empty-sample.groovy]}}}}
```

4.4 关系分析

4.4.1 路径分析

4.4.1.1 最短路径

选取两个节点，设置最短路径的度数，返回两点之间的最短路径。

```
g.V('16536').repeat(bothE().otherV().simplePath()).until(hasId('188656').or
().loops().is(gte(3))).hasId('188656').path().limit(1)
```

4.4.1.2 路径分析

选取两个点，设置路径度数n，起点，关系label、方向和属性过滤，路径实体的label和属性过滤，返回两点之间n度以内所有满足条件的路径。

```
// 路径度数n=3时的gremlin语句（三条）：

g.V('16536').union(bothE('父女'),bothE('结婚'),bothE('离
婚')).otherV().hasId('188656').simplePath().path()

g.V('16536').union(bothE('父女'),bothE('结婚'),bothE('离
婚')).otherV().hasLabel('人员').union(bothE('父女'),bothE('结婚'),bothE('离
婚')).otherV().hasId('188656').simplePath().path()

g.V('16536').union(bothE('父女'),bothE('结婚'),bothE('离
婚')).otherV().hasLabel('人员').union(bothE('父女'),bothE('结婚'),bothE('离
婚')).otherV().hasLabel('人员').union(bothE('父女'),bothE('结婚'),bothE('离
婚')).otherV().hasId('188656').simplePath().path()
```

注意：

- 可选择的关系label为全库所有的关系类型，但是对关系的可选方向进行了修正：

  1、该关系的是有向的,a、两端实体一致，则出、入、无向都可选。b、两端不一致，则只可选无向。

  2、该关系是无向的,则只能选无向的。
- 可选择的路径实体，是根据**选择的关系**推测出来的，直接返回关系两端的实体。而**扩展关系**的目标实体是根据**起始实体和关系类型**的方向推测出来的。例如”人-号码“关系，起始实体为人，在扩展关系中目标实体为号码，在路径分析中可选路径实体为人员和号码。

4.4.2 来往分析

即为多个点之间的路径分析。

```
// 度数n=3的来往分析gremlin语句（三条）：
g.V().hasId(within('24816','8192')).union(bothE('人-号码'),bothE('人-账
号'),bothE('父女'),bothE('兄
弟')).otherV().hasId(within('204848')).simplePath().path()

g.V().hasId(within('24816','8192')).union(bothE('人-号码'),bothE('人-账
号'),bothE('父女'),bothE('兄
弟')).otherV().hasId(without('24816','8192','204848')).union(bothE('人-号
码'),bothE('人-账号'),bothE('父女'),bothE('兄
弟')).otherV().hasId(within('204848')).simplePath().path()

g.V().hasId(within('24816','8192')).union(bothE('人-号码'),bothE('人-账
号'),bothE('父女'),bothE('兄
弟')).otherV().hasId(without('24816','8192','204848')).union(bothE('人-号
码'),bothE('人-账号'),bothE('父女'),bothE('兄
弟')).otherV().hasId(without('24816','8192','204848')).union(bothE('人-号
码'),bothE('人-账号'),bothE('父女'),bothE('兄
弟')).otherV().hasId(within('204848')).simplePath().path()
```

4.4.3 共同邻居

多个点之间一度关系能连接的节点。

```
// 每一类点发一条gremlin:
g.V().hasId(within('204848')).inE('人-号码').otherV().hasLabel('人
员').inE('人-号
码').otherV().hasId(within('204848','24816','8192')).simplePath().path()

g.V().hasId(within('24816','8192')).union(bothE('父女'),bothE('兄
弟')).otherV().hasLabel('人员').union(bothE('父女'),bothE('兄
弟')).otherV().hasId(within('204848','24816','8192')).simplePath().path()
```

注意：

- 共同邻居分为本页数据和全库数据两种。本页数据的可选关系类型从当前页面中选择，全库数据的可选关系类型和**扩展关系**的推测逻辑相同。

4.4.4 群集分析

基于全库数据，查询多个节点之间的直连关系。返回满足条件的边和所有的初始点。可选关系为：所有节点类型可选关系的并集。不涉及关系的方向选择。

```
// 每一类节点发一条gremlin:

g.V().hasId('4096','24816','8192').union(bothE('人-号码'),bothE('父
女'),bothE('兄
弟')).as('A').otherV().hasId(within('4096','24816','8192','204848')).select
('A').dedup()

g.V().hasId('204848').bothE('人-号
码').as('A').otherV().hasId(within('4096','24816','8192','204848')).select(
'A').dedup()
```

4.5 图面操作

4.5.1一键扩展

选中单个节点，选择指定关系，进行一度关系扩展。

```
g.V().hasId(within('4096')).bothE().hasLabel('父子','夫妻','同事','姐
弟').otherV().path()
```

4.5.2 筛选功能

统计属性过滤

```
g.V().hasLabel('号码').bothE().hasLabel('通话关系').otherV().hasLabel('号
码').path().group().by(map{
def p = it.get();
p.get(0).id().toString() + "_" + p.get(2).id().toString()
})
.by(map{it.get().get(1)}.values('通话关系_通话次数').sum())
```

4.5.3 叶子节点统计

返回图面上所有的叶子节点，返回一个map，取值为1的不考虑孤立点。

```
g.V().sideEffect(bothE().otherV().dedup().path().store('a')).cap('a').unfol
d().groupCount().by(map{it.get().get(0)})
```

5、参考资料

号码：15152044030

人员：140105197409109176,140203199804019826

6、附录

图数据库基本概念

图数据库存储

janusGraph基本架构

gremlin语法

gremlin server服务

TinkerGraph内存图

gremlin字符串解析

gremlin的分批请求

请求结果解析，tinkerpop规范3转1（点、边）

基于hadoop的OLAP操作（序列化）

属性过滤条件的或且拼接

基于websocket的gremlin server服务和请求（lambda表达式序列化问题未解决。）

用户实体关系权限。（通过请求建模信息阶段控制用户可见实体）

kryo序列化问题。

图算法相关应用和家族图谱生成。