Chapter 2. Architectural Overview

JanusGraph is a graph database engine. JanusGraph itself is focused on compact graph serialization, rich graph data modeling, and efficient query execution. In addition, JanusGraph utilizes Hadoop for graph analytics and batch graph processing. JanusGraph implements robust, modular interfaces for data persistence, data indexing, and client access. JanusGraph’s modular architecture allows it to interoperate with a wide range of storage, index, and client technologies; it also eases the process of extending JanusGraph to support new ones.

Between JanusGraph and the disks sits one or more storage and indexing adapters. JanusGraph comes standard with the following adapters, but JanusGraph’s modular architecture supports third-party adapters.

* Data storage:
  + [Apache Cassandra](http://docs.janusgraph.org/latest/cassandra.html)
  + [Apache HBase](http://docs.janusgraph.org/latest/hbase.html)
  + [Oracle Berkeley DB Java Edition](http://docs.janusgraph.org/latest/bdb.html)
* Indices, which speed up and enable more complex queries:
  + [Elasticsearch](http://docs.janusgraph.org/latest/elasticsearch.html)
  + [Apache Solr](http://docs.janusgraph.org/latest/solr.html)
  + [Apache Lucene](http://docs.janusgraph.org/latest/lucene.html)

Broadly speaking, applications can interact with JanusGraph in two ways:

* Embed JanusGraph inside the application executing [Gremlin](http://tinkerpop.apache.org/docs/3.2.6/reference#graph-traversal-steps) queries directly against the graph within the same JVM. Query execution, JanusGraph’s caches, and transaction handling all happen in the same JVM as the application while data retrieval from the storage backend may be local or remote.
* Interact with a local or remote JanusGraph instance by submitting Gremlin queries to the server. JanusGraph natively supports the Gremlin Server component of the [Apache TinkerPop](http://tinkerpop.apache.org/) stack.

Chapter 3. Getting Started

**Table of Contents**

[3.1. Downloading JanusGraph and Running the Gremlin Console](http://docs.janusgraph.org/latest/getting-started.html#_downloading_janusgraph_and_running_the_gremlin_console)

[3.2. Loading the Graph of the Gods Into JanusGraph](http://docs.janusgraph.org/latest/getting-started.html#_loading_the_graph_of_the_gods_into_janusgraph)

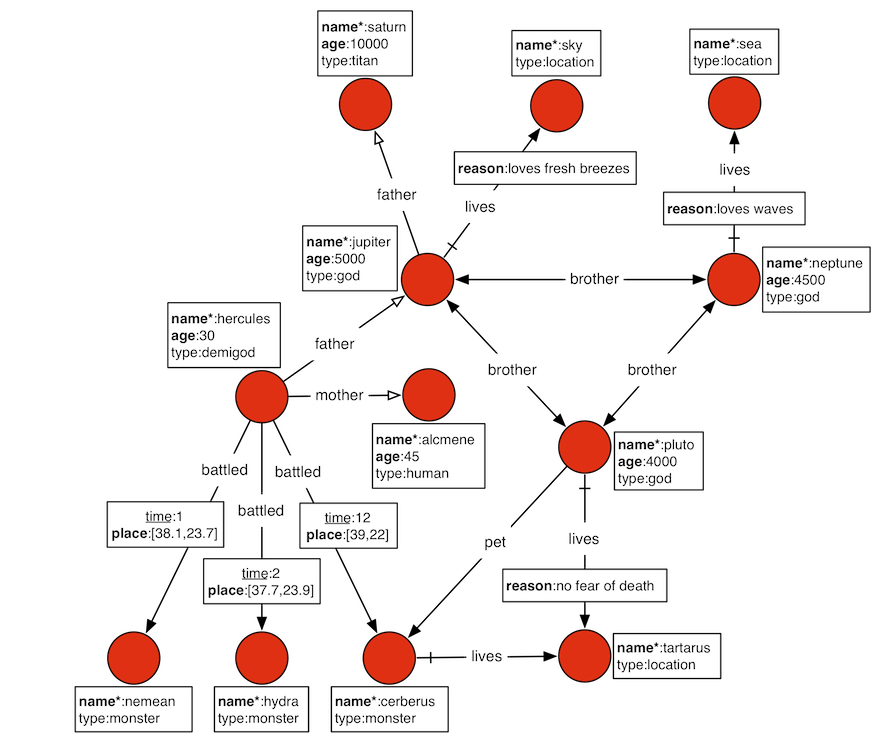
[3.3. Global Graph Indices](http://docs.janusgraph.org/latest/getting-started.html#_global_graph_indices)

[3.3.1. Graph Traversal Examples](http://docs.janusgraph.org/latest/getting-started.html#_graph_traversal_examples)

[3.3.2. More Complex Graph Traversal Examples](http://docs.janusgraph.org/latest/getting-started.html#_more_complex_graph_traversal_examples)

The examples in this section make extensive use of a toy graph distributed with JanusGraph called The Graph of the Gods. This graph is diagrammed below. The abstract data model is known as a [Property Graph Model](http://tinkerpop.apache.org/docs/3.2.6/reference#intro) and this particular instance describes the relationships between the beings and places of the Roman pantheon. Moreover, special text and symbol modifiers in the diagram (e.g. bold, underline, etc.) denote different schematics/typings in the graph.

本节的例子充分利用了一个叫做The Graph of the Gods的来作为JanusGraph的分布式图数据集, 这个抽象数据模型被称为Property Graph Model, 这个实例描述了人类与罗马众神之间的关系, 而且这个序列图中的文本文字以及符号描述了图中的类型.

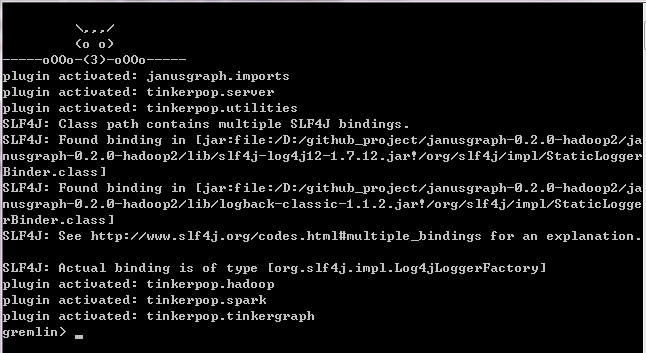


3.1. Downloading JanusGraph and Running the Gremlin Console

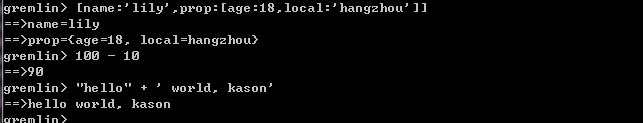
|  |  |
| --- | --- |
| [Important] | **Important** |
| JanusGraph requires Java 8 (Standard Edition). Oracle Java 8 is recommended. JanusGraph’s shell scripts expect that the $JAVA\_HOME environment variable points to the directory where JRE or JDK is installed. |

注意JanusGraph需要Java8的运行环境.

首先通过url[janusGraph 下载](https://github.com/JanusGraph/janusgraph/releases)下载[**janusgraph-0.2.0-hadoop2.zip**](https://github.com/JanusGraph/janusgraph/releases/download/v0.2.0/janusgraph-0.2.0-hadoop2.zip)然后解压之, window下执行bin/ gremlin-server.bat, linux下执行bin/ gremlin.sh来启动Gremlin Console



Gremlin Console 使用Apache Groovy 来进行解析命令. Groovy是Java的超集,它有很多简化符号来让交互变的更加容易, 同样地Gremlin-Groovy是Groovy的超集,它可以使得遍历图非常容易.下面的例子证明了处理数字, 字符串, 以及映射map的方式, 后面也会讨论指定图的构造方式.



|  |  |
| --- | --- |
| [Tip] | **Tip** |
| Refer to [Apache TinkerPop](http://tinkerpop.apache.org/docs/3.2.6/reference), [SQL2Gremlin](http://sql2gremlin.com/), and [Gremlin Recipes](http://tinkerpop.apache.org/docs/3.2.6/recipes/) for more information about using Gremlin. |

3.2. Loading the Graph of the Gods Into JanusGraph

官网有很多例子, 通过导入conf下的配置文件来创建graph实例, 这里仅仅使用几个简单的方式,inmemory模式.

graph = JanusGraphFactory.open('inmemory')

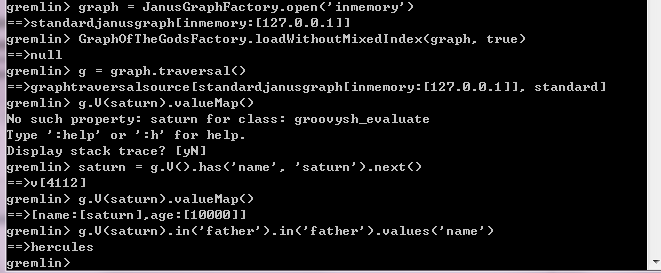
GraphOfTheGodsFactory.loadWithoutMixedIndex(graph, true)

g = graph.traversal()

saturn = g.V().has('name', 'saturn').next()

g.V(saturn).valueMap()

g.V(saturn).in('father').in('father').values('name')



3.3. Global Graph Indices

The typical pattern for accessing data in a graph database is to first locate the entry point into the graph using a graph index. That entry point is an element (or set of elements) — i.e. a vertex or edge. From the entry elements, a Gremlin path description describes how to traverse to other elements in the graph via the explicit graph structure.

典型的接触图数据库数据的模式是首先先根据图的索引定位到入口点, 入口点是一个元素或者元素集合(如边和顶点), 根据这个入口元素, Gremlin可以通过明确的图结构去遍历其他图中的元素.

Given that[假定] there is a unique index on name property, the Saturn vertex can be retrieved. The property map (i.e. the key/value pairs of Saturn) can then be examined. **As demonstrated[作为证明]**, the Saturn vertex has a name of "saturn, " an age of 10000, and a type of "titan." The grandchild of Saturn can be retrieved with a traversal that expresses: "Who is Saturn’s grandchild?" (the inverse of "father" is "child"). The result is Hercules.

假定有一个name属性作为唯一的索引, 则Saturn 顶点可以被获取到.然后Saturn的键值对 map属性结果就可以得到.

**gremlin>** saturn = g.V().has('name', 'saturn').next()

**==>**v[256]

**gremlin>** g.V(saturn).valueMap()

**==>**[name:[saturn], age:[10000]]

**gremlin>** g.V(saturn).in('father').in('father').values('name')

**==>**hercules

The property place is also in a graph index. The property place is an edge property. Therefore, JanusGraph can index edges in a graph index. It is possible to query The Graph of the Gods for all events that have happened within 50 kilometers of [Athens](http://en.wikipedia.org/wiki/Athens) (latitude:37.97 and long:23.72). Then, given that information, which vertices were involved in those events.

Place属性也在图的索引中, 这个可以通过图来看到, place属性是一个边属性.因此, JanusGraph可以在图索引中索引边属性.比如可以从诸神图中查询发生在Athens 50km内(纬度37.97,经度:23.72)内的事件. 然后根据上面的信息 可以获取哪些顶点被包含在那些事件中.

**gremlin>** g.E().has('place', geoWithin(Geoshape.circle(37.97, 23.72, 50)))

**==>**e[a9x-co8-9hx-39s][16424-battled->4240]

**==>**e[9vp-co8-9hx-9ns][16424-battled->12520]

**gremlin>** g.E().has('place', geoWithin(Geoshape.circle(37.97, 23.72, 50))).as('source').inV().as('god2').select('source').outV().as('god1').select('god1', 'god2').by('name')

**==>**[god1:hercules, god2:hydra]

**==>**[god1:hercules, god2:nemean]

Graph indices are one type of index structure in JanusGraph. Graph indices are automatically chosen by JanusGraph to answer which ask for all vertices (g.V) or all edges (g.E) that satisfy one or multiple constraints (e.g. has or interval). The second aspect of indexing in JanusGraph is known as vertex-centric indices. Vertex-centric indices are utilized to speed up traversals inside the graph. Vertex-centric indices are described later.

### 3.3.1. Graph Traversal Examples

[Hercules](http://en.wikipedia.org/wiki/Hercules), son of Jupiter and [Alcmene](http://en.wikipedia.org/wiki/Alcmene), bore super human strength. Hercules was a [Demigod](http://en.wikipedia.org/wiki/Demigod) because his father was a god and his mother was a human. [Juno](http://en.wikipedia.org/wiki/Juno_(mythology)), wife of Jupiter, was furious with Jupiter’s infidelity. In revenge, she blinded Hercules with temporary insanity and caused him to kill his wife and children. To atone for the slaying, Hercules was ordered by the [Oracle of Delphi](http://en.wikipedia.org/wiki/Oracle_at_Delphi) to serve [Eurystheus](http://en.wikipedia.org/wiki/Eurystheus" \t "_top). Eurystheus appointed Hercules to 12 labors.

In the previous section, it was demonstrated that Saturn’s grandchild was Hercules. This can be expressed using a loop. In essence, Hercules is the vertex that is 2-steps away from Saturn along the in('father') path.

**gremlin>** hercules = g.V(saturn).repeat(\_\_.in('father')).times(2).next()

**==>**v[1536]

Hercules is a demigod. To prove that Hercules is half human and half god, his parent’s origins must be examined. It is possible to traverse from the Hercules vertex to his mother and father. Finally, it is possible to determine the type of each of them — yielding "god" and "human."

**gremlin>** g.V(hercules).out('father', 'mother')

**==>**v[1024]

**==>**v[1792]

**gremlin>** g.V(hercules).out('father', 'mother').values('name')

**==>**jupiter

**==>**alcmene

**gremlin>** g.V(hercules).out('father', 'mother').label()

**==>**god

**==>**human

**gremlin>** hercules.label()

**==>**demigod

The examples thus far have been with respect to the genetic lines of the various actors in the Roman pantheon. The [Property Graph Model](http://tinkerpop.apache.org/docs/3.2.6/reference#intro) is expressive enough to represent multiple types of things and relationships. In this way, The Graph of the Gods also identifies Hercules' various heroic exploits --- his famous 12 labors. In the previous section, it was discovered that Hercules was involved in two battles near Athens. It is possible to explore these events by traversing battled edges out of the Hercules vertex.

**gremlin>** g.V(hercules).out('battled')

**==>**v[2304]

**==>**v[2560]

**==>**v[2816]

**gremlin>** g.V(hercules).out('battled').valueMap()

**==>**[name:[nemean]]

**==>**[name:[hydra]]

**==>**[name:[cerberus]]

**gremlin>** g.V(hercules).outE('battled').has('time', gt(1)).inV().values('name')

**==>**cerberus

**==>**hydra

The edge property time on battled edges is indexed by the vertex-centric indices of a vertex. Retrieving battled edges incident to Hercules according to a constraint/filter on time is faster than doing a linear scan of all edges and filtering (typically O(log n), where n is the number incident edges). JanusGraph is intelligent enough to use vertex-centric indices when available. A toString() of a Gremlin expression shows a decomposition into individual steps.

**gremlin>** g.V(hercules).outE('battled').has('time', gt(1)).inV().values('name').toString()

**==>**[GraphStep([v[24744]],vertex), VertexStep(OUT,[battled],edge), HasStep([time.gt(1)]), EdgeVertexStep(IN), PropertiesStep([name],value)]

3.3.2. More Complex Graph Traversal Examples

In the depths of Tartarus lives Pluto. His relationship with Hercules was strained by the fact that Hercules battled his pet, Cerberus. However, Hercules is his nephew — how should he make Hercules pay for his insolence?

The Gremlin traversals below provide more examples over The Graph of the Gods. The explanation of each traversal is provided in the prior line as a // comment.

#### 3.3.2.1. Cohabiters of Tartarus

**gremlin>** pluto = g.V().has('name', 'pluto').next()

**==>**v[2048]

**gremlin>** // who are pluto's cohabitants?

**gremlin>** g.V(pluto).out('lives').in('lives').values('name')

**==>**pluto

**==>**cerberus

**gremlin>** // pluto can't be his own cohabitant

**gremlin>** g.V(pluto).out('lives').in('lives').where(is(neq(pluto))).values('name')

**==>**cerberus

**gremlin>** g.V(pluto).as('x').out('lives').in('lives').where(neq('x')).values('name')

**==>**cerberus

#### 3.3.2.2. Pluto’s Brothers

**gremlin>** // where do pluto's brothers live?

**gremlin>** g.V(pluto).out('brother').out('lives').values('name')

**==>**sky

**==>**sea

**gremlin>** // which brother lives in which place?

**gremlin>** g.V(pluto).out('brother').as('god').out('lives').as('place').select('god', 'place')

**==>**[god:v[1024], place:v[512]]

**==>**[god:v[1280], place:v[768]]

**gremlin>** // what is the name of the brother and the name of the place?

**gremlin>** g.V(pluto).out('brother').as('god').out('lives').as('place').select('god', 'place').by('name')

**==>**[god:jupiter, place:sky]

**==>**[god:neptune, place:sea]

Finally, Pluto lives in Tartarus because he shows no concern for death. His brothers, on the other hand, chose their locations based upon their love for certain qualities of those locations.

**gremlin>** g.V(pluto).outE('lives').values('reason')

**==>**no fear of death

**gremlin>** g.E().has('reason', textContains('loves'))

**==>**e[6xs-sg-m51-e8][1024-lives->512]

**==>**e[70g-zk-m51-lc][1280-lives->768]

**gremlin>** g.E().has('reason', textContains('loves')).as('source').values('reason').as('reason').select('source').outV().values('name').as('god').select('source').inV().values('name').as('thing').select('god', 'reason', 'thing')

**==>**[god:neptune, reason:loves waves, thing:sea]

**==>**[god:jupiter, reason:loves fresh breezes, thing:sky]

Chapter 4. Configuration

**Table of Contents**

[4.1. Example Configurations](http://docs.janusgraph.org/latest/configuration.html#_example_configurations)

[4.1.1. Cassandra+Elasticsearch](http://docs.janusgraph.org/latest/configuration.html#_cassandra_elasticsearch)

[4.1.2. HBase+Caching](http://docs.janusgraph.org/latest/configuration.html#_hbase_caching)

[4.1.3. BerkeleyDB](http://docs.janusgraph.org/latest/configuration.html#_berkeleydb)

[4.1.4. Further Examples](http://docs.janusgraph.org/latest/configuration.html#_further_examples)

[4.2. Using Configuration](http://docs.janusgraph.org/latest/configuration.html#_using_configuration)

[4.2.1. JanusGraphFactory](http://docs.janusgraph.org/latest/configuration.html#_janusgraphfactory)

[4.2.2. JanusGraph Server](http://docs.janusgraph.org/latest/configuration.html#_janusgraph_server)

[4.3. Global Configuration](http://docs.janusgraph.org/latest/configuration.html#configuration-global)

[4.3.1. Changing Offline Options](http://docs.janusgraph.org/latest/configuration.html#_changing_offline_options)

A JanusGraph graph database cluster consists of one or multiple JanusGraph instances. To open a JanusGraph instance, a configuration has to be provided which specifies how JanusGraph should be set up.

需要制定一个configuration来提供如何创建一个JanusGraph

A JanusGraph configuration specifies which components JanusGraph should use, controls all operational aspects of a JanusGraph deployment, and provides a number of tuning options[调优选项] to get maximum performance from a JanusGraph cluster.

At a minimum[最低限度], a JanusGraph configuration must define the persistence engine that JanusGraph should use as a storage backend. [Part III, “Storage Backends”](http://docs.janusgraph.org/latest/storage-backends.html) lists all supported persistence engines and how to configure them respectively. If advanced graph query support (e.g full-text search, geo search, or range queries) is required an additional indexing backend must be configured. See [Part IV, “Index Backends”](http://docs.janusgraph.org/latest/index-backends.html) for details. If query performance is a concern, then caching should be enabled. Cache configuration and tuning is described in [Chapter 11, *JanusGraph Cache*](http://docs.janusgraph.org/latest/caching.html).

## 4.1. Example Configurations

Below are some example configuration files to demonstrate how to configure the most commonly used storage backends, indexing systems, and performance components. This covers only a tiny portion of the available configuration options. Refer to [Chapter 13, *Configuration Reference*](http://docs.janusgraph.org/latest/config-ref.html) for the complete list of all options.

### 4.1.1. Cassandra+Elasticsearch

Sets up JanusGraph to use the Cassandra persistence engine running locally and a remote Elastic search indexing system:

storage.backend=cassandra

storage.hostname=localhost

index.search.backend=elasticsearch

index.search.hostname=100.100.101.1, 100.100.101.2

index.search.elasticsearch.client-only=true

### 4.1.2. HBase+Caching

Sets up JanusGraph to use the HBase persistence engine running remotely and uses JanusGraph’s caching component for better performance.

storage.backend=hbase

storage.hostname=100.100.101.1

storage.port=2181

cache.db-cache = true

cache.db-cache-clean-wait = 20

cache.db-cache-time = 180000

cache.db-cache-size = 0.5

### 4.1.3. BerkeleyDB

Sets up JanusGraph to use BerkeleyDB as an embedded persistence engine with Elasticsearch as an embedded indexing system.

storage.backend=berkeleyje

storage.directory=/tmp/graph

index.search.backend=elasticsearch

index.search.directory=/tmp/searchindex

index.search.elasticsearch.client-only=false

index.search.elasticsearch.local-mode=true

[Chapter 13, *Configuration Reference*](http://docs.janusgraph.org/latest/config-ref.html) describes all of these configuration options in detail. The conf directory of the JanusGraph distribution contains additional configuration examples.

### 4.1.4. Further Examples

There are several example configuration files in the conf/ directory that can be used to get started with JanusGraph quickly. Paths to these files can be passed to JanusGraphFactory.open(...) as shown below:

// Connect to Cassandra on localhost using a default configuration

graph = JanusGraphFactory.open("conf/janusgraph-cassandra.properties")

// Connect to HBase on localhost using a default configuration

graph = JanusGraphFactory.open("conf/janusgraph-hbase.properties")

## 4.2. Using Configuration

How the configuration is provided to JanusGraph depends on the instantiation mode.

### 4.2.1. JanusGraphFactory

#### 4.2.1.1. Gremlin Console

The JanusGraph distribution contains a command line Gremlin Console which makes it easy to get started and interact with JanusGraph. Invoke bin/gremlin.sh (Unix/Linux) or bin/gremlin.bat (Windows) to start the Console and then open a JanusGraph graph using the factory with the configuration stored in an accessible properties configuration file:

graph = JanusGraphFactory.open('path/to/configuration.properties')

#### 4.2.1.2. JanusGraph Embedded

JanusGraphFactory can also be used to open an embedded JanusGraph graph instance from within a JVM-based user application. In that case, JanusGraph is part of the user application and the application can call upon JanusGraph directly through its public API.

#### 4.2.1.3. Short Codes

If the JanusGraph graph cluster has been previously configured and/or only the storage backend needs to be defined, JanusGraphFactory accepts a colon-separated string representation of the storage backend name and hostname or directory.

graph = JanusGraphFactory.open('cassandra:localhost')

graph = JanusGraphFactory.open('berkeleyje:/tmp/graph')

### 4.2.2. JanusGraph Server

JanusGraph, by itself, is simply a set of jar files with no thread of execution. There are two basic patterns for connecting to, and using a JanusGraph database:

**Patterns**

1. JanusGraph can be used by embedding JanusGraph calls in a client program where the program provides the thread of execution.
2. JanusGraph packages a long running server process that, when started, allows a remote client or logic running in a separate program to make JanusGraph calls. This long running server process is called **JanusGraph Server**.

For the JanusGraph Server, JanusGraph uses [Gremlin Server](http://tinkerpop.apache.org/docs/3.2.6/reference#gremlin-server) of the [Apache TinkerPop](http://tinkerpop.apache.org/) stack to service client requests. JanusGraph provides an out-of-the-box configuration for a quick start with JanusGraph Server, but the configuration can be changed to provide a wide range of server capabilities.

Configuring JanusGraph Server is accomplished through a JanusGraph Server yaml configuration file located in the ./conf/gremlin-server directory in the JanusGraph distribution. To configure JanusGraph Server with a graph instance (JanusGraph), the JanusGraph Server configuration file requires the following settings:

...

graphs: {

graph: conf/janusgraph-berkeleyje.properties

}

plugins:

- janusgraph.imports

...

The entry for graphs defines the bindings to specific JanusGraph configurations. In the above case it binds graph to a JanusGraph configuration at conf/janusgraph-berkeleyje.properties. The plugins entry enables the JanusGraph Gremlin Plugin, which enables auto-imports of JanusGraph classes so that they can be referenced in remotely submitted scripts.

Learn more about configuring and using JanusGraph Server in [Chapter 7, *JanusGraph Server*](http://docs.janusgraph.org/latest/server.html).

#### 4.2.2.1. Server Distribution

The JanusGraph zip file contains a quick start server component that helps make it easier to get started with Gremlin Server and JanusGraph. Invoke bin/janusgraph.sh start to start Gremlin Server with Cassandra and Elasticsearch.

|  |  |
| --- | --- |
| [Note] | **Note** |
| For security reasons Elasticsearch and therefore janusgraph.sh must be run under a non-root account |

## 4.3. Global Configuration

JanusGraph distinguishes between local and global configuration options. Local configuration options apply to an individual JanusGraph instance. Global configuration options apply to all instances in a cluster. More specifically, JanusGraph distinguishes the following five scopes for configuration options:

* **LOCAL**: These options only apply to an individual JanusGraph instance and are specified in the configuration provided when initializing the JanusGraph instance.
* **MASKABLE**: These configuration options can be overwritten for an individual JanusGraph instance by the local configuration file. If the local configuration file does not specify the option, its value is read from the global JanusGraph cluster configuration.
* **GLOBAL**: These options are always read from the cluster configuration and cannot be overwritten on an instance basis.
* **GLOBAL\_OFFLINE**: Like GLOBAL, but changing these options requires a cluster restart to ensure that the value is the same across the entire cluster.
* **FIXED**: Like GLOBAL, but the value cannot be changed once the JanusGraph cluster is initialized.

When the first JanusGraph instance in a cluster is started, the global configuration options are initialized from the provided local configuration file. Subsequently changing global configuration options is done through JanusGraph’s management API. To access the management API, call g.getManagementSystem() on an open JanusGraph instance handle g. For example, to change the default caching behavior on a JanusGraph cluster:

mgmt = graph.openManagement()

mgmt.get('cache.db-cache')

// Prints the current config setting

mgmt.set('cache.db-cache', true)

// Changes option

mgmt.get('cache.db-cache')

// Prints 'true'

mgmt.commit()

// Changes take effect

### 4.3.1. Changing Offline Options

Changing configuration options does not affect running instances and only applies to newly started ones. Changing GLOBAL\_OFFLINEconfiguration options requires restarting the cluster so that the changes take effect immediately for all instances. To change GLOBAL\_OFFLINEoptions follow these steps:

* Close all but one JanusGraph instance in the cluster
* Connect to the single instance
* Ensure all running transactions are closed
* Ensure no new transactions are started (i.e. the cluster must be offline)
* Open the management API
* Change the configuration option(s)
* Call commit which will automatically shut down the graph instance
* Restart all instances

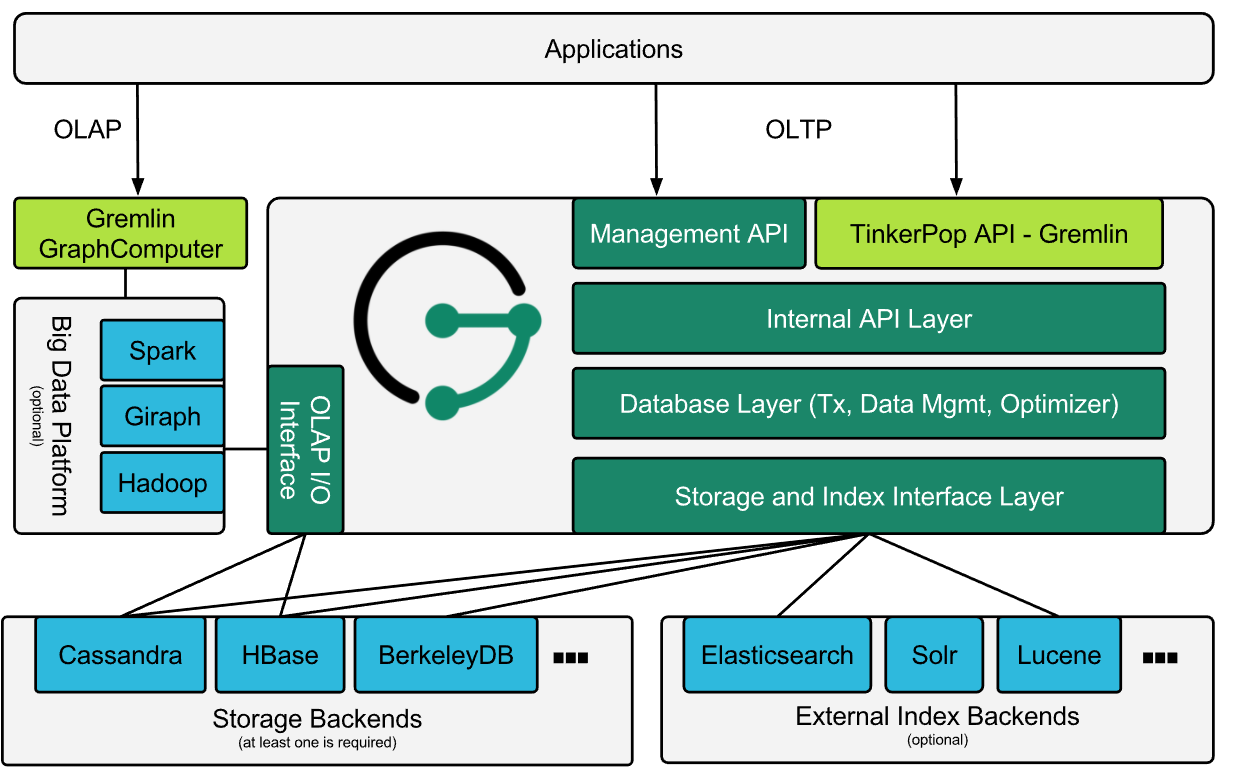
Refer to the full list of configuration options in [Chapter 13, *Configuration Reference*](http://docs.janusgraph.org/latest/config-ref.html) for more information including the configuration scope of each option.

JanusGraph

JanusGraph graph database implementation of the Blueprint's interface.这是一个接口, 定义了图数据库的接口实现,  
\* Use {**@link** JanusGraphFactory} to open and configure JanusGraph instances.

通过JanusGraphFactory来打开或者配置一个JanusGraph实例.

JanusGraphTransaction



博客文章:

Extending JanusGraph Server

JanusGraph支持两种类型的索引：graph index和vertex-centric index。**graph index常用于根据属性查询Vertex或Edge的场景；vertex index在图遍历场景非常高效，尤其是当Vertex有很多Edge的情况下。**

Graph Index

Graph Index是整个图上的全局索引结构，用户可以通过属性高效查询Vertex或Edge。如下面的代码：

g.V().has('name','hercules')

g.E().has('reason', textContains('loves'))

**上面的例子即为根据属性查找Vertex或Edge的实例**，**如果没有设置索引，上述的操作将会导致全表扫描，对大图来说是不可接受的。**

JanusGraph支持两种不同的Graph Index，Composte index和Mixed Index，Compostie非常高效和快速，但只能应用对某**特定的**，**预定义**的属性key组合进行**相等**查询。Mixed index可用在查询任何index key的组合上并支持多条件查询，除了相等条件要依赖于后端索引存储。

这两种类型的Index都是通过JanusGraph的management操作的：

JanusGraphManagement.buildIndex(String,Class）

第一个参数是index的名称，第二个参数是要索引的类（如Vertex.class），name必须唯一。如果是在同一事务中新增的属性key所构成Index将会即刻生效，否则需要运行一个reindex proceudre来同步索引和数据，直到同步完成，否则索引不可用。推荐在初始化schema时同时定义索引。

注意：如果没有建索引，会进行全表扫面，此时性能非常低，可以通过配置**force-index**参数禁止全表扫描。

Composite Index

Comosite index通过一个或多个固定的key组合来获取Vertex Key或Edge，也即查询条件是在Index中固定的。

[复制代码](javascript:void(0);)

// 在graph中有事务执行时绝不能创建索引（否则可能导致死锁）

graph.tx().rollback()

mgmt = graph.openManagement()

name = mgmt.getPropertyKey('name')

age = mgmt.getPropertyKey('age')

// 构建根据name查询vertex的组合索引

mgmt.buildIndex('byNameComposite',Vertex.class).addKey(name).buildCompositeIndex()

// 构建根据name和age查询vertex的组合索引

mgmt.buildIndex('byNameAndAgeComposite',Vertex.class).addKey(name).addKey(age).buildCompositeIndex()

mgmt.commit()

//等待索引生效

mgmt.awaitGraphIndexStatus(graph,'byNameComposite').call()

mgmt.awaitGraphIndexStatus(graph,'byNameAndAgeComposite').call()

//对已有数据重新索引

mgmt = graph.openManagement()

mgmt.updateIndex(mgmt.getGraphIndex("byNameComposite"),SchemaAction.REINDEX).get()

mgmt.updateIndex(mgmt.getGraphIndex("byNameAndAgeComposite"),SchemaAction.REINDEX).get()

mgmt.commit()

[复制代码](javascript:void(0);)

需要注意的是，Composite index需要在查询条件完全匹配的情况下才能触发，如上面代码，g.V().has('name', 'hercules')和g.V().has('age',30).has('name','hercules')都是可以触发索引的，但g.V().has('age',30)则不行，因并未对age建索引。g.V().has('name','hercules').has('age',inside(20,50))也不可以，因只支持精确匹配，不支持范围查询。

### Index Uniqueness

Composite Index也可以作为图的属性唯一约束使用，如果composite graph index被设置为unique()，则只能存在最多一个对应的属性组合。

[复制代码](javascript:void(0);)

graph.tx().rollback()//Never create new indexes while a transaction is active

mgmt = graph.openManagement()

name = mgmt.getPropertyKey('name')

mgmt.buildIndex('byNameUnique',Vertex.class).addKey(name).unique().buildCompositeIndex()

mgmt.commit()

//Wait for the index to become available

mgmt.awaitGraphIndexStatus(graph,'byNameUnique').call()

//Reindex the existing data

mgmt = graph.openManagement()

mgmt.updateIndex(mgmt.getGraphIndex("byNameUnique"),SchemaAction.REINDEX).get()

mgmt.commit()

[复制代码](javascript:void(0);)

注意：对于设置为最终一致性的后端存储，index的一致性必须被设置为允许锁定。

Mixed Index

Mixed Index支持通过其中的任意key的组合查询Vertex或者Edge。Mix Index使用上更加灵活，而且支持范围查询等（不仅包含相等）；从另外一方面说，Mixed index效率要比Composite Index低。

与Composite key不同，Mixed Index需要配置索引后端，JanusGraph可以在一次安装中支持多个索引后端，而且每个索引后端必须使用JanusGraph中配置唯一标识：称为indexing backend name。

[复制代码](javascript:void(0);)

graph.tx().rollback()//Never create new indexes while a transaction is active

mgmt = graph.openManagement()

name = mgmt.getPropertyKey('name')

age = mgmt.getPropertyKey('age')

mgmt.buildIndex('nameAndAge',Vertex.class).addKey(name).addKey(age).buildMixedIndex("search")

mgmt.commit()

//Wait for the index to become available

mgmt.awaitGraphIndexStatus(graph,'nameAndAge').call()

//Reindex the existing data

mgmt = graph.openManagement()

mgmt.updateIndex(mgmt.getGraphIndex("nameAndAge"),SchemaAction.REINDEX).get()

mgmt.commit()

[复制代码](javascript:void(0);)

上面的代码建立了一个名为nameAndAge的索引，该索引使用name和age属性构成，并设定其索引后端为"search"，对应到配置文件中为：index.serarch.backend，如果叫solrsearch，则需要增加：index.**solrsearch**.backend配置。

下面展示了如果使用text search作为默认的搜索行为：

mgmt.buildIndex('nameAndAge',Vertex.class).addKey(name,Mapping.TEXT.getParameter()).addKey(age,Mapping.TEXT.getParameter()).buildMixedIndex("search")

更加详细的使用参考：Charpter21, Index Parameter and Full-Test Search

在使用上，支持范围查询和索引中任何组合查询，而不仅局限于“相等”查询方式：

g.V().has('name', textContains('hercules')).has('age', inside(20,50))

g.V().has('name', textContains('hercules'))

g.V().has('age', lt(50))

Mixed Index支持全文检索，范围检索，地理检索和其他方式，参考Chapter20, Search Predicates and Data Types。

**注意：不像composite index，mixed index不支持唯一性。**

### Adding Property Keys

可以向已经存在的mixed index中新增属性，之后就可以在查询条件中使用了。

[复制代码](javascript:void(0);)

//Never create new indexes while a transaction is active

graph.tx().rollback()

mgmt = graph.openManagement()

//创建一个新的属性

location = mgmt.makePropertyKey('location').dataType(Geoshape.class).make()

nameAndAge = mgmt.getGraphIndex('nameAndAge')

//修改索引

mgmt.addIndexKey(nameAndAge, location)

mgmt.commit()

//Wait for the index to become available

mgmt.awaitGraphIndexStatus(graph,'nameAndAge').call()

//Reindex the existing data

mgmt = graph.openManagement()

mgmt.updateIndex(mgmt.getGraphIndex("nameAndAge"),SchemaAction.REINDEX).get()

mgmt.commit()

[复制代码](javascript:void(0);)

如果索引是在同意事务中创建的，则在该事务中马上可以使用。如果该属性Key已经被使用，需要执行reindex procedure来保证索引中包含了所有数据，知道该过程执行完毕，否则不能使用。

### Mapping Parameters

当向mixed index增加新的property key时（无论通过何种方式创建），可以指定一组参数来设置property value在后端的存储方式。参考mapping paramters overview章节。

Ordering

图查询的集合返回顺序可由order().by()指定，该方法包含了两个参数：

* 排序依据的属性名称
* 升降序，incr和decr

如：

g.V().has('name', textContains('hercules')).order().by('age', decr).limit(10)

返回了name属性中包含‘hercules’且以'age'降序返回的10条数据。

使用Order时需要注意：

* composite graph index原生不支持对返回结果排序，数据会被先加载到内存中再进行排序，对于大数据集合来讲成本非常高
* Mixed graph index本身支持排序返回，但排序中要使用的property key需要提前被加到mix index中去，如果要排序的property key不是index的一部分，将会导致整个数据集合加载到内存。

Label Constraint

有些情况下，我们不想对图中具有某一label的所有Vertex或Edge进行索引，例如，我们只想对有GOD标签的节点进行索引，此时我们可以使用indexOnly方法表示只索引具有某一Label的Vertex和Edge。如下：

[复制代码](javascript:void(0);)

//Never create new indexes while a transaction is active

graph.tx().rollback()

mgmt = graph.openManagement()

name = mgmt.getPropertyKey('name')

god = mgmt.getVertexLabel('god')

//只索引有god这一label的顶点

mgmt.buildIndex('byNameAndLabel',Vertex.class).addKey(name).indexOnly(god).buildCompositeIndex()

mgmt.commit()

//Wait for the index to become available

mgmt.awaitGraphIndexStatus(graph,'byNameAndLabel').call()

//Reindex the existing data

mgmt = graph.openManagement()

mgmt.updateIndex(mgmt.getGraphIndex("byNameAndLabel"),SchemaAction.REINDEX).get()

mgmt.commit()

[复制代码](javascript:void(0);)

label约束对mix index也是类似的，当一个有label约束的composite index被设置为唯一时，唯一约束只应用于具有此label的vertex或edge属性上。

Composite versus Mixed Indexes

1. 使用comosite key应用与确切的匹配场景，composite key不需要外部索引系统且通常具有更好的性能。

    作为一个例外，如果要精确匹配的值数量很小（如12个月份）或一个元素与图中很多的元素有关联，此时应使用mix index。

2. 对取范围，全文检索或位置查询这样的应用场景，应该使用mix index，而且使用mixed index可以提供order().by()的性能。

Vertex-centric Indexs

Vertex-centric index（顶点中心索引）是为每个vertex建立的本地索引结构，在大型graph中，每个vertex有数千条Edge，在这些vertex中遍历效率将会非常低（需要在内存中过滤符合要求的Edge）。Vertex-centric index可以通过使用本地索引结构加速遍历效率。

如：

h = g.V().has('name','hercules').next()

g.V(h).outE('battled').has('time', inside(10,20)).inV()

如果没有vertex-centric index，则需要便利所有的batteled边并找出记录，在边的数量庞大时效率非常低。

建立一个vertex-centric index可以加速查询：

[复制代码](javascript:void(0);)

//Never create new indexes while a transaction is active

graph.tx().rollback()

mgmt = graph.openManagement()

//找到一个property key

time = mgmt.getPropertyKey('time')

// 找到一个label

battled = mgmt.getEdgeLabel('battled')

// 创建vertex-centric index

mgmt.buildEdgeIndex(battled,'battlesByTime',Direction.BOTH,Order.decr, time)

mgmt.commit()

//Wait for the index to become available

mgmt.awaitGraphIndexStatus(graph,'battlesByTime').call()

//Reindex the existing data

mgmt = graph.openManagement()

mgmt.updateIndex(mgmt.getGraphIndex("battlesByTime"),SchemaAction.REINDEX).get()

mgmt.commit()

[复制代码](javascript:void(0);)

上面的代码对battled边根据time以降序建立了双向索引。buildEdgeIndex()方法中的第一个参数是要索引的Edge的Label，第二个参数是index的名称，第三个参数是边的方向，BOTH意味着可以使用IN/OUT，如果只设置为某一方向，可以减少一半的存储和维护成本。最后两个参数是index的排序方向，以及要索引的property key，property key可以是多个，order默认为升序（Order.ASC）。

[复制代码](javascript:void(0);)

graph.tx().rollback()//Never create new indexes while a transaction is active

mgmt = graph.openManagement()

time = mgmt.getPropertyKey('time')

rating = mgmt.makePropertyKey('rating').dataType(Double.class).make()

battled = mgmt.getEdgeLabel('battled')

mgmt.buildEdgeIndex(battled,'battlesByRatingAndTime',Direction.OUT,Order.decr, rating, time)

mgmt.commit()

//Wait for the index to become available

mgmt.awaitRelationIndexStatus(graph,'battlesByRatingAndTime','battled').call()

//Reindex the existing data

mgmt = graph.openManagement()

mgmt.updateIndex(mgmt.getRelationIndex(battled,'battlesByRatingAndTime'),SchemaAction.REINDEX).get()

mgmt.commit()

[复制代码](javascript:void(0);)

上面的代码建立了battlesByRatingAndTime索引，并以rating和time构成，需要注意构成索引的property key的顺序非常重要，查询时只能根据propety key定义的顺序查询。

h = g.V().has('name','hercules').next()

g.V(h).outE('battled').property('rating',5.0)//Add some rating properties

g.V(h).outE('battled').has('rating', gt(3.0)).inV()

g.V(h).outE('battled').has('rating',5.0).has('time', inside(10,50)).inV()

g.V(h).outE('battled').has('time', inside(10,50)).inV()

对上面部分的代码，只有查询1,2是可以使用索引的，查询3使用time查询无法匹配先根据rating再根据time的index构造顺序。可以对一个label创建多个不同的索引来支持不同的遍历。JanusGraph自动选择最有效的索引，Vertex-centric仅支持相等和range/interval约束。

**注意：在vertex-centirc中使用的property key必须是显式定义的且未确定的class类型（不是Object.class）才能支持排序。如果数据类型浮点型，必须使用JanusGraph的Decimal或Precision数据类型。**

根据在同一事务中新建的label所创建的索引可以即刻生效，如果edge正在被使用，则需要运行reindex程序，直到该程序运行结束，否则该索引无法使用。

**注意：JanusGraph自动为每个edge label的每个property key建立了vertex-centric label，因此即使有数千个边也能高效查询。**

Vertex-centric label无法加速不受约束的遍历（在所有边中遍历），这种遍历随着边的增加会变的更慢，通常这些遍历可以作为受约束遍历重写来提高性能。

Ordering Traversals

下面的查询使用了local和limit方法获取了遍历过程的排序子集。

h = g..V().has('name','hercules').next()

g.V(h).local(outE('battled').order().by('time', decr).limit(10)).inV().values('name')

g.V(h).local(outE('battled').has('rating',5.0).order().by('time', decr).limit(10)).values('place')

如果排序字段和排序方向与vertex-centric index一致的话，上面的查询非常高效。

**注意：vertex 排序查询时JanusGraph对Gremlin的扩展，要使用该功需要一段冗长的语句，而且需要\_()步骤将JanusGraph转换为Gremlin管道。**

单词

Likewise 同样地

persistence engine 存储引擎

storage backend 后端存储

tuning 调优

Refer to 参考

make extensive use of 充分利用