



# LDBC Benchmark on JanusGraph

0：IO | 大数据系统与大规模数据分析课程报告 | 2018.6.19

张俊阳 | 2017E8018661163 | zhangjunyang@iie.ac.cn
张小洋 | 2017E8018661065 | zhangxiaoyang@iie.ac.cn
王传仁 | 2017E8018661118 | wangchuanren@iie.ac.cn
刘志磊 | 201728018629141 | liuzhilei@iie.ac.cn
冀海川 | 2017E8018661165 | jihaichuang@iie.ac.cn

# 1 LDBC SNB 简介

LDBC 社交网络图数据 Benchmark 是一项针对专业图数据库的基准测试工具，它包含一个数据生成器生成社交网络数据，三种工作负载（workloads）：交互式(Interactive)，商业智能(Business Intelligence)和图分析( Graph Analytics)。目前，只有交互式工作负载在初稿阶段发布。商业智能工作负载的只读部分目前能够预览。

主要的 SNB 主要部件有下列四项

1) SNB benchmark 规范文档
   https://github.com/ldbc/ldbc_snb_docs
2) SNB 数据生成器
   https://github.com/ldbc/ldbc_snb_datagen
3) LDBC 驱动（实现查询的驱动）
   https://github.com/ldbc/ldbc_driver
4) 交互式工作负载实现
   https://github.com/ldbc/ldbc_snb_implementations

# 2 JanusGraph 简介

JanusGraph 是一个可扩展的图数据库，可以把包含数千亿个顶点和边的图存储在多机集群上。它支持事务，支持数千用户实时、并发访问存储在其中的图。

除此之外 JanusGraph 还具有下列特点：

- 弹性，线性可扩展
- 针对性能和容错的数据分发和复制机制
- 多数据中心高可用性和热备份
- 支持 ACID 和 eventual consistency
- 支持多个后端存储
- 通过与大数据平台集成支持全局图数据分析，报表和 ETL

- 通过集成索引引擎支持地理位置，数字和全文搜索
- 原生集成 Apache Tinker Pop 图技术栈
- 开源，基于 Apache 2 Licenses
- 通过可视化工具展示存储在 JanusGraph 中的图

基于 LDBC 社交网络图实现 Benchmark，并比较不同的选项对性能的影响，具体分为以下五个部分：

**A) 数据生成：datagen**

利用 ldbc_smb_datagen 工具可以生成用于测试的社交网络数据，生成的数据以文件形式存储。

**B) 数据导入：importer**

利用已生成的数据文件，将数据导入到图存储数据库 Janusgraph 中。

**C) 工作负载：Workload**

工作负载（workload）是进行实际测试的方式，我们将介绍用于本测试的工作负载的实现。

**D) Benchmark 执行**

完成图数据的导入和工作负载的实现后，可基于这些数据和操作执行 benchmark。benchmark 在执行时也可对具体的测试方案进行定制。

**E) 性能比较及分析**

由 benchmark 测试结果可反映不同的 janusgraph 配置条件对其性能的影响。我们将对这些测试结果进行简单的分析。

# 1 数据生成：Datagen

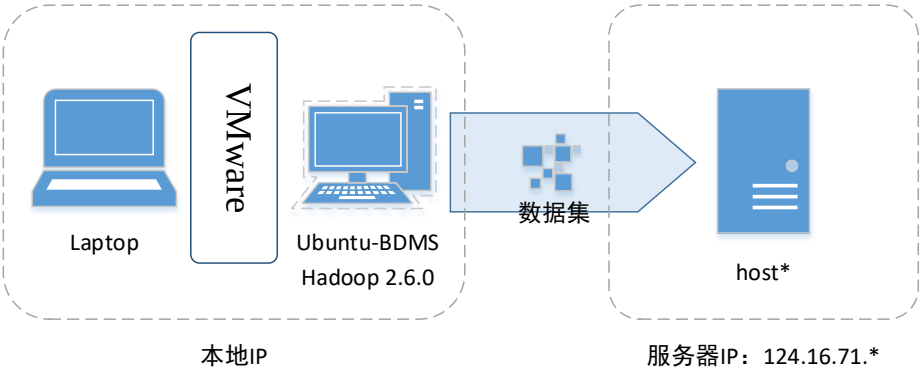
## 1.1 简介

测试数据的生成是进行 benchmark 测试的第一步，LDBC 为其制定的测试需求设计了一个数据生成器：ldbc_snb_datagen (github 仓库地址)，本实验使用该数据生成器进行数据生成。

## 1.2 ldbc_snb_datagen

### 1.2.1 环境需求

1) **Apache Hadoop v2.6.0**
ldbc_snb_datagen (以下简称 datagen)使用 Apache Hadoop v2.6.0 进行数据生成，实验机条件有限，我们按照以下的方式进行数据生成：




2) **Apache Maven**
ldbc_snb 项目主要使用 Apache Maven 工具进行项目管理和自动构建，在 datagen 的安装中，需要配置好 maven 工具以支持安装的进行。

3) **Python 2.7**
datagen 通过 python 实现对生成数据属性的自定义设置，官方给出的版本需求是 python 2.7。

### 1.2.2 参数配置

将 datagen 文件下载到本地后，根据 datagen 官方教程配置运行环境。参数配置主要是配置~/*/ldbc_snb_datagen-master/run.sh 文件中的路径：

- `DEFAULT_HADOOP_HOME=/usr/local/java/hadoop-2.6n.0`
- `$LDBC_SNB_DATAGEN_HOME/test_params.ini`

第一条参数设置对应 Hadoop 的实际安装路径；
第二条参数设置对应 datagen 进行数据生成时参数文件位置。

运行命令./run.sh 即可通过 maven 工具自动构建项目生成 jar 包。我们将利用这个 jar 包生成所需的测试数据集。

1) datagen 参数
datagen 通过参数文件的形式对生成数据的属性（图属性、数据规模、部分数据格式等）进行自定义设置，下面就实验中用到的几

个重要的参数进行介绍。

- **ldbc.snb.datagen.serializer.personSerializer**

a) 该参数主要用于设置社交网络中 person 节点和 knows 边数据序列化的模式；

b) 本次实验设置 *ldbc.snb.datagen.serializer.snb.interactive.CSVPersonSerializer*

- **ldbc.snb.datagen.serializer.invariantSerializer**

a) 该参数主要用于设置社交网络中 person 节点和 knows 边数据序列化的模式；

b) 本次实验设置 *ldbc.snb.datagen.serializer.snb.interactive.CSVInvariantSerializer*

- **ldbc.snb.datagen.serializer.personActivitySerializer**

a) 该参数主要用于设置社交网络中 person 节点和 knows 边数据序列化的模式；

b) 本次实验设置 *ldbc.snb.datagen.serializer.snb.interactive.CSVPersonActivitySerializer*

- **ldbc.snb.datagen.generator.scaleFactor**

a) 该参数主要用于设置社交网络数据的规模。本实验测试的数据用于 snb.interactive 测试，该条件下的数据规模从 0.1GB~1000GB 共有 9 个可用的参数选项，对于不同的数据规模设置，图属性大致如下：

| Scale Factor | 1 | 3 | 10 | 30 | 100 | 300 | 1000 |
|---|---|---|---|---|---|---|---|
| # of Persons | 11K | 27K | 73K | 182K | 499K | 1.25M | 3.6M |
| # of Years | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Start Year | 2010 | 2010 | 2010 | 2010 | 2010 | 2010 | 2010 |

b) 本次实验设置 *snb.interactive.0.1*

- **ldbc.snb.datagen.serializer.dateFormatter**

a) 本次实验基于 Janusgraph 实例的基础上完成，该实例实现了 importer 工具完成图数据的导入，要求数据及文件中日期格式必须为 Long 型，因此在生成数据时要对日期格式进行设置

b) 本次实验设置 *ldbc.snb.datagen.serializer.formatter.LongDateFormatter*

通过上述的参数设置，本次实验将生成标准的大小为 100MB 左右的测试数据，基本上每个不同 entity 的数据内容分别存储在不同的 csv 文件中。

## 1.3 数据

datagen 的数据生成利用 Hadoop 的 Mapreduce 方法实现，最后生成的数据将构成一张完整的社交网络图。每次执行 datagen 将生成 3 类数据文件，主要分布在 2 个文件夹中。下面将对 datagen 产生的数据位置、格式和属性进行介绍。

### 1.3.1 数据分布

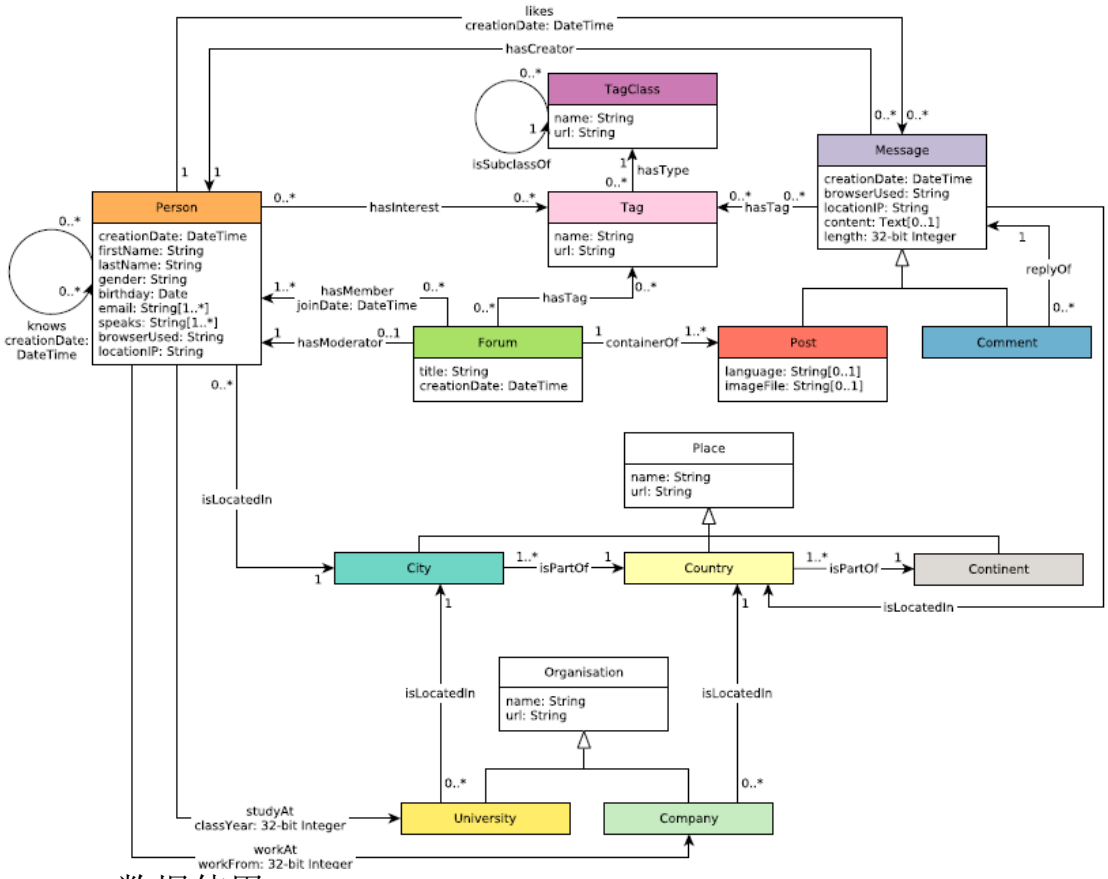
datagen 生成的与 benchmark 测试相关的数据分布在 2 个文件夹中，一个命

名为 social_network，一个命名为 substitution_parameters。

1) social_network
   该文件夹存储了 datagen 生成的社交网络图的数据，数据格式是 csv。这些文件由 mapreduce 任务生成，因此被存储在 hdfs 上。本次实验需要将文件从虚拟机上传到服务器上进行测试，因此在完成数据生成后，需要将该文件夹通过 get 方法下载到虚拟机本地上便于之后的文件传输。
2) substitution_parameters
   该文件针对 benchmark 中设计的 25 个 BI query 和 14 个 IC(interactive) query 设置了参数，这些参数是在进行查询时必须提供的。参数文件的格式为 txt 文件。

### 1.3.2 数据属性

1) 文件属性
   datagen 将生成三类文件：
   - 数据集：用于 benchmark 测试的主要数据集，大约占生成数据的 90%；
   - 更新流（update stream）：用于 update query 的数据，大约占生成数据的 10%；
   - 参数（substitution parameters）：用于 BI query 和 IC query 的参数。
2) data schema
   datagen 生成的所有文件构成完整的社交网络图。根据官方文件的说明，下图展示了这个图数据的 schema：




## 1.4 数据使用

本次实验的实验数据存储在~/janusgraph/test-data-100m/下，包括 social_network 和 substitution_parameters 文件夹，在进行数据测试时，对 workload 的参数进行如下设置：

- `ldbc.snb.interactive.parameters_dir=/home/user26/test-data-100m/substitution_parameters`
- `ldbc.snb.interactive.updates_dir=/home/user26/test-data-100m/social_network`

# 2 数据导入：Importer

数据生成后，可使用 importer 工具进行数据导入。

由于实现具体的 benchmark 查询(Query)和生成数据高度相关，因此对于数据生成所用到的参数会在下一章节工作负载实现中使用，该部分主要着重阐述将生成的 social_network 数据导入到 JanusGraph 后端存储中，通过执行 com.ldbc.snb.janusgraph.importer.Main 类来导入，基于 LDBC ldbc_snb_implementations 开源代码修改实现。

A) 启动 JansusGraph server（gremlin-server）

导入前需要启动 JansusGraph server（gremlin-server），具体命令：

```
java -cp target/janusgraphSNBInteractive-0.1-SNAPSHOT-jar-with-dependencies.jar com.ldbc.snb.janusgraph.importer.Main [-n 2] [-s 2000] [-d test-data-100m/social_network] [-c bdb.conf]
```

参数列表：

- *--numThreads/-n*

  加载过程中线程数

- *--transactionSize/-s*

  读取文件事务的大小（每个读取任务读取的行数）

- --dataset/-d

  要导入数据集的文件夹路径（数据集的时间戳必须采用 *Long*，使用 *ldbc.snb.datagen.serializer.formatter.LongDataFormatter* 生成数据 ）

- --backend-config/-c

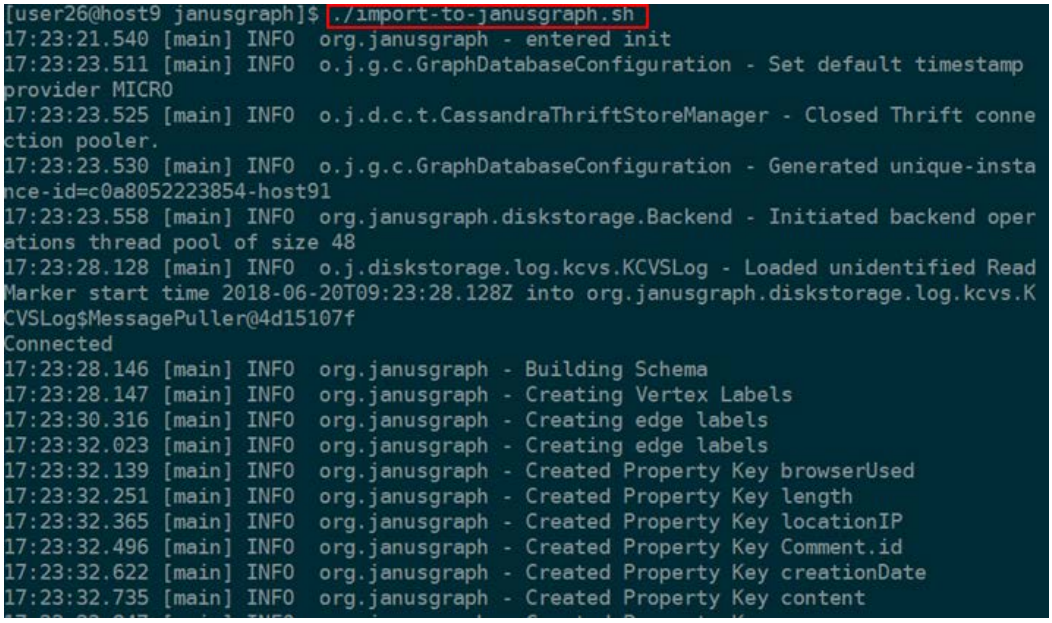
  配置后端存储的文件路径。示例文件为 resources/bdb.conf

B) 通过 shell 脚本 import-to-janusgraph.sh 导入数据

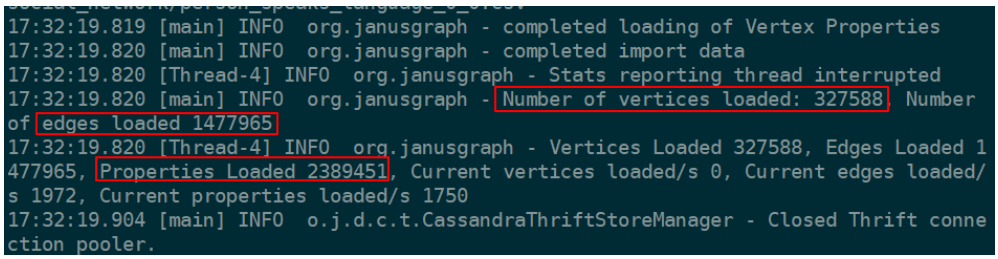
根据 Janusgraph 不同的存储后端，需用不同的参数进行数据导入。这里分别针对三类存储后端（Cassandra，BerkeleyDB，Hbase）使用脚本进行数据导入：

- *./import-to-janusgraph.sh* （Cassandra）
- *./import-to-janusgraph-berkeley.sh* （BerkeleyDB）
- *./import-to-janusgraph-hbase.sh* （Hbase）

通过上述的两步操作即可完成数据导入，下面以 Cassandra 导入过程为例介绍整个导入过程：




导入完成




测试 100m 数据导入了 327588 个点，1477965 条边，2389451 条 properties。

# 3 工作负载：workload

该部分基于 https://github.com/ldbc/ldbc_snb_implementations 官方 LDBC Interactive 代码中针对 Titan 实现的交互式工作负载 benchmark 部分实现。

## 3.1 主要组件

A) Workload

- ReadOperation
- UpdateOperation

B) DbConnector

- JanusGraphDb

C) Operation Handler

- LdbcQuery[1-14]Handler：14 个复杂查询
- LdbcQueryU[1-8]Handler.java：8 个更新操作
- LdbcShortQuery[1-7]Handler，7 个简单查询

具体的 Query 规范在 ldbc-snb-specification.pdf 文件中有详细描述。具体实现代码请看代码仓库或者附件。

此外，该部分需要 LDBC SNB Dirver 包，打包前在 pom.xml 引入如下依赖：

```
<dependency>
    <groupId>com.ldbc.driver</groupId>
        <artifactId>jeeves</artifactId>
        <version>0.3-SNAPSHOT</version>
</dependency>
```

# 4 Benchmark 执行

Benchmark 的执行主要需要完成两个部分：配置后端和执行 benchmark。

## 4.1 配置后端 janusgraph

JanusGraph 支持多种数据存储后端，包括：Apache Cassandra、Apache HBase、Google Cloud Bigtable 和 Oracle Berkeley DB。本次实践中使用了 Apache Cassandra 和 Orcale Berkeley DB Java Edition 分别作为 JanusGraph 的存储后端进行测试。

A) JanusGraph + Apache Cassandra 集成配置。

Apache Cassandra 是一个开源的、分布式、无中心、支持水平拓展、高可用的 KEY-VALUE 类型的 NOSQL 数据库[1]。根据 JanusGraph 的官方文档介绍，Cassandra 作为 JanusGraph 的存储后端时有多种集成模式[2]，这里使用了 Remote Server 模式，Cassandra 以集群方式存在， 运行在其他主机上的 JanusGraph 基于 Socket 的读/写来访问 Cassandra 集群。我们通过控制 Cassandra 集群的节点数量，运行 LDBC 的社交网络图 Benchmark 对 JanusGraph 的性能进行测试。测试中使用的 Cassandra 版本是 3.11.2。

1) 单节点的 Cassandra 连接配置

在本例中 JanusGraph 与单个节点的 Cassandra 进行连接，Cassandra 运行在实验主机 host5 上，其主要配置文件 conf/cassandra.yaml 中的关键配置如下：

```
# 集群名称
cluster_name: 'IO Cassandra Cluster'
# 设置种子节点 种子节点为 host9 ip 地址为 192.168.5.34
seed_provider:
    -class_name: org.apache.cassandra.locator.SimpleSeedProvider
     parameters:
         -seeds: "192.168.5.34"
# 设置监听地址
listen_address: 192.168.5.34
# 启用 thrift rpc server，否则 JanusGraph 无法与 Cassandra 建立连接
start_rpc: true
rpc_address: 192.168.5.34
```

JanusGraph 的相关配置文件（主要是 gremlin-server.sh 加载的配置文件和运行 benchmark 的程序加载的配置文件）中添加如下配置：

```
storage.backend=cassandrathrift
storage.hostname=192.168.5.34
```

2) 两节点的 Cassandra 连接配置

在本例中我们部署了两个节点的 Cassandra 集群，使用 host9 作为集群的 seed 节点，host4 作为普通节点，每个节点的 Cassandra 配置同上例中的单节点配置类似：seeds 设置为 host9 的 ip 地址 192.168.5.34，listen_address 和 rpc_address 设置为各自对应的 ip 地址，启动时先启动 seed 节点上的 Cassandra 进程，之后普通节点启动会自动加入到集群中。集群启动后使用 Cassandra 提供的 nodetool 工具查看各节点的状态，如图 1.1 所示，我们向 JanusGraph 导入了约 100M 的数据，这些数据被分配存储在了 Cassandra 集群的节点中。

```
[user26@host9 ~]$ .opt/cassandra/bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens       Owns (effective)  Host ID                               Rack
UN  192.168.5.29  61.83 MiB  256          48.3%             978c82e3-1562-4745-b825-e4d882e897c8  rack1
UN  192.168.5.34  46.61 MiB  256          51.7%             64ea6135-a87d-4d90-af8e-12c1be7e05f3  rack1

[user26@host9 ~]$
```

*图 1. 1 两节点 Cassandra 集群状态*

3) 四节点的 Cassandra 连接配置

为了进一步验证 Cassandra 集群节点数量对测试性能的影响，在本例中我们使用 host5、host7、host8、host10 部署在四个结点的 Cassandra 集群，seed 结点为 host5 和 host7，host8 和 host10 作为普通节点加入集

群。由于在这些机器上有其他用户运行的 Cassandra 进程造成了端口占用，除了前面例子中的配置外还需要修改 Cassandra 的端口，cassandra.yaml 中关键配置如下：

```
# 修改节点通信端口 默认为 7000，ssl 默认端口为 7001
storage_port: 7200
ssl_storage_port: 7201
# 修改 CQL 客户端通信端口 默认为 9042
native_transport_port: 9052
# 修改 rpc 端口 即 JanusGraph 与 Cassandra 的通信端口 默认端口为 9160
rpc_port: 9260
```

Cassandra 通过 JMX 监测结点，所以需要修改 conf/cassandra-env.sh 中相关的端口：

```
# 修改 JMX 连接端口 默认为 9042
JMX_PORT="8012"
```

除此之外，JanusGraph 相关的配置中除了指定存储后端 Cassandra 的地址，还要指定端口：

```
# 修改后端通信端口
storage.port=9260
```

做出如上修改后，就可以分别启动 Cassandra 集群和 JanusGraph 的 gremlin-server 进行连接了。Cassandra 集群按照 seed 节点和普通节点的顺序依次启动，启动完成后使用 nodetool 查看集群节点状态，如图 1.2 所示，同上例中一样，导入 JanusGraph 的数据被分配到了所有的节点进行存储。

```
[user26@host5 ~]$ .opt/cassandra/bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens       Owns (effective)  Host ID                               Rack
UN  192.168.5.30  19.28 MiB  256          24.1%             cfbblad7-7322-44df-bed7-0fe0a8a343f6  rack1
UN  192.168.5.32  22.11 MiB  256          26.2%             b3a9df85-790d-4c64-b497-569911eb15b3  rack1
UN  192.168.5.33  21.83 MiB  256          25.6%             3073e423-53e7-4a40-ad8a-71111825dd7f  rack1
UN  192.168.5.35  25.96 MiB  256          24.1%             3f99fb7d-05d6-4d3a-9892-1f9adcd24610  rack1
```

*图 1. 2 四节点 Cassandra 集群状态*

B) JanusGraph + Orcale Berkeley DB Java Edition 集成配置。

Berkeley DB 是一个是一个开源的文件数据库，介于关系数据库与内存数据库之间，使用方式与内存数据库类似，它提供的是一系列直接访问数据库的函数，而不是像关系数据库那样需要网络通讯、SQL 解析等步骤[3]。

JanusGraph 使用的相应存储后端为 Orcale Berkeley DB Java Edition，它与 JanusGraph 运行在同一个机器的 JVM 中，导入 JanusGraph 的图数据全部都会保存在本地磁盘上。这就将图数据的大小限制在了硬盘空间和内存空间

的范围内(大约 10-100 万个顶点的图)。但是，对于这种规模的图，使用 Berkeley DB 作为存储后端具有高于其他分布式数据库的性能。

JanusGraph 使用 Berkeley DB 作为存储后端集成的方法比较简单，无需安装 Berkeley DB，Orcale Berkeley DB Java Edition 以 jar 包的形式存在于 JanusGraph 的 lib 目录下，所以只需在 JanusGraph 相关的配置文件（gremlin-server 和 LDBC Benchmark）中指定后端名称和存储路径即可：

```
# 指定后端名称为 berkeleyje
storage.backend=berkeleyje
# 指定 berkeley db 的数据存储路径
storage.hostname=../data/graph
```

### 4.2 交互式工作负载 benchmark

首先获取 https://github.com/ldbc/ldbc_snb_driver 中的源码，然后使用 maven 编译安装 driver 包

*mvn clean install -DskipTests*

接着进入项目 janusgraph 目录下生成实现 benchmark 的 JAR 包

*mvn clean assembly:assembly -DskipTests*

对于我们的 Janusgraph 交互式工作负载 benchmark 实现，首先根据系统配置修改 ldbc_snb_interactive.properties 文件，接着使用脚本 interactive-benchmark.sh 执行驱动程序运行基准测试。

```
[user26@host9 janusgraph]$ ./interactive-benchmark.sh
22:18:58,103  INFO ExecuteWorkloadMode:40 - Driver Configuration
22:18:58,131  INFO ExecuteWorkloadMode:40 - Workload Start Time:       2018-06-20 - 14:19:03.096
Parameters:
        Name:                           LDBC
        DB:                             com.ldbc.snb.janusgraph.drivers.interactive.JanusGraphDb
        Workload:                       com.ldbc.driver.workloads.ldbc.snb.interactive.LdbcSnbInteractiveWorkload
        Operation Count:                500
        Warmup Count:                   100
        Skip Count:                     0
        Worker Threads:                 1
        Status Display Interval:        00:02.000 (m:s.ms)
        Time Unit:                      MILLISECONDS
        Results Directory:              /home/user26/janusgraph/results
        Time Compression Ratio:         1.0000000
        Validation Creation Params:     null
        Database Validation File:       null
        Calculate Workload Statistics:  false
        Spinner Sleep Duration:         00:00.000 (m:s.ms) / 0 (ms)
        Print Help:                     false
        Ignore Scheduled Start Times:   false
```

## 5 性能比较及分析

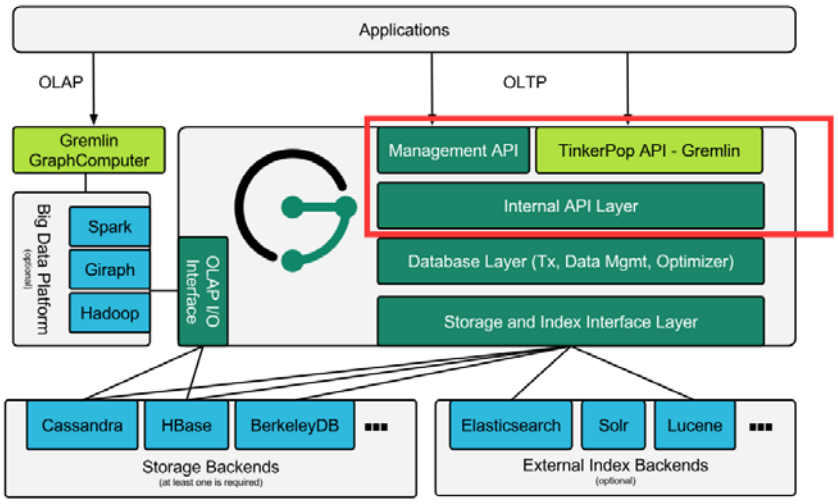
性能的比较分析主要从以下几个方面进行讨论：janusgraph 不同存储后端间的性能比较，benchmark 性能测试中对 query 的不同 API 实现方式的比较，以及 janusgraph 存储图时不同的划分方式之间的比较。

在这三类性能比较中，将简单介绍不同 API 实现方式和不同图划分方式，重点以不同存储后端的性能比较为例来进行分析。

## 5.1 API 实现方式：Gremlin API

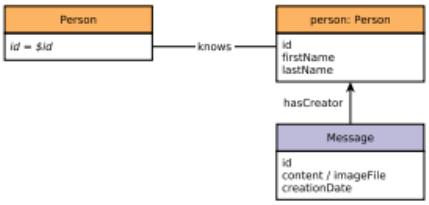
janusgraph 中 query 的实现主要依靠两类 API：**janusgraph 本身提供的 API** 和 **Gremlin API**。本次实验主要是

JanusGraph 发行版包含一个 Gremlin 控制台命令行，可以轻松上手并与 JanusGraph 交互。调用 bin/gremlin.sh 即可启动控制台，然后使用 JanusGraphFactory 打开 JanusGraph 图形。JanusGraphFactory 也可用于在基于 JVM 的用户应用程序中打开嵌入式 JanusGraph 图形实例。在这种情况下，应用程序可以直接通过其公共 API 调用 JanusGraph。




如上图所示，Gremlin API 位于 Internal API Layer 上层，直接面向应用程序。Gremlin 是 Apache TinkerPop 的一个组件，它独立于 JanusGraph 开发，得到大多数数据库的支持。JanusGraph 的发行版会自带一个 Gremlin。Gremlin 是一种面向路径的语言，能够简洁地表达复杂的图形遍历和变异操作。Gremlin 是一种功能性语言，便利运算符连接在一起形成类似路径的表达式。ldbc_snb_driver 中定义了对于 interactive workload 的基本的查询与更新操作。基于给定的查询操作，我们基于 gremlin java api 实现了相关的 handler。以其中的中的一个 query2 为例，如下图所示：

**Interactive / complex / 2**

| | | |
|---|---|---|
| IC 1 | query | Interactive / complex / 2 |
| IC 2 | title | Recent posts and comments by your friends |
| IC 3 | pattern |  |
| IC 4 | desc. | Given a start Person, find (most recent) Messages from all of that Person's friends, that were created before (and including) a given date. |
| IC 5 | params | |
| IC 6 | result | |
| IC 7 | sort | |
| IC 8 | limit | 20 |
| IC 9 | CPs | 1.1, 2.2, 2.3, 3.2 |
| C 10 | relevance | This is a navigational query looking for paths of length two, starting from a given Person, going to their friends and from them, moving to their published Posts and Comments. This query exercices both the optimizer and how data is stored. It tests the ability to create execution plans taking advantage of the orderings induced by some operators to avoid performing expensive sorts. This query requires selecting Posts and Comments based on their creation date, which might be correlated with their identifier and therefore, having intermediate results with interesting orders. Also, messages could be stored in an order correlated with their creation date to improve data access locality. Finally, as many of the attributes required in the projection are not needed for the execution of the query, it is expected that the query optimizer will move the projection to the end. |
| C 11 | | |
| C 12 | | |
| C 13 | | |
| C 14 | | |



params

| | | | |
|---|---|---|---|
| 1 | Person.id | ID | Person |
| 2 | date | DateTime | Date0 |

result

| | | | | |
|---|---|---|---|---|
| 1 | Message-hasCreator->Person.id | ID | R | personId |
| 2 | Message-hasCreator->Person.firstName | String | R | personFirstName |
| 3 | Message-hasCreator->Person.lastName | String | R | personLastName |
| 4 | Message.id | ID | R | postOrCommentId |
| 5 | Message.content or Post.imageFile | String | R | postOrCommentContent |
| 6 | Message.creationDate | DateTime | R | postOrCommentCreationDate |

sort

| | | |
|---|---|---|
| 1 | Message.creationDate | ↓ |
| 2 | Message.id | ↑ |

根据描述，此条查询的目的是给定一个起始的 people，寻找其朋友在给定的日期前发布的消息，返回的结果为 personId，personFirstName，personLastName，postOrCommentId，personOrCommentContent 和 postOrCommentCreationDate。ldbc_snb_driver 对每一个查询的返回结果定义了一个接收类，query2 的数据结构定义在 LdbcQuery2Result 中。查询语句如下：

```
query="g.V().has('Person.id', $id)."+
        "out('knows').as('friend').valueMap().as('x').in('hasCreato
    r').has('creationDate',P.lte($maxDate))."+
    "order().by('creationDate',decr).by('messageId',incr)."+
    "limit($limit).as('post').valueMap().as('y')"+
    ".select('x','y')\n";
```

通过

```
ResultSet resultSet = dbConnectionState.runQuery(query, parameters);
```

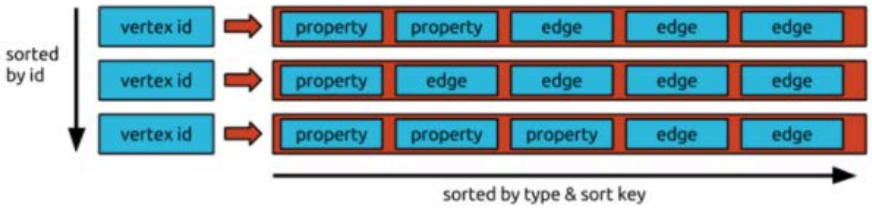
即可获得结果集，使用到的接口在 *org.apache.tinkerpop.gremlin* 中，具体实现可参考程序代码。

以下为 benchmark 给出的测试结果：


```
"unit" : "MILLISECONDS",
"throughput" : 13.820335636722607,
"all_metrics" : [ {
  "name" : "LdbcQuery1",
  "count" : 31,
  "unit" : "MILLISECONDS",
  "run_time" : {
    "name" : "Runtime",
    "unit" : "MILLISECONDS",
    "count" : 31,
    "mean" : 4.67741935483871,
    "min" : 3,
    "max" : 8,
    "25th_percentile" : 0,
    "50th_percentile" : 4,
    "75th_percentile" : 0,
    "90th_percentile" : 6,
    "95th_percentile" : 6,
    "99th_percentile" : 8,
    "99.9th_percentile" : 8,
    "std_dev" : 1.0282179000328533
  }
}, {
  "name" : "LdbcQuery2",
  "count" : 17,
  "unit" : "MILLISECONDS",
  "run_time" : {
    "name" : "Runtime",
    "unit" : "MILLISECONDS",
    "count" : 17,
    "mean" : 5.235294117647059,
    "min" : 3,
    "max" : 7,
    "25th_percentile" : 0,
    "50th_percentile" : 5,
    "75th_percentile" : 0,
    "90th_percentile" : 6,
    "95th_percentile" : 6,
    "99th_percentile" : 7,
    "99.9th_percentile" : 7,
    "std_dev" : 1.0017286097603766
  }
```


## 5.2 不同图划分模式

JanusGraph 以邻接表形式存储图数据，顶点的分配到机器上的方法就决定了图的分区。




图的分区主要有两种策略：默认策略和显示分区策略。

A) 默认策略

随机分区策略。随机安排顶点到所有机器上。缺点：查询效率慢，因为存在大量的跨实例的通信。

B) 显示分区策略

具有强关联性和经常访问的子图存储在相同的机器上，这样可以减少跨机器的通信成本。

- 配置参数

```
cluster.partition = true       //开启集群自定义分区策略
cluster.max-partitions = 32      //最大的虚拟分区数
ids.flush = false
```

- *max-partitions*：最大虚拟分区数，建议配置为存储数据个数的两倍。
- 精确分区只有支持 key 排序的存储后端
  - Hbase：支持图自定义分区
  - Cassandra：需要配置 ByteOrderedPartitioner 来致辞图分区
- 在图分区下有 edge cut 和 vertex cut 两方面可以单独控制。

1) Edge Cut（默认）

- 目的：对于频繁遍历的边，应该减少 cut edge 的存在，从而减少跨设备间的通信，提高查询效率。即把进行遍历的相邻顶点放在相同的分区，减少通信消耗。
- 为顶点确定分区：JanusGraph 通过配置好的 placement strategy 来控制 vertex-to-partition 的分配。
- 默认策略：在相同事务中创建的顶点分配在相同分区上。

- 缺点：如果在一个事务中加载大量数据，会导致分配不平衡。
- 定制分配策略：实现 IDPlacementStragegy 接口，并在通过配置文件的 ids.placement 项进行注册。

2) Vertex Cut

- Vertex Cut：顶点切割，即把一个顶点进行切割，把一个顶点的邻接表分成多个子邻接表存储在图中各个分区上。
- 目的：一个拥有大量边的顶点，在加载或者访问时会造成热点问题。Vertex Cut 通过分散压力到集群中所有实例从而缓解单顶点产生的负载。
- 方法：JanusGraph 通过 label 来切割顶点，通过定义 vertex label 成 partition，那么具有该 label 的所有顶点将被分配在集群中所有机器上。
- 案例：对于 product 和 user 顶点，product 顶点应该被定义为 partition，因为用户和商品有购买记录（edge），热销商品就会产生大量的购买记录，从而会造成热点问题。

```
mgmt = graph.openManagement()
mgmt.makeVertexLabel('user').make()      //正常vertex label
mgmt.makeVertexLabel('product').partition().make()  // 分区vertex label
mgmt.commit()
```

## 5.3 不同存储后端的性能比较

同样参数情况下对于不同的 Cassandra 集群测试结果如下：

Operation Count: 482

A) embedded Cassandra

Throughput: 65.78 (op/s)

Throughput: 65.92 (op/s)

Throughput: 65.92 (op/s)

Throughput: 65.59 (op/s)

B) Cassandra cluster 2 节点

Throughput: 24.34 (op/s)

Throughput: 66.05 (op/s)

Throughput: 65.89 (op/s)

Throughput: 65.96 (op/s)

C) Cassandra cluster 4 节点

Throughput: 21.35 (op/s)

Throughput: 45.25 (op/s)

Throughput: 66.05 (op/s)

Throughput: 48.00 (op/s)

由于测试图数据集较小结果差距不大，但多节点 Cassandra 集群 benchmark 稳定性较弱，这可能和集群之间需要较多通信有关。