香港中文大學計算機科學與工程學系
Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI5120 Phase 2 - Project 1 report

Group Member Information:

FU Junchen 1155164822

LIU Xingchen 1155150431

ZHANG Weidong 1155166098

DATE: Dec 21, 2021

Contents

Abstract

In this report, we explore the performances of four databases, namely Neo4j, JanusGraph, AgensGraph, and orientDB. We also design a new benchmark, RecGDBBench (Recommendation Graph Database Benchmark) based on the LDBC social network interactive dataset and schema, which is designed to identify the efficiency for Databases in terms of recommendation. The recommendation procedure we defined is from the interests discovery from one-hop neighbors and the aggregation of graph convolution neural networks. In our experiments, the RecGDBBench is tested on Neo4j, JanusGraph, and AgensGraph respectively. A comparison among these databases is drawn. The results show that AgenseGraph triumph on all queries. Neo4j has the similar performance to AgenseGraph. On the other hand, JanusGraph shows the worst performance.

1. Introduction

Nowadays, the graph data structure has become more and more popular. Many companies are applying graph databases for storing and querying the social network data they are using. However, there are many databases we could choose from, and how should we examine which database is the most suitable one? Benchmark could help us find the most appropriate database in a reasonable manner. The Linked Data Benchmark Council Benchmark is considered the state-of-the-art benchmark for testing the graph database in terms of social network data [1]. Therefore, we adopt the data schema from the LDBC and focus on the process in the recommendation. The RecGDBBench is designed and tested on several selected databases.

1.1 LDBC

In phase 1, we simply introduce the definition of the LDBC benchmark. It is a non-profit organization aiming to define standard graph benchmarks to foster a community around graph processing technologies.

We need to figure out some choke points of our benchmark. A benchmark is valuable if its workload stresses the important technical functionality of actual systems. This stress on elements of particular technical functionality we call choke points [1].

Choke point can be an important design element during benchmark definition. The overall goal of the choke-point based approach is to ensure that a benchmark workload covers a spectrum of technical challenges, forcing systems onto a path of technological innovation.

In our daily life, there are many apps where we can communicate with each other. And these pieces of information generate a social network. And the Social Network Benchmark (SNB) in LDBC is designed for evaluating a broad range of technologies for tackling graph data management workloads.

In this network, some people may have interests in specific areas and SNB includes a data generator that enables the creation of synthetic social network data with the following characteristics: the data schema is representative of a real social network see,

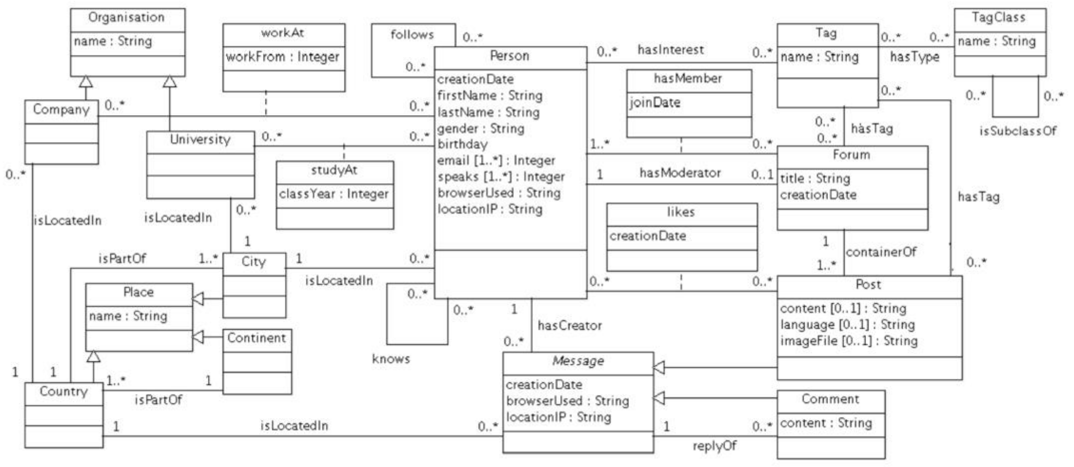


Figure 1: The data schema of the Social Network Benchmark represented in UML

We would implement the experiment based on this schema. And we can divide the process into 5 steps. Analysis, design, implementation, testing, and distribution. Every person has its properties like gender and location. First, we can select a specific tag, and get the number of people who have an interest in this tag, then we can have some analysis.

1.2 Neo4j

Neo4j is a kind of graph database being widely used because of its optimal management, storage, and traversal of nodes and relationships [2]. It mainly uses a linked list and pointer to store vertices, edges, and properties. Neo4j achieves extremely fast lookups and high space utilization, which is the main reason that makes it famous. Figure blew briefly shows how it stores graph data [3].

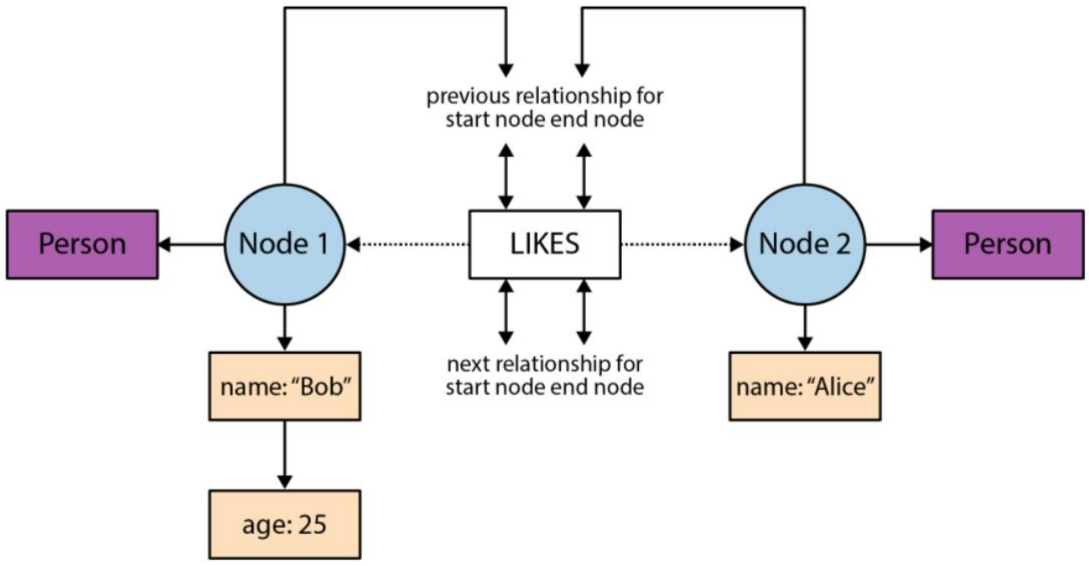


Figure 2 A graph physically stored in Neo4j

Cypher is Neo4j's graph query language that allows users to store and retrieve data from the graph database. Cypher's syntax allows users to match patterns of nodes and edges in a logistic and visual way. Users can filter entities with their properties and labels and extract nodes or edges from any part of the matched pattern.

Neo4j utilizes the index to improve the speed of traversal. there are three different index types: b-tree, full-text, and token lookup. All three types of indexes can be created and dropped using Cypher.

1.3 JanusGraph

The JanusGraph [4] is designed to support the scalability of processing a large amount of data. It adopts the Tinkerpop structure [5],

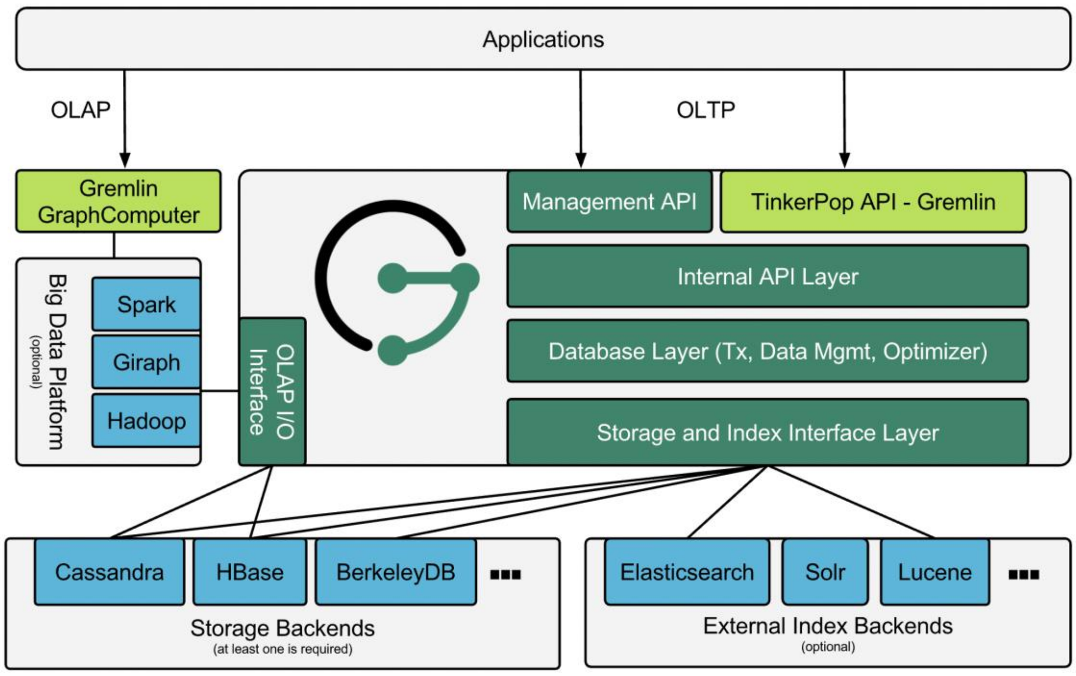


Figure 3 Tinkerpop structure

The storage backend relies on third-party platforms, namely Cassandra [6], HBase [7], and BerkeleyDB [8]. The in-memory method is also accepted, but the data does not persist after shutdown in that way, which means we could not save the data. Therefore, in our experiment, we tried the BerkelyDB and Cassandra as the storage backend, and the external index backend is Lucene. BerkeleyDB is integrated into the installation of the JanusGraph, meaning that we do not have to install it separately.

The JanusGraph stores graphs in an adjacency list which means that a graph is stored as a collection of vertices with their adjacency list. The adjacency list of a vertex contains all of the vertex's incident edges (and properties).

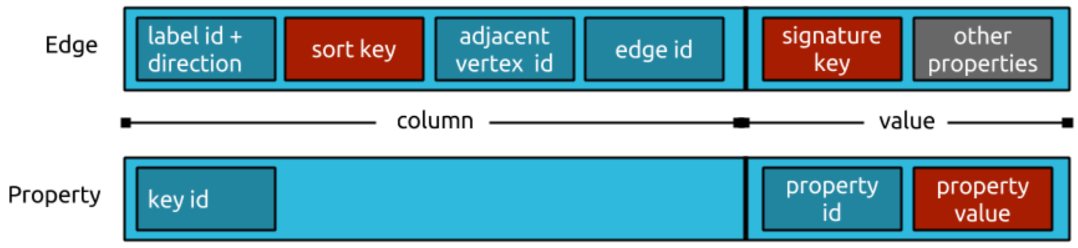


Figure 4 storage design of JanusGraph

By storing a graph in adjacency list format JanusGraph ensures that all of a vertex's incident edges and properties are stored compactly in the storage backend which speeds up traversals. The downside is that each edge has to be stored twice - once for each end vertex of the edge.

JanusGraph supports two different kinds of indexing to speed up query processing: graph indexes and vertex-centric indexes. Graph indexes are global index structures over the entire graph which allow efficient retrieval of vertices or edges by their properties for sufficiently selective conditions, which could be classified into Composite indexes and Mixed Indexes. In our experiment, we set the composite indexes on the “id” property for each vertex.

1.4 AgensGraph

Agensgraph is a multi-model database. It supports both the property graph model and the relational model. Vertex and edge in Agensgraph can have an arbitrary number of attributes and they can be well categorized with various labels, which are VLABEL and ELABEL respectively. The properties of vertex or edge are stored in many JSON files comprised of six data types: strings, numbers, booleans, null, objects, and arrays, which are suitable for the storage of key-value pairs for property.

Agensgraph is a multi-model database. It supports both the property graph model and the relational model. Vertex and edge in Agensgraph can have an arbitrary number of attributes and they can be well categorized with various labels, which are VLABEL and ELABEL respectively. The properties of vertex or edge are stored in many JSON files comprised of six data types: strings,

numbers, booleans, null, objects, and arrays, which are suitable for the storage of key-value pairs for property.

The data model is shown in the below figure. Each database can contain one or more schemas and graphs. Schemas are for relational tables, and graph objects are for graph data.

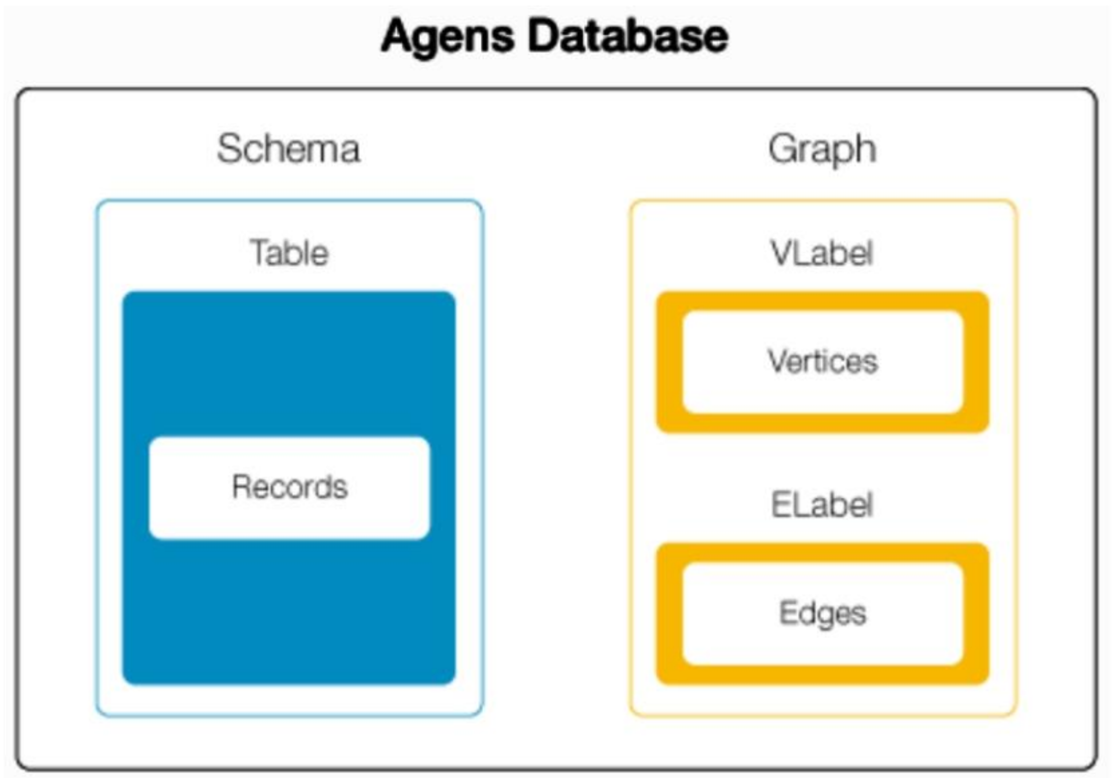


Figure 5 AgensGraph Data model

Agensgraph uses LABEL to manage vertices and edges and it also provides inheritance of label for the hierarchies of many labels. Figure 5 is an example of this mechanism.

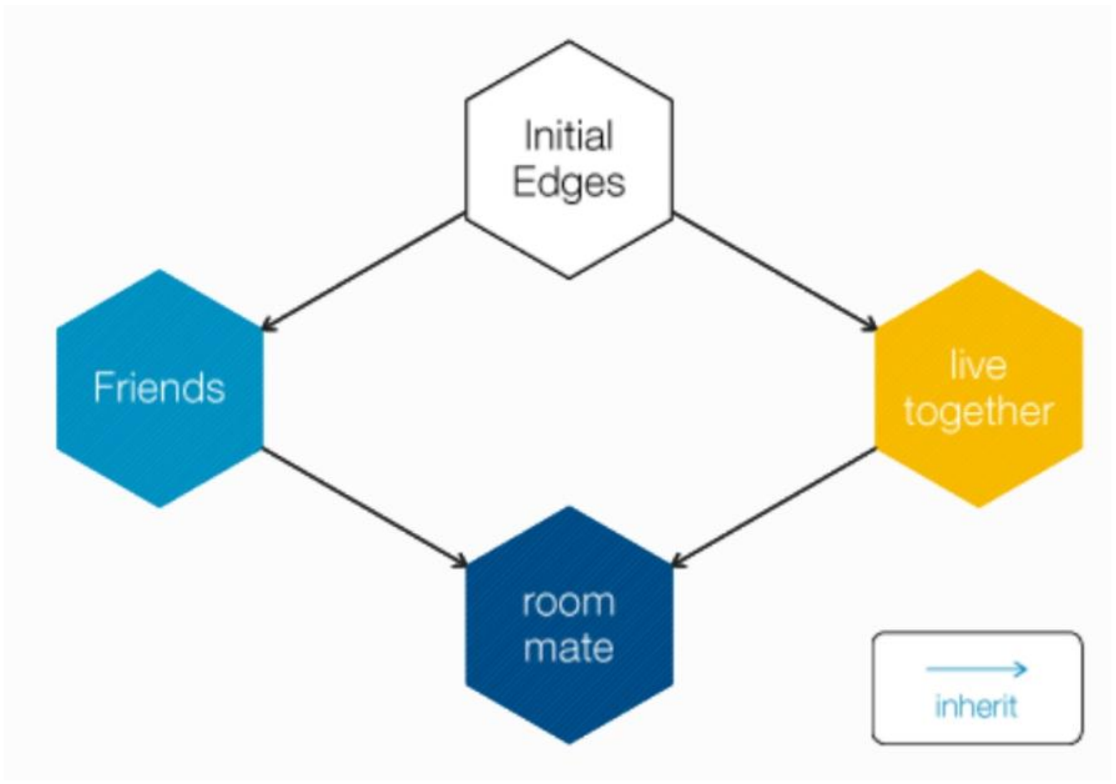


Figure 6 Label inheritance example

AgensGraph supports SQL for relational data queries and Cypher for graph data queries. It even supports hybrid query language, which allows mixed SQL and Cypher query statements. Cypher is a more declarative language for graph datasets since its syntax visually describes the patterns found in graphs.

1.5 OrientDB

OrientDB [9] is a multi-model open-source NoSQL database management system that supports data models in documents, graphs, key-value pairs, and objects. OrientDB was engineered from the ground up with performance as a key specification. It’s fast on both read and write operations. It has no more Joins command and the relationships are physical links to the records.

Here we mainly introduce the graph model we will test. A graph represents a network-like structure consisting of vertexes, interconnected by edges. OrientDB's graph model is represented by the concept of a property graph.

Vertex is an entity that can be linked with other vertexes and has these mandatory properties: a unique identifier, set of incoming Edges, set of outgoing Edges.

Edge is an entity that links two vertexes and has these mandatory properties: a unique identifier, link to an incoming Vertex, link to an outgoing Vertex, label that defines the type of connection between head and tail vertex.
In addition to mandatory properties, each vertex or edge can also hold a set of custom properties. These properties can be defined by users, which can make vertexes and edges appear similar to documents.

2. Benchmark Design

The RecGDBBench is mainly defined for testing the performance of doing a recommendation for the person vertexes in the social network. The main graph schema for the experiment is inspired by [10]. We combine this and the LDBC benchmark. The data schema we use in our benchmark is below,

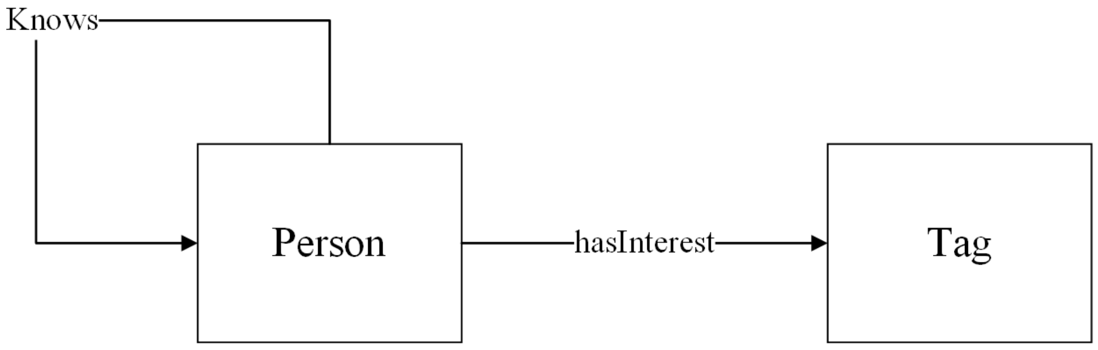


Figure 7 Schema of RecGDBBench

Therefore, the recommendation task is to recommend potential Tags for person vertexes. The whole procedure we applied in our benchmark can be simply classified into interests discovering and aggregation of the Graph neural network procedure.

The first workload is the query for the tags of the one-hop neighbor for the targeted vertex. We will refer to this type of query as “type one query”. This query can be expressed with Cypher, *MATCH(p1:Person{id:$targetId})-*

[:KNOWS]->(p2:Person{"filterProperties":"properties"})-

[:HAS_INTEREST]->(t:Tag)

RETURN

t.id AS tagId,

t.name AS tagName

The second type of query is the simulation of weight aggregation in the Graph Neural Networks to acquire the topology structure and information of the adjacency neighbors. The aggregation is namely the "Encoder" structure of the neural networks. Although the encoder training procedure is mainly on the main memory and copy them to GPU, FPGA, etc, and not many of them are done directly from the databases according to the query languages, we could rule out the possibility that some developers and researchers may want to conduct that procedure directly on the database. The aggregation method we apply in our report is the "collect". In our report, we adopt the two-hop vertexes information collection for the target person. We will refer to this type of query as "type two query". The Cypher expression is below,

MATCH(p1:Person{id:$targetId})-

[:KNOWS]->(p2:Person{"filterProperties":"properties"})-

[:KNOWS]->(p3:Person{“filterProperties”:”properties”})

RETURN

collect(p2.id),

collect(p3.id);

The dataset to test is the LDBC social network dataset since it contains the structure we require. We try to run our benchmark on four systems and manage to finish three of them. We wrote the benchmark in Gremlin and Cypher and select three representative vertexes as the targeted vertexes. They represent a high, middle, low level of out degrees of “KNOWS” edges. The detail can be checked in the Appendix.

3. Experiment

We conduct this experiment on the CUHK CSE project 10 cluster. The main information about this cluster is shown below,

CPU: Two Intel(R) Xeon(R) E5-2620 v3 @ 2.40GHz, 6 cores

RAM: 64GB

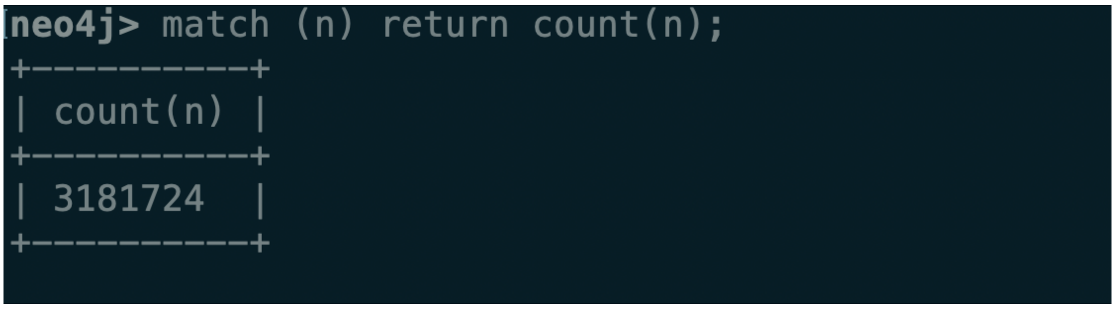
System: CentOS Linux release 7.9

This section mainly presents the deployment of the systems, data importing, and benchmark experiments on Neo4j, JanusGraph, and AgenseGraph.

3.1 Neo4j

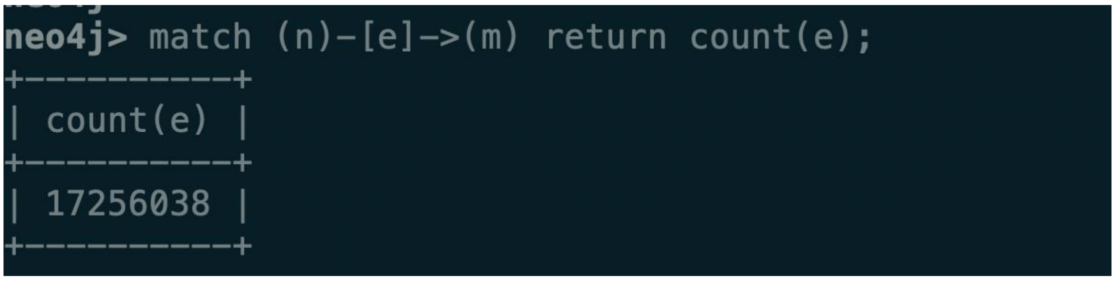
In this experiment we adopt the same version of Neo4j 3.4.5, therefore, the deployment is similar to the phase one report. We make a trial run on the Neo4j 4.1.1, however, that version does not show stable status. Our type two query cannot be run successfully on that version of Neo4j.

The dataset we adopt is the social network dataset from the LDBC benchmark. This dataset is bigger compared with the dataset from the phase one report. The number of vertexes is shown below,



```
neo4j> match (n) return count(n);
+-----------+
| count(n)  |
+-----------+
| 3181724   |
+-----------+
```

The number of edges is,

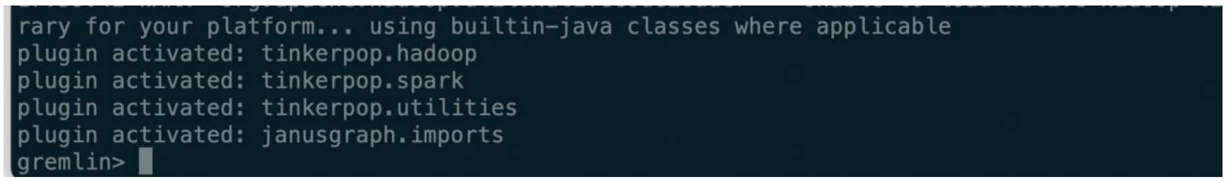


```
neo4j> match (n)-[e]->(m) return count(e);
+-----------+
| count(e)  |
+-----------+
| 17256038  |
+-----------+
```

After the experiment, we find the importing method of Neo4j is incredibly convenient. Since our dataset is too large, and most databases do not support inherent loading tools. Therefore, the loading procedure is so slow or even not able to load a huge dataset into it. However, the Neo4j shows favorable performance in terms of importing datasets. The integrated "Neo4j-admin" shows remarkable loading speed could load these data within a minute.

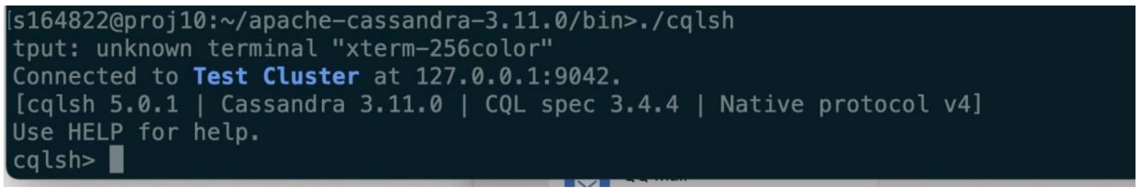
3.2 JanusGraph

The deployment of JanusGraph is not very difficult. We implement the 0.5.2 version of JanusGraph because the 0.5.x version is the most stable. Since this database adopts the Tinkerpop architecture, we could use the "gremlin.sh" to connect to the databases.



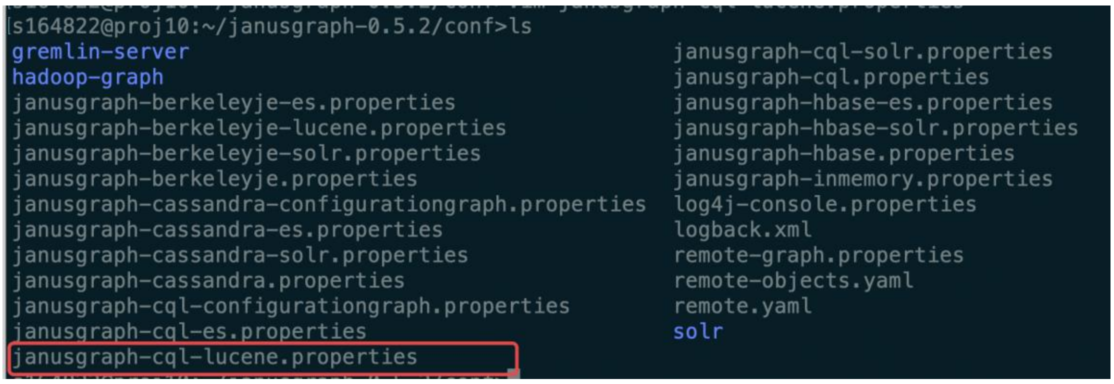
```
rary for your platform... using builtin-java classes where applicable
plugin activated: tinkerpop.hadoop
plugin activated: tinkerpop.spark
plugin activated: tinkerpop.utilities
plugin activated: janusgraph.imports
gremlin>
```

The storage backend we use in the experiment is the Cassandra database. We use the 3.11.0 version as the storage backend. The Cassandra has its own query language called cql. We could use the cqlsh under the bin folder.



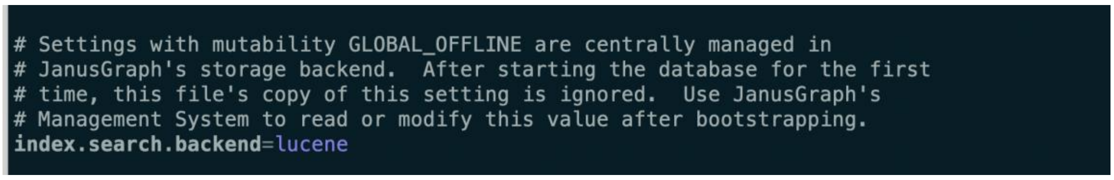
```
s164822@proj10:~/apache-cassandra-3.11.0/bin>./cqlsh
tput: unknown terminal "xterm-256color"
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.0 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh>
```

We would like our index engine to be "lucene", the properties file needs to be created,



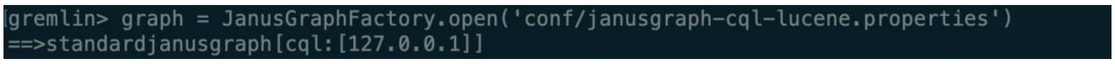
```
s164822@proj10:~/janusgraph-0.5.2/conf>ls
gremlin-server                                         janusgraph-cql-solr.properties
hadoop-graph                                           janusgraph-cql.properties
janusgraph-berkeleyje-es.properties                    janusgraph-hbase-es.properties
janusgraph-berkeleyje-lucene.properties                janusgraph-hbase-solr.properties
janusgraph-berkeleyje-solr.properties                  janusgraph-hbase.properties
janusgraph-berkeleyje.properties                       janusgraph-inmemory.properties
janusgraph-cassandra-configurationgraph.properties     log4j-console.properties
janusgraph-cassandra-es.properties                     logback.xml
janusgraph-cassandra-solr.properties                   remote-graph.properties
janusgraph-cassandra.properties                        remote-objects.yaml
janusgraph-cql-configurationgraph.properties           remote.yaml
janusgraph-cql-es.properties                           solr
janusgraph-cql-lucene.properties
```

The difference between the file we create and the "cql-es" or "cql", is that we add one setting, which specifies the backend.



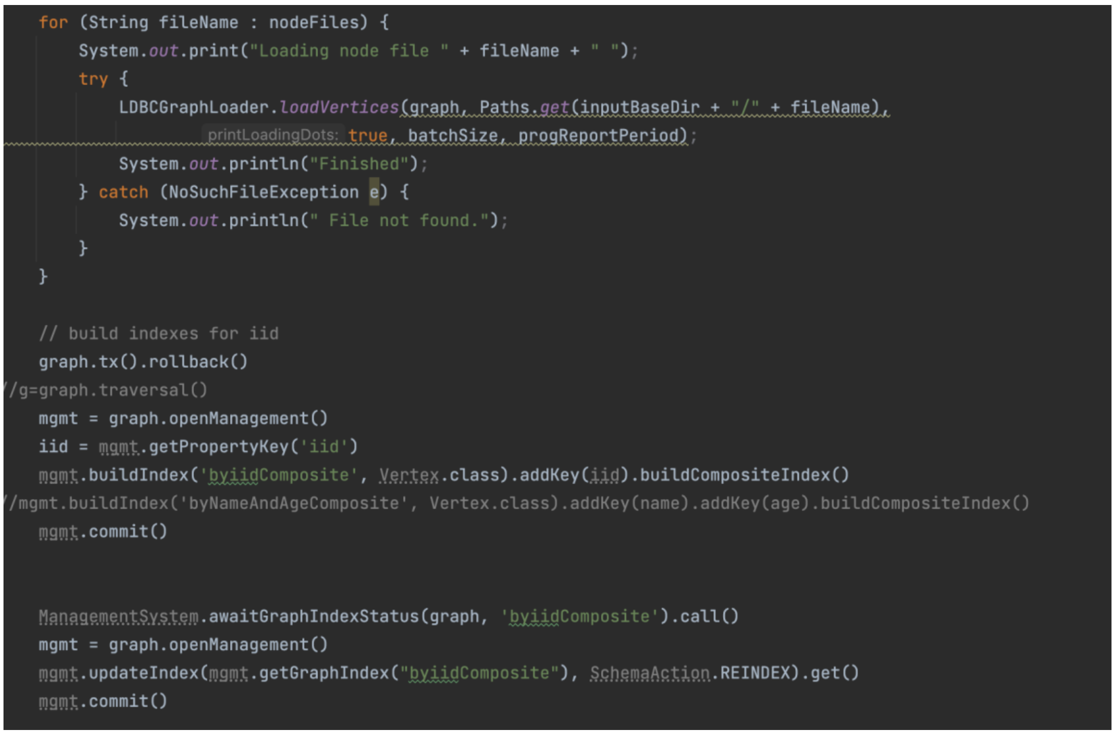
```
# Settings with mutability GLOBAL_OFFLINE are centrally managed in
# JanusGraph's storage backend.  After starting the database for the first
# time, this file's copy of this setting is ignored.  Use JanusGraph's
# Management System to read or modify this value after bootstrapping.
index.search.backend=lucene
```

After the configuration, the graph can be acquired directly from this one instruction,



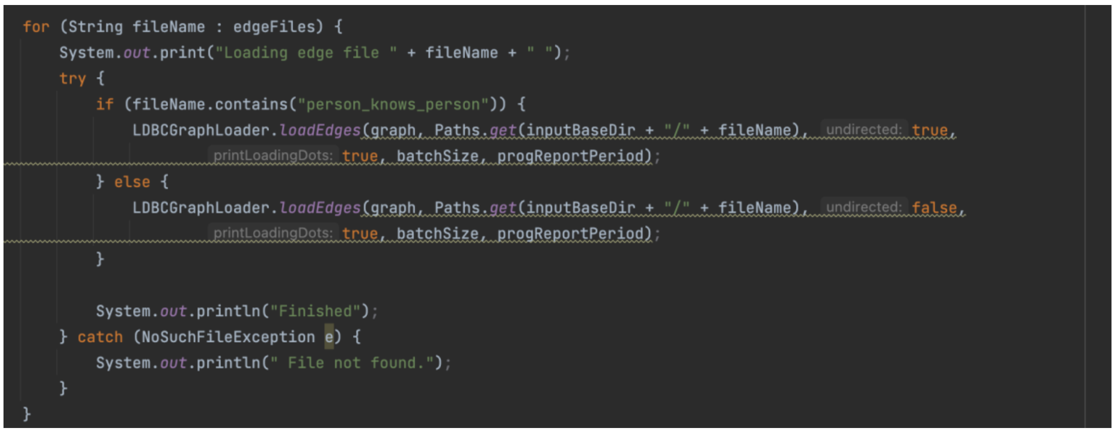
```
gremlin> graph = JanusGraphFactory.open('conf/janusgraph-cql-lucene.properties')
==>standardjanusgraph[cql:[127.0.0.1]]
```

The data loading procedure is quite troublesome in JanusGraph, especially for the CSV file and large-size dataset. The most common method for the loading CSV file to this database is to apply a groovy script since the "gremlin.sh" can be directly used to execute the groovy language. Our experiment also adopts this method. The loading procedure can be roughly split into four procedures, namely loading vertexes, building indexes on the vertex id property, loading edges. Here is the part of the code for loading the vertexes and building indexes,



```
for (String fileName : nodeFiles) {
    System.out.print("Loading node file " + fileName + " ");
    try {
        LDBCGraphLoader.loadVertices(graph, Paths.get(inputBaseDir + "/" + fileName),
                printLoadingDots: true, batchSize, progReportPeriod);
        System.out.println("Finished");
    } catch (NoSuchFileException e) {
        System.out.println(" File not found.");
    }
}

// build indexes for iid
graph.tx().rollback()
/g=graph.traversal()
mgmt = graph.openManagement()
iid = mgmt.getPropertyKey('iid')
mgmt.buildIndex('byiidComposite', Vertex.class).addKey(iid).buildCompositeIndex()
/mgmt.buildIndex('byNameAndAgeComposite', Vertex.class).addKey(name).addKey(age).buildCompositeIndex()
mgmt.commit()


ManagementSystem.awaitGraphIndexStatus(graph, 'byiidComposite').call()
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getGraphIndex("byiidComposite"), SchemaAction.REINDEX).get()
mgmt.commit()
```

Here is the main function for loading the edges.

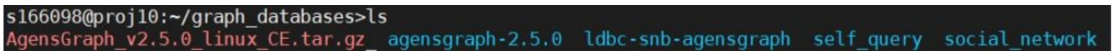


```
for (String fileName : edgeFiles) {
    System.out.print("Loading edge file " + fileName + " ");
    try {
        if (fileName.contains("person_knows_person")) {
            LDBCGraphLoader.loadEdges(graph, Paths.get(inputBaseDir + "/" + fileName),  undirected: true,
                    printLoadingDots: true, batchSize, progReportPeriod);
        } else {
            LDBCGraphLoader.loadEdges(graph, Paths.get(inputBaseDir + "/" + fileName),  undirected: false,
                    printLoadingDots: true, batchSize, progReportPeriod);
        }

        System.out.println("Finished");
    } catch (NoSuchFileException e) {
        System.out.println(" File not found.");
    }
}
```

The detailed implementation of the functions can be found in the attachment files under the JanusGraph directory. This procedure took us 5-6 hours to finish loading. Therefore, the loading procedure for CSV should really be optimized like Neo4j's “Neo4j-admin import”, otherwise, this procedure is undisputedly time-consuming. The groovy file is derived from [11]

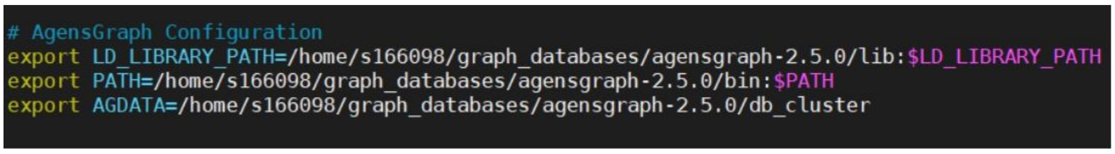
3.3 AgensGraph

Download the pre-compiled binary package from AgensGraph's website and decompress the package.



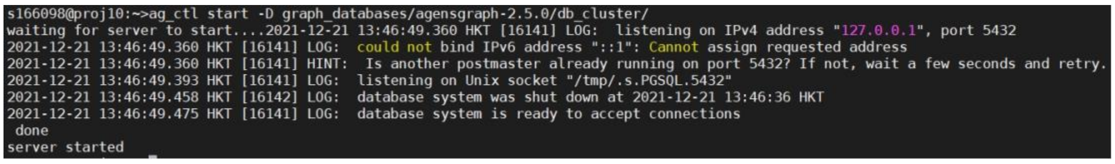
```
s166098@proj10:~/graph_databases>ls
AgensGraph_v2.5.0_linux_CE.tar.gz  agensgraph-2.5.0  ldbc-snb-agensgraph  self_query  social_network
```

Configurate some corresponding environmental variables by adding the following commands into a shell start-up file .bashrc



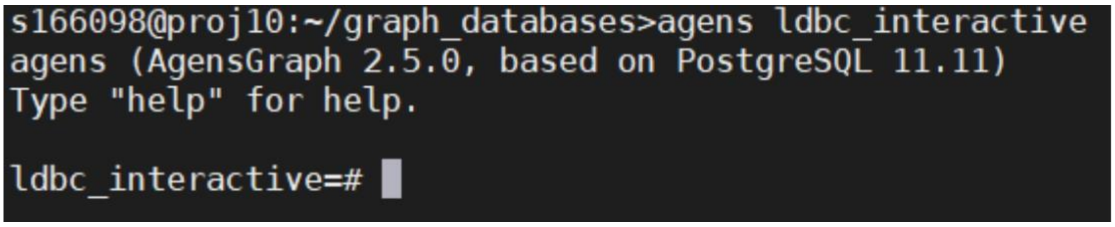
```
# AgensGraph Configuration
export LD_LIBRARY_PATH=/home/s166098/graph_databases/agensgraph-2.5.0/lib:$LD_LIBRARY_PATH
export PATH=/home/s166098/graph_databases/agensgraph-2.5.0/bin:$PATH
export AGDATA=/home/s166098/graph_databases/agensgraph-2.5.0/db_cluster
```

Creating a database cluster and starting the AgensGraph's server



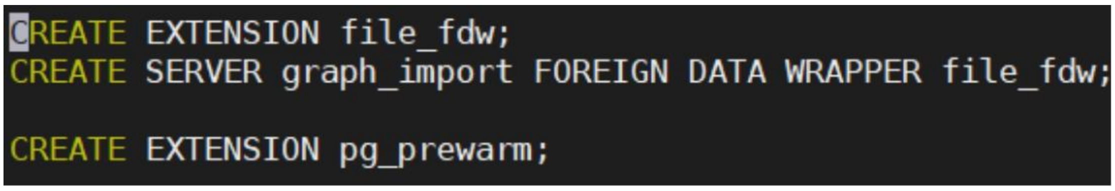
```
s166098@proj10:~>ag_ctl start -D graph_databases/agensgraph-2.5.0/db_cluster/
waiting for server to start....2021-12-21 13:46:49.360 HKT [16141] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2021-12-21 13:46:49.360 HKT [16141] LOG:  could not bind IPv6 address "::1": Cannot assign requested address
2021-12-21 13:46:49.360 HKT [16141] HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds and retry.
2021-12-21 13:46:49.393 HKT [16141] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
2021-12-21 13:46:49.458 HKT [16142] LOG:  database system was shut down at 2021-12-21 13:46:36 HKT
2021-12-21 13:46:49.475 HKT [16141] LOG:  database system is ready to accept connections
 done
server started
```

The creating the database with a name and using the command "agens" can access the console.

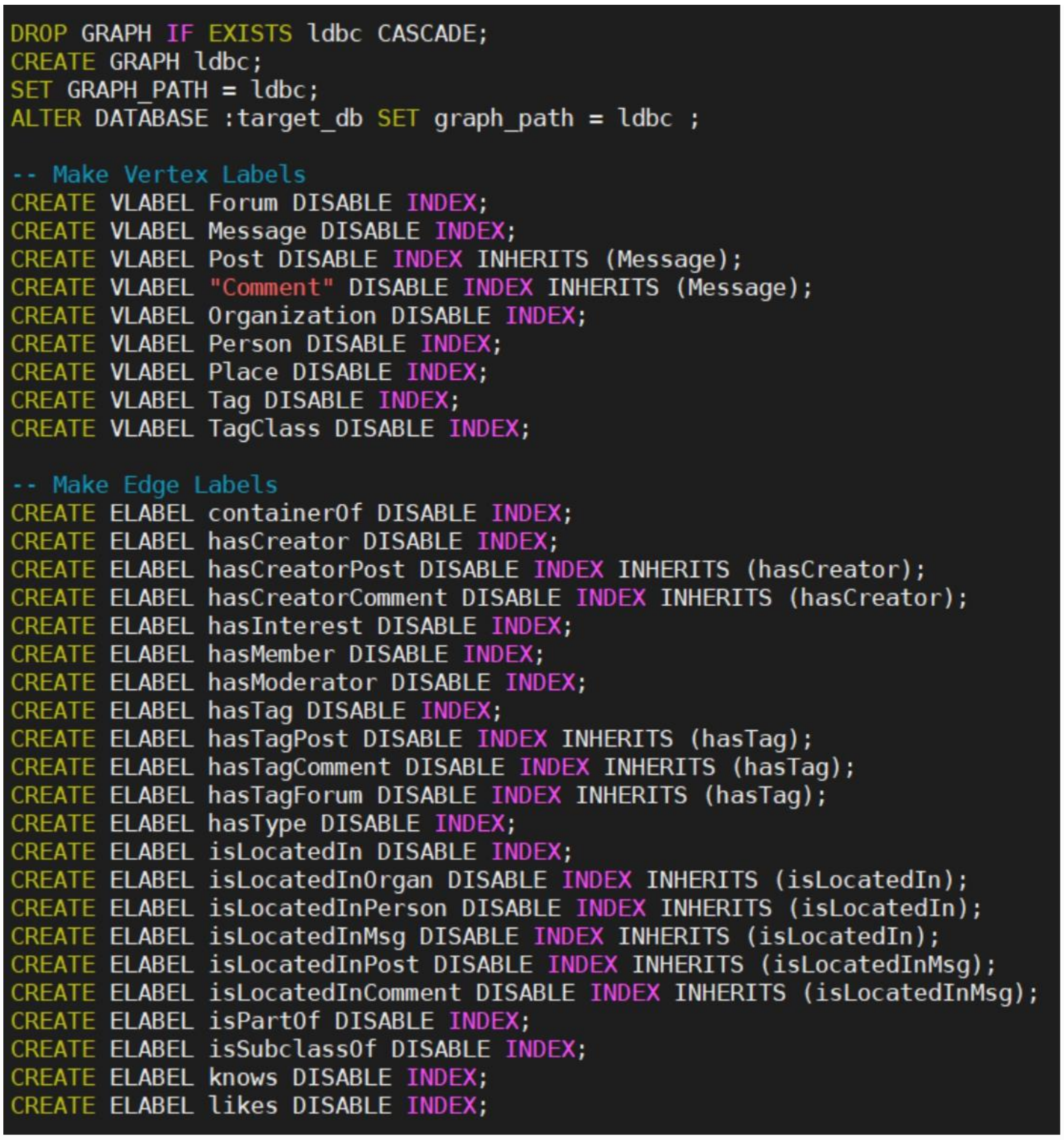


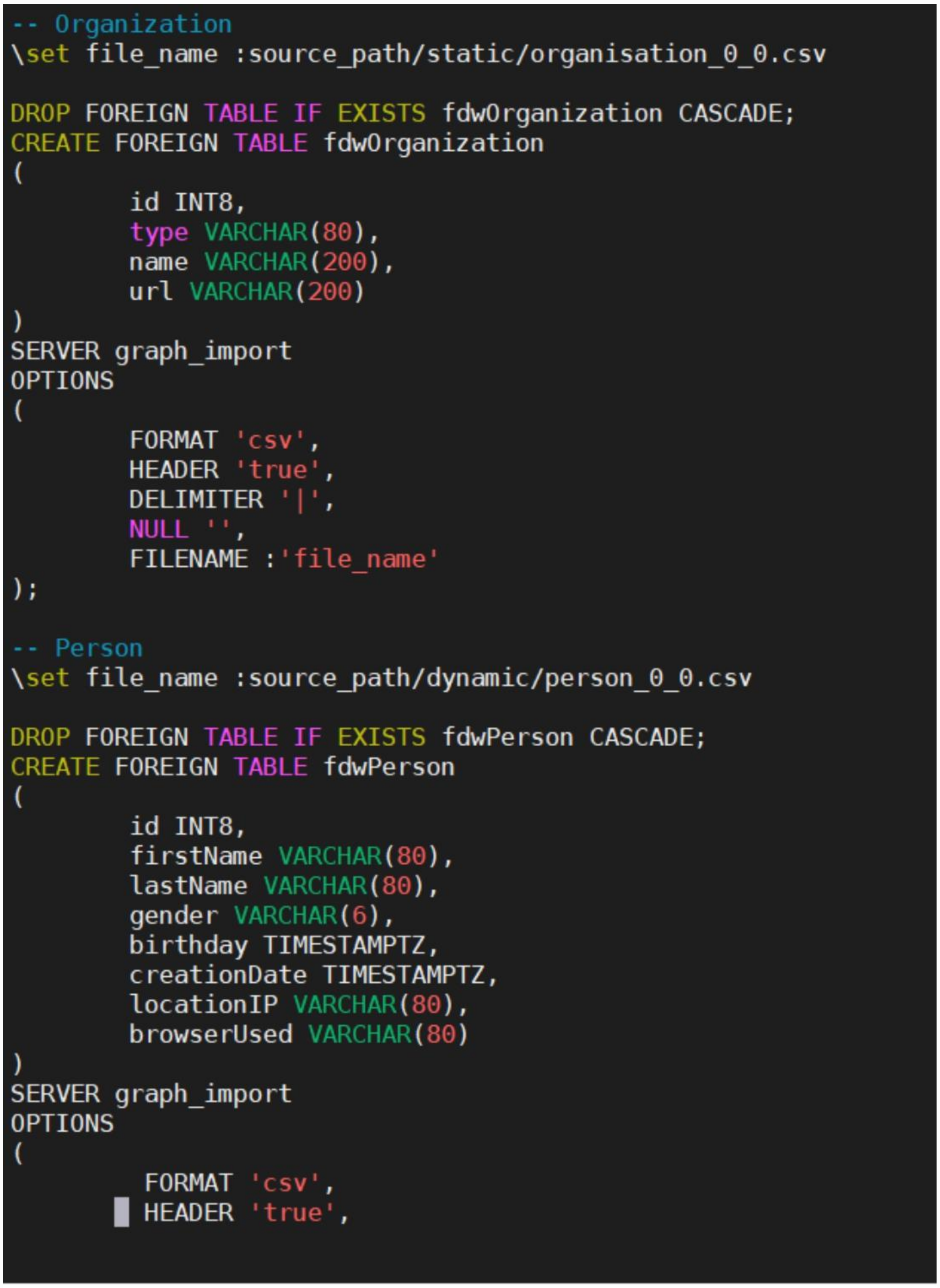
```
s166098@proj10:~/graph_databases>agens ldbc_interactive
agens (AgensGraph 2.5.0, based on PostgreSQL 11.11)
Type "help" for help.

ldbc_interactive=#
```

Prepare the social network datasets generated by the LDBC data generator firstly. Install the extension 'file_fdw' necessary to use the AgensGraph foreign-data wrapper to interface with files on the server's filesystem. And create data import server.


```
CREATE EXTENSION file_fdw;
CREATE SERVER graph_import FOREIGN DATA WRAPPER file_fdw;

CREATE EXTENSION pg_prewarm;
```

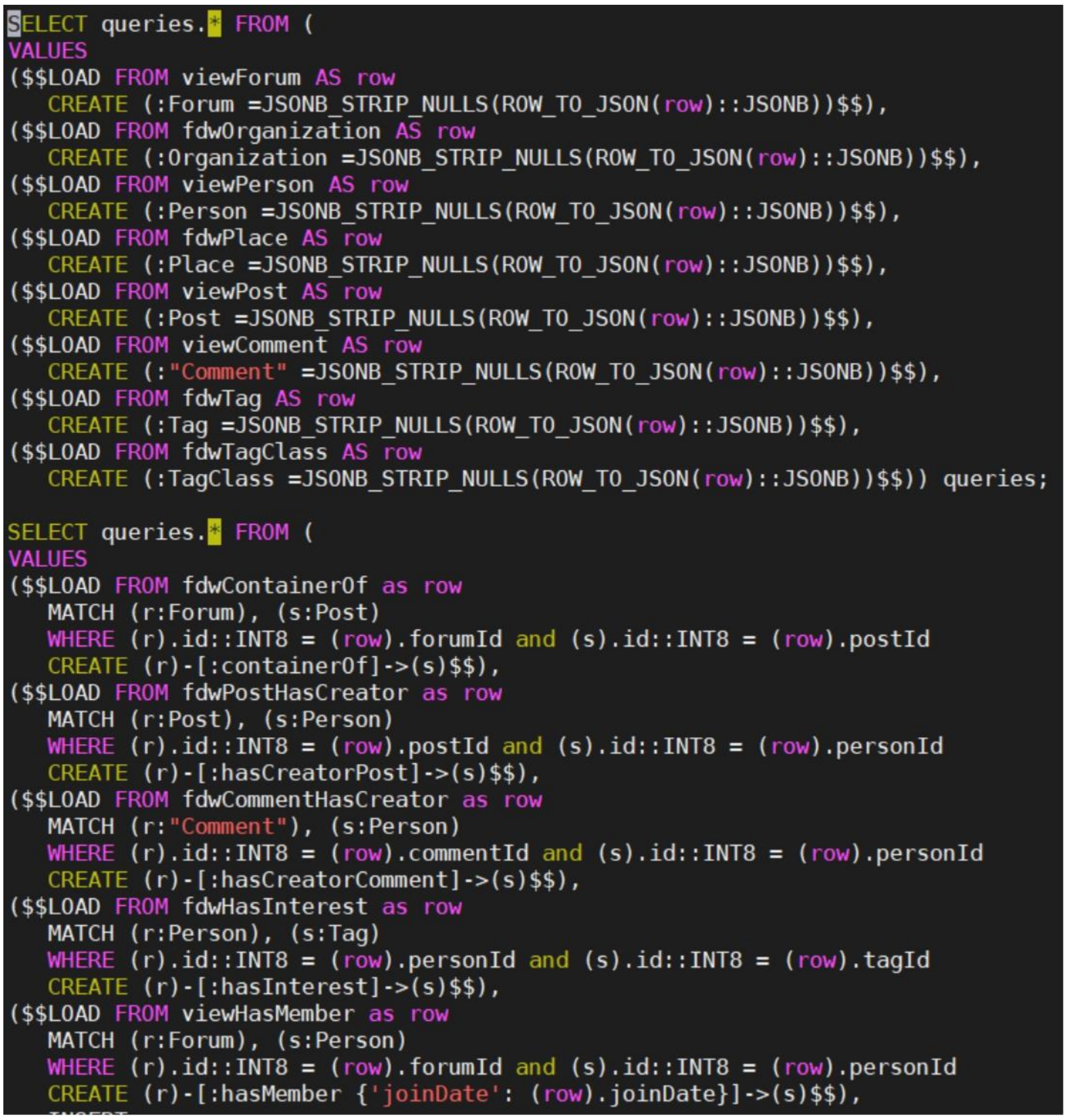
Create the Label for the vertices and edges. Using a foreign table to receive data from a foreign file. The following figure shows a part of this procedure.


```
DROP GRAPH IF EXISTS ldbc CASCADE;
CREATE GRAPH ldbc;
SET GRAPH_PATH = ldbc;
ALTER DATABASE :target_db SET graph_path = ldbc ;

-- Make Vertex Labels
CREATE VLABEL Forum DISABLE INDEX;
CREATE VLABEL Message DISABLE INDEX;
CREATE VLABEL Post DISABLE INDEX INHERITS (Message);
CREATE VLABEL "Comment" DISABLE INDEX INHERITS (Message);
CREATE VLABEL Organization DISABLE INDEX;
CREATE VLABEL Person DISABLE INDEX;
CREATE VLABEL Place DISABLE INDEX;
CREATE VLABEL Tag DISABLE INDEX;
CREATE VLABEL TagClass DISABLE INDEX;

-- Make Edge Labels
CREATE ELABEL containerOf DISABLE INDEX;
CREATE ELABEL hasCreator DISABLE INDEX;
CREATE ELABEL hasCreatorPost DISABLE INDEX INHERITS (hasCreator);
CREATE ELABEL hasCreatorComment DISABLE INDEX INHERITS (hasCreator);
CREATE ELABEL hasInterest DISABLE INDEX;
CREATE ELABEL hasMember DISABLE INDEX;
CREATE ELABEL hasModerator DISABLE INDEX;
CREATE ELABEL hasTag DISABLE INDEX;
CREATE ELABEL hasTagPost DISABLE INDEX INHERITS (hasTag);
CREATE ELABEL hasTagComment DISABLE INDEX INHERITS (hasTag);
CREATE ELABEL hasTagForum DISABLE INDEX INHERITS (hasTag);
CREATE ELABEL hasType DISABLE INDEX;
CREATE ELABEL isLocatedIn DISABLE INDEX;
CREATE ELABEL isLocatedInOrgan DISABLE INDEX INHERITS (isLocatedIn);
CREATE ELABEL isLocatedInPerson DISABLE INDEX INHERITS (isLocatedIn);
CREATE ELABEL isLocatedInMsg DISABLE INDEX INHERITS (isLocatedIn);
CREATE ELABEL isLocatedInPost DISABLE INDEX INHERITS (isLocatedInMsg);
CREATE ELABEL isLocatedInComment DISABLE INDEX INHERITS (isLocatedInMsg);
CREATE ELABEL isPartOf DISABLE INDEX;
CREATE ELABEL isSubclassOf DISABLE INDEX;
CREATE ELABEL knows DISABLE INDEX;
CREATE ELABEL likes DISABLE INDEX;
```



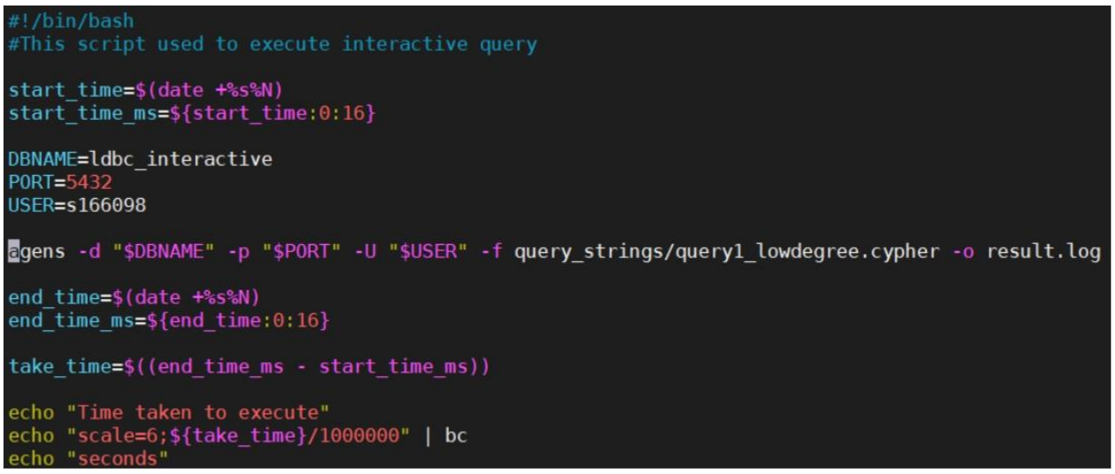
```
-- Organization
\set file_name :source_path/static/organisation_0_0.csv

DROP FOREIGN TABLE IF EXISTS fdwOrganization CASCADE;
CREATE FOREIGN TABLE fdwOrganization
(
        id INT8,
        type VARCHAR(80),
        name VARCHAR(200),
        url VARCHAR(200)
)
SERVER graph_import
OPTIONS
(
        FORMAT 'csv',
        HEADER 'true',
        DELIMITER '|',
        NULL '',
        FILENAME :'file_name'
);

-- Person
\set file_name :source_path/dynamic/person_0_0.csv

DROP FOREIGN TABLE IF EXISTS fdwPerson CASCADE;
CREATE FOREIGN TABLE fdwPerson
(
        id INT8,
        firstName VARCHAR(80),
        lastName VARCHAR(80),
        gender VARCHAR(6),
        birthday TIMESTAMPTZ,
        creationDate TIMESTAMPTZ,
        locationIP VARCHAR(80),
        browserUsed VARCHAR(80)
)
SERVER graph_import
OPTIONS
(
         FORMAT 'csv',
         HEADER 'true',
```

Execute the import. The data must be cast as type JSONB since vertices and edges in AgensGraph are stored in the JSONB format. The following figure shows a part of this procedure. Then the vertices and edges are all be loaded into the graph in the database.



```
SELECT queries.* FROM (
VALUES
($$LOAD FROM viewForum AS row
   CREATE (:Forum =JSONB_STRIP_NULLS(ROW_TO_JSON(row)::JSONB))$$),
($$LOAD FROM fdwOrganization AS row
   CREATE (:Organization =JSONB_STRIP_NULLS(ROW_TO_JSON(row)::JSONB))$$),
($$LOAD FROM viewPerson AS row
   CREATE (:Person =JSONB_STRIP_NULLS(ROW_TO_JSON(row)::JSONB))$$),
($$LOAD FROM fdwPlace AS row
   CREATE (:Place =JSONB_STRIP_NULLS(ROW_TO_JSON(row)::JSONB))$$),
($$LOAD FROM viewPost AS row
   CREATE (:Post =JSONB_STRIP_NULLS(ROW_TO_JSON(row)::JSONB))$$),
($$LOAD FROM viewComment AS row
   CREATE (:"Comment" =JSONB_STRIP_NULLS(ROW_TO_JSON(row)::JSONB))$$),
($$LOAD FROM fdwTag AS row
   CREATE (:Tag =JSONB_STRIP_NULLS(ROW_TO_JSON(row)::JSONB))$$),
($$LOAD FROM fdwTagClass AS row
   CREATE (:TagClass =JSONB_STRIP_NULLS(ROW_TO_JSON(row)::JSONB))$$)) queries;

SELECT queries.* FROM (
VALUES
($$LOAD FROM fdwContainerOf as row
   MATCH (r:Forum), (s:Post)
   WHERE (r).id::INT8 = (row).forumId and (s).id::INT8 = (row).postId
   CREATE (r)-[:containerOf]->(s)$$),
($$LOAD FROM fdwPostHasCreator as row
   MATCH (r:Post), (s:Person)
   WHERE (r).id::INT8 = (row).postId and (s).id::INT8 = (row).personId
   CREATE (r)-[:hasCreatorPost]->(s)$$),
($$LOAD FROM fdwCommentHasCreator as row
   MATCH (r:"Comment"), (s:Person)
   WHERE (r).id::INT8 = (row).commentId and (s).id::INT8 = (row).personId
   CREATE (r)-[:hasCreatorComment]->(s)$$),
($$LOAD FROM fdwHasInterest as row
   MATCH (r:Person), (s:Tag)
   WHERE (r).id::INT8 = (row).personId and (s).id::INT8 = (row).tagId
   CREATE (r)-[:hasInterest]->(s)$$),
($$LOAD FROM viewHasMember as row
   MATCH (r:Forum), (s:Person)
   WHERE (r).id::INT8 = (row).forumId and (s).id::INT8 = (row).personId
   CREATE (r)-[:hasMember {'joinDate': (row).joinDate}]->(s)$$),
```

I write a bash script, using the command "agens" and link to the console with declared user, database name, and service port.



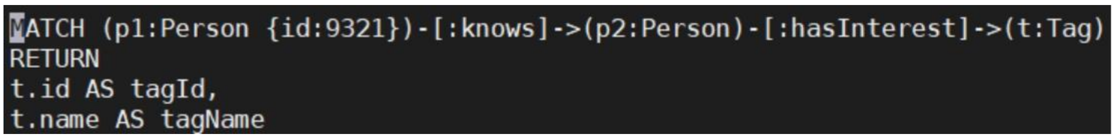
```
#!/bin/bash
#This script used to execute interactive query

start_time=$(date +%s%N)
start_time_ms=${start_time:0:16}

DBNAME=ldbc_interactive
PORT=5432
USER=s166098

agens -d "$DBNAME" -p "$PORT" -U "$USER" -f query_strings/query1_lowdegree.cypher -o result.log

end_time=$(date +%s%N)
end_time_ms=${end_time:0:16}

take_time=$((end_time_ms - start_time_ms))

echo "Time taken to execute"
echo "scale=6;${take_time}/1000000" | bc
echo "seconds"
```

Then running 4 kinds of queries in the type of cypher scripts. For the beginning vertices, we choose three different degrees in high, medium, and low respectively. The result is shown below

The first kind of query is to execute one-hop traversal without filter.

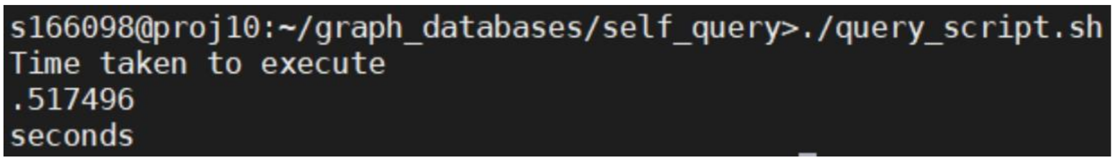
Low degree

Cypher script



```
MATCH (p1:Person {id:9321})-[:knows]->(p2:Person)-[:hasInterest]->(t:Tag)
RETURN
t.id AS tagId,
t.name AS tagName
```

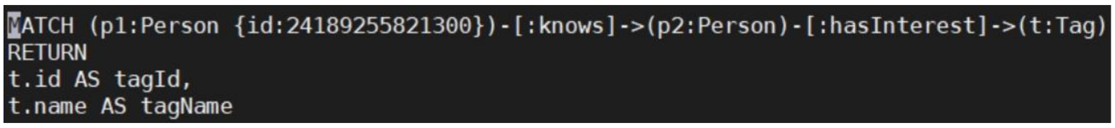
Result



```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.517496
seconds
```

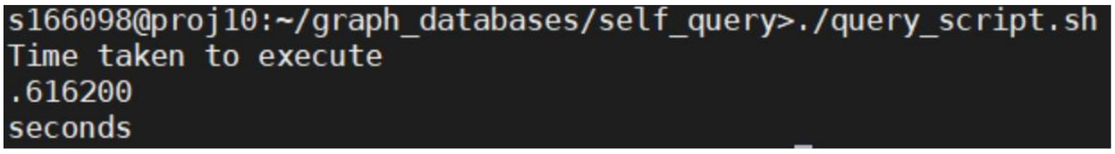
Medium degree

Cypher script



```
MATCH (p1:Person {id:24189255821300})-[:knows]->(p2:Person)-[:hasInterest]->(t:Tag)
RETURN
t.id AS tagId,
t.name AS tagName
```

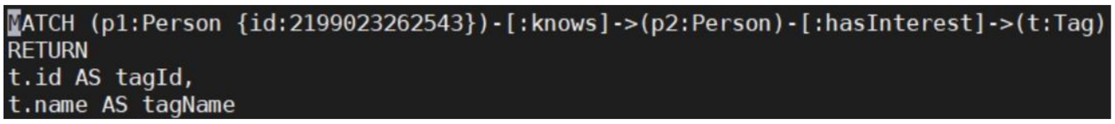
Result



```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.616200
seconds
```

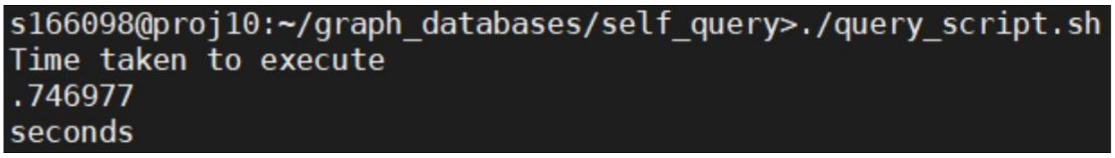
High degree

Cypher script



```
MATCH (p1:Person {id:2199023262543})-[:knows]->(p2:Person)-[:hasInterest]->(t:Tag)
RETURN
t.id AS tagId,
t.name AS tagName
```

Result

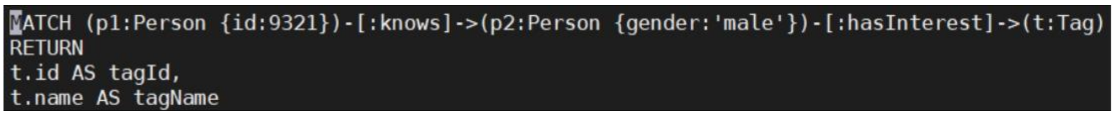


```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.746977
seconds
```

The second kind of query is to execute one-hop traversal with filter.

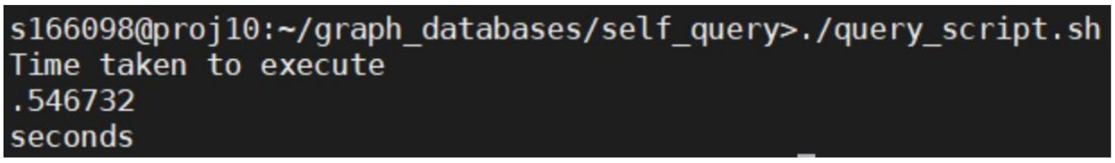
Low degree

Cypher script



```
MATCH (p1:Person {id:9321})-[:knows]->(p2:Person {gender:'male'})-[:hasInterest]->(t:Tag)
RETURN
t.id AS tagId,
t.name AS tagName
```

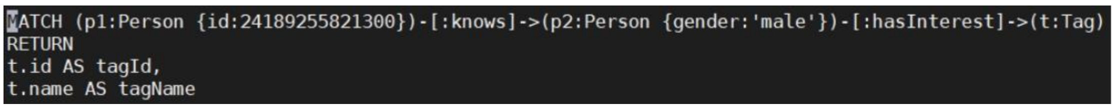
Result



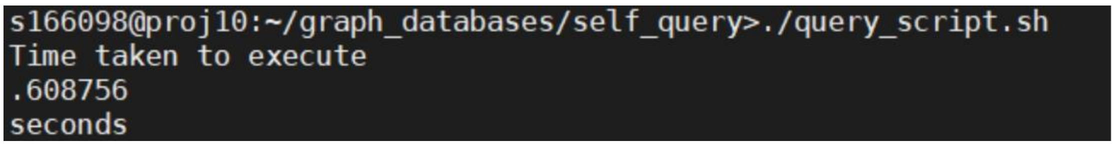
```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.546732
seconds
```

Medium degree

Cypher script

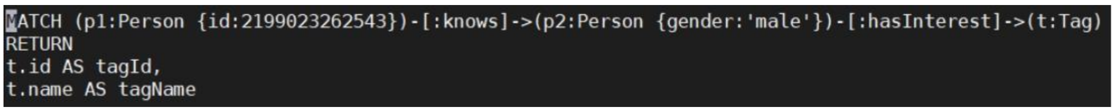

```
MATCH (p1:Person {id:24189255821300})-[:knows]->(p2:Person {gender:'male'})-[:hasInterest]->(t:Tag)
RETURN
t.id AS tagId,
t.name AS tagName
```

Result

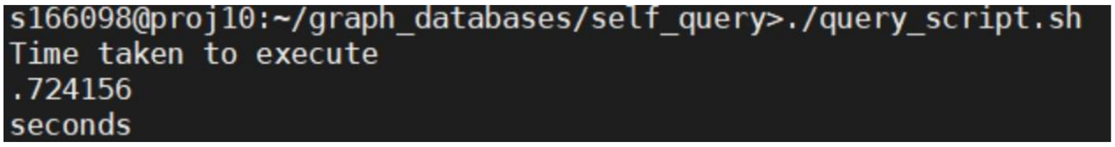

```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.608756
seconds
```

High degree

Cypher script


```
MATCH (p1:Person {id:2199023262543})-[:knows]->(p2:Person {gender:'male'})-[:hasInterest]->(t:Tag)
RETURN
t.id AS tagId,
t.name AS tagName
```

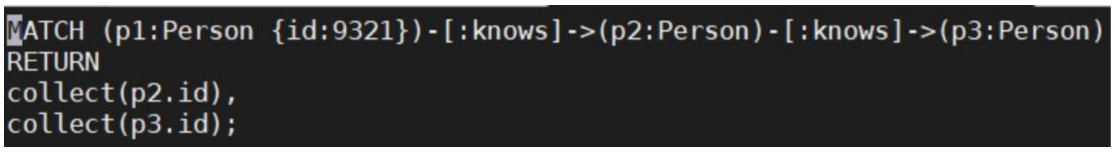
Result


```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.724156
seconds
```

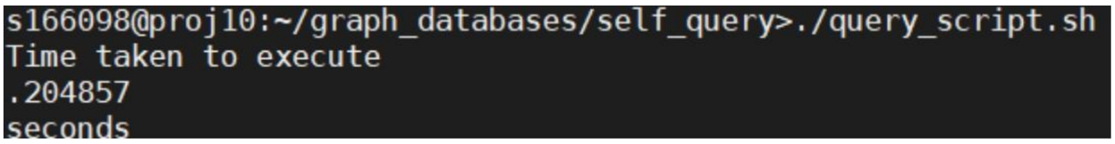
The third kind of query is to execute aggregation among two-hop neighbors from the beginning vertex without filter.

Low degree

Cypher script

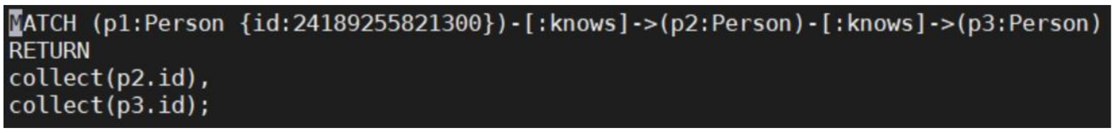

```
MATCH (p1:Person {id:9321})-[:knows]->(p2:Person)-[:knows]->(p3:Person)
RETURN
collect(p2.id),
collect(p3.id);
```

Result

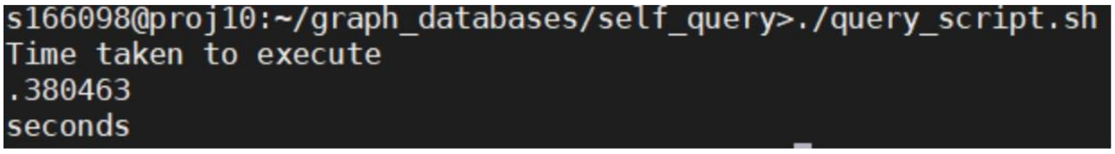

```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.204857
seconds
```

Medium degree

Cypher script

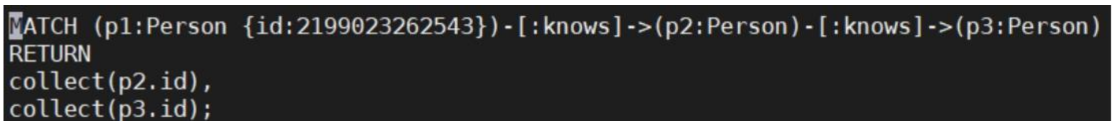

```
MATCH (p1:Person {id:24189255821300})-[:knows]->(p2:Person)-[:knows]->(p3:Person)
RETURN
collect(p2.id),
collect(p3.id);
```

Result

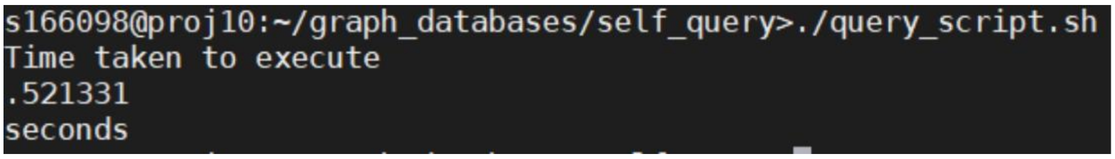

```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.380463
seconds
```

High degree

Cypher script


```
MATCH (p1:Person {id:2199023262543})-[:knows]->(p2:Person)-[:knows]->(p3:Person)
RETURN
collect(p2.id),
collect(p3.id);
```

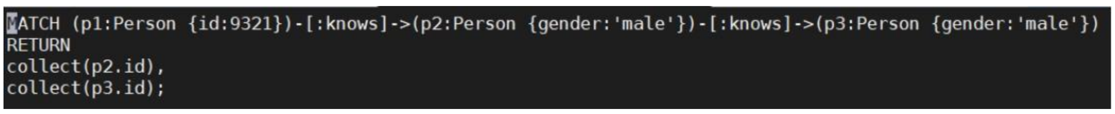
Result


```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.521331
seconds
```

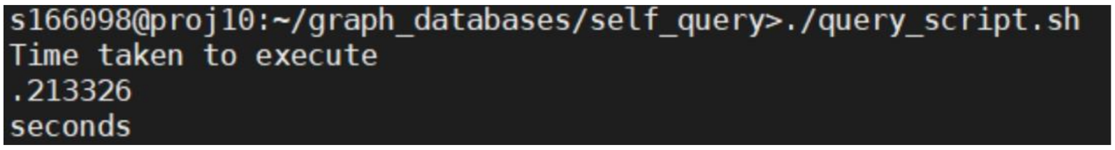
Forth kind of query is to execute aggregation among two-hop neighbors from the beginning vertex with filter.

Low degree

Cypher script


```
MATCH (p1:Person {id:9321})-[:knows]->(p2:Person {gender:'male'})-[:knows]->(p3:Person {gender:'male'})
RETURN
collect(p2.id),
collect(p3.id);
```

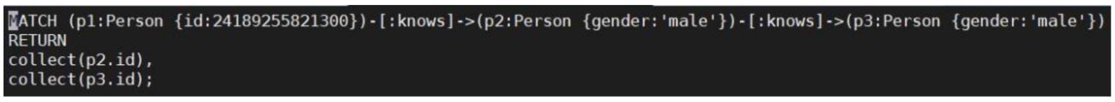
Result



```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.213326
seconds
```

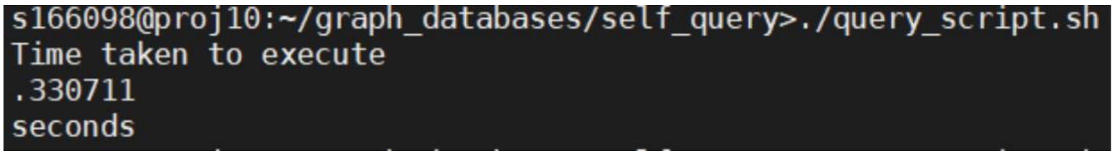
Medium degree

Cypher script



```
MATCH (p1:Person {id:24189255821300})-[:knows]->(p2:Person {gender:'male'})-[:knows]->(p3:Person {gender:'male'})
RETURN
collect(p2.id),
collect(p3.id);
```

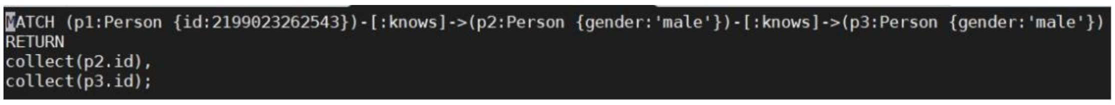
Result



```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.330711
seconds
```

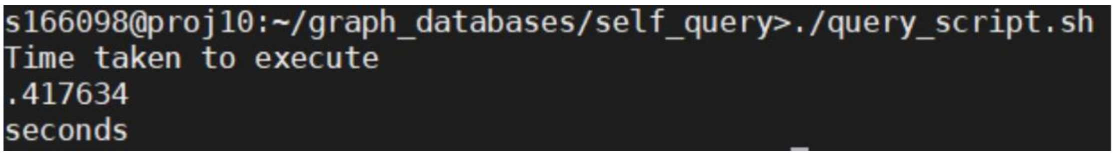
High degree

Cypher script



```
MATCH (p1:Person {id:2199023262543})-[:knows]->(p2:Person {gender:'male'})-[:knows]->(p3:Person {gender:'male'})
RETURN
collect(p2.id),
collect(p3.id);
```

Result

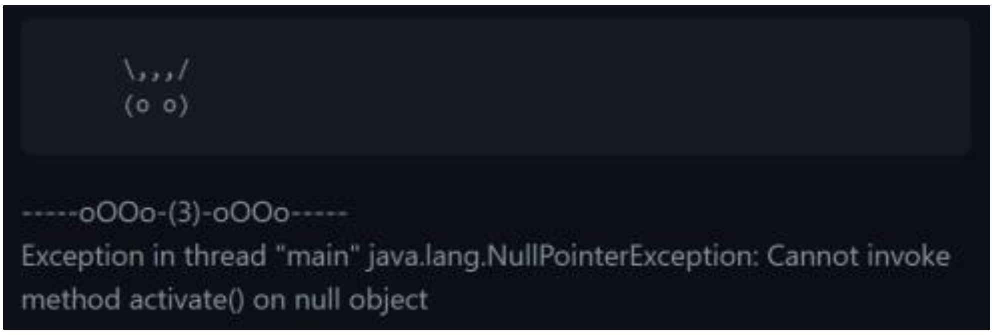


```
s166098@proj10:~/graph_databases/self_query>./query_script.sh
Time taken to execute
.417634
seconds
```

3.4 OrientDB

There is a graph computing framework called TinkerPop. When a data system is TinkerPop-enabled, its users are able to model their domain as a graph and analyze that graph using the Gremlin graph traversal language. And OrientDB adheres to the Apache TinkerPop standard and implements TinkerPop Stack interfaces.

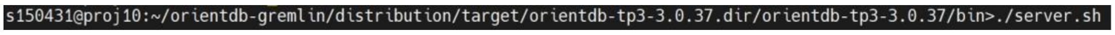
First, we need to install orientdb then we can build orientdb-gremlin, and the source code can be found in Github. The first version is 3.2.4. After installation, the system keeps showing the error below.



```
        \,,,/
        (o o)
-----oOOo-(3)-oOOo-----
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
method activate() on null object
```

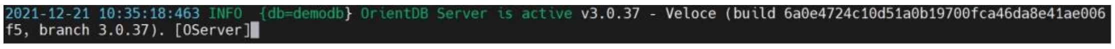
After searching, I find that there are some bugs in the latest version. Then I choose the version 3.0.37. And it works. And we can start the Gremlin console. The Gremlin Console is an interactive terminal that can be used to traverse graphs and interact with the data that they contain.

First thing first, we need to start the orientdb server by the command below.



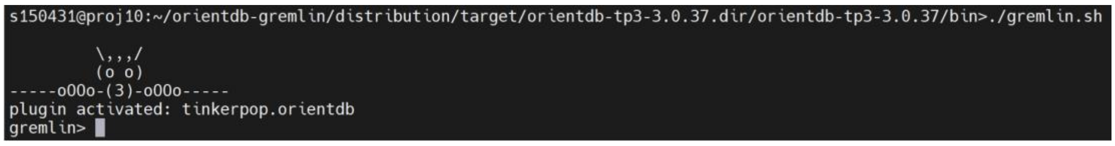
```
s150431@proj10:~/orientdb-gremlin/distribution/target/orientdb-tp3-3.0.37.dir/orientdb-tp3-3.0.37/bin>./server.sh
```

And it shows that it starts successfully below.



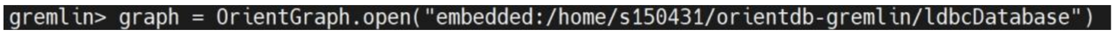
```
2021-12-21 10:35:18:463 INFO  {db=demodb} OrientDB Server is active v3.0.37 - Veloce (build 6a0e4724c10d51a0b19700fca46da8e41ae006
f5, branch 3.0.37). [OServer]
```

We can start the Gremlin console run the *gremlin.sh* script located in the bin directory of the OrientDB Gremlin Distribution below.



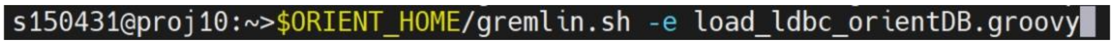
```
s150431@proj10:~/orientdb-gremlin/distribution/target/orientdb-tp3-3.0.37.dir/orientdb-tp3-3.0.37/bin>./gremlin.sh

         \,,,/
         (o o)
-----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.orientdb
gremlin>
```

Then we can connect to a database ldbcDatabase using the command below.



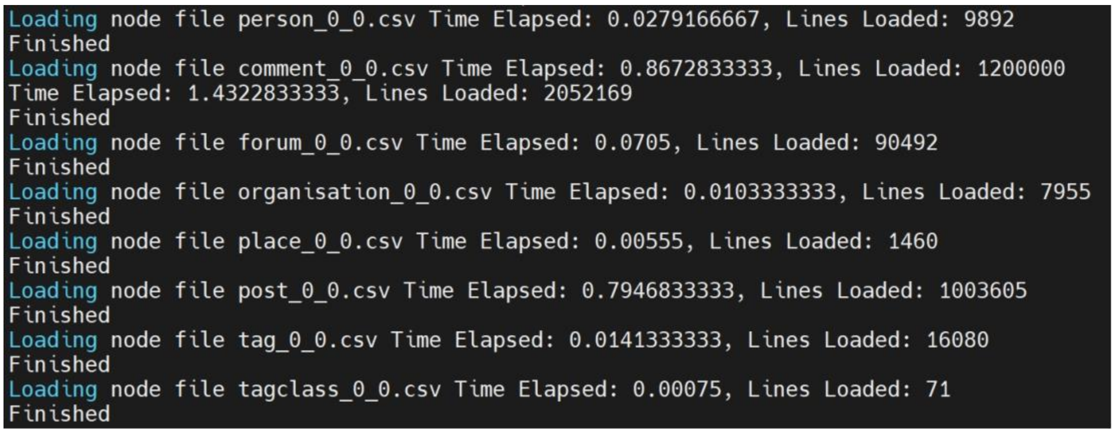
```
gremlin> graph = OrientGraph.open("embedded:/home/s150431/orientdb-gremlin/ldbcDatabase")
```

Now the database is created, we need to import the test data into it. Here we run a groovy file below which can be used as both a programming language and a scripting language for the Java platform.



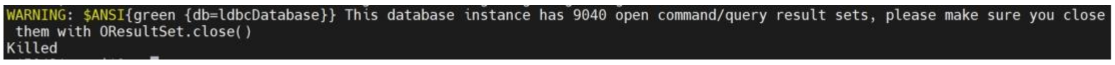
```
s150431@proj10:~>$ORIENT_HOME/gremlin.sh -e load_ldbc_orientDB.groovy
```

The output shows that all nodes have been imported.



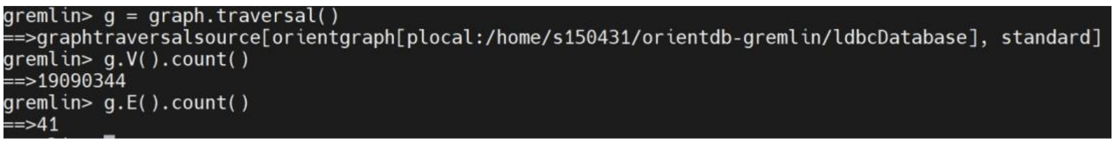
```
Loading node file person_0_0.csv Time Elapsed: 0.0279166667, Lines Loaded: 9892
Finished
Loading node file comment_0_0.csv Time Elapsed: 0.8672833333, Lines Loaded: 1200000
Time Elapsed: 1.4322833333, Lines Loaded: 2052169
Finished
Loading node file forum_0_0.csv Time Elapsed: 0.0705, Lines Loaded: 90492
Finished
Loading node file organisation_0_0.csv Time Elapsed: 0.0103333333, Lines Loaded: 7955
Finished
Loading node file place_0_0.csv Time Elapsed: 0.00555, Lines Loaded: 1460
Finished
Loading node file post_0_0.csv Time Elapsed: 0.7946833333, Lines Loaded: 1003605
Finished
Loading node file tag_0_0.csv Time Elapsed: 0.0141333333, Lines Loaded: 16080
Finished
Loading node file tagclass_0_0.csv Time Elapsed: 0.00075, Lines Loaded: 71
Finished
```

However, when it imports properties and edges, it shows a warning and finally be killed due to memory leak.



```
WARNING: $ANSI{green {db=ldbcDatabase}} This database instance has 9040 open command/query result sets, please make sure you close
 them with OResultSet.close()
Killed
```

Once I checked the number of vertexes and edges, the edges haven’t been imported fully.

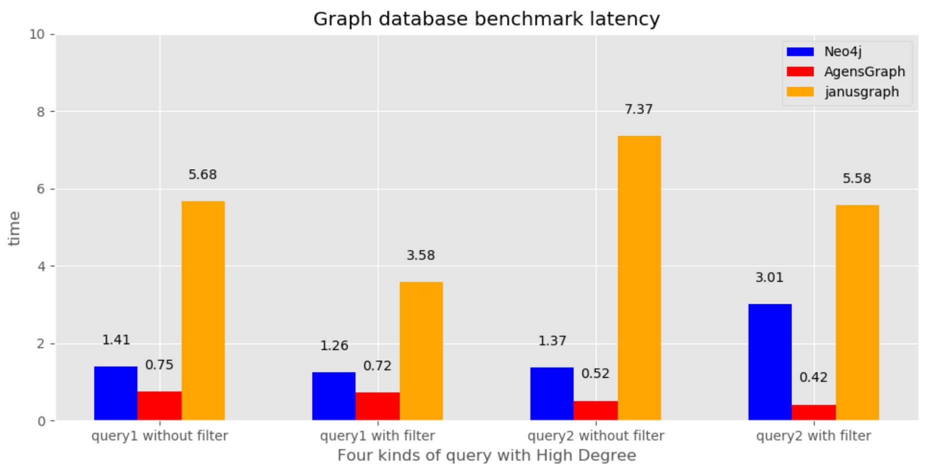


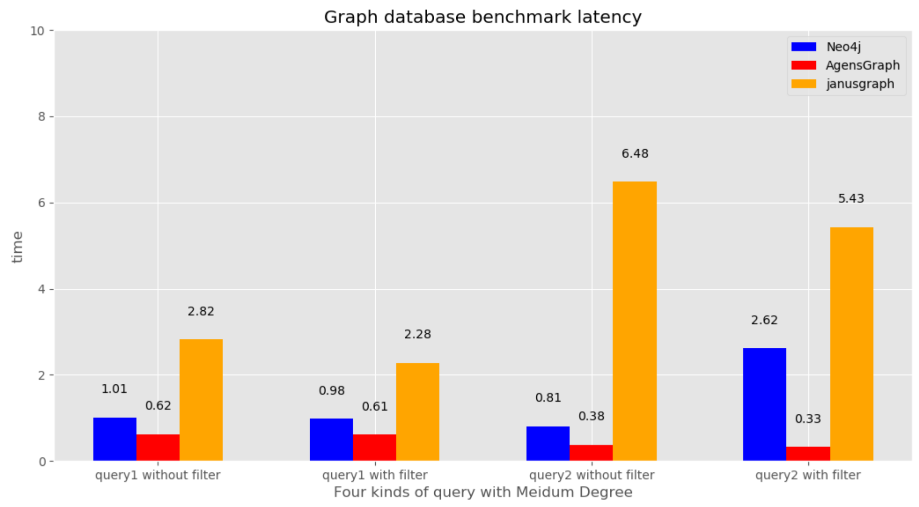
```
gremlin> g = graph.traversal()
==>graphtraversalsource[orientgraph[plocal:/home/s150431/orientdb-gremlin/ldbcDatabase], standard]
gremlin> g.V().count()
==>19090344
gremlin> g.E().count()
==>41
```

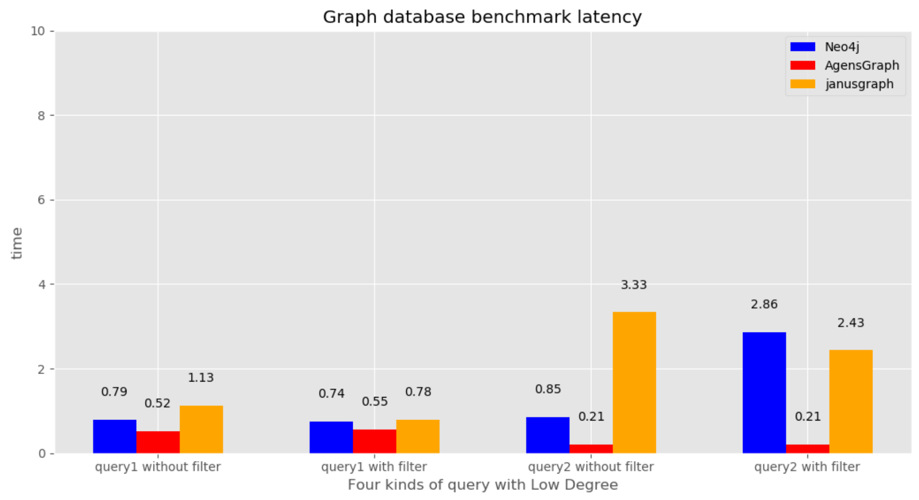
I also search for the solution, it says it’s a kind of bug, but there is no time for me to download the right version. Because the way we import these data to OrientDB is very similar to JanusGraph, we sepculate that the performance of OrientDB is close to JanusGraph. However, OrientDB does not mention Index, So the query may be slower than JanusGraph. In the following days, I will keep working on relative areas. And OrientDB will not be discussed in the next contents.

3.5 Results of RecGDBBench

The first type of query is the interests discovering, and the second is the aggregation. We test two types of queries on three databases and three types of vertexes respectively. AgenseGraph performs the best on all queries. On the first type of query which is essentially one-hop traversal, Neo4j shows a similar performance to AgenseGraph, however, the aggregation performance is relatively subpar.







The Neo4j has the better performance in terms of the high degree vertex, but it performs nearly the same as the JanusGraph on the low degree vertex.

Conclusion

Our experiments examine the differences and advantages of these databases. AgenseGraph is the optimal option to conduct recommendations among the databases we tested. Neo4j also has merit that other databases do not have, which is the importing speed. Although the importing procedure on AgenseGraph is not too long, it still takes more than ten minutes to load the data. However, the Neo4j takes less than one minute to finish the same loading task. JanusGraph is not recommended for this kind of task in either importing or the recommendation tasks we designed.

Reference

1. Angles, R., et al., *The linked data benchmark council: a graph and RDF industry benchmarking effort.* ACM SIGMOD Record, 2014. **43**(1): p. 27-31.
2. Webber, J. *A programmatic introduction to neo4j*. in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. 2012.
3. Mattias Finné, A.N., etc., *Neo4j.* 2019.
4. Authors, J., *JanusGraph: distributed graph database; 2017.*
5. TinkerPop, A., *"Tinkerpop3*. 2016.
6. Cassandra, A., *Apache cassandra.* Website. Available online at

http://planetcassandra. org/what-is-apache-cassandra, 2014. **13**.

7. George, L., *HBase: the definitive guide: random access to your planet-size data*. 2011: " O'Reilly Media, Inc.".

8. Olson, M.A., K. Bostic, and M.I. Seltzer. *Berkeley DB*. in *USENIX Annual Technical Conference, FREENIX Track*. 1999.

9. Tesoriero, C., *Getting started with OrientDB*. 2013: Packt Publishing Ltd.

10. Angles, R., et al. *Benchmarking database systems for social network applications*. in *First International Workshop on Graph Data Management Experiences and Systems*. 2013.

11. Matteo Lissandrini, M.B., *Graph Databases Benchmarking Suite.* 2019.

Contribution

FU Junchen

1. Coordinate and assign the tasks among the group members

2. Generate the social network datasets

3. Deploy the Neo4j, JanusGraph two databases, loading the generated dataset into the systems

4. Design the RecGDBBench, and implement it in Gremlin.

5. Running the experiments on Neo4j and JanusGraph

6. In charge of the report structure design, abstract, JanusGraph

introduction, Neo4j and JanusGraph’s experiment, Results of RecGDBBench, Conclusion.

Zhang Weidong

1. Deploy the AgensGraph, loading the generated dataset into database.
2. Write and implement bash script to exectute Cypher queries in AgensGraph.
3. Plot the latency of queries executed in three different graph databases.
4. In charge of agensGraph introduction and experiment, neo4j introduction.

LIU Xingchen

1. In charge of the introduction of Neo4j and LDBC benchmark in phase 1

2. Install graph database OrientDB and do some experiments in the framework of gremlin in phase 2.

3. Coordinate with groupmates to accomplish the tasks of my own part

Appendix

RecGDBBench Cypher version in this experiment,

Type one without filter,

```
MATCH (p1:Person {id:$personId})-[:KNOWS]->(p2:Person)-
[:HAS_INTEREST]->(t:Tag)
RETURN
t.id AS tagId,
t.name AS tagName;
```

Type one with filter,

```
MATCH (p1:Person {id:$personId})-[:KNOWS]->(p2:Person
{gender:'male'})-[:HAS_INTEREST]->(t:Tag)
RETURN
t.id AS tagId,
t.name AS tagName;
```

Type two without filter,

MATCH (p1:Person {id:$personId})-[:KNOWS]->(p2:Person)-[:KNOWS]->(p3:Person)

RETURN

collect(p2.id),

collect(p3.id);

Type two with filter,

MATCH (p1:Person {id:$personId})-[:KNOWS]->(p2:Person {gender:'male'})-[:KNOWS]->(p3:Person {gender:'male'})

RETURN

collect(p2.id),

collect(p3.id);

RecGDBBench Gremlin version in this experiment,

Type one without filter,

g.V().has('iid',$personId).out("knows").out("hasInterest").properties().profile()

Type one with filter,

g.V().has('iid',$personId).has("gender","male").out("knows").out("hasInterest").properties().profile()

Type two without filter,

g.V().has('iid',$personId).union(out("knows").properties('iid'),out("knows").out("knows").properties('iid')).profile()

Type two with filter,

g.V().has('iid',$personId).union(out("knows").has("gender","male").properties('iid'),out("knows").has("gender","male").out("knows").has("gender","male").properties('iid'))