A Project Report

on

InfiniteGraph

Course: CSC8711

Prof: Dr. Raj Sunderraman

By:

Sadhna Kumari

Sara Karamati

TABLE of CONTENTS

LIST OF FIGURES

Part 1

INTRODUCTION

Our digital ecosystem is expanding due to the data gathered by web traffic, social media, financial transactions, email, phone calls, IT logs and more. This data is huge in gigabytes to terabytes and even petabytes, complex and interconnected. In fact, IBM estimates that 90% of the data in the world today has been created in the last two years alone. Buried in this mountain of data is intelligence that can be used to shape strategy, improve business processes and increase profits. Thus, it is important to understand and quickly act on extremely large data sets.

Leading analyst firm, Gartner, reports global enterprise data assets to grow by an additional 650 percent by the end of 2014.

1.1 NOSQL Databases

NOSQL databases are developed to deal with large scale data needs and the storage capacity limitations of traditional relational databases. NOSQL approaches helps in unlocking valuable information by organizing large interconnected data. These solutions can be divided into four different technology categories:

1.1.1 Key- value Databases

A Key-value database is similar to a relational database with rows, but only two columns. The indexing system uses a single string (key) to retrieve the data (value) .

- Very fast for direct look –ups
- Schema - less, meaning the value could be anything, such as an object or a pointer to data in another data store

1.1.2 Column Family Databases

Column family databases also have rows and columns like a relational database, but storage on disk is organized so that columns of related data are grouped together in the same file. As a

result, attributes (columns) can be accessed without having to access all of the other columns in the row.

- Results in very fast actions related to attributes, such as calculating average age
- Performs poorly in regular OLTP applications where the entire row is required

1.1.3 Document Databases

Document databases are similar to object databases, but without the need to predefine an object's attributes (i.e., no schema required).

- Provides flexibility to store new types or unanticipated sizes of data/objects during operation

1.1.4 Graph Databases

Graph databases are also similar to object databases, but the objects and relationships between them are all represented as objects with their own respective sets of attributes.

- Enables very fast queries when the value of the data is the relationships between people or items
- Use Graph Databases to identify a relationship between people/items, even when there are many degrees of separation
- Where the relationships represent costs, identify the optimal combination of groups of people/items

1.2 Distributed Graph Databases

Typical Use Cases for distributed graph databases:

- Social Graph Analysis
- Catching Bad Guys
- Fraud / Financial (more bad guys)
- Data Intensive Science
- Web / Advertising Analytics

These types of application tend to grow quickly. Some analytics require navigation of large sections of the graph.

Graph database should be optimized around data relationships unlike SQL. In SQL, relationships are not treated as first class citizens. Small focused API (typically not SQL) comes handy in graph databases. The applications require navigating through the graph. Since the data size is huge, a distributed graph must distribute data and go parallel.

1.3 Distributed Graph Partitioning

Graph partitioning is ugly in distributed graph as shown in the Fig.1.

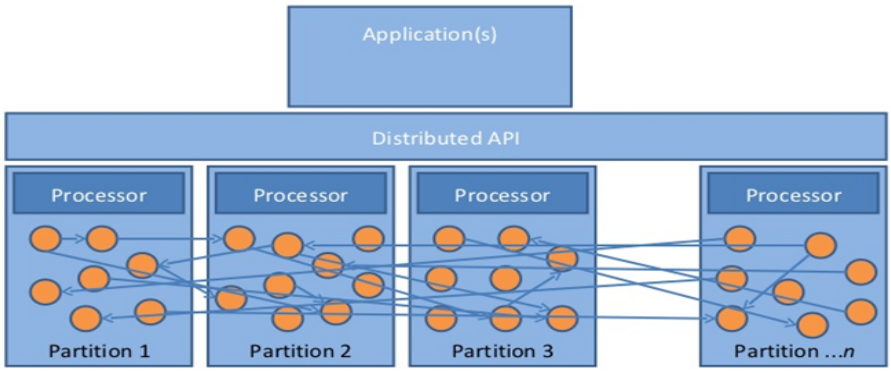


Figure 1 Graph Partitioning

There are some challenges like Graph operations are rarely partition bound, Repartitioning is expensive and Partitions must co-operate. The other aspect of distributed graph is that graph algorithms naturally branch. For example, if the start node for querying is "Alice" as shown in the diagram, and there are two edges, then we can have two processors (threads) working on the same query. Breaking up the process in these types of algorithm is relatively simple but orchestrating is more challenging.

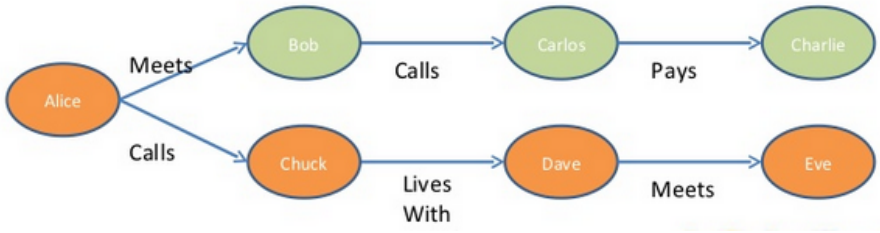


Figure 2 Distributed Navigation

Part 2

INFINITEGRAPH: OVERVIEW

InfiniteGraph enables organizations to achieve greater return on their data related investment by helping them “connect the dots” on a global scale, ask deeper and more complex questions, across new or existing data stores.

There is no other graph technology available today, offered by any other commercial vendor or open source project, that can match InfiniteGraph’s combined strengths of persisting and traversing complex relationships requiring multiple hops, across vast and distributed data stores.

2.1 Releases

InfiniteGraph started as an internal project in Objectivity whose focus was on management and analysis of graph data. It took the high performance distributed data engine from Objectivity/DB and married it to a graph management and analysis platform which makes development of complex graph analytic applications significantly easier.

1. Release 1.0: first iteration and was offered as a public beta.

2. Release 2.1: first commercial release

3. Release 3.0: more features were added focused around scaling the graph in a distributed environment.

4. Release 3.1: Offers Improved Data Ingestion, Faster Search Results, and Open Use Data Connectors

2.2 Features

- Derived from Objectivity/DB core
- Distributed Object Database Native core (broad platform support) with C++, Java, C# and Python Bindings

- Simple Graph focused API
- Automated distribution and deployment
- A distributed data tier supports parallel IO
- Ability to deal with remote data reads (fast)
- High performance distributed persistence (Java Class based)
- Distributed navigation processing: Asynchronous navigation
- Distributed, multi-source concurrent ingest
- Indexing framework
- Write modes supporting both strict and eventual consistency

2.3 Applications

Thousands of deployments, many are 24x7x365 Markets : VLDB, Data Fusion / Metadata, Complex Object Models, Relationship Analytics

Figure 3 Customers and Partners

2.4 Architecture

Infinite graph provides many user friendly plugins and small API for developers to build applications. API interacts with underlying distributed data using locks.

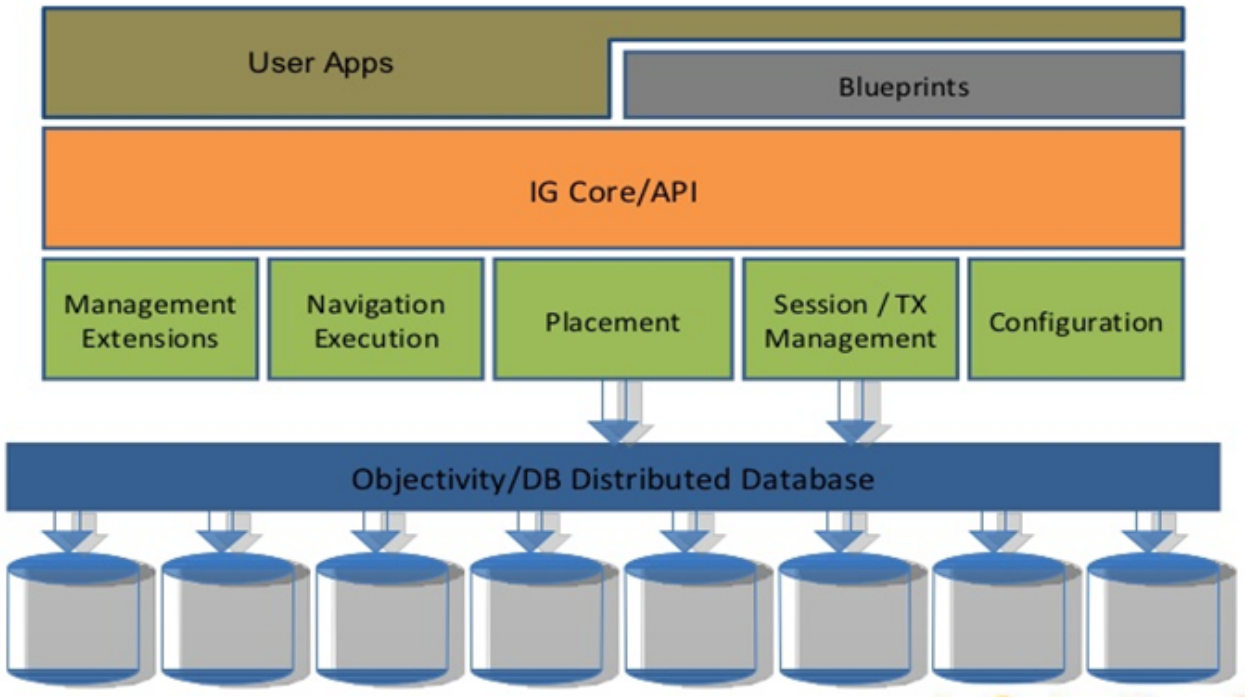


Figure 4 Architecture

Infinite graph provides two types of concurrency modes. An application can choose between modes: Full consistency and relaxed consistency. It trades off full consistency for performance. Locking is relaxed, thus allowing stale reads. Applications like social graphs tend to have relaxed consistency requirements and enjoys relaxed feature by having higher performance.

Part 3

TECHNICAL SPECIFICATIONS

3.1 Schema Model

InfiniteGraph saves edges and vertices as persistent data. The library consists of two classes namely BaseVertex and BaseEdge which are defined as persistent. All the instances of vertices should inherit from BaseVertex or subclass of BaseVertex. Similarly, edge instances should inherit from BaseEdge. The instances can be saved in an InfiniteGraph graph database. Instances of a persistent class can act both as standard Java runtime objects and as persistent elements stored in an InfiniteGraph graph database.

If more than one application needs to use persistent elements of a given class, each application must have a class definition that includes both declarations for fields and implementations for application-defined methods. At the time of database write, the values of fields are also persistently stored. Persistent fields must be one of the following data types.

Category	Types
Numeric	char byte short int long float double boolean
String	java.lang.String java.lang.StringBuffer
Date or time	java.util.Date java.sql.Date java.sql.Time

	java.sql.Timestamp

3.2 System Overview

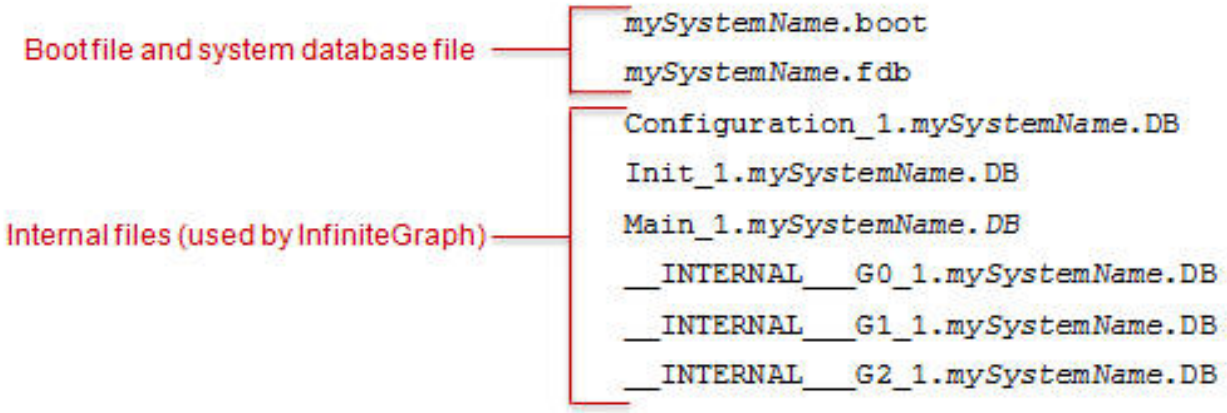
Graph database is created by providing a system name, which is a logical name for the graph. Following code creates the graph in default working directory.

GraphFactory.create("mySystemName");

To create the graph in user specified location, flollowing code is used which creates in path specified in ".properties" file.

GraphFactory.create("mySystemName", "myPropertyFilePathName") ;

Upon creation of graph database, Infinite Graph creates following files.



To connect to Graph, logical name is provided as follows:

GraphFactory.open("mySystemName");

or

GraphFactory.open("mySystemName", "myPropertyFilePathName");

A graph can be created with one property file, and opened with a different one. Once connected to a graph, an application can access, update, or instantiate persistent elements inside a read or read/write transaction.

The first time persistent elements are added to the graph, database files are created to store those elements

- Vertex instances are placed in VertexGroup_*n.systemName*.DB
- Edge instnaces are placed in EdgeGroup_*n.systemName*.DB
- Internal information related to edges is stored in ConnectorGroup_*n.systemName*.DB.

Locations of those database files are added to the system database file. The schema definitions for the elements are also added to the system database file. Moving forward, additional instances of that type have access to the schema. Each new persistent element that is added to the graph is given a unique identifier and stored in the appropriate database file. As applications make updates to the graph, journal files are created. These files are used to return the graph to its previously committed state if a transaction is aborted or terminated abnormally.

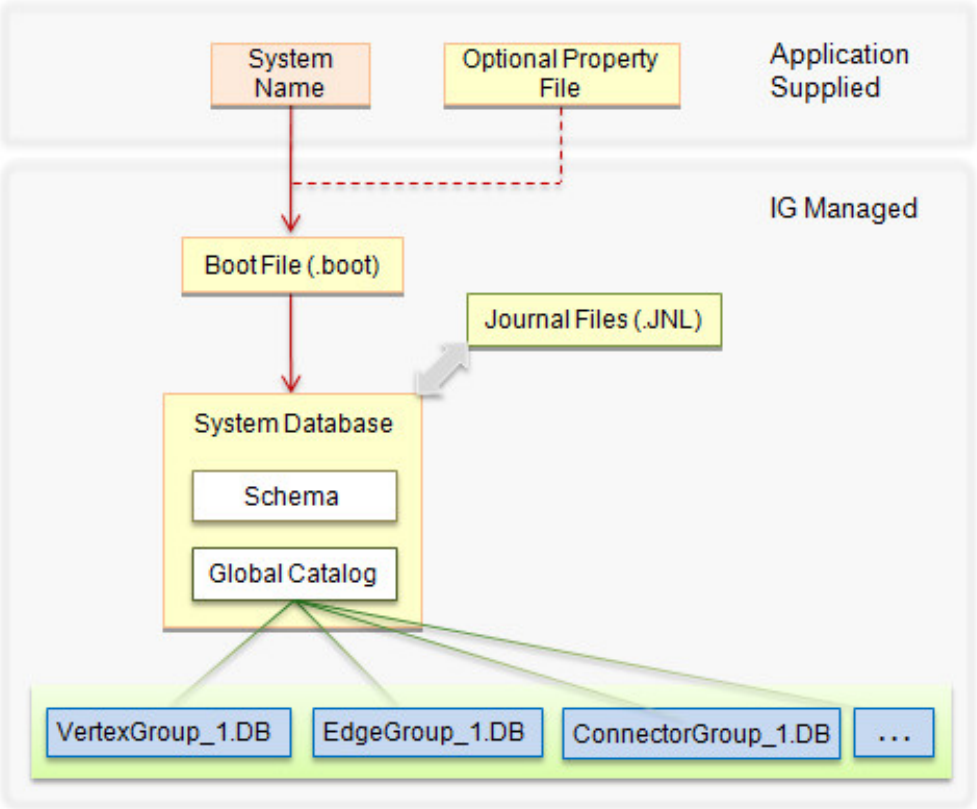


Figure 5 Graph with Persistent Elements

The initial placement model has rules that specify that Vertex and Edge instances are placed in individual database files named VertexGroup_*n.systemName*.DB and EdgeGroup_*n.systemName*.DB, respectively. Internal information related to edges is stored in ConnectorGroup_*n.systemName*.DB file. The database files themselves are stored either in the default storage location or in a location in the MSG (if storage locations were registered for the graph).

Each database file starts with a single initial container that can grow to a size of 3200 kilobytes. New containers are added as needed. By default, each container starts with 100 storage pages, which are 16 kilobytes in size. New storage pages are added as needed. In distributed environment, Advanced Multithreaded Server serves data. Each application has an XML rank file that designates its preferred storage locations.

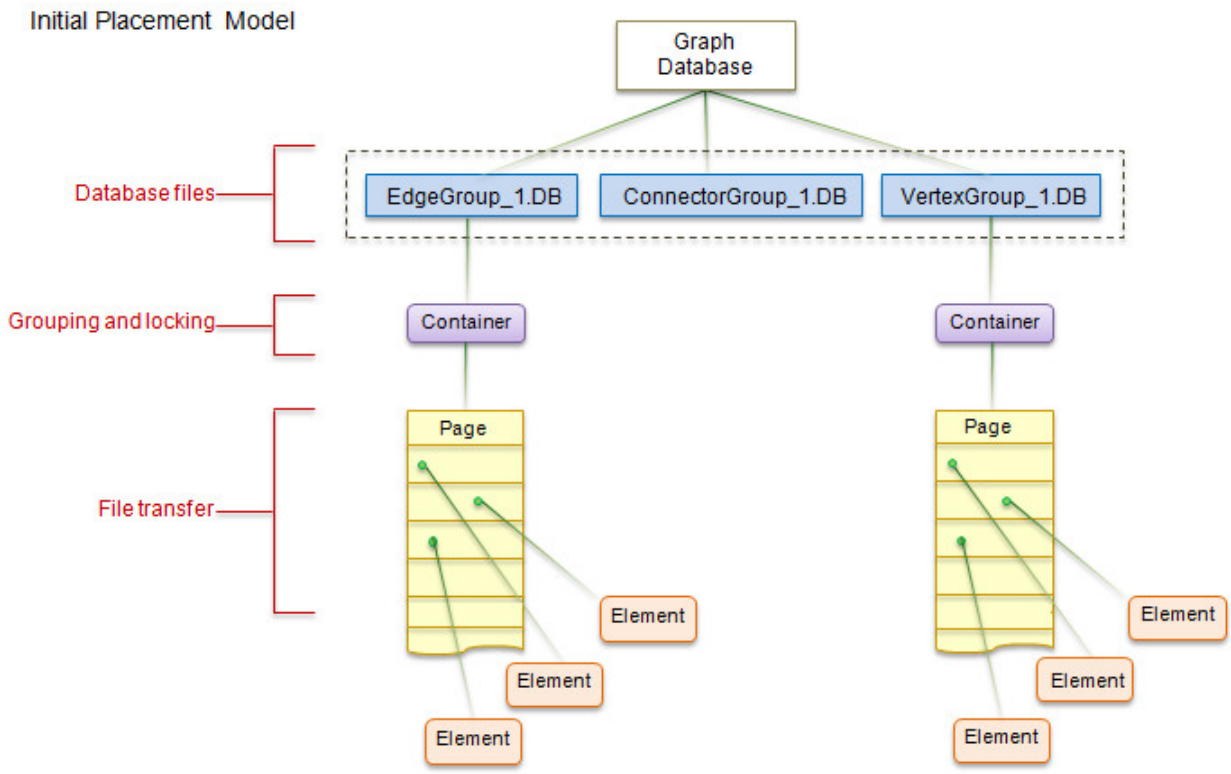


Figure 6 Placement of Persistent Elements

3.3 Ingesting Data

Following code creates an Employee vertex that can be used as a Java runtime.

```
public class Employee extends BaseVertex {
```

```
        // Fields
        private String name;
        private String department;
        private int id;
        private boolean permanent;
        ...
    }
    Employee emp1 = new Employee("John");
myGraph.addVertex(emp1);
```

The vertex becomes persistent when it is explicitly added to the graph database. An edge becomes persistent after it is passed to an addEdge method.

Data ingestion flow:

Start a new transaction.

For each from vertex,

check whether or not it already exists in the database.

If no, create the vertex and return a reference to it.

If yes, return a reference to it.

For each to vertex, repeat the above process.

Create the edge, passing in the from and to vertices.

Increment counters, commit the transaction, and repeat the cycle.

Infinite graph supports two types of ingest:

1. Standard ingest

It is easy to set up and use appropriate when ingesting data in a single thread or process. It uses InfiniteGraph APIs such as addVertex and addEdge to ingest data inside a read/write transaction. The ingested data is immediately consistent and available upon commit of a transaction.

2. Accelerated ingest

It is particularly effective when ingesting data with large numbers of edges. It can provide optimal performance when ingesting large amounts of data using multiple threads or multiple processes. The ingested data has eventual consistency because not all edges are immediately available after a transaction is committed.

3.3 Graph Navigation

Given a source vertex, a navigation query is performed to traverse the graph searching for target elements that meet certain criteria. First step is to construct a navigator object that encapsulates information needed for the query. A navigator identifies the source object, specifies the criteria for qualifying target elements, and defines the mechanism for receiving and outputting found results. A navigator also includes filtering capabilities and traversal algorithms. Navigation policies can be used to affect the behavior of the navigator as a whole, such as by setting the maximum traversal depth (degrees of separation) or the maximum number of returned results.

For example, Fig 7 shows a graph database with start node Lisa. Get the Member vertex named "Lisa“ as starting point.

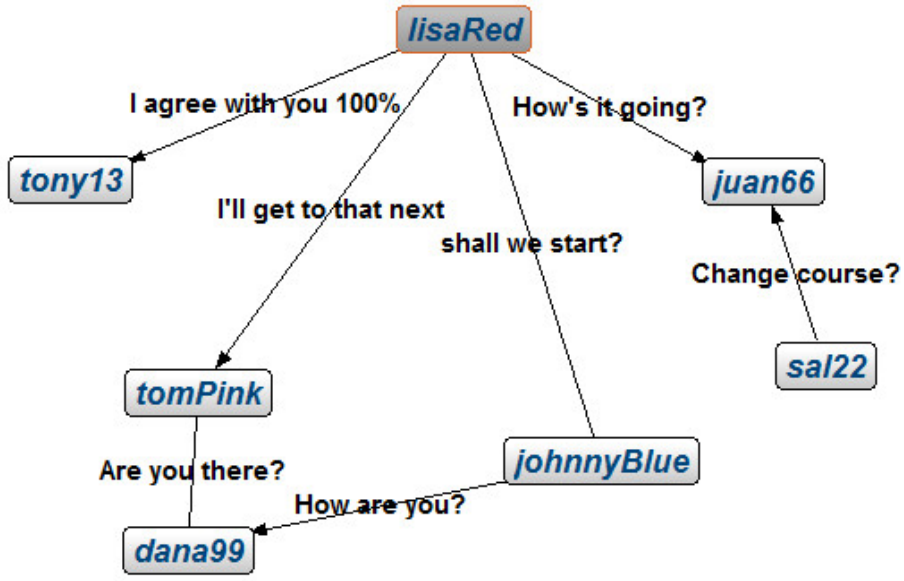


Figure 7 Infinite Graph Navigation Example

Member lisa = (Member)WebGroupSampleDB.getNamedVertex("Lisa");

Create instance of result handler

PrintPathResultsHandler resultPrinter = new PrintPathResultsHandler();

Navigator myNavigator = lisa.navigate(null, Guide.SIMPLE_DEPTH_FIRST, Qualifier.FOREVER, Qualifier.ANY, null, resultPrinter)

myNavigator.start();

The result is as follows.

... - > Starting an update transaction ...

... - FOUND MATCHING PATH:

... - lisaRed < Sat May 21 08:45:22 PDT 2011 > tomPink

... - FOUND MATCHING PATH:

... - lisaRed < Sat May 21 08:45:22 PDT 2011 > tomPink < Sun May 22 03:45:22 PDT 2011 > dana99

... - FOUND MATCHING PATH:

... - lisaRed < Sun May 22 07:45:22 PDT 2011 > juan66

... - FOUND MATCHING PATH:

... - lisaRed < Sun May 22 03:45:22 PDT 2011 > johnnyBlue

... - FOUND MATCHING PATH:

... - lisaRed < Sun May 22 03:45:22 PDT 2011 > johnnyBlue < Sat May 21 10:45:22 PDT 2011 > dana99

... - FOUND MATCHING PATH:

... - lisaRed < Sun May 22 03:45:22 PDT 2011 > johnnyBlue < Sat May 21 10:45:22 PDT 2011 > dana99 < Sun May 22 03:45:22 PDT 2011 > tomPink

... - FOUND MATCHING PATH:

... - lisaRed < Sat May 21 09:45:22 PDT 2011 > tony13

... - > Program completed ...

3.4 Indexing

InfiniteGraph provides several indexing options. The most powerful and easy to use is the graph index, which automatically index data according to the class name and field swpecified.

IndexManager.addGraphIndex("personGraphIndex", Person.class.getName(), new String[] {"name"}, false);

Every Person vertex added to the graph database is automatically included in the index. You can also create a graph index with multiple key fields. The first key you provide is used as the primary sort key.

IndexManager.addGraphIndex("personGraphIndex", Person.class.getName(), new String[] {"name"}, false);

3.5 Query

You can execute a high performance database-wide query with the help of the placement manager and any graph indexes that are available. Following code creates a query object that identifies Person vertices whose name field value is John.

Query<Person> personQuery = myGraph.createQuery(Person.class.getName(), "name== 'John'");

Assuming there is a graph index on the name field of the Person class, this query will have optimal performance when executed. Query object can be created if graph database does not have graph indexes, but it doesn’t provide performance gains seen when a corresponding graph index exists. To use the query object, execute it to create an iterator that lets you cycle through any matching elements:

Iterator personItr = (Iterator) personQuery.execute();

while (personItr.hasNext()) {

    Person myPerson = (Person) personItr.next();

    System.out.println("Found person named " + myPerson.getName());

}

The following code works even though the age field of the Person class is not one of the indexed fields:

Query<Person> personQuery = myGraph.createQuery(Person.class.getName(), "name== 'John' && age < 100");

The performance of the above query is improved when both name and age are indexed.

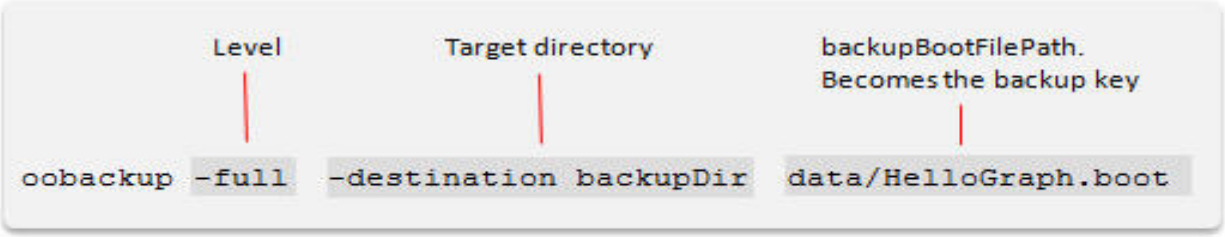
3.5 Lock Server

The lock server manages concurrent access to persistent elements by granting or refusing locks to requesting transactions. A transaction requests data from a graph database. InfiniteGraph locates the lock server for that graph and requests a lock on the container holding the data. If an application attempts to write data to a container that is already locked, the second lock is granted only if it is compatible with the existing lock. Two read/write locks cannot be granted on the same container at the same time.

When such a conflict occurs, InfiniteGraph reacts according to the application's configured LockWaitTime property. By default, InfiniteGraph fails immediately on such a conflict, issuing an exception. You can change the default behavior to wait for a specified number of seconds or to wait indefinitely. InfiniteGraph does allow multiple read operations to occur concurrently with a single read/write operation (MROW). You can change the UseMrowTransactions configuration property to false to disable MROW.

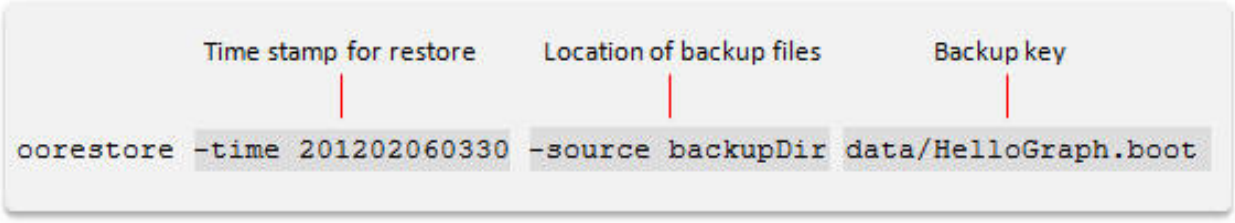
3.6 Backup and Restore

A backup is a snapshot of a graph database at a particular point in time.The first time you perform a full backup, you start what is known as a backup set. You can add to the backup set with periodic updates. InfiniteGraph provides a basic backup capability: Automatically generates the names of all backup files and implicitly manages the backup set.



Alternatively, you can perform a custom backup in which you name the backup set and choose from 10 backup levels for the backup events.

Whether using basic or custom backups, each backup event on a given backup set represents a potential point of restore. When you perform a restore operation, InfiniteGraph always restores the entire graph database to ensure its integrity. To restore from a basic backup, a timestamp is specified as a point of restore. If no backup corresponds exactly to the specified time, it selects the latest backup that was started prior to the specified time.



InfiniteGraph allows full read and write access to the graph database during the backup. However, during a restore, the graph database is locked until the entire restore is completed.

Part 4

APPLICATION

4.1 Introduction & Overview

In this project, we implemented a Friends Network website. Since a social network is heavily dependent on connections between database entities in order to manage user profile, user news feed, and their friends' network, this project is an ideal application to represent some of the capabilities that a graph database can offer.

In this project, infiniteGraph NoSQL database is used as the backend database system. Java Servlet, HTML, and CSS are used to implement data processing and the presentation of the project. In the next sections, we'll go over major parts of the project and discuss their functionality and role in the application.

4.2 Database structure

The graph data of the Friends Network application is shown in Fig. 8:

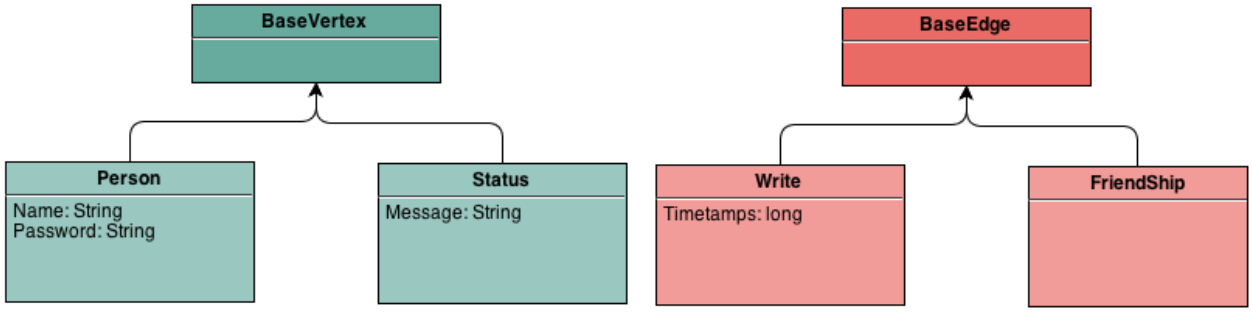


Figure 8 Graph Data

The Friends Network application consists of two vertex types:

1) Person vertex: This vertex saves the client's user name and password. Each account should have a unique user name.
2) Status vertex: This vertex is used to save user status.

The following edge types are also defined to connect different vertices:

1) Friendship edge: Each person is connected with his/her friends with a friendship edge.
2) Write edge: This edge stores the information about status posting time. Each status is associated with the user who posted the status by a write edge.

Figure 9 shows a snapshot of the infiniteGraph data model. The infiniteGraph visiulizer software is used to illustrate these data.

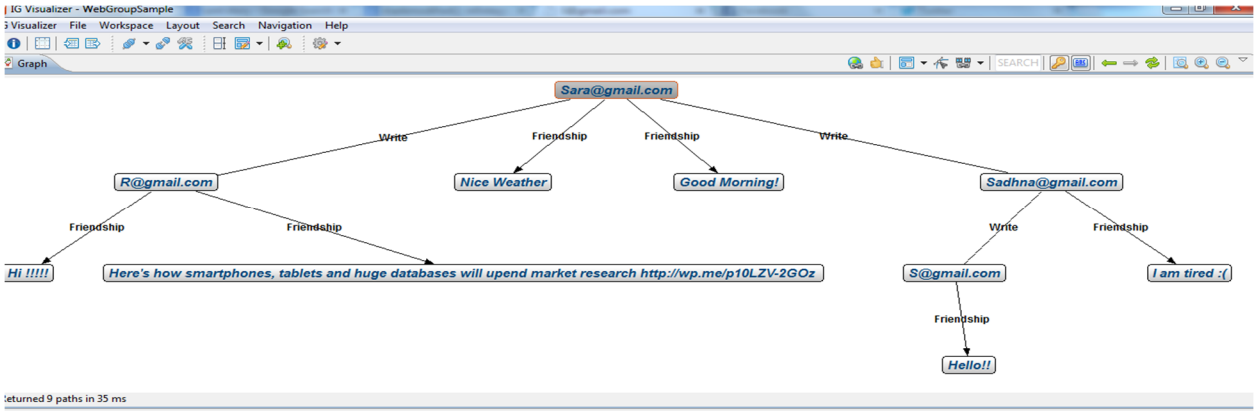


Figure 9 infiniteGraph visualizer result for vertex named "Sara@gmail.com"

To represent vertices and edges in infiniteGraph database four Java classes are defined. The class definitions for status vertex and write edge are shown in listing 1:

```
public class Status extends BaseVertex
{
     private String message;
     public Status(String message)
     {
          setMessage(message);
     }
     public void setMessage(String message)
     {
          markModified();
          this.message = message;
      }
      public String getMessage()
      {
            fetch();
            return message;
      }
      @Override
       public String toString()
        {
             fetch();
             return this.message;
```

```
Public class Write extends BaseEdge
{
   private long timestamp;
   public Write(Calendar date)
   {
      setTimestamp(date.getTimeInMillis());
   }
   public Calendar getTimeStamp()
   {
      fetch();
      Calendar myCal = Calendar.getInstance();
      myCal.setTimeInMillis(timestamp);
      return myCal;
   }
   protected void setTimestamp(long timestamp)
   {
      markModified();
      this.timestamp = timestamp;
   }
   @Override
   public String toString()
   {
```

```
        }
}
```

```
        fetch();
        Calendar myCal = Calendar.getInstance();
        myCal.setTimeInMillis(timestamp);
        return myCal.getTime().toString();
    }
}
```

Listing 1 Status vertex and Write edge class difinition

4.3 Front End

The front end is responsible for generating appropriate presentation to interact with the user. This part of the application collects data from the user and presents their friends and their news feed. The news feed is a list of most recent statuses posted by each user and his/her friends. No user can see statuses made by other users unless they are connected with a friendship edge.

After a successful registration, the user can search for a person in the database, add a person to his/her friendship network, and post a status. Figure 10 shows the actions that a user can do by the user interface.

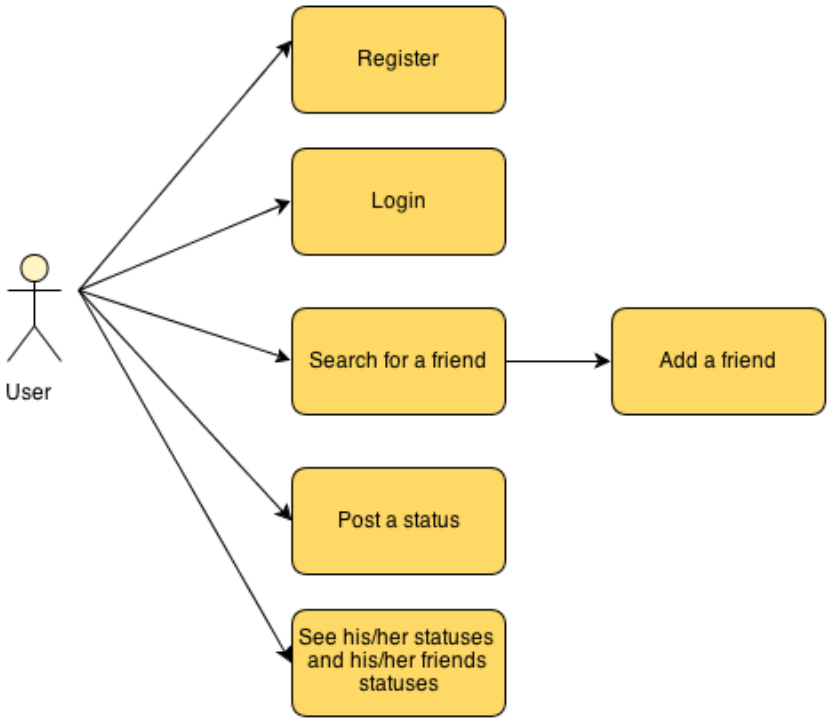


Figure 10 Friends network user story

Figure 11 shows the registration page of the user interface. This page collects the new user's information and adds them as a new person vertex to the infiniteGraph database. Each person should have a unique user name, otherwise the registration will not complete.

Figure 11 Registration page user interface

Listing 2 shows Java code for adding a person vertex to the graph during registration.

```
if(user==null){
      find=false;
      Person newPerson = new Person(MemberName, password);
      WebGroupSampleDB.addVertex(newPerson);
      WebGroupSampleDB.nameVertex(MemberName, newPerson);
}
```

Listing 2 Registration Java code

After registration, users will be directed to their homepage where they can search for a person in the database and add the people from the output of the search query to their friends' network. After adding a friend, these to persons are connected by a friendship edge as shown in listing 3.

```
Friendship newFriend = new Friendship();
Person user = (Person)WebGroupSampleDB.getNamedVertex(userName);
Person friend = (Person)WebGroupSampleDB.getNamedVertex(friendName);

user.addEdge(newFriend, friend, EdgeKind.BIDIRECTIONAL, (short) 0);
```

Listing 3 The code for connecting two friends

As shown in Fig. 12, users have the ability to post a new status in their homepage. To add a status to graph database we need to create a new status vertex and connect the status to the user

vertex using write edge. After creating a status, it will be populated in the user's and his/her friends' news feed.

Figure 12 User interface for post a status

In order to populate the news feed section, we need to navigate the Friends Network graph database and find appropriate statuses to be shown in the news feed. Listing 4 shows the Java code for this graph navigation. As seen in the code, a navigation policy named myPolicies is used to set the maximum path depth for traversal to two. For navigation we also use VertexType qualifier which only qualifies paths that contain vertices of type status.

```
resultPrinter = new PrintPathResultsHandler(statusId);

PolicyChain myPolicies = new PolicyChain(new MaximumPathDepthPolicy(2));

Person user = (Person)WebGroupSampleDB.getNamedVertex(userName);

VertexTypes statusVertexType= new
VertexTypes(WebGroupSampleDB.getTypeId("com.infinitegraph.samples.webgroup.St
atus"));

Navigator myNavigator = user.navigate(null, Guide.SIMPLE_DEPTH_FIRST,
Qualifier.FOREVER, statusVertexType, myPolicies, resultPrinter);
```

Listing 13 The code for navigate Friends network

Part 6

CONCLUSION

The InfiniteGraph enables organizations to identify complex relationships between distributed data. Applications such as fraud detection, surveillance tools, prescription analytics, and network security information and event management (SIEM) which require real time discovery of connections between “n” degrees of distributed data will require the use of a robust and proven distributed graph database.

Advantages:

Simple Graph focused API

Automated distribution and deployment

Mostly configuration driven

Java class based persistence

Property model support

Asynchronous navigation

Indexing framework

REFERENCES

[1] http://www.objectivity.com/infinitegraph

[2] http://wiki.infinitegraph.com

[3] http://www.nosqldatabases.com/main/tag/infinitegraph

[4] http://www.objectivity.com/resources/white-papers/

APPENDIX

Source Code

Person.java

```
package com.infinitegraph.samples.webgroup;
// Import the BaseVertex class
import com.infinitegraph.BaseVertex;

public class Person extends BaseVertex
{
   private String name;
   private String screenName;
   private String password;

   public Person(String name, String password)
   {
      setName(name);
      setPassword(password);

   }

   public void setName(String name)
   {
      markModified();
      this.name = name;
   }
```

```
public String getName()
        {
    fetch();
    return this.name;
}
public void setPassword(String password)
{
    markModified();
    this.password = password;
}

public String getPassword()
        {
    fetch();
    return this.password;
}

public void setScreenName(String screenName)
{
    markModified();
    this.screenName = screenName;
}

public String getScreenName()
        {
```

```
        fetch();
        return this.screenName;
    }

    @Override
    public String toString()
    {
        fetch();
        return this.name;
    }
}
```

FriendShip.java

```
package com.infinitegraph.samples.webgroup;
// Import the BaseEdge class
import com.infinitegraph.BaseEdge;
import java.util.Calendar;

class Friendship extends BaseEdge
{
        // private Calendar date;
        public Friendship()
        {

        }
```

```
}
```

PostInfo.java

```
package com.infinitegraph.samples.webgroup;
// Import the BaseVertex class
import com.infinitegraph.BaseVertex;

public class PostInfo
{
   private String writer;
   private String message;

   PostInfo(String writer, String message){
          this.writer = writer;
          this.message = message;
   }
   public void setWriter(String writer)
   {
     this.writer = writer;
   }

   public String getWriter()
         {

      return this.writer;
   }
```

```
    public void setMessage(String message)
    {

        this.message = message;
    }

    public String getMessage()
            {
        return this.message;
    }

}
```

Status.java

```
package com.infinitegraph.samples.webgroup;

import java.util.Calendar;

//Import the BaseVertex class
import com.infinitegraph.BaseVertex;

public class Status extends BaseVertex
{
            private String message;
```

```
public Status(String message)
{
        setMessage(message);
    }

    public void setMessage(String message)
    {
        markModified();
        this.message = message;
    }

    public String getMessage()
    {
        fetch();
        return message;
    }

    @Override
  public String toString()
  {
     fetch();
     return this.message;
  }
```

```
}
```

Write.java

```
package com.infinitegraph.samples.webgroup;
// Import the BaseEdge class
import com.infinitegraph.BaseEdge;
import java.util.Calendar;

class Write extends BaseEdge
{
	private long timestamp;
	public Write(Calendar date)
	{
		setTimestamp(date.getTimeInMillis());

	}

	public Calendar getTimeStamp()
	{
		fetch();
		Calendar myCal = Calendar.getInstance();
		myCal.setTimeInMillis(timestamp);
		return myCal;
	}

	protected void setTimestamp(long timestamp)
```

```
        {
                markModified();
                this.timestamp = timestamp;
        }


        @Override
        public String toString()
        {
                fetch();
                Calendar myCal = Calendar.getInstance();
                myCal.setTimeInMillis(timestamp);
                return myCal.getTime().toString();
        }
}
```

WebGroupSampleCreate.java

```
package com.infinitegraph.samples.webgroup;
//Import InfiniteGraph packages
import com.infinitegraph.*;

// Import SLF4J logging packages
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import java.net.URL;
// Import Calendar
import java.util.Calendar;
import java.util.Scanner;

public class WebGroupSampleCreate {
	// Set up logging for the WebGroupSampleCreate class.
	static Logger logger = null;

	public static Logger getLogger() {
		if (logger == null)
			logger = LoggerFactory.getLogger(WebGroupSampleCreate.class);
		return logger;
	}

	public void createGraph() {
		// Handle for the graph database instance
		GraphDatabase WebGroupSampleDB = null;

		// Names for the graph database and its properties file
		String graphDbName = "WebGroupSample";
		String propertiesFileName = "/config/Relations.properties";
		URL url= getClass().getResource(propertiesFileName);

		try
		{
```

```
                try
                {
                        // Delete graph database if it already exists.
                        GraphFactory.delete(graphDbName, url.getPath());
                }
                catch (StorageException sE)
                {
                        WebGroupSampleCreate.getLogger().info(sE.getMessage());
                }

                // Create the graph database instance.
                WebGroupSampleCreate.getLogger().info("> Creating graph database
...");
                GraphFactory.create(graphDbName, url.getPath());


                // HINT: Call to createGraphData method here.
                // Create the vertices, edges, and connect them.


                createGraphData(GraphFactory.open(graphDbName, url.getPath()));
        }
        catch (ConfigurationException cE)
        {
                WebGroupSampleCreate.getLogger().info("> CONFIGURATION
EXCEPTION: " + cE.getMessage());
        }
```

```
            catch (StorageException sE)
            {
                WebGroupSampleCreate.getLogger().info("> STORAGE EXCEPTION: "
+ sE.getMessage());
            }

            finally {
                {
                    try
                    {
                        // Close the graph database.
                        if (WebGroupSampleDB != null)
                            WebGroupSampleDB.close();
                    }
                    catch (StorageException sE)
                    {
                        WebGroupSampleCreate.getLogger().info("> STORAGE
EXCEPTION: " + sE.getMessage());
                    }
                }
                WebGroupSampleCreate.getLogger().info("> Program completed ...");
            }
        }
        public static void createGraphData(GraphDatabase myGraph) {
           WebGroupSampleCreate.getLogger().info("> Creating Data ...");

           // Begin an update transaction.
```

```
Transaction tx = myGraph.beginTransaction(AccessMode.READ_WRITE);

// Delete objects from previous run.
myGraph.clear();

try {

  // Create vertices and add them to the graph.
  // Members
  WebGroupSampleCreate.getLogger().info("> Creating vertices ...");
  Person john = new Person("John", "1");
  myGraph.addVertex(john);

  Calendar dateTime = Calendar.getInstance();
  dateTime.set(2011, 4, 21, 9, 45, 22);
  Status status1 = new Status("How are you?");
  myGraph.addVertex(status1);
  // Commit the changes to the graph database
  tx.commit();
  WebGroupSampleCreate.getLogger().info("> Committed changes ...");
}
catch (Exception e)
{
  WebGroupSampleCreate.getLogger().info("> Exception in createGraphData: " +
e.getMessage());
  e.printStackTrace();
  tx.rollback();
```

```
            }
            finally
            {
              tx.complete();
              WebGroupSampleCreate.getLogger().info("> Finished completing transaction ...");
            }
          }
}
```

WebGroupSampleRun.java

```
package com.infinitegraph.samples.webgroup;

//Import InfiniteGraph packages
import java.net.URL;
import java.util.Calendar;
import java.util.Collections;
import java.util.Map;
import java.util.TreeMap;
import java.util.Vector;

import com.infinitegraph.*;
import com.infinitegraph.navigation.*;
import com.infinitegraph.navigation.policies.*;
import com.infinitegraph.navigation.qualifiers.*;
import com.infinitegraph.policies.PolicyChain;
```

```
//Import SLF4J logging packages
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class WebGroupSampleRun {
        static Logger logger = null;      // Set up logging

        public static Logger getLogger()
        {
          if (logger == null)
            logger = LoggerFactory.getLogger(WebGroupSampleRun.class);
          return logger;
        }

        public boolean CreateMemberNode(String MemberName, String password)
        {
          Transaction tx = null;      // Handle for the transaction
          boolean find =false;
          // Name of the existing graph and properties file, handle for graph
          String graphDbName = "WebGroupSample";
          String propertiesFileName = "/config/Relations.properties";
          GraphDatabase WebGroupSampleDB = null;
          URL url = getClass().getResource(propertiesFileName);
          try {
                WebGroupSampleDB = GraphFactory.open(graphDbName, url.getPath()); //
Open the graph
                WebGroupSampleRun.getLogger().info("> Starting an update transaction ...");
```

```
 tx = WebGroupSampleDB.beginTransaction(AccessMode.READ_WRITE);
 Person user = (Person)WebGroupSampleDB.getNamedVertex(MemberName);
  // Create vertices and add them to the graph.
  // Members
 System.out.println(password);
 if(user==null){
  find=false;
  WebGroupSampleCreate.getLogger().info("> Creating Node ...");
  Person newPerson = new Person(MemberName, password);
  WebGroupSampleDB.addVertex(newPerson);
  WebGroupSampleDB.nameVertex(MemberName, newPerson);
  WebGroupSampleCreate.getLogger().info("> Node Created ...");



 }
 else
 {
        find=true;
 }

  // Commit the changes to the graph database
  tx.commit();
  WebGroupSampleCreate.getLogger().info("> Committed changes ...");
}
catch (Exception e)
{
```

```
                    WebGroupSampleCreate.getLogger().info("> Exception in createGraphData:
" + e.getMessage());
                    e.printStackTrace();
                    tx.rollback();
                   }
                   finally
                   {
                    tx.complete();
                    WebGroupSampleCreate.getLogger().info("> Finished completing transaction
...");
                   }
          return find;
         }
         public boolean ExistMemberNode(String MemberName, String password)
         {
          Transaction tx = null;      // Handle for the transaction
          boolean find =false;
          // Name of the existing graph and properties file, handle for graph
          String graphDbName = "WebGroupSample";
          String propertiesFileName = "/config/Relations.properties";
          GraphDatabase WebGroupSampleDB = null;
          URL url = getClass().getResource(propertiesFileName);
          try {
                WebGroupSampleDB = GraphFactory.open(graphDbName, url.getPath()); //
Open the graph
                WebGroupSampleRun.getLogger().info("> Starting an update transaction ...");
                  tx = WebGroupSampleDB.beginTransaction(AccessMode.READ_WRITE);
                  Person user = (Person)WebGroupSampleDB.getNamedVertex(MemberName);
```

```
                    if(user!=null && user.getPassword().equals(password)){
                        find=false;
                    }
                    else
                    {
                        find=true;
                    }

                     // Commit the changes to the graph database
                     tx.commit();
                     WebGroupSampleCreate.getLogger().info("> Committed changes ...");
                    }
                    catch (Exception e)
                    {
                     WebGroupSampleCreate.getLogger().info("> Exception in createGraphData:
" + e.getMessage());
                     e.printStackTrace();
                     tx.rollback();
                    }
                    finally
                    {
                     tx.complete();
                     WebGroupSampleCreate.getLogger().info("> Finished completing transaction
...");
                    }
            return find;
```

```
}
public void CreateStatusNode(String memberName, String status)
{
  Transaction tx = null;      // Handle for the transaction

  // Name of the existing graph and properties file, handle for graph
  String graphDbName = "WebGroupSample";
  String propertiesFileName = "/config/Relations.properties";
  GraphDatabase WebGroupSampleDB = null;
  URL url = getClass().getResource(propertiesFileName);
  try {
        WebGroupSampleDB = GraphFactory.open(graphDbName, url.getPath()); // Open the graph
        WebGroupSampleRun.getLogger().info("> Starting an update transaction ...");
          tx = WebGroupSampleDB.beginTransaction(AccessMode.READ_WRITE);

          WebGroupSampleCreate.getLogger().info("> Creating vertices ...");

          Calendar dateTime = Calendar.getInstance();

           Status newStatus = new Status(status);
           WebGroupSampleDB.addVertex(newStatus);

           // Name some root vertices (for searching)


           Write newPost = new Write(dateTime);
```

```
                    // Create connections
                    System.out.println(memberName);
                    WebGroupSampleCreate.getLogger().info("> Connecting vertices ...");
                    Person currentMember =
(Person)WebGroupSampleDB.getNamedVertex(memberName);
                   //  System.out.println(currentMember.getName());
                    currentMember.addEdge(newPost, newStatus, EdgeKind.OUTGOING,
(short) 1);

                    WebGroupSampleCreate.getLogger().info("> Finished creating vertices ...");

                    // Commit the changes to the graph database
                    tx.commit();
                    WebGroupSampleCreate.getLogger().info("> Committed changes ...");
                   }
                   catch (Exception e)
                   {
                    WebGroupSampleCreate.getLogger().info("> Exception in createGraphData:
" + e.getMessage());
                    e.printStackTrace();
                    tx.rollback();
                   }
                   finally
                   {
                    tx.complete();
```

```
                    WebGroupSampleCreate.getLogger().info("> Finished completing transaction
...");
                }
        }
        public String checkFriend(String userName, String friendName)
        {
         Transaction tx = null;      // Handle for the transaction

         // Name of the existing graph and properties file, handle for graph
         String graphDbName = "WebGroupSample";
         String propertiesFileName = "/config/Relations.properties";
         GraphDatabase WebGroupSampleDB = null;
         URL url = getClass().getResource(propertiesFileName);
         Person friend=null;
         checkFriendResultsHandler friendCheckHandler=null;
               try {
                        WebGroupSampleDB = GraphFactory.open(graphDbName,
url.getPath());

              WebGroupSampleRun.getLogger().info("> Starting an update transaction ...");
                 tx = WebGroupSampleDB.beginTransaction(AccessMode.READ_WRITE);

                  WebGroupSampleCreate.getLogger().info("> Creating vertices ...");
                 friend = (Person)WebGroupSampleDB.getNamedVertex(friendName);
                 Person user = (Person)WebGroupSampleDB.getNamedVertex(userName);
                 friendCheckHandler = new checkFriendResultsHandler();
             PolicyChain myPolicies = new PolicyChain(new MaximumPathDepthPolicy(1));
```

```
                VertexIdentifier myVertexId = new VertexIdentifier(friend);

                    Navigator myNavigator = user.navigate(null, Guide.SIMPLE_DEPTH_FIRST,
Qualifier.FOREVER, myVertexId, myPolicies, friendCheckHandler);


                     // Perform the navigation

                     myNavigator.start();

                     myNavigator.stop();

                    tx.commit();

                     WebGroupSampleCreate.getLogger().info("> Committed changes ...");

                    }

                    catch (Exception e)

                    {

                     WebGroupSampleCreate.getLogger().info("> Exception in createGraphData:
" + e.getMessage());

                     e.printStackTrace();

                     tx.rollback();

                    }

                    finally

                    {

                     tx.complete();

                     WebGroupSampleCreate.getLogger().info("> Finished completing transaction
...");


                    }

                 if(friend==null )

                              return "NO";

                         else if(friend!=null && friendCheckHandler.find==false)

                              return "YES";
```

```
                else
                        return "Friend";

        }
        public void AddFriend(String userName, String friendName)
        {
          Transaction tx = null;      // Handle for the transaction

          // Name of the existing graph and properties file, handle for graph
          String graphDbName = "WebGroupSample";
          String propertiesFileName = "/config/Relations.properties";
          GraphDatabase WebGroupSampleDB = null;
          URL url = getClass().getResource(propertiesFileName);
          try {
                WebGroupSampleDB = GraphFactory.open(graphDbName, url.getPath()); //
Open the graph
                WebGroupSampleRun.getLogger().info("> Starting an update transaction ...");
                   tx = WebGroupSampleDB.beginTransaction(AccessMode.READ_WRITE);
                   WebGroupSampleCreate.getLogger().info("> Creating vertices ...");
                   Friendship newFriend = new Friendship();
                   Person user = (Person)WebGroupSampleDB.getNamedVertex(userName);
                   System.out.println(userName);
                   System.out.println(friendName);

                   Person friend = (Person)WebGroupSampleDB.getNamedVertex(friendName);

                  System.out.println(friend.getName());
```

```
                    user.addEdge(newFriend, friend, EdgeKind.BIDIRECTIONAL, (short) 0);

                    WebGroupSampleCreate.getLogger().info("> Finished creating vertices ...");

                    // Commit the changes to the graph database
                    tx.commit();
                    WebGroupSampleCreate.getLogger().info("> Committed changes ...");
                  }
                  catch (Exception e)
                  {
                    WebGroupSampleCreate.getLogger().info("> Exception in createGraphData:
" + e.getMessage());
                    e.printStackTrace();
                    tx.rollback();
                  }
                  finally
                  {
                    tx.complete();
                    WebGroupSampleCreate.getLogger().info("> Finished completing transaction
...");
                  }
          }
          @SuppressWarnings("finally")
         public Map FindPosts(String userName)
          {
            Transaction tx = null;      // Handle for the transaction
```

```
        // Name of the existing graph and properties file, handle for graph
        String graphDbName = "WebGroupSample";
        String propertiesFileName = "/config/Relations.properties";
        GraphDatabase WebGroupSampleDB = null;
        URL url = getClass().getResource(propertiesFileName);
        PrintPathResultsHandler resultPrinter = null;
        try {
            WebGroupSampleDB = GraphFactory.open(graphDbName, url.getPath()); // Open the graph
            tx = WebGroupSampleDB.beginTransaction(AccessMode.READ_WRITE);
            long statusId = WebGroupSampleDB.getTypeId("com.infinitegraph.samples.webgroup.Status");
            System.out.println(statusId);
            resultPrinter = new PrintPathResultsHandler(statusId);
            //GraphView myGraphView = new GraphView();

//myGraphView.excludeClass(WebGroupSampleDB.getTypeId(InstantChat.class.getName()));
                // HINT: Add policies here
                PolicyChain myPolicies = new PolicyChain(new MaximumPathDepthPolicy(2));
                // HINT: Add navigation/qualification code here.
                Person user = (Person)WebGroupSampleDB.getNamedVertex(userName);
                // Create navigator
                VertexTypes statusVertexType= new VertexTypes(WebGroupSampleDB.getTypeId("com.infinitegraph.samples.webgroup.Status"));
                //Member john = (Member)WebGroupSampleDB.getNamedVertex("John");
                //VertexIdentifier myVertexId = new VertexIdentifier(john);
```

```
                    Navigator myNavigator = user.navigate(null,
Guide.SIMPLE_DEPTH_FIRST, Qualifier.FOREVER, statusVertexType, myPolicies,
resultPrinter);


                    // Perform the navigation

                    myNavigator.start();

                    myNavigator.stop();

                    // Commit the changes to the graph database

                    tx.commit();

                    WebGroupSampleCreate.getLogger().info("> Committed changes ...");

                  }

                  catch (Exception e)

                  {

                    WebGroupSampleCreate.getLogger().info("> Exception in createGraphData:
" + e.getMessage());

                    e.printStackTrace();

                    tx.rollback();

                  }

                  finally

                  {

                    tx.complete();

                    WebGroupSampleCreate.getLogger().info("> Finished completing transaction
...");

                    return resultPrinter.treeMap;

                  }

        }

        public Vector FindFriends(String userName)

        {
```

```
        Transaction tx = null;      // Handle for the transaction


        // Name of the existing graph and properties file, handle for graph
        String graphDbName = "WebGroupSample";
        String propertiesFileName = "/config/Relations.properties";
        GraphDatabase WebGroupSampleDB = null;
        URL url = getClass().getResource(propertiesFileName);
        FriendResultsHandler friendPrinter = null;
        try {
            WebGroupSampleDB = GraphFactory.open(graphDbName, url.getPath()); //
Open the graph
            tx = WebGroupSampleDB.beginTransaction(AccessMode.READ_WRITE);
            long statusId =
WebGroupSampleDB.getTypeId("com.infinitegraph.samples.webgroup.Person");
            friendPrinter = new FriendResultsHandler(statusId);
            PolicyChain myPolicies = new PolicyChain(new MaximumPathDepthPolicy(1));
            System.out.println("hiiiiiiiiiiii");
               Person user = (Person)WebGroupSampleDB.getNamedVertex(userName);
                // Create navigator
                VertexTypes statusVertexType= new
VertexTypes(WebGroupSampleDB.getTypeId("com.infinitegraph.samples.webgroup.Person"));
                //Member john = (Member)WebGroupSampleDB.getNamedVertex("John");
                //VertexIdentifier myVertexId = new VertexIdentifier(john);
                Navigator myNavigator = user.navigate(null,
Guide.SIMPLE_DEPTH_FIRST, Qualifier.FOREVER, statusVertexType, myPolicies,
friendPrinter);


                // Perform the navigation
                myNavigator.start();
```

```
                    myNavigator.stop();
                    // Commit the changes to the graph database
                    tx.commit();
                    WebGroupSampleCreate.getLogger().info("> Committed changes ...");
                  }
                  catch (Exception e)
                  {
                    WebGroupSampleCreate.getLogger().info("> Exception in createGraphData:
" + e.getMessage());
                    e.printStackTrace();
                    tx.rollback();
                  }
                  finally
                  {
                    tx.complete();
                    WebGroupSampleCreate.getLogger().info("> Finished completing transaction
...");
                    return friendPrinter.friends;
                  }
          }
}

class PrintPathResultsHandler implements NavigationResultHandler
{
        public Map<Calendar, PostInfo> treeMap;
   private Logger logger = null;
   long statusId;
```

```
PrintPathResultsHandler(long statusId)
{
        this.treeMap = new TreeMap<Calendar, PostInfo>(Collections.reverseOrder());
        this.statusId = statusId;
        logger = LoggerFactory.getLogger(PrintPathResultsHandler.class);
}
@Override
public void handleResultPath(Path result, Navigator navigator) {

        logger.info("FOUND MATCHING PATH: ");
        String path = result.get(0).getVertex().toString();
        Person person = (Person)result.get(0).getVertex();
        // For h in p
    for(Hop h : result)
    {
        if(h.hasEdge())
        {
                System.out.println(h.getVertex().getTypeId());
                System.out.println(statusId);
                if(h.getVertex().getTypeId()==statusId){
                        Status status = (Status)h.getVertex();
                                Write post = (Write)h.getEdge();
                                PostInfo postInfo = new PostInfo(person.getName(),
status.getMessage());
                                treeMap.put(post.getTimeStamp(), postInfo);
                }
                else
```

```
                {
                        person = (Person)h.getVertex();
                }
                path = path + " < " + h.getEdge().toString() + " > " + h.getVertex().toString();
            }
        }
        logger.info("{}", path);
    }
    @Override
    public void handleNavigatorFinished(Navigator navigator) {}
}

class FriendResultsHandler implements NavigationResultHandler
{
        public Vector friends = new Vector();
    private Logger logger = null;
    long statusId;
    FriendResultsHandler(long statusId)
    {
        this.statusId = statusId;
        logger = LoggerFactory.getLogger(PrintPathResultsHandler.class);
    }
    @Override
    public void handleResultPath(Path result, Navigator navigator) {

        logger.info("FOUND MATCHING PATH: ");
```

```
        String path = result.get(0).getVertex().toString();

        // For h in p
      for(Hop h : result)
      {
        if(h.hasEdge())
        {

                if(h.getVertex().getTypeId()==statusId){
                        Person person = (Person)h.getVertex();
                                friends.add(person.getName());
                }
                path = path + " < " + h.getEdge().toString() + " > " + h.getVertex().toString();
        }
      }
      logger.info("{}", path);
   }
   @Override
   public void handleNavigatorFinished(Navigator navigator) {}
}
class checkFriendResultsHandler implements NavigationResultHandler
{
        public boolean find=false;
   private Logger logger = null;

   checkFriendResultsHandler()
```

```
    {

        logger = LoggerFactory.getLogger(PrintPathResultsHandler.class);
    }
    @Override
    public void handleResultPath(Path result, Navigator navigator) {

        logger.info("FOUND MATCHING PATH: ");
        String path = result.get(0).getVertex().toString();

        // For h in p
      for(Hop h : result)
      {
        if(h.hasEdge())
        {
                this.find=true;
                path = path + " < " + h.getEdge().toString() + " > " + h.getVertex().toString();

        }

      }
      logger.info("{}", path);
    }
    @Override
    public void handleNavigatorFinished(Navigator navigator) {}
}
```