University Of Southern California

Benchmarking Object-Graph Mapping of

Neo4j using TinkerPop Stack

|  |  |
| --- | --- |
| Ameya Koshti | Ankit Kalwar |
| University of Southern California | University of Southern California |
| koshti@usc.edu | kalwar@usc.edu |

May 07, 2014

# Objective

The objective of this project is to create an object-graph mapping, using TinkerPop stack, for Neo4j and benchmark it using BG Benchmark.

# Summary

We implemented BG Client for Neo4j using Object Graph Mapping to compare the performance of the database with native API calls and with objects. This included developing interfaces for each type of object namely, User, Resource and Manipulation, benchmarking the BG Client with different loads , image sizes, join behavior and impact of updates.

# Test Environment

The Benchmarking environment was setup as follows:

For BG Client we used a single node with 2.4GHz intel i5, 10GB RAM and Rexster was hosted on a single node with 2.4GHz intel i7, 8GB RAM.

We used Neo4j 1.8.3, Rexster 2.5.0, Blueprints 2.6.0, Frames 2.5.0

# Description of the project

We intended to benchmark Neo4j using OGM implementation. We used Blueprints interface for this purpose instead of Spring Neo4j as Blueprints is a generic graph API which provides a common set of interfaces to allow developers to plug-and-play their graph database backend. Within the TinkerPop software stack, Blueprints serves as the foundational technology. We have used the following from TinkerPop stack:

1. Frames: An object-to-graph mapper

**Frames** exposes any Blueprints graph as a collection of interrelated domain objects. With Frames, it’s easy to ensure that data within a graph is respective of a schema represented as a collection of annotated Java interfaces.

We developed the interfaces for User, Resource and Manipulation (based on BG Schema). These basically contained setters and getters for properties of the nodes and the relationship edges between nodes. We developed a Blueprints Client for BG Benchmark for the 11 social actions. The interface was invoked while node creation and their properties were set based on BG’s default values. These interfaces were also invoked while accessing a particular vertex. These objects interacted with the embedded database to perform the social actions. We found implementation using Frames API much more convenient and straightforward compared to native Neo4j API mainly because we can define adjacencies and incidences in the interface and Blueprints takes care of implementation. As such, the code to be written by developer reduces.

1. Rexster: A graph server

Rexster is a graph server that exposes any Blueprints graph through REST. We were able to implement a sample Neo4j database using Rexster with Native Blueprint API. We were also able to implement using Frame Kibbles which exposes graph elements through a frame in the REST API. But we were unable to host Neo4j database on Rexster for BG Benchmark as it kept failing for multiple actions mainly due to  Rexster Extension never really kept up with the development of Frames itself as there was not much interest from the community. We were receiving blank responses from Rexster Server and hence we decided to discontinue with further implementation using kibbles and opted for the embedded Neo4j using Blueprints.

# BG Benchmarking results

The time taken to load the data is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Image Availability** | **Image Size** | **Threadcount to load data** | **Load Time (seconds)** |
| False | NA | 1 | 20000 |
| 10 | 19950 |
| 100 | 19970 |
| True | 12 kb | 1 | 20020 |
| 10 | 20000 |
| 100 | 20000 |
| True | 500 kb | 1 | 20100 |
| 10 | 20070 |
| 100 | 20060 |

The load time for each of these cases was fairly similar and large as we had start and shutdown the database for creation of each object. This was necessary as we observed that the objects were not persisting in the database when we had a single global instance of the database. Hence, the load time was similar and large as lot of time was taken up in starting and shutting down the database after each object creation.

After loading the data we benchmarked the Blueprints Frame client. We performed benchmarking on ViewProfile action, ListFriends action and Symmetric updates. The maximum execution time for each of these experiments was 180 seconds. The results obtained for the results are as follows:

1. **ViewProfile Action** – We used 10 threads to load the data store and varied the size of image and threadcount.

|  |  |  |  |
| --- | --- | --- | --- |
| **Benchmarking Phase Threadcount** | **Overall Throughput (Actions/second) – No Images** | **Overall Throughput (Actions/second) – 12kb Images** | **Overall Throughput (Actions/second) – 500kb Images** |
| 1 | 13888.7121 | 16926.6788 | 8716.7656 |
| 10 | 21467.5282 | 34682.8866 | 12208.5678 |
| 100 | 26347.6892 | 23732.1289 | 6926.6754 |

For the experiment 12kb and 500kb images, we see that there is a drop in throughput when threadcount is increased from 10 to 100. This can be attributed to increase in threads which in turn creates an overhead for thread synchronization in accessing these images.

1. **ListFriends** Action – We used 10 threads to load the data and varied the number of friends per user and the image sizes.
2. With no images:

|  |  |  |  |
| --- | --- | --- | --- |
| **Benchmarking Phase Threadcount** | **Overall Throughput (Actions/second) – 10 friends/user** | **Overall Throughput (Actions/second) – 20 friends/user** | **Overall Throughput (Actions/second) – 50 friends/user** |
| 1 | 106479.6858 | 118888.8068 | 111902.6601 |
| 10 | 126709.5282 | 143765.4451 | 137655.6756 |
| 100 | 113084.1961 | 106391.2311 | 106897.1922 |

* ListFriendsAction (no images, 10 friendsperuser) OVERALL THROUGHPUT (Actions/sec)
* ListFriendsAction (no images, 20 friendsperuser) OVERALL THROUGHPUT (Actions/sec)
* ListFriendsAction (no image, 50 friendsperuser) OVERALL THROUGHPUT (Actions/sec)

For the above three experiments we observed similar trends in throughput. They gave incrementally better throughput on increasing the threadcount, however, we see a drop in throughput when threads were increased from 10 to 100. This maybe because when we have too many threads they are waiting for each other to finish accessing the same members.

We got similar throughputs even though numbers of friends have increased. This is because we have Adjacency implemented in such a way that it gives us all the friends of a user with a single access.

1. With 12kb images:

|  |  |  |  |
| --- | --- | --- | --- |
| **Benchmarking Phase Threadcount** | **Overall Throughput (Actions/second) – 10 friends/user** | **Overall Throughput (Actions/second) – 20 friends/user** | **Overall Throughput (Actions/second) – 50 friends/user** |
| 1 | 117773.6758 | 118673.2139 | 126892.5891 |
| 10 | 139797.4519 | 146859.8983 | 153214.9642 |
| 100 | 112850.6471 | 114598.0503 | 133678.8977 |

* ListFriendsAction (12kb images, 10 friendsperuser) OVERALL THROUGHPUT (Actions/sec)
* ListFriendsAction (12kb images, 20 friendsperuser) OVERALL THROUGHPUT (Actions/sec)
* ListFriendsAction (12kb images, 50 friendsperuser) OVERALL THROUGHPUT (Actions/sec)

For the above three experiments we observed similar trends in throughput. They gave incrementally better throughput on increasing the threadcount, however, we see a drop in throughput when threads were increased from 10 to 100. This maybe because when we have too many threads they are waiting for each other to finish accessing the same members.

We got similar throughputs even though numbers of friends have increased. This is because we have Adjacency implemented in such a way that it gives us all the friends of a user with a single access.

1. **Updates** – We used 10 threads to load the data and varied the percentage of updates using 3 different workloads. There were no images inserted while performing these experiments. Blueprints commits data only when we shut down the graph database but if we shutdown graph, we got deadlock errors as other threads were trying to access the graph. After a lot of debugging and due limited online support, we were not able to get past this issue. While in the debug mode of Eclipse, we did not face any problems so we were no able to replicate the same error in actual run.
2. Symmetric Very Low Update Action:

|  |  |  |
| --- | --- | --- |
| **Benchmarking Phase Threadcount** | **Overall Throughput (Actions/sec)** | **Overall Staleness (staleReads/totalReads)** |
| 1 | 1133.9412 | 0 |
| 10 | NA | NA |
| 100 | NA | NA |

1. Symmetric Low Update Action:

|  |  |  |
| --- | --- | --- |
| **Benchmarking Phase Threadcount** | **Overall Throughput (Actions/sec)** | **Overall Staleness (staleReads/totalReads)** |
| 1 | 583.2175 | 0 |
| 10 | NA | NA |
| 100 | NA | NA |

1. Symmetric High Update Action:

|  |  |  |
| --- | --- | --- |
| **Benchmarking Phase Threadcount** | **Overall Throughput (Actions/sec)** | **Overall Staleness (staleReads/totalReads)** |
| 1 | 398.6433 | 0 |
| 10 | NA | NA |
| 100 | NA | NA |

# Comparison between Native API and Blueprints

# Future Activities

We will be benchmarking other graph databases using the same Client by plugging in a different Blueprints supported graph database such as OrientDB, Titan, DEX, and InfiniteGraph.

We are also planning of preparing a detailed documentation for OGM using Blueprints for BG Benchmark illustrating various issues we faced while developing this client.

Once documentation of latest Blueprints release is updated, we will do the BG benchmarking for the latest Neo4j version.