**Albano, Jones, Tournour rev. 1.0.0-2015**

(Revision 2015-03-29)

**Comparison of**

**Open Source Databases in Completing Common Queries**

Sponsor

**The Department of Electrical, Computer, Software & Systems Engineering at**

**Embry-Riddle Aeronautical University**

Released March 29, 2015

**Stephen Jones**

**Justin Albano**

**Dominick Tournour**

**Abstract:** Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean sit amet dolor turpis. Nam posuere lorem nibh, nec posuere lorem ultrices et. Proin est diam, volutpat nec leo ac, congue ultricies odio. Fusce turpis sapien, porta sed nunc eget, interdum dictum odio. Vestibulum id est id lacus feugiat dictum. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec viverra augue orci, ac posuere elit interdum id. Vestibulum justo orci, suscipit non elit sed, placerat consectetur mi. Vivamus et odio ullamcorper, semper lorem nec, auctor urna. Proin risus nisi, ullamcorper a varius eget, elementum vel lorem. Maecenas justo ligula, dignissim et diam et, rhoncus lobortis erat. Nam molestie lorem ac mauris blandit eleifend. Nunc gravida sodales nisl, sed eleifend ante dapibus vitae. Duis convallis quam sit amet rutrum placerat. Aenean blandit in elit eu luctus. Maecenas nec tortor vitae leo rhoncus placerat.

Department of Electrical, Computer, Software & Systems Engineering

Senior Design 2013-2014

Embry-Riddle Aeronautical University

600 S. Clyde Morris Blvd.

Daytona Beach, FL 32114

This document is property of the Stephen Jones, Justin Albano, and Dominick Tournour, 2015, at Embry-Riddle Aeronautical University and may not be reproduced in any form or altered without the permission of the aforementioned authors.

**Contents**

[1. Graph Database Analysis 2](#_Toc415420332)

[1.1 System under Test & Component under Study 2](#_Toc415420333)

[1.2 Selected Workloads 2](#_Toc415420334)

[1.3 Methodology 2](#_Toc415420335)

[References 2](#_Toc415420336)

# Graph Database Analysis

With the advent of more ubiquitous social networking applications, and the increased connectedness of once-strangers, the demand for graph databases has in turn increased. What was once field steeped in mathematical rigor and formulae is now a part of the daily life. The main advantage of graph databases over other database types is its focus on the connections between data rather than the data itself. In essence, graph databases treat the relationships between data as first-class entities, while the data itself is thought of as secondary entities. For example, in a social network, the names and birthdays of the people included in the social graph are not as important as the relationships, such as friendship and marriage, between the people in the graph. Therefore, as this discussion of graph databases progresses, the focus of this study will be on the relationships of the graph, rather than the data within.

Given that graph databases have emerged so quickly in the software industry as a result of a pragmatic need, the workloads that the graph databases in this study will be subjected to will focus on practical use cases, such as those seen in Social Networking Applications (SNAs). Using this perspective, the goal of this study is to find the open source graph database that outperforms the others in a practical environment. While other studies, such as [cite] focus on the theoretical performance gained through academic studies of various graph database, this paper focuses on the performance of various graph databases when executing algorithms that are likely to be seen in a modern deployment environment.

As stated in [Alb15], it is difficult to pin-point a single, all-encompassing graph database solution. Therefore, the goal of this paper is not to find the best open source graph database, in absolute terms, but rather, start with a pool of the most common open source graph databases, and upon subjecting each to practical use cases, find the graph database that performs the best with these specific workloads. Likewise, the weighting of each category (in this case, each use case) will be equal, and therefore, the graph database that performs best on average will be selected as the highest performing open source graph database.

## System under Test & Component under Study

Throughout the analysis of the graph databases presented in this section of this paper, the system under test is the computer of the analyst performing the analysis. In this case, the specifications of this machine are

* **Operating system**: Windows 8.1 Professional x64
* **Processor**: Intel Core i7-5500U Processor, 4M Cache, up to 3.00 GHz (5th generation)
* **Memory**: 8GB (2x4GB) Dual Channel DDR3L 1600MHz
* **Hard drive**: 1TB (5,400 RPM) SATA

Accompanying the hardware specifications of the machine, the following software was used to develop and execute the workloads presented in this section:

* **Java:** JDK 1.8.0 x64 (build 1.8.0\_40-b26)
* **Eclipse**: Eclipse EE for Java Developers Luna Service Release 2 (4.4.2, build 20150219-0600)

Being that the software written to exercise the selected graph databases are encompassed in an Apache Maven project, the specific versions of the software used in the application executing the workloads can be found in the Project Object Model (POM) file of the project. The source code, including the POM file, for this project can be found at https://github.com/albanoj2/se655\_final\_project/tree/master/individual/graph/java.

Within the system under test, the components under study are the open source graph databases selected for this performance analysis; namely,

* Neo4j[[1]](#footnote-1)
* Raxster[[2]](#footnote-2)
* OrientDB[[3]](#footnote-3)
* Tinker[[4]](#footnote-4)

## Selected Workloads

In accordance with finding the highest performing graph database in terms of practicality, the following algorithms were selected as the use cases that each database must execute:

1. **Friends-of-Friends**: The common SNA use case, where all of the friends of a person are found, and then the friends of these friends are found. This algorithm is implemented by collecting all friends for a starting node, and then iterating through each of these friends, and in turn, finding the friends of these friends. This algorithm is then repeated for each of the nodes in the graph. A friend of some node, A, is defined as any node, B, where a *knows* relationship originates from node A, is directed outward, and terminates at node B.
2. **Get property**: A simple use case that retrieves a property from each of the nodes in the graph. Commonly, properties in property graph databases are represented as key-pairs, but do to the focus on relationships, some graph databases have difficulty in retrieving properties on a large scale. This algorithm simply iterates through each of the nodes in the graph and obtains the value for a given property (with a constant property key, i.e., the property obtained from each node has the same key).
3. **Shifting relationships**: A use case in which every outgoing relationship from every nodes in a graph is shifted to the next sequential node in the graph. For example, if node A has a relationship that terminates at node D, this relationship is removed and replaced with a relationship originating at node A and terminating at node E (the next sequential node after node D). If a relationship terminates at the final node in a graph (for example, if there are 20 nodes in the graph, and a relationship terminates at node 20, of which there is no node that follows sequentially), then the new relationship terminates at the first node in the graph. This algorithm is intended to exercise the capability of a graph database to make updates (removals and writes) on a large scale.

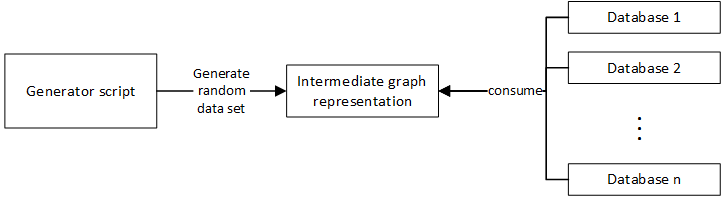
For each of the algorithms described above, four variations of the graph size is used:

1. **Small**: 1,000 nodes, where each node has a maximum of 50 outgoing relationships
2. **Medium**: 10,000 nodes, where each node has a maximum of 50 outgoing relationships
3. **Large**: 100,000 nodes, where each node has a maximum of 50 outgoing relationships
4. **Very large**: 1,000,000 nodes, where each node has a maximum of 50 outgoing relationships

While the number of nodes in these workloads are fixed, the number of outgoing relationships is random, with a maximum of 50 relationships per node. This randomized approach it taken to ensure that unnatural patterns do not form in the shape and connectivity of the graph (such as would be case if each nodes had a constant number of outgoing relationships). For each of these algorithms and workloads, a total of thirty replications are taken, ensuring that a standard Z-distribution can be used to model the results of the samples for each graph databases. Using this technique, a three-factor analysis of variance (ANOVA) can be used to discover the highest performing graph database.

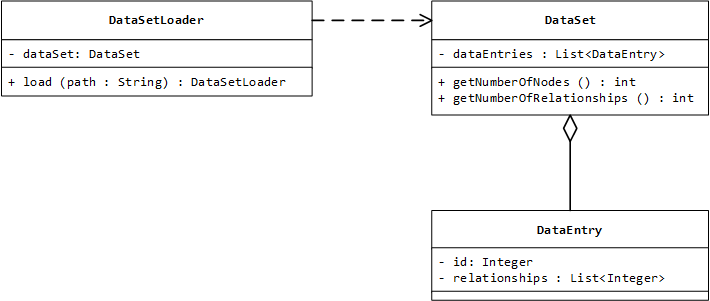
## Methodology

In order to exercise each of the selected databases with the described workloads, a standardized workload is needed. As previously stated, the workloads used to analyze each graph database should be random, but this leads to a challenge: Creating a random, but consistent workload. In order for the comparison between databases to be unbiased, the workload used by each database must be the same; but, in order to ensure that patterns do not form in the workload, the workload must be random. In essence, the workload used by each graph database must be the same, random dataset. In order to solve this challenge, a workload generation script is used to create an intermediate, implementation-independent representation of a graph. This intermediate representation is then consumed by each database, which creates a graph in the native representation of the database. This process is illustrated in **Figure 1**.



**Figure 1.** Using an intermediate graph representation, the same, random data set can be used by each of the graph databases, ensuring an unbiased comparison of each database.

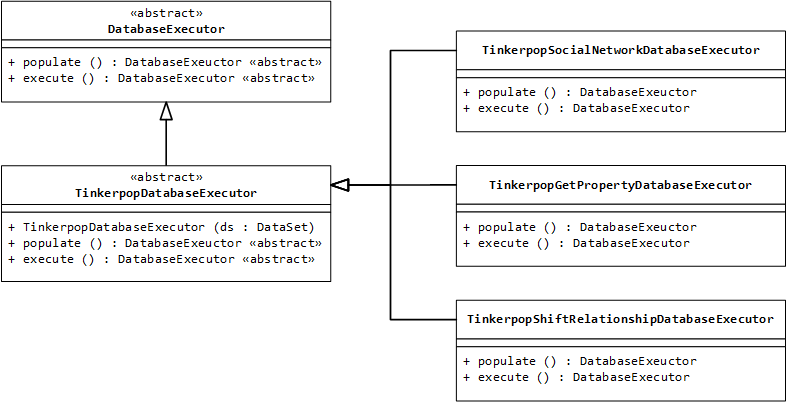
In order to consume this intermediate graph representation, an Application Programming Interface (API) is created that consumes the intermediate graph representation and produces an object structure that allows an adapter for each database to use this object structure to populate its corresponding database. This object structure is illustrated below in **Figure 2**.



**Figure 2.** By creating an object structure representing the intermediate graph structure, each database can use the structure to populate its corresponding graph.

When the application exercising the databases starts, a DataSetLoader is created and the load(path : String) method is called, supplying the path of the intermediate graph representation file. Once the data set is loaded, the data set can be accessed using the getDataSet() : DataSet method of the DataLoaderLoader. Given this DataSet object, clients can iterate through each DataEntry object, which represents a node in the graph. This DataEntry object specifies the identifier of the node, as well as the identifiers of the nodes to which this node is related. For example, if the identifier for a DataEntry object is set to 2, and the relationships attribute is set to a list containing the value 4 and 5, the node with the identifier 2 has two relationships: (1) one originating at node 2 and terminating at node 4, and (2) one originating at node 2 and terminating at node 5. Given the four data set sizes previously described, four DataSetLoaders are created in order to create the four DataSet objects.

In order to create the algorithms used by each of the graph databases, the Tinkerpop[[5]](#footnote-5) Blueprints API is used, as suggested by [Jou13]. The Tinkerpop Blueprints API abstracts the details of each data implementation allowing generic graph algorithms to be written and executed against any supported graph database without concerning the developer with the details of each database. Using this technique, an abstract base class called DatabaseExecutor is created, with abstract populate() and execute() methods. In order to execute each of the prescribed algorithms, a concrete class for each algorithm is created, overriding the populate() and execute() to include the logic specific to each algorithm. This technique is illustrated below in **Figure 3**.



**Figure 3.** Using the Tinkerpop Blueprint API, a DatabaseExecutor is created for each algorithm, reducing the boilerplate code required to exercise each of the databases.

While the overridden methods do not include a parameter for the data set object representing the graph to populate, the constructor of each of the concrete implementation classes accepts a data set object as a parameter. This object is stored and then used within the populate() and execute() method. One of each concrete implementation class is instantiated per each algorithm for each graph database selected. For example, four TinkerpopSocialNetworkDatabaseExecutor objects are instantiated: (1) one for Neo4j, (2) one for Raxster, (3) one for OrientDB, and (4) one for Tinker. This is repeated for each of the DatabaseExecutor concrete class. Thus a total of twelve concreate classes are instantiated (for three algorithms and four databases).

## Results

[Results to follow upon completion of experiments]

# References

[Alb15] Albano, Justin. “A Comparison of Open Source Graph Databases.” *Department of Electrical, Computer, Software, and Sytems Engineering at Embry-Riddle Aeronautical University.* 22 Mar. 2015. Print. 29 Mar. 2015.

[Jou13] Jouili, Salim, and Valentin Vansteenberghe. "An Empirical Comparison of Graph Databases." *Social Computing (SocialCom)* (2013): pg 708-715. *EURA NOVA*. IEEE, 8-14 Sept. 2013. Web. 1 Feb. 2015. <http://euranova.eu/upl\_docs/publications/an-empirical-comparison-of-graph-databases.pdf>.

Acryonyms

* Analysis of variance (ANOVA)
* Social Networking Application (SNA)

1. http://neo4j.com/ [↑](#footnote-ref-1)
2. https://github.com/thinkaurelius/titan/wiki/Rexster-Graph-Server [↑](#footnote-ref-2)
3. http://www.orientechnologies.com/orientdb/ [↑](#footnote-ref-3)
4. https://github.com/tinkerpop/blueprints/wiki/TinkerGraph [↑](#footnote-ref-4)
5. http://www.tinkerpop.com/ [↑](#footnote-ref-5)