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

(Revision 2015-04-19)

**Comparison of**

**Open Source Databases in Completing Common Queries**

Sponsor

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

**Embry-Riddle Aeronautical University**

Released April 19, 2015

**Stephen Jones**

**Justin Albano**

**Dominick Tournour**

**Abstract:** In one form or another, databases are a ubiquitous aspect of nearly all software systems. While persistent storage of data is a general need among a majority of software systems, the manner which this data is stored can vary greatly, depending on the category of database used. In the age of cloud computing and distributed systems, three categories of databases have battled to the forefront: (1) SQL databases, (2) NoSQL databases, and (3) graph databases. Although many absolute claims have been made about the advantage of one category over another, a database comparison in not this simple. In an attempt to provide insight into the performance of databases of each type, as well as a comparison of each category of database against one another, this paper uses statistical techniques to select the highest performing database from a group of databases of each category. Using these selected databases as a representative sample for all databases of its respective category, an overall comparison is made between SQL, NoSQL, and graph databases. The results of these comparisons are then statistically analyzed. The category represented by the winning database is then selected as the best performing database category. While this selection provides a general, *best in its class* title to one of the examined categories, the selection process also provides insight in the nature of each database category, as well as the advantages and disadvantages of each category in practical software applications.

Department of Electrical, Computer, Software & Systems Engineering

SE655, Spring 2015

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**

[Contents ii](#_Toc417233057)

[1. Introduction 1](#_Toc417233058)

[1.1 Analysts 1](#_Toc417233059)

[1.2 Metrics & Factors 1](#_Toc417233060)

[1.3 Performance Analysis in the Context of Databases 2](#_Toc417233061)

[1.4 Intended Audience 2](#_Toc417233062)

[1.5 Further Reading 3](#_Toc417233063)

[1.6 Structure of Document 3](#_Toc417233064)

[2. Graph Database Analysis 4](#_Toc417233065)

[2.1 System under Test & Component under Study 4](#_Toc417233066)

[2.2 Benchmarks & Workloads 5](#_Toc417233067)

[2.3 Methodology 6](#_Toc417233068)

[2.4 Results 8](#_Toc417233069)

[2.5 Conclusion 10](#_Toc417233070)

[3. NoSQL Database Analysis 11](#_Toc417233071)

[3.1 System under Test & Component under Study 12](#_Toc417233072)

[3.2 Benchmarking and Workloads 12](#_Toc417233073)

[3.3 Methodology 15](#_Toc417233074)

[3.4 Results 15](#_Toc417233075)

[3.5 Conclusion 17](#_Toc417233076)

[4. Relational Database Analysis 18](#_Toc417233077)

[4.1 System under Test & Component under Study 18](#_Toc417233078)

[4.2 Benchmarks & Workloads 18](#_Toc417233079)

[4.3 Collection of Data 20](#_Toc417233080)

[4.4 Results 21](#_Toc417233081)

[4.5 Concerns 23](#_Toc417233082)

[5. Group Comparison of Databases 24](#_Toc417233083)

[5.1 System under Test & Component under Study 24](#_Toc417233084)

[5.2 Benchmarks & Workloads 25](#_Toc417233085)

[5.3 Methodology 26](#_Toc417233086)

[5.4 Results 27](#_Toc417233087)

[6. Alternative Methods 30](#_Toc417233088)

[7. Conclusion 31](#_Toc417233089)

[Acronyms & Abbreviations 33](#_Toc417233090)

[Appendix A: Statistical Calculations 34](#_Toc417233091)

[Individual Graph Database Analysis 34](#_Toc417233092)

[Individual NoSQL Database Analysis 36](#_Toc417233093)

[Individual Relational Database Analysis 42](#_Toc417233094)

[Group Database Analysis 56](#_Toc417233095)

# Introduction

Databases have always been a very important part of most computer applications. Whether it is a small SQLite database operating on a mobile device or a giant graph database holding the information of millions of social network users, databases are used everywhere. Whenever an entity is used so widely throughout the world, it begs the question, “Which one is the best?” Unfortunately, that question is not a fair one to answer, since there exists so many different types of databases: relational, key-value, graph, object-oriented, document, as well as many others. With numerous, diverse database types on the market, it is difficult to know which type to use for a specific application scenario. Furthermore, once the correct database type is chosen the next question is, “Which one [product] is the best?”

In order to answer these questions, three databases categories will be selected for comparison: (1) relational, (2) graph, and (3) No Structured Query Language (NoSQL). Therefore, the objective of this study is to ascertain the highest performing database, both in terms of its performances against other databases within its category (how well does it compare against databases within its category, i.e., relational, graph, or NoSQL), as well as against databases of different categories.

## Analysts

The point of this research would be to answer both questions presented above, with the limitation that all databases being tested are open source (due to the availability of these databases and the free licensing that protect these products). The database categories will be split amongst the three analysts, based on the research conducted by each member for the individual paper:

* **Dominick Tournour**: Relational databases
* **Justin Albano**: Graph databases[[1]](#footnote-2)
* **Stephen Jones**: NoSQL databases (key-value store, document store and column family)

## Metrics & Factors

The analysis of these databases will be divided into two stages: (1) a comparison of the best database(s) within each category, and (2) a comparison between the databases selected within stage (1). Each project team member will independently discuss the advantages and disadvantages of the assigned database design. We will then independently run performance analysis on the most popular databases within each category, using common queries to test different metrics (see below for a complete list of the metrics used to complete this analysis).

Based on each of our findings, the top one or two databases from each category (relational, graph, and NoSQL) will be selected for further analysis and compared to one another using a similar set of metrics, as well as new queries that will incorporate a combination of strengths and weaknesses of each database. This second round of performance analysis should be interesting to interpret because each database may perform better in while executing the queries designed for its implementation but will struggle with queries that target the strengths of other databases. For example, a graph database may outperform a relational database and NoSQL database while searching for friends-of-friends, but may falter when attempting to read the value of a property, compared to a relation or NoSQL database.

In both stages of performance analysis the metrics being measured will be:

* **Execution time**: measuring the fixed-computation execution time from the start of a query to the completion of the query, using the start and end timestamps, respectively

In order to obtain usable results using these metrics, a set of preselected queries will be run with each database, and the resulting value of the metrics above will be recorded. Each query will be executed with increasing data set sizes, allowing the analysts to observe the performance of each database on both a small and large scale. The data set size for the second stage of analysis (due to the discrepancies in the nature of the data represented in each database) will be scaled in terms of the abstract entities in the desired result.

For example, if a friends-of-friends algorithm is used, the data set scale will be in terms of the number of friends each person in the database has, rather than directly in terms of rows, nodes, and key-value pairs for relational, graph, and NoSQL databases, respectively (although, the size of the former set will have a direct influence on the size of the latter data set). This variation in data sets ensures that the results of the analysis are understood in terms of a wide range of data sets, not simply a single database size.

## Performance Analysis in the Context of Databases

Performance analysis is a common tool used to compare various aspects of computer systems, and while there are many commonalities in this field, each aspect of a computer system presents its own difficulties and intricacies. In the context of this paper, the execution times of various databases are analyzed using differing workloads and data set sizes. While this is similar the performance loading of a computer system in general, there are many inherit details in the comparison of databases; namely,

* **Transactional vs. non-transactional**: Some databases implement a transactional interface that ensures the consistency of data in the database, while other database use less stringent means of ensuring consistency (or place the responsibility of consistency on the client interacting with the database).
* **Caching vs. non-caching**: Many databases, especially databases intended for large-scale, distributed systems, implement some form of caching, which provides an increase in performance the longer a single data set is accessed. The effectiveness of these caches vary based on multiple parameters, including the cache size, the locality of data in the database, and the nature of the accesses to the data in the database.
* **Data-centric vs. relationship-centric**: Some database, such as SQL and NoSQL, focus on the data stored in the database, while other databases, such as graph databases, focus on the relationship between data, providing benefits when completing some use cases and disadvantages when executing others.

Although there are many more idiosyncrasies, especially when viewed in the light of the many categories of databases used in industry today, this paper describes these details as they arise and when applicable. Therefore, the performance analysis details specific to SQL, NoSQL, and graph database are explored in the SQL, NoSQL, and graph database analysis sections of this paper, respectively. It is also important to note that while some databases many have the same intricacies, each section deals with these details in the manner most fitting for the database discussed in that section. For example, although the SQL databases and graph databases discussed in this paper may both employ caching, the manner in which caching is factored in the performance analysis of each type (SQL and graph) is handled according the manner most fitting for the common use cases of SQL databases and graph databases, respectively.

## Intended Audience

This paper is intended for a technical audience with experiment in performance analysis, specifically in the field of computer performance analysis. While many of the details and descriptions contained within this document are readable by the lay reader, many of the calculations and computations performed are done without accompanying explanation. For example, when Analysis of Variance (ANOVA) is completed for the overall database comparison, little accompanying explanation is provided as to how the process was conducted and the reasoning behind the selection of technique (since a two-factor is being conducted with replication): It is expected that the reader already possess the background necessary to understand these implicit details.

Likewise, it is also expected that the reader possess and understanding of databases and database concepts. For example, when a graph database is used, the term vertex and edge are not defined. Rather, it is the expectation that the reader understand that these terms are synonymous with the terms node and relationship, respectively. Likewise, when SQL statements are included, it is expected that the basic structure of the query can be understood by the reader, and therefore, an extensive description of the meaning of the query is not included (it is the assumption of the authors that given a knowledge of SQL, the reader should be able to interpreter the meaning of the query from the SQL source text).

## Further Reading

Performance analysis of computer systems, and in particular, performance analysis of database systems, is an involved topic. It is not possible for the authors of this paper to include sufficient detail in all topics discussed in this paper, and therefore, it is the strong suggestion of the authors that the interested reader use accompanying sources for greater detail and to understand the topics discussed in this paper to a greater depth. In particular, the following sources and references are suggested to the reader to gain a deeper understanding of the topics contained in and discussed in this paper:

* [Lil04]: The textbook of the SE 655 course at Embry-Riddle Aeronautical University that was used throughout both the design of the experiments and the execution of the experiments conducted within this paper. This concise textbook also provides a detailed description of many of the statistical methodologies used throughout this paper to analyze and interpret the results of the experiments conducted on each of the databases.
* [Rob13]: While property graphs are conceptually simple, there are many details in property graph databases that are required to provide the degree of functionality necessary for pragmatic applications. For example, the linked-list structuring of Neo4j is essential to its execution time characteristics under various workloads and categories of workouts (read, write, update, etc.). Likewise, the use of Atomic, Consistent, Isolated, and Durable (ACID) transactions by Neo4j plays a major role in its ability to update its internal graph structure in large, parallel environments. Therefore, it is suggested that the reader refer to [Rob13] for the technical details of not only Neo4j, but property graph databases in general.

## Structure of Document

The analysis contained in this document is divided into four categories, based on the database or database under study: Section 2 contains the performance analysis and selection of the top-performing graph database; Section 3 contains the performance study of the NoSQL databases selected for this paper, as well as the methodology used to select the top-performing NoSQL database; Section 4 contains the analysis and selection of the top-performing SQL databases; Finally, section 5 contains the comparative analysis and selection of the top-perform database. Section 5, unlike the previous three sections, results a single, “best” performing database based on the aggregate analysis and results conducted in this paper. Apart from the analysis of individual database categories and the combined database group, section 6 provides a brief overview of alternative, non-measurement methods of analyzing and comparing the performance of databases, including analytical and simulation techniques. Lastly, section 7 provides conclusive comments by the authors on the results recorded in this document, including notes about the conclusions that can be drawn from the selection of the "best” database.

This document also contains a **References** and an **Acronyms & Abbreviations** section that provide the sources used throughout this paper and a comprehensive enumeration of the acronyms and abbreviations used in this document, respectively. **Appendix A: Statistical Calculations** contains the detailed calculations and computations performed in order to analyze each individual database, as well as compare the best databases from each database category in section 5. All lengthy calculations are contained in this appendix, and therefore, this appendix should be consulted for all analytical information pertaining to the conclusions and judgments stated in this paper.

# 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[[2]](#footnote-3)
* OrientDB[[3]](#footnote-4)

## Benchmarks & 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, e.g., the property obtained from each node has the same key).

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). The actual workloads used in the experimentation for the graph databases in this section are listed in **Table 1**. For more information on the projected workload versus the actual workload, see the **Methodology** section below.

These workload values are selected in order to create three distinct categories of memory usage [Alb15]:

1. The graph fits completely in cache
2. The graph fits partially in cache and wholly in memory
3. The graph fits partially in memory and wholly in persistent storage

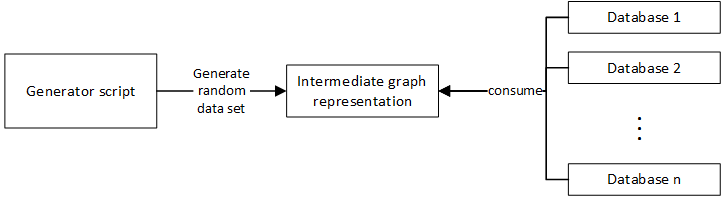
For each of these algorithms and workloads, a total of thirty replications are taken, ensuring that a standard Z-distribution can be used to model the results of the samples for each graph databases. Using this technique, a pairwise comparison of each workload can be used to discover the highest performing graph database. Note that for the graph database experiments conducted in this paper, the caching mechanism for each database is set to its default value. Therefore, caching is incorporated into the sample values recorded for each of the workloads with each of the algorithms.

Table 1: Upon generating the workloads, each with a maximum count of 50 relationships per node, a fixed number of nodes and relationships is used to exercise each of the graph databases.

|  |  |  |
| --- | --- | --- |
| **Name** | **No. Nodes** | **No. Relationships** |
| Small | 1,000 | 10,009 |
| Medium | 10,000 | 100,486 |
| Large | 100,000 | 1,001,176 |
| Very large | 1,000,000 | 10,015,575 |

## Methodology

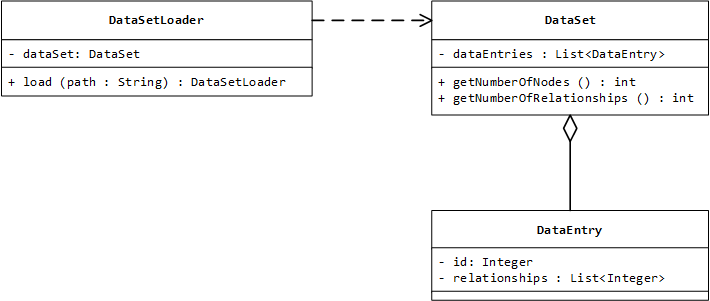
In order to exercise each of the selected databases with the described workloads, a standardized workload is needed. As previously stated, the workloads used to analyze each graph database should be random, but this leads to a challenge: Creating a random, but consistent workload. In order for the comparison 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.

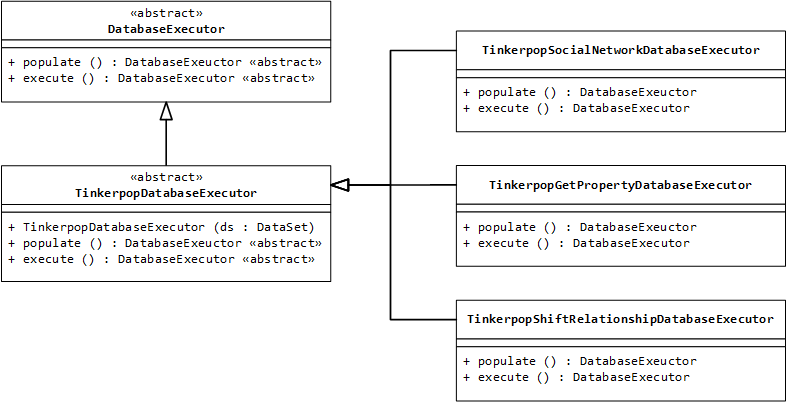
In order to consume this intermediate graph representation, an Application Programming Interface (API) is created that consumes the intermediate graph representation and produces an object structure that allows an adapter for each database to use this object structure to populate its corresponding database.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

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.



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

In order to create the algorithms used by each of the graph databases, the Tinkerpop[[4]](#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

Each of the workloads presented above was executed on the selected graphs, each with the default settings for the database and the default Java Virtual Machine (JVM) memory allocation of . Upon completion of the prescribed workloads, the sample data from each executor was collected. The complete means of the sample data for these workloads is presented in **Table 2**.

Table 2: After completion of the prescribed experiments, it was found that on a whole, Neo4j had a faster execution time than OrientDB, but Neo4j was not able to complete as many of the prescribed workloads as OrientDB.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Database** | **Algorithm** | **Small (ms)** | **Medium (ms)** | **Large (ms)** | **Very Large (ms)** |
| Neo4j | Friends-of-friends |  |  | *CNC* | *CNC* |
| Get property |  |  | *CNC* | *CNC* |
| OrientDB | Friends-of-friends |  |  |  | *CNC* |
| Get property |  |  |  | *CNC* |
| CNC: Could not complete | | | | | |

Upon executing the workloads prescribed for each database previously described, it was found that the default JVM memory size was insufficient to execute the very large workloads. Therefore, the JVM memory size was increased to , but this was also found to be insufficient to execute the very large workloads. Therefore, this workload is considered incomplete by both databases, and therefore cannot be used to compare the databases. While the JVM memory size can be further increased, it is difficult to estimate the memory required to populate each database with the prescribed workload, and thus, a maximum of was provided for each database[[5]](#footnote-6).

A similar scenario arose with Neo4j for the case of the large workload, where Neo4j was unable to load the graph with the memory allocated. Therefore, samples could not be taken for the large workload for Neo4j. Interestingly, OrientDB did not suffer from this deficiency. Although OrientDB required a great deal of time to complete the large workload, it was nonetheless able to complete it. This likely has to do with the manner that each graph is represented: Neo4j appears to require a larger memory footprint to maintain a graph when compared with OrientDB storing the same graph[[6]](#footnote-7). Thus, although OrientDB is noticeably slower than Neo4j in the comparable workloads, it is able to support graphs of a much largest size than Neo4j.

Taking into account the inability of both graphs to complete some of the prescribed workloads, the sample data for all experiments that could be completed are recorded in **Appendix A: Statistical Calculations: Individual Graph Database Analysis**. There are a few points of interest about the sample recorded during experimentation:

1. Since each database was set to its default configuration when performing the experiments, caching was enabled for both databases; therefore, the first execution time for each experiment has the highest execution time, since during this sample, the database cache is cold and must be warmed up (there are no entries currently in cache). Once this first experiment is complete, the cache is warmed up, which results in the following samples having a noticeably lower execution time.
2. Neo4j is starkly faster than OrientDB for all algorithms and graph sizes executed. Since one of the algorithms focuses on the relationships between the nodes in the graph and the other focuses on the properties of the nodes in the graph (the data, rather than the relationships between data), Neo4j is therefore more adept at executing workloads that are both relationship-read-intensive, as well as property-read-intensive (in a general sense, Neo4j performs better than OrientDB in call categories studied in this paper).

In order to illustrate the findings in point (1), **Figure 4** below depicts the execution times for Neo4j executing the friend-of-friends algorithms with the medium graph size.

**Figure 4.** Due to the overhead of warming the cache, the first execution time sample taken for Neo4j friend-of-friend algorithm is exponentially higher than that of the samples taken after the cache has warmed.

As can be seen, the first execution time in this experiment is exponentially higher than the subsequent execution times (the first execution time is , while the mean of the remaining samples is only ). Therefore, although this first value appears to be an outlier in the samples recorded, it is in fact important to include this value, since it represents the first, uncached execution of the algorithm. This recording method is common among many of the graph database benchmarks and is the suggested course of action, according to [Rob13]. While **Figure 4** only depicts one of the experiments conducted, this behavior is common among all of the experiments conducted, and is likewise reflected in the experiments using OrientDB.

In accordance with point (2), it can be seen that in all experiments taken were both Neo4j and OrientDB were able to complete the prescribed workloads, Neo4j is strikingly faster than OrientDB. The mean values for each of these pairwise experiments are illustrated below in **Figure 5**.

**Figure 5.** The execution times for OrientDB are consistently greater than those for Neo4j for all experiments in which a pairwise comparison can be made between the two databases.

As can be seen in **Appendix A: Statistical Calculations: Individual Graph Database Analysis**, the differences in execution times for each experiment (when comparing the execution times of the experiments conducted for Neo4j and for OrientDB in a pairwise manner) are statistically significant, and therefore can be used as a solid indication that Neo4j is the higher performing database between the pair.

## Conclusion

Based on the statistical significance of the difference between the execution times of Neo4j and OrientDB (for the workloads under which a pairwise comparison could be made; namely the small and medium graph sizes), it is clear that Neo4j is the higher performing graph database between the pair. Although Neo4j is unable to complete the workloads that use the large graph size and OrientDB is capable of completing these workloads, Neo4j is nonetheless the clear winner when Neo4j and OrientDB are compared head-to-head.[[7]](#footnote-8) Therefore, Neo4j will be used in the remainder of this paper as the representative graph database when used to analyze the performance relational, NoSQL, and graph databases.

# NoSQL Database Analysis

In today’s connected world, millions of concurrent users and web applications are generating massive amounts of complex and unstructured data on the internet. In order to house and process this data, it is imperative that databases are scalable. NoSQL databases are most commonly used in big data applications, as they are typically more efficient than their relational database counterparts at managing large volumes of data. NoSQL databases typically follow a distributed system model and as a result they are easily horizontally scalable and can achieve a consistent linear increase in performance as more processors are added to the distributed system. There is no official definition for the term ‘NoSQL’, however it is commonly referred to as meaning “Not Only Structured Query Language (SQL)”. NoSQL databases are typically characterized as having a non-relational structure, which is in contrast to conventional Relational Database Management Systems (RDBMSs) such as SQL.

RDBMSs maintain transaction control by ACID properties to insure transactions are reliable. ACID systems focus on the consistency and integrity of data above all other considerations. A large amount of overhead is required to achieve consistency and integrity, particularly in a distributed system. As a result, RDBMSs often degrade in performance as they scale up to the levels of scalability that are required to manage big data. NoSQL databases do not in general make ACID guarantees. They instead place a large focus on availability. The concept used with the majority of NoSQL databases is Basic Availability, Soft-state, and Eventual consistency (BASE). Basic availability allows systems to be temporarily inconsistent in order to simply the management of transactions. Soft-state means that some inaccuracy is temporarily allowed and data may change while being used to reduce the amount of consumed resources. With eventual consistency, eventually, when all service logic is executed, the system returns to a consistent state. One of the main objectives of BASE systems is allowing new data to be stored, even at the risk of being out of sync for a short period of time. They focus on keeping the process moving and dealing with the consequences at a later time.

There are four types of data models that exist for NoSQL databases [Abr14]:

* **Key-value store**: In these databases all the stored data is represented by a pair of key and value per record, meaning that each key is unique and it allows accessing record’s information, represented as value. This structure is also known as “hash table” where data retrieval is usually performed by using a key to access a value.
* **Document Store**: These databases are designed to manage data stored in documents that use different format standards, such as, XML or JSON. This type of storage is more complex in comparison to storage used by Key-value Stores and enables data querying.
* **Column Family**: The database structure of this NoSQL type is similar to the standard Relational Database Management System (RDMS) since all the data is stored as sets of columns and rows. Columns, that store related data that is often retrieved together, may be grouped.
* **Graph Database**: These databases are mostly used when the stored data may be represented as a graph with interlinked elements such as, social networking, road maps or transport routes.

Graph databases are being analyzed separately as part of this project as they are significantly different to the other NoSQL database types. There are currently over 150 NoSQL database applications available on the market and it is not clear which product has the best overall performance. This project aims to discover the best performing NoSQL database (stage 1) and then compare this best performer to the best performers of other types of databases to determine an overall best performer for common database queries.

## System under Test & Component under Study

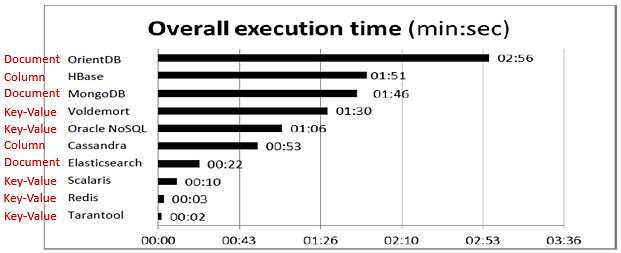
During the first stage of analysis, the system under test (SUT) will be the computer system of each analyst. For analysis of NoSQL databases, the SUT is as follows:

* **Operating system**: Ubuntu 14.04.2 LTS Server (32-bit) Virtual Machine executing on Windows 7 Home Premium SP1 64-bit host
* **Processor**: Intel Core i5-2410M @ 2.3GHz
* **Memory**: 8GB DDR3 1600MHz
* **Hard drive**: Samsung 840 EVO 250GB SSD

Once a group of databases has been selected by each analyst for each category of databases, the computer system of one of the analysts will be selected as the SUT for the second stage where the databases of each category will be compared against one another.

The component under study (CUS) will naturally be the individual database programs. [Abr14] presents results of benchmarks conducted on an Ubuntu Server VM in a centralized environment, much like the SUT to be utilized as part of this project. The paper is assessed as carrying suitable credibility, thus an attempt will be made to replicate a subset of the experiments conducted in [Abr14]. The aggregated execution times for all workloads executed in [Abr14] are displayed in **Figure 6** below.

The initial aim was to compare the top four databases (with lowest execution time), however countless compatibility issues were contended with and thus only Redis v2.8.19 and MongoDB v2.4.9 could be made functional with the Yahoo! Cloud Serving Benchmark (YCSB – discussed later) tool despite there being official client plugins hosted on the YCSB Github site at [Coo15] for most of the database applications.



**Figure 6.** The experimental execution times of various NoSQL databases, as presented in [Abr14].

## Benchmarking and Workloads

A benchmark is a surrogate program used as a standard reference for comparing performance results [Lil04]. A benchmark is ideally a perfect representation of the system workload executing on a system. Benchmarks facilitate the standardization of experiment parameters and thus aid in comparing the results of different experiments against one another. The component under study should be the only parameter that varies in an experiment. By keeping a standardized and consistent workload, this goal can be achieved.

The Yahoo! Cloud Serving Benchmark (YCSB) is a widely accepted application-program benchmark suite for analyzing the performance of NoSQL databases [Coo10b]. YCSB was utilized in the vast majority of the papers reviewed as part of the survey paper at [Jon15]. YCSB was developed in the Java language, is open-source and highly extensible. The system architecture is summarized in **Figure 7**.

Default, or ‘core’ workloads are distributed with the benchmark package, but custom workloads can be easily defined. Each workload represents a particular mix of read/write operations, data sizes, request distributions, etc, and can be used to evaluate systems at one particular point in the performance space [Coo10b]. The core YCSB workloads were developed based on the fundamental kinds of workloads web applications place on cloud data systems (a typical application of NoSQL databases). The workloads do not exactly model a particular application or set of applications, such as those workloads that may be utilized in other benchmarking software. Such benchmarks give realistic performance results for a narrow set of use cases. In contrast, the goal of YCSB was to examine a wide range of workload characteristics, in order to understand in which portions of the space of workloads systems performed well or poorly. For example, some systems may be highly optimized for reads but not for writes, or for inserts but not updates, or for scans but not for point lookups. The workloads in the core YCSB package were chosen to explore these tradeoffs directly. The workloads in the core package are a variation of the same basic application type. In this application, there is a table of records, each with F fields. Each record is identified by a primary key, which is a string like “user234123”. Each field is named field0, field1 and so on. The values of each field are a random string of ASCII characters of length L [Coo10b]. As part of stage 1 of this project, 1,000 byte records will be utilized by using F = 10 fields, each of L = 100 bytes, as was performed in [Coo10b] and [Abr14].

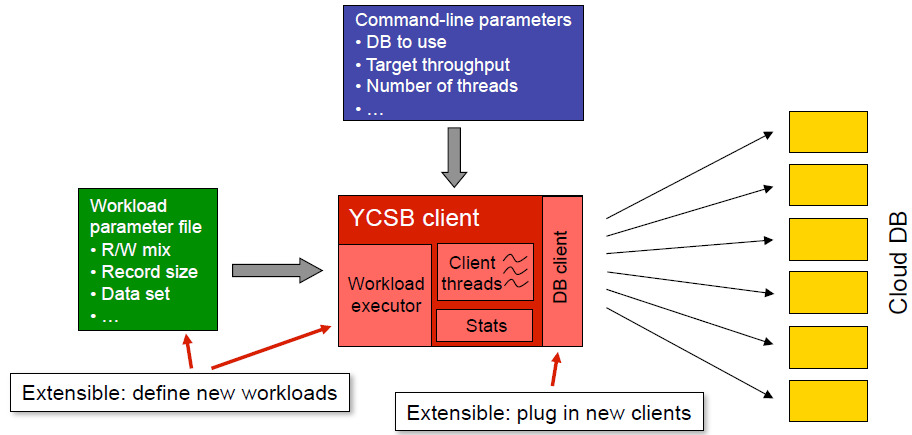


Figure 7. YCSB System Architecture [Coo10a]

Each operation against the data store is randomly chosen to be one of [Coo10b]:

* **Insert**: Insert a new record.
* **Update**: Update a record by replacing the value of one field.
* **Read**: Read a record, either one randomly chosen field or all fields.
* **Scan**: Scan records in order, starting at a randomly chosen record key. The number of records to scan is randomly chosen.

These operations are quite simple, representing the standard “CRUD” operations: Create, Read, Update, Delete, with Read operations to read one record or to scan records. Despite its simplicity, the API maps well to the native APIs of many cloud serving systems. For scan specifically, the distribution of scan lengths is chosen as part of the workload. Thus, the scan() method takes an initial key and the number of records to scan. Of course, a real application may instead specify a scan interval (e.g. from February 1st to February 15th). The parameter for the number of records allows control over the size of these intervals, without having to determine and specify meaningful endpoints for the scan [Coo10b].

All the core package workloads use the same dataset, so it is possible to load the database once and then run all the workloads. However, workloads A and B modify records, and D and E insert records. If database writes are likely to impact the operation of other workloads (e.g., by fragmenting the on-disk representation) it may be necessary to re-load the database. A particular database loading strategy is not prescribed, since different database systems have different loading mechanisms (including some that have no special bulk load facility at all) [Coo10b].

The core YCSB workloads are summarized in **Table 3**. The experiments conducted in [Abr14] used core workloads A, B, C and F (as well as two other custom workloads). For the purposes of stage 1 of this project, workloads A, B and F will be utilized. The justification for this is as follows: firstly, conducting all of the workloads will be a substantially time-consuming process, thus three workloads is considered a reasonable sample-size considering time constraints. Secondly, related to time constraints, and as described above, workloads D and E modify the size of the data store which thus would require flushing and reloading of the data store after those workloads have executed in order to keep the data store consistent across experiments. Thirdly, workload C is very similar to workload B, thus only one of these workloads need be selected. Workload B has been chosen instead of workload C as it is a more stressing workload.

Table 3: Core YCSB Workloads [Coo15], [Coo10b]

|  |  |  |  |
| --- | --- | --- | --- |
| **Workload** | **Operations** | **Record Selection** | **Application Example** |
| A – Update heavy | Read: 50%  Update: 50% | Zipfian Distribution | Session store recording recent actions in a user session |
| B – Read heavy | Read: 95%  Update: 5% | Zipfian Distribution | Photo tagging; add a tag is an update, but most operations are to read tags |
| C – Read only | Read: 100% | Zipfian Distribution | User profile cache, where profiles are constructed elsewhere (e.g., Hadoop) |
| D – Read latest | Read: 95%  Insert: 5% | Latest | User status updates; people want to read the latest updates |
| E – Short ranges | Scan: 95%  Insert: 5% | Zipfian/Uniform Distribution\* | Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id) |
| F – Read-Modify-Write | Read: 50%  Read-Modify-Write: 50% | Zipfian Distribution | User database, where user records are read and modified by the user or to record user activity. |
| \* Worload E uses the Zipfian distribution to choose the first key in the range, and the Uniform distribution to choose the number of records to scan. | | | |

YCSB has the capability to measure workload execution time, throughput and latency. For the purposes of this project, only execution time will be measured and analyzed. Execution time will be measured for the four selected database programs, for three workloads A, B and F, and four tiers of total data store size. [Abr14] used a fixed data store size (600,000 records of 1kB each) and executed 1,000 operations on the data store for each workload. In this project however, the data store size tiers will be utilized as described above, and the number of operations executed on the data stores will be dependent on the data store size. This number of operations is one-tenth of the number of records in the data store, with a minimum of 1,000 operations to be executed per workload. The data store size tiers are displayed in **Table 4**. Each record will be 1kB in size as described earlier. Thirty repetitions of each workload will be executed per database per tier, ensuring that a standard Z-distribution can be used to model the results of the samples for each database. Using this technique, a pairwise comparison of each workload can be used to discover the highest performing NoSQL database.

Table 4: Data store size tiers

|  |  |  |  |
| --- | --- | --- | --- |
| **Tier** | **Identifier** | **No. records** | **No. operations** |
| 1 | 1k | 1,000 | 1,000 |
| 2 | 10k | 10,000 | 1,000 |
| 3 | 100k | 100,000 | 10,000 |
| 4 | 1M | 1,000,000 | 100,000 |

## Methodology

In order to conduct a fair comparison of the database applications, a common dataset is required. As previously stated, the workloads used to analyze each graph database should be random, however creating a random, but consistent workload is challenging. 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. YCSB solves this challenge by utilizing a dataset generation script to create an intermediate, implementation-independent representation of a dataset. This intermediate representation is then consumed by each database, which creates a dataset in the native representation of the database. Each workload is generated via an identical process, utilizing the Zipfian and Uniform distributions as listed in **Table 3**.

As stated previously, the initial aim of this first stage of the project was to compare the top four databases (with lowest execution time) from [Abr14]. However, despite there being official client plugins hosted on the YCSB Github site at [Coo15] for most of the database applications, countless compatibility issues were contended with and ultimately only Redis v2.8.19 and MongoDB v2.4.9 could be made functional with the Ubuntu VM and YCSB.

In order to automate collection of the experiment results, Linux bash scripts were developed and executed on the Ubuntu VM. All results were exported to comma-separated value files and copied to the host machine for analysis in Microsoft Excel 2013. The developed bash scripts, all results and documentation of actions taken can be accessed on the public GitHub repository at:

https://github.com/albanoj2/se655\_final\_project/tree/master/individual/nosql.

## Results

YCSB core workloads A, B and F were executed on the Redis and MongoDB databases. The number of records in each database and the number of operations are summarized in **Table 4** above. Thirty repetitions of each workload were conducted and the mean execution times are presented in **Table 5**.

Table 5: After completion of the prescribed experiments, the results revealed that Redis had a faster execution time than MongoDB, and was also capable of completing more of the prescribed workloads than MongoDB.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Database** | **Workload** | | **1k (ms)** | | | **10k (ms)** | **100k (ms)** | **1M (ms)** |
| Redis | A | | 246.8333 | | | 236.4333 | 878.5333 | 7120.967 |
| B | | 239.7 | | | 232.8 | 851.5333 | 7461.2 |
| F | | 288.9333 | | | 286.3333 | 1209.1 | 10646.63 |
| MongoDB | A | | 559.2667 | | | 570.5333 | 1974.267 | *CNC* |
| B | | 493.4667 | | | 498.9667 | 1523.6 | *CNC* |
| F | | 644.6333 | | | 649.2667 | 2441.633 | *CNC* |
| CNC: Could not complete | |  | |  |

The 32-bit version of MongoDB that was used has a limitation in that it will only allow a maximum storage size of 2GB including data and indexes [Mon15]. It was not anticipated that this would be an issue as the maximum data size utilized in these experiments is 1 GB. Nonetheless when trying to load 1 million records, MongoDB could only load 568,920 records before an exception was raised and execution was halted. As a result the database became corrupted and could not be accessed. The only solution to refresh the database was to physically delete the database archive files in directory */var/lib/mongodb/* and reload with a smaller quantity of records.

The mean execution times for Redis vs MongoDB for all database sizes and all workloads (for which a comparison can be made) are plotted in **Figure 8**. Redis clearly has a shorter execution time by approximately 50% in all cases. This is statistically significant for all workloads and database sizes, as confirmed by the calculations in the **Appendix A: Statistical Calculations: Individual NoSQL Database Analysis** section.

Figure 8. Mean execution times for Redis vs MongoDB for all database sizes and all workloads. Redis clearly has a shorter execution time in all cases.

When considering the results for the 100k size databases, it is interesting to note that although Redis’ execution times were consistently significantly faster, the variance of Redis’ execution times was consistently significantly larger for workloads that modify data in the database (ie. non-read operations). This is due to the underlying architecture of Redis. In contrast to MongoDB, Redis is an in-memory database, meaning that it stores as much of the database records as possible in volatile memory (ie. RAM). The amount of RAM allocated to the SUT was much larger than Redis needed to store the entire database in RAM, so there were no limitations requiring Redis to access the physical drive to perform an operation due to a ‘cache miss’. However, in order to avoid data loss in case of a memory malfunction or power loss, Redis also stores the database in non-volatile memory (ie. on the HDD). In order to maintain fast execution, Redis stores the changes to the HDD at regular intervals, dependent on the number of changes that have occurred. The HDD thus has eventual consistency, which is one of the properties of a BASE system. Redis stores the changes to the HDD at least once every 15 minutes (if there are no changes, no writing to disk is required). If there are at least 10 changes made to the database in a 5 minute window, Redis then stores the changes to the HDD. Similarly, if there are at least 10,000 changes made to the database in a 60 second window, the changes are then stored to the HDD. This functionality is clearly evident in the chart illustrated in **Figure 9**, which plots the execution times for the 30 repetitions of Workload F on a database containing 1 million records. For this workload, an average of 50,000 records were modified during each repetition. Assuming that the time in between each repetition is negligible, approximately 350,000 records were modified every 60 seconds (it could have actually been twice that amount, as discussed below). Redis was thus required to store all of the changes to the HDD every 60 seconds, and it took approximately 60 seconds to complete this process while the database was simultaneously being regularly modified. This resulted in an increased execution time of approximately 50% for the workloads executed during this time. It can be deduced from the chart that Redis does not restart its timer to detect changes until after the process of writing to the HDD is complete. Though it is out of the scope of this study, it would be interesting to know how Redis deals with changes to the in-memory database whilst writing backups to the HDD (ie. does Redis lock the volatile memory space and write changes to new memory locations whilst a HDD write is occurring? This could mean that in this scenario, instead of 350,000 changes occurring, 700,000 changes would occur and need to be written to the HDD in the next cycle, as the next backup would not begin until 2 minutes after the previous backup began).

Figure 9. Redis’ execution times for workload F on a database containing 1 million records. Execution of a workload takes approximately 50% longer while Redis is writing to the HDD.

## Conclusion

The results consistently reveal that there is a statistically significant difference between the performance of the Redis and MongoDB databases. Redis has a significantly shorter execution time for all database sizes and all workloads. Redis not only performed better than MongoDB in all cases, but it also could handle much larger quantities of data in its data store. Redis will thus be utilized in the remainder of this paper as the representative NoSQL database when used to compare the performance of relational, NoSQL, and graph databases.

# Relational Database Analysis

Relational Databases are the most popular type of database available and because of this there are a multitude of RDBMSs available in both the open source and proprietary markets. Relational databases are rooted in most enterprise systems because of their popularity; this leads to the belief that they are the most efficient choice for a database.

Relational Databases focus on relationships between different tables. Each table in the database holds a primary key that can uniquely identify each row in the table. Other tables in the database will use that value as a foreign key to access the information stored. Storing information separately in different tables allows for better storage of data. It reduces the amount of replicated data in the system and in turn makes changing values that would normally affect multiple elements in the database very simple [Tou15]. When this table structure is used correctly relational databases are extremely efficient when it comes to relating data.

## System under Test & Component under Study

The machine that will be used to test the different Relational Databases, the SUT, will have the following specifications:

* **Operating System:** OSX Yosemite 10.10.1 64-bit
* **Processor:** Intel Core i5 @ 1.4 GHz
* **Memory:** 8GB DDR3 1600 MHz
* **Hard Drive:** 250 GB Apple SSD SD0256F

The CUS for the performance analysis will be the databases being tested. After research on which relational databases were popular amongst users it was decided that the databases being tested would be:

* MySQL
* PostgreSql

## Benchmarks & Workloads

One nice thing about relational databases is that almost all of them use SQL to access and manipulate fields in the database. The databases will each be loaded with the same tables that are filled with identical data. This only requires one creation SQL query that can be used on all databases. To get more out of the comparison the initial population of the tables will be tested in four stages:

* **Small:** 1,000 cumulative rows between all tables
* **Medium**: 10,000 cumulative rows between all tables
* **Large**: 100,000 cumulative rows between all tables
* **Extra-Large**: 1,000,000 cumulative rows between all tables

The same queries will be run on the two databases with the four initial configurations. These queries will be repeated 30 times each to get a population sized dataset for each query. This means there will need to be 60 execution times recorded per query.

To get the performance analysis information tables need to exist in the database. The fake data is created using a python script, which is located in the attached github account. This fake database models a factory with orders, products, and clients. For simplicity there will be only four tables used in the performance analysis: ClientTable, OrderTable, ProductTable, and OrderProductTable. The ClientTable will hold information about clients that are ordering from the factory. The OrderTable will relate clients to their orders. The OrderProductTable will relate orders to products. And finally the ProductTable will hold information about products being sold.

Table 6: Format of for the four tables in the relational database: (a) OrderTable, (b) ClientTable, (c) ProductTable, and (d) OrderProductTable.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **OrderTable** | | | Client\_ID | Order\_ID | | varchar(200) | varchar(200) [P] |   (a) | |  |  |  | | --- | --- | --- | | **ClientTable** | | | | Client\_ID | Email | Address | | varchar(200) [P] | varchar(200) | varchar(200) |   (b) |
| |  |  | | --- | --- | | **ProductTable** | | | Description | Product\_ID | | varchar(200) | varchar(200) [P] |   (c) | |  |  |  |  | | --- | --- | --- | --- | | **OrderProductTable** | | | | | Order\_Number | Order\_ID | Count | Product\_D | | varchar(200) [P] | varchar(200) | INT | varchar(200) |   (d) |

The tables in the database are shown in **Table 6**. Each of the tables lists each of the columns in the table, tell the format of that column, and list which column is the primary key. From this graphical look at the database it is easy to see how keys relate from one table to another. Client\_ID is used as the primary key of the ClientTable and can be referenced in the OrderTable to find orders. The Order\_ID of the OrderTable can be used in the OrderProductTable to find all parts of the order. Each of these individual parts of the order is listed with a different Order\_Number. Finally Product\_ID can be related to the ProductTable to get the description of the item.

Each of these tables in the database will have a different number of rows depending on the initial configuration. It was decided that the most data should be relating products to a specific order. This way a customer could have multiple products in each order. Number of Clients and Products would remain the smallest to allow clients to have a wide variety of order and to cause repeat of products ordered. Finally OrderTable was made to be 10 times larger than the ClientTable so that on average each client would have 10 orders. **Table 7** shows the initial population of each table for the four configurations.

Table 7: Initial populations for relational databases.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **OrderTable** | **ClientTable** | **OrderProductTable** | **ProductTable** | **Total** |
| **Small** | 100 | 10 | 880 | 10 | 1,000 |
| **Medium** | 1,000 | 100 | 8,800 | 100 | 10,000 |
| **Large** | 10,000 | 1,000 | 88,000 | 1,000 | 100,000 |
| **Extra-Large** | 100,000 | 10,000 | 880,000 | 10,000 | 1,000,000 |

It was decided that there would be five queries run and compared between the two databases. These queries will test different aspects of the database and are explained in **Table 8**.

Table 8: Explanation of relational queries.

|  |  |
| --- | --- |
| **Query** | **Explanation** |
| SELECT \* FROM OrderTable; | Simple query to return all rows in the OrderTable |
| SELECT OrderProductTable.Order\_Number, OrderProductTable.Product\_ID, OrderProductTable.Count FROM OrderProductTable ORDER BY OrderProductTable.Count; | This query tests the ability of the database to order all the data in a table by a value in the row. |
| SELECT OrderTable.Client\_ID, COUNT(OrderTable.Client\_ID) as count FROM OrderTable GROUP BY OrderTable.Client\_ID ORDER BY count DESC; | This query is similar to the second query in that it is ordering data. However the data being ordered is the number of orders each client has in the OrderTable. This involves grouping the appearances of each Cliend\_ID in the table. |
| UPDATE OrderProductTable SET Count = 5 WHERE Count > 5; | Simple update query that will find any order that wants more than 5 of a single product and reduce that order amount to 5. |
| SELECT ProductTable.Description FROM ProductTable WHERE ProductTable.Product\_ID = ANY (SELECT OrderProductTable.Product\_ID FROM OrderProductTable WHERE OrderProductTable.Order\_ID = ANY (SELECT OrderTable.Order\_ID FROM OrderTable WHERE OrderTable.Client\_ID = ANY (SELECT ClientTable.Client\_ID FROM ClientTable WHERE ClientTable.Email = "Client\_0@blah.com"))); | Query that uses nested selects to use information from all tables. The query will find the Cliend\_ID that matches a specific email. This Cliend\_ID will be used to find all Order\_IDs from that client. The Order\_IDs will be used to find all Product\_IDs associated with products each of the orders. Finally the description of these products will be returned. |

## Collection of Data

The collection of data, much like the creation of queries, will use a python script. This python script will use the python libraries MySQLdb (for MySQL) and psycopg2 (for PostgreSql). These libraries handle the connection to the database with the credentials the user created when the database was installed. Timestamps are taken before each of the queries run and again they have returned. The difference between the two timestamps is found and used as the execution time of the query. Since execution time is taken from both databases in this way the errors shouldn’t have as big of an effect.

Table 9: Rows returned/changed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Small** | **Medium** | **Large** | **Extra-Large** |
| **Query 1** | 100 | 1,000 | 10,000 | 100,000 |
| **Query 2** | 880 | 8,800 | 88,000 | 880,000 |
| **Query 3** | 10 | 100 | 1,000 | 10,000 |
| **Query 4** | 459 | 4,440 | 43,824 | 439,629 |
| **Query 5** | 10 | 62 | 45 | 49 |

The number of returned/changed values for each of the queries is listed in **Table 9**. This shows how the information returned generally increases at the same rate as the overall size of the database. Queries 1, 2, and 3 return all elements of three different tables so they grow with the increased table size. Query 4 grows at about 10 times the return for each query, also like the data in the database. Finally query 5 returns the description of products the client has ordered. This means it is equivalent to the number of different products the client has ordered. The random distribution of products starts to show since the Large and Extra-Large have about the same amount of data points returned, both of which are less than the Medium configuration.

The execution times of each of the queries were collected for each of the 30 runs for the four initial configurations. This process was done for both MySQL and PostgreSql. The execution times are given in **Appendix A: Statistical Calculations: Individual Relational Database Analysis**. The average and standard deviation are collected for each of the queries. These values will then be used to calculate a percent confidence that one database was better per query per configuration.

## Results

Gathering the information from [APPENDIX] the average execution times along with the standard deviations for each of the queries per configuration are given in **Table 10**.

Table 10: Results from 30 runs of each query.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Results (ms)** | | **Query 1** | | **Query 2** | | **Query 3** | | **Query 4** | | **Query 5** | |
| **Average** | **Std\_dev** | **Average** | **Std\_dev** | **Average** | **Std\_dev** | **Average** | **Std\_dev** | **Average** | **Std\_dev** |
| **Small** | **MySQL** | 0.3111 | 0.0578 | 5.324 | 0.484 | 0.417 | 0.0869 | 1.8029 | 0.496 | 2.437 | 0.4869 |
| **PostgreSql** | 1.7188 | 0.246 | 4.3544 | 2.962 | 2.057 | 0.648 | 6.8869 | 0.6693 | 3.956 | 0.418 |
| **Medium** | **MySQL** | 1.1638 | 0.143 | 43.385 | 3.741 | 1.687 | 4.827 | 14.884 | 4.8275 | 25.785 | 6.103 |
| **PostgreSql** | 4.2004 | 1.050 | 23.3558 | 2.260 | 7.889 | 1.588 | 54.049 | 4.636 | 10.956 | 4.802 |
| **Large** | **MySQL** | 9.5801 | 0.96 | 577.222 | 37.485 | 13.331 | 4.95 | 140.835 | 15.914 | 245.69.2 | 28.411 |
| **PostgreSql** | 34.676 | 18.66 | 332.849 | 21.491 | 32.316 | 15.481 | 702.753 | 108.56 | 75.779 | 16.489 |
| **Extra Large** | **MySQL** | 128.172 | 11.333 | 6919.29 | 188.94 | 129.734 | 16.63 | 1921.8 | 197.812 | 2855.03 | 586.81 |
| **PostgreSql** | 109.229 | 16.19 | 3070.22 | 220.214 | 103.039 | 16.6.07 | 11682.4 | 463.33 | 478.169 | 50.6203 |

There are two varying factors in the analysis, initial configuration size and the queries. The first analysis done uses the static initial configuration while the second analysis focuses on a certain query.

Analyzing the results by initial configuration gives more information to a user that knows the target size of their database. Since each of the five queries being run test a different aspect of the database it is possible to focus on the important queries and ignore the results of queries that wouldn’t be run. Kiviat diagrams for each of the initial configurations are given in Appendix A: Statistical Calculations [APPENDIX]/Kiviat Diagrams. These diagrams graphically present the information given in Table 10. It can be gathered from either the table or the Kiviat diagrams that MySQL is better in a higher majority of the queries. For the smaller initial configuration size it can be seen that 4 of the 5 queries are performed faster on MySQL. The medium and large initial configuration both have MySQL execute faster on 3 of the 5 queries. Finally the extra-large initial configuration has MySQL only executing 1 of the 5 queries faster. Each of these differences of means was tested to 90% confidence. The data as well as work to show the confidence test is given in Appendix A: Statistical Calculations/[APPENDIX] /Confidence Intervals. This information shows that MySQL is the more efficient database until the size of the database reached 1,000,000 rows.

Another way to view this data is to compare the difference in execution time as the size of the database grows. Each query is graphed for the initial configuration sizes. It was expected that the execution time would go up as the data in the database increased. The rate at which the execution times increased was not the same for the two databases. PostgreSql was slower in its queries in the beginning but didn’t have its execution time increase with size as fast as MySQL did for most queries. By the time one million elements are put into the database PostgreSql is able to pass MySQL in performance for most of the queries. This information is shown graphically in Appendix A: Statistical Calculations/[APPENDIX].

Table 11: Regression Analysis

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Regression Analysis | | | | |
| Query | Database | Regression Model | Correlation Coefficient  (r) | Correlation of Determination  (r2) |
| 1 | MySQL | Y = 1.0132 + 0.0001 \* X | 0.99968 | 0.99935 |
| PostgreSql | Y = 9.2953 + 0.0001 \* X | 0.97783 | 0.95615 |
| 2 | MySQL | Y = -46.5109 + 0.007 \* X | 0.99989 | 0.99977 |
| PostgreSql | Y = 6.1209 + 0.0031 \* X | 0.99995 | 0.99991 |
| 3 | MySQL | Y = 0.3568 + 0.0001 \* X | 1 | 1 |
| PostgreSql | Y = 10.1968 + 0.0001 \* X | 0.9811 | 0.96256 |
| 4 | MySQL | Y = -18.0186 + 0.0019 \* X | 0.99967 | 0.99935 |
| PostgreSql | Y = -173.2058 + 0.0118 \* X | 0.99931 | 0.99862 |
| 5 | MySQL | Y = -13.8889 + 0.0029 \* X | 0.99991 | 0.99982 |
| PostgreSql | Y = 12.4078 + 0.0005 \* X | 0.99871 | 0.99742 |

The correlation of determination, given in Table 11, shows that the regression models explain 95% of the variation in the time required to execute the queries. With this level of certainty in the regression model the two formulas are compared to see at what point the one database becomes more efficient than the other. The correlation of coefficient shows positive correlation between size of the database and execution time of the queries. This information is given in more detail in Appendix A: Statistical Calculations/

/Regression Analysis.

Table 12: Intersection of Regression Models

|  |  |  |  |
| --- | --- | --- | --- |
| Query | Database | Regression Model | Intersection |
| 1 | MySQL | Y = 1.0132 + 0.0001 \* X | No Intersection |
| PostgreSql | Y = 9.2953 + 0.0001 \* X |
| 2 | MySQL | Y = -46.5109 + 0.007 \* X | 13,496 |
| PostgreSql | Y = 6.1209 + 0.0031 \* X |
| 3 | MySQL | Y = 0.3568 + 0.0001 \* X | No Intersection |
| PostgreSql | Y = 10.1968 + 0.0001 \* X |
| 4 | MySQL | Y = -18.0186 + 0.0019 \* X | 15,676 |
| PostgreSql | Y = -173.2058 + 0.0118 \* X |
| 5 | MySQL | Y = -13.8889 + 0.0029 \* X | 10,957 |
| PostgreSql | Y = 12.4078 + 0.0005 \* X |

The correlation regression model, shown in Table 12, doesn’t hold well with the actual data collected. This has to do with only four data points being supplied to the formula. The initial assumption was that it may have been an exponential growth instead of a linear growth but upon looking at the data it is pretty obvious at the increase in execution time is no exponential. With more data the regression model would be more accurate. The raw data shows that query 1 does have an intersection before 1,000,000 rows where PostgreSql becomes more efficient. The raw data for query 2 shows that there is no intersection and PostgreSQL always executes the query faster. The raw data for query 3 shows that there is intersection of before 1,000,000 rows where PostgreSQL becomes more efficient. The raw data for query 4 shows that MySQL is more efficient throughout not just past 15,676 rows. Finally the raw data for query 5 is close to the regression model. It shows that right before 10,000 rows PostgreSql executes the query faster than MySQL. The data being spread out between 100,000 rows and 1,000,000 rows really left a lot to be guessed by the regression analysis tool. For a more realistic model there should have been more measurements taken at intermediate sizes.

After all the analysis was done on the data collected it is safe to say that MySQL was the more efficient database. The assumption being made is that the database will not have millions of rows because as the analysis of individual queries showed that PostgreSql seems to handle the larger dataset (1,000,000+ rows) better. Since the comparison of the best databases from each group is going to be off a dataset of only a couple thousand it makes sense to not focus on the performance loss as the dataset grows.

## Concerns

For the small and medium initial population caching will be a major factor. If the queries are run in a loop that executes the exact same query 30 times in a row then the execution time that had no caching (the true execution time) will look like an outlier in the dataset. The caching of data will make it seem like the execution time is lower than it really is. The way around this was to clear and repopulate the database before each query is run. This added significant time (2.5 hours to run the five queries on extra-large PostgreSql) to the process of collecting data. It wad one for all four configurations even though it probably wasn’t necessary for the large and extra large configurations since there is not way hundreds of thousands of rows could have been cached. The idea was to keep consistency between the configurations.

There should have been a query that ran the delete functionality against the two databases. The problem was that PostgreSql wouldn’t allow for data to be deleted from multiple tables in the same way MySQL would. After researching the issue it was decided that it would be better to leave the query out then try to translate the query into the PostgreSql version and try to claim they still do the exact same operation. It is left out in case the two ways of deleting are not equivalent and thus unfair to compare.

# Group Comparison of Databases

With the best performing databases selected from each category, the next step in the analysis of each database is to compare each of the highest performing databases against one another. This comparison is accomplished by selecting three distinct queries to execute on each database: (1) a query that is NoSQL in nature, (2) a query that is graph-centric in nature, and (3) a query that is relational in nature.

This mixture of queries provides a fair comparison, where each category of database will have the advantage of executing a query designed to excel in its own category, but will also be subjected to the disadvantage of executing a query from each of the other database types. This cross-category comparison is intended to expose not only how well each of the best performing databases execute queries designed for that database, but more importantly, see how well each database can execute queries that are designed for another category of databases. Using these results, the overall best-performing database can be selected.

## System under Test & Component under Study

Based on the results of the NoSQL, graph, and SQL database results, the following databases were selected for the comparison conducted in this section:

* **NoSQL**: Redis 2.8.19 (64-bit)[[8]](#footnote-9)
* **Graph**: Neo4j 2.2.0
* **SQL**: MySQL 14.14 (distribution 5.6.24) for Win64 (x86\_64)

These selected databases make up the CUS for the experiments to following. In order to provide single, consistent baseline for executing these experiments, the machine used to conduct the graph database analysis in the

**Structure of** Document

The analysis contained in this document is divided into four categories, based on the database or database under study: Section 2 contains the performance analysis and selection of the top-performing graph database; Section 3 contains the performance study of the NoSQL databases selected for this paper, as well as the methodology used to select the top-performing NoSQL database; Section 4 contains the analysis and selection of the top-performing SQL databases; Finally, section 5 contains the comparative analysis and selection of the top-perform database. Section 5, unlike the previous three sections, results a single, “best” performing database based on the aggregate analysis and results conducted in this paper. Apart from the analysis of individual database categories and the combined database group, section 6 provides a brief overview of alternative, non-measurement methods of analyzing and comparing the performance of databases, including analytical and simulation techniques. Lastly, section 7 provides conclusive comments by the authors on the results recorded in this document, including notes about the conclusions that can be drawn from the selection of the "best” database.

This document also contains a **References** and an **Acronyms & Abbreviations** section that provide the sources used throughout this paper and a comprehensive enumeration of the acronyms and abbreviations used in this document, respectively. **Appendix A: Statistical Calculations** contains the detailed calculations and computations performed in order to analyze each individual database, as well as compare the best databases from each database category in section 5. All lengthy calculations are contained in this appendix, and therefore, this appendix should be consulted for all analytical information pertaining to the conclusions and judgements stated in this paper.

Graph Database Analysis section of this paper. Namely,

* **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

Supplemental to the hardware selected for this analysis, the following software (and associated versions) are used to the conduct the experiments described in this section:

* **Java:** JDK 1.8.0 x64 (build 1.8.0\_40-b26)
* **Python:** Python 2.7.9
* **Eclipse**: Eclipse EE for Java Developers Luna Service Release 2 (4.4.2, build 20150219-0600)

## Benchmarks & Workloads

In order to provide an unbiased experimentation of each database, the workload used to analyze the best-performing databases of each category is composed of a query from the domain of each database category. Therefore, a NoSQL-focuses, graph-focused, and SQL-focused are executed by each type of database. In the interest of simplicity, a single data set is used for all three selected experiments. This data set is a recommendation data, where people work at a given number of places of work and have a given number of interests. The concept behind this recommendation data set is that, based on the places that each person has worked, an algorithm can be devised to recommend interests, based on the interests of coworkers (other people who have worked at the same place of work as the subject for whom the recommendations are being made).

Using this data set, the following queries were created:

1. **Get all interests for a person**: this NoSQL focused query exercises the ability of each database to retrieve small, independent snippets of data. In particular, this query is designed to provide the greatest advantage to a key-value store style of NoSQL database (which is the type of the NoSQL database chosen in the previous experiments).
2. **Recommend interests based on the interests of coworkers**: this query is focused on relationship-traversals, and is therefore advantageous for graph databases. This query is a common query executed by many large-scale, enterprise-grade graph databases and therefore, reflects the ability of each database to quickly traverse the relationships contained in the database to obtain pertinent information.
3. **Obtain all coworkers and common place of work for a subject**: given a person, this query requires that each database obtain a list of all coworkers for the subject, as well as the common workplace shared by the subject and the coworkers. This composite-retrieval algorithm is designed to provide the greatest advantage to SQL databases, which are designed with the ability to aggregate data.

Given these algorithms, 30 replications are taken for each algorithm for each database, ensuring that a standard Z-distribution can be achieved. Due to the graph-size limitations discovered during the individual graph database analysis, the data set size was restricted to a relatively small data set. The selected data set is as follows:

* people
* interests to choose from
* possible places of work

Given these data sizes, the following restrictions were placed on the relationships between the data:

* Each person has exactly interests
* Each person has exactly places of work (has worked at exactly 5 places of work)

Therefore, the data set used contains data entities (a combination of people, interests, and places of work) and relationships ( interests and places of work for each of the people). Since this data set is essentially very small in scale, the queries enumerated above are executed for each of the people in the data set. For example, the first query, *get all interests for a person*, becomes *get all interests for all people.* The advantage of this iterative technique is two-fold:

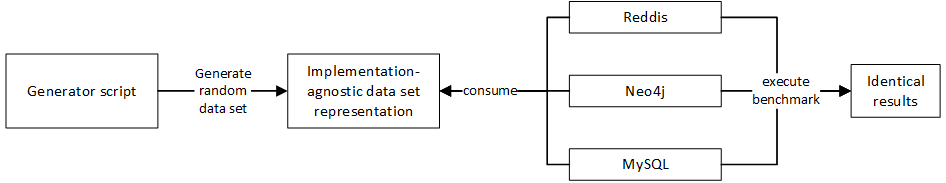
1. Since each database can cache small amounts of data, if each algorithm were executed for only a single person, the cache would easily contain all of the data associated with the single person. Therefore, on the replications that follow the first experiment, the data associated with the subject will likely be cached, and therefore, the execution time for each algorithm will be nearly instantaneous. If, instead, each algorithm is repeated for each person, the cache for each database will become warmed with the data of different subjects, and therefore, will more accurately mimic the real-world cache conditions of the database (in real-world applications, caches are filled with the data of many different subjects, rather than the same, repeated subject). This will also reduce the cache hit ratio, resulting in more stable results as each of the 30 replications are execute for all of the people in the data set (a single replication is defined as the execution of a single algorithm over all of the people in the data set).
2. Since each database can quickly execute each algorithm on a single subject, compounding each algorithm over all people in the data set results in a more comprehensive view of the execution time of the algorithm (since the inconsistencies of caching and other variable factors are taken into account as the algorithm is executed from various subjects, or starting points). This technique is similar to the measuring of the thickness of a thin object by measuring the combined thickness of objects and dividing by .

Throughout each of these algorithms, the execution time of each database when performing the above queries will be the primary focus of each experiment. Therefore, a fixed-computation, variable-time benchmark will be used to determine the performance of each database.

## Methodology

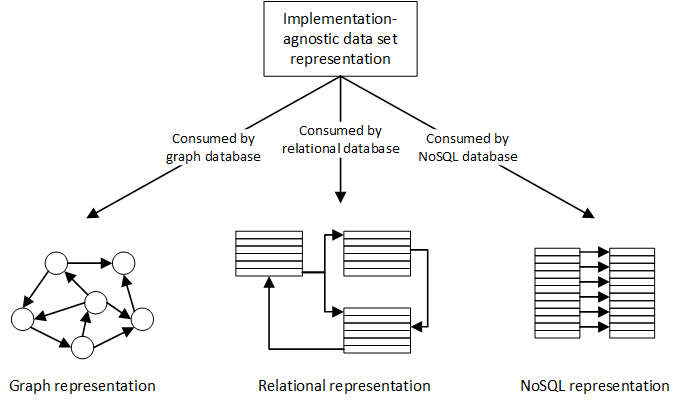
Due to the comparison of disparate databases (each database represents data in a fashion that is largely different than the other two databases against which it is being compared), a common, implementation-agnostic representation of the data set to be executed is needed. This problem closely matches the issue resolved in the Methodology section of the graph database analysis conducted in this paper. Therefore, the same solution is used in this triple-category comparison: A generator is used to create the implementation-agnostic data set representation; then each database consumes this common representation.

Upon consuming this common data set, each database then contains the same data (same data and same relationships and connections between data) as the other databases. This approach provides for an unbiased comparison among the databases, as well as a way to verify that execution of the benchmark by each database results in the same output data. This ensures that the same work was accomplished by each database, ensuring that each is compared on equal footing. This generation and consumption technique is illustrated below in **Figure 10**.



**Figure 10.** Using an implementation-agonistic representation, each database can consume a common data set, ensuring each query is executed by each database against the same data.

The concept of a common data set for each database (despite the various data representation styles intrinsic to each) is illustrated below in **Figure 11**.



**Figure 11.** Although a common, implementation-independent data set is used by each database, the data stored in each database will be represented using the constructs inherent to each database type.

## Results

Using the benchmark prescribed above, each database executed each query on the machine previously described, which resulted in the mean execution times illustrated in **Figure 12** (note that the scale for this figure is exponential, rather than linear).

**Figure 12.** In general, the graph database outperformed the relational and NoSQL databases, while the NoSQL database outperformed the selected relational database.

Based on the calculations performed in **Appendix A: Statistical Calculations: Group Database Analysis**, it can be seen that the difference in all mean values within each query are statistically significant (or, rather, there is no evidence to the contrary) for a confidence level, except for the different in mean execution times for the NoSQL and SQL databases executing the graph query (which is significant to approximately confidence). It is important to note that no calculations were performed to test the statistical significance between the mean execution times for each of the queries executed by a single database, since the focus of this paper is to compare the selected databases, rather than the execution of various queries by the same database. Therefore, it is inappropriate to draw a conclusion about the relative performance of one query over another for a single database. For example, it is inappropriate to claim that the graph database has a statistically-significantly lower mean execution time for the NoSQL query than the graph query, since no calculations have been performed to justify this statement.

Based on the result depicted in **Figure 12**, it is clear that the graph database outperformed the NoSQL and SQL databases in all executed queries. As a corollary to this finding, it is also clear that the NoSQL database out-performed the SQL database. While it was expected that each database would outperform the other databases in the query designed for that database, this was not the case. Instead, it appears that there is a more general result that can be concluded from these results: In general, the graph database outperforms both the NoSQL and SQL databases, while the NoSQL database ranks higher than the SQL database in terms of mean execution times.

The advantage of each database over the other has a great deal to do due with the nature in which database store and interpret data. In the case of the graph query, the ability of each database to traverse relationships is taxed, and therefore, the ability of each database to join together information is put to the test. In the case of the graph database, this is the natural manner in which data is stored and accessed, but this is not the case for the NoSQL and SQL databases. In the case of NoSQL, data is stored in singular segments, but although this is less efficient than the graphical manner of storing and accessing data, it is still a much more efficient manner than SQL. With NoSQL, the references to entities in the graph can be stored in a list and traversed to obtain the entities related to the source entity. Conversely, with SQL, a composite query must be created, which requires that data be joined together from disparate tables, resulting in a lengthy look-up process. Therefore, NoSQL essentially accesses the data in a singular but sequential manner, while SQL must access the data in a composite but still sequential manner.

In the case of the NoSQL query, each database is challenged to read properties from each entity. With NoSQL, this is the natural access paradigm and therefore, it was able to excel at this algorithm. In the case of the graph database, the property graph structure of Neo4j enabled the look-up of properties result in simple key-value lookups [Alb15]. With SQL, again, this lookup amounted to the traversal of each entity in a table, and then a subsequent reading of properties from a column in the table. Although this is a simple task, it appears that there is a great deal of overhead in reading such properties, which resulted in a mean execution time of over (compared to mean execution times of about and for the NoSQL and graph databases, respectively).

For the last algorithm, the aggregation of all coworkers and common places of work is essentially a union of data from multiple tables. Due to the relationship-based access pattern of the graph database, this algorithm was easily completed by Neo4j. In the case of the NoSQL database, this algorithm is relatively taxing, since it requires that multiple lists be traversed, and that key-value pairs be retrieved for each element in these lists. Therefore, its mean execution time was noticeably greater than that of the graph database, but surprisingly, equally-noticeably less than the mean execution time of the SQL database. While this algorithm is designed as a common use case of an SQL database, MySQL struggled to complete this query, which resulted in an execution time of nearly (compared to the mean execution times of and for Redis and Neo4j, respectively).

While it is important to know which database outperforms the others, it is equally as important (if not more important) to know by what degree one database outperforms another. In order to obtain this degree, a relative change calculation can be used to compute a percent increase in perform of one database over another. These change calculations are illustrated below for all databases and all queries (except for the comparison of NoSQL and SQL for the graph query, which is statistically insignificant for the level of confidence used in the calculations depicted in **Appendix A: Statistical Calculations: Group Database Analysis**). The notation, , is used to denote the relative speedup of database compared to database for query , which is used to denote the mean execution time of database executing query .

Using these computations, it can be seen that the graph database executed the NoSQL query faster than the NoSQL database and faster than the SQL database, and the NoSQL database performed the NoSQL query faster than the SQL database. The change values for the graph query are computed below:

The remaining SQL query relative change calculated are depicted below:

Using these relative change values, it not only be seen that one database outperformed another (in a statistically significant sense), but also by what degree.

# Alternative Methods

Although measurement is the more accurate and most realistic means of ascertaining the performance characteristics of a database, there exist multiple non-measurement-based methods for judging the performance of databases. In particular, [Mac13] provides an analytical approach for judging the performance of a graph database. The approach devised in [Mac13] is to divide the operations of a graph database into three distinct groups: (1) micro-operations, (2) macro-operations, and (3) graph algorithms, with micro-operations being the most fundamental operations that can be performed by a graph database, macro-operations being a composite of micro-operations, and graph algorithms being a high-level composite of macro-operations. Using this layering technique, the performance (in terms of order, as in the big-O notation order) can be computed by assigning each micro-operation an analytical equations. For example, traversing all incoming, outgoing, or either direction edges (a micro-operation in the vernacular of [Mac13]) can be modeled as

where is the time required to read either a vertex, edge, or property, respectively, is a vertex from the set of all vertices, is an edge from the set of all edges, and is a property from the set of all properties. Using this foundational equation, the macro-operation *get all neighbors via incoming, outgoing, both edges* can be modeled in terms of this equation; specifically,

where is the number of neighbors of the vertex of interest. Thus, it can be seen that a macro-operation (a very common one for that matter) can be analytically modeled in terms of the execution times for the reading of a vertex and the reading of an edge. Once these foundational value are estimated (or measured), an estimation of the time required to read all neighboring vertices of a vertex can be computed.

Simulations can also be used to model the performance of database, especially databases that are transactional in nature. Many SQL databases, as well as some graph databases (Neo4j in particular) use ACID transactions to ensure the consistency of the data set managed by the database, and therefore, transactional analysis tools can be used to model the time and execution characteristics of these databases. In particular, [Men08] provides a simulation framework for comparing various transactional databases. There are many benefits to using a simulator prior to testing a database in the eventual deployment environment:

* Simulations allow an analyst to systematically judge whether a database will be appropriate for the deployment environment before committing the resource necessary to deploy the database. This ensures that preliminary estimations of performance can be used to test how well a database performs the computations that will be seen in production (e.g. how well the database copes in a read-intensive environment or a write-intensive setting) before placing the database in the production environment.
* Simulations allow trade-off analysis to be conducted. The use of simulations not only allow a transactional database to be evaluated in terms of its generalized performance in the production environment, but also provide an analyst with a mechanism by which one database can be compared to another, providing insight into the trade-offs between databases and the alternatives that are possible for the production environment.

Although the above examples are brief in nature, there are many non-measurement, alternative techniques that can be used to model and gauge the performance characteristics of databases without deploying a database into the production environment on which it will eventually reside.

# Conclusion

Databases have become an essential part of nearly every large-scale software application, and provide a mechanism through which data can be persistently stored, as well as accessed in a rapid manner. As the need for such databases grow, the number and type of these systems have concurrently grow, providing developers with a nearly overwhelming set of databases to choose from. With this growth, the importance of database performance analysis has increased in lock-step. As the array of possible databases grows, it becomes increasingly important that the “best” performing databases are found from this group. The purpose of this paper is to the provide the reader with both a small-scale collection of the “best” performing databases of the three major database categories (relational, NoSQL, and graph), as well as provide the reader with a systematic approach to analyzing the performance of these databases. In pursuance of this goal, the three categories of databases were analyzed in isolation, and then the best performers from each database category were selected and compared head-to-head.

In order to select the best performing graph database, common graph queries were executed by Neo4j and OrientDB, and while OrientDB was capable of completing larger workloads than Neo4j, Neo4j was vastly superior in terms of its mean execution times for the workloads that both database could complete. Therefore, Neo4j was chosen that the “best” performing graph database. In the case of NoSQL, the YCSB tool was used to analyze the performance of two popular NoSQL databases: MongoDB and Redis. At the conclusion of this analysis, it was found that Redis outperformed MongoDB in the majority of benchmarks completed, and therefore, advanced to the group comparison. In the case of the SQL comparison, a similar approach to that of the graph database was taken: Two common SQL databases, MySQL and PostgreSql, executed queries common to relational database use cases with varying data set sizes. At the end of this analysis, it was found that MySQL outperformed PostgreSQL and therefore advanced, along with Neo4j and Redis, to the group comparison.

In the group comparison, a single query was selected from each database domain that reflected the common use cases for each domain. For example, a recommendation algorithm, consisting of relationship traversals and extended inference of information, was selected for the graph database domain. A single data set, over which each of the algorithms was to be executed, was also selected, and each database executed these algorithm. While it was expected by the authors that each database would excel at the queries designed from its domain, empirically, this was not the case. Instead, a more generalized result was found: The graph database outperformed the NoSQL and SQL databases for all queries executed, while the NoSQL database outperformed the SQL database for all queries executed. Although this generalized answer to the question of, “Which database type is best?” is supported statistically, there are some caveats that must be bore in mind.

The queries executed for the group comparison in this paper are all read-centric queries. In essence, this means that the read capability of each database was tested, and therefore, the “best” database found in this paper only reflects a partial superlative: The “best” database found is only the “best” database for the types of queries executed. It was found that during execution, many of the write- or update-intensive algorithms required a much larger sum of memory than the default provided by both Java and Python; compounding this, many of the write-intensive algorithms required a much longer execution time than was practical for the scope of this project. Therefore, many of the common read-intensive algorithms for each database were selected for the sake of pragmatism and practicality. Had the scope of this project been larger and a more detailed study were feasible in the allotted timeframe, a larger and more comprehensive array of queries would have been selected. Therefore, it is imperative that the reader not surmise that the “best” database selected in this paper is actual the best in all cases, but rather, the best, given the nature of the experiments designed and conducted in this paper.

Given these qualifications, this paper, using statistical analysis and empirical experimentation, has found that in general, graph databases outperform both NoSQL and relational databases for all algorithms executed in this paper, and likewise, NoSQL outperforms SQL in all queries contained in this paper. This results in a simple conclusion: Noe4j is the “best” database in the context of this paper, Redis is “second best,” and MySQL is “last.”

References

|  |  |
| --- | --- |
| [Abr14] | V. Abramova, J. Bernardino and P. Furtado. “Experimental Evaluation of NoSQL Databases.” *International Journal of Database Management Systems 6(3)*, pp. 1-16. 2014. |
| [Alb15] | Albano, Justin. “A Comparison of Open Source Graph Databases.” *Unpublished.* 22 Mar. 2015. Print. 29 Mar. 2015. |
| [Coo10a] | B.F. Cooper. “Yahoo! Cloud Serving Benchmark”. Yahoo! Inc. 2010. Web.  <http://labs.yahoo.com/files/ycsb-v4.pdf> |
| [Coo10b] | B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. “Benchmarking cloud serving systems with YCSB”. In Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10). 2010. ACM, New York, NY, USA, 143-154. |
| [Coo15] | "Brianfrankcooper/YCSB." *GitHub*. N.d. Web. 27 Mar. 2015. <https://github.com/brianfrankcooper/YCSB> |
| [Jon15] | S. Jones. “Performance Comparison of NoSQL Databases: A Survey.” *Unpublished*. 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>. |
| [Lil04] | D.J. Lilja. “Measuring Computer Performance, A Practitioner’s Guide,” *Cambridge University Press*. 2004. ISBN 0-521-64105-5. |
| [Mac13] | Macko, Peter, Daniel Margo, and Margo Seltzer. "Performance Introspection of Graph Databases." *SYSTOR, Haifa, Israel (June 30 - July 02, 2013)* (2013): n. pag. *Havard School of Engineering & Applied Science*. Association for Computing Machinery, 13 June 2013. Web. 1 Feb. 2015. <http://www.eecs.harvard.edu/~pmacko/papers/pig-systor13.pdf>. ISBN: 978-1-4503-2116-7. |
| [MaC14] | McColl, Robert C., David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. "A Performance Evaluation of Open Source Graph Databases." *Proceedings of the First Workshop on Parallel Programming for Analytics Applications* (2014): 11-18. *ACM Digital Library*. Web. 21 Jan. 2015. <http://www.stingergraph.com/data/uploads/papers/ppaa2014.pdf>. ISBN: 978-1-4503-2654-4. |
| [Men08] | Mentis, Anakreon, Panagiotis Katsaros, and Lefteris Angelis. "ACID Sim Tools: A Simulation Framework for Distributed Transaction Processing Architectures." *SIMUTOOLS* (2008): n. pag. *Aristotle University Department of Informatics.* AMC, 4-6 Mar. 2008. Web. 17 Apr. 2015. <http://delab.csd.auth.gr/papers/STTCNS2008mka.pdf>. |
| [Mon15] | “FAQ: MongoDB Fundamentals – MongoDB Manual 3.0.2.” *MongoDB,* Inc, 2015. Web. 17 Apr. 2015. <http://docs.mongodb.org/manual/faq/fundamentals/#faq-32-bit-limitations>. |
| [Rob13] | Robinson, Ian, Jim Webber, and Emil Eifrem. "Graph Databases." *O'Reilly Media*. O'Reilly Media, July 2013. Web. 01 Feb. 2015. |
| [Tou15] | D. Tournour. “A Performance Comparison of Open Source and Proprietary Relational Database Management Systems.” *Unpublished*. 2015. |
| [Zai14] | Zaiontz, Charles. "Contrasts for Two Factor ANOVA." *Real Statistics Using Excel*. N.p., 2014. Web. 16 Apr. 2015. <http://www.real-statistics.com/two-way-anova/contrasts-two-factor-anova/>. |

# Acronyms & Abbreviations

|  |  |
| --- | --- |
| **Entry** | **Expanded Phrase** |
| ACID | Atomic, Consistent, Independent, and Durable |
| ANOVA | Analysis of Variance |
| BASE | Basic Availability, Soft-state, and Eventual consistency |
| CRUD | Create, Read, Update, Delete |
| CSV | Comma Separated Value |
| CUS | Component under Study |
| JVM | Java Virtual Machine |
| RDBMS | Relational Database Management Systems |
| RDMS | Relational Database Management System |
| SNA | Social Network Application |
| SQL | Structured Query Language |
| SUT | System under Test |
| YCSB | Yahoo! Cloud Serving Benchmark |

# Appendix A: Statistical Calculations

## Individual Graph Database Analysis

Upon completion of the experiments prescribed for each database, thirty samples were recorded for each workload, algorithm, and database. The recorded samples for Neo4j are presented in **Table 13** below; the recorded samples for OrientDB are likewise presented below in **Table 14**.

Table 13: Although only the small and medium workloads could be completed, Neo4j excelled at capturing data in cache, greatly reducing the time consumed by each experiment, and thus reducing the overall sample mean for each workload.

|  |  |
| --- | --- |
| **Experiment** | **Samples Execution Times (ms)** |
| Friend-of-friends (small) | 93, 71, 19, 12, 47, 18, 21, 15, 31, 32, 15, 42, 1, 16, 16, 0, 15, 16, 21, 10, 0, 15, 0, 16, 15, 0, 16, 16, 0, 15 |
| Friend-of-friends (medium) | 981, 78, 117, 92, 106, 78, 105, 95, 109, 93, 102, 86, 97, 78, 105, 86, 89, 78, 86, 93, 86, 81, 94, 86, 80, 83, 94, 90, 94, 85 |
| Get property (small) | 16, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 15, 0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0 |
| Get property (medium) | 7, 0, 0, 16, 0, 0, 15, 0, 0, 16, 0, 0, 0, 15, 0, 0, 0, 16, 0, 0, 16, 0, 0, 0, 15, 0, 0, 0, 16, 0 |

Table 14: While OrientDB executed each of the prescribed workloads much slower than Neo4j, OrientDB was able to complete both algorithms using the large graph size, and is thus able to scale with a smaller footprint (when compared to Neo4j).

|  |  |
| --- | --- |
| **Experiment** | **Samples Execution Times (ms)** |
| Friend-of-friends (small) | 1326, 612, 482, 480, 517, 442, 483, 451, 440, 459, 451, 455, 442, 452, 485, 447, 481, 444, 428, 440, 399, 436, 418, 451, 409, 384, 405, 397, 414, 416 |
| Friend-of-friends (medium) | 1457, 837, 809, 832, 822, 833, 810, 812, 822, 829, 792, 815, 822, 813, 810, 824, 813, 819, 815, 840, 836, 796, 828, 818, 817, 810, 816, 822, 807, 810 |
| Friend-of-friends (large) | 38395, 34673, 33568, 33798, 29736, 28835, 34797, 36511, 35954, 38227, 35232, 34383, 35626, 38557, 34555, 36619, 33873, 32131, 36841, 37873, 28946, 29849, 27919, 26930, 28199, 28159, 26662, 27940, 28229, 26769 |
| Get property (small) | 544, 325, 304, 381, 286, 293, 333, 244, 364, 194, 374, 189, 396, 163, 368, 192, 350, 215, 304, 260, 251, 321, 157, 394, 174, 324, 227, 275, 299, 147 |
| Get property (medium) | 558, 363, 414, 267, 391, 351, 316, 408, 234, 431, 302, 354, 335, 290, 412, 247, 437, 222, 435, 202, 442, 205, 442, 193, 459, 198, 410, 228, 403, 244 |
| Get property (large) | 6346, 5333, 4479, 6406, 6274, 4119, 3030, 5495, 4876, 3556, 4935, 4771, 4795, 3477, 5002, 4748, 3609, 5810, 4769, 3445, 4821, 4804, 3520, 5780, 5488, 3633, 5530, 4434, 3179, 4415 |

With two databases, two algorithms, and four graph sizes, pairwise comparison is used to discover if the difference in execution times between the two databases is statistically significant. This calculation is computed in accordance with the technique prescribed in [Lil04]. Given the difference of means, , and the standard deviation, , of the pairwise comparison, the confidence interval for the difference between the samples taken for Neo4j and OrientDB can be expressed as

With an alpha level of ( confidence level), the resulting equation is

where for all experiments conducted for the graph databases and for . The resulting difference of means, standard deviations, and confidence intervals are presented below in **Table 15**.

Table 15: Since the confidence interval for each category of experiment conducted does not contain the value , the results of comparison between Neo4j and OrientDB can be considered statistically significant.

|  |  |  |  |
| --- | --- | --- | --- |
| **Experiment** | **Mean ()** | **Standard Deviation ()** | **Confidence Interval** |
| Friend-of-friends (small) |  |  |  |
| Friend-of-friends (medium) |  |  |  |
| Get property (small) |  |  |  |
| Get property (medium) |  |  |  |

Being that all resulting confidence intervals do not contain the value , there does not exist sufficient evidence to prove that there is not a statistically significant difference between the results of the Neo4j and OrientDB samples. Therefore, the differences between the results obtained through experimentation of Neo4j and OrientDB can be considered statistically significant.

## Individual NoSQL Database Analysis

The raw data extracted from the individual experiments conducted on NoSQL databases and the associated means and standard deviations for each workload are displayed in **Table 16** to **Table 19** below.

Table 16: Execution time measurements for a database size of 1k records for each database and each workload.



Table 17: Execution time measurements for a database size of 10k records for each database and each workload.



Table 18: Execution time measurements for a database size of 100k records for each database and each workload.



Table 19: Execution time measurements for a database size of 1M records for each database and each workload.



The factors for these experiments were ultimately two databases, three workloads, and three database sizes. The difference of means method for comparing alternatives as presented in [Lil04] is used to discover if the difference in execution times between the two databases is statistically significant. Given the difference of means, then the standard deviation of this difference of means is the sum of the standard deviations of each set of measurements, appropriately weighted by the total number of measurements in this set (both 30 in this case). The standard deviation of the difference of means is thus [Abr14]:

Where is the standard deviation of the workload repetitions for the first database alternative (ie. Redis), and is the standard deviation of the workload repetitions for the second database alternative (ie. MongoDB). The number of repetitions conducted for each workload on each database is .

The confidence interval can be expressed as [Abr14]:

For a 95% confidence level, , and . The resulting differences of means and confidence intervals for each respective database size and workload are presented in

**Table** 20 to **Table 22** below.

Table 20: Differences of means and confidence intervals for a database containing 1k records.



Table 21: Differences of means and confidence intervals for a database containing 100k records.



Table 22: Differences of means and confidence intervals for a database containing 10k records.



All resulting confidence intervals do not include the value, thus at 95% confidence, there is no evidence to prove that there is not a statistically significant difference between the results of the measurements taken from the Redis and MongoDB databases. Thus, the differences between the results obtained through experimentation of Redis and MongoDB can be considered statistically significant.

Additional charts visualizing the gathered data are provided below.

Figure 13. Execution Time vs Database Size for each Workload for Redis

Figure 14. Execution Time vs Database Size for each Workload for MongoDB

Figure 15. Workload A Execution Time vs Database Size for each Database

Figure 16. Workload B Execution Time vs Database Size for each Database

Figure 17. Workload F Execution Time vs Database Size for each Database

## Individual Relational Database Analysis

### Raw data

Since each of the five queries was run 30 times for both MySQL and PostgreSql, to get a population sample, there are 1200 execution times recorded in the performance analysis. The data was originally split up by initial configuration. The data for the small initial configuration is given in Figure 18. The data for the medium initial configuration is given in Figure 19. The data for the large initial configuration is given in Figure 20. Finally the data for the extra-large initial configuration is given in Figure 21. Standard deviation and average are found from the dataset and given at the bottom of the figures.



Figure 18. Small Relational Database Execution Times



Figure 19. Medium Relational Database Execution Times



Figure 20. Large Relational Database Execution Times



Figure 21. Extra-Large Relational Database Execution Times

### Kiviat Diagrams

To compare the two databases by initial configuration size it would be helpful to graph the information on the same graph. A kiviat diagram is used to show the difference in execution time. The data presented in the kiviat diagrams (Figure 22, Figure 23, Figure 24, and Figure 25) are just a visual representation of data presented in the tables above.

Figure 22. Kiviat Diagram for Small Relational Database Execution Times

Figure 23. Kiviat Diagram for Medium Relational Database Execution Times

Figure 24. Kiviat Diagram for Large Relational Database Execution Times

Figure 25. Kiviat Diagram for Extra-Large Relational Database Execution Times

### Confidence Intervals

Since there were 30 values taken for each of the queries there is enough data to assume that the population values can be found. This allows for a difference of alternatives to be used. Since the average and standard deviation of each of the queries is known for each query the data can be taken to find the difference between the two databases.

The difference in means can be found by taking the average execution of the first sample set (MySQL) and subtracting the average execution time from the second sample set (PostgreSql). This difference of means will be used when calculating the confidence interval.

The standard deviation of the difference is found by taking the standard deviations of the first and second dataset as well as the sample size of both of the datasets. Since there were 30 runs of each query the sample size for both datasets is equal to 30.

The confidence interval for all the comparisons is done at 90% confidence. Using a 2-tail lookup in the z table it is found that the z-score needed is 1.64.

The high and low values of the confidence interval can be found by taking the difference of means and adding and subtracting out the z-score times the standard deviation of the difference. If 0 doesn’t fall between c1 and c2 then the data is significantly significant. This formula is used on all 5 queries for the four initial configurations. The confidence intervals are given, below, in Figure 26.

|  |  |
| --- | --- |
| (a) | (b) |
| (c) | (d) |

Figure 26. Confidence Interval for Alternatives (90%)

### Time vs. Execution Size

To visually see how changing the size of the database affects the execution time of the query the data is mapped per query. The execution time increases in each database when the data is larger, however the growth is not equal in both databases. The figures below (Figure 27, Figure 28, Figure 29, Figure 30, and Figure 31) show this relationship for each of the configuration sizes.

Figure 27. Query 1 Execution Time vs. Size

Figure 28. Query 2 Execution Time vs. Size

Figure 29. Query 3 Execution Time vs. Size

Figure 30. Query 4 Execution Time vs. Size

Figure 31. Query 5 Execution Time vs. Size

### Regression Analysis

The regression equation in excel was used to calculate the regression model as well as the coefficient of determination and the correlation coefficient. The Figures below give the data for all regression analysis models.



Figure 32. PostgreSql Query 1 Regression



Figure 33. PostgreSql Query 2 Regression



Figure 34. PostgreSql Query 3 Regression



Figure 35. PostgreSql Query 4 Regression



Figure 36. PostgreSql Query 5 Regression



Figure 37. MySQL Query 1 Regression



Figure 38. MySQL Query 2 Regression



Figure 39. MySQL Query 3 Regression



Figure 40. MySQL Query 4 Regression



Figure 41. MySQL Query 5 Regression

# Group Database Analysis

After conducting 30 replications of the experiments prescribed in the **Group Comparison of Databases** section of this paper, the sample executions times for each of the queries on each database were collected. The sample executions times for Redis (NoSQL database), Neo4j (graph database), and MySQL (SQL database) are recorded below in **Table 23, Table 24**, and **Table 25**, respectively.

Table 23: The NoSQL database consistently performed well in all categories of queries, but was nonetheless out-performed by the graph database, including during execution of the NoSQL query.

|  |  |
| --- | --- |
| **Experiment** | **Samples Execution Times (ms)** |
| SQL Query | 2703, 2854, 2865, 2834, 2884, 2899, 2865, 2910, 2895, 2916, 2880, 2916, 2927, 2897, 2933, 2911, 2898, 2922, 2926, 2926, 2921, 2957, 2926, 2946, 2919, 2925, 2928, 2946, 2943, 2921 |
| Graph Query | 185658, 185337, 185824, 186842, 186493, 187747, 185964, 184907, 185622, 184752, 185056, 186243, 184721, 187044, 184891, 185679, 186752, 186429, 187796, 187515, 184963, 185860, 186173, 185428, 186311, 185350, 184909, 185101, 185186, 185597 |
| NoSQL Query | 93, 115, 93, 110, 94, 93, 94, 109, 110, 109, 94, 109, 94, 94, 94, 93, 125, 94, 94, 109, 94, 141, 109, 109, 94, 94, 94, 93, 110, 109 |

Table 24: While the graph database is optimized for graph structures, it performed well in each of the categories, consistently performing better than the other databases, even when executing the queries native to the other databases.

|  |  |
| --- | --- |
| **Experiment** | **Samples Execution Times (ms)** |
| SQL Query | 204, 140, 94, 109, 110, 93, 95, 78, 78, 63, 62, 78, 63, 62, 78, 63, 62, 78, 63, 62, 79, 68, 62, 63, 78, 63, 78, 62, 78, 63 |
| Graph Query | 1862, 906, 719, 703, 692, 656, 688, 672, 703, 687, 718, 702, 672, 691, 672, 687, 709, 688, 703, 662, 687, 681, 689, 687, 671, 719, 759, 719, 828, 735 |
| NoSQL Query | 16, 0, 0, 15, 0, 16, 0, 0, 16, 0, 0, 15, 0, 0, 0, 16, 0, 0, 16, 0, 15, 0, 0, 16, 0, 0, 15, 0, 0, 16 |

Table 25: Although MySQL is optimized for relational data sets, this SQL database performed poorly in all categories, requiring a longer execution time for all queries than both the graph and NoSQL databases.

|  |  |
| --- | --- |
| **Experiment** | **Samples Execution Times (ms)** |
| SQL Query | 6498, 6291, 6208, 6255, 6224, 6228, 6222, 6239, 6240, 6229, 6396, 6208, 6271, 6227, 6193, 6270, 6239, 6223, 6309, 6223, 6224, 6238, 6213, 6223, 6199, 6229, 6239, 6212, 6200, 6241 |
| Graph Query | 268171, 268031, 268934, 269560, 269708, 269227, 269249, 267880, 268114, 268017, 270366, 268543, 268750, 268564, 267943, 267955, 267948, 269429, 268071, 268400, 269122, 268732, 268615, 269517, 269280, 268067, 268288, 268293, 268420, 269303 |
| NoSQL Query | 3365, 3271, 3354, 3348, 3290, 3354, 3348, 3303, 3307, 3317, 3302, 3384, 3282, 3338, 3324, 3300, 3353, 3323, 3297, 3334, 3317, 3284, 3340, 3333, 3332, 3339, 3335, 3318, 3366, 3304 |

Based on these three database options, crossed with the three categories of queries selected (along with 30 replications for each experiment), two-factor Analysis of Variance (ANOVA) can be conducted to surmise the statistical significance of the results of the experiments conducted on each database. Based on the samples recorded above, the resulting mean and variance values for each of the databases and each category is recorded in **Table 26**.

Table 26: The mean and variance for each database, crossed with each query category, reflects the sample data recorded in Table 23, Table 24, and Table 25.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **NoSQL (ms)** | **Graph (ms)** | **SQL (ms)** |
| **SQL Query** | **Mean** | 2,903.10 | 80.97 | 6,247.03 |
| **Variance** | 2,247.68 | 876.65 | 3,799.34 |
| **Graph Query** | **Mean** | 185,871.67 | 745.57 | 268,683.23 |
| **Variance** | 795,442.78 | 46,936.67 | 430,723.36 |
| **NoSQL Query** | **Mean** | 102.23 | 5.73 | 3,325.40 |
| **Variance** | 135.36 | 58.82 | 765.70 |

Based on the recorded sample data, the sums of squares () for the database factor (), query factor (), interaction between the database factor and query factor (), within each factor (), and total () can be calculated as follows,

where is the number of databases compared (thus, ), is the number of queries compared (thus, ), and is the number of replications (thus, ). The degrees of freedom for each of these two-factor ANOVA components can be likewise calculated as follows,

In order to provide a statistical comparison between each of the two-factor ANOVA components, the mean square () terms for each component must be calculated. These terms can be computed as follows,

Using these three sets of equations (, , and ), the sum of squares, degrees of freedom, and mean square values for each of the databases, crossed with each query category, (with ) is recorded below in **Table 27**.

Table 27: Based on the provided equations, the sum of squares, degrees of freedom, and mean square values can be calculated for each two-factor ANOVA component.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Sum of Squares ()** | **Degrees of Freedom ()** | **Mean Square ()** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

In order for there to be a statistical significance in the variance between each factor, the F-statistic for , , and must be computed. This F-statistic value is then compared to a known, tabular F-statistic value (called F-critical). If the F-statistic value is greater than the known F-critical value, the variations seen between each of the factors is statistically significant (or, in strict terms, there is no evidence to support the hypothesis that there is no statistical significance to these variations). The F-statistic and F-critical values for , , and are recorded in **Table 28**.

Table 28: Using the F-statistic and F-critical values, it is found that the variation for all two-factor ANOVA components are statistically significant.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **F-Statistic** | **F-Critical** |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Based on these results, it is clear that there is a statistical significance to the variation discovered between each database for each query category. It was also found during this computation that the p-value, or the probability that the variations between the databases and query categories is a result of random chance, is equal to for all interactions (interactions , , and ).

While this calculation proves that there is no evidence that there does not exist a statistical significance between each group of samples from each database crossed with each query category, it does not provide enough granularity to prove that there does not evidence that there is not a statistical significance *between* each of the database-query sample groups. In order to make this judgement, a contrast must be created. A contrast, , is defined as

with the constraint that

Using this contrast value, two alternatives can be compared by creating a confidence interval for the contrast and testing whether this confidence interval contains the value 0. This confidence interval (using the calculations presented in [Lil04]) is equal to

where

Since the ANOVA analysis performed is two-factor in nature, this simple contrast cannot be performed. Instead, the two-factor ANOVA is divided into three sets of single-factor ANOVA analyses: (1) one for the NoSQL query, (2) one for the graph query, and (3) one for the SQL query [Zai14]. This allows a single-factor contrast to be performed for each of the queries, allowing the analyst to surmise the statistical significance of the differences between the mean execution time of each database completing each query.

For example, a contrast can be completed for the NoSQL database vs. the graph database while completing the NoSQL query; the NoSQL database vs. the SQL database completing the NoSQL query; and the graph database vs. the SQL database completing the NoSQL query. This process can be completed for the remaining queries, resulting a total of nine confidence intervals. These confidence intervals, along with the calculations for each, are presented below (all calculations are performed at and ). The notation used is

* : The confidence interval, , for the contrast between databases and executing query .
* : Effect of database while executing query
* : The mean-square error value for query
* : The standard deviation of the contrast, , for query

**NoSQL Query**

*Error Mean Square Computation:*

*Database Effect Computations:*

*Contrast Computations:*

*Standard Deviation of Contrast Computation:*

Note that the term is the same for all computations of and will not be calculated again for each

*Confidence Interval Computations:*

Since all confidence intervals for the NoSQL query do not contain the value , there is no evidence to suggest that there is not a statistical significance in the differences seen between the mean of the execution times sampled from each of the databases.

**Graph Query**

*Error Mean Square Computation:*

*Database Effect Computations:*

*Contrast Computations:*

*Standard Deviation of Contrast Computation:*

*Confidence Interval Computations:*

Based on these results, there is no evidence to suggest that the differences between the mean execution times of the graph database and NoSQL database are not statistically significant, but based on the confidence interval for the NoSQL-SQL comparison, it has been shown that the difference is statistically insignificant at confidence.

The confidence level that would suggest statistical significance can be calculated as follows:

**SQL Query**

*Error Mean Square Computation:*

*Database Effect Computations:*

*Contrast Computations:*

*Standard Deviation of Contrast Computation:*

*Confidence Interval Computations:*

Based on the above computations, it can be seen that there does not exist evidence that there is not a statistically significant difference between the mean execution times of the all three databases for the SQL query.

1. While graph databases are often categories under NoSQL databases, graph databases have seen such widespread use apart from the typical NoSQL umbrella that we feel as though it warrants its own performance analysis in the context of this paper, rather than an analysis as a subset of NoSQL databases [↑](#footnote-ref-2)
2. http://neo4j.com/ [↑](#footnote-ref-3)
3. http://www.orientechnologies.com/orientdb/, While OrientDB is a document-based NoSQL database, Orient Technologies has included a graph database layer for OrientDB, and therefore, it has become a graph database that is in common use among the open source graph database community. This database is also included in the analysis of both [MaC14] and [Joe13]. [↑](#footnote-ref-4)
4. http://www.tinkerpop.com/ [↑](#footnote-ref-5)
5. Note that all experiments conducted, including those that were able to complete with the default memory size, were given of memory to execute. Those experiments that were able to execute with the default memory size were re-executed with the memory size, and the resulting sample values and means recorded in this paper reflect the experiments conducted with this larger-than-default memory size. [↑](#footnote-ref-6)
6. Interestingly, this judgement contradicts the findings in [MaC14], which found that Neo4j had a lower memory footprint than OrientDB (although, both were found to have the two highest memory footprints). While this contradiction holds, [MaC14] used a data set that was much smaller than the one used in this paper to compare the memory footprint, and therefore, there is a possibility that Neo4j has a higher memory footprint than OrientDB as a large graph is injected into the database. [↑](#footnote-ref-7)
7. While OrientDB was able to complete larger workloads than Neo4j, this capability was not considered greater than the ability of Neo4j to complete the comparable workloads with a much lower execution time. During the group comparison of graph, NoSQL, and SQL databases, the data set is relatively small, and therefore, the ability of OrientDB to complete larger workloads than Neo4j is essential mute (since the selection of OrientDB would mean lower execution times for the size of workload projected for the group comparison and the ability of OrientDB to complete a large workload would go unused). Therefore, this capability of OrientDB is not applicable to the nature of the group comparison and Neo4j was chosen as the best performing graph database. [↑](#footnote-ref-8)
8. Although Redis is not officially supported on Windows (the operating system on which the experiments were executed), the Microsoft Open Technology Group has created a Windows port of Redis, found at https://github.com/MSOpenTech/redis. This unofficial port of Redis was used to execute the experiments conducted in this section (the use of this unofficial version of Redis is not applicable to the data obtained in the individual analysis of the NoSQL databases under consideration in this paper). [↑](#footnote-ref-9)