A Comparison of a Graph Database and a Relational Database

A Data Provenance Perspective

Chad Vicknair
Department of Computer and Information Science
University of Mississippi
cmvickna@olemiss.edu

Michael Macias
Department of Computer and Information Science
University of Mississippi
mamacias@olemiss.edu

Zhendong Zhao
Department of Computer and Information Science
University of Mississippi
zzhao@olemiss.edu

Xiaofei Nan
Department of Computer and Information Science
University of Mississippi
xnan@olemiss.edu

Yixin Chen
Department of Computer and Information Science
University of Mississippi
ychen@cs.olemiss.edu

Dawn Wilkins
Department of Computer and Information Science
University of Mississippi
dwilkins@cs.olemiss.edu

ABSTRACT

Relational databases have been around for many decades and are the database technology of choice for most traditional data-intensive storage and retrieval applications. Retrievals are usually accomplished using SQL, a declarative query language. Relational database systems are generally efficient unless the data contains many relationships requiring joins of large tables. Recently there has been much interest in data stores that do not use SQL exclusively, the so-called NoSQL movement. Examples are Google's BigTable and Facebook's Cassandra. This paper reports on a comparison of one such NoSQL graph database called Neo4j with a common relational database system, MySQL, for use as the underlying technology in the development of a software system to record and query data provenance information.

1. MOTIVATION

This paper is a comparison of the relative usefulness of the relational database MySQL and the graph database Neo4j to store graph data. A directed acyclic graph (DAG) is a common data structure to store data provenance information relationships. The goal of this study was to determine whether a traditional relational database system like MySQL, or a graph database, such as Neo4j, would be more effective as the underlying technology for the development of a data provenance system.

A graph is one of the fundamental data abstractions in computer science. As such, there are many other applications of graphs. Virtually every graph application has a need to store and query the graph. Recently there has been increased interest in graphs to represent social networks, web site link structures, and others. Within the field of biology itself, there are many uses for graphs, including metabolic networks, protein-protein interaction networks, chemical structure graphs, gene clusters, and genetic maps. Graphs truly are one of the most useful structures for modeling objects and interactions.

Within computer science, there currently exist several options for storing data outside of a traditional relational model. BigTable is, in effect, a database system created and used by Google. It is fast, large-scale, and distributed. It contains characteristics of both row-oriented and column-oriented databases [2]. Dynamo is a key-value storage system used extensively by Amazon because of its scalability, reliability, efficiency, and ability to be highly distributed [3]. Facebook developed Cassandra, an open-source, distributed key-value data store, which, like BigTable, is a row-based/column-based hybrid system [6]. Project Voldemort[8] is LinkedIn's large-scale, persistent hash table. It is open-source software that functions in a distributed environment and is designed to gracefully handle errors.

In the past few years, users have begun to reach beyond the relational model. At some point, the cost-benefit of adopting or creating a new storage model is almost always addressed. Relational databases have been the workhorse of the database industry for decades, but many new searchintensive applications recently seem to be adopting alternative models.

"Provenance" means lineage. The provenance of a data item is the lineage of that data item, describing what it is and how it came to be. Provenance about a data item includes details about processes used to create the item and provenance about input data used to create the item. Complete provenance of an item might include source data recorded by hand, by sensors, etc. and all of the subsequent processes and file versions that came to make up the item. Provenance of data can exist at different granularities. For

ACMSE '10, April 15-17, 2010, Oxford, MS, USA

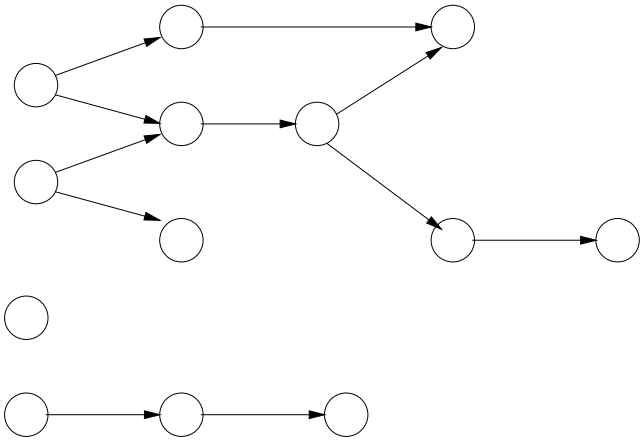

Figure 1: A sample DAG.

example, provenance can refer to entire databases, tuples within the database, or files.

Very often, provenance is stored as a directed acyclic graph (DAG). See Figure 1. While storing a graph in a relational database is simple, querying it, particularly traversing it, can be time-inefficient due to the number of potential joins. As such, it is appropriate to consider non-traditional data storage models. Since the logical model for data provenance is a DAG, a graph database seems to be a logical alternative model. This paper documents a benchmark study to compare the relational model to the graph model before making a final decision about a data store.

2. BACKGROUND

The relational database management system was first created in the 1970s. Since then, its popularity has skyrocketed, and it has become a primary data storage structure in both academic and commercial pursuits. Relational databases range from small, personal databases like Microsoft Access to large-scale database servers like Oracle, Microsoft SQL Server, and MySQL. This paper focuses on MySQL.

Research into graph databases was popular in the early 1990s, but died out for a series of reasons including the surge of hypertext and XML research [1]. With the rise of the Internet as a tool for the general public, data began to increase both in volume and interconnectedness. The graph model was used to represent tremendous amounts of data more often than it had in the past. Traditional data stores were often capable of handling graph data. Yet, they were often neither designed to do so nor efficient at it. There was a clear desire for a data store tailored to the needs of graph data.

2.1 NoSQL

In recent years, software developers have been investigating storage alternatives to relational databases. NoSQL is a blanket term for some of those new systems. Cassandra, BigTable, CouchDB, Project Voldemort, and Dynamo are all NoSQL projects, as they are all high-volume data stores that actively reject the relational and object-relational models.

Atomicity, consistency, isolation, and durability (ACID) are a set of governing principles of the relational model. Together, they guarantee database reliability. NoSQL rejects ACID.

The term "NoSQL," as a term for modern web data stores, first began to gain popularity in early 2009. It is a topic that has gained recognition from the IT community but has yet to garner large-scale academic study. Still, the NoSQL movement has its own discussion groups, blogs, and conferences.

As the typical database administrator attempts to question whether to move from the relational model to a NoSQL model, the NoSQL community presents him or her with potential flags that the data might be more suitable for a NoSQL system [5].

1. Having tables with lots of columns, each of which is only used by a few rows.
2. Having attribute tables.
3. Having lots of many-to-many relationships.
4. Having tree-like characteristics.
5. Requiring frequent schema changes.

Data provenance meets several of these criteria, so it would be fitting to investigate NoSQL solutions to the provenance storage problem. An experiment was conducted to test the viability of a NoSQL data store, Neo4j, for data provenance needs.

3. DESIGN OF EXPERIMENTS

3.1 Measures

The evaluation methodology designed to compare MySQL and Neo4j involves objective benchmarks and subjective comparisons based on system documentation and experience. The objective tests include processing speed based on a predefined set of queries, disk space requirements, and scalability. Subjective tests include maturity/level of support, ease of programming, flexibility, and security.

3.2 Setup

A total of twelve MySQL databases and twelve Neo4j databases were constructed as described below. Each database stores a DAG consisting of some number of nodes and edges. The databases contains only enough structural information to represent the DAG and some payload data associated with each node. The payload data is meant to represent data that might be associated with the node object in the DAG. Thus, each node in the graph database and the node table in the relational database contain a payload attribute. For the data provenance application, the character payload would likely be semistructured data in the form of XML or JavaScript Object Notation (JSON).

A partial schema of the relational database is as follows:

- node(nodeid, payload)
- edge(source, sink)

Source and sink are both foreign keys to node.nodeid, and indicate an edge exists in the graph directed from source to sink.

Graphs were created to contain approximately 1000, 5000, 10000, and 100000 nodes to aid in assessing scalability. The

Table 1: Databases with sizes.

Database	#Nodes	Data Type	MySQL Size	Neo4j Size
1000int	1000	Int	0.232M	0.428M
5000int	5000	Int	0.828M	1.7M
10000int	10000	Int	1.6M	3.2M
100000int	100000	Int	15M	31M
1000char8k	1000	8K Char	18M	33M
5000char8k	5000	8K Char	87M	146M
10000char8k	10000	8K Char	173M	292M
100000char8k	100000	8K Char	1700M	2900M
1000char32k	1000	32K Char	70M	85M
5000char32k	5000	32K Char	504M	406M
10000char32k	10000	32K Char	778M	810M
100000char32k	100000	32K Char	6200M	7900M

type of payload data stored in each node also varied. The payloads consisted of random integers, random 8KB strings, and random 32KB strings. Thus, twelve MySQL and twelve Neo4j databases were constructed from the random graphs. Table 1 gives details about the databases and the disk space required for each. Both the MySQL and the Neo4j databases used full-text indexing. In general, each Neo4j database was about 1.25 to 2 times the size of the corresponding relational database. The MySQL database was larger only once.

The DAGs themselves were randomly generated both in structure and in payload data. The payload data consisted of a series of words pulled from a dictionary of roughly 112,000 words. Furthermore, all data was randomly generated for each of the 12 databases independently. The corresponding Neo4j and MySQL databases are logically identical in both structure and payload so the correctness of the queries could be compared and verified.

The system used for testing runs Ubuntu Linux, version 9.10. It has an Intel Core 2 Duo CPU running at 3.00 GHz and has 4 GB of RAM. The benchmarking program was the only application running when the results were generated, but the machine was connected to the Internet and standard system processes were running.

In order to generate directed acyclic graphs, the graphs were created in layers: first two random sized sets of nodes were created and a random number of edges between the two layers were added. Then additional layers of random nodes were created and random edges were established from the previous layer to the current one. For each graph size, the process was repeated until a graph of at least the target size was created.

As the graph database was being created, the data was output to a series of tab-delimited data files suitable for loading into MySQL in bulk. The goal of creating logically equivalent databases using the two models took precedence over including benchmarks for measuring time to construct the databases since in a production data provenance system the data would be incorporated incrementally, not in one big batch.

3.3 MySQL

The implementation chosen for the relational database was MySQL Community Server version 5.1.42[1]. MySQL has significant market penetration in the academic and scientific fields. Furthermore, MySQL has significant support, both from the manufacturers and from the user community. It is a pure relational database, as opposed to an object-relational database like Oracle and SQL Server.

In April 2009, Oracle acquired Sun Microsystems, the company that owns MySQL. Oracle has since been investigated by antitrust authorities in the European Union. At the time of the writing of this paper, the deal is still under investigation. The purchase has been cleared by the US Department of Justice. Oracle has pledged to continue to support MySQL. Still, the future MySQL is not entirely known at this time.

3.4 Neo4j

Neo4j version 1.0-b11[2] is the implementation chosen to represent graph databases. It is open source for all noncommercial uses. It has been in production for over five years. It is quickly becoming one of the foremost graph database systems. According to the Neo4j website, Neo4j is "an embedded, disk-based, fully transactional Java persistence engine that stores data structured in graphs rather than in tables"[7]. The developers claim it is exceptionally scalable (several billion nodes on a single machine), has an API that is easy to use, and supports efficient traversals. Neo4j is built using Apache's Lucene [3] for indexing and search. Lucene is a text search engine, written in Java, geared toward high performance.

3.5 Queries

The queries were designed to simulate some of the types of queries used in provenance systems. For example, traversals are necessary to determine data objects (nodes) derived from or affected by some starting object or node. That is, if a data object is determined to be incorrect, that information must be propagated to all "descendants" of the node. Searching for specific values within the payload is another common operation.

The queries were divided into two types: structural and data queries. The structural queries reference the DAG structure but not the payload, and data queries use the payload data.

Three structural queries were defined:

S0: Find all orphan nodes. That is, find all nodes in the

[1] `http://www.mysql.com/`

[2] `http://neo4j.org`

[3] `http://lucene.apache.org`

Table 2: Structural query results, in milliseconds.

Database	MySQL S4	Neo4j S4	MySQL S128	Neo4j S128	MySQL S0	Neo4j S0
1000int	38.9	2.8	80.4	15.5	1.5	9.6
5000int	14.3	1.4	97.3	30.5	7.4	10.6
10000int	10.5	0.5	75.5	12.5	14.8	23.5
100000int	6.8	2.4	69.8	18.0	187.1	161.8
1000char8K	1.1	0.1	21.4	1.3	1.1	1.1
5000char8K	1.0	0.1	34.8	1.9	7.6	7.5
10000char8K	1.1	0.6	37.4	4.3	14.9	14.6
100000char8K	1.1	6.5	40.9	13.5	187.1	146.8
1000char32K	1.0	0.1	12.5	0.5	1.3	1.0
5000char32K	2.1	0.5	29.0	1.6	7.6	7.5
10000char32K	1.1	0.8	38.1	2.5	15.1	15.5
100000char32K	6.8	4.4	39.8	8.1	183.4	170.0

graph that are singletons, with no incoming edges and no outgoing edges.

S4: Traverse the graph to a depth of 4 and count the number of nodes reachable.

S128: Traverse the graph to a depth of 128 and count the number of nodes reachable.

There were 3 types of data queries. The following only apply to integer databases:

I1: Count the number of nodes whose payload data is equal to some value.

I2: Count the number of nodes whose payload data is less than some value.

The following only applies to character databases:

C1: Count the number of nodes whose payload data contains some search string (length ranges from 4 to 8).

4. EXPERIMENTAL RESULTS

Each query was run 10 times on each database and execution times were collected. All times are in milliseconds (ms). The longest and shortest times were dropped, and the remaining eight times were averaged. This was done to ensure that any caching or system process activity would not affect the timings. The data queries are parameterized with a data value (integer or search string). The data values were generated randomly beforehand and the same values were used to search both the MySQL and Neo4j databases.

4.1 Objective Measures

For the traversal queries, S0, S4, and S128, Neo4j was clearly faster, sometimes by a factor of 10, as detailed in Table 2. This was expected since relational databases are not designed to do traversals. In this case, the MySQL database was accessed using a Java program and JDBC. A standard breadth-first-search (BFS) was performed and a SELECT to the database was issued for each node removed from the queue. Many such queries were executed, but no join was required since only the "edge" table was accessed. The built-in traversal framework was used for the Neo4j queries.

The query to find orphan nodes resulted in fairly comparable results between the two systems. The Neo4j API was used to retrieve a list of all nodes in the database. Those nodes were iterated through, checking each node for the presence of edges. The single SQL query below was executed in MySQL.

```
SELECT COUNT(*)
FROM node
WHERE node.nodeid NOT IN (SELECT source FROM edge)
AND node.nodeid NOT IN (SELECT sink FROM edge);
```

The data queries for the integer payload databases demonstrated the efficiency of the relational database. Neo4j uses Lucene for querying, and Lucene, by default, treats all data as text. Thus, equivalence and inequality comparisons are not very fast since conversions must be done. The results for the queries on the four integer payload databases are presented in Table 3.

Table 3: Query results on Integer databases, in milliseconds.

Database	Rel I1	Neo I1	Rel I2	Neo I2
1000int	0.3	33.0	0.0	40.6
5000int	0.4	24.8	0.4	27.5
10000int	0.8	33.1	0.6	34.8
100000int	4.6	33.1	7.0	43.9

The full-text searches for the database with character payloads yielded interesting results. Four databases with 8K character payload and four with 32K character payload fields were used. The character strings were generated randomly using from a dictionary. Full-text searches in the payload were performed. For MySQL, an example full-text SQL statement is:

```
SELECT count(*)
FROM pnode
WHERE MATCH (payload) AGAINST (SearchString);
```

For Neo4j, the Lucene indexing service was used.

In verifying the correctness of our experiment, we conducted searches based on fully random data. By using random data, we could verify correctness based on expected values. The formula is:

$$E(N, n, d) = N * (1 - (1 - (1/26)^d)^n)$$

N is the number of nodes in the database, n is the size of the payload field (8K or 32K), and d is the length of the search string. When conducting tests on fully random data with only letters, MySQL outperformed Neo4j by a healthy margin. Thus, further testing was done on data that more closely resembled real-world data that contained spaces.

Random search strings pulled from the same dictionary as the payload data of size two through eight were tested (same search string on each database).

The timing statistics show that MySQL is much slower than the graph database on these queries. At small scale, MySQL performed slightly better than Neo4j, but scaling upward dramatically shifted the search times in favor of Neo4j. Partial results for the search string searches is given in Table 4, where d is the length of the search string.

4.2 Subjective Measures

The subjective measures used in the comparison are: maturity and level of support, ease of programming, flexibility, and security. While difficult to quantify, these criteria are significant in deciding which type of database to adopt, as well as which implementation. Where necessary, we identify problems that plague a type of database versus problems that are part of a particular implementation

4.2.1 Maturity/Level of Support

Maturity refers both to how old a particular system is and to how thoroughly tested it is. More tests means more time to ensure system stability, more bugs identified, and more questions answered. Obviously, a higher rate of adoption means more time to test the system. Maturity is usually proportional to level of support. A more mature piece of software will have more people using it; more people talking about it on the web, in industry, and in academia; and more people testing it in field.

The relational database, in general, is one of the most used data stores, and it has been for years. Relational databases are used all over the world to support both commercial and academic pursuits and have spawned several commercial ventures. For example, both Oracle[4] and Microsoft[5] offer extensive support for their commercial database products. Oracle and Microsoft are both giant corporations who stake their reputations on the functionality of their databases and who have significant financial motivation to ensure their continued performance. Furthermore, market penetration of the relational model means increased user support. Relational databases benefit from having a unified language, SQL, through which to interact with the database. Because SQL does not differ greatly between implementations, support for any one implementation is usually applicable to other implementations. MySQL has extensive support both from its parent company, Sun, and its users.

Graph databases, in general, have less market penetration. Their commercial ventures are usually smaller. They lack a unified language, so support for one implementation will not necessarily work for others. Neo4j, specifically, is a commercial venture when used in a for-profit environment. It does have a reasonable amount of support from its parent company on its website. Most of its user support comes from a wiki on the Neo4j site[6]. Support is limited from outside of the Neo4j site itself, meaning if Neo4j ever collapses as a company, the majority of support for it collapses, too. While there is a small, vocal community, graph database user support pales in comparison to that of relational databases.

4.2.2 Ease of Programming

Relational databases use SQL. The common language makes transitioning between implementations easier than with graph databases. Graph databases are language-specific and have their own APIs. Thus, transitioning between graph databases is significantly more difficult than relational databases.

The actual ease of programming is task-dependent. Graph traversals are fairly simple in Neo4j. The Neo4j API contains methods for doing so. MySQL graph traversals are much more complicated and can involve looping or recursing through the graph possibly executing multiple expensive joins along the way.

Scanning a table for a particular attribute value can be extremely easy with a relational database as it can consist of a simple WHERE clause. With graph databases, the performance of retrieval queries are tightly linked to the indexing service used. Neo4j uses Lucene, which is optimized for full-text indexing. Retrieval of other types of data, such as numeric and date are significanly less efficient.

Relational databases have the ability to store graph data, and with some minor programming, a graph interface can be created to make the relational store transparent to the user. The opposite is not necessarily true. Graph databases can severely obscure some relational data. Even the textbook customer/order database, when shoehorned into a graph model, becomes significantly more difficult to comprehend. Data provenance is, inherently, a graph, but outside of provenance, this can become a huge issue.

4.2.3 Flexibility

Both relational and graph databases perform extremely well in the environments for which they were created. Flexibility is a measure of how well they perform outside of those environments.

Neo4j is small, light-weight, and efficient. As a Java component, it lacks the overhead ordinarily associated with server applications, yet it maintains its ability to perform well at server scale.

MySQL is a large server application meant to exist in a large-scale multi-user environment. While still efficient and extremely optimized, for small applications users must accept some overhead for features that are not necessarily beneficial for the small user. For example, logging may not be as important for a home user as it would be for a bank.

Neo4j has an easily mutable schema, while relational databases are less mutable. The MySQL schema can be altered once the database is deployed, but doing so is a much more significant undertaking than with Neo4j.

4.2.4 Security

Relational databases in general and MySQL specifically have extensive built-in multi-user support. Many graph databases, on the other hand, lack much support for multi-user environments. Neo4j is among them. It forces all user management to be handled at the application level.

[4] `http://www.oracle.com/us/support/index.html`

[5] `http://www.microsoft.com/sqlserver/2008/en/us/support.aspx`

[6] `http://wiki.Neo4j.org/content/Main_Page`

Table 4: Query results on Character databases, in milliseconds.

Database	Rel	Neo	Rel	Neo	Rel	Neo	Rel	Neo	Rel	Neo
	d=4	d=4	d=5	d=5	d=6	d=6	d=7	d=7	d=8	d=8
1000char8k	26.6	35.3	15.0	41.6	6.4	41.6	11.1	41.6	15.6	36.3
5000char8k	135.4	41.6	131.6	41.8	112.5	36.5	126.0	33.0	91.9	41.6
10000char8k	301.6	38.4	269.0	41.5	257.8	41.5	263.1	42.6	249.9	41.5
100000char8k	3132.4	41.5	3224.1	41.5	3099.1	42.6	3077.4	41.8	2834.4	36.4
1000char32k	59.5	41.5	41.6	42.6	30.9	41.5	31.9	41.4	31.9	35.4
5000char32k	253.4	42.3	242.9	41.5	229.4	35.3	188.5	38.5	152.0	41.5
10000char32k	458.4	36.3	468.8	41.6	468.3	41.6	382.1	41.5	298.8	36.3
100000char32k	3911.3	41.4	4859.1	33.3	6234.8	37.3	4703.3	41.5	2769.6	41.5

By extension, then, Neo4j does not have any built-in security support [4]. It assumes a trusted environment. Clearly, this presents a significant challenge for any domain where such assumptions cannot be made. The Neo4j website does contain some rudimentary design for an access control list (ACL) security mechanism, but like multi-user support, ACL management is handled at the application level. MySQL, as with many relational databases, contains extensive support for ACL-based security.

5. CONCLUSIONS

Both systems performed acceptably on the objective benchmark tests. In general, the graph database did better at the structural type queries than the relational database. In full-text character searches, the graph databses performed significantly better than the relational database. The fact that the indexing mechanism used in the graph database was based on strings made the numeric queries less efficient. While a query on non-integer numeric data, such as doubles, was not included in the benchmark tests, the result would have likely been even worse for the graph database. The documentation on the Lucene site suggests padding numerics with spaces or zeroes, as appropriate. While this works, it is too restrictive for the purpose of storing user-supplied parameters and values in the payload. In scientific data, it is not reasonable to set a particular precision level since some parameters might require two decimal places and others may require ten or more. Speed issues related to index searching in Neo4j for numbers are related to the Lucene. This problem is known, and at the time of this writing, numerical indexing is being developed for Lucene.

The other factor that must play a key role in choosing a database system for a data provenance system is security. The lack of support in Neo4j is an issue.

Further investigation into Neo4j might yield workarounds to the search issues documented here. There are add-on components that allow Neo4j to be accessed as a Resource Description Framework (RDF) store, and potentially queried with SPARQL, an RDF query language, which has a W3C recommendation. That functionality is not well documented and was not tested in this study.

Overall, for the data provenance project, it seems premature to use the graph database for a production environment where many queries will be on parameters stored in a semistructured way even in the face of Neo4j's much better string searches. In addition, the need for securing user data is imperative. And lack of support in Neo4j is a significant drawback.

6. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under award number EPS-0903787. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. The authors appreciate the suggestions from our group members Sheng Liu and T.J. Clayton.

7. REFERENCES

[1] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1–39, 2008.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[4] T. Ivarsson. [neo] security. `http://lists.neo4j.org/pipermail/user/2009-November/001955.html`, 2009.

[5] M. Kleppmann. Should you go beyond relational databases? `http://carsonified.com/blog/dev/should-you-go-beyond-relational-databases/`, 2009.

[6] A. Lakshman. Cassandra - a structured storage system on a p2p network. `http://www.facebook.com/note.php?note_id=24413138919`, 2008.

[7] Neo4j. Home. `http://neo4j.org`, 2009.

[8] Project Voldemort. Project voldemort: A distributed database. `http://project-voldemort.com/`, 2009.