A New Representation of WordNet® using Graph Databases

Khaled Nagi
Dept. of Computer and Systems Engineering
Faculty of Engineering, Alexandria University
Alexandria, Egypt
khaled.nagi@alexu.edu.eg

***Abstract*— WordNet® is one of the most important resources in computation linguistics. The semantically related database of English terms is widely used in text analysis and retrieval domains, which constitute typical features, employed by social networks and other modern Web 2.0 applications. Under the hood, WordNet® can be seen as a sort of read-only social network relating its language terms. In our work, we implement a new storage technique for WordNet® based on graph databases. Graph databases are a major pillar of the NoSQL movement with lots of emerging products, such as Neo4j. In this paper, we present two Neo4j graph storage representations for the WordNet® dictionary. We analyze their performance and compare them to other traditional storage models. With this contribution, we also validate the applicability of modern graph databases in new areas beside the typical large-scale social networks with several hundreds of millions of nodes.**

Keywords-WordNet®; semantic relationships; graph databases; storage models; performance analysis.

I. Introduction

WordNet® [1] is a large lexical database of English terms and is currently one of the most important resources in computation linguistics. Several computer disciplines, such as information retrieval, text analysis and text mining, are used to enrich modern Web 2.0 applications; typically, social networks, search engines, and global online marketplaces. These disciplines usually rely on the semantic relationships among linguistic terms. This is where WordNet® comes to action.

A parallel development over the last decade is the emergence of NoSQL databases. Certainly, they are no replacement for the relational database paradigm. However, Web 2.0 builds a rich application field for managing billions of objects that do not have the regular and repetitive pattern suitable for the relational model. One major type of NoSQL databases is the *graph database* model. Since social networks can be easily modeled as one large graph of interconnected users, they can be the killer application for graph databases.

However, little to no work has been done to investigate the use of graph database management systems in moderate sized databases. Of course, the database has to be relationship-rich for the implementation to make sense. In our work, we implement a new storage technique for WordNet® based on Neo4j [2]; currently, a leading graph database. WordNet® dictionary has several characteristics that promote our proposition: *it is used in several modern Web 2.0 applications*, such as social networks; *it is has a moderate size of datasets*; and *traversing the semantic relationship graph is a common use case*.

Since the modeling and benchmarking experiences of these new graph databases are not as established as in the relational database model, we implement two variations and conduct several performance experiments to analysis their behavior and compare them to the relational model.

The rest of the paper is organized as follows. Section II provides a background on WordNet® and its applications as well as a brief survey on graph database technology. Our proposed system is presented in Section III. Section IV contains the results of our performance evaluation and Section V concludes the paper and presents a brief insight in our future work.

II. Background

A. WordNet®

The WordNet® project began in the Princeton University Department of Psychology, and is currently housed in the Department of Computer Science. WordNet® is a large lexical database of English [1]. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. A synset contains a brief definition (gloss). Synsets are interlinked by means of conceptual-semantic and lexical relations. WordNet® labels the semantic relations. The most frequently encoded relation among synsets is the super-subordinate relation (also called hyperonym, hyponym or IS-A relation). Other semantic relations include meronyms, antonyms, and holonyms. The majority of the WordNet®'s relations connect words from the same part of speech (POS). Currently, WordNet® comprises 117,000 synsets and 147,000 words.

Today, WordNet® is considered to be the most important resource available to researchers in computational linguistics, text analysis, text retrieval and many related areas [3]. Several projects and associations are built around WordNet®.

The Global WordNet Association [4] is a free, public and non-commercial organization that provides a platform for discussing, sharing and connecting wordnets for all languages in the world. The Mimida project [5], developed by Maurice Gittens, is a WordNet-based mechanically-generated multilingual semantic network for more than 20 languages based on dictionaries found on the Web. EuroWordNet [6] is a multilingual database with wordnets for several European languages (Dutch, Italian, Spanish, German, French, Czech and Estonian). It is constructed according to the main principles of Princeton's WordNet®. One of the main results of the European project that started in 1996 and lasted for 3 years is to link these wordnets to English WordNet® and to provide an Inter-Lingual-Index to connect the different wordnets and other ontologies [7]. MultiWordNet [8], developed by Luisa Bentivogli and others at ITC-irst, is a multilingual lexical database. In MuliWordNet, the Italian WordNet is strictly aligned with the Princeton WordNet®. Unfortunately, it comprises a small subset of the Italian language with 44,000 words and 35,400 synsets. Later on several projects; such as ArchiWN [9], attempt to integrate WordNet with domain-specific knowledge.

RitaWN [10], developed by Daniel Howe, is an interesting library built on WordNet®. It provides simple access to the WordNet ontology for language-oriented artists. RitaWN provides semantically related alternatives for a given word and POS (e.g., returning all synonyms, antonyms, hyponyms for the noun "cat"). The library also provides distance metrics between ontology terms, and assigns unique IDs for each word sense/pos.

Several projects aim at providing access to the WordNet® native dictionary. For example, JWNL [11] provides a low-level API to the data provided by the standard WordNet® distribution. In its core, RitaWN uses JWNL to access the native file-based WordNet® dictionary. Other projects, such as WordNetScope [12], WNSQL [13], and wordnet2sql® [14], provide a relational database storage for WordNet®.

B. *Graph Databases*

NoSQL databases are older than relational databases. Nevertheless, their renaissance came first with the emergence of Web 2.0 during the last decade. Their main strengths come from the need to manage extremely large volumes of data that are collected by modern social networks, search engines, global online marketplaces, etc. For this type of applications, ACID (Atomicity, Consistency, Isolation, Durability) transaction properties [15] are simply too restrictive. More relaxed models emerged such as the CAP (Consistency, Availability and Partition Tolerance) theory or eventually consistent [16], which in general means that any large scale distributed DBMS can guarantee for *two* of *three* aspects: *Consistency*, *Availability*, and *Partition tolerance*. In order to solve the conflicts of the CAP theory, the BASE consistency model (Basically, soft state, eventually consistent) was defined for modern applications [16]. In contrast to ACID, BASE concentrates on availability at the cost of consistency. BASE adopts an optimistic approach, in which consistency is seen as a transitional process that will be *eventually* reached. Together with the publication of Google's BigTable and Map/Reduce frameworks [17], dozens of NoSQL databases emerged. A good overview of existing NoSQL database management systems can be found in [18].

Mainly, NoSQL database systems fall into four categories:

- Key-value systems,
- Column-family systems,
- Document stores, and
- Graph databases.

Graph databases have a long academic tradition. Traditionally, research concentrated on providing new algorithms for storing and processing very large and distributed graphs. These research efforts helped a lot in forming object-oriented database management systems and later XML databases.

Since social networks can be easily viewed as one large graph of interconnected users, they offer graph databases the chance for a great comeback. Since then, the whole stack of database science was redefined for graph databases. At the heart of any graph database lies an efficient representation of entities and relationships between them. All graph database models have, as their formal foundation, variations on the basic mathematical definition of a graph, for example, directed or undirected graphs, labeled or unlabeled edges and nodes, hypergraphs, and hypernodes [19]. For querying and manipulating the data in the graph, a substantial work focused on the problem of querying graphs, the visual presentation of results, and graphical query languages. Old languages such as G, G++ in the 80s [20], the object-oriented Pattern Matching Language (PaMaL) in the 90s [21], through Glide [22] in 2002 appeared. G is based on regular expressions that allow simple formulation of recursive queries. PaMaL is a graphical data manipulation language that uses patterns. Glide is a graph query language where queries are expressed using a linear notation formed by labels and wildcards. Glide uses a method called GraphGrep [22] based on sub-graph matching to answer the queries.

However, modern graph databases prefer providing traversal methods instead of declarative languages due to its simplicity and ease use within modern languages such as Java. Taking Neo4j as example, when a `Traverser` is created, it is parameterized with two evaluators and the relationship types to traverse, with the direction to traverse each type. The evaluators are used for determining for each node in the set of candidate nodes if it should be returned or not, and if the traversal should be pruned (stopped) at this point. The nodes that are traversed by a `Traverser` are each visited exactly once, meaning that the returned iterator of nodes will never contain duplicate nodes [2].

Several systems such as Neo4j [2], InfoGrid [23], and many other products are available for research and commercial use today. Typical uses of these new graph database management systems include social networks, GIS, and XML applications. However, they did not find application in moderate sized text analysis applications or relationship mining.

III. Proposed Implementation

Fig. 1 provides an overview of the proposed implementation. RitaWN [10] provides synonyms, antonyms, hypernyms, hyponyms, holonyms, meronyms, coordinates, similars, nominalizations, verb-groups, derived-terms glossaries, descriptions, support for pattern matching, soundex, anagrams, etc. In Fig. 1, RitaWN is represented by an arbitrary client in this domain which sends semantic inquiries and receives the results as a list of related terms. In the actual RitaWN, the library wraps Jawbone/JWNL [11] functionality for Java processing; which, in turn, accesses the native WordNet® dictionary.

In order to separate the storage layer from the logic, we extract a `RiWordNetIF` Java interface. The interface defines methods to return semantically related words. The methods are categorized into 4 groups:

- Attribute inquiries: these methods return single attribute values for a given word, such as `String getBestPos(String w)` and `boolean isNoun(String w)`.
- Semantic relationships inquiries: in this set, methods return all semantically related words for a given word and POS, such as `String[] getHolonyms(String w, String pos)` and `String[] getHypernyms(String w, String pos)`. In our system, we define eight such methods.
- Relationship tree inquiries: in this set of methods, the library returns the whole path from the first synset for a given word and POS to the root word. Typical root words in WordNet® are “Entity” or “Object”. In our implementation, we have `String[] getHyponymTree(String w, String pos)` and `String[] getHypernymTree(String w, String pos)`; which basically trace back `getHyponym(String w, String pos)` and `getHypernym(String w, String pos)` respectively to the root word.
- Common parent inquiries: methods of this group find a common semantic path between two words in a POS subnet by traversing the WordNet® synset graph. For example, the method `String[] getCommonParent(String w1, String pos, String w2)` finds the following path dog : canis familiaris : domestic dog → domestic animal : domesticated animal → animate being : beast : animal for the nouns “dog” and “animal”. Traversal is done based on a Depth First Search algorithm with a slight adaptation to stop traversing whenever one of the synsets of the sink term `w2` is reached.

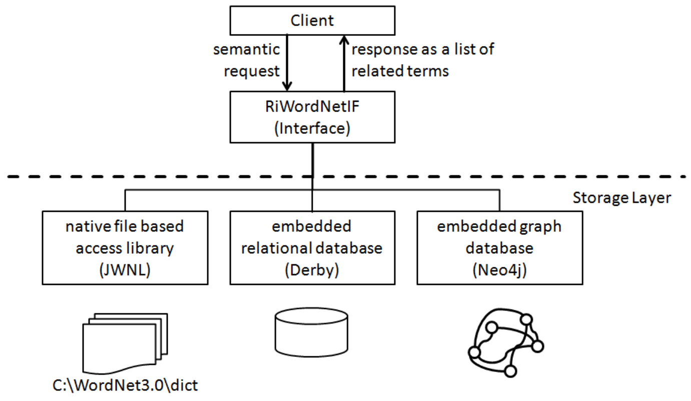
Client
semantic request
response as a list of related terms
RiWordNetIF (Interface)
Storage Layer
native file based access library (JWNL)
embedded relational database (Derby)
embedded graph database (Neo4j)
C:\WordNet3.0\dict

Figure 1. Architecture of the proposed system.

A. *Storage Layer*

In the storage layer, we provide four different representations for the WordNet® dictionary as described in the following subsections.

1) *File-based Storage*

In its original implementation, RiTa.WordNet uses the JWNL [11] library to directly browse the native dictionary provided by a standard WordNet® installation. As will be shown later, this implementation has the worst performance. We use it for validation purposes for the other three implementations.

2) *Relational Database Storage*

We use a database model similar to the one used in [14]. Fig. 2 illustrates a UML class diagram for the relevant classes. The `words` entity has a `wordid` as a primary key, the `lemma` definition and the different `POSs` are coded as string with the best POS as the first character of the string. Similarly, the `synsets` entity holds all WordNet® synsets, their `POS`, and definition. The primary key is `synsetid`. The many-to-many relationship between words and synsets is modeled by the `senses` entity. It contains the foreign keys `wordid` and `synsetid`. `Synsets` are related to each other via the `semlinks` entity. `Synset1id` points to the `from` direction and `Synset2id` to the `to` direction. The types of semantic links are defined by `linkid` which is a foreign key to the `linktype` entity. All types of links are listed in the `linktype` entity. We choose Apache Derby [24] as the database management system to hold this data model. Apache Derby is part of the Apache Group. It gained a good reputation and a high spread for applications requiring embedded relational DBMS. It is distributed as a java jar file to be added to the classpath of the application. It also comes as a stand-alone version.

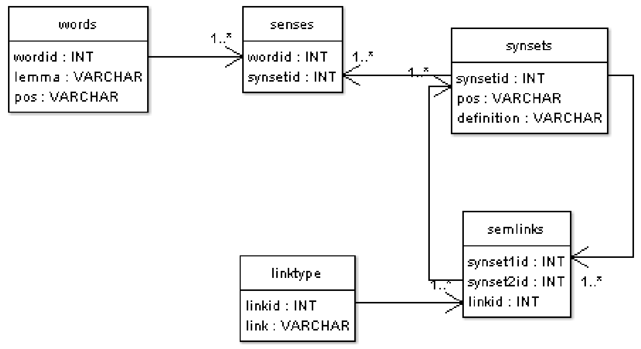

Figure 2. UML class diagram for the relational database.

3) Graph Database Storage

In our proposed work, we model the WordNet® as a graph database. An object diagram is illustrated in Fig. 3. We have two types of nodes: `words` (illustrated as ellipses) and `synsets` (illustrated as hexagons). The attributes of a word are a lemma and the different `POSs`, which are coded as a string with the best POS as the first character of the string. The `synset` has a property `definition`. There exists a bi-directional relation `Rel_sense` between `words` and `synsets`. The attribute `pos` of the relation indicates the POS associated with the sense. `Synsets` are interconnected by directed relations. These relationships `Rel_SemanticLink` carry the `type` of the link in the attribute type. For example, in Fig. 3, word *w1* has one sense as a noun with link to `sysnset` *sa* and two senses as verbs for `synsets` *sc* and *sd*. `Synset` *sa* has two hyponyms *sb* and *se* by following the relationships `Rel_SemanticLink` with type "hyponym". *w4* has one sense *sb* as a noun. *w2* and *w3* – as nouns - share the same `synset` *se*. *w5* has only one sense as a verb which is *sc*. So, if `getHoponyms("w1", "n")` is called, the result will be *w2*, *w3*, and *w4*.

4) Graph Database Storage with Extra Directly Derived Relationships

In the RiTa.WordNet application scenario, we expect lots of inquiries about semantically related words (e.g., hyponyms, synonyms, meronyms, etc.). Synsets are mainly the means to return the semantically related words. At the same time, the application is typically read-only and represents a good example for a wide range of read-only (or low-update/high-read) applications. The graph database is only updated with the release of a new WordNet® dictionary. This motivates us to augment the design mentioned in the previous section with the derived semantic relationships between words and not only synsets. The idea is similar to materialized views known in relational databases. the result of semantic relationship inquiries (e.g., `getHyponyms()`, `getSynonyms()`, `getMeronyms()`, etc.) is generated by traversing only one relationship for each result word. We intuitively expect a quicker response time at the cost of a high storage volume since the connectivity of the graph is highly increased.

In terms of implementation, these relationships are identified through the relationship type. Fig. 4 illustrates the derived relationships for the example in Fig. 3. Only the relationship of type `Rel_Hyponym` for noun POS of word *w1*; namely, *w2*, *w3*, and *w4* is drawn. For more complex inquiries of category "relationship tree" and "common parent", a combination of original and derived relationships are used in the traversal.

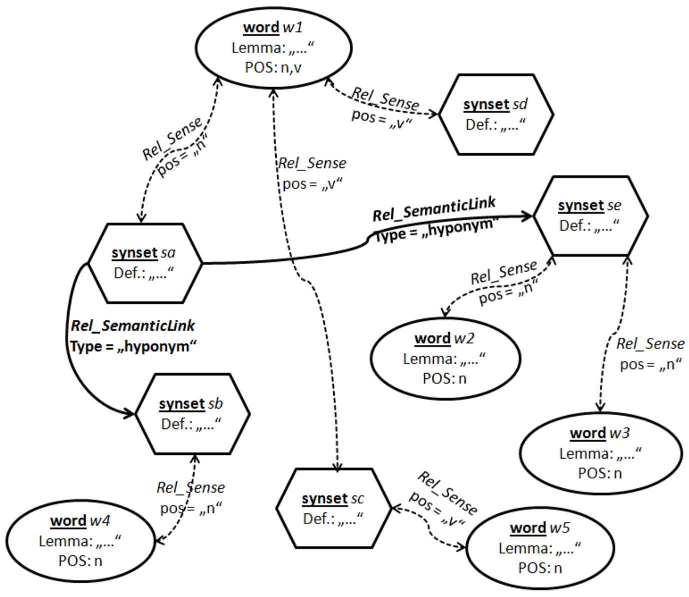

Figure 3. Object diagram for the proposed WordNet® graph database storage.

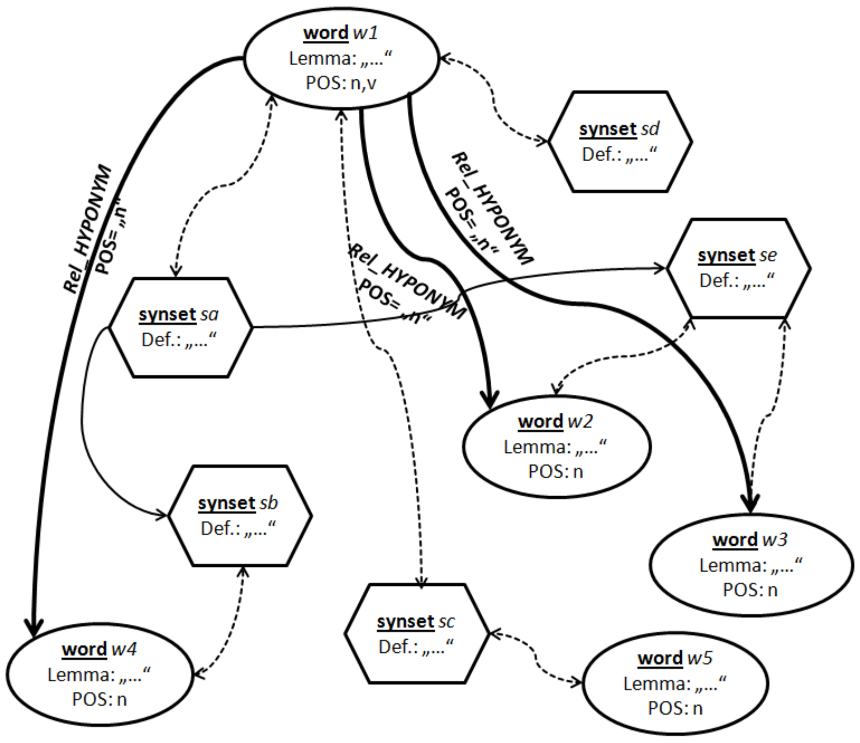

Figure 4. Object diagram with the extra derived relationships.

IV. Performance Evaluation

In order to evaluate the performance of our proposed system, we provide *four* implementations for the Java interface `RiWordNetIF` mentioned in Section III. The implementations are file-based storage, relational DBMS using Apache Derby, the graph database using Neo4j, and a second implementation using the directly derived relationships also using Neo4j.

It is important to notice that the purpose of this evaluation is to give a general impression on the performance impact and not to give concrete benchmarking figures. For sure, the optimization of all DBMS implementations; such as using indices or even exchanging the DBMS itself versus using future versions of Neo4j might lead to different results. *We would be satisfied when our proposed solution provides slightly better results than relational DBMS.* It is also clear that in-memory databases and large caching mechanisms will outperform all implementations. But we rule them out assuming memory size restrictions.

We develop a simple performance evaluation toolkit around these four implementations. A workload generator sends inquiries to all back-ends. The inquiries are grouped into four categories, as mentioned in Section III. The workload generator submits the inquiries in parallel to the application with each inquiry executing in a separate thread.

The input for the inquiry is chosen at random from an input file containing WordNet® words and their associated best POS. In case of `getCommonParent()`, another input file is used, which contains tuples of somehow related words, together with their common POS (e.g., "tiger", "cat", and "noun"). The tuples are chosen carefully to yield paths of different lengths.

The performance of the system is monitored using a performance monitor unit that records the response time of each inquiry and the number of inquiries performed by each thread in a regular time interval.

A. Input Parameters and Performance Metrics

The number of concurrent inquiry threads is increased from 1 to 50. Each experiment executes on each backend for 5 minutes in order to eliminate any transient effects. The experiments are conducted for each type of inquiries separately.

In all our experiments, we monitor the system *response time* in terms of micro-seconds per operation from the moment of submitting the inquiry till receiving the result.

We also monitor the system *throughput* in terms of inquires per hour for each thread.

B. System Configuration

In our experiments, we use an Intel CORE™ i7 vPro 2.7GHz processor, 8 GB RAM and a Solid State Drive (SSD). The operating system is Windows 7 64-bits. We use JDK 1.6.0, Neo4j version 1.6 for the graph database engine, embedded Derby™ version 10.7.1.1 for the SQL backend, JWNL library version 1.4 [11] for file system based storage.

C. Experiment Results

The performance evaluation considers all four types of inquiries:

- Attribute,
- Semantic relationships,
- Relationship trees, and
- Common parent

for the four back-end implementations.

We drop plotting the results of the native file system-based implementation from our graphs, although it is the only available implementation previous to this work. The reason behind this is that the results are far worse than the other implementations. The difference in most case is more than *one order of magnitude*.

1) Attribute Inquiries

In this set of experiments, the inquiries sent by the workload generator comprise attribute inquiries only. Both response time, illustrated in Fig. 5, and throughput, illustrated in Fig. 6, degrade gracefully with the increase in number of threads while having good absolute values. Remarkably, the simple Neo4j implementation (without the extra directly derived relationships) has a 20% better response time than the other two implementations, while the full blown Neo4j implementation has a 40% decrease in system throughput. The reason for that is the attribute inquiries are mainly affected by the node (or tuple in case of relational databases) retrieval and caching. No relationship traversal is done and hence the Neo4j only suffers from its large database size especially with the augmented directly derived relationships (see Section IV.D). In summary, this set of experiments demonstrates that the caching mechanisms of graph databases are in general as good as the relational databases and that simple operations without graph traversals are not underprivileged in this environment.

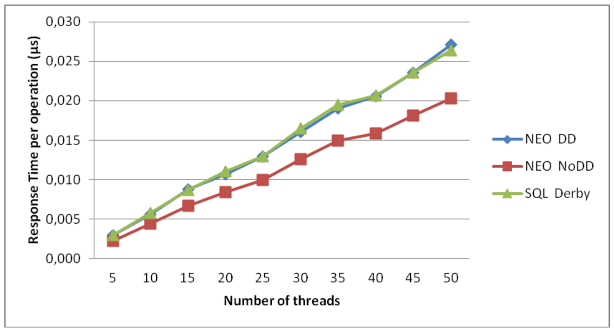

Figure 5. Response time for attribute inquiries.

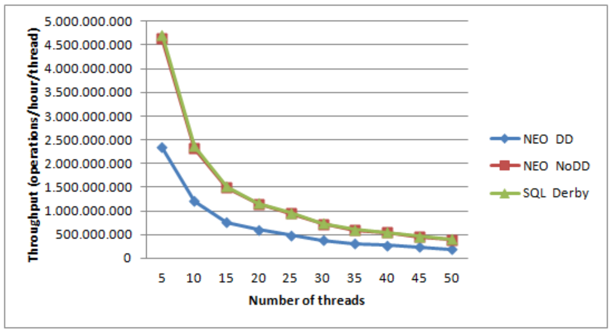

Figure 6. Throughput for attribute inquiries.

2) Semantic Relationship Inquiries

In this set of experiments, the explicit storage of semantic relationships shows its benefit. The results are retrieved by traversing one relationship only, in contrast to 3 for the simple implementation and several joins in the relational database implementation. The response time, as illustrated in Fig. 7 is enhanced by approx. 50% for all number of threads when compared to SQL Derby and 30% by adding these directly derived relationships to a simple Neo4j implementation. However, all three back-ends behave identically when it comes to throughput as illustrated in Fig. 8. The absolute values are far below those of the simple attribute inquiries described in the previous section which is expected due to the complexity of these inquiries as compared to attribute inquiries. In case of response time, it is almost 10 times higher than the previous set of experiments. The same applies to the throughput, which is lower by a factor of 10 as well.

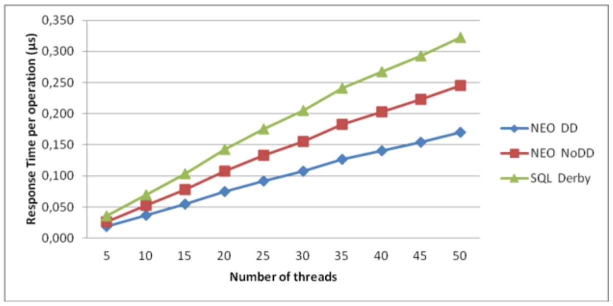

Figure 7. Response time for semantic relationship inquiries.

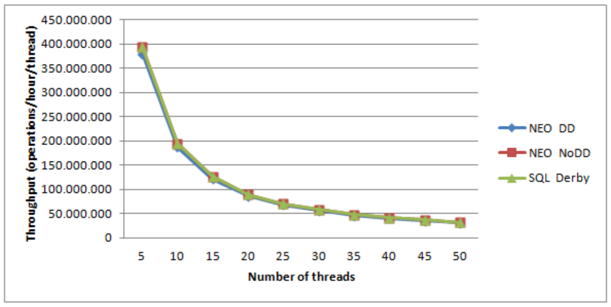

Figure 8. Throughput for semantic relationship inquiries.

3) *Relationship Tree Inquiries*

The operations of this set of experiments are more complex than the previous ones. This explains the drop in absolute values of the response time and throughput, illustrated in Fig. 9 and Fig. 10, respectively when compared to the previous experiment. This time the degradation factor is only 4. Yet, the system behavior remains the same. The response time of Neo4j with the directly derived relationships is half that's of the SQL implementation. Even without the extra relationships, the response time Neo4j is 25-30% better than the relational model. Here, again, the throughput, illustrated in Fig. 10, for all three implementations is the same. The equality of the throughput performance index of Derby and the Neo4j implementation, despite the short response time of the later, is an indication that the internal pipeline capabilities of Neo4j is *not* as good as that of the relational model.

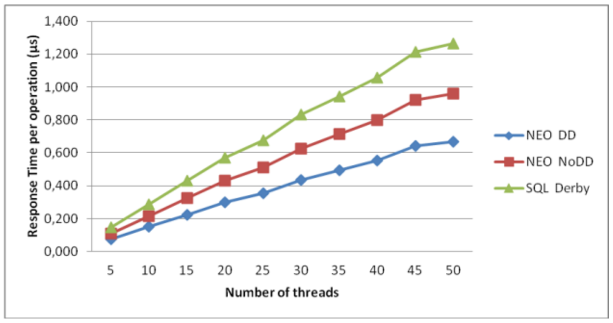

Figure 9. Response time for relationship tree inquiries.

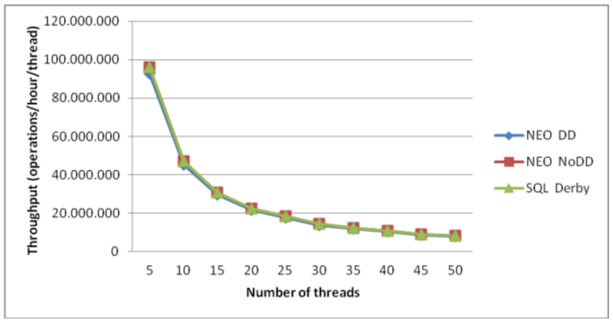

Figure 10. Throughput for relationship tree inquiries.

4) *Common Parent Inquiries*

The inquiries for this set of experiments are the most complicated among all experiments. Yet, this is a very common use case in social networks. For example, in XING [25], the user can always see all paths of relationships leading from the user to any arbitrary user in the network. No wonder here that Neo4j implementations outperform the SQL Derby implementation (and the file system implementation which seems to be not able to handle all the running threads) in requesting depth first searches of the semantic network of WordNet®. Again, Fig. 11 illustrates the extreme superiority of graph database, especially with the addition of the extra relationships. The response time is also enhanced by 45% and 30% with and without directly derived relationships, respectively. The throughput, illustrated in Fig. 12, holds its trend across all experiments of being almost the same for the three implementations (and omitting the file system implementation of course, whose values cannot be plotted with the same scale next to their counterparts).

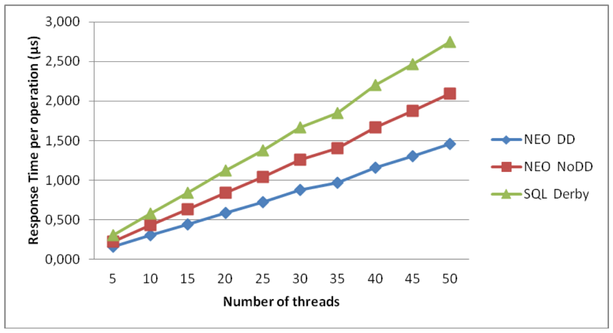

Figure 11. Response time for common parent inquiries.

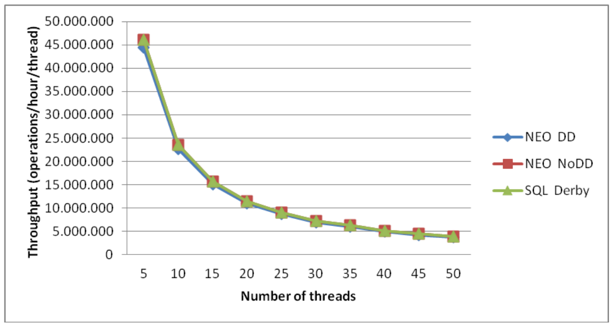

Figure 12. Throughput for common parent inquiries.

D. *Storage Requirements*

Performance in terms of good response time comes with its price. Fig. 13 illustrates the storage requirements for all four implementations. The SQL Derby and the normal Neo4j implementation occupy slightly more than double the original size of the WordNet® file-based dictionary. The redundant relationships account for more than 350 MB, making the size of the graph database 12 times larger than the file-based dictionary taken as a reference point. The good side of this particular application scenario is the absolute size of the back-ends is affordable by any desktop application.

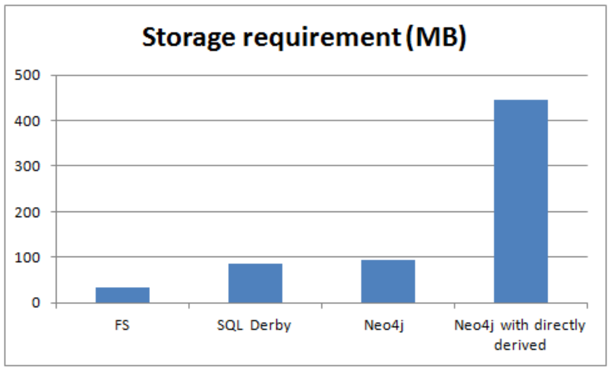

Figure 13. Storage for each backend implementation.

V. Conclusion and Future Work

In this paper, we presented two Neo4j graph storage representations for the WordNet® dictionary. We use Ri.WordNet as a typical client application that submits semantic inquiries discovering the relationships between English terms. We divide the inquiries into 4 categories depending on the complexity of their operations. Our performance analysis demonstrates that graph databases yield much better results than traditional relational databases in terms of response time even under extreme workloads thus speaking for their promised scalability. We also show that storing directly derived relationships can improve the performance by factors of 2. This redundancy has its price in terms of storage requirements, which is acceptable due to the moderate size of the database with 117,000 synsets and 147,000 terms and the read-only nature of this small scale social network.

One important contribution of this work is that it opens the door for new application areas for NoSQL databases (in this case the Neo4j graph database), namely smaller read-intensive database applications, in contrast to typical applications of the NoSQL in large scale Web 2.0 such as social networks.

Yet, this is only the beginning. In the future, we plan on benchmarking other graph database providers, such as InfoGrid [23]. We also plan on migrating several research done on relationship mining to work on graph database back-ends. If the benchmarking experiments show promising results, this will open the door for the application of graph databases in OLAP applications.

References

[1] Fellbaum, C.: WordNet and wordnets. In: Brown, Keith et al. (eds.) Encyclopedia of Language and Linguistics, Second Edition, pp. 665--670. Elsevier, Oxford , 2005.

[2] Neo4j. The World's Leading Graph Database, http://www.neo4j.org [retrieved: November, 2012].

[3] Voorhees, E.: Using WordNet for Text Retrieval. In: Fellbaum, C. (ed.) WordNet An Electronic Lexical Database, 0-262-06197-X. MIT Press, 1998.

[4] The Global WordNet Association, http://www.globalwordnet.org [retrieved: November, 2012].

[5] Mimida: A mechanically generated Multilingual Semantic Network, http://gittens.nl/gittens/topics/SemanticNetworks.html [retrieved: November, 2012].

[6] Vossen, P.: EuroWordNet: a multilingual database for information retrieval. In: Proceedings of the DELOS workshop on Cross-language Information Retrieval. Zürich , 1997.

[7] Vossen, P., Peters, W., and Gonzalo, J.: Towards a Universal Index of Meaning. In: Proceedings of the ACL-99 Siglex workshop, Maryland, 1999.

[8] Pianta, E., Bentivogli, L., and Girardi, C.: MultiWordNet: developing an aligned multilingual database. In: Proceedings of the First International Conference on Global WordNet, Mysore, India, 2002.

[9] Bentivogli, L., Bocco, A., and Pianta, E.: ArchiWordNet: Integrating WordNet with Domain-Specific Knowledge. In: Proceedings of the Second Global WordNet Conference, pp. 39—46, Brno, Czech Republic, 2004.

[10] RiTa.WordNet: a WordNet library for Java/Processing, http://www.rednoise.org/rita/wordnet/documentation [retrieved: November, 2012].

[11] Java WordNet Library, http://sourceforge.net/projects/jwordnet [retrieved: November, 2012].

[12] WordNetScope, http://wnscope.sourceforge.net [retrieved: November, 2012].

[13] WordNetSQL, http://wnsql.sourceforge.net [retrieved: November, 2012].

[14] wordnet2sql, http://www.semantilog.org/wn2sql.html [retrieved: November, 2012].

[15] Gray, J. and Reuter, A.: Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1983.

[16] Brewer, E.: Towards Robust Distributed Systems. In: ACM Symposium on Principles of Distributed Computing, Keynote speech, 2000.

[17] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., and Fikes, A., Bigtable: A distributed storage system for structured data. In: the Seventh Symposium on Operating System Design and Implementation. Seattle, 2006.

[18] Edlich, S., Friedland, A., Hampe, J., and Brauer, B: NoSQL: Introduction to the World of non-relational Web 2.0 Databases (In German) NoSQL: Einstieg in die Welt nichrelationaler Web 2.0 Datenbanken. Hanser Verlag, 2010.

[19] Angles, R. and Gutierrez, C.: Survey of Graph Database Models. In: ACM Computing Surveys, Vol. 40. No. 1 Article 1, 2008.

[20] Cruz, I.F., Mendelzon, A.O., and Wood, P.T.: A graphical query language supporting recursion. In: Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data, pp. 323—330. ACM Press, 1987.

[21] Gemis, M. and Paredaens, J.: An object-oriented pattern matching language. In: Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software, pp. 339–355. Springer-Verlag, 1993.

[22] Giugno, R. and Shasha, D.: GraphGrep: A fast and universal method for querying graphs. In: Proceedings of the IEEE International Conference in Pattern recognition, 2002.

[23] InfoGrid: The Web Graph Database, http://infogrid.org/trac [retrieved: November, 2012].

[24] Apache Derby, http://db.apache.org/derby [retrieved: November, 2012].

[25] XING das professionelle Netzwerk, http://www.xing.com [retrieved: November, 2012].