A New Representation of WordNet® using Graph Databases On-Disk and In-Memory

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 extended paper, we present two new graph data models for the WordNet® dictionary. We use the emerging graph database management system Neo4j and deploy the models *on-disk* as well as *in-memory*. We analyze their performance and compare them to other traditional storage models based on native file systems and relational database management systems. 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; Neo4j; on-disk and in-memory DBMS; performance analysis.

I. Introduction

This paper is an extension of the work done in [1], whose aim is to provide new data representation models for WordNet® based on modern NoSQL graph databases. In this paper, we implement various data *storage* models for these representations varying from in-disk models, creating in-memory virtual disk representations and using pure in-memory models. It is worth mentioning that the size of the WordNet® dictionary enables the efficient employment of these variations and offers the best benchmarking platform for applications of this moderate size.

WordNet® [2] 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 with their strength in traversing and navigating through huge graphs.

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 graph databases. For this purpose, we present two data models and implement them on an emerging graph database management system: Neo4j [3]. Currently, Neo4j is the leading graph database management system in terms of installations and user base. WordNet® dictionary has several characteristics that promote our proposition: *it is used in several modern Web 2.0 applications*, such as social networks; *it 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 analyze 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 and data models are presented in Section III. In Section IV, we describe the storage models. Section V contains the results of our performance evaluation and Section VI 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 [2]. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

(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 meronym (a term which denotes part of something but which is used to refer to the whole of it), antonym (a word opposite in meaning to another), and holonym (a word that names the whole of which a given word is a part). The majority of the WordNet®'s relations connect words from the same part-of-speech (POS). Valid WordNet parts-of-speech include (noun="n", verb="v", adj="a", and adverb="r"). Currently, WordNet® comprises 117,000 synsets and 147,000 words. Today, WordNet® is considered the most important resource available to researchers in computational linguistics, text analysis, text retrieval and many related areas [4]. Several projects and associations are built around WordNet®.

The Global WordNet Association [5] 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 [6], 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 [7] 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 [8]. MultiWordNet [9], 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 [10], attempt to integrate WordNet with domain-specific knowledge.

RitaWN [11], 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 parts-of-speech (POS) such as 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 [12] 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 [13], WNSQL [14], and wordnet2sql® [15], 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 [16] are simply too restrictive. More relaxed models emerged such as the CAP (Consistency, Availability and Partition Tolerance) theory or eventually consistent [17], 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 [17]. 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 [18], dozens of NoSQL databases emerged. A good overview of existing NoSQL database management systems can be found in [19].

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 [20]. 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 [21], the object-oriented Pattern Matching Language (PaMaL) in the 90s [22], through Glide [23] 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

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

labels and wildcards. Glide uses a method called GraphGrep [23] 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 of 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 [3].

Several systems such as Neo4j [3], InfoGrid [24], 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 System And Data Model

Fig. 1 provides an overview of the proposed implementation. RitaWN [11] 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 [12] functionality for Java processing; which, in turn, accesses the native WordNet® dictionary.

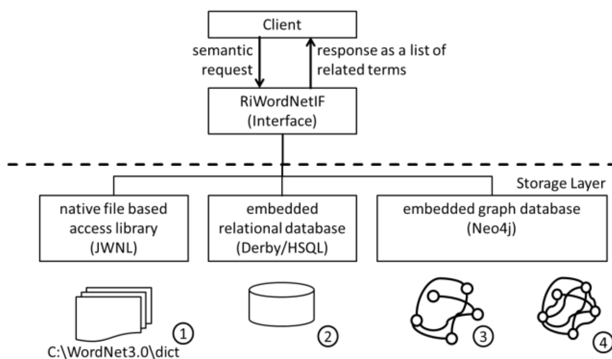

Figure 1. Architecture of the proposed system.

In order to separate the data representation model 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 in ascending complexity with respect to reaching the returned values:

- 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 illustrated in Fig. 2 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.

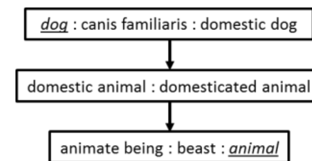

Figure 2. Semantic path from ‘dog’ to ‘animal’ .

A. Data Model

In the storage layer, illustrated in the lower part of Fig. 1, we provide four different representations for the WordNet® dictionary as described in the following subsections.

1) File-based Model

In its original implementation, RiTa.WordNet uses the JWNL [12] 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 Model

We use a database model similar to the one used in [15]. Fig. 3 illustrates a UML class diagram for the relevant

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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.

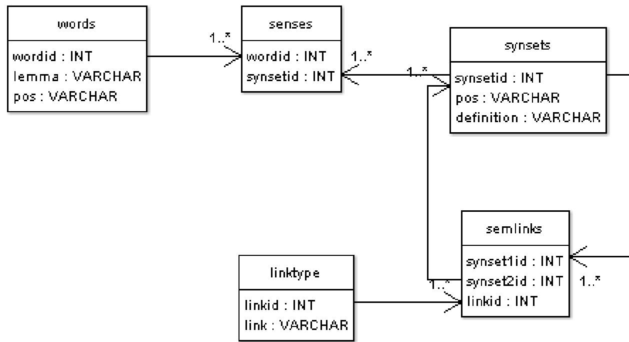

Figure 3. UML class diagram for the relational database.

3) Graph Database Model

In our proposed work, we model the WordNet® as a graph database. An object diagram is illustrated in Fig. 4. 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. 4, 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*.

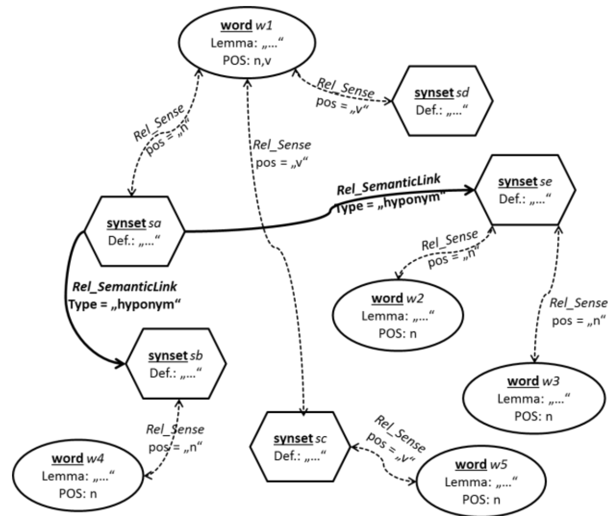

Figure 4. Object diagram for the proposed WordNet® graph database model.

4) Graph Database Storage with Additional Directly Derived Relationships

In the RiTa.WordNet application scenario, we expect many 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 the case of the limitation of the client application to inquiries within the above-mentioned four categories, the original relationships can be even dropped.

In terms of implementation, these relationships are identified through the relationship type. Fig. 5 illustrates the derived relationships for the example in Fig. 4. Only the relationship of type `Rel_Hyponym` for noun POS of word *w1*; namely, *w2*, *w3*, and *w4* is drawn. For more complex inquiries outside the categories "relationship tree" and "common parent", a combination of original and derived relationships are used in the traversal.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

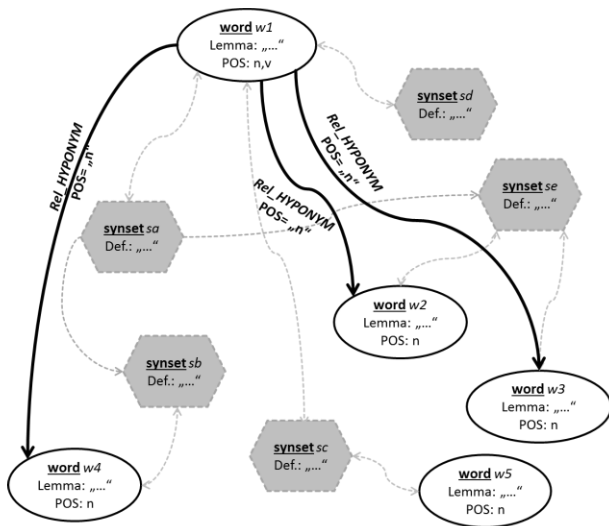

Figure 5. Object diagram with the extra derived relationships.

IV. STORAGE MODEL

We implement the graph data models using the currently leading graph database management system: Neo4j [3]. For all implementation models, we attempt to store the data *on-disk*. In addition to our work done and presented in [1], we also provide implementations stored *in-memory*.

A. On-Disk implementations

Using on-disk implementations is the traditional way for storing data. It preserves the content after system shutdown but suffers from the latency of hard disks.

In the file-based data model, WordNet® data is stored within the WordNet installation directory on disk. Native access is done through JWNL [12] library.

As for the relational database model, we choose Apache Derby [25] 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. We explicitly rule out the usage of larger relational database management systems running in server mode, such as Oracle or DB2, since we are concerned with the use case of relatively small-sized read-only interrelated data sets. Apache Derby is distributed as a Java jar file to be added to the classpath of the application. It also comes as a stand-alone version. In this case, the data resides in the database container on disk. We follow the common practices for standard relational database by building indices on the primary and foreign keys.

For the two data models we introduce in our research, we provide implementations for the emerging graph database management system Neo4j [3].

From its background and growing customer base, it is clear that Neo4j enjoys an increasing wide spread especially in the industry. Another advantage over InfoGrid [24] is its ease of use as it does not require the explicit definition of the model of the schema in XML as in the case of InfoGrid, which renders the addition of more entity types to the graph more simple. The basic setup for Neo4j is that the data is stored in a proprietary format on-disk. Neo4j then provides various data caching strategies in memory for so-called hot-spot data access.

B. In-Memory implementations

For the in-memory implementations, the whole WordNet® content is loaded in memory from the permanent storage during system startup. Having the content cached in memory avoids any access to the hard disk. The moderate size of the WordNet® data enables this setting.

We create a virtual disk out of RAM using RamDisk Plus [26], which uses a patented memory management component that makes a predefined portion of the RAM appear as a physical hard disk to the operating system and programs. The file-based data model of WordNet® is simply deployed on this virtual hard disk and the same JWNL [12] library is used to access the content.

In the case of the relational model, we experiment using two options:

- Similar to the file-based implementation, the Apache Derby database is stored in the virtual RAM Disk.
- We migrate the implementation to HSQL [27], which provides an in-memory transient storage mechanism for its tables. During startup, the content is loaded from the permanent storage into the in-memory tables created by the `CREATE MEMORY TABLE` SQL command.

Finally, for the two Neo4j data models, we also try the following two settings:

- Similar to the file-based and the relational implementations, we store both graph data models on the virtual RAM Disk.
- We set the cache management policy in Neo4j to `strong`. This cache setting holds on to all data that gets loaded to never release it. Additionally, Neo4j store can use memory mapped I/O for reading/writing. For optimized I/O access, Neo4j uses the `java.nio` package. Native I/O results in memory being allocated outside the normal Java heap so that memory usage needs to be taken into consideration. In order to get the best out of this setting, we increase the size of the cache used and the size of the memory mapped I/O to hold all the WordNet® data content.

V. 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 and HSQL, the graph database using Neo4j, and a second implementation using the materialized directly derived relationships also using Neo4j. For each one of the settings, we deploy the implementation twice: *on-disk* and *in-memory*.

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

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 if our proposed solution provides slightly better results than relational DBMS.* It is interesting to observe the effect of using in-memory and large caching settings for the different data model strategies on a moderately sized content like WordNet® as well.

We develop a simple performance evaluation toolkit around our 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 back-end for 5 minutes in order to eliminate any transient effects and measure the system performance after the 'warm-up' phase. The experiments are conducted for each type of inquiries separately.

In all our experiments, we monitor the system *response time* in terms of microseconds per operation from the moment of submitting the inquiry until 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. In order to build in-memory storage, we use RamDisk Plus [26].

We use JDK 1.6.0, Neo4j version 1.6 for the graph database engine, embedded Derby™ version 10.7.1.1 and HSQL version 2.3.0 for the SQL back-ends, JWNL library version 1.4 [12] 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 for both *on-disk* and *in-memory* settings.

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 research. The reason behind this is that the results are far worse than the other implementations. The difference in most cases is more than *one order of magnitude* as can be seen on the exemplary plot of Fig. 6 of the response time of one the experiments. We also drop plotting the results of HSQL implementation in-memory, since the deployment using the combination of Apache Derby and RamDisk Plus always supersedes the relational implementation of HSQL using its in-memory feature. In all legends of the subsequent figures, *NEO DD* means using Neo4j with the additional Directly Derived Relationships, *NEO noDD* means using Neo4j with the original relationships, and *SQL Derby* denotes the implementation using the SQL Apache Derby embedded relational database management system.

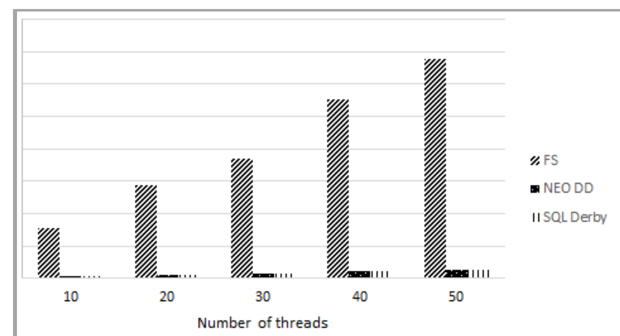

Figure 6. Average response time across increasing the number of threads with the File System (FS) included in the grpah.

1) Attribute inquiries

a) On-disk experiments

In this set of experiments, the inquiries sent by the workload generator comprise attribute inquiries only. Both response time, illustrated in Fig. 7, and throughput, illustrated in Fig. 8, 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 V.*E*).

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.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

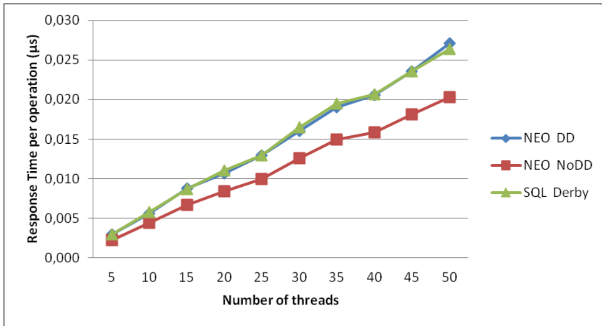

Figure 7. Response time for attribute inquiries (on-disk).

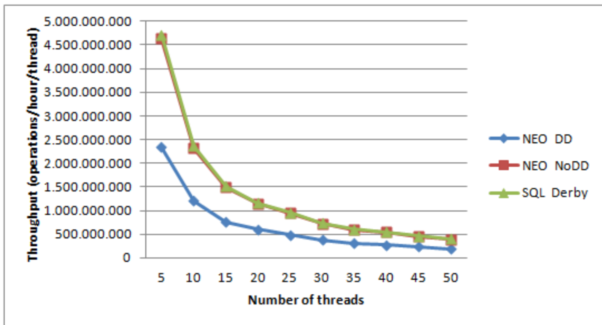

Figure 8. Throughput for attribute inquiries (on-disk).

b) In-Memory experiments

We repeat the same set of experiments using the RamDisk Plus settings explained in Section IV.*B*. The response time is plotted in Fig. 9 and the throughput for attribute inquiries in Fig. 10.

These figures indicate exactly the same behavior as their corresponding experiments in the on-disk Section. The relative decrease in response time and the relative increase in system throughput is explained separately and more elaborately in Section V.*D*.

From Fig. 9, it is clear that the response time of the simple Neo4j implementation is still the best by approx. 20%, while the throughput of the full-blown Neo4j has the worst values among the three implementations.

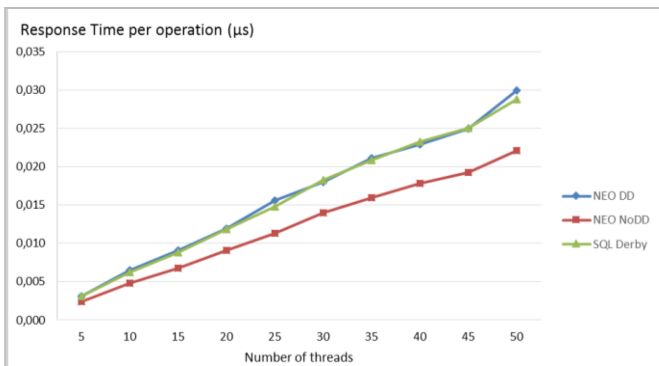

Figure 9. Response time for attribute inquiries (in-memory).

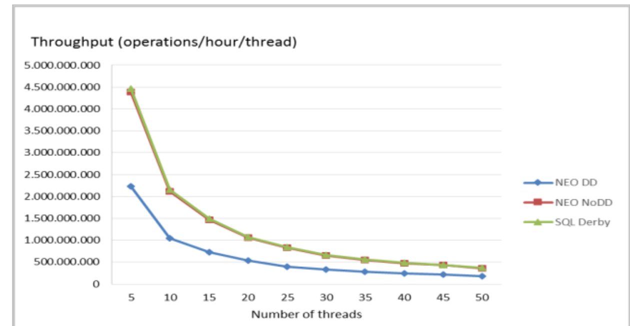

Figure 10. Throughput for attribute inquiries (in-memory).

2) Semantic relationship inquiries

a) On-disk experiments

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. 11, is enhanced by approx. 50% for all number of threads when compared to Apache 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. 12. 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.

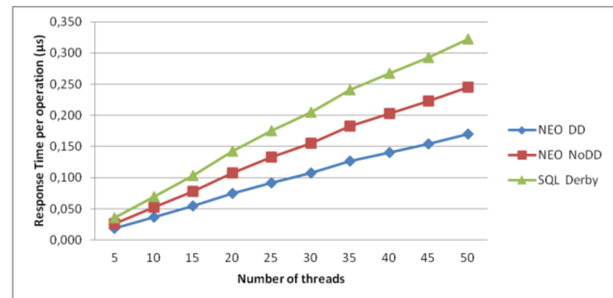

Figure 11. Response time for semantic relationship inquiries (on-disk).

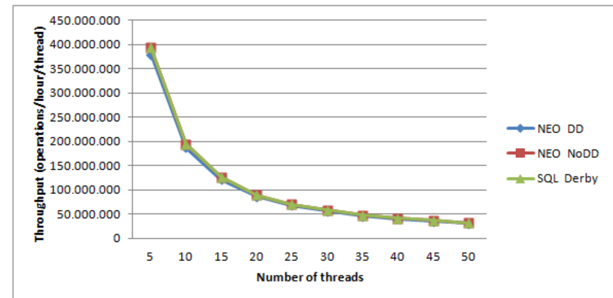

Figure 12. Throughput for semantic relationship inquiries (on-disk).

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

b) In-Memory experiments

The semantic relationship inquiries are repeated for the virtual disk settings. Here again, the same system behavior in terms of response time and througput is identical as the on-disk experiments. Fig. 13 illustrates the same response time pattern as in Fig. 11 and Fig. 14 illustrates that all three back-ends behave identically when it comes to throughput; which is the same scalability behavior as in the on-disk setting. The absolute values, as illustrated in Section V.*D* are almost the same as compared to Fig. 11 and Fig. 12.

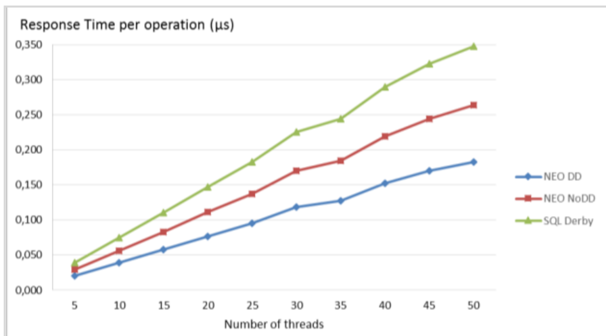

Figure 13. Response time for semantic relationship inquiries (in-memory).

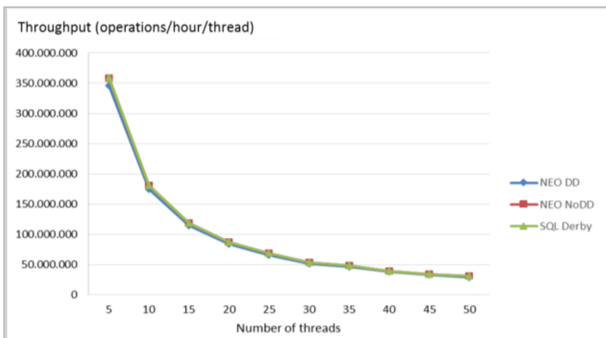

Figure 14. Throughput for semantic relationship inquiries (in-memory).

3) Relationship tree inquiries

a) On-disk experiments

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. 15 and Fig. 16, 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 of Neo4j is 25-30% better than the relational model. Here, again, the throughput, illustrated in Fig. 16, for all three implementations is the same. The equality of the throughput performance index of Apache Derby and the Neo4j implementations, 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.

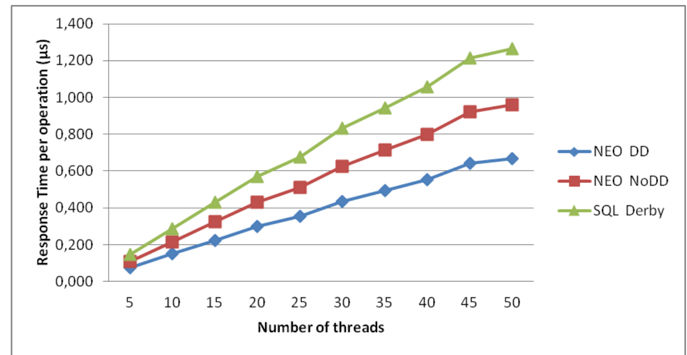

Figure 15. Response time for relationship tree inquiries (on-disk).

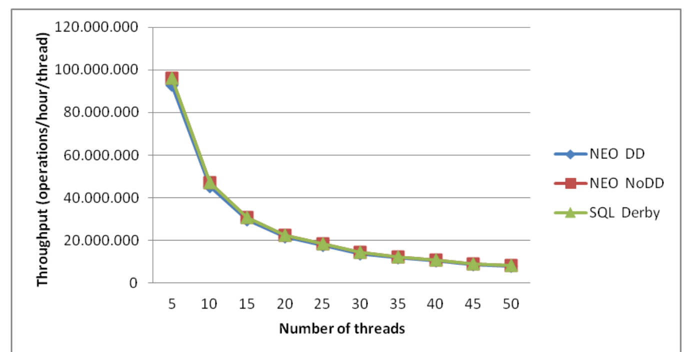

Figure 16. Throughput for relationship tree inquiries (on-disk).

b) In-Memory experiments

The same trend as the semantic relationship inquiries continues with the relationship tree inquiries when running in-memory.

The same drop in absolute values by a factor of 4 when compared to the semantic relationship inquiries is also reported here. As illustrated in Fig. 17, the response time of Neo4j with the directly derived relationships is half that's of the SQL implementation using Apache Derby.

The response time of Neo4j without the extra relationship remains in the middle of both curves. The Throughput illustrated in Fig. 18 for all implementations remains identical.

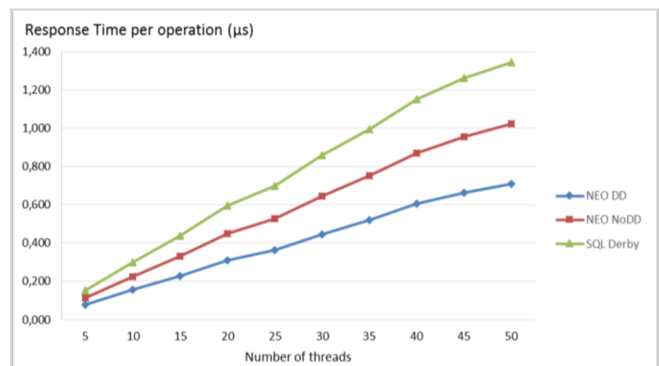

Figure 17. Response time for relationship tree inquiries (in-memory).

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

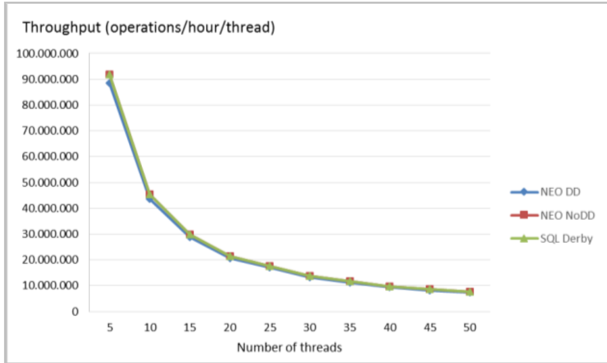

Figure 18. Throughput for relationship tree inquiries (in-memory).

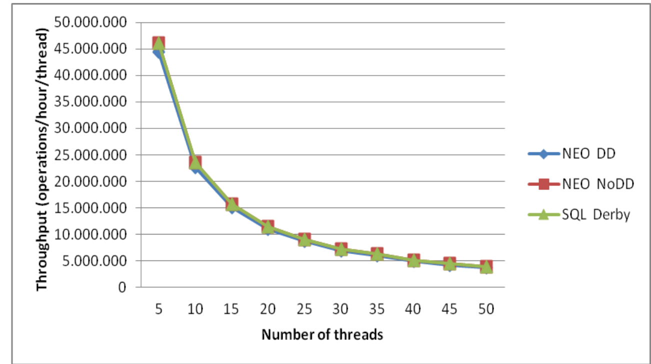

Figure 20. Throughput for common parent inquiries (on-disk).

4) Common parent inquiries

a) On-disk experiments

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 [28], 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 Apache 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. 19 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. 20, 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).

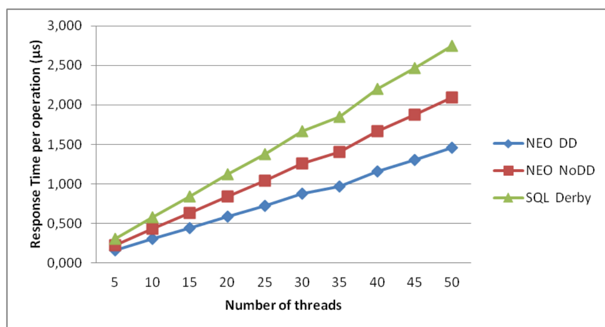

Figure 19. Response time for common parent inquiries (on-disk).

b) In-Memory experiments

Similar to all previous in-memory experiments, the common parent inquiries yield the exact same curves as their on-disk counterparts illustrated in Fig. 21 and Fig. 22.

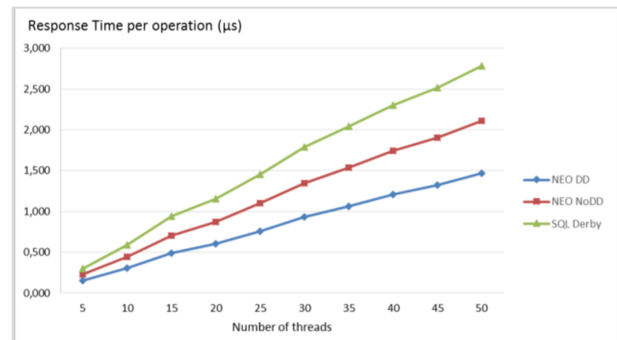

Figure 21. Response time for relationship tree inquiries (in-memory).

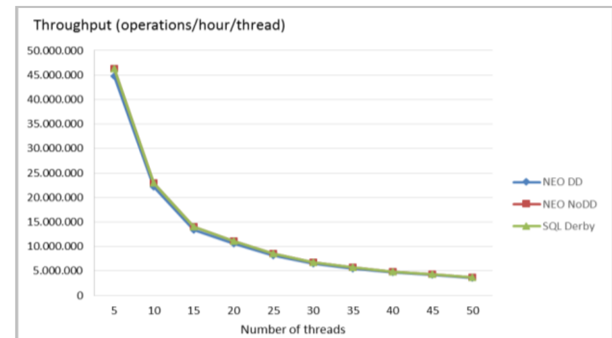

Figure 22. Throughput for relationship tree inquiries (in-memory).

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

D. *Comparison Between On-Disk and In-Memory Performance*

In this section, we compare the performance of the on-disk implementations versus their counterpart experiments done in-memory. The target is to evaluate the performance gain – if any- when keeping the whole content of WordNet® in memory. In Table I, we list the relative change in response time for each inquiry type. We define the average relative change in response time over all experiments to be:

$$relative\ change\ in\ response\ time = \frac{response\ time(in - memory) - response\ time(on - disk)}{response\ time\ (on - disk)}$$

TABLE I. CHANGES IN RESPONSE TIME HD VS. MEM

Inquiry type	FS	SQL Derby	NEO DD	NEO NoDD
Attribute	-11%	8%	11%	8%
Semantic relationships	7%	7%	7%	7%
Relationship trees	4%	5%	4%	5%
Common parent	4%	5%	4%	5%

Similarly, we list the relative change in throughput for each inquiry type in Table II. Analogously, we define the average relative change in throughput over all experiments to be:

$$relative\ change\ in\ throughput = \frac{throughput(in - memory) - throughput(on - disk)}{throughput(on - disk)}$$

TABLE II. CHANGES IN THROUGHPUT HD VS. MEM

Inquiry type	FS	SQL Derby	NEO DD	NEO NoDD
Attribute	13%	-8%	-10%	-8%
Semantic relationships	-6%	-6%	-6%	-6%
Relationship trees	-4%	5%	-5%	5%
Common parent	-4%	-4%	-4%	-4%

Remarkably, the performance does not increase substantially. In several experiments, the performance indices even slightly degrade. In all cases, the increase/decrease in performance remains within the ±10% range. This is attributed to the relatively *small size* of the WordNet® content as will be seen in the coming Section. The normal caching mechanisms provided by Apache Derby and Neo4j result in loading the whole content in-memory and renders the usage of the virtual RAM disk and all further memory optimization settings *needless*.

E. *Storage Requirements*

Performance in terms of good response time comes with its price. Fig. 23 illustrates the storage requirements for all four implementations. The Apache 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. As the in-memory experiments also show, there is no need to implement extra virtual disks or extravagant caching settings, since the size of the largest implementation fits easily in the heap of any Java virtual machine of moderate size.

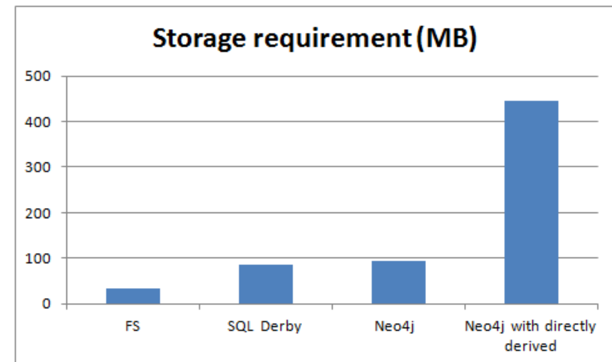

Figure 23. Storage for each back-end implementation.

VI. CONCLUSION AND FUTURE WORK

In this paper, we present two Neo4j graph data models 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 materialized 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. We also prove that there is no need for extra measures to hold the moderate size WordNet® content in memory by using in-memory databases, creating virtual RAM disks, or substantially increasing the caching mechanisms. In all our experiments, the on-disk deployments yield almost the same performance as the in-memory settings. On the long run, i.e., after having the Neo4j warm-started, almost all of the dataset is cached in memory by the underlying graph database management system. *The reason is that the WordNet® database fits in the heap of the normal Java virtual machine even with the materialization on the redundant relationships.* This adds to the advantages of using the graph databases in such moderate-sized scenarios, since the benchmarks demonstrate that there is no real need to spend extra effort in tweaking the memory usage.

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

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 to benchmark other graph database providers, such as InfoGrid [24]. We also plan to migrate 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. Another extension area is the comparison against other types of NoSQL such as XML databases, document stores or column-family systems.

REFERENCES

[1] K. Nagi, "A New Representation of aWordNet® using Graph Databases," 5th International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDA, Seville, 2013.

[2] C. Fellbaum, "WordNet and wordnets," in Encyclopedia of Language and Linguistics, Second Edition, Brown, Keith et al., Eds. Elsevier, Oxford, 2005, pp. 665—670.

[3] Neo4j. The World's Leading Graph Database, http://www.neo4j.org [retrieved: December, 2013].

[4] E. Voorhees, "Using WordNet for Text Retrieval," In WordNet An Electronic Lexical Database, C. Fellbaum, Ed., 0-262-06197-X. MIT Press, 1998.

[5] The Global WordNet Association, http://www.globalwordnet.org [retrieved: December, 2013].

[6] Mimida: A mechanically generated Multilingual Semantic Network, http://gittens.nl/gittens/topics/SemanticNetworks.html [retrieved: December, 2013].

[7] P. Vossen, "EuroWordNet: a multilingual database for information retrieval," DELOS workshop on Cross-language Information Retrieval, Zürich, 1997.

[8] P. Vossen, W. Peters, and J. Gonzalo, "Towards a Universal Index of Meaning," ACL-99 Siglex workshop, Maryland, 1999.

[9] E. Pianta, L. Bentivogli, and C. Girardi, "MultiWordNet: developing an aligned multilingual database," 1st International Conference on Global WordNet, Mysore, India, 2002.

[10] L. Bentivogli, A. Bocco, and E. Pianta, "ArchiWordNet: Integrating WordNet with Domain-Specific Knowledge," 2nd Global WordNet Conference, Brno, Czech Republic, 2004, pp. 39—46.

[11] RiTa.WordNet: a WordNet library for Java/Processing, http://www.rednoise.org/rita/wordnet/documentation [retrieved: December, 2013].

[12] Java WordNet Library, http://sourceforge.net/projects/jwordnet [retrieved: December, 2013].

[13] WordNetScope, http://wnscope.sourceforge.net [retrieved: December, 2013].

[14] WordNetSQL, http://wnsql.sourceforge.net [retrieved: December, 2013].

[15] wordnet2sql, http://www.semantilog.org/wn2sql.html [retrieved: December, 2013].

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

[17] E. Brewer, "Towards Robust Distributed Systems," ACM Symposium on Principles of Distributed Computing, Keynote speech, 2000.

[18] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, and A. Fikes, "Bigtable: A distributed storage system for structured data," 7th Symposium on Operating System Design and Implementation. Seattle, 2006.

[19] S. Edlich, A. Friedland, J. Hampe, and B. Brauer, "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.

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

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

[22] M. Gemis, and J. Paredaens, "An object-oriented pattern matching language," 1st JSSST International Symposium on Object Technologies for Advanced Software. Springer-Verlag, 1993, pp. 339–355.

[23] R. Giugno, and D. Shasha, "GraphGrep: A fast and universal method for querying graphs," IEEE International Conference in Pattern recognition, 2002.

[24] InfoGrid: The Web Graph Database, http://infogrid.org/trac [retrieved: December, 2013].

[25] Apache Derby, http://db.apache.org/derby [retrieved: December, 2013].

[26] RamDisk Plus, http://www.raxco.com/home/ramdiskplus_workstation.aspx [retrieved: December, 2013].

[27] HyperSQL, http://hsqldb.org [retrieved: December, 2013].

[28] XING das professionelle Netzwerk, http://www.xing.com [retrieved: December, 2013].

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org