Comparing Small Graph Retrieval Performance for Ontology Concepts in Medical Texts

Daniel R. Schlegel, Jonathan P. Bona, and Peter L. Elkin

Department of Biomedical Informatics
University at Buffalo, Buffalo NY 14260, USA
{drschleg, jpbona, elkinp}@buffalo.edu

Abstract. Some terminologies and ontologies, such as SNOMED CT, allow for post–coordinated as well as pre-coordinated expressions. Post–coordinated expressions are, essentially, small segments of the terminology graphs. Compositional expressions add logical and linguistic relations to the standard technique of post-coordination. In indexing medical text, many instances of compositional expressions must be stored, and in performing retrieval on that index, entire compositional expressions and sub-parts of those expressions must be searched. The problem becomes a small graph query against a large collection of small graphs. This is further complicated by the need to also find sub-graphs from a collection of small graphs. In previous systems using compositional expressions, such as iNLP, the index was stored in a relational database. We compare retrieval characteristics of relational databases, triplestores, and general graph databases to determine which is most efficient for the task at hand.

1 Introduction

Some terminologies and ontologies, such as SNOMED CT, allow for concepts which are post-coordinated as well as those which are pre-coordinated. Post-coordinated concepts are, essentially, small segments of the terminology or ontology graph. Compositional expressions (CEs) extend the idea of post-coordination, adding logical and linguistic relations. The advantage of using CEs created from ontologies and terminologies is that multiple linguistic surface forms for the same concept are mapped to a single logical form (and hence, a graph structure). For example, the following three forms, all representing the idea of hypertension which is controlled, map to a logical form in which the SNOMED CT concept for "hypertension" is the first argument in a binary `hasModifier` relation with the SNOMED CT concept for "uncontrolled."

1. Uncontrolled hypertension
2. HT, uncontrolled
3. Uncontrolled hypertensive disorder

It's important to note that you cannot simply encode medical texts using SNOMED and get similar results without CEs. In fact, 41% of clinical problems require CEs in order to be represented properly [5].

Compositional expression graphs are quite small, generally including only a handful of nodes and relations. For information retrieval purposes, these nodes and relations are tagged with the document from which they are derived. We derive CEs from text automatically using a natural language processing system. A single document may lead to the creation of hundreds of compositional expressions, so a large corpus results in a very large collection of small graph CEs.

In performing information retrieval, a system must search this very large collection of small graphs for specific instances of the query graph. The query graph may be a subgraph of the one stored in the database and still be a match.

Previous work has measured the performance of graph databases on various graph operations [4, 3], and traversal/retrieval of subgraphs of large graphs [13, 14, 22], but no work has been done on the problem we encounter here (retrieving a small graph from a collection of such graphs).

In Section 2 we discuss the three types of database systems used in this study and the sample queries. We also discuss the CE representations used, and the formats of the queries for each database system. Our evaluation methodology is presented in Section 3, followed by Section 4, which presents the results of running each of the queries on each of the database systems. Finally we conclude with Section 5.

2 Methods

Three different types of database systems will be used in this study. Representing traditional relational databases we will use Microsoft SQL Server 2014 [8] and Oracle 11g R2 [12]. A general graph database (Neo4j 2.2.3 [10]) will be evaluated, along with an RDF triplestore (Ontotext's GraphDB 6.4 SE [11]). RDF triplestores are also a kind of graph database, but tailored to a specific representation, namely the use of RDF triples.

The information output of the NLP process which creates the CEs can be represented as instances of fifteen logical relations, four unary, and the rest binary (see Table 1). Some of these relations are adapted from SNOMED CT itself [18], while others are standard logical relations, and still others have been created in-house. The method by which these relations are represented as a graph is presented in the respective database subsection.

Three queries have been selected for evaluation (see Figure 1). These queries were selected because of the prevalence of results in our dataset, the likelihood that these are real representative queries, and the ability of the queries to test various characteristics of the database systems. Queries one and two both have a single relation, but whereas nearly all CEs which contain the two codes in query one are answers to that query, this is the case for only about half of those for query two. Query three is more complex in that it uses multiple relations conjoined. Queries two and three also have many more results than query one.

Relation	Argument 1	Argument 2	Semantics
`and`	and_1	and_2	and_1 and and_2 are True.
`or`	or_1	or_2	Either or_1 or or_2 are True.
`not`	*not*	–	*not* is False.
`non`	*non*	–	*non* is False.
`possible`	*possible*	–	*possible* is Possible.
`exception`	*exception*	–	*exception* is an exception.
`exceptionTo`	*clause*	*exception*	*exception* is an exception to *clause*.
`accompaniedBy`	*clause*	*with*	*clause* is accompanied by *with*.
`hasModifier`	*modified*	*modifier*	*modified* is modified by *modifier*.
`hasQualifier`	*qualified*	*qualifier*	*qualified* is qualified by *qualifier*.
`hasLaterality`	*concept*	*laterality*	*concept* has laterality *laterality*.
`hasSpecimen`	*measurement*	*specimen*	*measurement* has specimen type *specimen*
`hasFindingSite`	*condition*	*bodysite*	*condition* has finding site *bodysite*
`hasProcedureSite`	*procedure*	*bodysite*	*procedure* has finding site *bodysite*
`hasScaleType`	*test*	*scale*	*test* has scale type *scale*

Table 1. Relations used in CEs. The relation name is given, along with labels for one or both arguments (dependent on whether the relation is unary or binary). The semantics of each relation are provided in the last column.

In the following three subsections we discuss details of the CE representations used for each of the respective database systems, and the nature of the queries to be performed using each of those systems.

2.1 Relational Database

The relational database format adopted for this study is that which was previously used in iNLP [7, 6, 9]. The table containing the CEs is made up of 10 columns, as seen in Table 2. Each row contains space for a relation and two operands (*relation*, *operand*1, and *operand*2, respectively), along with pedigree data about where the source of the relation is in the text (*docid*, *secid*, and *senid*). Each relation from Table 1 is mapped to a numeric identifier for use in the *relation* column. The column *ceid* gives an identifier unique to the CE within the document, and *cerowid* provides a reference to a row within a CE. There are then two helper columns, *optype*1 and *optype*2, which indicate the kind of thing in *operand*1 and *operand*2 respectively (either, a SNOMED CT code, a reference to a *cerowid*, or null).

Table 3 shows some of the rows for the CE with ID 13 in document 102153. In this snippet, there are three rows for SNOMED codes (*cerowid*s 1, 3, and 5), representing a finding of tobacco smoking behavior, pack years, and history, respectively. The "1" in column *optype*1 indicates that these are in fact SNOMED CT codes. The row with *cerowid* 3 connects the contents of *cerow*s 1 and 3 with relation 100, which is `hasModifier`. The 2s in *optype*1 and *optype*2 indicate that the contents of *operand*1 and *operand*2 are both *cerow* IDs. The row with $cerowid = 4$ connects the contents of *cerow*s 1 and 4 with relation 101,

1. Right cataract extraction

```
hasLaterality( < Cataract extraction (54885007) >,
               < Right side (24028007) >)
```

2. Controlled hypertension

```
hasModifier( < Hypertension (38341003) >,
             < Controlled (31509003) >)
```

3. Pack year smoking history

```
and(  hasModifier( < Finding of tobacco smoking behavior (365981007) >,
                   < Pack years (315609007) >),
     hasQualifier( < Finding of tobacco smoking behavior (365981007) >,
                   < History of (392521001) >))
```

Fig. 1. Queries used in evaluation of the databases.

Column Name	Description
docid	Document ID
secid	Record section ID
ceid	ID of the CE
senid	Sentence ID
cerowid	ID of the row within the CE
operand1	First operand of the relation
relation	The relation
operand2	Second operand of the relation
optype1	Type of thing in the operand1 cell
optype2	Type of thing in the operand2 cell

Table 2. Column descriptions of the relational database format.

docid	secid	ceid	senid	cerowid	operand1	relation	operand2	optype1	optype2
102153	51	13	1	1	365981007	null	null	1	0
102153	51	13	1	3	315609007	null	null	1	0
102153	51	13	1	2	1	100	3	2	2
102153	51	13	1	5	392521001	null	null	1	0
102153	51	13	1	4	1	101	5	2	2
102153	51	13	1	6	2	0	4	2	2

Table 3. Relational database representation of a match for query three: pack year smoking history.

which is `hasQualifier`. Finally, the `hasModifier` and `hasQualifier` relations are combined in the last row with the `and` relation (relation 0).

Performing a query using this representation is a two-step process. First, all CEs which contain the SNOMED CT codes for the concepts of interest are retrieved. This query is written as follows:

```
SELECT a.docid, a.ceid, cerowid, operand1,
       relation, operand2
FROM ce_codes as a
INNER JOIN (SELECT docid, ceid FROM ce_codes
            WHERE operand1 in (codes)
            GROUP BY docid, ceid
            HAVING count(distinct operand1) = codes.length) b
ON
  a.docid = b.docid
  AND a ceid = b.ceid
ORDER BY a.docid, a.ceid
```

where `codes` is the list of SNOMED CT codes of interest.[1] Then, each candidate CE structure must be verified against the query. We have written code to verify the structure of the retrieved CE in parallel with that of the user query.

As mentioned previously, the relational database systems used in this project are Microsoft SQL Server 2014 and Oracle 11g R2. These were chosen since they are large, enterprise-grade database systems, and previous experimentation by the iNLP group showed that queries of the type discussed here executed much more quickly on these than other database systems.

2.2 General Graph Database

There are many graph database solutions available, each with somewhat different capabilities [2]. We chose Neo4j because of its relatively recent rise in popularity, and benchmarks which show that it has both very good performance and scales well to large datasets. It's worth mentioning that Neo4j is also very good at some graph operations which may be useful in practice, such as path matching.

[1] Appropriate table indexes were created to speed execution as much as possible.

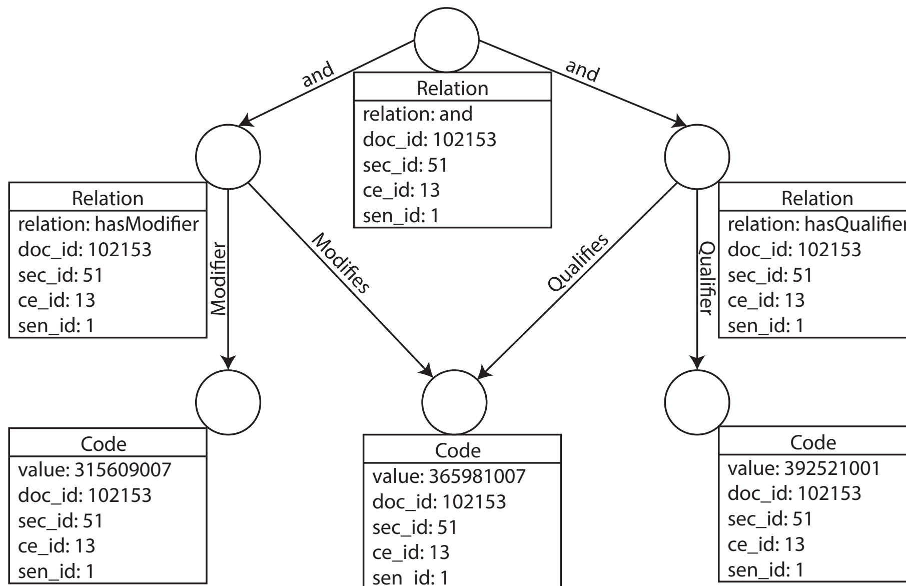

Fig. 2. Graph database representation of a match for query three: pack year smoking history.

In order to store CEs in Neo4j, we use a directed graph in which nodes are used to represent both SNOMED CT concepts and the relations previously discussed. Nodes are annotated with the pedigree information discussed for the relational database. Edges are labeled with the roles played by each argument of the logical relation (given in columns 2 and 3 of Table 1), and are directed from each relation to its arguments.[2] One major advantage of this graph representation is that we can easily represent a relation being an argument of another relation without leaving the graph formalism, or relying on any "hacks". We use properties for many node attributes (such as pedigree information) instead of relations only because Neo4j indexes on properties. We use two such indexes, on relation names and code values.

In Figure 2 the same example from Table 3 is presented using the graph formalism defined here. The three bottom nodes have the type `Code`, and represent the three SNOMED CT terms in the example. The four edges incident to those nodes indicate the roles played by those nodes in the relations represented by the sources of those edges. The nodes with type `Relation` represent relations, such as `hasModifier`.

[2] This is a simplified, impure, version of a propositional graph, as used in the SNePS family [17] of knowledge representation and reasoning systems, and for which a formal mapping is defined between logical expressions and the graph structure [16, 15].

Queries for Neo4j can be done in a single step — there is no need to verify the graph structure after the query is complete. The query for "controlled hypertension" (Query 2) is given below, where the relation node is given the label `rel`, which has two edges to nodes for codes (labeled `controlled` and `hypertension` for readability) – one with a `MODIFIER` label, and one with a `MODIFIES` label. The codes the edges point to are 38341003 and 31509003, respectively. These queries, using the Cypher query system, are created using a kind of ASCII art which is easy for users, and easy to generate programmatically.

```
MATCH (controlled:Code {value:38341003})
  <-[:MODIFIES]-(rel:Relation)-[:MODIFIER]->
    (hypertension:Code {value:31509003})
RETURN rel.docid
```

Neo4j also allows queries to be written entirely programmatically, using the Java API. This is a much more involved process. The user first accesses all nodes with a certain property value (*e.g.*, code value), and explores outward, first getting attached edges of a certain type, then nodes at the other end of those edges. This process continues as necessary, allowing the user to find and verify graph structures.

2.3 Triplestore

In addition to the general graph database system Neo4j, we have translated the compositional expression database to an OWL/RDF representation stored in a triplestore. An RDF (Resource Description Framework)[20][21] triplestore is a specialized graph database designed to store RDF triples: statements each consisting of a *subject*, a *predicate*, and an *object*. The Web Ontology Language (OWL)[19] is built on RDF and extends it to support richer modeling and logical inference. An OWL ontology consists of *Classes*, *Individuals* that are instances of the classes, and *Properties* used to express relations among them. *Literals* store literal data values.

Our OWL/RDF representation of the CE database uses just a few classes, mainly to separate CEs components that correspond to codes from those that represent relations. It contains many named individuals (one for each CE row / entry), and several properties, discussed more below.

For each CE element (uniquely identified by a *cerowid* in the relational model), we created a `NamedIndividual` and used OWL `DataProperties` to relate that individual to its ce id, document id, section id, etc. Relations between CE elements are encoded as OWL `ObjectProperties`. Each CE element that stands for a SNOMED code is related to that code using an `AnnotationProperty`.

OWL `DataProperties` are used to relate individuals to literals. We created a data property called *document id* to relate each CE element individual to its literal document identifier. We use similar data properties to relate each CE element to its section id, sentence id, etc.

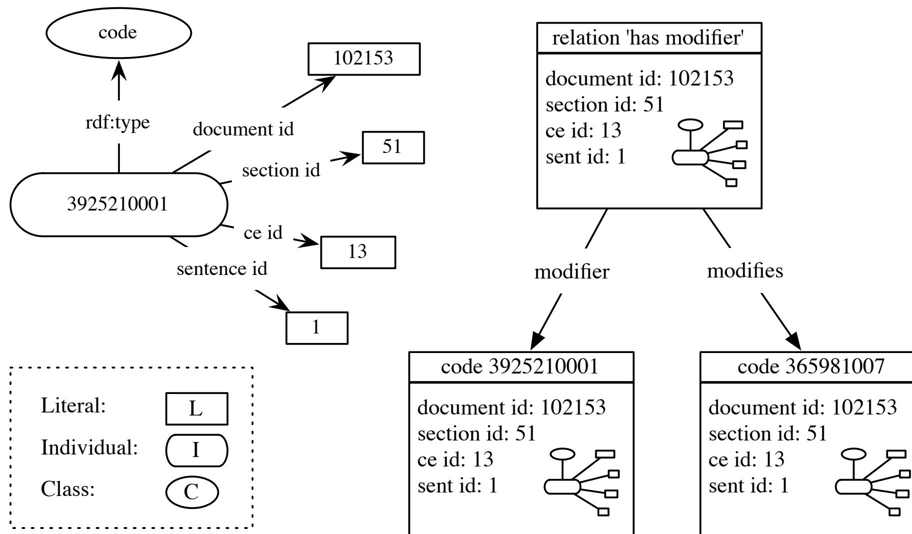

Fig. 3. OWL/RDF representation for compositional expressions

`AnnotationProperties` are used to provide annotations for resources. For instance, the annotation property *rdfs:label* is commonly used to associate an individual with a readable label. We use *rdfs:label* and an annotation property of our own, *has code*, which is used to label each CE individual that represents a code with the corresponding code.

`ObjectProperties` are used to relate individuals to each other. The fact that one CE element indicates the negation of another CE element can be represented by relating the two with an object property – in this case one we've labeled *not-rel*:

```
{ce1 not-rel ce2 }
```

CE binary relations are represented in the OWL/RDF version in a similar manner to the Neo4j propositional graph. Each relation has an individual that stands for the relation CE element, and two data properties are used to connect it to its relata. The triples that encode the *has modifier* relation between two code CE elements, `cee1` and `cee2`, where that relation is expressed by a third (`cee3`), are as follows:

```
{cee3 modifier cee1 }
{cee3 modifies cee2 }
```

Here both `modifier` and `modifies` are object properties.

Figure 3 shows the basic representation scheme. The left side depicts a small RDF graph for a single code CE element. The individual that represents the entry is labeled with the corresponding code as an annotation property. That individual is an instance of the class `code`, which is used to easily distinguish

entries for codes from those for relations (represented with the class `relation`). The edges from the individual to the rectangular boxes with literal values represent data properties that are used to connect this individual to the literal values of its document id, section id, etc.

The right side of Figure 3 shows three boxes, each of which stands for a single CE element subgraph like the one just described. An edge between two boxes corresponds to a data property asserted between the two individuals at the center of those subgraphs (*i.e.*, the individuals for those CE elements). These three boxes taken together with the data properties between them represent the fact that in the document with id `102153`, at the indicated section, sentence, and so on, the entry that corresponds to code `3925210001` modifies the CE element with code `365981007`. The other CE relations, including unary relations, other binary relations, and logical relations (*and*, *or*, etc.) are represented similarly using object properties between individuals.

Our queries for the CE data triplestore are in the SPARQL query language, which works by specifying triple patterns that are matched against the contents of the RDF graph to retrieve results. The following is a SPARQL version of the query for "controlled hypertension" (Query 2). The first few lines are used to map the long identifiers for properties to shorter names used in the query. An alternative would be to include the properties' label assertions among the triples in the query itself.

```
PREFIX hascode: <http://example.com/db-comparison#dbc0000455>
PREFIX modifier: <http://example.com/db-comparison#dbc0000438>
PREFIX modifies: <http://example.com/db-comparison#dbc0000437>
PREFIX docid: <http://example.com/db-comparison#dbc0000412>
select ?did
WHERE {
       ?rel modifies: ?c1 .
       ?c1 hascode: "38341003" .
       ?rel modifier: ?c2 .
       ?c2 hascode: "31509003" .
       ?rel docid: ?did .
}
```

3 Evaluation Methodology

Evaluation was performed by running each of the three queries on each of the six database configurations under study — Microsoft SQL Server 2014, Oracle 11g R2, Neo4j with the Cypher query language, Neo4j using the Java API, and the triplestore GraphDB. Each test was run 10 times, restarting the database systems in between tests to remove any possibility of cached results.[3] We did use a "warmup query" for each system, using a query unrelated to this study, to

[3] All tests were run on a laptop with a Core i7 4600U CPU, 16GB of RAm, and an SSD. Evaluation code was run in a VirtualBox VM running Ubuntu 14.04.

ensure that the database was fully loaded before running our queries. Averages of the 10 runs are reported in the next section. Significance of the differences between the systems was ascertained using the Student's t-test.

4 Results

We used a collection of 41,585 medical records for this study. The result of running these records through our natural language processing system was 840,184 compositional expressions. These compositional expressions were stored in each of the three types of database systems described above. Queries were written for each database system, designed to be fairly efficient. The relational databases and graph database contain slightly over 9 million rows and nodes, respectively. The triple store translation of the CE data consists of nearly 72 million triples. The average query times for each database on each query are presented in Table 4.

The average speed of each query using each database system and technique was compared with all others using a Student's t-test. We found that the differences between each system were extremely statistically significant for queries one and three with $p < .0001$. On query two, Neo4j's Cypher query engine was not significantly different from its Java API. Also on this query, Neo4j with Cypher was found to be faster than GraphDB with $p < .05$ and Neo4j with the Java API was faster with $p < .005$. All other differences were extremely significant, with $p < .0001$.

DB Type	DB Product	Query 1	Query 2	Query 3
Relational	SQL Server 2014	2434ms	2517ms	2471ms
	Oracle 11g R2	1904ms	2229ms	3037ms
Graph	Neo4j (Cypher)	287ms	272ms	729ms
	Neo4j (Java API)	105ms	263ms	128ms
Triplestore	Ontotext GraphDB	49ms	339ms	206ms

Table 4. Average execution time over 10 runs of each query on each of the database systems under consideration.

It appears that the query time for Microsoft SQL Server is fairly stable, regardless of the number of results or number of codes used in the query. This suggests that the query time is largely dependent on the size of the database, and not on the number of query results. Oracle, on the other hand, seems more sensitive to the number of codes being queried (but, this is obviously a small sample). The complexity of the CE graph to be verified did not appear to play a big part − this is implemented as fairly efficient Java code and occupies an extremely small percentage of the processing time. In Table 5, the number of CEs returned by the query (column 2) is presented, along with the number of CEs which were verified (column 3), and the number of unique documents matched.

Query	Query CEs	Verified CEs	Unique Docs
1	13	12	10
2	107	58	54
3	107	76	74

Table 5. The number of CEs returned by each relational query, along with the number which were verified to be instances of the query graph, and the number of unique documents the CEs come from.

The time for execution of the queries using Neo4j's Cypher query engine do suggest that the query complexity (*i.e.*, the complexity of the graph being matched) has some effect on query time. The sample size is rather small here, so more investigation will be needed to determine the exact variables at fault for increased query time. What we can say here, though, is that queries of the complexity of those examined here execute much faster with Neo4j than with either relational database system.

We have also explored writing Neo4j queries programmatically using the Java API instead of using the Cypher query engine. As others have found (*e.g.*, [1]), this usually is even faster than using Cypher. The disadvantage to this approach is that there is a need to understand the data in great detail. Whereas Cypher is able to use cost analysis to determine which order to perform query segments, such facilities are not readily available from the Java API. It is possibly for this reason that query 2 is slower using the Java API than using Cypher.

As with Neo4j, the triplestore queries are fast enough that they are certainly acceptable for any imagined use of the system. Both Neo4j and GraphDB claim to scale to billions of entries, but we did not assess how well either would scale to handle much larger data sets for this application.

Both Neo4j and triplestores fare well in straightforward matching of subgraphs in the query comparison, but it's worth noting that these systems have potential advantages, not realized here, which are dependent on the specific application. Neo4j allows for queries involving paths in the graph, allowing some types of reasoning to be performed very quickly. It also allows for graph operations such as shortest-path. With triplestores, the availability of logical reasoners for OWL — including the powerful inference capabilities built into Ontotext's GraphDB — creates the potential for much more interesting queries than examined here.

As a very simple example, we might want to query for CEs that are about not just one particular SNOMED code, but that code and all of its sub-concepts. Both Neo4j and GraphDB have ways of doing this. In GraphDB, one could load SNOMED into the triple store, establishing connections between SNOMED concepts and the CEs that use them, and testing queries that require reasoning about relations between concepts and between concepts and CEs. Such functionality could also be achieved using Neo4j's ability to efficiently follow paths in a graph, or by using a reasoner external to the CE data store.

None of the database systems here have been configured to any great extent for the fastest query processing possible. On the relational databases we created appropriate indexes, but didn't evaluate options such as alternate views of the data or stored procedures. With Neo4j we didn't attempt to use any of the available toolkits which lie atop the Java API to do fast querying. GraphDB provides a variety of configuration parameters that can be adjusted to affect performance for different kinds and amounts of data. We did not utilize these options in our comparison. The triplestore version of the CE data currently uses string literals to store values that could be stored as numeric literals. A slight performance increase might be obtained by a version that uses numeric literals for this instead.

With all of these systems, it is unknown how query speed scales as query size/complexity increases. While returning more data from a query could affect the performance of queries in RDBs and neo4j, it could be the case that this has a slightly larger effect on triplestore because it would be required to match more triples in order to retrieve additional properties associated with a particular resource. In the absence of tests with such queries, we have no reason to suspect that this would significantly affect the running time.

5 Conclusion

Fourty-one percent of clinical problems require post-coordinated CEs to represent their knowledge. For information retrieval purposes, medical texts which have been coded using CEs must be stored in some sort of database. Traditionally, relational databases would be used for this task. We have found that graph databases — both general and RDF triplestores — outperformed relational databases by a factor of up to fifty over the queries. On average, the graph database Neo4j was 5.7x faster than relational databases when using the Cypher query engine, and 16.1x faster when using the Java API. Our chosen triplestore, GraphDB, averaged 20.6x faster than relational databases. These performance issues can make the difference between a practical and an impractical solution.

References

1. Andrš, J.: Metadata repository benchmark: PostgreSQL vs. Neo4j (2014), `http://mantatools.com/metadata-repository-benchmark-postgresql-vs-neo4j`
2. Angles, R.: A comparison of current graph database models. In: Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on. pp. 171–177. IEEE (2012)
3. Ciglan, M., Averbuch, A., Hluchy, L.: Benchmarking traversal operations over graph databases. In: Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on. pp. 186–189. IEEE (2012)
4. Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vañó, A., Gómez-Villamor, S., Martínez-Bazan, N., Larriba-Pey, J.L.: Survey of graph database performance on the hpc scalable graph analysis benchmark. In: Web-Age Information Management, pp. 37–48. Springer (2010)

5. Elkin, P.L., Brown, S.H., Husser, C.S., Bauer, B.A., Wahner-Roedler, D., Rosenbloom, S.T., Speroff, T.: Evaluation of the content coverage of snomed ct: ability of SNOMED clinical terms to represent clinical problem lists. In: Mayo Clinic Proceedings. vol. 81, pp. 741–748. Elsevier (2006)
6. Elkin, P.L., Froehling, D.A., Wahner-Roedler, D.L., Brown, S.H., Bailey, K.R.: Comparison of natural language processing biosurveillance methods for identifying influenza from encounter notes. Annals of Internal Medicine 156(1_Part_1), 11–18 (2012)
7. Elkin, P.L., Trusko, B.E., Koppel, R., Speroff, T., Mohrer, D., Sakji, S., Gurewitz, I., Tuttle, M., Brown, S.H.: Secondary use of clinical data. Stud Health Technol Inform 155, 14–29 (2010)
8. Microsoft: SQL server 2014. http://www.microsoft.com/en-us/server-cloud/products/sql-server/ (2015)
9. Murff, H.J., FitzHenry, F., Matheny, M.E., Gentry, N., Kotter, K.L., Crimin, K., Dittus, R.S., Rosen, A.K., Elkin, P.L., Brown, S.H., et al.: Automated identification of postoperative complications within an electronic medical record using natural language processing. Jama 306(8), 848–855 (2011)
10. Neo Technology, Inc.: Neo4j, the world's leading graph database. http://neo4j.com/ (2015)
11. Ontotext: Ontotext GraphDB. http://ontotext.com/products/ontotext-graphdb/ (2015)
12. Oracle: Database 11g R2. http://www.oracle.com/technetwork/database/index.html (2015)
13. Partner, J., Vukotic, A., Watt, N., Abedrabbo, T., Fox, D.: Neo4j in Action. Manning Publications Company (2014)
14. Rodriguez, M.: MySQL vs. Neo4j on a large-scale graph traversal (2011), https://dzone.com/articles/mysql-vs-neo4j-large-scale
15. Schlegel, D.R.: Concurrent Inference Graphs. Ph.D. thesis, State University of New York at Buffalo (2015)
16. Schlegel, D.R., Shapiro, S.C.: Visually interacting with a knowledge base using frames, logic, and propositional graphs. In: Croitoru, M., Rudolph, S., Wilson, N., Howse, J., Corby, O. (eds.) Graph Structures for Knowledge Representation and Reasoning, Lecture Notes in Artificial Intelligence 7205, pp. 188–207. Springer-Verlag, Berlin (2012)
17. Shapiro, S.C., Rapaport, W.J.: The SNePS family. Computers & Mathematics with Applications 23(2–5), 243–275 (January–March 1992)
18. The International Health Terminology Standards Development Organisation: SNOMED CT technical implementation guide (July 2014)
19. W3C OWL Working Group: Owl 2 web ontology language document overview (second edition) (2012), http://www.w3.org/TR/owl2-overview/
20. W3C RDF Working Group: Rdf 1.1 semantics (2014), http://www.w3.org/TR/rdf11-mt/
21. W3C RDF Working Group: Rdf schema 1.1 (2014), http://www.w3.org/TR/rdf-schema/
22. Zhao, F., Tung, A.K.: Large scale cohesive subgraphs discovery for social network visual analysis. Proceedings of the VLDB Endowment 6(2), 85–96 (2012)