Benchmarking Graph Database Backends—What Works Well with Wikidata?

Tibor Kovács, Gábor Simon, Gergely Mezei

Abstract: Knowledge bases often utilize graphs as logical model. RDF-based knowledge bases (KB) are prime examples, as RDF (Resource Description Framework) does use graph as logical model. Graph databases are an emerging breed of NoSQL-type DBMSs (Database Management System), offering graph as the logical model. Although there are specialized databases, the so-called triple stores, for storing RDF data, graph databases can also be promising candidates for storing knowledge. In this paper, we benchmark different graph database implementations loaded with Wikidata, a real-life, large-scale knowledge base. Graph databases come in all shapes and sizes, offer different APIs and graph models. Hence we used a measurement system, that can abstract away the API differences. For the modeling aspect, we made measurements with different graph encodings previously suggested in the literature, in order to observe the impact of the encoding aspect on the overall performance.

Keywords: Graph Database, Knowledge Base, Wikidata, Benchmark

Introduction

Representing knowledge as a graph seems to be a natural choice. People even without any specialized technical or natural science knowledge often organize concepts and relations between them as nodes connected by edges. Some knowledge representation techniques also embraced this abstraction: RDF [5] represents metadata as a graph. Even the concept of knowledge graph has been floating around in the recent years, without a clear definition [6]. We use this concept aligned with [7]: an RDF graph encoding a set of knowledge.

In the DBMS world, graph as a data model is used since the dawn of database systems. As the NoSQL movement gained traction and as problem spaces with large-scale highly interconnected schemas—such as network simulation and social networks demanded, a new family of NoSQL databases emerged, replacing the key-value and document concepts with graphs. The landscape of NoSQL graph databases is in flux even today, with various graph models [13] [4], without standardized APIs, and even without a clear definition of a native graph database [12]. We use the graph database concept as a DBMS offering some graph construct (property graph, RDF graph, etc.) as logical model.

While connecting the dots above, storing knowledge represented as a graph in a database specialized to store graphs also seems a natural choice. However, one has to choose a graph database implementation first, that in turn determines the graph model and the API. Another decisive aspect is the graph encoding method. The RDF model gives a straightforward encoding for basic knowledge structures, however, there are different encoding models for reification [8], i.e. statements about statements, which is extensively used in KBs with reference management, where every statement should be backed up by external sources.

In order to help with these decisions, we selected a few graph database implementations and loaded with the same real-life, large-scale dataset, then queried with the same set of queries randomly generated from predefined query patterns. We run different measurements with different reification strategies. From the timings of the query runs, we were able to construct the performance profile of each database—encoding strategy combination.

Our research aims to determine performance characteristics of utilizing graph databases in various problem spaces. For the field of KBs, in the early phase, we worked with an algorithm-generated graph. Our initial results [10] showed counter-intuitive performance trends where more selective queries run slower than queries with more unbound values. In [9] the authors also encountered similar phenomena with a real-life dataset. Hernández et al. evaluated databases from different families, i.e. relational, graph, triplestore, and used the publicly available and collaboratively edited knowledge base Wikidata [5] as dataset.

In this phase of our research, we also used Wikidata data, but we chose the databases exclusively from the family of NoSQL graph databases.

Experimental Setting

As we wanted to compare our results with the ones described in [9], we chose to use the same January 2016 dated JSON dump as they used in their survey. For the same reason, we measured the so-called atomic-lookup queries, introduced in [9]. The basic idea behind this method is that every reified statement has five parts: a subject, a predicate, an object, a qualifier and a qualifier value. If we fix a subset of them while the others are kept variables, we get one of the 32 possible query pattern we measured in this paper.

Even though [9] introduced 4+1 representation models, we did not examine all of them. In our research, we measured three types of encoding: (i) the property graph representation which represents the qualifiers as edge properties, (ii) the standard reification, and (iii) the n-ary relation models which introduce a new node per each statement. The qualifiers are connected to this statement node, and the parts of the reified statement are connected to this node as described in [9].

Next step was to select the database implementations. The Wikidata query service is built on a customized Blazegraph [1] database engine, so we decided to select its current stable version (2.1.4). This graph database is designed to work with large RDF datasets using the standardized SPARQL query language. We chose Neo4j [11] as it is currently the most popular graph database [2]—specifically version 3.3.3. It uses the property graph model to represent the graph dataset and defines an own declarative query language, called Cypher. Our last benchmarked DBMS was the JanusGraph [3] 0.2.0. We selected this TinkerPop-based system because it is quite popular [2] and it has an active community. This database implements almost all of its functionalities through the integration of other technologies, it uses BerkeleyDB as storage, Apache Tinkerpop as graph processing engine.

We investigated other DBMSs as well, such as Grakn, OrientDB and Graph API of Azure SQL Database, but we encountered some difficulties during the modeling and loading phase in case of these systems. The main cause of the problem was that these systems hardly support having multiple different values of the same property in a node or we did not find any available description how to load them into a database.

One of our first tasks was to ensure that the DBMSs have the same runtime environment. Otherwise, the deviations in the measuring circumstances may distort the overall query timings. We achieved this by using separate virtual machines with Azure Standard E4s v3 specification for every system, the same as in [10]. Besides the operating system and the concrete DBMS, we installed the Java and .NET Core runtime environments on all VMs.

We defined a general workflow for the measurements. The 0th step was to delete all data that remained after the previous run. In the 1st phase, we transformed the decompressed JSON data to the import format of the concrete DBMS's import tool. After the transformation, we loaded the newly created dataset into the database. Finally, our tool inserted the previously random selected variable bindings into the query templates, measured the execution times and collected the mean query times.

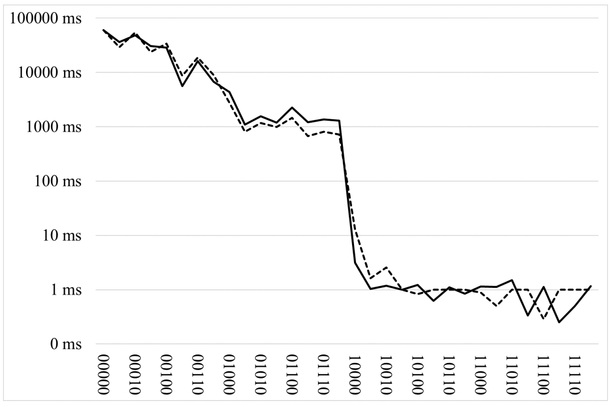
Results

We executed each 32 possible query patterns with ten different randomly selected variable bindings. To avoid first-time run transient phenomena, we ran every 320 query 2 times on every DBMS-encoding pair. We set a query time limit to 60 seconds, just like in [9] [10] to avoid it. Figure 1 shows the mean query time results in milliseconds on the logarithmical scale. Every timeout was set to 60 seconds.

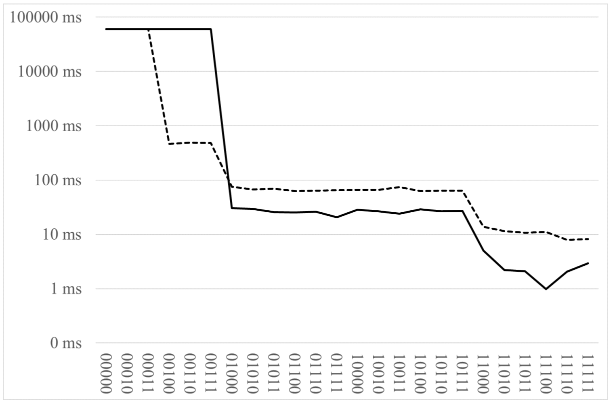
Despite its popularity, Neo4j was the slowest examined system. Every query must have been terminated before it finishes due to the time limit. Considering [9] and [10], it was expected that this system would have the slowest results, but the measured query times suggest that—despite the optimization done in the new version—this system could be several orders of magnitude times slower than its competitors on the same query, dataset and index structures.

The left diagrams of Figure 1 shows that if we used Blazegraph, there was no significant difference between the performance of the standard and the n-ary reification models. Looking at the figure, it is quite conspicuous that there is a significant gap between the times of queries with and without a variable subject. One can see that the execution times continuously decreased before and after this gap as well. This constant performance improvement can be the result of the declarative SPARQL language, whose execution can be optimized using the up-to-date DB statistics.

The other diagrams show the mean query results of the JanusGraph, and they show some similarities



Mean query times in Blazegraph using standard (solid) and n-ary (dashed) encodings.



Mean query times in JanusGraph using edge properties (solid) and n-ary (dashed) encodings.

Figure 1: Mean query times of the examined DBMS-encoding pairs.

and differences to the Blazegraph's results. The main similarity is that both system's results have steps on its diagram. The results show that in case of the JanusGraph this gap is located between the two query sets, one containing the queries without any node information binding, while every other query belongs to the other set. One can see that there is a second, smaller step at the 1 to 2 variable binding transition. The figure shows that the n-ary encoding is much faster on the first query patterns, but it is significantly slower on the later ones. Another interesting phenomenon is that the performance is almost constant between the steps, which can be explained by the imperative kind of the Gremlin query language.

Conclusions and Future Work

Regarding the mean query times, we concluded that the execution times depend heavily on both the query pattern and the system-encoding pair. The general tendency is that the less node variable a query has, the faster its execution is. The results show, even though the execution times slightly depend on the selected representation model, its impact is far less than the DBMS implementation used.

Based on the measured query times, it seems to us that the best overall performance for this kind of workload can be reached by using Blazegraph with either n-ary or standard encoding. Considering other factors than performance, our choice would be Blazegraph with n-ary representation, as this pair performed the smallest query times, while it required almost the least storage space.

We plan to involve other databases and reification techniques in our research, and replace the currently used atomic lookups to another query set that contains the most frequently used query patterns provided by the Wikidata statistics, to help the selection of the best DBMS-encoding pair for more real-life use cases.

Acknowledgments

This work was performed in the frame of FIEK_16-1-2016-0007 project, implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the FIEK_16 funding scheme. Cloud computing resources were provided by a Microsoft Azure for Research award.

References

[1] Blazegraph products. https://www.blazegraph.com/product/. Accessed: 2018-03-13.

[2] Db-engines ranking of graph dbms. https://db-engines.com/en/ranking/graph+dbms. Accessed: 2018-03-13.

[3] Janusgraph. http://janusgraph.org/. Accessed: 2018-03-13.

[4] R. Angles. A comparison of current graph database models. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, ICDEW '12, pages 171–177, Washington, DC, USA, 2012. IEEE Computer Society.

[5] W. W. W. Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

[6] L. Ehrlinger and W. Wöß. Towards a definition of knowledge graphs. In *SEMANTiCS*, 2016.

[7] M. Färber, F. Bartscherer, C. Menne, and A. Rettinger. Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago. *Semantic Web*, pages 1–53, 2016.

[8] D. Hernández, A. Hogan, and M. Krötzsch. Reifying RDF: what works well with Wikidata? In *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2015)*, volume 1457 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[9] D. Hernández, A. Hogan, C. Riveros, C. Rojas, and E. Zerega. Querying wikidata: Comparing sparql, relational and graph databases. In *International Semantic Web Conference*, pages 88–103. Springer, 2016.

[10] T. Kovács. Nagyméretű szemantikus adathalmazok tárolási megoldásainak teljesítményközpontú összehasonlítása. In *BME-VIK TDK*, 2017.

[11] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. OReilly Media, 2015.

[12] M. A. Rodriguez. A letter regarding native graph databases. https://www.datastax.com/dev/blog/a-letter-regarding-native-graph-databases, 2013.

[13] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.