Acta Cybernetica **24** (2019) 43–60.

Benchmarking Graph Database Backends — What Works Well with Wikidata?

Tibor Kovács, Gábor Simon, and Gergely Mezei

Abstract

Knowledge bases often utilize graphs as logical model. RDF-based knowledge bases (KB) are prime examples, as RDF (Resource Description Framework) uses graph as logical model. Graph databases are an emerging breed of NoSQL-type databases, offering graph operations to process and manipulate data. 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

1 Introduction

Representing knowledge as a graph seems to be a natural choice from several aspects. People even without any specialized technical or natural science knowledge often organize concepts and relations between the concepts as nodes connected by edges. Some knowledge representation techniques also embraced this abstraction: RDF [21] represents metadata as a graph. Even the concept of knowledge graph has been floating around in recent years, without a clear definition [23]. We use this concept aligned with [26]: an RDF graph encoding a set of knowledge. A set of standards and technologies are built around the RDF concept. The so-called triple stores [15] emerged, a form of storage engines optimized to store a massive amount of RDF-modelled data. SPARQL standard [27] was also introduced as a query language to roam the RDF graphs.

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

DOI: 10.14232/actacyb.24.1.2019.5

key-value and the document concepts with graphs. The landscape of NoSQL graph databases (GDBs) is in flux even today, with various graph models, e.g., property graphs, hypergraphs, RDF graphs [42, 19], without standardized APIs, and even without a clear definition of a native graph database [41]. In our research, we focused on GDBs offering property graph model through Apache Tinkerpop API [13], a widespread property graph framework.

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 [29], i.e., statements about statements. Reification 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 timing result of the query runs, we were able to construct the performance profile of each database—encoding strategy combination.

Our research aims to determine the 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 [34] showed counter-intuitive performance trends where more selective queries run slower than queries with more unbound values. In [30] the authors also encountered similar phenomena with a real-life 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.

In this paper, we review the most important results connected to the research area. In Section Related Work, we present other's work related to this paper: benchmarks using Wikidata in which graph databases are involved and possible modeling solutions to the problem of reification. In Section Background, we describe the relevant part of the previous phase of the research: we give a short description of the already existing measurement system and how it is used to measure the performance of different DBMSs. After that, we give a detailed description of the measurement process in Section Experimental Settings: we introduce the dataset we used in the measurements, define the unified workflow of the benchmarking process, introduce the investigated database implementations, reification models and query patterns, and present the physical infrastructure on which the benchmarks were executed. Then we describe and analyze the results we got from the measurements in the Results section. Finally, we summarize our work, make some conclusions based on the results and present some of our plans for further enhancements.

2 Related Work

As performance is a key factor in the field of databases, several benchmarks have been conducted on graph databases. These measurements usually differ in the dataset used, in the query workloads, and in the benchmarked systems. In [32] several GDBs were loaded with the same generated graph and evaluated using a workload of loading, primitive graph operations, and traversals. Social networking is one of the primary problem spaces for GDBs. In [20] Angles et al. generated a synthetic graph with similar characteristics as a real-life social network, then executed a workload typical to this problem space (common friends, path search, etc.) on selected graph databases, triple stores, and relational engines. They have found that graph databases are more scalable in compute intensive graph problems than the concurrents.

The Linked Data Benchmark Council (LDBC)[9] is an independent authority "responsible for specifying benchmarks [...] for software systems designed to manage graph and RDF data." LDBC is continuously widening its benchmark portfolio: it has a framework for graph analytic tasks (breadth-first search, page rank, etc.)[31], social networking [24] and linked data (RDF)[33]. In [38] the authors run the LDBC social network benchmark against graph databases, triple stores and relational engines. They have found that more mature systems with heavily optimized query execution pipelines have the advantage over the more innovative newcomers—regardless of the database model type.

Meanwhile, the Linked Data community is looking for efficient storage solutions for RDF data. The LDBC's Semantic Publishing Benchmark [33] offers a measurement specification for comparing the performance of RDF engines. Recently, Pan et al.[39] surveyed the contemporary RDF benchmarks and management solutions. Moreover, the authors run the benchmarks against distributed RDF systems. In the end, they could not announce a clear winner, the performance depended heavily on the type of the query workload.

One of the key aspects of the benchmark dataset, that whether is it synthetic or real-life. Although synthetic datasets are trying to mimic some characteristics of a real-life dataset, Duan et al.[22] pointed out that benchmark datasets are tend to differ significantly in performance impacting metrics. In [35] Morsey et al. proposed a benchmark dataset and workload based on a real-life knowledge base. They also concluded that measurement results of a real-world dataset can be substantially different from the results of a synthetic dataset.

In [29] Hernández et al. compared the performance of several triple store databases on the same reified KB dataset. Later, as a follow-up, also Hernández et al.[30] compared the performance of DBMS's with different data models. They evaluated databases from different families, including relational, graph, triplestore, and used the publicly available and collaboratively edited knowledge base Wikidata [21] as the dataset. Due to the diverse data models, they had to use various encoding strategies for different database implementations. In [34] we loaded several graph databases with the same generated reified dataset, i.e., with an abstract, artificial knowledge base. As a natural next step, in the current phase of our research,

we replaced the generated data with a real-life knowledge base.

3 Background

Modeling reification

The quasi-standardized way of reification was introduced in the early stages of the RDF specification [12]. It introduces a special vocabulary and a new node for every statement. The parts of the original statements are connected to this node with separate statements through to meta-predicates (rdf:subject, rdf:property, rdf:object) of the special vocabulary. Then, the meta-statements can use the intermediate node as the subject. We will be referring to this approach as standard reification. Standard reification is considered cumbersome and unnecessarily verbose. A somewhat leaner approach is proposed by implementing n-ary relations over the RDF model in [37]. Similarly, an intermediate node is introduced, connecting the object as well as other claim metadata to the subject. Hartig et al.[28] introduced an extension to the original RDF notation called RDF* by enabling using a whole statement as the subject, resulting a much shorter and clearer notation (Figure 2). Other reification modes like n-ary [25], singleton property [36] and named graph [29] were also proposed in the literature.

Graph databases usually offer more elaborate graph models than the basic RDF graph model. It seems promising that one can take advantage of these advanced constructs throughout the modeling of the reification. In [30] the authors mapped reified data to edge properties of the property graph model. At load time it worked, but typical queries involved edge properties had such a poor support, that they dropped this model. As a fallback, a form of standard reification model was implemented.

Previous work

In [34], we created an easy-to-extend system for benchmarking graph DBMSs that we enhanced in the next phase of the research. The framework can be structured into several layers: the data source layer, the 1st conversion layer, the intermediate representation layer, the 2nd conversion layer, and the concrete implementation layer.

The data source layer is only responsible for providing the dataset for the measurement system so that it can be any kind of information source, like a Wikidata JSON dump or the output of a generator tool. As the experimental dataset is quite large, the loading process can be done efficiently using the DBMSs' bulk loader tools. As every inspected importer tool has a different input format, we defined an intermediate format so in case of n different input type and m different DBMS, one had to implement only $n+m$ converters instead of $n*m$, which is one of the key factors in extensibility. The responsibility of the 1st conversion layer is to convert the dataset from the data source layer to the defined intermediate format while the 2nd conversion layer is responsible for converting the intermediate

format dataset to the format expected by the DBMS specific import tool. When the dataset is loaded to a system, it goes into the concrete implementation layer.

For this infrastructure, we defined a unified measurement workflow in [34] for every DBMS-model pair: (i) every data from previous measurement (if any) must be deleted (ii) the source data must be converted to the intermediate format (iii) the data in intermediate format must be converted to the concrete loader format (iv) the dataset must be loaded into the DBMS (v) the queries must be converted to the current language-reification model representation (vi) the queries must be executed, measured and the results must be collected.

4 Experimental Setting

Dataset

As we wanted to compare our results with the ones in [30], we used the same Wikidata JSON dump from January 2016. This dataset holds a massive knowledge base with 67 million statements. Wikidata highly encourages to back up the statements with references, hence reification is used extensively. As of December 2018, Wikidata holds 1.48 billion references for 654 million statements [18].

Workload

For the same comparability reasons, we similarly generated the so-called atomic-lookup queries as in [30]. This simple query generation technique is based on the atomic parts of a single reified statement: the three parts of the base statement, with the property and the object of the metastatement. We generate a query by for each statement part either fill it with a fixed value or define it as a variable to project. As a result, we get 32 different query patterns. One of these patterns is explained in Figure 1.

Reification models

Throughout our research, we examined three different reification techniques: (i) the property graph representation which encodes the qualifiers as edge properties, (ii) the standard reification, and (iii) the n-ary relation models which introduce a new node per each statement. Figure 2 depicts these models.

As opposed to [30], we did measurements using the property graph model since we wanted to compare the level of support between different graph database implementations. Edge properties look like a straightforward way to encode the reified claims. However, a reification claim referring to a resource would be an edge between an edge and a node resulting in an invalid graph model. One has to encode this kind of reference as an edge property with simple literal value referencing the identifier of the resource. Additionally, reification claims can reference multiple objects, e.g., a statement can be backed up by multiple sources. In that case, the edge property value would be a collection of identifiers.

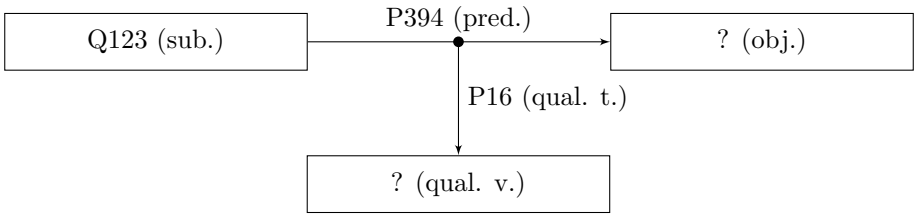

Figure 1: A reified statement has five parts: subject (sub.), predicate (pred.), object (obj.), qualifier type (qual. t.) and qualifier value (qual. v.), therefore the variability of this kind of statements can be described with five digit binary patterns, like 11010, where the first digit represents whether the subject part has a concrete value (1) or it is a variable (0), the second digit represents the same for the predicate part etc. in the previous exact order. This figure demonstrates the pattern 11010. The second and fourth numbers are 1, as they are bound to concrete values: the subject is a resource with id Q123, the predicate is P394, and the qualifier type is P16. The object and the qualifier value parts are variables. The example can be interpreted as follows: What values have the resource Q123 for property P394 and what values have these claims for property P16?

Database implementations

In the current phase of the research, we selected three systems for the following reasons:

(i) We have chosen Blazegraph [1] database engine as a measurement subject, as its customized version currently serves [17] as a backend for Wikidata Query Service (WDQS)[16]. The primary model of Blazegraph is RDF, while SPARQL endpoint is offered for query purposes. These features make Blazegraph closely related to triple stores. However, the RDF model can be viewed and queried as property graph through Blazegraph's Apache Tinkerpop implementation [2].

(ii) The open source GDB called Titan [14] was the first pick for the WDQS backend role, but was dropped eventually due to governance changes and the high risk of abandonment. Later, the source code of Titan was forked, and JanusGraph [7] was born. This database implements almost all of its functionalities through the integration of other technologies; it supports various storage options [8] (e.g., BerkeleyDB, Apache Cassandra, Apache HBase) while utilizing Apache Tinkerpop as property graph engine.

(iii) Neo4j [40] was chosen, as it is currently considered one of the most popular graph database [3]. It is based upon the property graph model and supports the Tinkerpop stack and its Gremlin query language. Besides Gremlin, it defines its own declarative query language, called Cypher.

We did not examine every implementation-reification model pair for our experiments. We did not apply the property graph model on Blazegraph, as its primary model is RDF, it would map the edge properties (only available through

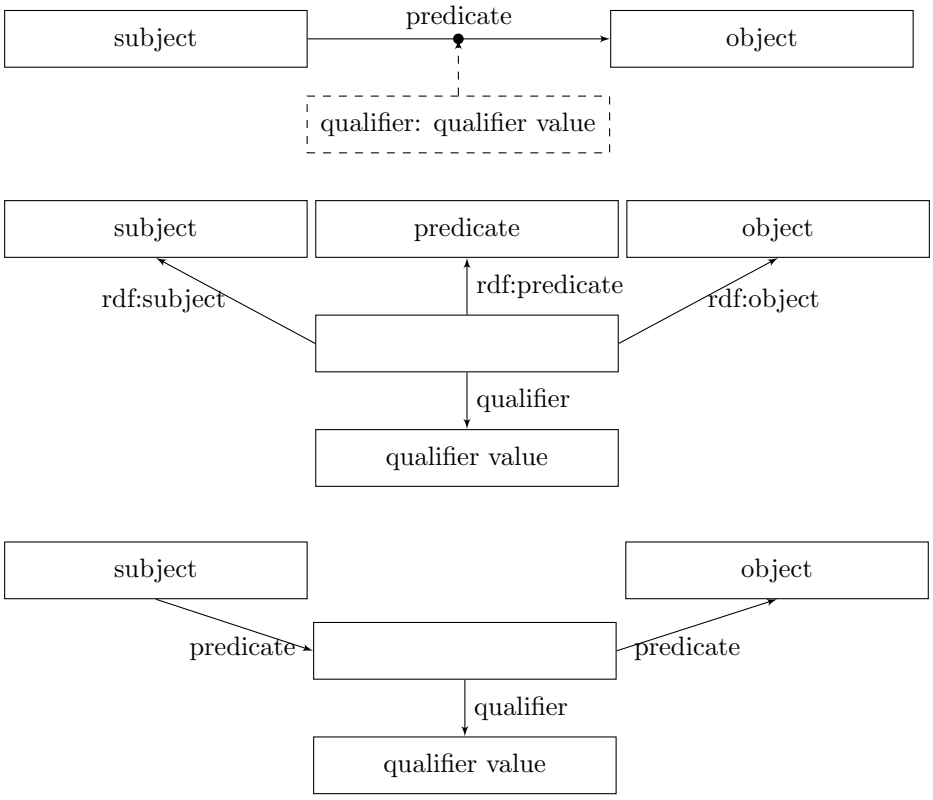

Figure 2: Visual representation of the different reification models. The first one is the property graph model, below that the standard model and finally the n-ary reification model.

the Tinkerpop interface) to RDF constructs. We did not utilize the RDF* notation support of Blazegraph. Moreover, the standard model was not measured on JanusGraph due to the extremely slow loading process. Table 1 summarizes the implementation-dependent measurement configurations.

We have investigated other GDBs as well, such as Grakn [4], OrientDB [10] and Gremlin API of Azure Cosmos DB [6], but we encountered a few difficulties during the modeling and loading phase. The primary cause of the problems was that these systems hardly support having multiple different values of the same property in a node or we did not find any available documentation on how to bulk load them into a database. OrientDB is a multi-model database; its earlier version was benchmarked as a GDB, e.g., in [32], but not with a real-life KB workload. Grakn and Cosmos DB's Gremlin are quite newcomers without any previous involvement in a significant benchmark effort in the academic literature.

Table 1: Overview of database configurations

Impl.	Version	Model	Reification modes	Query language
Neo4j	3.3.3	Prop. graph	Prop. graph Standard N-ary	Cypher
Blazegraph	2.1.4	RDF (primary) TinkerPop	Standard N-ary	SPARQL
JanusGraph	0.2.0	TinkerPop	Prop. graph N-ary	Gremlin

In order to ensure the correctness of the results, the dataset is converted to the natively supported graph format of a system during the conversion processes, for example, we introduced edge and node properties in case of Neo4j and JanusGraph, RDF triples in case of Blazegraph. This means that every database worked with its native graph format, so the source of the dataset (a knowledge graph) is basically irrelevant.

Installation environment

We provisioned separate virtual machine (VM) instances in the Azure public cloud for every GDB implementation. All VMs had a size of *Linux E4s v3*. They were configured with Intel XEON E5-2673 v4 processor containing 4 virtual CPU cores that support Intel Hyper-Threading Technology and 32 GiB of memory. A 64 GiB SSD was used as storage for the Ubuntu 16.04 LTS operating system and the particular DBMS. As the dataset had to be stored more times simultaneously—for example, during the conversions the source and the result dataset existed together at the same time—we added another SSD with 512 GiB capacity to store the dataset and the temporary files.

Besides the concrete DBMS implementation, we installed the Java and .NET Core runtime environments on all VMs. Even though the hosting environment is the same for the different DBMSs, the configuration of the systems can have a massive impact on their performance. As every investigated system is Java-based, we specified a uniform 20 GiB heap size for each system. On any other settings, we used their default configurations like [30] did.

Measurement workflow

In this benchmark, we basically used the same measurement workflow described previously. The initial step was to delete all data that remained after the previous run. In the first 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. This workflow is depicted in Figure 3.

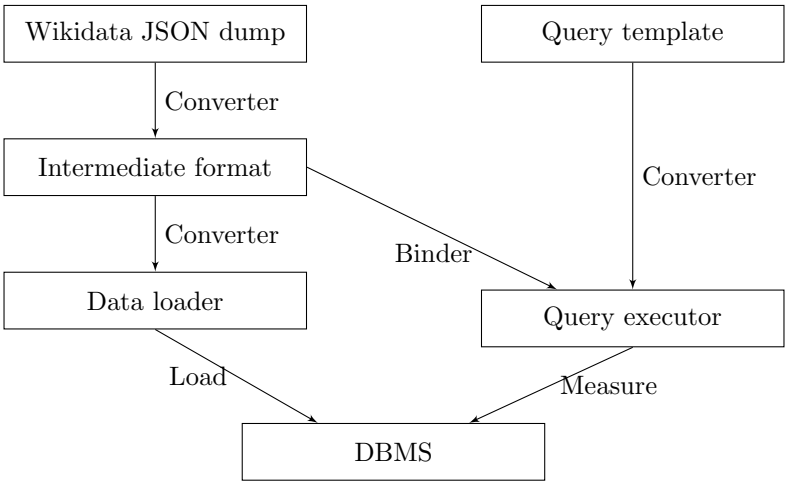

Figure 3: Overview of the measurement workflow

While in [34] we used a custom dataset generator to artificially create the dataset for our benchmarks, in this paper we replaced this component with a Wikidata JSON dump. For that reason, the 1st conversion layer had to be reimplemented as well, as the conversion between the data source and the intermediate representation is the responsibility of this layer.

In this paper, we present the measurements of a DBMS that we did not benchmark in [34], so we added a new component to the 2nd conversion layer that converts the dataset from intermediate representation to the format expected by the new DBMS's data loader tool. With the help of the intermediate representation layer, every other component could be reused in our new benchmark without nearly any modification.

Every query pattern (except the one without any variable) was run with ten different variable bindings. The nth query pattern (q_n) supplied with the mth variable binding for it ($b_{n,m}$) forms the runnable query $qb_{n,m}$. The values of the variables were randomly selected from the dataset in such a way that every query

would have a non-empty result set. To avoid first-time run transient phenomena, we ran all of the queries two times on every DBMS-encoding pair. In the beginning, we did not apply any time limit during the measurements, the first queries run by Neo4j were manually terminated after more than 15 minutes as the two runs would have taken more than a week continuous execution time based on our estimations. Afterwards, we set a query time limit to one minute, just like in [30] and in [34].

The workflow of the measurement phase consisted of the following steps (in this order): we applied the first bindings to the 32 query patterns $\{q_1, ...q_{32}\}$, then run the set runnable queries of $\{qb_{1,1}, qb_{2,1}, ...qb_{32,1}\}$, limiting each query separately to one minute. Then, we proceeded the same way with the remaining nine variable sets $\{qb_{1,2}, qb_{2,2}, ...qb_{32,2}, qb_{1,3}, ...qb_{1,10}, ...qb_{32,10}\}$. When the execution of all the runnable queries completed, the DBMSs were restarted to remove every memory content that could distort the results for the later runs. Finally, we made a second run by repeating the whole process with exactly the same pattern-binding combinations. The average response time values on the figures are calculated as the average of the twenty results for a given query pattern: the response times of the ten different variable bindings, i.e., the results from $\{qb_{i,1}, qb_{i,2}, ...qb_{i,10}\}$ for query pattern i—from the two separate runs. When a query had to be terminated because of the time limit, it was counted as 60 sec (actual time limit) in the average.

5 Results

Despite its popularity, Neo4j was the least performant system—in lots of cases by far compared to the other systems, as every query must have been terminated due to the time limit. We got these timeouts irrespective of the reification model. This experience is in line with [30] and [34].

We examined some of the query plans, and we found that the main reason for the poor performance is the lack of proper optimization. In some cases, we experienced that even though there were concrete nodes in the graph pattern to match, the pattern matching and the graph traversal started from variable nodes and the concrete information was used only in later phases. We investigated the usage of so-called planner hints [11]—adding explicit directions to the query optimizer into the query—but it resulted in performance improvements only in some cases. Furthermore, even the official documentation does not consider it as a general optimization technique [11].

The diagrams in Figure 4 show the results we got after measuring the performance of the Blazegraph. In contrast with Neo4j, most of the queries terminated before the time limit; only the most general patterns reached the one-minute timeout—the ones with only the qualifier part bound.

As can be seen in the diagram, there is no significant difference between the performance of the standard and the n-ary reification models. The results show that the two models do not just perform similarly, but they react almost the same way to the changes in the query patterns as it can be seen between 00100 and 00110 and in 01100.

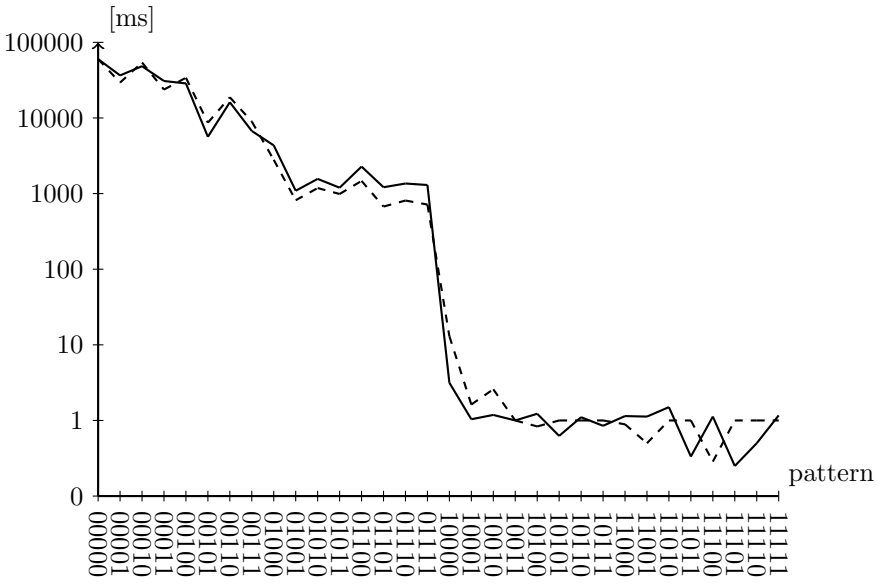

Figure 4: Blazegraph query mean response times for the standard (solid) and n-ary (dashed) models.

Even though the performance of the two models was quite similar, the n-ary model has an advantage against the standard model: it requires one less node to represent a statement, so on more massive datasets (like Wikidata), it requires significantly less storage space than the standard model.

Looking at the figure, it is quite conspicuous that there is a significant break in the middle of the graph. We have found that—in case of using Blazegraph—the most important factor that affects the elapsed time of an atomic-lookup is whether the subject (the starting point of the traversal) is concrete or not. Concretizing the subject means a significant performance boost, which suggests that Blazegraph pattern matching engine can perform reasonably well only if the graph traversal and the edge are pointing to the same direction.

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 nature of the SPARQL language, whose execution can be optimized using the up-to-date DB statistics in the more and more concrete queries.

Figure 5 shows the mean query response times of JanusGraph. One can see that there are fewer patterns in the diagrams, the ones ending with 01 are missing. That is because we encountered some difficulties in translating these queries into Gremlin queries. We tried using the proper Gremlin *Has* step variant in a couple of ways to filter the traversal according to the *01 patterns, but all of these queries

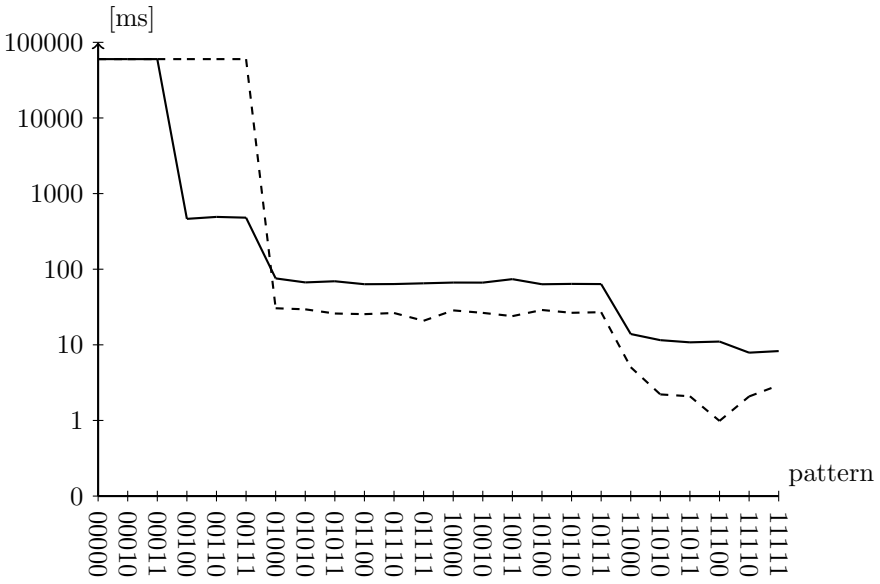

Figure 5: JanusGraph query mean response times for the n-ary (solid) and property graph (dashed) models.

returned with an empty result set. As we could not figure out why did this happen, eventually we removed these patterns from the measurement of JanusGraph.

This system also performs much better than Neo4j in most cases, but its query characteristic for atomic-lookups shows some similarities but also some fundamental differences with Blazegraph's. In case of both of these systems, the queries had to be terminated because of reaching the timeout only in the most general query patterns. While the n-ary model on JanusGraph timeouts on the patterns containing only qualifier information—like Blazegraph—, the queries with property graph model terminated before the time limit only when the query presented any type of concrete node information.

One can see that both systems have notable performance steps, but it emerges in a much more visible way in the case of JanusGraph. The n-ary model has a significant step at 00100 and two more small steps at 01000 and 11000. From that result, we concluded that the key factors that determine the performance of JanusGraph and n-ary model are the existence of concrete nodes but in contrast with Blazegraph, binding a concrete value to the predicate can result in much shorter response times. This DBMS-model pair can be ideal for queries containing edge information only, where it outperformed any other investigated system-model pairs.

The property graph model has a similar query characteristic to Blazegraph, as

it has its large performance step, where any kind of node information is presented in the query. In these cases, even though it performs significantly better than the n-ary model, it is much slower than Blazegraph. As the Blazergaph queries performed better for every query pattern than JanusGraph with property edge, we do not recommend using this pair in a real-life application.

Another interesting phenomenon is that the performance is almost constant between the steps on both models. This can be explained by the imperative kind of the Gremlin query language, as it gives a relatively small space to the optimizer to improve the query plans.

Figure 6: Comparing the results of the three investigated DBMSs in case of n-ary model: Neo4j (dotted), Blazegraph (dashed), JanusGraph (solid).

Based on the comparison of three measured systems on the only common reification model (Figure 6), one can come to the conclusion that Blazegraph offer the lowest response times if the subject part of the query is specified, otherwise JanusGraph outperforms all its competitors. Furthermore, neither of the systems could efficiently answer the questions that contain only qualifier information.

6 Conclusions and Future Work

In our work, we examined the performance of several graph database implementation — reification model pairs. We used a real-life knowledge base as dataset and simple atomic lookup queries as workload. Although the graph models offered by

GDBs seem rather suitable for knowledge graphs at first, one can hit quite a few limitations with datasets utilizing reification heavily. The direct, straightforward encoding of reified claims often resulted in a subpar performance as they relied on unoptimized features. In that manner, databases specialized to store knowledge models (e.g., RDF-based stores) and with the dedicated support of modeling reification clearly have advantage over general-purpose GDBs.

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. Event though as the results show, the execution times slightly depend on the selected representation model, its impact is far less than the DBMS implementation used.

Based on the overall average query times measured, the best 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 representation has lower storage footprint.

As every benchmark, our work has its limitations. In the current phase, we had to apply several constraints, for example on the workflow, on the measured configurations or on the used query languages. Currently, we are working on relaxing these constraints. Our measurements were limited to atomic-lookup queries, but in the future, we plan to investigate the performance of the DBMS's on a more real-life workload. To achieve this, we are analyzing the most frequently used query patterns provided by the Wikidata query service. Once we get these statistics, we can compose a query set that can simulate nearly real-life questions. Using these queries, we can measure the performance of the implementation-model pairs on a realistic load, which will give a better view on when and how to use these systems.

In the current phase of our work, we analyzed the query execution results and some of the obtained query plans to find out why does a query run slowly while others are fast. In the future, we are planning to make some deeper analysis, even at the implementation level, as most of these systems are open-sources.

As we introduced the use of an intermediate representation in the conversion phase, the measurement system can be extended effortlessly. Thus, we are planning to involve other databases, like Grakn [4], Azure CosmosDB [6] or HypergraphDB [5]. We are also planning the investigate other (even system specific) reification techniques such as the general singleton property [29] approach or the RDF* mode of Blazegraph [28].

Furthermore, we are planning to introduce the query language as a new dimension in the future. Until now, we investigated only the "native" language of a DBMS, even though they usually support more than one, for example, Blazegraph supports Gremlin. We are planning to measure the same database with different languages (if possible) to determine how much influence does it have on the performance. This may answer a couple of open questions, like whether the declarativity of a language or the language itself has any impact on the performance of these systems.

Acknowledgments

The project was funded by the European Union, cofinanced by the European Social Fund (EFOP-3.6.2-16-2017-00013). Cloud computing resources were provided by a Microsoft Azure for Research award.

References

[1] Blazegraph products. https://web.archive.org/web/20171125161035/https://www.blazegraph.com/product/. Accessed: 2018-03-13.

[2] Blazegraph TinkerPop implementation. https://web.archive.org/web/20180611150556/https://github.com/blazegraph/tinkerpop3. Accessed: 2018-09-09.

[3] DB-Engines ranking of graph DBMS. https://web.archive.org/web/20180911002043/https://db-engines.com/en/ranking/graph+dbms. Accessed: 2018-03-13.

[4] Grakn.AI - the knowledge graph. https://web.archive.org/web20180918085112/http://www.grakn.ai/grakn-core. Accessed: 2018-08-02.

[5] HypergraphDB - a graph database. https://web.archive.org/web/20180809121925/http://hypergraphdb.org/. Accessed: 2018-08-02.

[6] Introduction to Azure Cosmos DB: Graph API. https://web.archive.org/web/20180911002034/https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction. Accessed: 2018-08-02.

[7] JanusGraph. https://web.archive.org/web/20180919165133/http://janusgraph.org/. Accessed: 2018-03-13.

[8] JanusGraph storage backends. https://web.archive.org/web/20180209145536/http://docs.janusgraph.org:80/latest/storage-backends.html. Accessed: 2018-03-13.

[9] Linked Data Benchmark Council. https://web.archive.org/web/20181228154821/http://ldbcouncil.org/. Accessed: 2018-08-06.

[10] OrientDB. https://web.archive.org/web/20181016165245/https://orientdb.com. Accessed: 2018-12-07.

[11] Planner hints and the USING keyword. https://web.archive.org/web/20180206081105/http://neo4j.com:80/docs/developer-manual/current/cypher/query-tuning/using/. Accessed: 2018-08-03.

[12] RDF schema 1.1 - reification vocabulary. https://web.archive.org/web/20180920184035/https://www.w3.org/TR/rdf-schema/. Accessed: 2018-08-06.

[13] TinkerPop3 documentation. https://web.archive.org/web/20180923130832/http://tinkerpop.apache.org/docs/3.3.3/. Accessed: 2018-08-06.

[14] Titan Graph Database. https://web.archive.org/web/20180910214447/http://titan.thinkaurelius.com/. Accessed: 2018-09-09.

[15] What is RDF triplestore? https://web.archive.org/web/20170506152814/http://ontotext.com:80/knowledgehub/fundamentals/what-is-rdf-triplestore/. Accessed: 2018-09-11.

[16] Wikidata Query Service. https://query.wikidata.org/. Accessed: 2018-09-09.

[17] Wikidata Query Service - user manual. https://web.archive.org/web/20180917181601/https://www.mediawiki.org/wiki/Wikidata_Query_Service/User_Manual. Accessed: 2018-09-09.

[18] Wikidata statistics dashboard for references. https://grafana.wikimedia.org/d/000000182/wikidata-datamodel-references?orgId=1&from=1514836723618&to=1543694323619. Accessed: 2018-12-11.

[19] Angles, Renzo. 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. DOI: 10.1109/ICDEW.2012.31.

[20] Angles, Renzo, Prat-Pérez, Arnau, Dominguez-Sal, David, and Larriba-Pey, Josep-Lluis. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 15:1–15:7, New York, NY, USA, 2013. ACM. DOI: 10.1145/2484425.2484440.

[21] Cyganiak, Richard, Wood, David, Lanthaler, Markus, Klyne, Graham, Carroll, Jeremy J, and McBride, Brian. RDF 1.1 concepts and abstract syntax. *W3C recommendation*, 25(02), 2014.

[22] Duan, Songyun, Kementsietsidis, Anastasios, Srinivas, Kavitha, and Udrea, Octavian. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. pages 145–156, 01 2011. DOI: 10.1145/1989323.1989340.

[23] Ehrlinger, Lisa and Wöß, Wolfram. Towards a definition of knowledge graphs. In *SEMANTiCS*, 2016.

[24] Erling, Orri, Averbuch, Alex, Larriba-Pey, Josep, Chafi, Hassan, Gubichev, Andrey, Prat, Arnau, Pham, Minh-Duc, and Boncz, Peter. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM. DOI: 10.1145/2723372.2742786.

[25] Erxleben, Fredo, Günther, Michael, Krötzsch, Markus, Mendez, Julian, and Vrandečić, Denny. Introducing Wikidata to the linked data web. In *Proceedings of the 13th International Semantic Web Conference (ISWC'14)*, volume 8796 of *LNCS*, pages 50–65. Springer, 2014.

[26] Färber, Michael, Bartscherer, Frederic, Menne, Carsten, and Rettinger, Achim. Linked data quality of DBpedia, Freebase, OpenCyc, Wikidata, and Yago. *Semantic Web*, pages 1–53, 2016.

[27] Harris, Steve, Seaborne, Andy, and Prudhommeaux, Eric. SPARQL 1.1 query language. *W3C recommendation*, 21(10), 2013.

[28] Hartig, O. and Thompson, B. Foundations of an Alternative Approach to Reification in RDF. *ArXiv e-prints*, June 2014.

[29] Hernández, Daniel, Hogan, Aidan, and Krötzsch, Markus. 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.

[30] Hernández, Daniel, Hogan, Aidan, Riveros, Cristian, Rojas, Carlos, and Zerega, Enzo. Querying wikidata: Comparing SPARQL, relational and graph databases. In *International Semantic Web Conference*, pages 88–103. Springer, 2016.

[31] Iosup, Alexandru, Hegeman, Tim, Ngai, Wing Lung, Heldens, Stijn, Prat-Pérez, Arnau, Manhardto, Thomas, Chafio, Hassan, Capotă, Mihai, Sundaram, Narayanan, Anderson, Michael, Tănase, Ilie Gabriel, Xia, Yinglong, Nai, Lifeng, and Boncz, Peter. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.*, 9(13):1317–1328, September 2016. DOI: 10.14778/3007263.3007270.

[32] Jouili, S. and Vansteenberghe, V. An empirical comparison of graph databases. In *2013 International Conference on Social Computing*, pages 708–715, Sept 2013. DOI: 10.1109/SocialCom.2013.106.

[33] Kotsev, Venelin, Minadakis, Nikos, Papakonstantinou, Vassilis, Erling, Orri, Fundulaki, Irini, and Kiryakov, Atanas. Benchmarking RDF query engines: The LDBC semantic publishing benchmark. In *BLINK@ ISWC*, 2016.

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

[35] Morsey, Mohamed, Lehmann, Jens, Auer, Sören, and Ngonga Ngomo, Axel-Cyrille. Dbpedia SPARQL benchmark – performance assessment with real queries on real data. In Aroyo, Lora, Welty, Chris, Alani, Harith, Taylor, Jamie, Bernstein, Abraham, Kagal, Lalana, Noy, Natasha, and Blomqvist, Eva, editors, *The Semantic Web – ISWC 2011*, pages 454–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[36] Nguyen, Vinh, Bodenreider, Olivier, and Sheth, Amit. Dont like RDF reification? *Proceedings of the 23rd international conference on World wide web - WWW 14*, page 759770, Apr 2014. DOI: 10.1145/2566486.2567973.

[37] Noy, Natasha, Rector, Alan, Hayes, Pat, and Welty, Chris. Defining n-ary relations on the semantic web. *W3C working group note*, 12(4), 2006.

[38] Pacaci, Anil, Zhou, Alice, Lin, Jimmy, and zsu, M. Tamer. Do we need specialized graph databases?: Benchmarking real-time social networking applications. pages 1–7, 05 2017. DOI: 10.1145/3078447.3078459.

[39] Pan, Zhengyu, Zhu, Tao, Liu, Hong, and Ning, Huansheng. A survey of RDF management technologies and benchmark datasets. *Journal of Ambient Intelligence and Humanized Computing*, 9(5):1693–1704, Oct 2018. DOI: 10.1007/s12652-018-0876-2.

[40] Robinson, Ian, Webber, Jim, and Eifrem, Emil. *Graph Databases*. OReilly Media, 2015.

[41] Rodriguez, Marko A. A letter regarding native graph databases. https://web.archive.org/web/20180828112004/https://www.datastax.com/dev/blog/a-letter-regarding-native-graph-databases, 2013.

[42] Rodriguez, Marko A. and Neubauer, Peter. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.