Object-UOBM: An Ontological Benchmark for Object-Oriented Access

Martin Ledvinka(✉) and Petr Křemen

Czech Technical University in Prague, Prague, Czech Republic
{martin.ledvinka,petr.kremen}@fel.cvut.cz

Abstract. Although many applications built on top of market-ready ontology storages are generic and lack dependence on the particular application domain, most users prefer applications tailored to their particular task. Such applications are typically built using object-oriented paradigm that accesses data differently than generic applications. In this paper, we define a benchmark consisting of an ontology and ontological queries tailored for testing suitability of ontological storages for object-oriented access. We present results of experiments on several state-of-the-art ontological storages and discuss their suitability for the purpose of object-oriented application access.

Keywords: Ontological storage · Benchmark · Object-oriented applications

1 Introduction

Although many applications built on top of market-ready ontology storages are generic and lack dependence on the particular application domain, most users prefer applications tailored to their particular task. Such applications are typically built using object-oriented paradigm, which sticks to particular entities and their relationships in the domain, see [11].

In order to construct/store an object model expressed in the given application language (e.g. Java), specific types of ontological queries are posed to the underlying storage. Different frameworks for object-oriented access to OWL [17] ontologies use different access techniques to the underlying ontology stores, including custom interface or SPARQL [9] queries. To unify these ontology storage access techniques, OntoDriver was introduced in [14] as a layer for storing object models into ontological storages. OntoDriver defines an API to be used by an application access layer, like the JOPA framework [12], [13]. The API defines optimized ontological methods that are suitable for object-oriented access, allowing CRUD[1] operations, transactional support, multiple contexts, SPARQL queries, as well as integrity constraints checking.

[1] Create, Retrieve, Update, Delete.

© Springer International Publishing Switzerland 2015
P. Klinov and D. Mouromtsev (Eds.): KESW 2015, CCIS 518, pp. 132–146, 2015.
DOI: 10.1007/978-3-319-24543-0_10

In this paper, we contribute to this story by comparing the actual performance of various ontological queries tailored for object-oriented access to ontology storages. As a side-effect, we experimentally justify the OntoDriver design. For this purpose we define a benchmark consisting of an ontology and ontological queries. Next we perform experiments on several state-of-the-art storages and discuss their suitability for the purpose of object-oriented application access.

Section 2 shows the relationship of our work to the state-of-art research. Section 3 reviews the JOPA architecture, and the OntoDriver API in particular, together with a running example used in benchmark queries. Section 4 defines the benchmark based on existing datasets. Section 5 presents experiments on existing ontological storages and discusses their suitability for object-oriented applications. The paper is concluded in Section 6.

2 Related Work

Interoperability benchmarking for semantic web is discussed in [5]. More interesting for our purposes are various ontological benchmarks combining expressive power of ontological languages with the medium-to-large dataset size. We consider the following two benchmarks.

LUBM [8] was one of the first OWL [18] benchmarks, featuring support for extensional queries, data scaling and moderate ontology size. It contains fourteen generic queries and a data generator allowing to produce randomly created datasets in the university domain. The generation is parametrized by the requested number of universities, producing around 100k triples for each university. The complexity of the ontological schema is significantly below OWL expressiveness, lacking nominals, number restrictions, or negation.

UOBM [16] is an extension of LUBM aimed at leveraging expressiveness towards OWL-DL by augmenting LUBM schema with the missing constructs (see Section 4.1). The dataset contains around 250k triples for each university.

For SPARQL benchmarking, the Berlin SPARQL benchmark (BSBM) [2] has been defined. This benchmark is purely RDF-oriented, focusing at SPARQL queries without ontological inference.

Neither of these benchmarks, however, is tailored to the scenario of object-oriented access to ontological knowledge.

3 Background

The main motivation for our benchmark is finding out how some of the most advanced ontology storages are suitable for access by object-oriented applications. Such applications often use frameworks providing object-ontological mapping (OOM) to facilitate working with the data. OOM is a technique for mapping ontological classes to classes in object-oriented paradigm, individuals to object instances and properties to instance attributes. Such object representation is arguably easier to use for application programmers. Examples of frameworks providing, among other features, OOM are Empire [7], AliBaba[2] and JOPA.

[2] https://bitbucket.org/openrdf/alibaba, accessed 07-14-2015.

3.1 JOPA

JOPA (Java OWL Persistence API) is a Java persistence API and provider for applications working with ontological data. It tries to present API resembling JPA 2 [10], with which Java developers are familiar. In addition to the compiled object model present also in other OOM frameworks, JOPA provides access to properties not mapped by the object model, capturing thus also the dynamic nature of the underlying knowledge [12].

3.2 OntoDriver

In order to provide access to various ontology storages without committing to a vendor-specific API of the storage, we proposed the concept of *OntoDriver* as a software layer which separates the object-ontological mapping performed by JOPA and the actual ontology access [13]. We designed the OntoDriver API to provide a single API for object-oriented access to different storages as well as enough margin for vendor-specific optimizations of the operations performed by OntoDriver. In its nature, OntoDriver is very similar to standard JDBC[3] drivers used for accessing relational databases, although instead of being based on statements written in query and data definition language (SQL in case of JDBC), we defined specialized CRUD operations.

When one examines the operations required by JOPA or any other object-ontological mapping performing framework, it turns out that their set is rather small. Object-ontological mapping does not require any complex queries consisting of joins over multiple properties or selecting data with unbound subject and object variables. On the contrary, typical operations required by JOPA consist of selecting values of beforehand known set of properties of a single individual or retracting and asserting values of such properties.

3.3 Example Object Model

The queries used in our benchmark are based on a relatively simple object model, which we will now describe. The object model, however, exercises a wide range of possible property usage, including single and multi-valued data and object properties, types specification or unmapped properties [13].

Since our benchmark is based on the LUBM and UOBM datasets, the object model is also built upon the university and students domain. A class diagram of the domain model can be seen in Figure 1. Besides simple data properties like name, email and telephone, the model also contains a number of object properties. Most notable of them are

- *hasSameHomeTownWith*, which is a transitive and symmetric property and occurs only in UOBM datasets,
- *isFriendOf*, which is a symmetric property and occurs only in UOBM datasets,

[3] Java Database Connectivity.

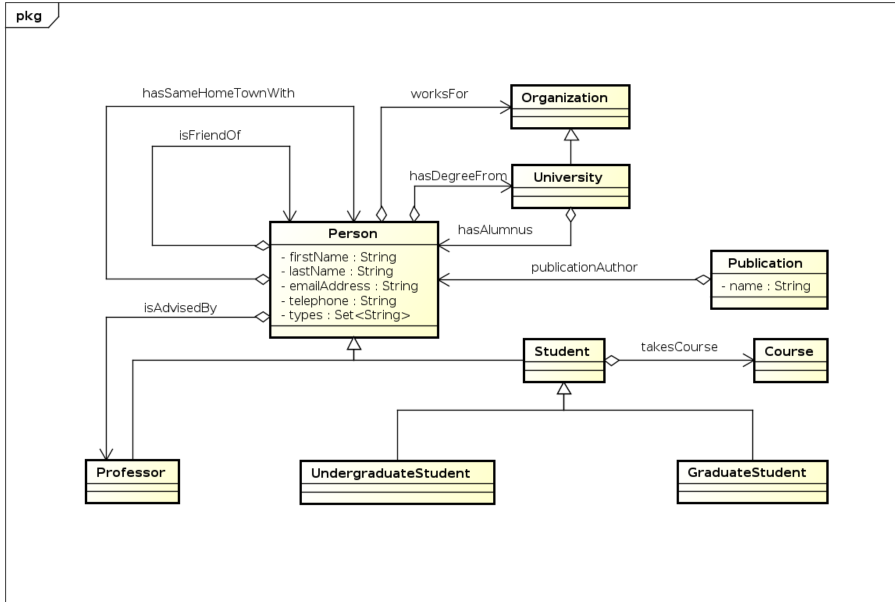

Fig. 1. Object model used as a base for the benchmark queries. The properties are from UOBM, most of them having LUBM equivalents, although sometimes with different names.

– *hasAlumnus*, which is an inverse of *hasDegreeFrom* and is never used explicitly in the datasets, therefore always requiring reasoning to return results,
– *hasDegreeFrom*, which has a number of sub-properties.

4 Benchmark

In this section we describe the benchmark set up, namely the datasets and queries used, storages that we evaluated and the chosen metrics. Let us begin with description of the datasets.

4.1 Datasets

The proposed benchmark reuses parts of existing OWL benchmark datasets – LUBM and UOBM.

LUBM (Lehigh University Benchmark) is an ontology and benchmark developed to evaluate semantic web knowledge base systems [8]. The ontology concerns university domain and contains basic OWL constructs like concept and property

hierarchies and inverse properties, domains and ranges. A simple dataset generator is provided, enabling creation of datasets of various sizes and structure. The generator creates synthetic datasets with the specified number of universities, where each university contains approximately 100k triples.

The generated datasets were split in files by departments, where each university contained in average 15 departments. This lead in the largest server case to loading the data from approximately 16 000 files.

UOBM (University Ontology Benchmark) is an ontology benchmark built upon LUBM [16]. The ontology itself is based on the LUBM ontology, but uses different names for some of the properties and, in addition, contains several more complex OWL constructs. UOBM ontology exists in two versions - a less expressive OWL-Lite version and a more expressive OWL-DL version. We used the OWL-DL version, which adds constructs like transitive properties, equivalent properties, class instance enumeration and cardinality restrictions. UOBM as a benchmark contains a pre-defined set of datasets, but the number of these datasets is rather low, because only three datasets (containing 1, 5 and 10 universities) for each version exist. Fortunately, a data generator was developed at the University of Oxford [21]. This generator creates datasets for the more expressive OWL-DL version and each generated university contains approximately 250k triples.

In contrast to the LUBM datasets, data generated for UOBM were split in files by universities.

4.2 Queries

Both of the benchmarks that we decided to utilize contain their own sets of SPARQL queries that are used to evaluate the storage systems. These queries are general-purpose statements containing joins, unbound subjects and/or objects. Such queries are tailored for reasoner benchmarking, trying to capture complex data structure. Most of the time, applications using object model require a rather narrow set of operations used for creating, retrieving, updating and deleting data. These operations are in case of JOPA exposed through the OntoDriver API and are described in greater detail in [14].

For the purpose of evaluating ontology storages in terms of efficiency for object-oriented application access, we have defined a set of eight SPARQL and SPARQL Update [6] queries, which correspond to a major subset of the operations declared in OntoDriver API. Let us now briefly describe each query.

Q_{S1} selects all statements with the given individual as subject. This query corresponds to a retrieve operation, which loads an entity instance from the storage. In this case the entity attributes either cover most of the individual's properties so that it is more efficient to ask for them in a single query instead of asking for each property separately (see query Q_{S2}), or the entity contains a field which holds all the unmapped property values (see [14]). A single query with bound subject and unbound property and object is also used to load entities in Empire.

Q_{S2} is similar to query Q_{S1}, it represents the same entity retrieval operation. However, in this instance the required properties are enumerated and the whole query is thus a union of triple patterns with bound subject and predicate. Such a query would be used in case the entity contained several attributes, but the ontological individual had multiple other properties connected to it. As in case of Q_{S1}, it is the responsibility of the OOM provider to map the returned values to entity attributes.

$Q_{S2^{OPT}}$ is a variation of Q_{S2}, only in this case using SPARQL's *OPTIONAL* operator instead of *UNION*. Although the properties of $Q_{S2^{OPT}}$ do not seem to be very suitable for the OOM case (it returns the result as a cartesian product of the matching values for multiple variables), we keep it in the comparison for the sake of completeness of our study.

Q_{S3} represents a *find all* operation – it returns attributes of all individuals of the specified class. This operation is actually not defined in the OntoDriver API, but retrieval of all instances of a certain type is a very common operation in object-oriented applications. Therefore, we decided to add such a query into the benchmark. In addition, given the structure of the datasets, this query returns a large amount of results (hundreds of thousands for larger datasets), so we also used it to see how the evaluated storages were able to handle such situations.

Q_{S4} retrieves values of a single property for the given individual. This query can be viewed as an attempt to load values of a single attribute and in JOPA such operation is used to provide lazily loaded attributes[4] for entity instances. Moreover, this query requires reasoning to take place to return any results, because it looks for values of the *hasAlumnus* property, which is never explicitly used in the dataset. This type of query is also used by AliBaba, which loads entities by querying attributes one by one.

Q_{U5} removes values of several properties of the given individual and then inserts new values for those properties. Thus it corresponds to an entity attribute update. In the general case of ontological storages, this cannot be achieved by updating the values in-place, but the old values have to removed and new inserted.

Q_{I6} inserts assertion about individual's type(s) and property values into the ontology. Such query represents a *persist* operation, when new entity instance is saved into the storage.

Q_{D7} deletes property assertions about the given individual. JOPA performs *epistemic remove* [14], in which only values of properties known to the object model are removed. Thus the query deletes statements by specifying triple patterns with bound subject and property, instead of simply removing everything concerning the removed individual.

The queries themselves are written in full in our technical report [15], appendix A. The aforementioned queries represent the core operations defined in the OntoDriver API and used by JOPA. Other methods defined in the API,

[4] Attributes loaded only when actually accessed.

e.g. *getTypes*, *updateTypes*, are simply variations of the core methods. The key difference between our queries and the queries in LUBM and UOBM is that our queries do not require any joins and most of them use bound subject. LUBM and UOBM queries, on the other hand, always have variable in the subject position. In addition, neither LUBM nor UOBM contain SPARQL Update queries, which are crucial for application access.

It is worth mentioning that the current experimental prototype of OntoDriver uses dedicated methods of the storage access framework (Sesame API [3] in this case) instead of relying on generation of SPARQL queries. However, the OntoDriver implementations are not restricted to either approach. Also, we are using SPARQL queries in our benchmark because the same queries can be used over various storages without any modifications.

4.3 Storages

Following our theoretical study of OntoDriver operations complexity for storages GraphDB and Stardog in [14], we decided to experimentally verify our conclusions. Another reason for choosing GraphDB and Stardog is that they represent two complementing approaches to reasoning – the former performing total materialization using forward chaining, the latter executing inference at real time without any materialization.

GraphDB. GraphDB, formerly known as OWLIM [1], is a storage implementing Sesame's storage and inference layer (SAIL) paradigm. Thus, it can be accessed through the Sesame API in the same way as any other SAIL-compliant storage.

GraphDB performs reasoning by materializing all possible inferred knowledge on insertion. This theoretically leads to very fast query answering, but slower updates. Especially statement retraction can have detrimental effect on performance, since GraphDB employs a combination of forward and backward chaining in order to identify inferred knowledge that is no longer backed by any explicit statement.

GraphDB's inference engine is rule-based. Therefore, besides pre-defined rulesets for various OWL and OWL 2 profiles, it provides the possibility to define user's own inference rules. The rule-based inference, however, represents a restriction on expressiveness, since some OWL constructs, e.g. full logical negation, cannot be expressed using rules.

Stardog. Stardog[5] is an RDF database with its own proprietary API, but providing bridges for Jena [4] and Sesame API as well. Stardog does no materialization. Instead, it performs real time model checking, inferring knowledge at query time. This has obviously impact on query performance, however it also results in smaller repository size and theoretically faster updates.

[5] http://www.stardog.com, accessed 05-14-2015.

Stardog uses Pellet [20] as its reasoner and thus supports full OWL 2 DL reasoning over TBox. Queries involving ABox can use reasoning in profiles OWL 2 EL, OWL 2 QL, OWL 2 RL [18] and a combination of them (called in Stardog's documentation SL).

4.4 Benchmark Metrics

To thoroughly evaluate performance of tested storages, we have used the following metrics:

- **Dataset load time** – first task when running the benchmark was always loading the dataset. We measured how long it took the storages to fully load the data,
- **Repository size** – we measured the size of the resulting repository (in statements) to determine what effect has materialization on space consumption,
- **Query execution time** – the most obvious criterion, which tells us how fast can the storage perform a query and return results for it. For easier interpretation, we express the results in a more read-friendly *queries per second* measure,
- **Query completeness and soundness** – since reasoning was involved in some of the queries, we also verified soundness and completeness of the results.

Benchmark Application. We developed a small application which served as benchmark runner. Its task was to load the queries, run them against a given SPARQL endpoint, collect results and measure the execution time of the queries. The application itself was written in Java and is very similar to the benchmark application used by BSBM [2].

5 Experiments

In this section we present results we obtained by running the benchmark against both GraphDB and Stardog.

5.1 Experiment Setup

The following versions of the storages were used in the benchmark:

- GraphDB-SE 6.1 SP1,
- Stardog 3.0.

Both storages were evaluated in two modes, one with no reasoning and the other with maximum supported expressiveness, in case of Stardog, we thus used the *SL* reasoning level. In case of GraphDB, we used the *OWL-Max* rule set, which corresponds to the maximum subset of OWL that can be captured using rules.

Table 1. Parameters of machines that were used to run the benchmark.

PC	Server
• Linux Mint 17 64-bit • CPU Intel i5 2.67 GHz (4 cores) • 8 GB RAM • 500 GB SATA HDD • Java 8u40, -Xms6g -Xmx6g • Apache Tomcat 8.20	• Linux Debian 3.2.65 64-bit • CPU Intel Xeon E3-1271 3.60 GHz (8 cores) • 32 GB RAM • 100 GB SSD HDD • Java 8u40, -Xms20g -Xmx20g • Apache Tomcat 8.20

The experiments were run by our benchmark application in two batches, the first containing all the *select* queries, the second containing the *update* queries. The queries were run in rounds in which all the queries were executed sequentially. We ran 20 warm-up rounds without measuring anything and then 500 rounds, from which the results were computed.

The experiments were run in two environments, one being a regular PC with a SATA hard drive, the other a server with an SSD drive. Setup of the machines can be seen in Table 1.

We used datasets of varying size, starting with one university up to 100 universities for the PC, amounting to approximately 13 million statements in case of the UOBM dataset, and 600 universities for the server (around 153 million statements for the UOBM dataset).

Please note that due to space restrictions, we present here only some of the results we measured. Tables with all recorded data can be found in the technical report [15], appendices B, C, D, E and F.

5.2 Dataset Loading

Dataset loading was conducted using bulk loaders provided by both storages. In case of Stardog, this meant specifying files to load on repository creation.

Bulk loading on GraphDB was a more complicated matter, because since GraphDB does materialization on insertion, it is necessary to specify inference level (ruleset) before creating the repository. In addition, to perform better, GraphDB asks the user to specify the expected size of the storage, including the inferred statements. Therefore, we first had to load the datasets without measuring the performance and feed the resulting repository size to the configuration of the actual benchmark repository.

Stardog is in all cases able to load the datasets significantly faster than GraphDB. We expected Stardog to outperform GraphDB with the OWL-Max ruleset, because of the inference taking place in GraphDB during insertion. However, even for an empty ruleset, where no materialization occurs, Stardog performed much better than GraphDB. It is also interesting to note that in case of the more expressive UOBM datasets, the size of the GraphDB storage with inferred statements is more than twice the size of the loaded data.

Figure 2 shows a chart of dataset loading times on the server. It is clear that GraphDB, performing materialization, is significantly slower than Stardog.

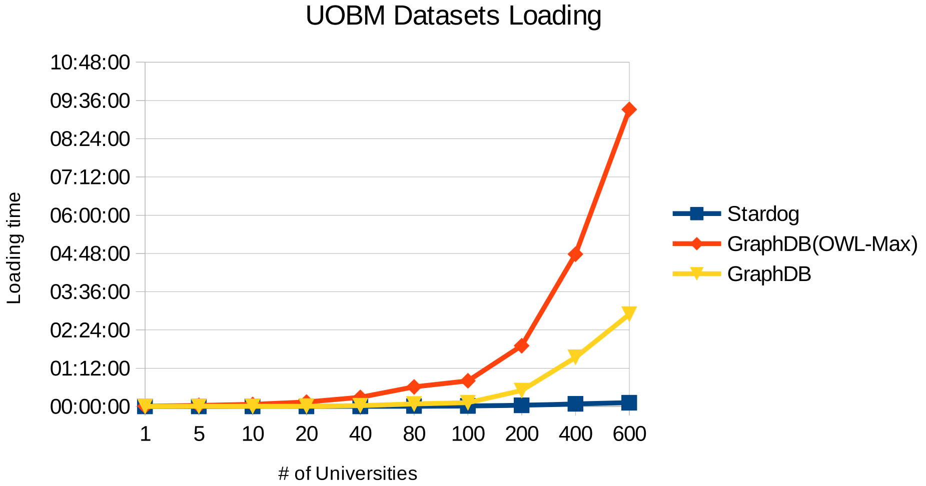

Fig. 2. Loading times for the UOBM datasets, executed on the server.

For the largest dataset the loading time in GraphDB almost reaches 9 hours and 30 minutes, while Stardog loads it in 7 minutes and 10 seconds.

Table 2 shows the loading times in numbers plus resulting sizes of the datasets. Notice that for the largest dataset and the OWL-Max ruleset, we were unable to determine repository size, because GraphDB kept failing with exceptions for any queries we issued to the storage through its SPARQL endpoint. Therefore, we cannot even be certain that the storage was able to load the whole dataset and perform all materialization.

Table 2. Loading of the UOBM datasets on the server. $\#U$ is the number of universities in dataset, T is the loading time, S is the size of the resulting repository in triples. ** It is not certain that the dataset was fully loaded, because the storage failed to process all queries.*

$\#U$	$T_{Stardog}$	$T_{GraphDB}^{OWL-Max}$	$T_{GraphDB}$	$S_{Stardog}$	$S_{GraphDB}^{OWL-Max}$	$S_{GraphDB}$
1	1 s	22 s	2 s	258 370	542 312	258 370
5	3 s	121 s	8 s	1 355 941	2 832 003	1 355 941
10	4 s	229 s	15 s	2 509 169	5 242 259	2 509 169
20	10 s	493 s	32 s	5 183 092	10 829 617	5 183 092
40	20 s	970 s	93 s	10 212 049	21 339 224	10 212 049
80	42 s	2212 s	280 s	20 286 983	42 390 542	20 286 983
100	53 s	2892 s	419 s	25 478 086	53 239 768	25 478 086
200	124 s	6770 s	1854 s	51 110 402	106 795 361	51 110 402
400	293 s	17 371 s	5760 s	102 224 372	-	102 224 372
600	430 s	30 758 s*	10 414 s	153 178 000	-	153 178 000

5.3 Benchmark Results for PC

Experiments undertaken on the PC confirm theoretical expectations in case of select queries. Indeed, GraphDB with materialized inference provides faster execution times than Stardog with reasoning at query time. The difference is actually very significant, as can be seen for example in Figure 3. The difference between GraphDB with and without inference can be easily explained by the fact that in case of storage with inference, the repository size is more than twice the size of the repository without inference.

More surprising is the fact that GraphDB actually performs better even in case of update queries. Theoretical expectations in this case favour Stardog, because it does not have to perform any inference during updates. However, Figure 4 shows that GraphDB not only performs better without inference, but even with inference it is able to execute more queries per second than Stardog. The fact that Stardog shows virtually the same performance with and without reasoning is no surprise, because in fact for these update queries no reasoning occurs. The situation is radically different only in case of Q_{D7}, where GraphDB with enabled inference performs by far worst, most likely due to the fact that a combination of forward and backward chaining occurs in order to find out which inferred statements should be removed as well [1].

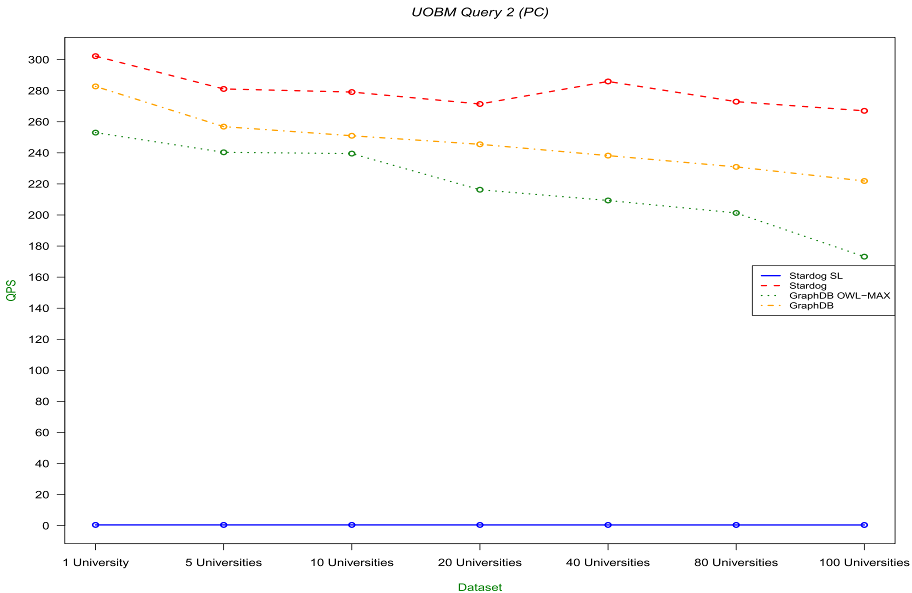

Fig. 3. Plot of storage performance for Q_{S2} on UOBM datasets. QPS represents the number of queries executed per second, higher is better. While Stardog without reasoning performs the best, it clearly looses when inference is required.

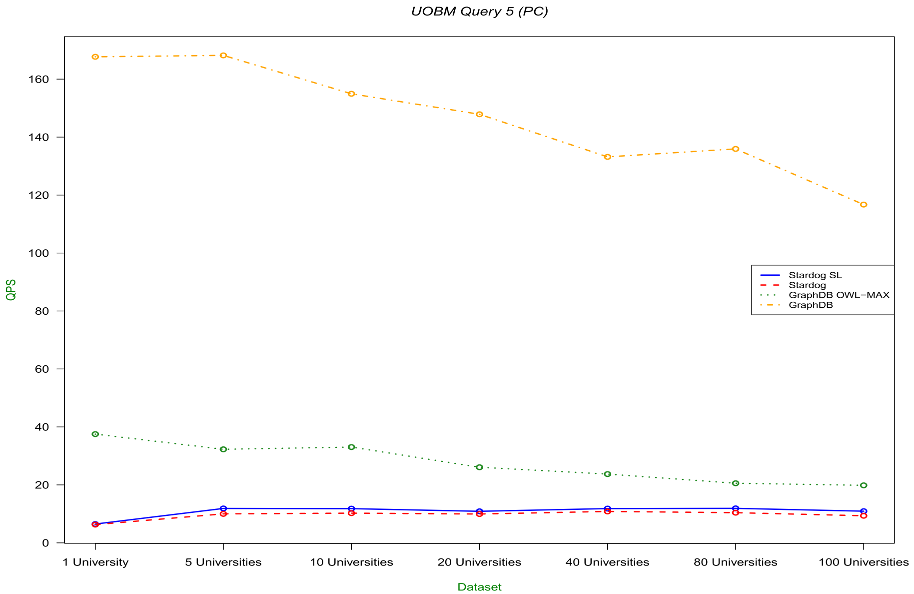

Fig. 4. Plot of storage performance for Q_{U5} on UOBM datasets. GraphDB outperforms Stardog even in update operation, although the difference for repository with inference is not so large.

5.4 Benchmark Results for Server

Evaluation on the server confirmed our observation from experiments on the PC only partially. For select queries, the order is the same as on the PC – Stardog without reasoning being the fastest, Stardog with reasoning being by far the slowest. The only exception in this situation is Q_{S3}, which returns a very large number of results, with which both storages struggle and for the larger datasets the query takes seconds to execute (see Figure 5).

However, the situation is very peculiar for the update queries. For query Q_{D7} Stardog now outperformed GraphDB in both inference settings, while on PC it was the case only for GraphDB with inference enabled. The situation becomes even more complicated for queries Q_{U5} and Q_{I6}, where considerable fluctuations in performance appear in Stardog. Because of these fluctuations, it is very difficult to discover any trends in its performance. We can see from the results, that the differences between Stardog and GraphDB are smaller than on the PC and sometimes Stardog even comes on top of the comparison, but the pattern is not steady. The fluctuations are clearly visible in Figure 6. After computing standard deviation for the Stardog results, it turned out that while for queries Q_{S1}, Q_{S2} and Q_{S3} the standard deviation was around 1 % of the average query execution time, for query Q_{S4} it grows to 27 % in average and for the update queries Q_{U5}, Q_{I6} and Q_{D7} it rises to 46 % in average. The performance of Stardog in these cases appears to be highly unpredictable.

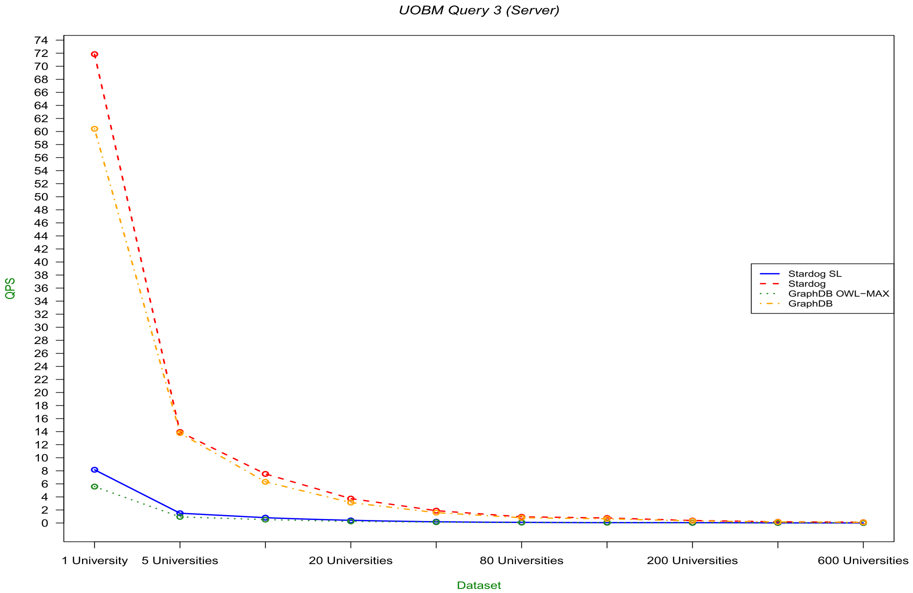

Fig. 5. Plot of storage performance for Q_{S3} on UOBM datasets. Large number of results has a significant impact on performance of the storages.

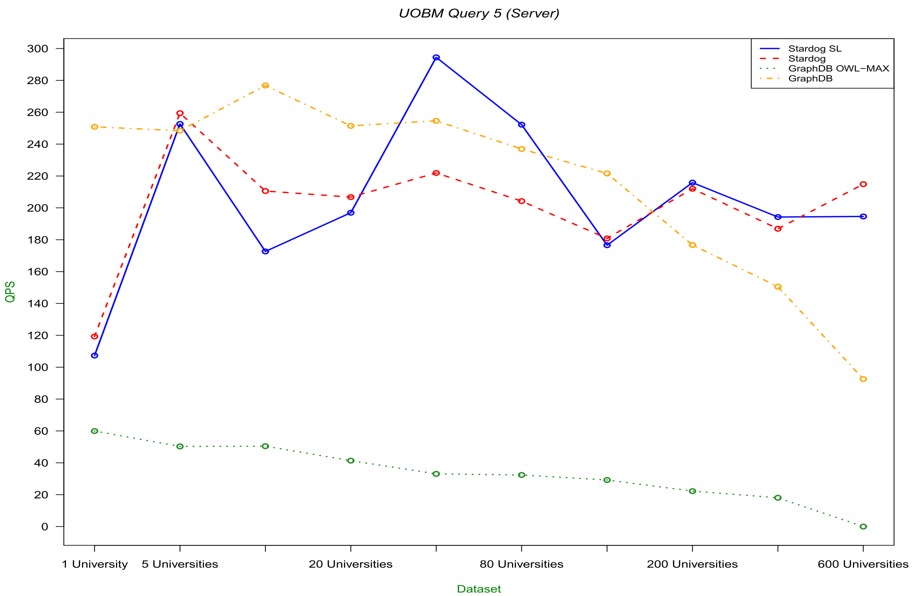

Fig. 6. Plot of storage performance for Q_{U5} on UOBM datasets. Considerable fluctuations appear in performance of Stardog.

It is also worth noting that Stardog was not able to handle the dataset with 800 LUBM universities and kept crashing on queries. GraphDB, on the other hand, failed to work with the UOBM 600 universities dataset (using the OWL-Max ruleset), throwing exceptions when queries were sent to it.

6 Conclusion

We have designed a benchmark for evaluating suitability of ontology storages for object-oriented applications and used it to test two of the most advanced contemporary storage engines. Both storages showed extremely good performance in case of select queries without reasoning. Our theoretical expectations about materialization being slow on bulk insert and real time reasoning at query time were also confirmed. More surprising was the fact that GraphDB, using total materialization, did in most cases outperform Stardog even for update operations, for which Stardog should be theoretically more suitable. Overall, GraphDB appears to be more suitable for the object-oriented application access scenario, in which frequent data updates are expected, because it provides satisfiable performance even for large datasets. Stardog clearly outperforms GraphDB only in case of select queries without reasoning.

However, GraphDB (and storages performing materialization in general) has a major disadvantage in that the user has to specify inference level before actually inserting data into the storages. Real time reasoning, on the other hand, lets the user choose reasoning level for every query he executes.

In the future, we would like to more closely investigate the performance fluctuations of Stardog in the server environment. They appear without any obvious reason and cause the performance of the storage to be extremely unpredictable. Also, based on the benchmark results and the characteristics of the storages, we will research possible optimizations for OntoDriver implementations, for example the results indicate that when loading an entity, it is more favourable to use only bound subject in the queries (Q_{S1}), instead of doing a union of triple patterns based on properties (Q_{S2}). $Q_{S2^{OPT}}$ has worse performance than Q_{S2}, which is not surprising, given that Q_{S2} uses only *UNION* and thus has PTIME complexity, whereas $Q_{S2^{OPT}}$ with *OPTIONAL* is PSPACE-complete [19].

Acknowledgment. This work was supported by grants No. SGS13/204/OHK3/3T/13 Effective solving of engineering problems using semantic technologies of the Czech Technical University in Prague and No. TA04030465 Research and development of progressive methods for measuring aviation organizations safety performance of the Technology Agency of the Czech Republic.

References

1. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: A family of scalable semantic repositories. Semantic Web - Interoperability, Usability, Applicability (2010)

2. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. International Journal On Semantic Web and Information Systems (2009)
3. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: a generic architecture for storing and querying RDF and RDF schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)
4. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: Proceedings of the 13th International World Wide Web conference (Alternate Track Papers & Posters), pp. 74–83 (2004)
5. Garcia-Castro, R.: Benchmarking semantic web technology. Studies on the Semantic Web. IOS Press, Amsterdam (2009)
6. Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 Update. Tech. rep., W3C (2013)
7. Grove, M.: Empire: RDF & SPARQL Meet JPA. semanticweb.com, April 2010. http://semanticweb.com/empire-rdf-sparql-meet-jpa_b15617
8. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3**(2–3), 158–182 (2005). http://dx.doi.org/10.1016/j.websem.2005.06.005, http://www.bibsonomy.org/bibtex/2924e60509d7e1b45c6f38eaef9a5c6bb/gromgull
9. Harris, S., Seaborne, A.: SPARQL 1.1 query language. Tech. rep., W3C (2013)
10. JCP: JSR 317: Java^{TM} Persistence API, Version 2.0 (2009)
11. Křemen, P.: Building Ontology-Based Information Systems. Ph.D. thesis, Czech Technical University, Prague (2012)
12. Křemen, P., Kouba, Z.: Ontology-driven information system design. IEEE Transactions on Systems, Man, and Cybernetics: Part C **42**(3), 334–344 (2012). http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6011704
13. Ledvinka, M., Křemen, P.: JOPA: developing ontology-based information systems. In: Proceedings of the 13th Annual Conference Znalosti 2014 (2014)
14. Ledvinka, M., Křemen, P.: JOPA: accessing ontologies in an object-oriented way. In: Proceedings of the 17th International Conference on Enterprise Information Systems (2015)
15. Ledvinka, M., Křemen, P.: Object-UOBM: An Ontological Benchmark for Object-oriented Access. Tech. rep., Czech Technical University in Prague (2015)
16. Ma, L., Yang, Y., Qiu, Z., Xie, G.T., Pan, Y., Liu, S.: Towards a complete OWL ontology benchmark. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 125–139. Springer, Heidelberg (2006)
17. Motik, B., Parsia, B., Patel-Schneider, P.F.: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3C recommendation, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/
18. Patel-Schneider, P.F., Motik, B., Grau, B.C.: OWL 2 Web Ontology Language Direct Semantics. W3C recommendation, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/
19. Schmidt, M., Meier, M., Lausen, G.: Foundations of sparql query optimization. In: Proceedings of the 13th International Conference on Database Theory, ICDT 2010, pp. 4–33. ACM, New York (2010)
20. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web **5**(2), June 2007
21. Zhou, Y., Grau, B.C., Horrocks, I., Wu, Z., Banerjee, J.: Making the most of your triple store: query answering in OWL 2 using an RL reasoner. In: Proceedings of the 22nd International Conference on World Wide Web (2013)