Performance Analysis and Optimization of Semantic Queries

Philipp Hertweck
Fraunhofer IOSB
Karlsruhe, Germany
Email: philipp.hertweck@iosb.fraunhofer.de

Erik Kristiansen
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: erik@kristiansen.de

Tobias Hellmund
Fraunhofer IOSB
Karlsruhe, Germany
Email: tobias.hellmund@iosb.fraunhofer.de

Jürgen Moßgraber
Fraunhofer IOSB
Karlsruhe, Germany
Email: juergen.mossgraber@iosb.fraunhofer.de

***Abstract*—Recently, the usage of triplestores has increased in complex computer systems. Traditionally, they are used for representing static knowledge. In the last years, systems started using semantic triplestores in highly dynamic scenarios, e.g., in the context of civil protection. In these use cases, performance characteristics are more and more important. There are various aspects influencing the query performance. We have noticed that already the query structure has a significant impact on the execution time. SPARQL Protocol And RDF Query Language (SPARQL) is a widely used standard for querying triplestores. In this work, we have developed SPARQL query patterns and evaluated their performance characteristics. For this, a literature review was done to select a suitable benchmark. As a result, we provide eight recommendations for formulating SPARQL queries. These can be easily used by everybody without a deeper knowledge about the implementation of the triplestore, which contains the desired data.**

***Keywords*–SPARQL Performance; Triplestore; Benchmark; Query optimization.**

I. Introduction

The World Wide Web was originally designed to be used by humans; to foster machine understanding of the incomprehensible large amount of data in the web, the Semantic Web was envisioned. This vision focuses on the reuse, availability and interoperability of data. A milestone on the path to reach this vision is the Resource Description Framework (RDF [1]), which defines a data model that encompasses Unique Resource Identifiers (URIs) and requires data structured as triples. A triple is a statement about data that consists of *subject*, *predicate* and *object*. Since all three are identified by an URI, they can be uniquely recognized and linked by machines. For example, the Linked-data project [2] started to link and structure the semantic data available on the Internet.

A set of RDF triples forms a graph. These graphs are stored in so-called triplestores. To systematically retrieve data from such stores, the World Wide Web Consortium (W3C) standardized SPARQL [3], a declarative query language for RDF based data. There are other query languages for data represented in RDF, but as SPARQL is the de-facto standard query language for the Semantic Web, we do not consider other languages. Since the implementation of triplestores varies from product to product, the performance of each is different as well. Since a growing amount of (critical) information systems integrate data in form of triples, the performance of SPARQL queries is increasingly important. The following two examples show the wide range of usage of semantic technologies. Semantic integration [4] [5] can be applied in the context of crisis response to support decision support [6]. Another example shows the implementation of semantic data to protect cultural heritage [7].

There are several possibilities to optimize the execution of SPARQL queries. Either on the data (representation) itself, the triplestore's implementation (internal representation, query execution, query optimization, etc.) or on the usage of the triplestore. In this paper, we are focusing on the later. We examine: are there some easily applicable rules an end-user should follow while formulating SPARQL queries?

To approach this question, first a literature review of existing triplestore benchmarks was conducted (Section III). Different query patterns were developed (Section IV). Those were compared by executing them, with the help of the selected benchmark against a triplestore. Our evaluation (Section V) uses Apache Fuseki, since it is a commonly used, open source triplestore implementation. With this evaluation, the influencing factors within a SPARQL query were elaborated and eight recommendations (Section VI) for query formulations were derived.

The contributions of this paper are: 1) A literature review of existing SPARQL benchmarks and a selection, which can be used to evaluate the performance of different SPARQL query patterns. 2) Definition of multiple SPARQL query patterns, to determine the performance implications of different query characteristics. 3) Derivation of eight easily applicable recommendations for formulating SPARQL queries.

II. Related Work

Evaluating and optimizing the performance of SPARQL triplestores is not new. Inspired by numerous existing optimizations for relational databases (internal representation, query execution, query optimization, etc.), lots of work was done in improving triple store performance by optimizing the query execution. For example, Weiss et al. [8] propose a sextuple-indexing storage scheme to enhance query processing. Atre et al. [9] focus on a Bitmatrix to optimize Join-Operations in RDF data query processing. Having knowledge about the distribution of the triplestores contained data, heuristics can be used to reorder query patterns [10] to optimize the query execution.

All of these approaches focus on optimizing the triplestore implementation, either by applying automated query optimizations or by optimizing the storage or representation of the RDF triples. Usually this happens in the background without the need of any interaction of the user of the triplestore. In contrast, this work focuses on the user side. Daily work showed that performance characteristics of SPARQL query patterns are widely unknown for triplestore users. Rietveld et al. [11] showed in their evaluation that 72.66% of their analyzed user queries are formulated inefficiently. This is taken into account by the work of Loizou et al. [12]. The authors describe five heuristics for creating performant queries. Although their work is based on a formal evaluation of SPARQL queries, their results are five easily applicable heuristics, namely: 1) minimize optional graph patterns 2) use named graphs to localize SPARQL sub-graph patterns 3) reduce intermediate results 4) reduce the effects of cartesian products 5) specify alternative URIs.

In addition to these heuristics which should be considered, users should keep in mind that there might exist equivalent (or nearly equivalent) SPARQL queries wich are often exchangable in applications. An easy to use guideline, helping to choose the more performant variant is not available until now. To bridge this gap, we are taking a triplestore implementation and evaluate, which SPARQL query patterns are influencing the execution performance. We are aware that the triplestore implementation automatically optimizes the internal execution. Nevertheless, we still expect some aspects a user should be aware of, when formulating queries. These are taken into account for recommendations on formulating SPARQL queries.

III. SPARQL BENCHMARKS

A. Evaluation Criteria for the Review of SPARQL Benchmarks

To find a suitable SPARQL benchmark for our evaluation, we performed a literature review of existing benchmarks. In combination with the work of [13], we then developed a categorizing schema that helped identify a suitable benchmark for this work.

- **User defined ontology**: Is it possible for a user to use an arbitrary ontology in the benchmark?
- **Data generator**: Is there a generator available to generate new triples to easily scale the data set?
- **Query generator**: Is there a tool available, which can dynamically generate queries or is there a fixed set of queries? Are the performed queries statically or dynamically generated?
- **User defined queries**: Is it possible to run user defined queries?
- **Query execution**: Is a query execution driver (running the SPARQL queries on a triplestore) available? Does it return performance metrics?
- **Code availability**: Is the benchmarks source code publicly available?
- **Last update**: Date of last change in the benchmarks source code.
- **License**: Under which license is the source code published?

To make use of an existing benchmark in the context of this work, some of the just mentioned features are mandatory. First of all, the **code** must be available under an appropriate **license**. To scale the data set a **data generator** is needed. Since we want to compare different SPARQL queries, it must be possible to use **user defined queries**. To simplify the usage a **query execution** component is needed. The other features are beneficial though not mandatory.

B. SPARQL Benchmark Selection

To select a suitable benchmark, we started our literature review with the W3C list for RDF store benchmarking [14]. Those benchmarks were evaluated, using the just mentioned criteria. The results are presented in Table I.

For the sake of brevity, only a few benchmarks are introduced in the text. Further information can be found in the sources. The Lehigh University Benchmark (LUBM) [16] offers an ontology about universities. Data scaling is conducted by adding new universities, whereas newly added data has no interconnections with the previous data. The benchmark is highly quoted (1500 direct quotes). The Berlin SPARQL Benchmark [15] is built around an e-commerce system with different products, vendors and consumers and other common information, such has reviews. The benchmark dynamically creates queries during the runtime [29]. The introducing paper is quoted over 650 times. 'SP2Bench: A SPARQL Performance Benchmark' [21] models the behavior of people within a social network with actions such as 'Likes', group management, and befriending persons. The paper is cited nearly 500 times. 'DB-pedia SPARQL Benchmark – PerformanceAssessment with Real Queries on Real Data' [30] created its queries from real-application queries distilled from the dbpedia-log [31] and performs these on the dbpedia data set. To this date, the benchmark nearly reached 300 cites. The Social Intelligence Benchmark (SIB) simulates the social media network of users and their interaction [22]. The project is not supported anymore. IGUANA [28] is the successor of this project. The paper was quoted 60 times to this date.

The *Berlin SPARQL Benchmark BSBM*, *Lehigh University Benchmark LUBM* as well as *LinkBench* fulfill our requirements. For this work, we decided to use the BSBM, since it is newer than the LUBM, but also widely used. Although the BSBM doesn't have a query generator, it implements a query templating engine, which allows to put placeholders in SPARQL queries, which again are substituted during query execution. This allows to generate different queries with the same structure.

IV. QUERY PATTERNS

After a benchmark was selected in the last section, the different SPARQL query patterns, used to derive the recommendations, need to be selected. Subsequently, we characterize and select the patterns for evaluation.

TABLE I. CONSIDERED BENCHMARKS

Name	**User ontology**	**Data generator**	**User queries**	**Query generator**	**Executor**	**Code available**	**Last Update**	**License**
Berlin SPARQL Benchmark BSBM [15]	No	Yes	Yes	No	Yes	✓	2012	Apache 2.0
Lehigh University Benchmark LUBM [16]	No	Yes	Yes	No	Yes	✓	2004	GPL 2.0
FedBench [17]	No	No	No	No	Yes	✓	2013	LGPL
Feasible [18]	No	No	Yes	Yes	No	✓	2018	AGPL
LargeRDFBench [19]	No	No	Yes	No	No	✓	2018	AGPL
University Ontology Benchmark UOBM [20]	No	Yes	No	No	No	✗	2005	GPL 2.0
SPARQL Performance Benchmark [21]	No	Yes	No	No	No	✓	2009	Berkeley License
Social Network Intelligence Benchmark [22]	No	Yes	No	No	Yes	✗	2015	GPL 3.0
Linked Data Integration Benchmark [23]	No	Yes	Yes	No	No	✓	2012	BSD
Linked Open Data Quality Assessment [24]	No	No	No	No	Yes	✓	2012	BSD
LinkBench [25]	Yes	Yes	Yes	No	Yes	✓	2015	Apache 2.0
Waterloo SPARQL Diversity Test Suite [26]	No	Yes	Yes	Yes	No	✓	2014	MIT
Semantic Publishing Benchmark [27]	No	Yes	No	No	Yes	✓	2019	Apache 2.0
IGUANA [28]	Yes	No	Yes	No	Yes	✓	2019	AGPL

A. *Identifying Query Patterns*

We studied the syntactical elements of SPARQL queries and developed variants of query patterns. Those query variants either make use of the SPARQL algebra equivalences or use specific elements of the SPARQL query language. In the first case we are expecting only small differences in performance, since triple store-internal optimizers already make use of semantic equivalences. The second case might show differences, due to the different query results. In some use-cases these different results matter, whereas there are use-cases where only the user's negligence or unawareness causes inperformant SPARQL queries. The results of this work should call the user's attention as well as provide easy usable guidelines for formulating performant queries.

To gather SPARQL patterns, queries used in various past projects were considered. In addition, informal interviews and discussions with users (colleagues, students, etc.) were conducted. This approach showed that the main focus while formulating SPARQL queries is on writing syntactically correct queries returning the right values. Performance impacts were rarely considered. As a result of the discussions, a list of query patterns causing uncertainty in their expected performance characteristic were developed. It is to be noted that the semantics of the compared query patterns might not be completely the same; yet, on the data set they are applied on, their result is expected to be the same.

To determine the influence of these patterns, we formulated two variants of each SPARQL query. Those pairs are used as query templates filled by the BSBM. With the concept of query templates, BSBM allows to use placeholders in a SPARQL query, which are replaced by random values before the query is executed. This allows to slightly change the content of the query, without changing its structure to avoid caching mechanisms in the triple store, which otherwise would tamper our results. Based on the execution times of those variants, we identified eight simple and applicable recommendations.

As an example: the first query variant of *Filter size* looks like this:

```
select ?review ?rating2 where {
    ?review bsbm:rating1 ?rating1.
    ?review bsbm:rating2 ?rating2
    filter (?rating1 >= %rating1% &&
            ?rating2 < %rating2%)
}
```

Listing 1. Variant 1 of *Filter size*

where *%rating1%* and *%rating2%* are placeholders, replaced by BSBM during execution. The performance of this variant is compared with the following one:

```
select ?review ?rating2 where {
    ?review bsbm:rating1 ?rating1.
    filter (?rating1 >= %rating1%)
    ?review bsbm:rating2 ?rating2.
    filter (?rating2 < %rating2%)
}
```

Listing 2. Variant 2 of *Filter size*

B. *Patterns for Evaluation*

We identified the following query patterns, with the described variants:

- **Number of results**: Querying instances with a large amount of instances (1) (Those numbers are used in Section V to identify the variants) or with a low number of instances (2).
- **Limiting results**: Getting all results (1) or limiting the number of results, using the *LIMIT* operator (2).
- **Projection**: Using a projection allows to specify the needed variables. Either selecting all *SELECT* * (1) or only one variable $SELECT\ ?var$ (2), or only the number *SELECT(count(?var))* of results (3).

- **String functions**: Either filtering a variable based on a regular expression (1) or using one of the string functions, e.g., *STRSTARTS* (2).
- **Filter size**: Providing all expressions in one filter term, combined by the logical *AND* (1) or having multiple smaller filter expressions (2).
- **Filter position**: Since a SPARQL query contains a set of triple patterns, the position of the filter statement in this set can be changed: having the filter at the end (1) or at the beginning (2) of the query.
- **String filter**: Filtering for numerical (1) or text based values (2).
- **Inverse**: Specifying the triple forward *?r rev:reviewer ?p* (1) or inverse *?p ^rev:reviewer ?r* (2).
- **Variable types**: The *rdf:type* of subject and object are specified by the predicate's definition, meaning it is implicitly available (1). Therefore, adding this type of information explicitly to the query is worth investigating (2).
- **Optional**: Querying triple patterns usually requires matching all variables. Some of the variables might be optional. There are two possibilities querying optional variables: using the *OPTIONAL* statement (1) or *UNION* the triples with and without the variable (2).
- **Graph structure**: An object of a triple might be either an instance of another type or a primitive data value. By this, a query can filter for instances (querying the RDF graph structure) (1) or for a data value (2).
- **Triple order**: The order of triple patterns in a SPARQL query is arbitrary. In the first variant the first pattern matches a large amount of triples and the second pattern reduces the result (1). The second variant is the opposite; the first triple pattern already limits the result to a small amount (2).
- **Limit in subselect**: SPARQL supports splitting a query into multiple select statements. This enables the user to already *LIMIT* the result of the subselect. In this case we compare selecting products and labels, with limit 5 (1) and subselecting 5 products and then selecting the corresponding labels (2).
- **Distinct**: The *DISTINCT* keyword can be used to eliminate duplicates in the result (1). As a variant the weaker *REDUCED* can be used (which might remove duplicates, but there is no guarantee) (2).
- **Minus**: Using *not exists* allows to filter for triples that do not match (1):
  ?product rdfs:label ?label
  filter(not exists {?product bsbm:number ?n})
  As an alternative the *MINUS* operator allows to remove triple from the result that match a given triple (2):
  ?product rdfs:label ?label
  MINUS {?product bsbm:number ?n}
- **Path**: Querying a path can be either done by explicitly querying the relation (1) or by using a property path sequence, e.g., *^bsbm:reviewFor/rev:reviewer* (2).

Having a basic understanding of triple stores or relational databases in mind, it is clear that some of the variants are faster. This is especially the case when the final or intermediate data set is reduced. Although this is obviously clear, we decided to keep them in our list, on the one hand to quantify the difference and on the other hand to return this to the users mind.

As part of the query execution, a triple store has to parse the SPARQL query into an abstract representation. Usually this is done by an internal optimizer, which makes use of equivalences in the SPARQL algebra to change the queries to an equivalent representation. This is especially expected for the variants of *Filter size*, *Filter position* and *Triple order*. We validated our assumption by parsing our queries with the Apache Jena query parser [32], which is also part of the Fuseki triple store. This showed for *Filter size* that big filters are split into multiple single filters, therefore internally the two variants are processed the same way. Also a comparison of the *Filter position* variants showed that in the internal representation the filters are moved to the same place. However, the *triple order* was not changed. In addition, we noted that querying the *inverse* relation leads back to the forward relation.

V. Evaluation

In our evaluation, we compare the query execution time of the previously described query pattern variants. In the following, we briefly introduce our system set-up before giving the results of the performance analysis.

A. System Set-Up and Experimental Procedure

We executed the queries presented in the previous section using the Berlin SPARQL Benchmark and a Fuseki triple store hosting a BSBM data set. The benchmark as well as Fuseki were running on commodity hardware: Laptop with Intel i7 CPU and 16GB of RAM. As explained in section III, the Berlin SPARQL Benchmark offers the possibility to generate data sets of arbitrary size. In first tests 1.8 million triple showed useful for a smaller data set: not too big, but already showing effects. For a bigger data set, we decided to use 4 million triples, since this still allowed the execution on our hardware in a reasonable time.

First tests showed that the Heap-Space, available for the Fuseki triple store process, is an important factor: the combination of too many triples with a small Heap-Space results in exceptions thrown by the triplestore. 1.1 GB and 2.25 GB, respectively, turned out to be the minimum Heap-Space sizes for our data sets.

For our evaluation, we rely on the available features of the Berlin SPARQL Benchmark (BSBM): BSBM ontology, BSBM data generator and BSBM query execution. The BSBM query execution takes a set of SPARQL query templates. Parameters were replaced with a set of predefined values and sent to a SPARQL endpoint. As a result the individual execution times, together with number of Queries Per Second (QPS) are returned by the benchmark. Our evaluation is based on QPS as an average over multiple queries. In addition to the benchmark, small scripts were developed to ease the execution of the different query variants and to store the results in an ordered manner. To initialize the triplestore correctly (e.g., creating indices, caches, etc.) a warm up phase of 30 queries was introduced for each variant. For the test run, each query variant was executed 150 times.

TABLE II. EVALUATION OF QUERY VARIANTS

Query Pattern	**4 million**		**1.8 million**			
	7 GB	2.45 GB	7 GB	4 GB	1.11 GB	VI
Number of results	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	1
Limiting result	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	1
Projection	$\checkmark_3$	$\checkmark_2$	$\checkmark_3$	$\checkmark_3$	$\checkmark_3$	2
String functions	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	6
Filter size	✗	✗	✗	✗	✗	-
FilterPos	✗	$\checkmark_2$	✗	✗	✗	-
String filter	$\checkmark_1$	$\checkmark_2$	$\checkmark_1$	$\checkmark_1$	$\checkmark_2$	5
Inverse	✗	✗	✗	$\checkmark_1$	✗	-
Variables type	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	4
Optional	✗	✗	$\checkmark_1$	$\checkmark_1$	✗	-
Graph structure	$\checkmark_2$	$\checkmark_2$	✗	✗	✗	-
Triple order	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	3
Limit in subselect	$\checkmark_1$	$\checkmark_1$	✗	✗	✗	-
Distinct	✗	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	7
Minus	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	8
Paths	✗	✗	✗	✗	$\checkmark_2$	-

B. Results

Table II summarizes our evaluation results. Each column shows a configuration of the triplestore (available Heap-Space and number of inserted triples). If there was not a significant difference (10 %) in the execution time of the query variants it is marked with the ✗sign. A ✓indicates a difference, where the subscript points out, which of the variants had the better performance. The last column anticipates the recommendation, presented in the following Section VI.

Unsurprisingly, the basic rule of thumb - limiting the result set size - proved to be true. The query returning less data was faster for all variants: *Number of results*, *Limiting result* and *Projection*. Like described in the previous Section IV there was no difference for *Filter size*. Only one configuration for *FilterPos* showed a difference. These results were expected, due to the same internal representation. There are some patterns where a difference was noted only in one configuration (*FilterPos*, *Inverse*, *Paths*). So, those patterns generally do not have a significant impact on the queries' performance. The example of *FilterPos* (having the same internal representation, but performance difference in one configuration) shows, there are additional (not further analyzed) influencing factors.

There seems to be no big difference between *Optional* and *Union*. In two configurations the use of *Optional* was slightly faster than *Union*. Also filtering a *Graph structure* doesn't show a clear difference. For the larger data set of 4 million triples, filtering for a graph structure was a bit slower than filtering for data values.

As known from relational data bases, filtering for text based values (*String filter*) is slower (although it seems that this effect only appears for larger data sets). If searching in text is needed, then the functions provided by SPARQL should be preferred over generic regular expressions (*String functions*). It also showed that using specially provided functions (like *Reduced* instead of *Distinct* or *Minus* instead of *not exists*) can improve the query performance.

Surprisingly, it is to be noted that providing additional type information (*Variables type*) can slow down the query execution. It seems that adding this type information results in additional checks during the query execution and does not support the query optimization as one might expect.

Regarding the result of the query, the *triple order* is arbitrary. In any case, it is shown that this holds not valid for the performance: more selective triple patterns should be stated first. Unexpectedly, the triples were not automatically reordered during the execution by the triplestore based on heuristics of the contained data. Since reordering triples results in equivalent results and users often have knowledge (or at least some idea) about the selectivity of a triple pattern, they should be careful about the order. Reducing the intermediate result by *Limiting a subselect* does not provide any performance optimization in our tests. Surprisingly, a subselect without a *Limit* was faster in two configurations; we expected the performance impact of a limited result set to be higher, than that of a subselect.

VI. Recommendations

Based on the previously presented evaluation, the following recommendations should be kept in mind when writing SPARQL queries:

1) **Small result set**: If possible, limit the returned result. If the complete result is not processable by the client, make use of *LIMIT* and get the next chunk of data by using *OFFSET*.
2) **Use projections**: Clearly specify the variables of interest and do not select everything (*SELECT* *). If only the number of results is of interest, make use of *COUNT*.
3) **Reduce intermediate results**: If known, list the most selective triple query pattern first.
4) **Do not add additional types**: Do not add *rdf:type* triples if they aren't needed.
5) **Avoid filtering for text**: If possible prefer filtering for numbers instead of text.
6) **Use String-functions**: Prefer to use the SPARQL *STR-functions* instead of regular expressions.
7) **Use reduce**: If duplicates in the result are tolerable, use *REDUCE* instead of *DISTINCT*.
8) **Minus-Operator instead of Filter**: Express your filter expression in a *MINUS* and avoid *FILTER* in conjunction with *not exists*.

VII. Conclusion

The execution time of SPARQL queries often is crucial for applications relying on semantic data stores. Only a few, easily applicable guidelines are available for users writing performant SPARQL queries. In this work, we closed this gap by 1) selecting a SPARQL benchmark, to compare different variants of SPARQL queries 2) extracting common patterns in SPARQL queries and formulating variants to determine their impact on performance 3) providing eight easily applicable recommendations that can be considered while writing SPARQL queries.

In comparison with Loizou's work [12], we can confirm his findings 1) and 3) and expand the suggestions with our findings.

Besides the provided recommendations, the evaluation showed that the results vary from configuration to configuration. There are many factors that influence the execution time of SPARQL queries. Therefore, the presented recommendations can be used as hints and thumb rules to guide users through formulating SPARQL queries, without the need of any specialized knowledge. If the performance of specific queries is crucial for a system, dedicated benchmarking tests needs to be done with the given data set and triplestore implementation. Our extension to the BSBM and benchmark execution can be used to simplify further evaluations.

Future work can extend the evaluation to other triplestore implementations and data sets, as well as the comparison of different triplestores among each other. Notably, an analysis of different query patterns on different triplestores is interesting as well, to find out if our suggestions hold valid on different triplestore implementations. Some of the recommendations change the result of the SPARQL query, whereas it should be noted that some of the tips result in equivalent queries. In future work these recommendations can be implemented into the triplestore's optimizer to automatically transform into more efficient, but semantically equivalent queries.

REFERENCES

[1] World Wide Web Consortium (W3C), "Rdf 1.1 concepts and abstract syntax," 25.02.2014, retrieved 09. 2020. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/

[2] "The Linked Open Data Cloud," retrieved 09. 2020. [Online]. Available: https://lod-cloud.net/

[3] World Wide Web Consortium (W3C) , "Sparql 1.1 overview," 21.03.2013, retrieved 09. 2020. [Online]. Available: https://www.w3.org/TR/sparql11-overview

[4] E. Kontopoulos et al., "Ontology-based representation of crisis management procedures for climate events," in 1st International Workshop on Intelligent Crisis Management Technologies for Climate Events (ICMT 2018), colocated with the 15th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2018), 2018, pp. 1064–1073.

[5] T. Hellmund, M. Schenk, P. Hertweck, and J. Moßgraber, "Employing geospatial semantics and semantic web technologies in natural disaster management," SEMANTICS Posters and Demos, 2019.

[6] P. Hertweck et al., "The backbone of decision support systems: The sensor to decision chain," International Journal of Information Systems for Crisis Response and Management (IJISCRAM), vol. 10, no. 4, 2018, pp. 65–87.

[7] T. Hellmund et al., "Introducing the heracles ontology—semantics for cultural heritage management," Heritage, vol. 1, no. 2, 2018, pp. 377–391.

[8] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," Proceedings of the VLDB Endowment, vol. 1, no. 1, 2008, pp. 1008–1019.

[9] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix bit loaded: a scalable lightweight join query processor for rdf data," in Proceedings of the 19th international conference on World Wide Web, 2010, pp. 41–50.

[10] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "Sparql basic graph pattern optimization using selectivity estimation," in Proceedings of the 17th international conference on World Wide Web, 2008, pp. 595–604.

[11] L. Rietveld and R. Hoekstra, "Yasgui: feeling the pulse of linked data," in International Conference on Knowledge Engineering and Knowledge Management, 2014, pp. 441–452.

[12] A. Loizou, R. Angles, and P. Groth, "On the formulation of performant sparql queries," Journal of Web Semantics, vol. 31, 2015, pp. 1–26.

[13] M. Saleem et al., "How representative is a sparql benchmark? an analysis of rdf triplestore benchmarks," in The World Wide Web Conference, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1623–1633. [Online]. Available: https://doi.org/10.1145/3308558.3313556

[14] World Wide Web Consortium (W3C), "Rdf store benchmarking," 20.10.2018, retrieved 09. 2020. [Online]. Available: https://www.w3.org/wiki/RdfStoreBenchmarking

[15] C. Bizer and A. Schultz, "The berlin sparql benchmark," International Journal on Semantic Web and Information Systems (IJSWIS), vol. 5, no. 2, 2009, pp. 1–24.

[16] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," Journal of Web Semantics, vol. 3, no. 2-3, 2005, pp. 158–182.

[17] M. Schmidt et al., "Fedbench: A benchmark suite for federated semantic data query processing," in International Semantic Web Conference, 2011, pp. 585–600.

[18] M. Saleem, Q. Mehmood, and A.-C. N. Ngomo, "Feasible: A feature-based sparql benchmark generation framework," in International Semantic Web Conference, 2015, pp. 52–69.

[19] M. Saleem, A. Hasnain, and A.-C. N. Ngomo, "Largerdfbench: a billion triples benchmark for sparql endpoint federation," Journal of Web Semantics, vol. 48, 2018, pp. 85–125.

[20] L. Ma et al., "Towards a complete owl ontology benchmark," in European Semantic Web Conference, 2006, pp. 125–139.

[21] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "Spˆ2bench: a sparql performance benchmark," in 2009 IEEE 25th International Conference on Data Engineering, 2009, pp. 222–233.

[22] M.-D. Pham, P. Boncz, and O. Erling, "S3g2: A scalable structure-correlated social graph generator," in Technology Conference on Performance Evaluation and Benchmarking, 2012, pp. 156–172.

[23] C. R. Rivero, A. Schultz, C. Bizer, and D. Ruiz Cortés, "Benchmarking the performance of linked data translation systems," in LDOW 2012: WWW2012 Workshop on Linked Data on the Web (2012), 2012.

[24] P. N. Mendes, H. Mühleisen, and C. Bizer, "Sieve: linked data quality assessment and fusion," in Proceedings of the 2012 Joint EDBT/ICDT Workshops, 2012, pp. 116–123.

[25] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: a database benchmark based on the facebook social graph," in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 1185–1196.

[26] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of rdf data management systems," in International Semantic Web Conference, 2014, pp. 197–212.

[27] V. Kotsev et al., "Benchmarking rdf query engines: The ldbc semantic publishing benchmark," in BLINK@ ISWC, 2016, pp. 1–16.

[28] F. Conrads, J. Lehmann, M. Saleem, M. Morsey, and A.-C. N. Ngomo, "Iguana: a generic framework for benchmarking the read-write performance of triple stores," in International Semantic Web Conference, 2017, pp. 48–65.

[29] C. Bizer and A. Schultz, "Benchmarking the performance of storage systems that expose sparql endpoints," in Proc. 4 th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS), 2008, p. 39.

[30] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo, "Dbpedia sparql benchmark–performance assessment with real queries on real data," in International semantic web conference, 2011, pp. 454–469.

[31] "DBpedia," retrieved 09. 2020. [Online]. Available: https://wiki.dbpedia.org/

[32] "Apache Jena," retrieved 09. 2020. [Online]. Available: https://jena.apache.org/documentation/query/index.html