StarBench: Benchmarking RDF-star Triplestores

Ghadeer Abuoda[1], Christian Aebeloe[1], Daniele Dell'Aglio[1], Arthur Keen[2] and Katja Hose[3,1]

[1]Department of Computer Science, Aalborg University, Aalborg, Denmark
[2]ArangoDB, San Francisco, United States
[3]Institute of Logic and Computation, TU Wien, Vienna, Austria

Abstract

RDF-star has rapidly gained popularity as a way to annotate RDF statements while avoiding the disadvantages of reification. Hence, a number of triplestores supporting this new standard have become available. Yet, it is difficult to assess the performance of these systems and to which degree they support RDF-star and the corresponding SPARQL-star query language. Hence, in this paper, we propose StarBench, a benchmark for testing SPARQL-star support and runtime performance. We ran StarBench on a number of state-of-the-art triplestores with RDF-star and SPARQL-star support and share our findings. Based on these findings, we highlight existing challenges and research opportunities.

Keywords

Benchmark, RDF-star, SPARQL-star

1. Introduction

The Resource Description Framework (RDF) [1] is a widely used W3C standard for representing and exchanging information on the Web, enabling the modeling of various types of relationships between resources. Formally, an RDF graph is a set of statements (s, p, o) describing that two resources s and o are connected by a relationship p. RDF-star [2] then enables the representation of more complex relationships as well as metadata about the statements in the RDF graph, such as provenance [3], knowledge evolution [4], archiving [5, 6], etc. Technically, RDF-star eases the annotation of statements by allowing the s or o resources to be statements as well (*embedded statements*), e.g., $((s, p, o), q, a)$. SPARQL-star [7] is an extension of SPARQL for RDF-star: it enables queries involving multi-edge relationships and recursive relationships. To meet the interest in RDF-star, a range of triplestores supporting this new model have become available.

While there are various initiatives to benchmark RDF triplestores and their query engines, e.g. [8, 9, 10], to the best of our knowledge, the only benchmark that supports RDF-star and SPARQL-star is the RDF Reification (REF) benchmark [11]. As the name suggests, REF was proposed to benchmark solutions for annotating statements, such as RDF reification and RDF-star. However, it does not cover several important aspects. First, REF only supports a subset of SPARQL-star and therefore does not support assessing to which degree the diverse RDF-star

QuWeDa 2023 : 7th Workshop on Storing, Querying and Benchmarking Knowledge Graphs @ ISWC2023
gsmas@cs.aau.dk (G. Abuoda); caebel@cs.aau.dk (C. Aebeloe); dade@cs.aau.dk (D. Dell'Aglio); arthur@arangodb.com (A. Keen); katja.hose@tuwien.ac.at (K. Hose)
0000-0003-4904-2511 (D. Dell'Aglio); 0000-0001-7025-8099 (K. Hose)

and SPARQL-star constructs are supported. For example, embedded patterns in object position or using *Union* or *Optional* between different embedded graph patterns. Moreover, REF only reports runtime performance but does not check if the query answers are correct.

Hence, we propose StarBench, a benchmark aimed at overcoming such limitations, tailored for evaluating and comparing the performance of SPARQL-star query engines. We built StarBench by extending REF: we designed a comprehensive set of queries that account for various specific constructs of RDF-star and SPARQL-star [12], totaling in 56 SPARQL-star queries. The benchmark is available at the URL: https://github.com/dkw-aau/SPARQL-star-Benchmark under open licences (CC-BY 4.0 for the data and ASL 2.0 for the code).

We discuss the effectiveness of StarBench by analyzing the performance of various SPARQL-star query engines, including Apache Jena, Oxigraph, and GraphDB. The insights we gather from our results hint the research and development directions need to fully support RDF-star to efficiently query it.

This paper is structured as follows: we introduce the related work in Section 2 and the background in Section 3. We describe the modification strategies used to generate StarBench queries and how we selected them in Section 4. We describe our experience applying StarBench to existing query engines in Section 5, and we conclude with final remarks in Section 6.

2. Related Work

Ever since the inception of SPARQL, there has been a long-standing tradition of benchmarking SPARQL query engines. This effort is still ongoing with several works on benchmarking SPARQL query engines being published over the past few years [8, 9, 10, 11, 13, 14, 15, 16, 17, 18]. These benchmarks cover various different aspects of SPARQL query engines, such as reasoning [8], federations [13, 14], stress-testing [10, 15], and query processing performance [16]. Furthermore, the Linked Data Benchmarking Council (LDBC) [19, 20] aims to organize management and development of RDF and SPARQL benchmarks in a collaborative environment.

While many SPARQL benchmarking suites focus on different aspects of query engines, they have some commonalities. Most of these benchmarks are community-driven initiatives. Several benchmarks focus on realistic and real-world data and queries [13, 14, 15, 16]. As an example, LargeRDFBench [13] is a benchmarking suite for federated SPARQL query engines: it comprises 13 different and interlinked datasets within various domains, such as general knowledge and biomedical data, and 40 queries of varying complexity and difficulty that access data across numerous of these datasets. Other benchmarks instead choose to provide synthetic datasets or queries [10, 8, 9]. They usually focus on performance-heavy metrics like stress-testing and only secondarily focus on the meaningfulness of the data or the query. For instance, the Waterloo SPARQL Diversity Test Suite (WatDiv) [10] provides data and query generators to create synthetic datasets with unlimited scale factors and different queries based on predefined query templates. Doing so helps assess the performance of query engines under stress for a diversified set of queries.

While plenty of benchmarking suites exist to test various aspects of SPARQL query engines, to the best of our knowledge, there is only one benchmark to assess SPARQL-star capabilities in query engines, called the RDF Reification Benchmark (REF) [11]. REF was initially proposed

to assess reification techniques over SPARQL-star engines. As such, REF provides an RDF-star representation of the Biomedical Knowledge Repository (BKR) dataset [21]. Furthermore, REF includes 12 queries, 7 of which were extracted from the original BKR paper [21] as well as [22]. The remaining 5 queries were created specifically for the REF benchmark. However, most queries in the REF benchmark contain only basic graph patterns with triple patterns in the form (s, p, o) or $((s, p, o), q, x)$, optionally with FILTER clauses.

We argue that we need a more diverse set of queries to comprehensively test the features of SPARQL-star engines. Therefore, when designing StarBench we opted for a hybrid approach: we use a real-world dataset, i.e., the one of the REF, and extend the REF query set with synthetic queries. Such a mixed set of real-world and synthetic queries can be beneficial for experimenting with realistic workloads as well as for testing the correct behavior of the SPARQL-star engines.

It is also worth noting that the goal of StarBench is to test the support of SPARQL-star engines. This is different from the goal of REF and other initiatives like [23], which aimed to evaluate reification strategies and compare them. As such, we believe that StarBench can be a valuable resource for researchers and practitioners focusing on the development and comparison of SPARQL-star query engines.

Finally, regarding the assessment of RDF-star support in RDF triple stores in general, previous studies focused on RDF-star implementations in commercial RDF stores [24]. In this paper, we consider different commercial and open-source RDF triplestores with a more practical, extensive analysis of RDF-star support.

3. Background

An RDF statement is a triple consisting of a subject, a predicate, and an object. The subject is the described resource, the predicate and object are the property-value pair of the resource.

Definition 1 (RDF statement). *Let I, B, and L be the disjoint sets of IRIs, blank nodes, and literals. Let $T = (I \cup B \cup L)$ be the set of* RDF terms. *An* RDF statement *is a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, representing that* subject s *and* object o *are in a relation p (*predicate*). An* RDF graph *is a finite set of RDF statements.*

RDF-star then extends the above definition of RDF statements by allowing subjects and objects to be statements themselves.

Definition 2 (RDF-star statement). *Let $s \in I \cup B$, $p \in I$, $o \in I \cup B \cup L$, then an RDF-star statement is a triple defined recursively as follows:*

- *Any RDF statement (s, p, o) is an RDF-star statement;*
- *Let t and $\bar{t}$ be RDF-star statements. Then, (t, p, o), (s, p, t), and $(t, p, \bar{t})$ are RDF-star statements – also known as* asserted statement. *t and $\bar{t}$ are referred to as* embedded *(or* quoted*) statements.*

SPARQL [25] is the standard language for querying RDF graphs. At its core, there is the notion of triple pattern.

Definition 3 (Triple pattern). *Let V be the set of query variables, infinite and disjoint from the set of RDF terms T. A SPARQL triple pattern conforms to:* $(T \cup V) \times (I \cup V) \times (T \cup V)$.

Triple patterns can then be combined into basic graph patterns (BGPs) via joins on the involved query variables. Additionally, SPARQL queries can be extended with operators, such as, such as left join (OPTIONAL), union (UNION), and selection (FILTER) as well as aggregations and solution modifiers, such as ORDER and LIMIT. For a full description, we refer the reader to the SPARQL standard [25]. The semantics of SPARQL queries are based on multisets (bags) of mappings, i.e., a SPARQL solution mapping μ is a partial function that maps query variables to RDF terms.

SPARQL-star then extends SPARQL to process RDF-star statements. One of the main differences is the definition of triple patterns, which accounts for embedded statements.

Definition 4 (SPARQL-star triple pattern). *A SPARQL-star triple pattern is a triple pattern recursively defined as follows:*

- *Every SPARQL triple pattern is a SPARQL-star triple pattern;*
- *If t and $\bar{t}$ are SPARQL-star triple patterns, x is an RDF term or a query variable, and p is an IRI or a query variable, then (t, p, x), (x, p, t), and $(t, p, \bar{t})$ are SPARQL-star* asserted triple patterns*, and t and $\bar{t}$ are* embedded triple patterns*.*

A SPARQL-star basic graph patterns (BGP-star) correspond to a conjunction of a set of SPARQL-star triple patterns. A SPARQL-star solution mapping μ is a partial function that maps variables to the RDF-star terms, i.e. URIs, blank nodes, literals, and RDF-star triples.

4. StarBench Design

When designing StarBench, we built upon REF [11] for our baseline queries and the dataset. We use the Biomedical Knowledge Repository (BKR) dataset [22], which contains 61,032,567 triples, including approximately 35 million distinct subjects, 8 million distinct objects, and over 33 million distinct predicates.

At its core, StarBench relies on a number of baseline queries (Section 4.1) that are then systematically modified using a well-defined set of *modification strategies* (Section 4.2), which are applied individually or in combination to generate new query variations.

4.1. Baseline Queries

REF contains twelve baseline queries, grouped into three distinct categories: A, B, and F[1]. In the following, we use the identifier format XN to indicate that a query is the Nth query in category X.

[1]The complete list of queries is available in our project repository: https://github.com/dgraux/RDFStarObservatory/tree/master/testSuits/REF-Benchmark

```
SELECT  ?s ?p ?o
WHERE { << ?s ?p ?o >> provenir:derives_from  pubmed:99992-INST }
```

Listing 1: Baseline query A1

```
SELECT  ?source_cl (COUNT(?source_cl) AS ?c)
WHERE {
  ?source_inst rdf:type ?source_cl .
  << meta:C0543467-INST bkr_sn:TREATS ?o >> provenir:derives_from ?source_inst
} GROUP BY ?source_cl
```

Listing 2: Baseline query A3.

Category A includes four queries, for example, the two queries in Listings 1 and 2. These queries are derived from the study conducted by Sahoo et al. [21], and are characterized by the presence of a unique statement with embedded statements. The queries in this category include a specific provenance predicate :derives_from[2].

```
SELECT  ?o1 ?o2 ?pmid2
WHERE {
  << ?o1 bkr_sn:CAUSES ?o2 >> provenir:derives_from ?pmid2 .
  << bkr_sn:C0543467-INST bkr_sn:TREATS ?o1 >> provenir:derives_from pubmed:10979521-INST
}
```

Listing 3: Baseline query B2.

Category B has three queries involving more complex triple patterns. An example is shown in Listing 3 containing two SPARQL-star triple patterns, where the two embedded triple patterns share a common variable.

```
SELECT  ?o1 ?o2 ?pmid2
WHERE {
  << bkr_meta:C0543467-INST bkr_sn:TREATS ?o1 >> provenir:derives_from pubmed:10979521-INST .
  << ?o1 bkr_sn:CAUSES ?o2 >> provenir:derives_from ?pmid2 .
  << ?o2 bkr_sn:AFFECTS ?o3 >> provenir:derives_from ?pmid3 .
}
```

Listing 4: Baseline query B3.

Another example of is the baseline query B3, shown in Listing 4: it contains three triple patterns where the join variables lie in the embedded triple patterns.

```
SELECT  ?o ?source1 ?source2
WHERE {
  << bkr:META_C0040300-INST bkr_sn:PART_OF ?o >>
              provenir:derives_from ?source1 ;
              provenir:derives_from ?source2 .
  FILTER ( str(?source1) > str(?source2) )
}
```

Listing 5: Baseline query F3.

Finally, category F includes five queries, introduced by the REF authors [11]. Such queries are characterized by the presence of FILTER clauses. These queries retrieve asserted statements containing both provenance and temporal annotations. An example query of this category, F3, is presented in Listing 5.

[2]For ease of presentation, we omit prefixes in the examples and the queries in this paper.

4.2. Modification Strategies

In total, we used six modification strategies (M1 through M6) as described below.

M1. RDF-star constructs. This modification strategy changes the SPARQL-star triple patterns in the queries to increase or reduce their complexity. In particular:

- it introduces double nesting of embedded graph patterns, as for example in the StarBench query C1 (Listing 6, Line 2) and
- it positions the embedded triple pattern in the subject or object position of the RDF-star triple, e.g., query C2 in Listing 7 is derived from baseline query A1 (Listing 1) by moving the embedded triple pattern from the subject to the object position.

```
1 SELECT (COUNT(*) as ?Triples)
2 WHERE { «« ?s ?p ?o » ?d ?e » ?t ?u }
```

Listing 6: Query C1: asserted triple pattern as the subject of another asserted triple pattern.

```
1   SELECT (COUNT(*) as ?Triples)
2   WHERE { ?d ?e « ?s ?p ?o » }
```

Listing 7: Query C2: embedded triple pattern in object position.

```
1 SELECT (COUNT(*) as ?Triples)
2 WHERE {
3   << ?o1 bkr_sn:CAUSES ?o2 >> provenir:derives_from ?pmid2
4   << meta:C0543467-INST bkr_sn:TREATS ?o1 >> provenir:derives_from ?pmid1 .
5 }
```

Listing 8: Query P19: the object of the triple pattern in Line 4 is replaced with a variable

M2. Replacing resources with variables. By replacing resources in a query template with variables, we can increase the number of retrieved results. For example, query P19 in Listing 8 has been derived from baseline query B2 in Listing 3; the resource `pubmed:10979521-INST` in Line 4 is replaced with variable `?pmid1`.

M3. FILTER conditions. This modification strategy modifies the query by introducing *FILTER* clauses in different flavors:

- condition on one of the variables of an embedded triple pattern,
- string comparisons with *REGEX* functions, and
- equality and inequality operators, such as <, >, =, between variables.

For instance, query S14 (Listing 9) is derived from baseline query F3 (Listing 5) by adding a filter condition on variable *?o1* of the embedded triple pattern.

```
SELECT (COUNT(*) as ?Triples)
WHERE {
  << ?s1 bkr_sn:PART_OF ?o1 >>
              provenir:derives_from  ?source1 ;
              provenir:derives_from  ?source2 .
  FILTER ( str(?o1a) > str(?s1))
}
```

Listing 9: Query S14: filter on the subject and object variables of the embbeded triple pattern.

M4. Triple pattern addition and removal. This modification strategy transforms a query by removing or adding triple patterns. As a consequence, this modification strategy affects query complexity as well as the number of retrieved results. An example of a query derived through M4 is query P17 in Listing 10, which is derived from baseline query B2 (Listing 3) by removing the triple pattern in Line 3.

```
SELECT (COUNT(*) as ?Triples)
WHERE {
  << ?o1 bkr_sn:CAUSES ?o2 >> provenir:derives_from ?pmid2 .
  << bkr_meta:C0543467-INST bkr_sn:TREATS ?o1 >> provenir:derives_from pubmed:10979521-INST .
}
```

Listing 10: Query P17: query with an asserted triple pattern.

M5. Advanced operators. This modification strategy adds *Optional* and *Union* graph patterns to a query. M5 is useful to both test the parsing capabilities of the query engines and to test their evaluation strategies beyond BGP evaluation.

```
SELECT (COUNT(*) as ?Triples)
WHERE {
  << bkr:META_C0543467-INST bkr_sn:TREATS ?o1 >>                    provenir:derives_from  pubmed
       :10979521-INST
  { << ?o1 bkr_sn:CAUSES ?o2 >> provenir:derives_from  ?pmid2 }
  UNION
  { << ?o2 bkr_sn:AFFECTS ?o3 >> provenir:derives_from  ?pmid3 }
}
```

Listing 11: Query C10: query with *Union*.

An example of a query obtained through this modification strategy is query C10 (Listing 11), derived from the baseline query B3 (Listing 4). By applying M5, a *Union* operator is added between the triple patterns in Lines 4 and 6.

M6. Solution modifiers. This strategy adds or removes solution modifiers, such as *DISTINCT*, *COUNT*, and projected variables in the *SELECT* clause. For example, we used this strategy to add *COUNT* to the queries in Listings 6-11.

4.3. StarBench Queries

We applied the modification strategies to the baseline queries to obtain a new set of 56 queries. We designed this set of queries to cover various SPARQL-star operators. We categorized the

StarBench queries into three categories: plain, selective, and complex, identified by P, S, and C, respectively. We describe them in the remainder of this section.

4.3.1. Category P Queries

StarBench contains 23 plain (P) queries. They are characterized by having a WHERE clause consisting of a basic graph pattern, where each triple pattern has at most one embedded statement as the subject. These queries are usually obtained through modification strategies M2, M4, and M6.

```
1 SELECT DISTINCT (COUNT(*) AS ?Triples)
2 WHERE {
3 << ?s ?p bkr:META_C0339897-INST >> provenir:derives_from ?pm
4 }
```

Listing 12: Query P8: the query replaces the predicate resource with a variable and counts the results.

For example, query P8, shown in Listing 12, is obtained by applying two modification strategies to baseline query A1 (Listing 1). First, by applying M2 (Line 2), the variable in the object position of the embedded triple pattern is replaced with the resource `bkr:META_C0339897-INST`, and the object of the asserted triple pattern is replaced by variable `?pm`. Next, by applying M6 (Line 1), the DISTINCT clause is added, and the aggregation function *Count(*)* replaces the projected variables. Other examples of plain queries are P19 and P17 in Listings 8 and 10.

4.3.2. Category S Queries

StarBench contains 22 S queries. Such queries are structurally similar to the P queries, with the addition of the selection algebraic operator (i.e. the FILTER clause). The queries are usually obtained by applying modification strategies M2, M5, and M6.

```
1 SELECT (COUNT(*) as ?Triples)
2 WHERE {
3     << ?s1 bkr_sn:PART_OF ?o >>
4         provenir:derives_from ?source1 ;
5         provenir:derives_from ?source2.
6     FILTER(str(?source1) > str(?source2)).
7 }
```

Listing 13: Query S11: the query replaces a resource with a variable, and counts the number of results.

Queries S14 (Listing 9) and S11 (Listing 13) are examples of S queries. Query S11 is obtained by applying modification strategies M2 and M6 to baseline query F3 (Listing 5). The former leads to a replacement of the subject of the embedded statement with a variable `?s1` while the latter replaces the projected variables with the *Count(*)* aggregate function.

4.3.3. Category C Queries

StarBench includes 11 complex queries, which may use *Union*, *Optional* and *Group By* operators, or complex SPARQL-star triple patterns, i.e., triple patterns with an embedded triple pattern in the object position, or triple patterns with multiple nested triple patterns.

Examples of C queries are C1, C2, and C10 in Listings 6, 7, and 11, respectively. Listing 14 shows another example of a complex query: query C9 is obtained by applying modification strategies M4 and M6 to baseline query B3 (Listing 4), which add the *Optional* operator (Line 5) and the *Count(*)* aggregate function (Line 1).

```
1 SELECT (COUNT(*) AS ?Triple)
2 WHERE {
3     << bkr_meta:C0543467-INST bkr_sn:TREATS ?o1 >> provenir:derives_from pubmed:10979521-INST .
4     << ?o1 bkr_sn:CAUSES ?o2 >> provenir:derives_from ?pmid2 .
5     OPTIONAL { << ?o2 bkr_sn:AFFECTS ?o3 >> provenir:derives_from ?pmid3 . }
6 }
```

Listing 14: Query C9: query with *Optional* graph pattern

5. StarBench in Action

In this section, we use StarBench to compare six different triplestores with support for RDF-star and SPARQL-star. Our analysis focuses on the following three aspects: (i) support for RDF-star and SPARQL-star in the triplestores, (ii) performance of the queries when evaluated over the triplestores, and (iii) correctness of the obtained query results.

Experimental Setup. We considered the following six triplestores and query engines using StarBench: (1) Apache Jena/TDB2 [26] 4.7.0 exposed via Fuseki, (2) Engine X[3], (3) Oxigraph [27] 0.3.16 via Docker, (4) GraphDB [28] 10.0.2 as a standalone server, (5) AnzoGraph [29] 2.5.16 via Docker, and (6) BlazeGraph [30] 2.1.6 as a standalone Jetty server. However, as we explain in Section 5.1, AnzoGraph and BlazeGraph had issues loading the datasets. Hence, we were able to run all the tests only on the remaining four engines.

We ran these engines on a machine with 16 vCPU cores (AMD 7281) with a clock speed of 2.7 GHz, 512KB L1 cache, 8MB L2 cache, 32MB L3 cache, 256GB RAM, 240GB SSD, and 8TB HDD. We issued the queries from the command line on a different machine with the same specifications as above and on the same network, using cURL to send the HTTP requests. We ran the queries sequentially (and not concurrently) for each engine 3 times and report the averages in this section. We used a timeout value of 30 minutes (1,800 seconds) in our experiments.

The full set of queries provided in StarBench, an overview of the query characteristics and the expected results, as well as the full experimental setup, including scripts to benchmark the considered systems, can be found at: https://relweb.cs.aau.dk/starbench/.

5.1. Support for RDF-star and SPARQL-star

Unfortunately, neither AnzoGraph nor BlazeGraph was able to load the dataset into their native datastores successfully, and we were thus not able to run StarBench on either of them due to parsing errors. According to the AnzoGraph documentation[4], there is a limit on the allowed number of property values per edge (255 properties to be exact). As such, when loading the

[3]Engine X is a commercial engine, and the company behind it preferred to obfuscate the name in this article.
[4]https://docs.cambridgesemantics.com/anzograph/v2.5/userdoc/lpgs.htm#insert-properties

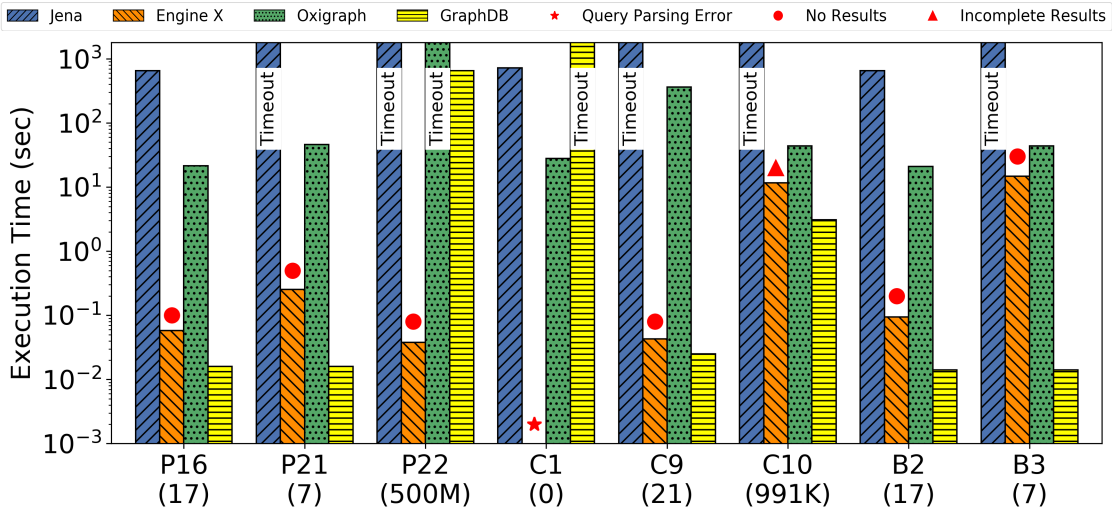

Figure 1: Execution time in seconds (log scale) for queries with inconsistent results. P16 (17) indicates that query P16 has 17 expected results.

dataset, we obtained a "`Element larger than allowed - too many properties`" error, which prevents the dataset from being loaded. BlazeGraph, on the other hand, threw a parsing error when trying to load nested statements, meaning that the parser was not able to parse RDF-star documents, and the support for RDF-star in BlazeGraph is severely limited. Due to the errors explained above, we omit AnzoGraph and BlazeGraph from the remainder of this section. However, in the future, we plan to investigate these issues further and adapt the dataset so that AnzoGraph and BlazeGraph can successfully load the data in order to test their SPARQL-star capabilities. The remaining query engines were all able to parse and load the data without any issues.

Another issue we came across was that, during query execution, Engine X raised parsing exceptions when attempting to process query C1 (Listing 6). We observe that the peculiarity of this query is the double nesting, i.e., the triple pattern `«« ?s ?p ?o » ?d ?e » ?t ?u»`, which includes a statement that is both embedded and asserted. This suggests that the query parser in Engine X is not able to accommodate such cases with double nesting. The remaining queries were all parsed successfully by Engine X, and all other query engines were able to parse all the queries as well successfully.

5.2. Correctness of the query answers

In this section, we report our findings on result completeness and correctness and provide hypotheses for these inconsistent or missing results. Figure 1 shows the execution times of queries for which some systems report inconsistent results. Besides the parsing error of query C1, Engine X does not return results for queries P16, P21, P22, C9, B2, and B3, as well as incomplete results for query C10.

As an example of a query with missing results, consider query P22 (Listing 15), derived from

the baseline query B3 (Listing 4). For this particular query, Jena and Oxigraph time out, while ENGINE X returns no results, leaving GraphDB as the only system able to successfully answer the query within the timeout (Figure 1), albeit still taking more than 10 minutes to do so ($\sim$ 657 seconds). Furthermore, GraphDB answers all the queries in Figure 1 (except C1) more efficiently than the other query engines. We suspect this is due to the query featuring two embedded triple patterns that are joined on the common variable `?o1` (Listing 15); evidently, GraphDB is able to handle such joins between embedded triple patterns more efficiently than the other systems. Moreover, the most likely reason for the general slow performance of all query engines for query P22 is the large number of resulting bindings (more than 500 million results).

```
1 SELECT (COUNT(*) AS ?Triples)
2 WHERE {
3     << bkr_meta:C0543467-INST bkr_sn:TREATS ?o1 >>
4         provenir:derives_from pubmed:10979521-INST .
5     << ?o1 bkr_sn:CAUSES ?o2 >> provenir:derives_from ?pmid2 .
6     ?t provenir:derives_from ?pmid3 .
7 }
```

Listing 15: Query P22

Regarding the missing results for ENGINE X, we hypothesize that ENGINE X is not able to process the join between embedded triple patterns (as described above). In fact, to further investigate this behavior, we executed two additional queries: (1) without the triple pattern at Line 3, and (2) without the triple pattern at Line 5. ENGINE X was able to return the expected results in these cases. This is similar to other queries (mentioned above) with the same join, i.e., P16, P21, and C9, as well as B2 and B3.

Last, for query C10 (Listing 11), ENGINE X returned only $991,875$ results, while the expected number of results is $991,892$, i.e., ENGINE X is missing 17 results. Query C10 features the same join between embedded triple patterns as query P22 (Listing 15), but the second triple pattern is included in a UNION statement. Therefore, we hypothesize that the missing 17 results are the results of that particular join, which corresponds to the 17 results of query P16, whereas the $991,875$ results that ENGINE X returns are the results of the other part of the UNION statement (which does not feature a join on the `?o1` variable).

5.3. Queries timing out

The execution of 26 queries (out of 56) raised timeouts for at least one of the query engines. Table 1 shows an overview of these 26 queries and their performance for each query engine. Crucially, we observe that all query engines fail to answer at least one query within the 30-minute timeout threshold. From a quantitative perspective, ENGINE X and GraphDB perform the best with only a single query timing out for each of the systems (query P4 for ENGINE X and query C1 for GraphDB). Oxigraph follows with four timeouts, and Jena times out on 24 queries.

While Jena experiences a large number of timeouts, most of them are in the S query group. In fact, Jena is able to answer most P and C queries within the timeout threshold. S queries include one or more FILTER clauses (e.g., query S14 in Listing 9), and Jena times out for 17 out of the 22 queries in the S query group. These numbers show that Jena is unable to efficiently process SPARQL-star queries with FILTER clauses.

Table 1
Execution time in milliseconds for each query engine over all the queries that timed out for at least one query engine including timeout (*TO*), Query Parsing Error (★), no results (●), and incomplete results (▲). Bold numbers denote the fastest execution time

Query	**Jena**	ENGINE X	**Oxigraph**	**GraphDB**	*Query*	**Jena**	ENGINE X	**Oxigraph**	**GraphDB**
P4	1,328,823ms	*TO*	428,272ms	**404,736ms**	*S11*	*TO*	**635,389ms**	*TO*	1,366,135ms
P7	*TO*	433,702ms	351,525ms	**289,847ms**	*S12*	*TO*	**12,749ms**	227,221ms	21,577ms
P19	*TO*	2,562ms	*TO*	**1,317ms**	*S13*	*TO*	**10,476ms**	229,623ms	14,907ms
P21	*TO*	●	46,065ms	**16ms**	*S14*	*TO*	**138,510ms**	*TO*	441,293ms
P22	*TO*	●	*TO*	**657,441ms**	*S17*	*TO*	5,625ms	32,035ms	**1,961ms**
S3	*TO*	403ms	18,197ms	**174ms**	*S18*	*TO*	7,724ms	34,275ms	**1,900ms**
S4	*TO*	86ms	17,804ms	**70ms**	*S20*	*TO*	**162ms**	21,850ms	166ms
S5	*TO*	25,393ms	28,345ms	**9,650ms**	*S21*	*TO*	12,367ms	65,076ms	**7,233ms**
S6	*TO*	7,410ms	19,519ms	**5,816ms**	*S22*	*TO*	11,302ms	62,449ms	**4,562ms**
S7	*TO*	**7,957ms**	28,564ms	9,547ms	*C1*	719,331ms	★	**28,027ms**	*TO*
S8	*TO*	**11,884ms**	37,153ms	11,933ms	*C9*	*TO*	●	361,470ms	**25ms**
S9	*TO*	10,759ms	38,827ms	**10,060ms**	*C10*	*TO*	▲	43,895ms	**3,104ms**
S10	*TO*	**14,067ms**	241,406ms	22,089ms	*C11*	*TO*	7,534ms	44,122ms	**4,416ms**

We further notice that GraphDB seems to generally have the best performance over the queries that time out, with the fastest execution time for 17 of the 26 queries. Furthermore, as mentioned in Section 5.2, GraphDB was able to answer query P22 in just over 10 minutes, a query that no other system was able to successfully answer due to a large number of results (more than 500 million). Second is ENGINE X, with the best performance for 8 of the 26 queries, while Oxigraph has the best performance for query C1, which corresponds to the query that times out for GraphDB and for which ENGINE X has a parsing error. Jena does not have better performance for any of the queries in StarBench.

The queries for which Oxigraph times out generally are queries with multiple embedded triple patterns that are joined (e.g., query P19 in Listing 8), some of which are compared with FILTERs, e.g., query S14 in Listing 9. If not properly optimized, evaluating such queries could lead to a high number of intermediate results with a subsequent high number of filter evaluations or joins. GraphDB and ENGINE X generally seem to implement optimization techniques that accommodate such queries since they show increased performance for those queries.

5.4. Query Performance

As discussed in Section 5.3, GraphDB and ENGINE X generally show better performance than Jena and Oxigraph, where Jena seems to have the worst performance of all the query engines on StarBench. This is also the case for most of the queries that do not time out for any engine. Figure 2 shows the execution time of queries P1-P12 over each engine. With a few exceptions, GraphDB and ENGINE X have significantly better performance than both Oxigraph and Jena. For some queries, like query P10, GraphDB and ENGINE X outperform Oxigraph up to three orders of magnitude and Jena up to 5 orders of magnitude. These queries generally are the queries in the P query group with a low number of results (less than 51000).

We also observe that the engines can behave quite differently on different queries. For instance, while GraphDB and ENGINE X generally are more performant than Oxigraph, Oxigraph outperforms the other engines in query P2. This particular query contains an embedded triple

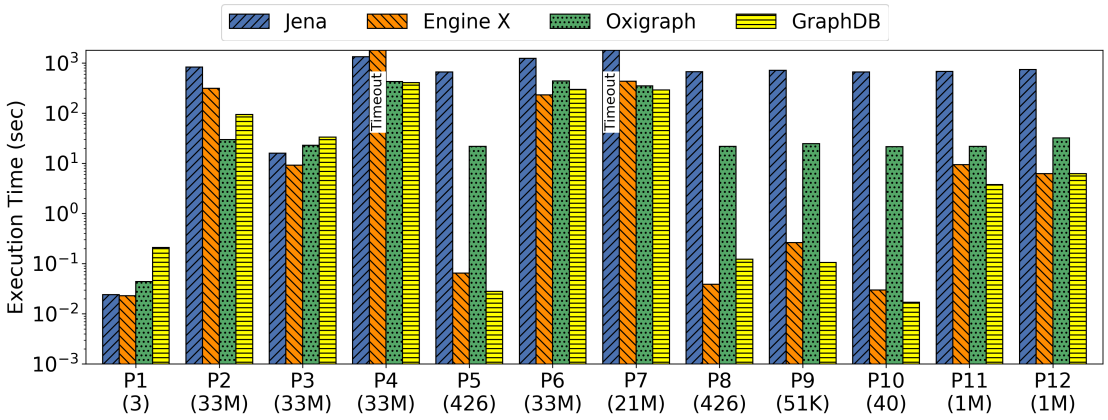

Figure 2: Execution time in seconds (log scale) for queries P1-P12. P1 (3) indicates that query P1 returns 3 results.

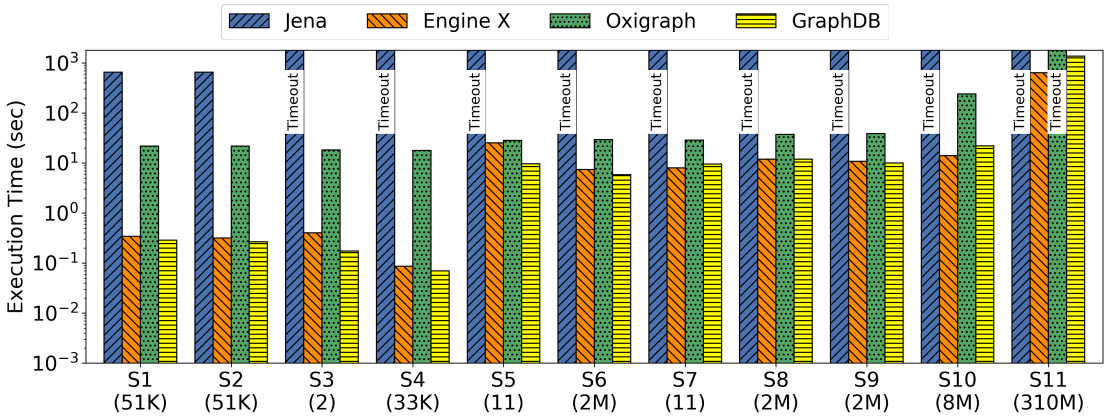

Figure 3: Execution time in seconds (log scale) for queries S1-S11. S1 (51K) indicates that query S1 returns 51 thousand results.

pattern with all variables, i.e., « `?s ?p ?o` », while query P10 contains some bindings within the embedded triple pattern. Nevertheless, Oxigraph has very similar performance for the two queries (30 seconds for P2 and 22 seconds for P10), while GraphDB has much better performance for the query with bindings in the embedded triple pattern (94 seconds for P2 and just 17 milliseconds for P10). This shows that Engine X and GraphDB are able to take advantage of given bindings in the embedded triple patterns, while Oxigraph and Jena are not able to do so. The above observations also hold for queries P13-P23; however, due to space restrictions, we omit details in this paper. They can be found on our website though.

As discussed in Section 5.3, Jena is unable to efficiently process queries with `FILTER` operations. This is in line with the results shown in Figure 3 showing the execution times of queries S1-S11, where Jena has the worst performance for all queries, even the two it does not time out for (S1 and S2). Oxigraph also times out for query S11; however, this is expected given the large number of results (310 million) and the fact that both GraphDB and Engine X struggle with this

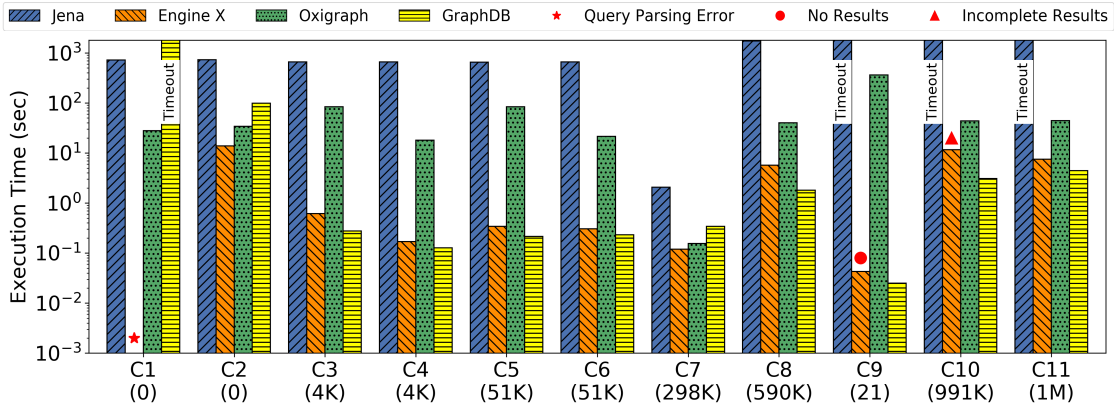

Figure 4: Execution time in seconds (log scale) for queries C1-C11. C3 (4K) indicates that query C3 returns 4 thousand results.

query as well (1,366 seconds for GraphDB and 635 seconds for Engine X). Furthermore, in line with the results for the P query group, GraphDB and Engine X show very similar performance for most of the queries in the S query group, as well as significantly better performance than Oxigraph and Jena.

Figure 4 shows the execution time of queries C1-C11. Similar to the previously recorded results, GraphDB and Engine X generally have better performance for most of the queries than Oxigraph and Jena. However, we also note that Engine X was unable to fully execute three of the queries in the C query group, as discussed in Section 5.2. Furthermore, we note that in the particular case of query C1, GraphDB actually times out as the only query engine. As discussed in Section 5.1, this particular query contains a double-nested triple pattern.

In summary, the experiments suggest that, generally, GraphDB and Engine X are able to process SPARQL-star queries more efficiently than Oxigraph and Jena. In most cases, this difference is quite significant; for instance, query P10 leads to up to three orders of magnitude faster query execution times for Engine X and GraphDB than Oxigraph, and up to five orders of magnitude faster query execution times than Jena. Nevertheless, we also reiterate the fact that Engine X is unable to process most queries with joins between embedded triple patterns, and throws a parsing error for queries with double-nested triple patterns. Our experimental analysis further shows that Jena generally has problems processing FILTER clauses efficiently. Finally, our analysis shows that GraphDB is able to successfully process all queries in StarBench except query C1 because of the double-nested triple pattern. Overall, our experimental analysis shows that StarBench is effective in comparing and contrasting different RDF-star and SPARQL-star query engines and that it can effectively highlight issues in the engines, such as missing support for certain SPARQL-star features.

6. Conclusion and Future Work

In this paper, we presented StarBench, a SPARQL-star benchmark for assessing the capabilities and support of triplestores for RDF-star. StarBench is built on top of the RDF-star representation

of the BKR dataset and the REF benchmark. We applied modification strategies to generate 56 variations from the initial baseline queries.

We applied StarBench to four triplestores: Jena, Engine X, Oxigraph, GraphDB. Our analysis highlighted limitations in loading the data, query parsing, correct evaluation of SPARQL-star queries, as well as contrasting query execution performance across all the engines. These results suggest that StarBench is effective in comparing and contrasting existing engines.

In future work, we plan to continue extending StarBench. A direction we envision is to extend the tested constructs to include SPARQL and SPARQL-star features, such as entailment regimes and subqueries, as well as incorporating other RDF-star datasets by generating RDF-star representations of publicly available RDF datasets. Another direction is the consolidation of the modification strategies in query templates. Such templates can later be used to automatically generate different query loads that fit the needs of the StarBench users. Similarly, we aim to use different datasets, especially the ones with RDF-star statements having more complex structures than the ones of REF.

Acknowledgments

This research is partially funded by the Independent Research Fund Denmark (DFF) under grant agreement no. DFF-8048-00051B and the Poul Due Jensen Foundation. We thank the Ontotext team for the fruitful exchange while preparing the camera-ready version of the article.

References

[1] R. Cyganiak, D. Wood, M. Lanthaler, RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation, W3C, 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

[2] O. Hartig, Foundations of RDF* and SPARQL* (An Alternative Approach to Statement-Level Metadata in RDF), in: AMW, 2017.

[3] E. R. Hansen, M. Lissandrini, A. Ghose, S. Løkke, C. Thomsen, K. Hose, Transparent Integration and Sharing of Life Cycle Sustainability Data with Provenance, in: ISWC, 2020, pp. 378–394.

[4] K. Hose, Knowledge Graph (R)Evolution and the Web of Data, in: MEPDaW@ISWC, 2021, pp. 1–7.

[5] O. Pelgrin, R. Taelman, L. Galárraga, K. Hose, Scaling Large RDF Archives To Very Long Histories, in: ICSC, 2023, pp. 41–48.

[6] O. Pelgrin, L. Galárraga, K. Hose, Towards fully-fledged archiving for RDF datasets, Semantic Web Journal 12 (2021) 903–925.

[7] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Trans. Database Syst. 34 (2009) 16:1–16:45.

[8] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, J. Web Sem. 3 (2005) 158–182.

[9] M. Schmidt, T. Hornung, G. Lausen, C. Pinkel, SP2Bench: A SPARQL performance benchmark, in: ICDE, 2009, pp. 222–233.

[10] G. Aluç, O. Hartig, M. T. Özsu, K. Daudjee, Diversified stress testing of RDF data management systems, in: ISWC, 2014, pp. 197–212.
[11] F. Orlandi, D. Graux, D. O'Sullivan, Benchmarking RDF Metadata Representations: Reification, Singleton Property and RDF, in: ICSC, 2021, pp. 233–240.
[12] G. Abuoda, D. Dell'Aglio, A. Keen, K. Hose, Transforming RDF-star to Property Graphs: A Preliminary Analysis of Transformation Approaches, in: QuWeDa, 2022, pp. 17–32.
[13] M. Saleem, A. Hasnain, A. N. Ngomo, LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation, J. Web Semant. 48 (2018) 85–125.
[14] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, T. Tran, FedBench: A Benchmark Suite for Federated Semantic Data Query Processing, in: ISWC, 2011, pp. 585–600.
[15] C. Stadler, M. Saleem, Q. Mehmood, C. Buil-Aranda, M. Dumontier, A. Hogan, A.-C. Ngonga Ngomo, LSQ 2.0: A linked dataset of SPARQL query logs, Semantic Web (2022).
[16] C. Bizer, A. Schultz, The berlin SPARQL benchmark, Int. J. Semantic Web Inf. Syst. 5 (2009) 1–24.
[17] S. Duan, A. Kementsietsidis, K. Srinivas, O. Udrea, Apples and oranges: a comparison of RDF benchmarks and real RDF datasets, in: SIGMOD, 2011, pp. 145–156.
[18] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, A. N. Ngomo, How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks, in: WWW, 2019, pp. 1623–1633.
[19] R. Angles, P. A. Boncz, J. L. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martínez-Bazan, V. Kotsev, I. Toma, The linked data benchmark council: a graph and RDF industry benchmarking effort, SIGMOD Rec. 43 (2014) 27–31.
[20] P. A. Boncz, LDBC: benchmarks for graph and RDF data management, in: B. C. Desai, J. L. Larriba-Pey, J. Bernardino (Eds.), IDEAS, 2013, pp. 1–2.
[21] S. S. Sahoo, O. Bodenreider, P. Hitzler, A. Sheth, K. Thirunarayan, Provenance Context Entity (PaCE): Scalable provenance tracking for scientific RDF data, in: SSDBM, 2010, pp. 461–470.
[22] V. Nguyen, O. Bodenreider, A. Sheth, Don't like RDF reification? Making statements about statements using singleton property, in: WWW, 2014, pp. 759–770.
[23] J. Frey, K. Müller, S. Hellmann, E. Rahm, M. Vidal, Evaluation of metadata representations in RDF stores, Semantic Web 10 (2019) 205–229.
[24] F. Orlandi, D. Graux, D. O'Sullivan, How many stars do you see in this constellation?, in: ESWC 2020 Satellite Events, 2020, pp. 175–180.
[25] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, W3C Recommendation, W3C, 2013. URL: https://www.w3.org/TR/sparql11-query/.
[26] Apache Software Foundation, Apache Jena, 2023. URL: https://jena.apache.org/, Accessed July 24 2023.
[27] T. Pellissier Tanon, Oxigraph, 2023. URL: https://doi.org/10.5281/zenodo.7669346. doi:10.5281/zenodo.7669346.
[28] Ontotext, Graphdb, 2023. URL: https://graphdb.ontotext.com/, Accessed July 24 2023.
[29] Cambridge Semantics, AnzoGraph, 2023. URL: https://cambridgesemantics.com/anzograph/, Accessed July 24 2023.
[30] B. Thompson, M. Personick, M. Cutcher, The bigdata® rdf graph database, in: Linked Data Management, Chapman and Hall/CRC, 2016, pp. 221–266.