Sparqloscope: A generic benchmark for the comprehensive and concise performance evaluation of SPARQL engines

Hannah Bast[0000-0003-1213-6776], Johannes Kalmbach[0000-0002-5582-1610], Robin Textor-Falconi[0009-0004-8164-5181], and Christoph Ullinger[0009-0000-1534-4664]

University of Freiburg, Freiburg im Breisgau, Germany
`{bast,kalmbach,textorr,ullingec}@cs.uni-freiburg.de`

Abstract. We provide a new benchmark, called Sparqloscope, for evaluating the query performance of SPARQL engines. The benchmark combines three unique features, which separates it from other benchmarks:

1. Sparqloscope is generic in that it can be applied to any given RDF dataset and it will then produce a comprehensive benchmark for that particular dataset. Existing benchmarks are either synthetic, designed for a fixed dataset, or require a query log.
2. Sparqloscope is comprehensive in that it considers most features of the SPARQL 1.1 query language that are relevant in practice. In particular, it considers advanced features like `EXISTS`, and various SPARQL functions for numerical values, strings, dates, language filters, etc.
3. Sparqloscope is specific in that it aims to evaluate relevant features in isolation and as concisely as possible. The benchmark generated for a given knowledge graph consists of only around 100 very carefully crafted queries, the results of which can and should be studied individually and not in aggregation.

Sparqloscope is free and open-source software and easy to use. As a showcase, we use it to evaluate the performance of six SPARQL engines (QLever, Virtuoso, MillenniumDB, GraphDB, Blazegraph, Jena) on two widely used RDF datasets (DBLP and Wikidata). The full materials and more are provided on `https://purl.org/ad-freiburg/sparqloscope`.

Keywords: Benchmark · SPARQL Engines · Performance Evaluation

1 Introduction

Recent years have seen a surge in the development of new SPARQL engines[1], as well as significant progress in existing ones. A recent survey lists and describes over 100 such engines [2], and that list is far from complete. The query processing performance of these engines varies widely, and a question of great practical

[1] By *SPARQL engine* we mean a system that can read and store RDF data and then process SPARQL queries on it; also called *RDF database* or *triplestore*.

relevance is which engine performs how well in this respect. This is a surprisingly hard question to answer, even for experts [16].

In the following, we first survey a selection of existing benchmarks and then critically discuss their strengths and weaknesses. We then describe our new benchmark and how it tries to address these weaknesses.

1.1 Existing benchmarks

The first three benchmarks we discuss all have the following form: they generate synthetic data of arbitrary given size, for a given manually constructed ontology, and have a small fixed set of queries on this data. The *Lehigh University Benchmark (LUBM)* [12] was originally developed for OWL reasoners, but quickly became the first widely adopted benchmark for the performance testing of SPARQL engines. It uses a simple university-themed ontology and has 14 fixed queries, all of which consist of simple triple patterns, that is, each of them can be computed by a sequence of join operations. The *SP2Bench* benchmark [22] uses a more complex ontology (based on the *DBLP* dataset [1] at that time) and has 12 fixed queries, which are more complex than those of LUBM and cover SPARQL constructs such as `OPTIONAL`, `FILTER`, `ORDER BY`, `DISTINCT`, `UNION`, `LIMIT`, and `OFFSET`. The *Berlin SPARQL benchmark (BSBM)* [7] uses an ontology from an enterprise setting, and 12 queries of varying complexity, which cover `FILTER`, `OPTIONAL`, `LIMIT`, `ORDER BY`, `DISTINCT`, `REGEX`, `UNION`, `DESCRIBE`, and `CONSTRUCT`. The queries use randomly sampled values from the dataset.

The *Waterloo SPARQL Diversity Test Suite (WatDiv)* [3] also generates synthetic data of arbitrary given size, but has a much larger set of queries. The focus is on basic graph patterns, with and without filters. The queries are generated using a template-based approach with different profiles regarding the number of triples patterns, join vertices, and join vertex degree. For their evaluation, the authors generate 12,500 queries from 125 templates.

The next three benchmarks we discuss all operate on *Wikidata*, which has emerged as the most widely used knowledge graph for general-purpose information about the world. The *Wikidata Graph Pattern Benchmark (WGPB)* [13] consists of 850 queries on the so-called *truthy* subset of Wikidata.[2] Each query involves two, three, or four triple patterns, for each of which the predicate is fixed and the subject and object are free variables. Computing the query therefore requires one, two, or three join operations. The *WDBench* benchmark [4] is a follow-up of WGPB and features five sets of queries: 280 queries with a single triple pattern each, 681 queries with two or more triple patterns, 498 queries with two or more triple patterns involving at least one `OPTIONAL` join, 660 queries with a single pattern each with a property path of varying complexity, and 539 queries involving a mix of triple patterns and property paths. The *Wikidata query service example queries (WDQS)* [6] benchmark provides a set of 298 example queries from the official Wikidata Query Service at that time. The benchmark focuses on queries using the core SPARQL 1.1 features

[2] About 40% of the complete dataset, see Section 3 for details.

(excluding, for example, queries using GeoSPARQL functions). Many queries involve language filters of the form `FILTER(LANG(?label) = "en")`, which are very common for Wikidata and potentially expensive to compute.

The *FEASIBLE* benchmark [18] is based on query logs. Queries are not taken literally from the log, but analyzed for frequently used predicates and other patterns, which are then used to construct the benchmark queries. This is similar in spirit to our benchmark, but we do not require a query log, and are therefore also not restricted by the quality of the query log. Their benchmark covers `SELECT`, `ASK`, `CONSTRUCT`, and `DESCRIBE` queries, involving `UNION`, `DISTINCT`, `ORDER BY`, `REGEX`, `LIMIT`, `OFFSET`, `OPTIONAL`, `FILTER`, and `GROUP BY`.

There are a multitude of other benchmarks that are either very similar to one of the benchmarks discussed above or less related to the work in this paper. Notable examples are: *FedBench* [21] (federated queries, mix of real and synthetic data), *LargeRDFBench* [17] (federated queries, more than one billion triples of real data), *FedShop* [10] (federated query scaling, synthetic data), *SPARQL Query Generator* [9] (OWL semantics, queries randomly generated from ontologies), *SQCFramework* [19] (query containment, based on a query log), *DBpedia SPARQL Benchmark* [14] (real queries on DBpedia, distilled from query logs), *SRBench* [26] (for streaming RDF/SPARQL, real data), and *ParlBench* [23] (10 hand-crafted queries on Dutch parliament data).

A meta-analysis of 11 selected SPARQL benchmarks can be found in [20].

1.2 Critical discussion of these benchmarks

All benchmarks previous to this work were either based on a *fixed real-world* dataset, or they generate *synthetic data of different sizes*. The exception was *FEASIBLE*, which can work with any RDF dataset, but requires a query log for query generation. Our goal is to get the best from these worlds and design a tool that can be used to generate a benchmark for *any* knowledge graph, given only the RDF data. We discuss the benefits of this approach in Section 1.3.

None of the previous benchmarks comes close to covering the complete set of SPARQL features. For example, with the exception of the WDQS example queries, none of them covers `EXISTS`, triple patterns with three free variables, language filters, and most SPARQL functions. In the words of [20], "Synthetic benchmarks often fail to contain important SPARQL clauses." Our goal is a benchmark that covers most of the features that are relevant in practice.

None of the previous benchmarks systematically tested individual SPARQL features in isolation, or only to a limited extent. To clarify what we mean by this, consider the following query, which computes all persons in Wikidata with their English label, ordered by the number of pages linking to them (`PREFIX` declarations omitted for brevity):

```
SELECT ?person ?label ?sitelinks WHERE {
  ?person wdt:P31/wdt:P279* wd:Q5 .
  ?person ^schema:about/wikibase:sitelinks ?sitelinks .
  ?person rdfs:label ?label . FILTER (LANG(?label) = "en")
} ORDER BY DESC(?sitelinks)
```

This is a typical query for Wikidata, and most SPARQL engines have trouble computing it efficiently. The question is, why are they having trouble. Is it because of one of the property paths? Or because of the large `rdfs:label` predicate and the language filter? Or because of the `ORDER BY`? Or because of the time needed to materialize the large result of almost 7 million rows with three columns each? This requires a careful query analysis with usually much effort (depending on the engine's support for such analyses). Our goal is to simplify this by having a dedicated query for each practically relevant feature.

Finally, the more thorough of the previous benchmarks have hundreds or even thousands of queries. An evaluation then has to resort to aggregate measures like means. Our goal is a moderate number of queries, which can be inspected individually. For example, in order to evaluate how well an engine can join two large predicates with a small result, there is no need to run dozens or even hundreds of queries; a single carefully crafted query is enough. That way, individual strengths or weaknesses of an engine can be pinpointed very effectively.

1.3 Contributions

We provide *Sparqloscope*, a benchmark generator with the following properties:

Sparqloscope is generic in the sense that it can produce a benchmark for an arbitrary given RDF dataset. This has three advantages. First, it naturally allows performance evaluations at different scales (by choosing datasets of different sizes). Second, the input can be real-world as well as synthetic data. Third, it provides an easy way to evaluate the suitability of a particular SPARQL engine for a new dataset. The generation process is described in detail in Section 2.

Sparqloscope is comprehensive in that it covers a large subset of features of the SPARQL 1.1 query language relevant in practical applications. In particular, Sparqloscope includes all the features from the benchmarks discussed in Section 1.1 and more, for example: `EXISTS`, triple patterns with three variables, SPARQL functions operating on numeric values or strings, and language filters. The coverage is not complete yet (see Section 1.4), but we designed Sparqloscope to be very easy to extend and we expect it to grow over time.

Sparqloscope is specific in the sense that it aims to evaluate features in isolation and as concisely as possible. By isolation we mean that each query measures the performance of only one particular feature (as much as possible). That way, when an engine is particularly slow or fast on a query, it is clear that it is due to its implementation for that particular feature. By concise we mean that our benchmark consists of only around 100 very carefully crafted queries, which can and should be studied individually. This becomes clear in Section 2.

Sparqloscope is easy to use and automatic. For a given benchmark, all that needs to be done is set up a SPARQL endpoint and then run the Sparqloscope generator, which is a Python script. For a large dataset, the first stage can take considerable time; see Section 2.1. However, Sparqloscope caches these results by default, so that subsequent runs of the benchmark generation are much faster.

Sparqloscope is free and open-source software and can be accessed under `https://purl.org/ad-freiburg/sparqloscope`. Along with the generator script and detailed instructions, we also provide ready-to-use benchmarks for a selection of widely used knowledge graphs, including those from our evaluation in Section 3. Further benchmarks can be provided upon request.

Finally, we use Sparqloscope to evaluate six SPARQL engines (QLever, Virtuoso, MillenniumDB, GraphDB, Blazegraph, Jena) on two benchmarks generated with Sparqloscope for DBLP (∼500 M triples) and Wikidata Truthy (∼8 B triples). This is only meant as a showcase and not as a complete performance evaluation. The point is to demonstrate that Sparqloscope provides interesting insights that would be much harder to obtain otherwise; see the discussion in Section 3.1.

1.4 Limitations

Sparqloscope does not cover the complete SPARQL 1.1 standard yet, but its coverage is significantly larger than for existing benchmarks and we designed it to be very easy to extend. Here is a list of the current limitations:

1. Sparqloscope currently only generates `SELECT` queries. It could be easily extended to also support `ASK` queries (which basically are `SELECT` queries with a `LIMIT` of 1) and `CONSTRUCT` queries (which are `SELECT` queries followed by some export logic). Note that benchmarking `DESCRIBE` queries is only moderately meaningful, since their semantics are not precisely defined by the standard and are interpreted very differently by different engines.
2. Sparqloscope does not yet evaluate `FROM [NAMED]` and `GRAPH` clauses. This will be added in the future. Note that many datasets (including DBLP and Wikidata Truthy from our evaluation in Section 3) do *not* make use of named graphs.
3. Sparqloscope evaluates the quality and speed of *query planning* only to a limited extent. This is a direct consequence of its feature-isolation property, which leads to syntactically simple queries that are expensive to compute but easy to plan. However, Sparqloscope does cover a variety of query planning challenges, like many triple patterns or `UNION` followed by join; see Section 2.3.
4. Sparqloscope does not evaluate the performance of update operations, which would require a significantly more complex benchmark infrastructure. In particular, such an evaluation would have to consider both the time for the update operations and the effect they have on the internal index data structures of the engine and, thus, subsequent queries. We consider this out of scope [sic] for now.

2 The Benchmark

Sparqloscope's benchmark generation proceeds in three stages. Stage 1 precomputes statistics of the RDF dataset, given only a SPARQL endpoint. Stage 2 uses these results to compute concrete values for a set of placeholders. Stage 3 generates the concrete benchmark queries from a set of templates by substituting the placeholders with their values. We explain each of these stages in detail.

2.1 Precomputing statistics for the RDF dataset

To obtain meaningful queries generically, we first need to analyze the given dataset. In particular, we precompute the following result sets using only a SPARQL endpoint for the dataset. The respective (carefully crafted) SPARQL queries can be found at `https://purl.org/ad-freiburg/sparqloscope`.

`predicate_sizes` For each predicate, we precompute the number of triples containing it. This can be achieved with a single query.

`predicate_sizes_numeric`, `predicate_sizes_strings`, `predicate_sizes_date` For each predicate, we precompute the number of triples in which it appears with a numeric, string, or date object, respectively. This is important for evaluating functions that only take arguments of the respective datatype.

`join_subject_sizes`, `join_object_sizes`, `join_diagonal_sizes` For each pair of different predicates, we precompute the size of the result when joining the two predicates on the subject, on the object, or diagonally (join the subject of one predicate with the object of the other). This is important for evaluating group graph patterns where the results have a certain size.

`multiplicity_object`, `multiplicity_subject` For each predicate, we precompute the average multiplicity of an object or subject, respectively. This is important for evaluating `GROUP BY` queries with different group sizes, which can make a big difference for query performance and choosing the best algorithm.

In our evaluation, we use QLever for this precomputation because it can compute these queries efficiently (and at all) also when the data is large; see Section 3. Still, this precomputation takes considerable time on large datasets (for example, 10 hours on Wikidata Truthy). We therefore implemented a caching mechanism: When first computed, the results of complex queries are saved to disk. In subsequent queries, these results can be accessed via a `SERVICE` request, served by Sparqloscope while it is running. Thus all computation can be done with SPARQL queries, without having to resort to a special syntax or mechanism.

2.2 Computing values for the placeholders

Using the statistics computed in Stage 1, Sparqloscope can now compute concrete values for the various placeholders used in its query templates. In the following, we explain this for a subset of these placeholders. The full list can be inspected at `https://purl.org/ad-freiburg/sparqloscope`.

`pred_join_1`, `pred_join_2` Compute two distinct predicates that have a non-empty join result when being joined on the subject. Compute multiple instantiations for these placeholders with different size characteristics, e.g., one large and one small predicate, or two large predicates with a small result.

`pred_join_multi_1`, `pred_join_multi_2` Compute two distinct predicates that have a non-empty join result when being joined on the subject *and* the object (`?s <p1> ?o. ?s <p2> ?o`). Again, compute multiple instantiations with different size characteristics.

`pred_star_1`, `pred_star_2`, `pred_star_3` Compute three large predicates that have a large result when joined on the subject. Note that here and in the following, by *large result* we mean the largest join result that is not more than three times larger than the sum of the input sizes. This is important to avoid queries that compute (almost) Cartesian products with enormous results, which no engine can compute in a reasonable amount of time.

`pred_chain_1`, `pred_chain_2`, `pred_chain_3` Compute three large predicates that have a large result when joined as a chain, as in `?x <p1>/<p2>/<p3> ?y`. For the placeholders `predicate_star_k` and `predicate_chain_k`, we first obtain candidates from `join_subject_sizes` and `join_diagonal_sizes` (which contain the join sizes for pairs) and then compute the size of the three-way join for these candidates. We do not precompute the sizes of all possible three-way star and chain joins because that is infeasible on large knowledge graphs.

`pred_large` The predicate with the most triples in the knowledge graph.

`pred_large_numeric`, `pred_large_string`, `pred_large_date` The largest predicate where the all the objects are numeric, strings, or dates, respectively.

`pred_transitive` Compute a predicate that is highly transitive, meaning that `?s <p>+ ?o` is much larger than the number of triples in the predicate without the transitive `+` operator. This is computed by finding candidate predicates that have a non-empty diagonal self-join (`?s <p>/<p> ?o`) and by then evaluating the size of the transitive hull of these candidates. We choose the predicate that maximizes the ratio between the size of the transitive hull and the size of the predicate.

2.3 Generating the queries

In a final step the placeholder values computed in Stage 2 (which depend on the given knowledge graph) are substituted into query templates (which are independent of the knowledge graph) to obtain the final benchmark queries. In the following, we describe a selection of these templates and resulting queries. The full set of query templates, as well as the concrete benchmark for a variety of knowledge graphs, can be found at `https://purl.org/ad-freiburg/sparqloscope`.

Basic graph patterns Sparqloscope contains the following templates that use the placeholders from the previous section:

```
SELECT (COUNT(*) AS ?count) {
  ?s %pred_join_1% ?o1 .
  ?s %pred_join_2% ?o2 .
}
```

```
SELECT (COUNT(*) AS ?count) {
  ?s %pred_join_multi_1% ?o .
  ?s %pred_join_multi_2% ?o .
}
```

```
SELECT (COUNT(*) AS ?count) {
  ?s %pred_star_1% ?o1 .
  ?s %pred_star_2% ?o2 .
  ?s %pred_star_3% ?o3 .
}
```

```
SELECT (COUNT(*) AS ?count) {
  ?x1 %pred_chain_1% ?x2 .
  ?x2 %pred_chain_2% ?x3 .
  ?x3 %pred_chain_3% ?x4 .
}
```

These templates cover the typical join patterns, in particular simple joins between two triples on one and two columns, star joins, and chain joins. Like most query templates, they use the COUNT(*) aggregate which forces the computation of the full joins to determine the result size, but doesn't materialize the full (possibly large) result. This is an example of how we isolate the evaluation of a particular feature, in this case the join of two or more graph patterns.

Optional graph patterns The templates are similar to the ones for basic graph patterns (BGPs) above, but with the last triple moved into an OPTIONAL. For the BGPs with three triples (chain and star) there are also templates with the last two triples inside a single OPTIONAL. For example, the following templates are derived from the chain BGP:

```
SELECT (COUNT(*) AS ?count) {
  ?x1 %pred_chain_1% ?x2 .
  ?x2 %pred_chain_2% ?x3 .
  OPTIONAL {
    ?x3 %pred_chain_3% ?x4 .
  }
}
```

```
SELECT (COUNT(*) AS ?count) {
  ?x1 %pred_chain_1% ?x2 .
  OPTIONAL {
    ?x2 %pred_chain_2% ?x3 .
    ?x3 %pred_chain_3% ?x4 .
  }
}
```

These templates cover the following interesting cases:

1. Different size characteristics of the left and right side of an optional join.
2. The left/right argument of OPTIONAL being the result of a join on the same variable as the optional join (using the star join placeholders).
3. The left/right argument of OPTIONAL being the result of a join on a different variable than the optional join (using the chain join placeholders).

Minus and Exists The templates are the same as for OPTIONAL but with OPTIONAL replaced by MINUS or FILTER EXISTS respectively.

Group By The templates in this group [sic] all have the following form:

```
SELECT ?x (COUNT(*) AS ?count) {
  %triples%
} GROUP BY ?x
ORDER BY DESC(?count)
LIMIT 10
```

The feature we want to isolate with these queries is the actual grouping. That is why we choose the COUNT(*) aggregate, which is cheap to compute. The performance of the harder to compute aggregates is evaluated separately, see the next paragraph. The purpose of the ORDER BY and LIMIT is to force the computation of all intermediate results without having a large final result. The placeholder triples is substituted with one of the following:

1. A single triple `?s %p1% ?x`.
2. Two triples `?s %p1% ?x. ?s2 %p2% ?x` that are joined on the grouped variable.
3. Two triples `?s %p1% ?o. ?s %p2% ?x` that are joined on a variable different from the grouped variable.

Each of these three substitutions results in two queries: one for which the values for the placeholders p1 and p2 are chosen such that average group size is large, and one where they are chosen such that the average group size is small.

Aggregate functions The templates have the following form:

```
SELECT (%agg%(?x) AS ?agg) {
 ?s %pred% ?x
}
```

Here, agg is replaced by one of the aggregate functions defined by SPARQL (COUNT, MIN, MAX, AVG, SAMPLE, GROUP_CONCAT). The predicate pred is chosen such that ?x is bound to valid inputs for the aggregates. For example pred is pred_large_numeric from Section 2.2 when agg is AVG. These queries isolate the performance of computing the aggregate functions, as the grouping is trivial, because there is a single implicit group.

Property Paths The templates have the following form:

```
SELECT (COUNT(*) AS ?count) {
 %transitive_triples%
}
```

Here transitive_triples is one of the following, using the pred_transitive placeholder from Section 2.2:

1. ?s %pred_transitive%+ ?o (full transitive closure).
2. ?x %p_join% ?y . ?y %pred_transitive%+ ?z (join between simple triple and transitive closure) for predicates p_join with different size characteristics, obtained from the join_diagonally_sizes statistics from Section 2.1.
3. %constant_iri% %pred_transitive%+ ?o, where constant_iri is computed such that the result size is maximized.

Builtin functions The templates in this group have the following form:

```
SELECT (%agg%(?result) AS ?agg) {
  ?s %pred% ?o.
  BIND (%func% AS ?result)
}
```

The three placeholders are instantiated as follows:

1. func is a function expression like STRBEFORE(?o, "a").
2. pred is a predicate such that ?o binds to suitable inputs for func. For example, if func is a function that accepts strings, then pred is the placeholder pred_large_strings from Section 2.2.
3. agg is an aggregate function such that the query result a single number. This forces the evaluation of func for each input. We use aggregate functions that return numeric values, in order to minimize the overhead of the aggregation (e.g., AVG(STRLEN(?result)) for functions that return strings).

Here is a complete example template for the STRBEFORE function:

```
SELECT (AVG(STRLEN(?result)) AS ?agg) {
  ?s %pred_strings% ?o.
  BIND (STRBEFORE(?o, "a") AS ?result)
}
```

For the commonly used `REGEX` function, Sparqloscope contains multiple such templates for different regular expressions (regexes), namely

1. A regex that only denotes an exact substring match (`"comp"`).
2. A regex that denotes a prefix match (`"^comp"`)
3. A complex regex that includes special characters (`"[Mm]illenn?ium"`).

Filter We use the following template:

```
SELECT (COUNT(*) AS ?count) {
  ?s %pred_largest% ?o.
  FILTER (?s=?o)
}
```

We use the function `?s=?o` because it is cheap to evaluate (the purpose of this query is to measure filtering performance, not expression evaluation). It typically filters out most rows. We also use the analogous template with `FILTER(?s!=?o)` to evaluate a filter that preserves most of the rows. Additionally, we use the following `FILTER` template:

```
SELECT (COUNT(*) AS ?count) {
  ?s %pred_numeric% ?o.
  FILTER (?o > %percentile%)
}
```

where `percentile` is substituted for the median, 70 percentile and 95 percentile of the objects of `pred_large_numeric` (these values are precomputed in stage 2). Such queries can be implemented efficiently using binary search and we want Sparqloscope to detect if an engine implements this optimization.

Union We include templates like the following:

```
SELECT (COUNT(*) AS ?count) {
  ?s %pred_union_small% ?o1.
  { ?s %pred_union_large_1% ?o2 }
  UNION
  { ?s %pred_union_large_2% ?o3 }
}
```

The predicates are chosen such that the pairwise joins between the small predicate and one of the large predicates have a small result. Such queries are faster if the engine "pulls" the join with `pred_union_small` into the `UNION`.

Statistic queries We also include queries that compute statistics over the complete dataset, e.g., `SELECT (COUNT(DISTINCT ?s) AS ?c) {?s ?p ?o}`, which computes the number of distinct subjects in the dataset. An engine can optimize such queries by leveraging precomputed metadata for its index data structures or even by precomputing the result. We include these queries because they are frequently used in practice, and to detect such optimizations.

Result serialization We use the following template for different values of the placeholder num_rows, to evaluate how fast an engine can materialize results:

```
SELECT * {
  ?s %pred_largest% ?o.
} LIMIT %num_rows%
```

3 Evaluation

We use Sparqloscope to generate benchmarks for two widely used datasets, and we evaluate six engines (five of them widely used, one relatively new) on these benchmarks. The full materials needed to reproduce our evaluation can be found at `https://purl.org/ad-freiburg/sparqloscope`. All evaluations were run on a machine running Ubuntu 24.04 LTS, equipped with an AMD Ryzen 9 9950X CPU (16 cores, 32 threads, 5.8 GHz), 190 GiB of DDR5 memory and four 8 TB NVMe disks in a RAID 0 configuration.

Datasets and settings (regarding memory and timeout)

The *DBLP* dataset contains manually curated bibliographic information on publications in computer science [1]. We use the dataset from 17.03.2025, which has 502,364,008 triples. For this dataset, we give each engine 32 GiB of RAM and set the query timeout to 180 s (3 minutes).

The *Wikidata Truthy* dataset is a subset of Wikidata, a collaboratively edited knowledge graph hosted by the Wikimedia foundation. The so-called "truthy" subset consists of a selection of statements marked as most pertinent for each property. We use the dataset from 22.04.2025, which has 7,941,292,526 triples. For this dataset, we give each engine 64 GiB of RAM and set the query timeout to 300 s (5 minutes).

SPARQL engines

We evaluate the following six SPARQL engines: QLever [5] (developed by the University of Freiburg and QLeverize AG), Virtuoso [11] (developed by OpenLink Software), MillenniumDB [25] (developed by the Chilean Millennium Institute for Foundational Research on Data), GraphDB [15] (developed by Ontotext), Blazegraph [24] (originally developed by Systap LLC, abandoned since 2018), and Jena [8] (a project of the Apache Software Foundation since 2012).

QLever, MillennniumDB, Blazegraph, and Jena are free and open-source software. Virtuoso has a commercial closed-source version and a free open-source version, and we use the latter. GraphDB is closed-source with a free edition, which we use. QLever and MillenniumDB are written in C++. Virtuoso is written in C. GraphDB, Blazegraph, and Jena are written in Java.

Our evaluation uses QLever at commit bb1bb54, Virtuoso at version 7.2.15, MillenniumDB at commit ecbf6dd, GraphDB at version 11.0.0, Blazegraph at version 2.1.6 RC and Apache Jena at version 5.5.0.

Sparqloscope results for DBLP (502,364,008 triples)

Queries (30 of 105 shown)	**QLV**	**VTS**	**MDB**	**GDB**	**BLZ**	**JNA**
BGP (2 triples): large result	0.01	0.48	4.75	20.52	85.55	44.88
BGP (2 triples): small result	0.01	0.35	0.01	1.14	4.57	7.62
BGP (3 triples): star	1.13	1.34	12.68	24.58	81.23	63.16
BGP (3 triples): chain	0.50	1.65	14.22	81.52	96.65	×
Optional: large predicates	0.25	0.54	7.33	32.02	54.22	85.30
Optional: 3 triple star	0.91	0.56	12.24	36.29	129.98	163.22
Optional: 3 triple chain	3.46	2.14	35.57	67.77	56.69	×
Minus: large predicates	0.24	0.47	5.17	5.68	36.42	102.37
Minus: 3 triple star	0.88	0.65	11.20	19.81	149.45	×
Minus: 3 triple chain	3.34	1.51	13.94	32.83	164.29	×
Exists: large predicates	0.50	0.48	5.30	43.57	47.88	38.16
Exists: 3 triple star	1.23	0.58	11.49	59.07	141.76	139.12
Exists: 3 triple chain	4.57	1.51	14.68	56.37	57.36	75.73
Union: constrained by small join	0.05	0.42	1.13	6.17	9.37	22.59
Group by: few groups	0.01	0.23	1.81	13.94	0.06	×
Group by: many groups	0.82	29.34	8.73	19.05	96.66	139.58
Group by: numeric min	0.05	0.10	0.12	1.41	3.87	1.66
Count distinct: low multiplicity	5.07	19.10	24.34	11.58	23.35	85.14
Transitive path: plus	0.07	×	0.10	0.32	11.89	0.65
Transitive path: join and plus	0.04	×	0.10	0.24	3.26	0.57
Regex: contains	6.16	2.20	1.60	10.97	18.67	34.85
Regex: prefix	0.01	1.56	10.89	10.74	16.70	33.04
String: strbefore function	5.03	1.59	6.56	12.66	5.54	34.18
String: strstarts function	3.81	1.52	1.45	11.91	7.60	33.52
Result export: small	0.05	0.01	0.00	0.01	0.01	0.02
Result export: large	0.93	0.06	0.26	0.82	0.47	6.03
Numeric: round function	0.09	0.08	0.14	2.43	3.68	1.66
Numeric: filter $\geq$ median	0.03	0.03	0.15	1.41	3.78	19.49
Filter: English literals	0.01	7.79	0.38	5.91	45.03	14.24
Date: year function	0.09	0.46	0.10	3.43	14.19	4.11
Aggregate metrics	**QLV**	**VTS**	**MDB**	**GDB**	**BLZ**	**JNA**
percentage failed	0.0%	4.9%	0.0%	0.0%	14.7%	18.6%
geometric mean (penalty 2)	0.18	0.53	1.25	5.91	13.58	15.29
geometric mean (penalty 10)	0.18	0.57	1.25	5.91	17.20	20.64
median	0.19	0.55	2.54	11.52	17.14	33.18

Table 1. Evaluation results of six engines (QLV: QLever, VTS: Virtuoso, MDB: MillenniumDB, GDB: GraphDB, BLZ: Blazegraph, JNA: Apache Jena) on the DBLP benchmark created using Sparqloscope. All query times are given in seconds. A red cross × signifies that the query timed out or failed; these queries are considered with twice and ten times the timeout for the computation of the geometric mean. The best result for a query is highlighted in blue. The upper part of the table shows only a selection of the full benchmark (which contains 105 queries), with a focus on the queries discussed in Section 3.1. The aggregate metrics below are for the full benchmark.

Sparqloscope results for Wikidata Truthy (7,941,292,526 triples)

Queries (30 of 105 shown)	**QLV**	**VTS**	**MDB**	**GDB**	**BLZ**	**JNA**
BGP (2 triples): large result	6.15	15.66	105.81	×	×	×
BGP (2 triples): small result	0.35	1.47	4.80	24.40	170.88	×
BGP (3 triples): star	10.50	10.46	206.79	×	×	×
BGP (3 triples): chain	6.95	36.17	287.28	×	×	×
Optional: large predicates	14.68	×	×	×	×	×
Optional: 3 triple star	25.45	37.82	148.65	×	×	×
Optional: 3 triple chain	148.14	214.60	×	×	×	×
Minus: large predicates	15.27	165.91	×	×	×	×
Minus: 3 triple star	25.18	11.65	95.79	×	×	×
Minus: 3 triple chain	141.06	176.87	224.45	×	×	×
Exists: large predicates	33.38	123.49	×	×	×	×
Exists: 3 triple star	36.99	12.09	93.15	×	×	×
Exists: 3 triple chain	164.97	176.78	226.25	×	×	×
Union: constrained by small join	1.76	4.08	19.03	108.04	223.79	×
Group by: few groups	0.01	0.05	0.03	0.34	0.09	×
Group by: many groups	24.13	×	×	×	×	×
Group by: numeric min	0.65	2.73	1.81	149.42	80.48	×
Count distinct: low multiplicity	189.08	×	×	×	×	×
Transitive path: plus	1.88	×	6.75	20.58	279.72	×
Transitive path: join and plus	0.05	×	0.09	0.19	1.75	×
Regex: contains	189.74	83.48	×	×	×	×
Regex: prefix	0.02	85.16	×	×	×	×
String: strbefore function	150.03	95.48	×	×	×	×
String: strstarts function	117.91	79.75	293.74	×	×	×
Result export: small	0.02	0.02	0.01	0.03	0.18	×
Result export: large	1.20	10.91	1.12	6.25	24.48	×
Numeric: round function	1.36	3.15	2.09	173.60	117.49	×
Numeric: filter $\geq$ median	0.19	1.84	2.06	×	87.57	×
Filter: English literals	0.01	×	×	×	×	×
Date: year function	1.24	3.64	2.16	×	111.34	×

Aggregate metrics	**QLV**	**VTS**	**MDB**	**GDB**	**BLZ**	**JNA**
percentage failed	2.0%	14.7%	30.4%	61.8%	58.8%	100.0%
geometric mean (penalty 2)	2.43	11.36	22.78	111.20	123.21	600.00
geometric mean (penalty 10)	2.51	14.40	37.15	300.49	317.54	3000.00
median	2.41	15.84	69.97	×	×	×

Table 2. Evaluation results of six engines (QLV: QLever, VTS: Virtuoso, MDB: MillenniumDB, GDB: GraphDB, BLZ: Blazegraph, JNA: Apache Jena) on the Wikidata Truthy benchmark created using Sparqloscope. All query times are given in seconds. A red cross × signifies that the query timed out or failed; these queries are considered with twice and ten times the timeout for the computation of the geometric mean. The best result for a query is highlighted in blue. The upper part of the table shows only a selection of the full benchmark (which contains 105 queries), with a focus on the queries discussed in Section 3.1. The aggregate metrics below are for the full benchmark.

3.1 Results

Tables 1 and 2 provide an excerpt of our evaluation for the named engines and datasets, showing results for 30 out of 105 queries each and some aggregate measures. The full results (and more benchmarks and evaluations) can be found at `https://purl.org/ad-freiburg/sparqloscope-evaluation`.

It is important to note that this evaluation is meant as a showcase for our benchmark, and *not* as a complete performance evaluation. The point is to demonstrate how Sparqloscope can reveal specific strengths and weaknesses of an engine. In some cases, we know the reasons for a particular slow or fast query time and provide an explanation (often for QLever, as it is our own engine). In other cases, this needs further investigation. In any case, we learn something about the engines that was not obvious from the start. Most of the explanations in the following pertain to the results for Wikidata Truthy, which is larger and therefore more challenging. Once more, note how the small number of queries (105 for the whole benchmark) allows the investigation and discussion of individual queries instead of having to interpret opaque aggregate measures.

Basic Graph Patterns QLever is fastest for basic graph patterns. It is much faster than Virtuoso, which in turn is much faster than MillenniumDB. The three Java engines are yet much slower and all time out for the larger Wikidata Truthy. These queries show how much optimization potential there is even for this basic (and fundamental) SPARQL operation. QLever performs so well because it heavily optimizes read time (by storing its disk-based index data in large compressed blocks) and write time (in this case by not producing any result bindings when only a count is needed; without this optimization QLever would be on par with Virtuoso), and by a very carefully engineered join algorithm.

Optional, Minus, and Exists For these queries, the trend is similar as for the basic graph patterns, but the difference between QLever and Virtuoso is less pronounced. For the 3-triple star, Virtuoso is faster on both datasets. The reason is that `OPTIONAL`, `MINUS`, and `EXISTS` are more complex operations with less predictable access patters, which QLever has not fully mastered yet, whereas the maturity of Virtuoso's underlying relational database shines through. Again, the three Java engines are much slower and all fail on Wikidata Truthy.

Group By QLever is the only engine that can compute all `GROUP BY` queries without encountering a timeout. In particular, it is the only engine that can handle `GROUP BY` with many different groups on Wikidata Truthy. This is a good example of Sparqloscope's ability to pinpoint individual strengths and weaknesses of an engine (apparently, none of the other engines has considered this relevant use case). QLever is also fastest in aggregating results (except for `GROUP_CONCAT` of strings, see the full results at the link above).

Property Paths QLever is fastest for all queries involving transitive closure (via the `+` operator). MillenniumDB is on par when the subject is fixed. Virtuoso fails for all these queries on Wikidata Truthy; apparently computing the transitive closure does not map well to Virtuoso's underlying relational database. All engines fail for the query with the largest result (not shown in the table, see the full results at the link above). This shows the significantly higher complexity of computing the transitive closure relative to other SPARQL operations.

String functions For string functions (like `STRAFTER` or `STRBEGINS`), Virtuoso is consistently the fastest engine. QLever is next, with roughly the same time for each function. This indicates that the bottleneck is not the function evaluation, but the way that the strings are stored and loaded. QLever stores its strings individually compressed, whereas Virtuoso's underlying relational database employs small-string optimizations, which are still missing in QLever. The three Java engines are much slower and all time out on Wikidata Truthy.

Numeric functions For numeric functions, QLever is consistently much faster than the other engines. All engines are at least one order of magnitude faster when evaluating numeric functions than when evaluating string functions. This indicates that they have a special handling for numeric literals, which apparently is best implemented for QLever. For all engines, the implementation of integer literals is vulnerable to overflows. This problem is worst for MillenniumDB, where almost all numeric queries on Wikidata Truthy give wrong results.

Range-like filters QLever implements filters via binary search whenever possible. This includes numeric relational filters like `?x > 3`, but also expressions like `REGEX(?x, "^auto")`, which are equivalent to prefix matching. For the respective queries, QLever beats the other engines by a large margin, which indicates that only QLever implements this (practically very relevant) optimization. The performance difference is particularly pronounced for prefix regexes, where the other engines have to load all involved strings and evaluate the regex.

Other filters QLever is faster than Virtuoso when the input is large and most results are filtered out, whereas Virtuoso is faster when only few results are filtered out (this is not shown in Tables 1 and 2, see the full evaluation at the link above). This is another good example of Sparqloscope's pinpointing property.

Export QLever and MillenniumDB export results at a similar speed, with a slight edge for MillenniumDB, probably due to QLever storing its IRIs and literals individually compressed (to save storage). Virtuoso is much slower for large results and silently truncates its output to at most 1 million rows (one of Virtuoso's many idiosyncrasies). Somewhat surprisingly, GraphDB is on par with Virtuoso.

Loading Wikidata Truthy QLever, Virtuoso, and MillenniumDB each load the Wikidata Truthy data in around 3 hours, but QLever requires significantly less RAM (20 GB vs. 64 GB). GraphDB required 70 GB of heap space and had to be resumed twice from a checkpoint after out-of-memory crashes; the whole process took over 24 hours. For Blazegraph, the data was loaded in chunks of one million triples and took over two days (this could have been accelerated somewhat using bulk loading). For Apache Jena, loading did not finish within one week. These experiences are the reason why we did not evaluate the engines on even larger datasets. QLever can handle datasets with hundreds of billions of triples in a resourceful manner. Virtuoso can also handle datasets of that size, but requires much more RAM and at least twice the disk space. For MillenniumDB, RAM consumption during loading becomes prohibitively large. For the Java engines, much larger datasets are out of reach.

Impact of dataset size Most of the discussion above relates to the benchmark on the larger Wikidata Truthy dataset (ca. 8 B triples). On the smaller DBLP dataset (ca. 500 M triples), the trends are similar but the difference between the engines are less pronounced. In particular, the weaker engines profit more when the data fits completely into RAM, whereas QLever is designed to not rely on this. Also, the current version of QLever incurs a fixed overhead for each query (mostly due to query planning), which is negligible for large datasets but shows for smaller datasets. This shortcoming will soon be addressed by QLever's developers. Virtuoso fails computing the transitive closure even on the smaller DBLP data, which points to a systematic problem.

4 Conclusion

We have presented the Sparqloscope benchmark generator, which automatically generates benchmark queries for any given RDF dataset, comprehensively covers a large set of SPARQL 1.1 features, evaluates the individual features in isolation, and includes no more queries than necessary. The result is a benchmark of moderate size (105 queries), so that each result can (and should) be inspected individually, instead of having to interpret opaque aggregate metrics.

We have used Sparqloscope to evaluate and compare the performance of six SPARQL engines (QLever, Virtuoso, MillenniumDB, GraphDB, Blazegraph and Jena) on two widely used datasets (DBLP and Wikidata). The benchmark revealed many interesting strengths and weaknesses of the engines. This is useful both for their users (to help them understand the performance they experience), as well as for the developers (to help them understand and improve on the revealed weaknesses). All benchmarks and results are publicly available on `https://purl.org/ad-freiburg/sparqloscope`. We will add more benchmarks, engines, and evaluations over time and upon request.

Sparqloscope is easy to extend and will be extended in the future. In particular, we plan to add queries for named graphs and some of the SPARQL functions that are not yet covered. We have already used and will continue to use insights from Sparqloscope to improve our own engine, QLever.

Resource Availability Statement: Source code and documentation are available on GitHub via `https://purl.org/ad-freiburg/sparqloscope` and are released publicly under the Apache 2.0 License. An archived copy of the exact version used for evaluation and the full results are available on Zenodo via `https://doi.org/10.5281/zenodo.15387491`.

Acknowledgments. We are grateful to Tanmay Garg who implemented a script to run a given benchmark against an arbitrary SPARQL endpoint, and a convenient web app for the visualization of the results.

References

1. Ackermann, M.R., Bast, H., Beckermann, B.M., Kalmbach, J., Neises, P., Ollinger, S.: The dblp Knowledge Graph and SPARQL Endpoint. TGDK **2**(2), 3:1–3:23 (2024)
2. Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.N.: A survey of RDF stores & SPARQL engines for querying knowledge graphs. VLDB J. **31**(3), 1–26 (2022)
3. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: ISWC (1). Lecture Notes in Computer Science, vol. 8796, pp. 197–212. Springer (2014), `https://link.springer.com/chapter/10.1007/978-3-319-11964-9_13`
4. Angles, R., Buil-Aranda, C., Hogan, A., Rojas, C., Vrgoc, D.: WDBench: A Wikidata Graph Query Benchmark. In: ISWC. Lecture Notes in Computer Science, vol. 13489, pp. 714–731. Springer (2022), `https://iswc2022.semanticweb.org/wp-content/uploads/2022/11/978-3-031-19433-7_41.pdf`
5. Bast, H., Buchhold, B.: QLever: A Query Engine for Efficient SPARQL+Text Search. In: CIKM. pp. 647–656. ACM (2017)
6. Bast, H., Kalmbach, J., Klumpp, T., Kramer, F., Schnelle, N.: Efficient and Effective SPARQL Autocompletion on Very Large Knowledge Graphs. In: CIKM. pp. 2893–2902. ACM (2022)
7. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. Int. J. Semantic Web Inf. Syst. **5**(2), 1–24 (2009), `http://wbsg.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark`
8. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW (Alternate Track Papers & Posters). pp. 74–83 (2004)
9. Chen, Y., Kokar, M.M., Moskal, J.J.: SPARQL Query Generator (SQG). J. Data Semant. **10**(3-4), 291–307 (2021)
10. Dang, M.H., Aimonier-Davat, J., Molli, P., Hartig, O., Skaf-Molli, H., Crom, Y.L.: FedShop: A Benchmark for Testing the Scalability of SPARQL Federation Engines. In: ISWC. Lecture Notes in Computer Science, vol. 14266, pp. 285–301. Springer (2023)
11. Erling, O.: Virtuoso, a Hybrid RDBMS/Graph Column Store. IEEE Data Eng. Bull. **35**(1), 3–8 (2012)
12. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Semant. **3**(2-3), 158–182 (2005), `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=aeedd9f5c0fe4ca3e850cfbf7425e3f312e30cf7`
13. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A Worst-Case Optimal Join Algorithm for SPARQL. In: ISWC (1). Lecture Notes in Computer Science, vol. 11778, pp. 258–275. Springer (2019), `https://aidanhogan.com/docs/SPARQL_worst_case_optimal.pdf`
14. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.N.: DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data. In: ISWC (1). Lecture Notes in Computer Science, vol. 7031, pp. 454–469. Springer (2011)
15. Ontotext: GraphDB, `https://graphdb.ontotext.com/`
16. Raasveldt, M., Holanda, P., Gubner, T., Mühleisen, H.: Fair benchmarking considered difficult: Common pitfalls in database performance testing. In: DBTest@SIGMOD. pp. 2:1–2:6. ACM (2018)

17. Saleem, M., Hasnain, A., Ngomo, A.N.: LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation. J. Web Semant. **48**, 85–125 (2018)
18. Saleem, M., Mehmood, Q., Ngomo, A.N.: FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In: ISWC (1). Lecture Notes in Computer Science, vol. 9366, pp. 52–69. Springer (2015)
19. Saleem, M., Stadler, C., Mehmood, Q., Lehmann, J., Ngomo, A.N.: SQCFramework: SPARQL Query Containment Benchmark Generation Framework. In: K-CAP. pp. 28:1–28:8. ACM (2017)
20. Saleem, M., Szárnyas, G., Conrads, F., Bukhari, S.A.C., Mehmood, Q., Ngomo, A.N.: How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks. In: WWW. pp. 1623–1633. ACM (2019)
21. Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In: ISWC (1). Lecture Notes in Computer Science, vol. 7031, pp. 585–600. Springer (2011)
22. Schmidt, M., Hornung, T., Meier, M., Pinkel, C., Lausen, G.: SP^{2}Bench: A SPARQL Performance Benchmark. In: Semantic Web Information Management, pp. 371–393. Springer (2009), `https://arxiv.org/abs/0806.4627`
23. Tarasova, T., Marx, M.: ParlBench: A SPARQL Benchmark for Electronic Publishing Applications. In: ESWC (Satellite Events). Lecture Notes in Computer Science, vol. 7955, pp. 5–21. Springer (2013)
24. Thompson, B.B., Personick, M., Cutcher, M.: The bigdata® RDF graph database. In: Linked Data Management, pp. 193–237. Chapman and Hall/CRC (2014)
25. Vrgoc, D., Rojas, C., Angles, R., Arenas, M., Arroyuelo, D., Buil-Aranda, C., Hogan, A., Navarro, G., Riveros, C., Romero, J.: MillenniumDB: A Persistent, Open-Source, Graph Database. CoRR **abs/2111.01540** (2021)
26. Zhang, Y., Pham, M., Corcho, Ó., Calbimonte, J.: SRBench: A Streaming RDF/SPARQL Benchmark. In: ISWC (1). Lecture Notes in Computer Science, vol. 7649, pp. 641–657. Springer (2012)