WDBench: A Wikidata Graph Query Benchmark

Renzo Angles[1,2], Carlos Buil Aranda[1,3], Aidan Hogan[1,4], Carlos Rojas[1], and Domagoj Vrgoč[1,5(✉)]

[1] IMFD Chile, Santiago, Chile
[2] DCC, Universidad de Talca, Talca, Chile
[3] Universidad Técnica Federico Santa María, Valparaíso, Chile
[4] DCC, Universidad de Chile, Santiago, Chile
[5] PUC Chile, Santiago, Chile
dvrgoc@ing.puc.cl

Abstract. We propose WDBench: a query benchmark for knowledge graphs based on Wikidata, featuring real-world queries extracted from the public query logs of the Wikidata SPARQL endpoint. While a number of benchmarks for graph databases (including SPARQL engines) have been proposed in recent years, few are based on real-world data, even fewer use real-world queries, and fewer still allow for comparing SPARQL engines with (non-SPARQL) graph databases. The raw Wikidata query log contains millions of diverse queries, where it would be prohibitively costly to run all such queries, and difficult to draw conclusions given the mix of features that these queries use. WDBench thus focuses on three main query features that are common to SPARQL and graph databases: (i) basic graph patterns, (ii) optional graph patterns, (iii) path patterns, and (iv) navigational graph patterns. We extract queries from the Wikidata logs specifically to test these patterns, clean them of non-standard features, remove duplicates, classify them into different structural subsets, and present them in two different syntaxes. Using this benchmark, we present and compare performance results for evaluating queries using Blazegraph, Jena/Fuseki, Virtuoso and Neo4j.

1 Introduction

Recent years have seen renewed interest in querying graphs, driven in particular by the growing popularity of knowledge graphs [25]. There are two related options for querying knowledge graphs. On the one hand, SPARQL [22] is the standard query language for RDF graphs/datasets [16], and has enjoyed significant developments down through the years, including the publication of hundreds of public query services [43], the development of hundreds of SPARQL query engines and prototypes [1], the release of dozens of benchmarks [1], an extended version of the original standard [22], etc. SPARQL is the query language of choice for prominent open knowledge graphs – such as DBpedia [29], Wikidata [44], etc. – which provide public query services that can receive in the order of hundreds of thousand

U. Sattler et al. (Eds.): ISWC 2022, LNCS 13489, pp. 714–731, 2022.
https://doi.org/10.1007/978-3-031-19433-7_41

or even millions of queries per day [30,35]. On the other hand, a variety of graph query languages, databases, etc., have been proposed and developed within the NoSQL/Database community [13], and have become widely used, particularly for enterprise knowledge graphs, with Neo4j [46] and its query language Cypher [20] leading the way in terms of popularity.[1]

With may options available, it can be difficult to choose a suitable engine to support queries over a given knowledge graph, which calls for graph query benchmarks that reflect real-world workloads. For example, the Wikidata community is currently seeking an alternative to replace Blazegraph [42], whose development team has moved on to work on other projects.[2]

While dozens of query benchmarks have been proposed down through the years for RDF/SPARQL [2,9,12,17,19,21,26,31,36,39,40,48] and graph databases [7,19], most rely on synthetic data generated according to a fixed schema [2,7,12,17,19,21,39,40]. While benchmarks based on synthetic data are useful for scalability testing, since most allow for generating graphs of arbitrary size, the schemas used for such benchmarks are hand-crafted and thus often much simpler than the organic, collaboratively-generated schemas that emerge within knowledge graphs such as DBpedia [29] and Wikidata [44].

A smaller number of benchmarks have been proposed based on real-world knowledge graphs [9,26,31,36,48], but either rely on synthetic queries [26], a small number of hand-selected queries [9,48], or instances of a small number of templates induced through log analysis [31,36]. One of the challenges of using query logs [30,35] for benchmarks is the sheer number and diversity of queries available, with, for example, millions of queries available in the Wikidata query logs [30]. Running all such queries over multiple engines on a large knowledge graph would not only be prohibitively costly, but would also generate results that are difficult to interpret, given that real-world queries will often mix features. Approaches to deal with this have focused on generating templates [31,36].

In this paper, we rather follow a *feature-based approach*: we generate a real-world benchmark by extracting a large and diverse set of queries from the query log of an open knowledge graph, but only for selected core features that are common to both SPARQL engines and graph databases [5]. Within these features, we define high-level subclasses in order to gain more detailed insights into the performance of different engines. The specific benchmark we propose here, which we call *WDBench*, is based on the Wikidata knowledge graph [44] and query logs [30]. The features we currently focus on are basic graph patterns, optional graph patterns, path patterns, and navigational graph patterns, which can be translated to SPARQL and Cypher. We use WDBench to compare the query performance of Blazegraph [42], Jena TDB [27], Virtuoso [18] and Neo4j [46].

Paper Structure. Section 2 discusses related work, Sect. 3 describes the design of WDBench, Sect. 4 describes the experimental design, Sect. 5 describes the results of these experiments, while Sect. 6 concludes.

[1] See https://db-engines.com/en/ranking/graph+dbms; retr. 2022-05-06.
[2] See https://phabricator.wikimedia.org/T206560; retr. 2022-05-06.

2 Related Work

As highlighted previously, dozens of benchmarks have been proposed for RDF and other graph databases over the years. They can be classified in two general classes: benchmarks based on synthetic and real-world graphs. Some benchmarks target RDF/SPARQL engines, while others target other graph databases; to the best of our knowledge, the latter exclusively use synthetic datasets.

Synthetic SPARQL-Oriented Benchmarks. The Lehigh University Benchmark (LUBM) [21] was one of the first benchmarks proposed for RDF/SPARQL, generating synthetic data about universities. Berlin [12] generates data following an e-commerce use-case, with comparable SPARQL and SQL queries provided. SP^2Bench [39] generates an arbitrarily-large graph following the schema of DBLP database, with queries provided in a variety of shapes. BowlognaBench [17] generates synthetic RDF graphs about universities, providing queries inspired by the Bologna reform of European universities. WatDiv [2] presents an approach that focuses on generating diverse graph data and basic graph patterns in order to address the "structuredness" problem of other benchmarks [37]; queries follow star, path and snowflake query shapes. TrainBench [40] is another synthetic benchmark, this time inspired by a railway network, defining six queries encoding network validation constraints.

Synthetic Graph Database-Oriented Benchmarks. gMark [7] provides a domain- and query language-independent driver, generating query workloads for a user-defined schema. The user can define the scenario from which the data is generated (i.e. social network, biological database, etc.) and from that data the driver generates the queries and translates them to the desired engine (i.e. Neo4J, SPARQL, etc.). The Linked Data Benchmark Council's Social Network Benchmark LDBC-SNB [19] is a benchmark that provides a common synthetic dataset for two different query workloads. The dataset represents a social network and the two workloads differ in the use case they evaluate the engine for: one focuses on transactional graph processing queries that target neighbouring nodes and update operations that continuously insert new data in the graph. The second workload focuses on aggregate queries accessing large parts of the graph.

Real-World RDF-Oriented Benchmarks. DBpedia SPARQL Benchmark (DBSBM) [31] generates queries for a specific version of DBpedia based on real-world query logs. The queries in the log files are cleaned and clustered according to the SPARQL features, generating 25 query templates from the most prominent clusters with placeholder variables that can be instantiated from the data in order to generate multiple instances per template. FEASIBLE [36] builds upon this idea of generating benchmarks from query logs. The query generation takes into account several query characteristics such as number of triple patterns or number of join vertices, generates vectors that represent queries according to the features, and generates queries based on the patterns from the vectors. BioBenchmark [48] defines a benchmark over five biomedical datasets (Cell, Allie, PDBJ,

DDBJ, and UniProt), providing 48 queries extracted from real-world applications. The Wikidata Graph Pattern Benchmark [26] is based on Wikidata, but rather uses synthetic queries following structural graph pattern templates.

Comparison and Novelty. We refer to Saleem et al. [37] for a detailed comparison of the benchmarks discussed here. In terms of the novelty of WDBench, it uses real-world data and queries; to the best of our knowledge, only DBSBM [31] and FEASIBLE [36] share this characteristic. Unlike these two benchmarks, WDBench (1) is based on Wikidata rather than DBpedia; (2) uses a larger graph (1.257 billion triples/edges vs. 232 million triples/edges); (3) contains path patterns that can match arbitrary length paths, which are a key feature of graph queries; (4) is offered in both SPARQL and Cypher variants; (5) does not apply templates or clustering, but rather contains a larger and more diverse query set that includes thousands of queries. It is important to note that the goal of WDBench is to complement existing benchmarks rather than to replace them. We see WDBench as being a useful resource to test query performance for core features of graph queries over a real-world knowledge graph using realistic workloads. However, other benchmarks may have other benefits, and could be run alongside WDBench. For example, synthetic benchmarks have the benefit of being able to generate graphs of arbitrary size, where one could be run alongside WDBench in order to stress-test scalability. Other benchmarks could be used to test SPARQL-specific or relational features not included in WDBench.

3 WDBench: Graph and Queries

We now discuss the design of WDBench. We start by explaining the rationale behind the subset of Wikidata used for benchmarking, and then specify the process for selecting a representative query set out of the millions of queries available in the Wikidata public endpoint log [13,30]. We then discuss conversion of the benchmark into a property graph with Cypher queries for running Neo4j.

3.1 WDBench Graph

In order to define the graph used in WDBench, we were guided by three criteria: (i) that it is representative of a diverse, large-scale, real-world knowledge graph; (ii) that it covers a wide range of queries from the public query log of Wikidata; and (iii) that it is succinct, i.e., that it does not contain massive amounts of data irrelevant for the queries that would increase load times for different engines. To balance these criteria, we base WDBench on the Wikidata truthy dump [41] for three reasons: (1) it is more concise and thus faster to load: some engines can take over a week to load the complete version of Wikidata; (2) it is sufficient to address the majority of queries in the log chosen: 86.8% of the queries in this log use only truthy properties; (3) it avoids issues relating to how Wikidata-specific qualifiers should be reified in different databases: this topic diverges from our goal of a general query benchmark for knowledge graphs and

is addressed elsewhere [23,24]. To further prune the dataset, we only kept triples in which (a) the subject position is a Wikidata entity, and (b) the predicate is a truthy (direct) property. This allows us to focus on structural properties of the queries and the graph, while increasing succinctness. The particular Wikidata truthy dump we used is `20210623-truthy-BETA`, which contains 18,579,709,438 triples. After pruning based on the described criteria, the final dataset contains 1,257,169,959 triples. Many of the triples pruned are labels and descriptions in multiple languages, which we deem as inessential for testing the performance of graph pattern evaluation (rarely are joins or paths expressed via labels or descriptions). The dataset is available for download online at [3], and the scripts used to prune a truthy dump can be found online at [4].

3.2 WDBench Queries

WDBench is based on real-world queries posted by Wikidata users, as found in Wikidata's query logs [30,47]. Given that the log files contain millions of queries, where it would be prohibitively costly to run them all, and where the results would be difficult to interpret given the mix of features that they use, we reduce the queries in several phases and classify them by their features.

The first choice we made was to concentrate exclusively on queries that timed out on the Wikidata endpoint (code 500 queries in the log files [30,47]). While endpoint timeouts can be caused by many factors (including temporary server load), we wish to focus on challenging queries, where this subset of queries largely filters out the multitude of trivial queries in the log. Additionally, focusing on the code 500 queries reduces the set to 122,980 queries. If the query uses vocabulary not present in our graph, we discard it (note that queries generating empty results are kept so long as they only use relevant vocabulary terms).

The next reduction was based on the operators used by the queries. Considering that we aim to compare note only RDF/SPARQL engines, but also other graph databases, we decided to focus on four types of graph patterns at the core of popular graph query languages [5]: (i) basic graph patterns; (ii) optional graph patterns; (iii) path patterns; and (iv) navigational graph patterns (using paths). Other features – including relational-style operators such as projection, difference, selection (filter), union, aggregation, solution modifiers, etc. – could be added in future using a similar methodology; however, adding more features would complicate generating comparable queries in distinct graph query languages. We thus pruned queries that use any operator different from basic graph patterns, optionals and property paths. However, we keep queries with `SERVICE`, since this operator is used in the majority of Wikidata queries in order to specify language preferences for labels; and `DISTINCT`, `GROUP BY`, `ORDER BY`, `LIMIT`, since these solution modifiers are generally applied after processing the base query pattern. In these exceptional cases, we remove the service and solution modifier clauses and keep the resulting query. Given that the labelling service can produce new variables that can be referenced in projected results, we use `SELECT *` such that our queries are of the form `SELECT * WHERE {graph_pattern}`. From there, we profiled the following four groups of graph pattern queries.

Basic Graph Patterns. These were the queries that consisted exclusively of joins between triple patterns. In order to eliminate duplicate queries, we sort the triple patterns, and rename the variables they use, allowing us to detect the queries which differ only in variable names, or the order of triples. The result is a set of 1,335 BGP queries. We further partition BGPs into two disjoint subsets:

- Single. This set contains BGPs with a single triple pattern. While relatively simple to evaluate, these queries test the engines' data retrieval and result enumeration capabilities, which are key to evaluating any query efficiently. We ended up with 280 queries in this set.
- Multiple. These are queries consisting exclusively of BGPs, which have at least two triple patterns, and thus require a join to be performed. Again, being able to evaluate joins within basic graph patterns efficiently is crucial for query performance. This set contains a total of 681 queries.

Optional Graph Patterns. We choose optional graph patterns as a focus of WDBench since they are frequently used to query incomplete knowledge graphs [13], and they have been widely studied in the literature as a characteristic feature of graph queries that can increase the computational complexity of query evaluation [33,34]. Queries in this set include (only) basic graph patterns and one or more (potentially nested) `OPTIONAL` patterns. We further remove queries that artificially create a cross product via `OPTIONAL` whereby the right-hand side of an `OPTIONAL` contains only variables that are not mentioned elsewhere; such queries might skew the benchmark results. This Optionals set contains 498 distinct queries. We partition Optionals into two disjoint sets:

- Well-Designed (WD). An Optional query Q is *well-designed* if and only if, for every optional clause $O = \{P_1\}$ `OPTIONAL` $\{P_2\}$ it contains, each variable in P_2 either appears in P_1 or appears nowhere else in Q besides P_2 [33]. Such queries avoid leaps in complexity associated with optional graph patterns [10, 33,34]. This subset contains 390 queries.
- Not-Well-Designed (NWD). These are Optional queries that are not well-designed, and are thus associated with leaps in computational complexity for key decision problems [33]. This subset contains 108 queries.

Path Patterns. We further test the performance of executing a single property path query (excluding simple predicates). These queries test the engines' ability to detect whether there is a path connecting two nodes that conforms to a regular expression. In the research literature this class of queries is known as two-way regular path queries (2RPQs) [6], and in SPARQL standard they are called property paths [22,28].[3] Given that property paths almost exclusively form part of a larger query in our log, we extracted path patterns from queries in order to achieve a larger query set. Thus, if a query contains two property paths, this will result in two new queries being added to Paths. After eliminating duplicates Paths contains 660 queries. We partition Paths into two disjoint subsets:

[3] Property paths include negated property sets that fall outside 2RPQs [28], but these are rarely used [13], and can be partially emulated through disjunction (`|`) [28].

- Recursive (R). We call a Path query *recursive* if and only if it uses Kleene star (*) or Kleene plus (+), i.e., if and only if it can match paths of arbitrary length. There were 594 queries in this subset.
- Non-Recursive (NR). We call a Path query non-recursive if and only if it does not use Kleene star (*) nor Kleene plus (+), i.e., if and only if it can match paths of fixed length. There were 66 queries in this subset.

Navigational Graph Patterns. The final set of queries considers navigational graph patterns, which incorporate property paths [22], triple patterns, and joins; i.e., they are BGPs with property paths. To be more precise, we keep queries which use either joins, or property paths, thus having a set of queries akin to conjunctive two-way regular path queries (C2RPQs) [15]. We call this query set Navigational [5]. In order to not have an overlap with the Paths query set, all queries in C2RPQs must perform at least one join. These are more advanced queries, and SPARQL engines are known to run into issues when evaluating them [8]. The set C2RPQs contains a total of 539 queries. We further partition Navigational into two disjoint subsets:

- Recursive (R). We call a Navigational query *recursive* if and only if it contains a recursive path pattern. There are 515 such queries.
- Non-Recursive (NR). We call a Path query non-recursive if and only if it does not contain a recursive path pattern. There were 24 such queries.

3.3 Conversion to Cypher

The Wikidata dump and query logs are natively expressed as RDF/SPARQL. However, we aim for WDBench to also be usable for comparing graph databases. A complication here is that graph databases often define their own declarative query language. For now we thus focus on creating a version of the benchmark for testing with Neo4j. This requires mapping the Wikidata graph to a property graph, which is straightforwardly achieved given that we only include binary (truthy) relations: each triple is simply represented as an edge in the property graph. The more complex part involves converting the queries to Cypher [20]: Neo4j's query language. Graph patterns are expressed using a `MATCH` clause, while optional graph patterns use the `OPTIONAL MATCH` clause. Within a `MATCH` clause, Neo4j applies an edge-isomorphism semantics, while SPARQL uses a homomorphism semantics [5]; thus Cypher's query results can differ, but we found such differences to be marginal in practice. Regarding path patterns, Neo4j only supports Kleene star (i.e., zero or more, which it denotes by "*"). Where possible, we rewrite path expressions into other available Neo4j operators, with concatenations rewritten to basic graph patterns, inverses rewritten by swapping source and target nodes, etc.; however, not all property paths (2RPQs) can be supported. Neo4j allows for returning string representations of paths; to be comparable with SPARQL, we project only the endpoints of paths.

4 Running WDBench

We now turn to using WDBench in order to test the performance of four query engines. This section specifies the operational parameters for these experiments.

The Machine. All experiments were run on a single commodity server with an Intel®Xeon®Silver 4110 CPU, and 128 GB of DDR4/2666 MHz RAM, running Linux Debian 10 with the kernel version 5.10. The hard disk used to store the data was a SEAGATE ST14000NM001G with 14 TB of storage.

How we Ran the Queries. To simulate a realistic database load, we do not split queries into cold/hot run segments. Rather we run them in succession, one after another, after a cold start of each system (and after cleaning the OS cache[4]). This simulates the fact that query performance can vary significantly based on the state of the system buffer, or even on the state of the hard drive, or the state of OS's virtual memory. For each system, queries were run in the same order. We record the execution time of each individual query, which includes iterating over all results. We set a limit of 100,000 distinct results for each query, again in order to enable comparability as some engines showed instability when returning larger results (also Virtuoso is hard-limited to $2^{20} = 1,048,576$ results). We replicated this setup for each query set described above. This allows us to gauge the systems' performance on each particular type of query.

Handling Timeouts. We defined a timeout of 1 min per query for each system. This is a common limit available of SPARQL endpoints, so we replicated it in the benchmark. Apart from that, we note that most systems had to be restarted upon a timeout as they often showed instability, particularly while evaluating path queries. This was done without cleaning the OS cache in order to preserve some of the virtual memory mapping that the OS built up to that point.

Tested Engines. We use four persistent graph query engines that are popular in practice. First, we include three RDF/SPARQL engines: Jena TDB version 4.1.0 [27], Blazegraph (BlazeG for short) version 2.1.6 [42], and Virtuoso version 7.2.6 [18]. We further include a property graph engine: Neo4J community edition 4.3.5 [46]. Jena and Blazegraph were assigned 64GB of RAM, and Virtuoso was set up with 64GB or more of RAM as is recommended. Neo4J was run with default settings. The size of the WDBench dataset when loaded into each of the engines can be found in Table 1.

5 Experimental Results

In this section we present results for the tested engines on each query set specified in WDBench. We divide the discussion by the different query features described

[4] This is done by the command "`# sync; echo 3 > /proc/sys/vm/drop_caches`".

Table 1. WDBench dataset sizes when loaded into each engine.

BlazeG	Jena	Virtuoso	Neo4J
70 GB	110 GB	70 GB	112 GB

in Sect. 3.2 – namely basic graph patterns, optional graph patterns, path patterns, navigational graph patterns and their sub-variants – and discuss explanations for the behaviour we observe. All experimental results, including runtimes for individual queries on each engine tested, can be found online [4].

5.1 Basic Graph Patterns

We begin by examining the performance of each query engine for basic graph patterns (BGPs), considering both SINGLE and MULTIPLE subsets. A summary of the results can be observed in Table 2 and Fig. 1. The box plots are generated in the standard manner, showing the range between the first and the third quartile, with the midline representing the median, and the whiskers represented by thin lines. Table 2 additionally indicates how many queries are supported by the engine, the number of errors and timeouts, the average, and the median.

We can observe that as far as SINGLE is concerned, Virtuoso is the most stable engine, returning no timeouts, nor errors, closely followed by Blazegraph. In terms of performance, Blazegraph is the clear winner in the SINGLE query set, followed thereafter by Virtuoso. Both Jena and Neo4j are lagging in terms of performance, with averages 4–5 times higher than the other two engines. Neo4j's median is also above the third quartile for both Blazegraph and Virtuoso. Queries from the SINGLE set have precisely the same structure. However, depending on the exact constants they use, the results can vary from timeouts to fast runs, depending on the data distribution, number of results, etc. For this reason we believe that it is beneficial to have a large number of queries that might be structurally similar, but that access different parts of the dataset.

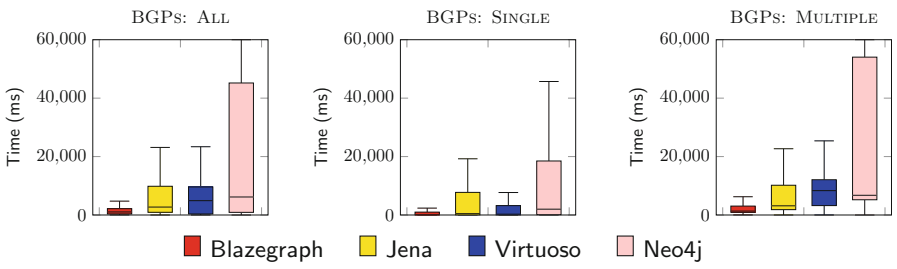

Fig. 1. Performance for all BGPs (left), SINGLE (middle), and MULTIPLE (right)

Table 2. Summary of runtimes (in seconds) for BGPs

Engine	Supported	Timeouts	Error	Average	Median
BGPs (961 queries)					
Blazegraph	961	55	0	6.51	1.05
Jena	961	79	0	10.73	2.71
Virtuoso	961	8	3	6.79	4.90
Neo4j	961	206	1	20.16	6.17
BGPs Single (280 queries)					
Blazegraph	280	3	0	1.73	0.07
Jena	280	25	0	9.92	0.46
Virtuoso	280	1	0	2.12	0.28
Neo4j	280	47	0	15.28	2.03
BGPs Multiple (681 queries)					
Blazegraph	681	52	0	8.47	1.34
Jena	681	54	0	11.06	3.16
Virtuoso	681	7	3	8.71	8.34
Neo4j	681	159	1	22.17	6.75

When considering join queries in Multiple, we observe a rather similar pattern. Virtuoso is again the most stable engine, but it falls behind Blazegraph slightly in the average case. Medians and boxplots tell another story here, showing that both Jena and Blazegraph outperform Virtuoso on the majority of the queries, where even Neo4j's median, and first to third quartiles, are lower than that of Virtuoso's. Thus it would seem that Blazegraph and Jena, in particular, can evaluate the majority of these queries faster than Virtuoso, but Virtuoso performs relatively better for higher percentiles (more costly queries).

5.2 Optional Graph Patterns

The results for Optionals is given in Table 3, and in Fig. 2. Blazegraph is the clear winner here, both in stability, with only 28 timeouts, and in speed, with its median being below the first quartile of the next best competitor, Jena. Jena also outperforms Virtuoso by a wide margin, and Neo4j trails further behind.

Considering only well-designed `OPTIONAL` patterns, the performance of Virtuoso improves drastically. Blazegraph wins in terms of runtimes, but Virtuoso surpasses other engines in stability, timing out on only 5 of 390 queries. Non well-designed optionals seems to be a major issue for Virtuoso, where it times out in

64 of 108 cases, and its performance drops significantly. Other engines actually perform significantly better on Optionals NWD. Looking a bit deeper into this performance gain, we speculate that this is mostly due to the non well-designed optionals simulating a cross-product, which generates 100,000 results, our query limit, quite fast, at least when the engine is optimised for such cases, per the results for Blazegraph and Jena (but not Virtuoso nor Neo4j).

Table 3. Summary of runtimes (in seconds) for optional graph patterns

Engine	Supported	Timeouts	Error	Average	Median
Optionals (498 queries)					
Blazegraph	498	37	0	8.55	2.16
Jena	498	59	0	13.56	4.34
Virtuoso	498	69	2	17.29	9.45
Neo4j	498	146	1	27.09	17.87
Optionals Well-Designed (390 queries)					
Blazegraph	390	36	0	9.99	2.32
Jena	390	56	0	14.91	4.66
Virtuoso	390	5	1	10.37	7.70
Neo4j	390	113	1	28.21	18.89
Optionals Not Well-Designed (108 queries)					
Blazegraph	108	1	0	3.37	1.89
Jena	108	3	0	8.68	3.46
Virtuoso	108	64	1	42.26	60.00
Neo4j	108	33	0	23.08	5.89

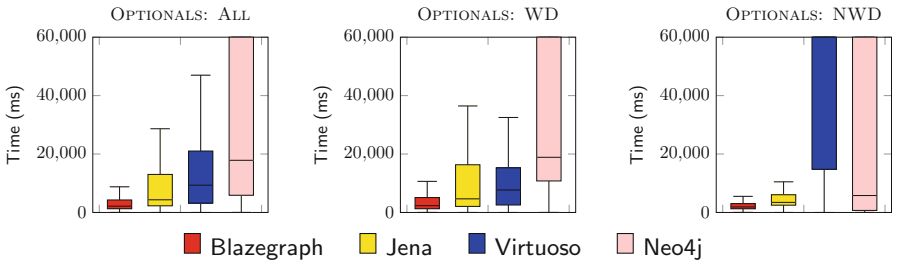

Fig. 2. Performance for Optionals (left), Optionals WD (middle), and Optionals NWD (right)

Table 4. Summary of runtimes (in seconds) for path patterns

Engine	Supported	Timeouts	Error	Average	Median
PATHS (660 QUERIES)					
Blazegraph	660	87	0	11.00	0.82
Jena	660	96	0	11.74	0.81
Virtuoso	660	24	27	4.71	0.70
Neo4j	639	134	6	20.89	9.74
PATHS RECURSIVE (594 QUERIES)					
Blazegraph	594	79	0	11.13	0.78
Jena	594	75	0	10.52	0.62
Virtuoso	594	24	25	4.65	0.43
Neo4j	575	104	5	19.48	9.36
PATHS NON-RECURSIVE (66 QUERIES)					
Blazegraph	66	8	0	9.89	1.19
Jena	66	21	0	22.71	3.04
Virtuoso	66	0	2	5.23	3.72
Neo4j	64	30	1	33.56	42.95

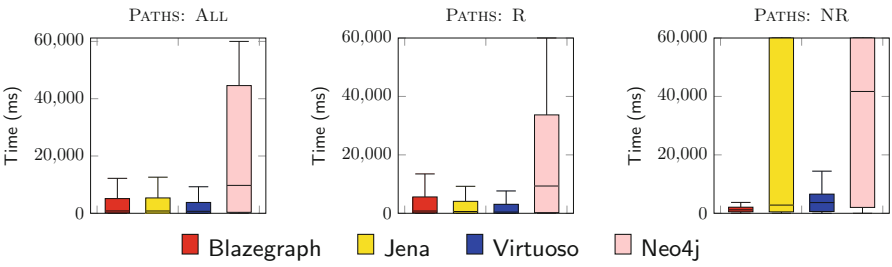

Fig. 3. Performance for PATHS (left), PATHS R (middle), and PATHS NR (right)

5.3 Path Patterns

Considering that property paths are known to give trouble to graph query engines [8], it is interesting to consider their performance in the context of this benchmark. We summarise our findings in Table 4, and in Fig. 3.

Considering all property paths in the PATHS query set, we can see that Virtuoso is the clear winner, both in stability and performance. Both Jena and Blazegraph trail some distance behind, and Neo4j is an order of magnitude slower in the median case. Similarly as in SINGLE, we can observe that the form of the

query (almost identical for all the queries in the set) does not matter much, but that the distribution of the data dictates query performance. We even managed to identify paths which use the exact same regular expression to specify the query, but have a different starting point for the search, where one finishes almost instantaneously, and the other one times out.

When we analyse queries that use recursion, versus the path queries that use no recursion, we can see that all engines except Jena perform similarly in the average case, with Virtuoso being again the most stable and the fastest in terms of the median case. Blazegraph performs better than the other engines in the median case for non-recursive paths. Interestingly, Jena seems to perform better on recursive patterns. Likewise, all systems perform better in the median case for recursive patterns as compared to non-recursive ones. This is a surprising result since one should expect recursive queries to be more costly. In the case of the RDF/SPARQL engines, the SPARQL standard indicates that (most) non-recursive path patterns should be rewritten to BGPs and unions of BGPs, rather than evaluating them directly as paths, meaning that implementations following this strategy will follow very different query evaluation plans when comparing recursive and non-recursive cases.

5.4 Navigational Graph Patterns

The results for Navigational are given in Table 5 and Fig. 4. As before, we provide the results for all the queries in this set, and then analyse the recursive and the non-recursive cases within the set.

When we consider all navigational graph patterns, this set is clearly the most challenging thus far, where we observe the highest average and median runtimes for all engines across all patterns, except in the case of Neo4j, which was slower in the average case for Optionals. Virtuoso is a clear winner in this category, particularly in the average case, although both Blazegraph and Jena come close in terms of median runtimes. Neo4j is again the slowest of all the engines.

When comparing recursive and non-recursive navigational graph patterns, we see different effects on different systems. Blazegraph is slightly slower for non-recursive queries, Jena is notably faster for non-recursive queries, Virtuoso is notably slower for non-recursive queries, and finally Neo4j is considerably slower for non-recursive queries. This is similar to what we observed for Paths, except in the case of Jena, where the trend is reversed. Many queries in Navigational that timed out contain a query in Paths that also times out. This would suggest that an important factor in timeouts is the performance of property paths.

Table 5. Summary of runtimes (in seconds) for navigational graph patterns

Engine	Supported	Timeouts	Error	Average	Median
Navigational (539 queries)					
Blazegraph	539	180	0	22.32	2.58
Jena	539	245	0	30.98	29.83
Virtuoso	539	37	2	10.42	4.36
Neo4j	531	211	0	31.07	24.83
Navigational Recursive (515 queries)					
Blazegraph	515	172	0	22.29	2.58
Jena	515	238	0	31.31	35.47
Virtuoso	515	36	2	10.15	4.03
Neo4j	509	199	0	30.69	24.09
Navigational Non-Recursive (24 queries)					
Blazegraph	24	8	0	22.96	2.94
Jena	24	7	0	23.97	10.07
Virtuoso	24	1	0	16.24	6.48
Neo4j	22	12	0	40.01	60.00

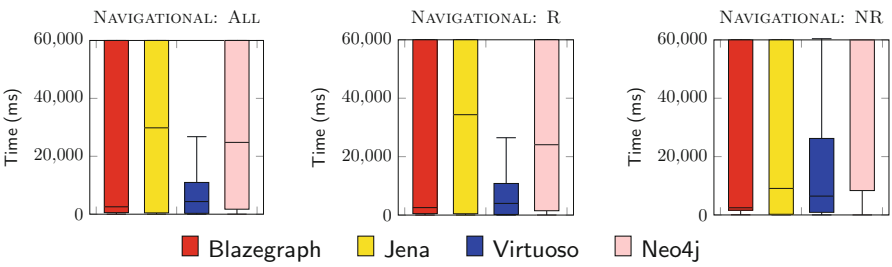

Fig. 4. Performance for Navigational (left), Navigational R (middle), and Navigational NR (right)

6 Conclusions

We conclude with a recap of our contributions, a summary of our results, and a discussion on limitations and future directions.

Contributions: We have developed WDBench: a query benchmark for knowledge graphs based on real-world data (Wikidata) and queries (from Wikidata logs). The benchmark allows for measuring the performance of RDF/SPARQL and graph query engines. In this first release, we have focused on analysing four classes of queries corresponding to core features of graph queries: basic graph patterns, optional graph patterns, path patterns, and navigational graph patterns. We have further partitioned these sets into finer subsets: single vs. multiple, well-designed

vs. not well-designed, and recursive vs. non-recursive. We have published two versions of the benchmark: an RDF/SPARQL version, and a property graph/Cypher version. We have further presented empirical results for the performance of Blazegraph, Jena, Virtuoso and Neo4j using this benchmark.

Results: We observed that Blazegraph and Virtuoso were the best-performing query engines for WDBench, followed by Jena, with Neo4j generally offering the slowest runtimes. Comparing Blazegraph and Virtuoso, the former is slightly faster than the latter for basic graph patterns, considerably faster for optional graph patterns (particularly for not well-designed patterns), considerably slower for path patterns (except the median case of non-recursive queries), and faster in the median case but slower in the average case for navigational graph patterns. In terms of cases where engines could be better optimised, we see that Virtuoso underperforms for not well-designed patterns, while Jena underperforms for non-recursive path queries. Neo4j does not appear to offer competitive performance in the Wikidata setting: while there is the caveat that the query semantics of Cypher varies slightly in some cases from SPARQL, the differences in performance would seem to go beyond such variations; indeed, our results are consistent with previous results for querying Wikidata with Neo4j [24].

Limitations and Future Directions: WDBench currently focuses on core features of graph queries, where languages such as SPARQL and Cypher include a wide range of other features that are frequently used in practice. As part of future work, the same methodology as presented here could be straightforwardly used for generating sets of SPARQL queries using other features and combinations thereof. However, as new features are introduced, it will become increasingly complex to offer analogous versions in Cypher (and other query languages), particularly for features using built-in expressions, such as filters, aggregations, and variable binding. We currently compare the performance of four query engines, but there are other systems that would be interesting to compare in future, including QLever [11], RDF-3x [32] (for SPARQL 1.0), RDF4j [14], etc.[5] Unlike synthetic benchmarks, the scale of WDBench is limited by the size of Wikidata. While it would be possible to test, for example, on the complete version of Wikidata, query results would not change. We view WDBench as a real-world benchmark that can complement other benchmarks, where synthetic benchmarks can be used for stress-testing scalability. Finally, WDBench is a read-only benchmark. An interesting direction for an extended version of the benchmark could include a workload of real-world updates mined from Wikidata [38].

Supplemental Material. The Wikidata graph is available on Figshare [3]. Scripts for data preparation, queries, and detailed results are available on Github [4].

[5] We also have results for MillenniumDB [45], which we do not include here since the system has been developed by the authors. We keep our results third-party.

References

1. Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.-C.N.: A survey of RDF stores & SPARQL engines for querying knowledge graphs. VLDB J. 1–26 (2021). https://doi.org/10.1007/s00778-021-00711-3
2. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 197–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_13
3. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: WDBench: a Wikidata graph query benchmark (2022). https://figshare.com/s/50b7544ad6b1f51de060
4. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: WDBench: a Wikidata graph query benchmark (2022). https://github.com/MillenniumDB/WDBench
5. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1–68:40 (2017)
6. Baeza, P.B., Querying graph databases. In: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA, 22–27 June 2013, pp. 175–188 (2013)
7. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gMark: Schema-driven generation of graphs and queries. IEEE Trans. Knowl. Data Eng. **29**(4), 856–869 (2017)
8. Baier, J.A., Daroch, D., Reutter, J.L., Vrgoc, D.: Evaluating navigational RDF queries over the web. In: Proceedings of the 28th ACM Conference on Hypertext and Social Media, HT 2017, Prague, Czech Republic, 4–7 July 2017, pp. 165–174 (2017)
9. Bail, S., et al.: FishMark: a linked data application benchmark. In: Fokoue, A., Liebig, T., Goodman, E.L., Weaver, J., Urbani, J., Mizell, D. (eds.) Proceedings of the Joint Workshop on Scalable and High-Performance Semantic Web Systems. CEUR Workshop Proceedings, Boston, 11 November 2012, vol. 943, pp. 1–15. CEUR-WS.org (2012)
10. Barceló, P., Kröll, M., Pichler, R., Skritek, S.: Efficient evaluation and static analysis for well-designed pattern trees with projection. ACM Trans. Database Syst. **43**(2), 8:1–8:44 (2018)
11. Bast, H., Buchhold, B.: QLever: a query engine for efficient SPARQL+Text search. In: Lim, E., et al. (eds.) Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, 06–10 November 2017, pp. 647–656. ACM (2017)
12. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. Int. J. Semant. Web Inf. Syst. **5**(2), 1–24 (2009)
13. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. VLDB J. 655–679 (2019). https://doi.org/10.1007/s00778-019-00558-9
14. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: an architecture for storing and querying RDF data and schema information. In: Fensel, D., Hendler, J.A., Lieberman, H., Wahlster, W. (eds.) Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential [Outcome of a Dagstuhl Seminar], pp. 197–222. MIT Press (2003)
15. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Reasoning on regular path queries. SIGMOD Rec. **32**(4), 83–92 (2003)

16. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (2014)
17. Demartini, G., Enchev, I., Wylot, M., Gapany, J., Cudré-Mauroux, P.: BowlognaBench—benchmarking RDF analytics. In: Aberer, K., Damiani, E., Dillon, T. (eds.) SIMPDA 2011. LNBIP, vol. 116, pp. 82–102. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34044-4_5
18. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng. Bull. **35**(1), 3–8 (2012)
19. Erling, O., et al.: The LDBC social network benchmark: interactive workload. In: Sellis, T.K., Davidson, S.B., Ives, Z.G. (eds.) Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, 31 May–4 June 2015, pp. 619–630. ACM (2015)
20. Francis, N., et al.: Cypher: an evolving query language for property graphs. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, 10–15 June 2018, pp. 1433–1445. ACM (2018)
21. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Semant. **3**(2–3), 158–182 (2005)
22. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C Recommendation (2013)
23. Hernández, D., Hogan, A., Krötzsch, M.: Reifying RDF: what works well with Wikidata? In: Liebig, T., Fokoue, A. (eds.) Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems Co-located with 14th International Semantic Web Conference (ISWC 2015), Bethlehem, PA, USA, 11 October 2015, vol. 1457. CEUR Workshop Proceedings, pp. 32–47. CEUR-WS.org (2015)
24. Hernández, D., Hogan, A., Riveros, C., Rojas, C., Zerega, E.: Querying Wikidata: comparing SPARQL, relational and graph databases. In: Groth, P., et al. (eds.) ISWC 2016. LNCS, vol. 9982, pp. 88–103. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46547-0_10
25. Hogan, A., et al.: Knowledge graphs. ACM Comput. Surv. **54**(4), 71:1–71:37 (2021)
26. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11778, pp. 258–275. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30793-6_15
27. Jena Team: TDB Documentation (2021)
28. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoč, D.: SPARQL with property paths. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_1
29. Lehmann, J., et al.: DBpedia - a large-scale, multilingual knowledge base extracted from Wikipedia. Semant. Web **6**(2), 167–195 (2015)
30. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the most out of Wikidata: semantic technology usage in Wikipedia's knowledge graph. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11137, pp. 376–394. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00668-6_23
31. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.-C.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In: Aroyo, L., et al. (eds.) ISWC 2011. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_29
32. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)

33. Pérez, J., Arenas, M., Gutiérrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16:1–16:45 (2009)
34. Romero, M.: The tractability frontier of well-designed SPARQL queries. In: den Bussche, Arenas, M. (eds.) Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, 10–15 June 2018, pp. 295–306. ACM (2018)
35. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.-C.N.: LSQ: the linked SPARQL queries dataset. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 261–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_15
36. Saleem, M., Mehmood, Q., Ngonga Ngomo, A.-C.: FEASIBLE: a feature-based SPARQL benchmark generation framework. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 52–69. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_4
37. 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: The World Wide Web Conference, pp. 1623–1633. ACM (2019)
38. Schmelzeisen, L., Dima, C., Staab, S.: Wikidated 1.0: an evolving knowledge graph dataset of Wikidata's revision history. In: Kaffee, L., Razniewski, S., Hogan, A. (eds.) Proceedings of the 2nd Wikidata Workshop (Wikidata 2021) Co-located with the 20th International Semantic Web Conference (ISWC 2021), Virtual Conference, 24 October 2021, vol. 2982. CEUR Workshop Proceedings. CEUR-WS.org (2021)
39. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^2Bench: a SPARQL performance benchmark. In: Ioannidis, Y.E., Lee, D.L., Ng, R.T. (eds.) Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, 29 March 2009–2 April 2009, Shanghai, China, pp. 222–233. IEEE Computer Society (2009)
40. Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The train benchmark: cross-technology performance evaluation of continuous model queries. Softw. Syst. Model. **17**(4), 1365–1393 (2017). https://doi.org/10.1007/s10270-016-0571-8
41. The Wikimedia Foundation. Wikidata: Database download (2021)
42. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata® RDF graph database. In: Harth, A., Hose, K., Schenkel, R. (eds.) Linked Data Management, pp. 193–237. Chapman and Hall/CRC, Boca Raton (2014)
43. Vandenbussche, P., Umbrich, J., Matteis, L., Hogan, A., Aranda, C.B.: SPARQLES: monitoring public SPARQL endpoints. Semant. Web **8**(6), 1049–1065 (2017)
44. Vrandecic, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014)
45. Vrgoc, D., et al.: MillenniumDB: a persistent, open-source, graph database. CoRR, abs/2111.01540 (2021)
46. Webber, J.: A programmatic introduction to Neo4j. In: Leavens, G.T. (ed.) Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH 2012, Tucson, AZ, USA, 21–25 October 2012, pp. 217–218. ACM (2012)
47. Wikimedia Foundation: Wikidata SPARQL Logs (2022). https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en
48. Wu, H., Fujiwara, T., Yamamoto, Y., Bolleman, J.T., Yamaguchi, A.: BioBenchmark Toyama 2012: an evaluation of the performance of triple stores on biological data. J. Biomed. Semant. **5**, 32 (2014)