Evaluation of a Representative Selection of SPARQL Query Engines using Wikidata

First Author[0000−1111−2222−3333], Second Author[1111−2222−3333−4444], and Third Author[2222−−3333−4444−5555]

Princeton University, Princeton NJ 08544, USA
lncs@springer.com

Abstract. In this paper, we present an evaluation of the performance of five representative RDF triplestores, including GraphDB, Jena Fuseki, Neptune, RDFox and Stardog, and one experimental SPARQL query engine, QLever. We compare importing time, loading time and exporting time using a complete version of the knowledge graph Wikidata, and we also evaluate query performances using 328 queries defined by Wikidata users. To put this evaluation into context with respect to previous evaluations, we also analyse the query performances of these systems using a prominent synthetic benchmark: SP^{2}Bench. We observed that most of the systems we considered for the evaluation were able to complete the execution of almost all the queries defined by Wikidata users before the timeout we established. We noticed, however, that the time needed by most systems to import and export Wikidata might be longer than required in some industrial and academic projects, where information is represented, enriched and stored using different representation means.

Keywords: RDF Triplestores · Knowledge Graphs · SPARQL Benchmarks · SP^{2}Bench · Wikidata.

1 Introduction

Wikidata [33] is a collaboratively edited multilingual knowledge graph hosted by the Wikimedia Foundation. Wikidata is becoming a prominent software artifact in academia and industry that offers a broad collection of terms and definitions that can improve data understandability, integration, and exchange. Wikidata is stored as an RDF [34] graph that can be queried with the SPARQL language [35]. With more than 16 billion triples and 100 million defined terms, as the size of the knowledge graph continuously increases, it might be challenging for state-of-the-art triplestores to import, export, and query Wikidata. This is also acknowledged by Wikimedia Foundation which is looking for alternatives to replace Blazegraph [11], an open-source triplestore no longer in development [38].

To investigate how efficiently RDF triplestores can handle Wikidata, in this paper, we present an evaluation of the performance of five representative RDF triplestores, including Ontotext GraphDB [19], Apache Jena Fuseki [5], Amazon Neptune [9], OST RDFox [23], and Stardog [30], and one experimental SPARQL

query engine - QLever [8]. We compare importing, loading and exporting time using a complete version of Wikidata, and we also evaluate query performances using 328 queries defined by Wikidata users.

Due to budget limitations, we limited our evaluation to only six representative tools. However, we tried to ensure a diverse selection. For instance, GraphDB, Neptune, RDFox, and Stardog are commercial applications, whereas QLever and Jena Fuseki are not. Neptune is based on Blazegraph and it is the only triplestore that is available as a native cloud-based service. RDFox is an in-memory triplestore, while the others are persistent. QLever and Jena Fuseki are in the pool of tools considered by Wikimedia Foundation to replace Blazegraph. So we thought both could represent a good baseline for our study.

After some deliberation, we also decided to conduct an evaluation of the six triplestores using the synthetic benchmark *SP²Bench* [27], which has strong theoretical foundations and a main focus on query optimization, also very relevant in our analysis of query performances of SPARQL engines using Wikidata. Budget restrictions prevented us from conducting additional evaluations using, for instance, a representative benchmark based on real-world datasets.

In our evaluation using a full version of Wikidata and large datasets generated by SP²Bench, we study the query execution plan with query profiling information to better understand the difference in the performance of the triplestores on the same query. The ultimate goal is to obtain a thorough understanding of the impact of SPARQL features on the performance and confirm common best practices in the design of SPARQL queries. We also consider import and export time. As service-oriented and decentralized architecture has become a popular design for software infrastructure in recent years, importing and exporting performance may be critical to enable efficient data transformation and exchange, especially for big data applications (e.g., big data pipelines or Machine Learning pipelines). Therefore, it is important to optimize importing and exporting functionality to avoid bottlenecks in the execution of such applications.

Despite not considering the evaluation of concurrent execution of SPARQL queries, we observed that most of the systems selected for the evaluation could complete the execution of almost all queries defined by Wikidata users before the timeout. We noticed, however, that the time needed by most applications to import and export a full Wikidata version might be longer than required in many industrial and academic projects, where information is represented, enriched, and stored using a diverse selection of applications offering different representation means. To help interested readers to dive into the details of this evaluation, all scripts, data and results have been uploaded to an open repository [40].

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 describes the evaluation setup. Section 4 provides a detailed discussion of the evaluation results using SP²Bench and Wikidata. Finally, Section 5 concludes the paper and presents future work directions.

2 Related Work

To better relate our evaluation using Wikidata with previous evaluation papers, we reviewed existent benchmarks based on both synthetic and real-world datasets. Benchmarks that use synthetic datasets include *LUBM* (Lehigh University Benchmark) (2005) [15], *UOBM* (University Ontology Benchmark) (2006) [17], *BSBM* (Berlin SPARQL Benchmark) (2009) [10], *SP²Bench* (SPARQL Performance Benchmark) (2009) [27], *Bowlognabench* (2011) [12], *WatDiv* (Waterloo SPARQL Diversity Test Suite) (2014) [1], *LDBC-SNB* (Linked Data Benchmark Council - Social Network Benchmark) (2015) [13], *TrainBench* (2018) [32] and *OWL2Bench* (2020) [28]. The sizes of the synthetic datasets used in the referenced papers ranged from 1M to 100M triples, and the numbers of SPARQL queries executed were between 12 and 29, with the exception of *WatDiv* that used a set of 12500 generated queries.

Amongst benchmarks that are based on real-world datasets or queries from real-world logs, we reviewed *DBPSB* (DBpedia SPARQL Benchmark) (2011) [18], *FishMark* (2012) [7], *BioBenchmark* (2014) [39], *FEASIBLE* (2015) [25], *WGPB* (Wikidata Graph Pattern Benchmark) (2019) [16] and *WDBench* (2022) [4]. The datasets for these benchmarks varied in size from 14M up to 8B triples, and the numbers of SPARQL queries defined were between 22 and 175, except for *WGPB* and *WDBench* that have 850 and more than 2000 queries respectively.

The study conducted by the Wikimedia Foundation to replace Blazegraph is the closest work we have been able to identify so far. This study provided a detailed analysis of relevant features of triple stores according to Wikimedia Foundation. This study, however, only considered open-source triplestores and it did not include execution times for importing, loading, exporting, or querying Wikidata [36]. Another related study [14] discussed the possibility of hosting a full version of Wikidata, and it measured import time of popular triplestores including Jena Fuseki, QLever, and Stardog. This study, however, did not discuss export time or query performances. *WGPB* and *WDBench* rely on a substantially reduced version of full Wikidata and they have very specific objectives. *WGPB* defines a large set of SPARQL basic graph patterns exhibiting a variety of increasingly complex join patterns for testing the benefits of worst-case optimal join algorithms. The design goal behind *WDBench* was to create an evaluation environment able to test not only graph databases supporting RDF data model and SPARQL query language. The authors of *WDBench* created a collection of more than 2000 SPARQL queries distributed in four different categories. These queries were selected from real Wikidata query-logs. Due to budget limitations, we discard the possibility of including the queries defined by *WDBench* in our study, but we would like to include them in an extended version of this evaluation and compare the results with the queries we selected.

3 Evaluation Setup

After discussing relevant related work, we start our presentation of the details of the evaluation by describing the operational setup.

Triplestores To ensure that our limited selection of triple stores is representative and diverse, the following triplestores were evaluated: (1) Jena Fuseki 4.4.0 with Jena TDB2 triplestore, (2) Amazon Neptune Engine 1.0.5.1, (3) GraphDB Enterprise Edition 9.10.0, (4) RDFox 5.4, (5) QLever (commit version 742213facfcc80af11dade9a971fa6b09770f9ca), and (6) Stardog 7.8.0. In this selection: there are commercial and non-commercial (Jena Fuseki and QLever) applications; there is one triplestore distributed as native cloud-service (Neptune); and there is one in-memory triplestore (RDFox). All triplestores support SPARQL 1.1 syntax and provide querying services via SPARQL endpoints.

Datasets We aim to evaluate the scalability and performance of the SPARQL query engines using large datasets. For SP^{2}Bench, we generated four different datasets with 125M, 250M, 500M, and 1B triples. For Wikidata, we used the full version `latest-all.nt.gz` (downloaded on 2021-11-19). Table 1 shows the general statistics of these datasets.

Table 1. Statistics of the datasets: number of distinct Triples, Sub[jects], Pred[idcates], Obj[ects], Class[es], Ind[ividuals], Obj[ect] Prop[erties] and Data Prop[erties].

Benchmark	Triples	Sub	Pred	Obj	Class	Ind	Obj Prop	Data Prop
SP^{2}Bench	125M	22.4M	78	59.5M	19	22.4M	64	21
	250M	45.9M	78	120.8M	19	46.2M	64	21
	500M	94M	78	244.9M	19	94.1M	64	21
	1B	190.3M	78	493M	19	190.5M	64	21
Wikidata	16.3B	1.78B	42.92K	2.93B	1.2K	1.77K	17.1K	27K

SPARQL Queries SP^{2}Bench comes with a set of 14 SELECT and 3 ASK queries which were designed to cover several relevant SPARQL constructs and operators as well as to provide diverse execution characteristics in terms of difficulty and result size [27]. For Wikidata, the set of 356 SPARQL query examples defined by Wikidata users [37] was selected. Some of these queries use proprietary service extensions deployed for the Wikidata Query Service. We modified the queries to not use these service extensions and discarded some queries that are not compliant with SPARQL 1.1 specification or use proprietary built-in functions not supported by the evaluated triplestores. As a result, a set of 328 queries is used for the evaluation.

Table 2. Coverage (%) of SPARQL features for each benchmark.

Benchmark	distinct	filter	optional	union	limit	order	bound	offset
SP^{2}Bench	35.29	58.82	17.65	17.65	5.88	11.76	11.76	5.88
Wikidata	33.14	30.84	31.12	5.19	14.7	48.7	2.59	0
	DateFnc	**SetFnc**	**NumFnc**	**StringFnc**	**TermFnc**	**exists**	**notexists**	**in**
SP^{2}Bench	0	0	0	0	0	0	0	0
Wikidata	8.65	27.67	2.59	9.8	16.14	0.58	3.75	1.44
	groupby	**bind**	**values**	**minus**	**coalesce**	**if**	**having**	**PropPath**
SP^{2}Bench	0	0	0	0	0	0	0	0
Wikidata	29.11	10.66	8.65	5.19	0.86	3.17	1.44	35.73

Table 2 presents an analysis of the queries from the two benchmarks regarding the SPARQL features and operators. These features may have a correlation

with the execution time of the queries [24,26]. Hence, they need to be taken into consideration when designing a SPARQL query benchmark. It can be observed that Wikidata queries provide a broader coverage of SPARQL features and operators than SP^{2}Bench. Wikidata queries have more advanced features [35] such as Property Path, or built-in functions such as Dates and Times (DateFnc), Set (SetFnc), Strings (StringFnc), and RDF Terms (TermFnc) functions.

Amazon Web Services (AWS) Infrastructure This evaluation was conducted on the AWS cloud. We used Amazon Elastic Compute Cloud (EC2) instances with Elastic Blob Store (EBS) volumes. Taking both budget limitation and resource requirements into consideration, `r5` instances with memory configurations between 128GB and 512GB were selected. This corresponds with instances of type `r5.4xlarge`, `r5.8xlarge`, `r5.16xlarge` [2]. This choice also matched the on-demand `r5` instances available for the fully managed Neptune triplestore [3]. As RDFox is an in-memory triplestore, it needs additional memory to load a full version of Wikidata. None of the available `r5` instances offers enough memory for RDFox. Therefore, `x1` instances which offer up to 1,952 GB of memory were selected instead, in particular `x1.32xlarge` [2] was employed to evaluate RDFox using Wikidata. Each EC2 instance was set up with a separate EBS `gp3` volume for data storage with the performance of 3,000 IOPS and 125 MB/s throughput.

Configuration Details We followed the recommended memory configuration for Stardog [29] and GraphDB [20], and applied it to all triplestores. We used the default settings for other configurations. In the case of RDFox, this implies that we use a persistence mode that stores incremental changes in a file [22]. RDFox can also be set up to run purely in-memory. According to the vendor, this would result in much lower import and export times than the ones presented in the paper. Similarly, RDFox offers the possibility to store datastores as binary files [21], which might significantly reduce loading times. These claims could not be verified before this study was submitted.

The evaluation was carried out simultaneously with one triplestore running on one instance. For SP^{2}Bench, `r5.8xlarge` was used to deploy all triplestores. For Wikidata, we ran the evaluation on `r5.4xlarge`, `r5.8xlarge`, and `r5.16xlarge`, except RDFox that was deployed only on `x1.32xlarge`. Due to some differences in the hardware configuration of `r5` and `x1`, we performed a sensitivity analysis of their performance. The result of this analysis is discussed in Section 4.2.

To avoid the impact of network latency, the triplestores (i.e., SPARQL server) and the evaluation scripts (i.e., SPARQL client) were deployed on the same machine. Neptune is provided as database-as-a-service in the cloud. Thus, the SPARQL client needs to run on a separate machine. To estimate the effect of network latency, we set up a test with GraphDB where the SPARQL client was running on a separate machine. This analysis helped us to adjust and make the results for Neptune comparable with the others. Detail about this analysis is discussed in Section 4.2.

The evaluation is comprised of the following stages:

1. **Data Import**. All datasets were imported into the selected triplestores.
2. **System Restart and Warm-up**. The triplestores were restarted, and the evaluated dataset was loaded again (if needed). Then, one test query was executed to warm up the triplestores.
3. **Hot-run**. Each query was executed ten times. We set the query timeout to 30 minutes for the SP^{2}Bench and 5 minutes for the Wikidata.

The following metrics are recorded in this evaluation:

- **Import Time**. The time required to import the dataset for the first time. This step involves building indexes and persisting the datasets to storage.
- **Load Time**. The time required to load the dataset after importing it.
- **Export Time**. The time required to write imported data to an external file.
- **Query Execution Time**: The time needed to finish one query execution.
- **Success Indicators**. The numbers of success, error, and timeout queries.
- **Global Performance**. We follow the proposal in [27] to compute both well-known arithmetic mean and geometric mean (the n^{th} root of the product over n number) of the execution times. Accordingly, the failed queries (e.g., timeout, error) were penalized with the double of timeout value. Arithmetic mean is used as an indicator of a high success and failure ratio (i.e., a smaller value indicates a higher ratio of success queries) while geometric mean is used to evaluate the overall performance over success queries (i.e, a smaller value as an indicator of shorter execution time for the success queries).

Although we ran our experiments on a cloud-based framework, it is worth mentioning that the executions of each run are remarkably consistent. Specifically, the standard deviation of the ten runs of 95% of the Wikidata queries is less than one millisecond.

4 Discussion of the Evaluation Results

In this section, we discuss the evaluation results using SP^{2}Bench and Wikidata. All experimental results, including runtimes for individual queries on each engine tested, can be found online [40].

4.1 Evaluation results using SP^{2}Bench

Import Time Table 3 includes the import time of SP^{2}Bench datasets. For all triplestores, the import time increases proportionally to the size of the dataset. Jena Fuseki showed poor import performance even though the `tdb2.xloader` [6] - a multi-threading bulk loader for very large datasets - was employed.

RDFox is the fastest when importing the datasets, even though it was configured using persistence mode. RDFox also exhibited similar loading times. As discussed in Section 3, RDFox importing and loading times might be reduced using a different configuration. With regards the other triplestores, they show similar import times and very fast loading times. For instance, they were able to load the synthetic dataset with 1B triples in less than a minute.

Table 3. Global performance of the triplestores on SP^{2}Bench. To compute the mean, Timeout and Error queries were penalized with 3600 seconds (1 hour).

Triple stores	125M			250M			500M			1B		
	Imp Time	Arith Mean	Geo Mean	Imp Time	Arith Mean	Geo Mean	Imp Time	Arith Mean	Geo Mean	Imp Time	Arith Mean	Geo Mean
QLever	17m	1694.23	3.96	35m	1694.23	4.10	1h9m	1694.24	4.24	2h20m	1702.58	**5.74**
Fuseki	33m	1089.78	14.04	1h6m	1324.24	23.18	2h15m	1370.08	29.28	5h40m	1816.33	45.03
Neptune	18m	898.35	7.11	37m	974.22	9.89	1h16m	1323.48	13.83	2h22m	1781.18	26.40
RDFox	**3m**	1061.38	**1.75**	**5m**	1065.10	**2.42**	**9m**	1074.52	**2.91**	**18m**	1528.20	6.98
Stardog	18m	862.86	2.76	37m	878.88	3.62	1h17m	**917.33**	5.54	2h33m	1378.62	9.10
GraphDB	17m	**728.62**	3.97	36m	**766.16**	5.00	1h11m	995.00	7.55	2h23m	**1248.67**	11.75

Table 4. Success indicators (S[uc]C[ess], T[ime]O[ut], ERR[or]) on SP^{2}Bench.

Triplestore	125M			250M			500M			1B		
	SC	TO	ERR	SC	TO	ERR	SC	TO	ERR	SC	TO	ERR
QLever	9	0	8	9	0	8	9	0	8	9	0	8
Jena Fuseki	12	5	0	11	6	0	11	6	0	9	8	0
Amazon Neptune	13	4	0	13	4	0	11	6	0	9	8	0
RDFox	12	5	0	12	5	0	12	5	0	10	7	0
Stardog	13	4	0	13	4	0	13	4	0	11	6	0
GraphDB	15	2	0	15	2	0	14	3	0	13	4	0

Query Execution Time Table 3 and 4 present success indicators and average execution time for the four SP^{2}Bench datasets. QLever is the only triplestore that had errors and no timeout. Seven queries could not be executed due to unsupported syntax or functions (e.g., ASK query, combined conditions in FILTER) and one "OutOfMemory" (OOM) error.

According to Table 4, the size of the dataset and results has a significant effect on the performance of all triplestores, in particular as the dataset grows from 125M to 1B triples. Neptune, as mentioned earlier, may suffer from network latency, especially for queries with large results because the server and client were deployed on different machines. For SP^{2}Bench, the timeout was set to 30 minutes which is relatively long enough for transferring big data between Amazon machines. In fact, in 8 timeout cases, Neptune failed to finish the execution of the queries and returned no result. Therefore, network latency is not the main issue for these timeout queries. Moreover, compared to the average execution time of the 17 SP^{2}Bench queries which is about 15 minutes, network latency may be considered as an insignificant factor. A thorough evaluation of network latency will be discussed in the next section with the Wikidata benchmark where there are a lot of queries executed in less than 100 milliseconds.

Regarding the global performance, the arithmetic means of GraphDB were superior to the others since it had a higher number of success queries. However, RDFox had better performance over successful queries, so its geometric means were the smallest; timeouts were limited to queries that introduce equi-joins using FILTER statements. In all cases, Stardog was always in the top two. It had more success queries than RDFox and executed difficult queries slightly faster than GraphDB. Jena Fuseki delivered the poorest performance while Neptune had mixed results on query execution. QLever was very fast on success queries, but it offered limited support for queries with complex SPARQL constructs.

Moreover, QLever automatically puts a limit up to 100.000 results for all queries. Therefore, its reported execution times may not be comparable, especially for queries with large results.

Analysis of Query Execution Plan A SPARQL query can be represented as a Basic Graph Pattern (BGP) which is a set of Triple Patterns (TPs) specified in the query [24]. Typically, the result of a BGP is obtained by joining the results of the TPs. Therefore, selective TPs that have smaller result sizes are usually executed first in order to minimize the number of intermediate results and therefore reduce the cost related to joining operations [16]. For the same purpose, filters are also moved closer to the part of the BGP where they apply. The execution order of the TPs in the BGP is the query execution plan. Typically, the query execution plan needs to be decided before the SPARQL engine executes the query. In order to do this, the triplestore requires precise estimation of the result size of each TP, which is done through building and updating different types of indexes [16]. In general, different triplestores may employ different data structures to implement their indexes and use different algorithms to optimize their query execution plan, therefore resulting in varying performance. In this study, except for QLever which does not provide the method to get the query plan, we investigated the execution plans of the other triplestores in order to gain a better understanding of their performances on the benchmarks.

In addition to the execution plan, RDFox, Neptune, and especially, Stardog also include very comprehensive profiling reports with the actual result sizes and the execution time of each TP. However, GraphDB provides only the estimation of the result sizes while Jena Fuseki produces only the complete results of each TP. Therefore, for these two triplestores, it was difficult to diagnose performance problems or identify expensive operations for the difficult queries.

Query 2 is one of the most difficult queries of the SP^{2}Bench where many triplestores timed out. Query 2 has a bushy BGP (i.e., single nodes that are linked to a multitude of other nodes) with 10 TPs, and the result size of this query grows with database size. Therefore, the execution time might be linear to the dataset size. According to the statistics provided by Stardog, the most expensive operation for this query is post-processing data (i.e., converting the results into the data structure that will be sent to the client). Similarly, Amazon Neptune also spent on `TermResolution` operator (e.g., translating internal identifiers to external string values). This observation illustrates the effect of large result sizes and large strings on the querying performance of SPARQL engines.

Query 3 (a, b, c) has just two TPs, one of which is of the form (?,?,?), and one FILTER with "equal to" operator. To avoid evaluating the TP (?,?,?) which may result in matching the whole dataset, all triplestores embedded the filter expression into this TP and transformed it into (?,p,?) form.

Query 4 is the most challenging query of the SP^{2}Bench benchmark. The result of the query is expected to be quadratic in the number of "journal" individuals in the dataset. To deal with this query, the author in [27] suggested that the query engines embed the FILTER expression into the computation of TPs (i.e., the same approach done in Query 3), which may help to reduce the

intermediate results earlier. However, it is more challenging for all triplestores to embed the "greater than" operator in this query than the "equal to" operator. As a result, all of the triplestores failed to complete the query due to timeout.

Query 5a and 12a also test optimizations for embedding FILTER expression. The expression in these queries is the "equal to" comparison between two variables while Query 3 filters a variable with a constant value. GraphDB is the only triplestore handling Query 5a in 30 minutes. After rewriting this query by explicitly embedding the filtering expression, the triplestores can execute the query before timeout. However, as the dataset increased to 1B, timeout still occurred for the others. Query 12a replaces the SELECT construct of Query 5a with the ASK, which has a positive effect on query performances in all triplestores with the exception of Qlever and RDFox. This might indicate that collecting the results of Query 5a also has a significant impact on performances.

Query 6, 7, and 8 test another different optimization approach related to reusing TP results. These queries have several TPs repeated multiple times. Thus, intermediate results of those TPs can be reused to save cost for matching those triples. From the execution plan, it is unclear whether the triplestores implement this optimization approach or the same TPs were executed again.

Overall, when evaluating the execution plans, we observed that the triplestores did not pass several optimization tests designed by the authors of the benchmark. Despite being a synthetic benchmark with only 17 queries where some tests may not be practical (e.g., Query 4) or biased towards some specific constructs (e.g., FILTER), SP^{2}Bench proved to be very useful to test common query optimization techniques and to collect useful insights of the triplestores selected for this evaluation. Next, we will present the evaluation results using a completed version of Wikidata and 328 queries defined by its user community.

4.2 Evaluation results using Wikidata

Import and Export Time The Wikidata dump used in this evaluation is available as a 112GB gzip file (738GB as unzip file). QLever is the only triplestore that has no support for gzip format. However, due to errors during importing, only Stardog, GraphDB, and RDFox managed to load the gzip file. Jena TDB2, in particular, was not able to import Wikidata due to 1319 URI syntax errors (e.g., special characters not allowed by Jena RIOT - the Jena syntax validator). After fixing these errors by replacing the special characters with their HTML numeric codes, we used this "clean" version to import into Jena Fuseki. Jena Fuseki also suffered from OOM error and succeeded in importing Wikidata only on `r5.16xlarge` machine.

Table 5 presents the performance for importing and exporting Wikidata. RDFox was much faster than the others. This result is consistent with the figures reported in SP^{2}Bench where all triplestores are evaluated using the same machine configuration. However, as the triplestores were restarted, RDFox required around 3.75 hours (40% faster than its initial import time) to reload the data while the others took only a few minutes. As discussed in Section 3, RDFox importing and loading times might be reduced using a different configuration.

Table 5. Import and Export performance of the triplestores on Wikidata.

Triplestore	VM	Import Time	(Re)Load Time	Export Time	Persisted Storage
QLever	r5.4xlarge	1d 17h 2m	11m	n/a	871 GB
Jena Fuseki (TDB2)	r5.16xlarge	3d 15h 27m	<1m	Timeout	1.52 TB
Stardog	r5.4xlarge	2d 1h 9m	<1m	Error	862 GB
Amazon Neptune	r5.4xlarge	3d 1h 50m	7m	Error	3.98 TB
GraphDB	r5.4xlarge	1d 8h 13m	10m	Timeout	1.11 TB
RDFox	x1.32xlarge	0d 6h 25m	3h42m	5h28m	202 GB

As QLever, Jena Fuseki, and Amazon Neptune used the unzip data and may not require any decompress operation during importing, their performance is expected to be slower on gzip Wikidata. Also, we observed that QLever reported a much larger number of loaded triples (21.5B triples). It is not enough to state whether QLever did not import Wikidata properly or the triplestore has a different way of building the statistics on the imported data.

To measure the export time, we set a timeout of 4 days for the triplestores. Except for QLever which has no support for data exporting, the other triplestores provide native functions to export the data. However, RDFox is the only triplestore that succeeded in exporting Wikidata within the timeout. Stardog did not show any progress or runtime output while Amazon Neptune encountered an error after exporting 503M statements in 1.5 hours. GraphDB took 28 days and 8 hours to export Wikidata. Due to cost constraints, we did not continue the exporting process for the others after 4 days. Based on this figure, it is obvious that exporting is not the prioritized feature of most triplestores.

Query Execution Time Table 6 presents the success indicators and mean execution time of the triplestores on the Wikidata benchmark. QLever reported the most errors (67% of all queries). Most of them are syntax errors due to limited support for the SPARQL 1.1 syntax. Furthermore, there are 5 queries where QLever returns only the first 100.000 results. They were also classified as errors. QLever also has several OOM errors that were resolved on more powerful machines.

Jena Fuseki is the second triplestore with the most errors. It had 13 query syntax errors. Particularly, it does not allow using an existing variable name for the AS operator (e.g., `(SAMPLE(?dob) AS ?dob)`). Jena Fuseki also suffered from memory issues. This triplestore either crashed or froze and produced no output while executing the other 8 error queries. Amazon Neptune and Stardog accounted for one error which was reported as an internal failure exception.

If we look at the performance of each individual triplestore on the three different `r5` configurations, GraphDB and Stardog were more robust with small variances among the machines. They had approximately the same query execution time and number of timeouts on the three machines. They may be optimized to work efficiently even on machines with less physical resources. In contrast, Jena Fuseki and Amazon Neptune performed better on more powerful machines as they had more success queries on those machines.

Table 6. Global performance of the triplestores on Wikidata benchmark. To compute the mean, Timeout and Error queries were penalized with 600 seconds (10 minutes). The table also contains the estimated mean (in brackets) for RDFox on `r5` machine and for Amazon Neptune 16x with network latency deducted.

Triplestore	SC	TO	ERR	Arithmetic Mean	Geometric Mean
Qlever 4x	106	0	222	404.87	12.33
Qlever 8x	107	0	221	403.05	11.91
Qlever 16x	108	0	220	401.23	11.73
Jena Fuseki 4x	224	83	21	192.30	1.43
Jena Fuseki 8x	231	76	21	180.21	1.29
Jena Fuseki 16x	250	57	21	148.57	1.20
Amazon Neptune 4x	309	18	1	39.29	0.34
Amazon Neptune 8x	310	17	1	36.26	0.31
Amazon Neptune 16x	312	15	1	31.65 (31.59)	0.28 (0.27)
Stardog 4x	307	20	1	43.46	0.19
Stardog 8x	308	19	1	42.01	0.16
Stardog 16x	308	19	1	41.77	0.18
GraphDB 4x	321	7	0	15.62	0.08
GraphDB 8x	322	6	0	14.48	0.08
GraphDB 16x	321	7	0	15.67	0.07
RDFox 32x	324	4	0	12.11 (9.23)	0.04 (0.016)

When comparing the execution time of all triplestores, RDFox and GraphDB are also the top two triplestores with arithmetic mean of around 14 seconds compared to around 31 seconds for Amazon Neptune. Jena Fuseki and QLever are the slowest triplestores due to a lot of errors and timeouts. Regarding geometric mean, RDFox is still the fastest with a value of 0.04 which is 50% smaller than the value of GraphDB. Stardog is in the third place. Its geometric mean is around 0.18 which is 35% faster than Amazon Neptune. This insight can also be noticed from Figure 1 that compares the best performance (i.e., on `r5.16xlarge` machines) of the triplestores for the top 50 easy and difficult queries (excluding error queries). Accordingly, for easy queries, it can be clearly identified the order of the triplestores where RDFox is the fastest and Amazon Neptune is the slowest. However, there is a mixed result for difficult queries.

Network Latency Analysis In order to estimate network latency which can be incorporated into the execution time of Amazon Neptune, especially for those queries with large results, a separate experiment was conducted. Specifically, the same setting was deployed for GraphDB on two `r5.4xlarge` machines. On average, the latency amounted to 100 milliseconds. In Figure 1(a) and Table 6, we also have the execution time of Amazon Neptune adjusted by removing the latency for each individual query. As the latency is just a few milliseconds for easy queries, the adjusted figure is not different from the original value. Amazon Neptune is still the slowest on the top 50 easy queries.

Sensitivity Analysis of R5 and X1 Instances To compare the performance of the triplestores on `x1` and `r5` instances, we performed a sensitivity analysis that: (1) evaluates the performance of GraphDB with Wikidata on `x1e.4xlarge` and `r5.4xlarge` machines, (2) evaluates the performance of RD-

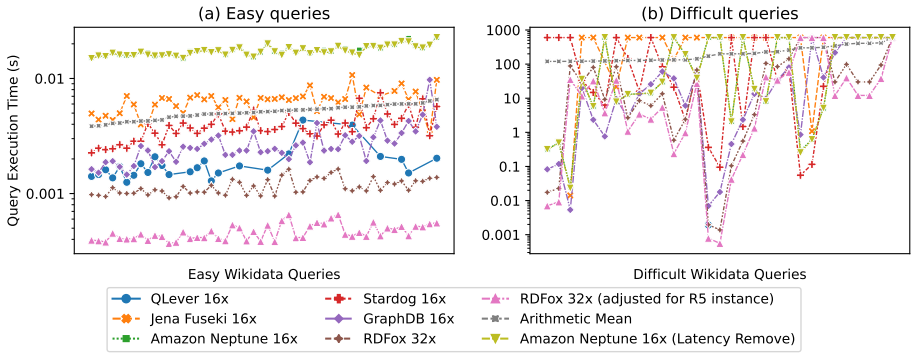

Fig. 1. Query execution time of the top 50 easy and difficult Wikidata queries on average (excluding error queries). The queries (x axis) are ordered by the arithmetic mean of the execution time of all triplestores.

Fox with SP^{2}Bench on `x1e.8xlarge` and `r5.8xlarge` machines. On average, GraphDB had approximately 50% performance degradation on `x1e.4xlarge` machine. Similarly, the performance of RDFox on SP^{2}Bench decreased about 59.75% on `x1e.8xlarge` machine. Thus, it is expected that RDFox may have better performance on `r5` instances for Wikidata queries if a suitable `r5` machine with sufficient RAM memory is available. To ensure the results of this analysis are reflected in the evaluation, in Table 6, we also provide the adjusted means for RDFox assuming the queries would be executed 59.75% on average faster.

Analysis of Query Execution Plan To better understand the evaluation results discussed earlier, several queries are studied in more detail. In particular, the following selection criteria were applied:

1. Queries not executed by most of the triple stores due to timeout.
2. Queries with large variation in execution times (i.e., timeout for some triplestores and executed in few seconds by the rest).
3. Queries where the numbers of results are not consistent (i.e., some triplestores returned different numbers of results for the same query).

Due to the page limitation, only a summary of the analysis is presented in this section. Firstly, for timeout queries, the most expensive operations are related to processing a large number of results, including both intermediate and final results. Therefore, to minimize overhead related to handling the results, highly selective triple patterns are usually prioritized to be executed earlier in order to reduce the scanning space for later triple patterns of the query. To do so, the query optimizer needs to have a good estimation of the outputs for each TP in the query. For queries with simple SPARQL constructs, most triplestores can manage to create an optimal execution plan. However, as the query employs complex constructs or features (e.g., nested SELECT query, built-in functions, property path, etc.), estimating a good execution plan becomes challenging. Based on the analysis of the query execution plans, the following observations can be considered when designing the queries for the evaluated triplestores:

- For queries with complicated patterns, most triplestores tend to keep the execution plan the same as the original order described in the query. In this case, it is recommended to rewrite the query using simple constructs. For example, the property path with arbitrary length can be rewritten explicitly with a sequence of TPs and UNION. For example, after we rewrite the property path of **Query 234** and **Query 235** with the sequence of three patterns explicitly, Stardog can produce a better plan and result in 2.6x and 4.2x faster respectively. If the query cannot be changed, the TPs need to be re-ordered manually to help the triplestores to optimize the execution plan. This can be done by repeating the execution with different orders [31].
- FILTER operations are usually moved earlier in the execution plan in order to reduce the number of intermediate results. However, filters with complex conditions (e.g., string and date-time functions) may slow down the execution, especially for a large number of results (e.g., **Query 192**, **Query 327**, **Query 343** and **Query 350**). If possible, such condition should be rewritten in an equivalent form without using FILTER or applied later in the execution plan when there are fewer intermediate results.
- The TP of the form (?,?,?) should be considered carefully as it may result in a bottleneck in the execution of most triplestores (e.g., **Query 45**).

For queries with large variation in execution times, the following observations about the evaluated triplestores are recorded:

- Due to its in-memory solution, RDFox tends to perform better on scanning indexes and joining results, especially for large results or complicated operations such as string functions (e.g., **Query 13**, **Query 233**, **Query 237** and **Query 350**).
- Jena Fuseki had very poor performance on scanning and joining large results compared with the others. It also has very simple optimization algorithms. Mostly, it does not change the order of triple patterns, which results in inefficient execution plans and timeouts (e.g., **Query 45**, **Query 84** and **Query 286**). Therefore, the triple patterns need to be re-ordered manually in order to improve the performance of this triplestore.
- Stardog may have issues with queries having many OPTIONAL constructs. The triplestore tends to produce exponential intermediate results when matching such triple patterns (e.g., **Query 176** and **Query 326**) while the others produced much fewer results, and therefore resulting in timeout.
- For queries with sequence paths, Amazon Neptune tends to prioritize triple patterns with such property path syntax. This strategy may result in an exponential increase in the intermediate results if those patterns are not the most selective (e.g., **Query 84** and **Query 286**).
- For queries with UNION construct, RDFox tends to keep the triple patterns inside UNION unchanged while GraphDB tends to expand this construct by moving the JOIN operation inside each of the operands of UNION. Stardog and Amazon Neptune are more flexible in estimating the optimal plan for the UNION pattern. Therefore, if the optimal plan can be anticipated, it is

recommended to rewrite the UNION patterns, especially for GraphDB and RDFox (e.g., **Query 184** and **Query 345**).

The third category of queries selected for further investigation is the one where the numbers of results are not consistent. Accordingly, we identified 10 queries where there is one triplestore that disagreed with the majority of other triplestores. As we only captured and compared the number of results, it is not sufficient to determine whether a triplestore is correct or not. However, if most of the triplestores report the same number of results, this might be a reasonable indication in terms of correctness. Based on the analysis of the execution plan, the following issues from the triplestores were identified:

- Amazon Neptune reported different results when executing a few queries with REGEX expressions. For instance, the triplestore returned no result after applying some filters with complex regex patterns (e.g., **Query 93**, **Query 133** and **Query 327**).
- Stardog reported different numbers of results for a few SELECT nested queries (e.g., **Query 82** and **Query 195**). Additionally, in **Query 284** which has a FILTER operator on date-time values, Stardog returned six more results than the others.
- GraphDB also had issues with a few nested SELECT queries. It returned no result for **Query 109** and **Query 319**. Additionally, in **Query 178** and **Query 233**, the triplestore returned much fewer results than the others.

5 Conclusions and Future Work

To the best of our knowledge, this study presents one of the most detailed analyses of the performances of a representative selection of the state-of-the-art triplestores using a complete version of the knowledge graph Wikidata. In this section, we conclude the paper with a summary of some of the most relevant observations produced by this evaluation.

With respect to the evaluation setup used in this study, all selected triplestores were tested on Amazon EC2 `r5` and `x1` instances. Despite some initial concerns about the reliability of the evaluation results, the execution times of each run were remarkably consistent. Amazon EC2 instances are also required to test Neptune, the only native cloud-based service in our evaluation. Some specific execution requirements posed by Neptune and RDFox difficult a fair comparative analysis. Sensitivity analyses were conducted to adjust the evaluation results for Neptune and RDFox and make them comparable with the others. While the impact of network latency in the Neptune client-server configuration seems to be small, the differences in performances between `r5` and `x1` instances seem to be significant (approximately 50% to 60% performance degrade).

SP^{2}Bench proved to be a great choice to test scalability and common query optimization techniques, which helps us to collect useful insights of the triplestores selected for this evaluation. RDFox was the fastest triplestore importing the synthetic datasets generated for this study and it has better performance

over success queries. Regarding the global query performances, GraphDB was superior to the others, followed very close by Stardog. After analyzing query execution plans and query profiling information, we observed that the triplestores did not pass several optimization tests designed by the authors of SP^{2}Bench. It is worth mentioning the excellent query profiling service implemented by Stardog, which establishes a reference for other triplestores.

SP^{2}Bench also has some limitations. It only provides 17 SPARQL queries offering limited coverage of SPARQL constructs and features. Moreover, some of these queries and the synthetic datasets do not seem to be practical in real use case applications. Our evaluation employing a complete version of Wikidata with 328 queries defined by its users seems to overcome these limitations. This evaluation helps us to stress the triplestores and identify relevant insights. Importing Wikidata, and especially, exporting Wikidata was challenging for all triplestores, where RDFox was significantly more efficient. Loading Wikidata, however, was done much faster by the other triplestores, although a different configuration for RDFox might reduce export and loading time significantly. Exporting Wikidata was an even bigger challenge. RDFox was the only triplestore that managed to export Wikidata, and it completed this operation in a few hours.

Importing Wikidata was also difficult because of syntax errors reported by some rigorous parsers such as the ones implemented in RDFox and Jena Fuseki. It seems that the complete dumps published by Wikidata might not strictly follow W3C recommendations. For instance, it was possible to find values not formatted according to these recommendations. The same problem arises with some queries published by Wikidata users. Some of these queries use, for example, proprietary service extensions deployed by the Wikidata Query Service team.

In terms of query performances, RDFox reported the best overall performances followed by GraphDB. It is remarkable how consistent GraphDB and Stardog were, in terms of query performances independent of the memory configuration of the machine. This indicates a careful optimization of the design of both triplestores in terms of memory consumption. With the exception of QLever and Jena Fuseki, most triplestores reported none or just one error in the execution of the queries. Few discrepancies in the number of results were identified in the case of Stardog, GraphDB, and Neptune. The cause of these discrepancies could not be explained in this study and it will require further investigation.

Due to budget limitations, we could not evaluate a larger collection of relevant triplestores or extend the queries used in the Wikidata evaluation with the queries defined by the benchmark WDBench. We plan to do this in future work.

Acknowledgment

The authors would like to thank Ontotext and Oxford Semantic Technologies (OST) for their support during the evaluation. This work has been funded by several public projects that we temporarily omitted for the review process.

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: International Semantic Web Conference. pp. 197–212. Springer (2014)
2. Amazon Web Services: Amazon EC2 Instance Types - Memory Optimized. https://aws.amazon.com/ec2/instance-types/#Memory_Optimized, accessed: 2022-12-12
3. Amazon Web Services: Amazon Neptune Pricing. https://aws.amazon.com/neptune/pricing/, accessed: 2022-12-12
4. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: WDBench: A Wikidata Graph Query Benchmark. In: International Semantic Web Conference. Springer (2022)
5. Apache Jena: Apache Jena Fuseki Documentation, https://jena.apache.org/documentation/fuseki2/
6. Apache Jena: Apache Jena TDB xloader. https://jena.apache.org/documentation/tdb/tdb-xloader.html, accessed: 2022-12-12
7. Bail, S., Alkiviadous, S., Parsia, B., Workman, D., Van Harmelen, M., Concalves, R., Garilao, C.: FishMark: A Linked Data Application Benchmark. CEUR (2012)
8. Bast, H., Buchhold, B.: QLever: A Query Engine for Efficient SPARQL+Text Search. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management. pp. 647–656 (2017)
9. Bebee, B.R., Choi, D., Gupta, A., Gutmans, A., Khandelwal, A., Kiran, Y., Mallidi, S., McGaughy, B., Personick, M., Rajan, K., et al.: Amazon Neptune: Graph Data Management in the Cloud. In: ISWC Posters & Demonstrations, Industry and Blue Sky Ideas Tracks. CEUR-WS.org (2018), http://ceur-ws.org/Vol-2180/paper-79.pdf
10. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal on Semantic Web and Information Systems (IJSWIS) **5**(2), 1–24 (2009)
11. Blazegraph: Blazegraph Official Website, https://blazegraph.com/
12. Demartini, G., Enchev, I., Wylot, M., Gapany, J., Cudré-Mauroux, P.: BowlognaBench—Benchmarking RDF Analytics. In: International Symposium on Data-Driven Process Discovery and Analysis. pp. 82–102. Springer (2011)
13. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.: The LDBC Social Network Benchmark: Interactive Workload. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 619–630 (2015)
14. Fahl, W., Holzheim, T., Westerinen, A., Lange, C., Decker, S.: Getting and hosting your own copy of Wikidata. In: Proceedings of the 3rd Wikidata Workshop 2022. CEUR-WS.org (2022), https://ceur-ws.org/Vol-3262/paper9.pdf
15. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3**(2-3), 158–182 (2005)
16. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A Worst-Case Optimal Join Algorithm for SPARQL. In: International Semantic Web Conference. pp. 258–275. Springer (2019)
17. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a Complete OWL Ontology Benchmark. In: European Semantic Web Conference. pp. 125–139. Springer (2006)
18. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In: International semantic web conference. pp. 454–469. Springer (2011)

19. Ontotext: GraphDB Official Website, https://graphdb.ontotext.com/
20. Ontotext: GraphDB Requirements. https://graphdb.ontotext.com/documentation/enterprise/requirements.html, accessed: 2022-12-12
21. OST: RDFox Documentation: Managing Data Stores. https://docs.oxfordsemantic.tech/5.4/data-stores.html#, accessed: 2022-12-12
22. OST: RDFox Documentation: Operations on Data Stores, persist-ds. https://docs.oxfordsemantic.tech/5.4/data-stores.html#persist-ds, accessed: 2022-12-12
23. Oxford Semantic Technologies: RDFox Official Website, https://www.oxfordsemantic.tech/product
24. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.C.N.: LSQ: The Linked SPARQL Queries Dataset. In: International Semantic Web Conference. pp. 261–269. Springer (2015)
25. Saleem, M., Mehmood, Q., Ngonga Ngomo, A.C.: FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In: International Semantic Web Conference. pp. 52–69. Springer (2015)
26. Saleem, M., Szárnyas, G., Conrads, F., Bukhari, S.A.C., Mehmood, Q., Ngonga Ngomo, A.C.: How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks. In: The World Wide Web Conference. pp. 1623–1633 (2019)
27. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^2Bench: A SPARQL Performance Benchmark. In: 2009 IEEE 25th International Conference on Data Engineering. pp. 222–233. IEEE (2009)
28. Singh, G., Bhatia, S., Mutharaju, R.: OWL2Bench: A Benchmark for OWL 2 Reasoners. In: International Semantic Web Conference. pp. 81–96. Springer (2020)
29. Stardog: Stardog Capacity Planning. https://docs.stardog.com/operating-stardog/server-administration/capacity-planning, accessed: 2022-12-12
30. Stardog: Stardog Official Website, https://www.stardog.com/
31. Stardog: 7 Steps to Fast SPARQL Queries. https://www.stardog.com/blog/7-steps-to-fast-sparql-queries/ (May 2017), accessed: 2022-12-12
32. Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The Train Benchmark: cross-technology performance evaluation of continuous model queries. Software & Systems Modeling **17**(4), 1365–1393 (2018)
33. Vrandečić, D., Krötzsch, M.: Wikidata: A Free Collaborative Knowledgebase. Communications of the ACM **57**(10), 78–85 (2014)
34. W3C: RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014. https://www.w3.org/TR/rdf11-concepts/, accessed: 2022-12-12
35. W3C: SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013. https://www.w3.org/TR/sparql11-query/, accessed: 2022-12-12
36. WDQS Search Team: WDQS Backend Alternatives: The process, details and result. Tech. rep., Wikimedia Foundation (2022), https://www.wikidata.org/wiki/File:WDQS_Backend_Alternatives_working_paper.pdf
37. Wikidata: SPARQL query service/queries/examples. https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples, accessed: 2022-12-12
38. Wikidata: SPARQL query service/WDQS backend update. https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/WDQS_backend_update, accessed: 2022-12-12
39. Wu, H., Fujiwara, T., Yamamoto, Y., Bolleman, J., Yamaguchi, A.: BioBenchmark Toyama 2012: an evaluation of the performance of triple stores on biological data. Journal of Biomedical Semantics **5**(1), 1–11 (2014)

40. Zenodo: Analysis and supplementary information for the paper, including queries, execution logs, query results and scripts. https://doi.org/10.5281/zenodo.7452322, accessed: 2022-12-12

Appendix

This section provides additional details about the evaluation results that we cannot include in the main paper due to page limitation.

EC2 Instances

Table 7 and 8 present the hardware and volume configurations of the EC2 instances used in the evaluation. The selection criteria of these instances were discussed in Section 3.

Table 7. Hardware details for the EC2 instances used in the evaluation.

Instance Type	Processor(s)	Instance Size	vCPU	Memory (GB)	Networking Performance (Gbps)
R5 (db.R5)	Up to 3.1 GHz Intel Xeon Platinum 8000 series	r5.4xlarge	16	128	Up to 10
		r5.8xlarge	32	256	10
		r5.16xlarge	64	512	20
X1/X1e	2.3 GHz Intel Xeon E7 8880 v3	x1e.4xlarge	16	488	Up to 10
		x1e.8xlarge	32	960	Up to 10
		x1.32xlarge	128	1 952	25

Table 8. Storage configuration for different triplestores. The size of the volumes varied for the different triplestores and was calculated based on the memory requirements found in the documentation. For the SP^{2}Bench datasets, we used a volume size of 500 GB across the triplestores. For the Wikidata dataset, larger volumes were required.

Triplestore	EC2 instance		EBS volume size (GB)	
	SP^{2}Bench	Wikidata	SP^{2}Bench	Wikidata
QLever	r5.8xlarge	r5.4xlarge r5.8xlarge r5.16xlarge	500	1000
Fuseki				1800
Stardog				1000
GraphDB				1600
Neptune				n/a
RDFox		x1.32large		300

SP^{2}Bench Result

Table 9 presents the query execution time of the triplestores for the four SP^{2}Bench datasets. The table connects to the discussion presented in Section 4.1

Detailed Analysis of SP^{2}Bench Query Execution Plan

This section provides an analysis of the execution plans of the SP^{2}Bench queries in more detail. A simplified discussion was presented in Section 4.1.

Query 2 (Figure 2) is one of the difficult queries of the SP^{2}Bench. Most triplestores failed to accomplish this query within 30 minutes for the four datasets.

Query	125M						250M					
	QL	JF	AN	RFx	SD	GDB	QL	JF	AN	RFx	SD	GDB
Q1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Q2	1.7	TO	TO	TO	TO	TO	1.7	TO	TO	TO	TO	TO
Q3a	0.1	40.1	8.5	4.4	10.2	7.3	0.1	96.2	14.7	9.1	20.1	12.6
Q3b	0.0	6.2	0.1	0.0	0.1	0.1	0.0	10.2	0.1	0.0	0.1	0.1
Q3c	0.0	6.2	0.3	0.0	0.1	0.1	0.0	9.9	0.6	0.1	0.1	0.3
Q4	ERR	TO	TO	TO	TO	TO	ERR	TO	TO	TO	TO	TO
Q5a	ERR	TO	TO	TO	TO	1800	ERR	TO	TO	TO	TO	1800
Q5b	0.1	209	27.7	7.6	74.3	46.7	0.1	316	149	12.7	140	85.3
Q6	ERR	TO	TO	TO	TO	1800	ERR	TO	TO	TO	TO	1800
Q7	ERR	TO	383	26.5	56.0	1384	ERR	TO	1067	74.7	124	1800
Q8	ERR	0.1	0.2	0.0	0.0	0.7	ERR	0.1	0.2	0.0	0.0	0.7
Q9	0.0	158	255	3.1	117	133	0.0	330.0	548	6.1	236	296
Q10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Q11	0.0	80.9	197	1.8	11.5	15.0	0.0	150	383	3.8	21.0	29.3
Q12a	ERR	25.8	0.1	TO	0.0	0.0	ERR	TO	0.1	TO	0.0	0.0
Q12b	ERR	0.0	0.1	0.0	0.0	0.1	ERR	0.0	0.0	0.0	0.0	0.1
Q12c	ERR	0.0	0.0	0.0	0.0	0.0	ERR	0.0	0.0	0.0	0.0	0.0

Query	500M						1B					
	QL	JF	AN	RFx	SD	GDB	QL	JF	AN	RFx	SD	GDB
Q1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Q2	1.7	TO	TO	TO	TO	TO	143.5	TO	TO	TO	TO	TO
Q3a	0.1	142.7	26.4	15.5	38.8	20.0	0.1	TO	TO	TO	76.0	TO
Q3b	0.0	19.6	0.2	0.1	0.2	0.2	0.1	37.8	0.5	0.1	0.3	0.4
Q3c	0.0	19.2	1.3	0.2	0.2	0.5	0.0	36.7	2.7	0.4	0.3	1.0
Q4	ERR	TO	TO	TO	TO	TO	ERR	TO	TO	TO	TO	TO
Q5a	ERR	TO	TO	TO	TO	1800	ERR	TO	TO	TO	TO	1800
Q5b	0.1	528	123	22.2	269	TO	0.1	TO	TO	TO	TO	TO
Q6	ERR	TO	TO	TO	TO	1800	ERR	TO	TO	TO	TO	1800
Q7	ERR	TO	TO	209	298	1800	ERR	TO	TO	739	745	1800
Q8	ERR	0.1	0.4	0.0	0.0	0.7	ERR	0.1	0.4	0.0	0.0	0.7
Q9	0.0	686	TO	12.3	550	636	0.0	1430	TO	25.0	1015	1314
Q10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Q11	0.0	296	748	7.2	38.7	57.2	0.0	573	1476	14.6	TO	112
Q12a	ERR	TO	0.0	TO	0.0	0.0	ERR	TO	0.1	TO	0.0	0.0
Q12b	ERR	0.0	0.0	0.0	0.1	0.1	ERR	0.0	0.0	0.0	0.0	0.1
Q12c	ERR	0.0	0.0	0.0	0.0	0.0	ERR	0.0	0.0	0.0	0.0	0.0

Table 9. SP^{2}Bench Query execution time (seconds) of the triplestores: QL[ever], J[ena] F[useki], A[mazon] N[eptune], R[D]F[o]x, S[tar]D[og], G[raph]DB. Timeout: 30 minutes (greater than 1800 seconds). Values less than 9 ms are reported as 0.0 s.

Query 2 has a bushy BGP (i.e., single nodes that are linked to a multitude of other nodes) with 10 TPs, 10 projection variables, OPTIONAL, and ORDER BY operators. The result size of this query grows with database size. Therefore, the execution time might be linear to the dataset size. Jena Fuseki had the execution plan in the same order as the original query while the other evaluated triplestores managed to reorder the TPs based on their selectivity estimations. As all the triplestore timed out, there is not enough information to evaluate which is the optimal execution plan for this query. According to the statistics provided by Stardog, the most expensive operation for this query is post-processing data (i.e., converting the results into the data structure that will be sent to the client). Similarly, Amazon Neptune also spent on `TermResolution` operator (e.g., translating internal identifiers to external string values). This observation again confirms the effect of large result size and large strings on the querying performance of SPARQL engines.

```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL {
    ?inproc bench:abstract ?abstract
  }
}
ORDER BY ?yr
```

Fig. 2. SP^{2}Bench - Query 2.

```
(a)
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:pages)
}

(b) Q3a, but "swrc:month" instead of "swrc:pages"
(c) Q3a, but "swrc:isbn" instead of "swrc:pages"
```

Fig. 3. SP^{2}Bench - Query 3.

Query 3 (a, b, c) (Figure 3) has just two TPs one of which is of the form (?,?,?), and one FILTER with "equal to" operator. To avoid evaluating the TP (?,?,?) which may result in matching the whole dataset, all triplestores embedded the filter expression into this TP and transformed it into (?,p,?). According to [27], the filtering results are very diverse in the three variants of the query (i.e., 92.61% of all articles for Q3a, 0.65% for Q3b, and no result for Q3c). Therefore, Q3a is expected to have execution time linear to the dataset size while Q3b and Q3c can be evaluated in nearly constant time. For Q3b and Q3c, the execution plans of almost all triplestores are very scalable to the dataset sizes. However, Jena Fuseki still showed bad performance even on those simple queries. It executed Q3b and Q3c in 6.2 seconds on the 125M dataset and up to 38 seconds on the 1B dataset while the others finished their execution in less than 3 seconds. Regarding Q3a, according to the number of results of the two TPs, keeping the original order of the TP seems to be a better execution plan. RDFox is the only triplestore that reordered the patterns and it outperformed the other triplestores on 125M, 250M, and 500M datasets. As RDFox is an in-memory triplestore, it has fewer I/O operations, which may be the bottleneck for scanning the indexes and matching TPs. However, when the dataset grows to 1B triples, except for Stardog, the other triplestores failed to complete the execution. Although Stardog was in the fourth position in the first three datasets, it demonstrated a good

scaling factor on Q3a. Its performance degraded approximately by a factor of 2 when the dataset doubled the size and amounted to 76 seconds for 1B dataset.

```
SELECT DISTINCT ?name1 ?name2
WHERE {
  ?article1 rdf:type bench:Article .
  ?article2 rdf:type bench:Article .
  ?article1 dc:creator ?author1 .
  ?author1 foaf:name ?name1 .
  ?article2 dc:creator ?author2 .
  ?author2 foaf:name ?name2 .
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal .
  FILTER (?name1<?name2)
}
```

Fig. 4. SP^{2}Bench - Query 4.

Query 4 (Figure 4) is the most challenging query of the SP^{2}Bench benchmark. This query has 8 TPs together with a DISTINCT modifier and a FILTER. The result of the query is expected to be quadratic in the number of journals in the dataset. According to the statistics provided by RDFox execution plan, for the 125M triples, there were 1.1B results before applying the filter to make 577M final results. To deal with this query, the author in [27] suggested that the query engines embed the FILTER expression into the computation of TPs (i.e., the same approach done in Query 3), which may help to reduce the intermediate results earlier. However, it is more challenging for all triplestores to embed the "greater than" operator in this query than the "equal to" operator. As a result, all of the triplestores failed to complete the query due to timeout. QLever, however, had a memory overflow error.

```
(a)
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person2 .
  ?person foaf:name ?name .
  ?person2 foaf:name ?name2 .
  FILTER (?name=?name2)
}

(b)
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person .
  ?person foaf:name ?name
}
```

Fig. 5. SP^{2}Bench - Query 5.

Query 5a (Figure 5) is another test for embedding FILTER expression into the TPs. The expression in this query is the "equal to" comparison between two variables while Query 3 filters a variable with a constant value. Although there were about 7.3M final results, most triplestores failed to finish this query. GraphDB is the only triplestore handling this query in 30 minutes. After rewriting this query by explicitly embedding the filtering expression, the triplestores can execute the query within the requested time. However, as the dataset increased to 1B, timeout still occurred for all triplestores. The analysis of Query 12a indicates that post-processing data, which is required to return the results of the query, is also a very expensive operation that has a significant impact on the execution time.

```
SELECT ?yr ?name ?document
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
  ?document dc:creator ?author .
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document .
    ?document2 rdf:type ?class2 .
    ?document2 dcterms:issued ?yr2 .
    ?document2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr)
  } FILTER (!bound(?author2))
}
```

Fig. 6. SP^{2}Bench - Query 6.

```
SELECT DISTINCT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?doc rdf:type ?class .
  ?doc dc:title ?title .
  ?bag2 ?member2 ?doc .
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document .
    ?doc3 rdf:type ?class3 .
    ?doc3 dcterms:references ?bag3 .
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document .
      ?doc4 rdf:type ?class4 .
      ?doc4 dcterms:references ?bag4 .
      ?bag4 ?member4 ?doc3
    } FILTER (!bound(?doc4))
  } FILTER (!bound(?doc3))
}
```

Fig. 7. SP^{2}Bench - Query 7.

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes rdf:type foaf:Person .
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string .
  {
    ?document dc:creator ?erdoes .
    ?document dc:creator ?author .
    ?document2 dc:creator ?author .
    ?document2 dc:creator ?author2 .
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?document2!=?document &&
            ?author2!=?erdoes &&
            ?author2!=?author)
  } UNION {
    ?document dc:creator ?erdoes.
    ?document dc:creator ?author.
    ?author foaf:name ?name
    FILTER (?author!=?erdoes)
  }
}
```

Fig. 8. SP^{2}Bench - Query 8.

Query 6, 7, and 8 (Figure 6, 7, 8) test another different optimization approach related to reusing TP results. Accordingly, those queries have several TPs repeated multiple times. Therefore, intermediate results from those patterns can be reused to save cost for matching those triples. From the execution plan, it is unclear whether the triplestores implement this optimization approach or the same TPs were executed again. Regarding the execution time, GraphDB is the only triplestore that completed Query 6. For Query 7, RDFox, Stardog, and GraphDB were in the top 3. Query 8 has a FILTER with a condition comprising multiple expressions. One promising optimization approach for this filter is decomposing the expressions and applying them separately to appropriate TPs. RDFox, Stardog, and Jena Fuseki applied this strategy while Amazon Neptune and GraphDB computed the condition as a whole.

```
(a) Q5a as ASK query
(b) Q8 as ASK query
(c) ASK { person:John_Q_Public rdf:type
                               foaf:Person }
```

Fig. 9. SP^{2}Bench - Query 12.

Query 12 (Figure 9) replaces the SELECT construct of queries 5a and 8 with the ASK construct. As the ASK form tests whether or not a query pattern has a solution, the triplestores need not to construct all the results for the query. Therefore, executing those queries could be much faster. Accordingly, Neptune, Stardog and GraphDB completed Query 12a within 0.1 second while timed out on Query 5a. Jena Fuseki took 25.8 and RDFox still timed out for Query 12a. Furthermore, as Query 8 is an easy query, most triplestores succeeded in execut-

ing its ASK version - Query 12b - in less than 0.1. Jena Fuseki and GraphDB were the two triplestores accounting for most substantial improvement. Specifically, Jena Fuseki completed Query 12b approximately 10 times faster than Query 8 while the figure for GraphDB was 7 times.

Wikidata Result

This section includes additional figures that support the discussion on the evaluation using Wikidata. Accordingly, Figure 10 presents the query execution time of each individual triplestore on the three EC2 instances. Detail about this discussion was presented in Section 4.2. Figure 11 illustrates the result of the network latency analysis discussed in Section 4.2. Finally, Figure 12 illustrates the result for the sensitivity analysis between `r5` and `x1` instances discussed in Section 4.2.

Detailed Analysis of Wikidata Query Execution Plan

In this section, we include the analysis of the execution plans of Wikidata queries. Summary of the findings was introduced in Section 4.2. Before investigating the query execution plans, we also performed an analysis on the impact of SPARQL query features on the query execution time. Particularly, we calculated Spearman's correlation of the query features with the query runtime for each triplestore. Figure 13 shows the correlation values of the top 10 query features. The result in this figure was used as a reference for the features to be considered when evaluating the query execution plan.

Queries not executed by most of the triplestores due to timeout Firstly, regarding the difficult queries, most of the triplestores failed to finish their execution within 5 minutes of timeout. For triplestores such as Stardog, RDFox, and Amazon Neptune, the execution plans also include the execution time and the actual number of results of each triple pattern of the query. This is done by the query profilers of the triplestores and thereby requiring the queries to be executed successfully. Therefore, in order to get the query execution plans from these triplestores, a timeout of 3 hours was set for those query profilers. As not every query can be finished within the timeout, we do not have all execution plans with complete statistics for the selected query. Also, QLever does not provide any method to obtain the execution for each individual query. Therefore, the discussion in this section may not include execution plans of all the triplestores for some queries. It is also worth mentioning that we use the native Command Line Interface (CLI) of Stardog and RDFox to execute and generate the execution plan without storing the query results in an external file. The query profiler is executed via the SPARQL endpoint for Amazon Neptune. Therefore, for some timeout queries (especially the ones with large number of results), Stardog and RDFox can finish the execution of the query much faster than their performances in the experiment reported above.

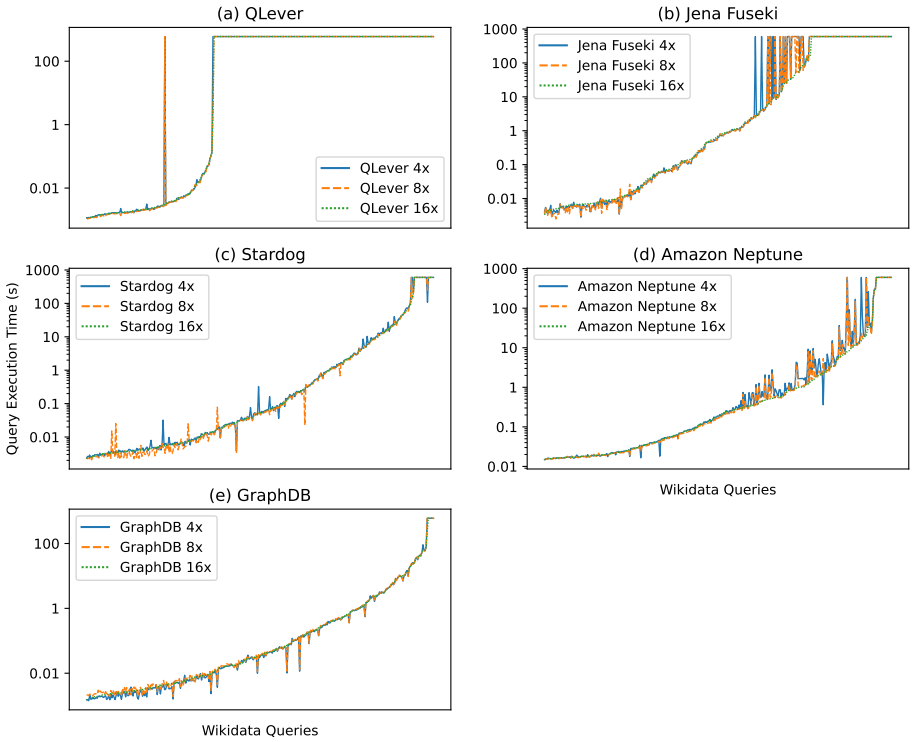

Fig. 10. Wikidata Query execution time (seconds) of each triplestore in different machine configurations. Timeout and Error queries were penalized with 600 seconds (10 minutes). In each plot, the queries (x axis) are ordered by the execution time on `r5.16xlarge` machine. As can be seen from the figure, for GraphDB and Stardog, there was no big difference between the execution time on the three machines. When it comes to difficult queries with long execution time, the difference becomes even smaller. These two triplestores may be optimized to work efficiently even on machines with less physical resources. QLever also has very consistent execution time between the three machines except for several spikes due to insufficient memory on machines with less RAM. In contrast, Jena Fuseki and Amazon Neptune performed much faster on machines with more powerful configurations. Accordingly, these two triplestores have more success queries on higher configuration machines.

Query 45 (Figure 14) is one of the most difficult queries where all triplestores timed out. This query has only four triple patterns together with GROUP BY, ORDER BY and COUNT operators. The most expensive triple pattern of this query 45 is the one with the form (?,?.?). In order to avoid scanning over the whole data with more than 16 billion triples, this triple pattern needs to be put to later in the execution plan to reduce the scanning space. Except for Jena Fuseki which kept the execution plan as the original order, the other triplestores re-

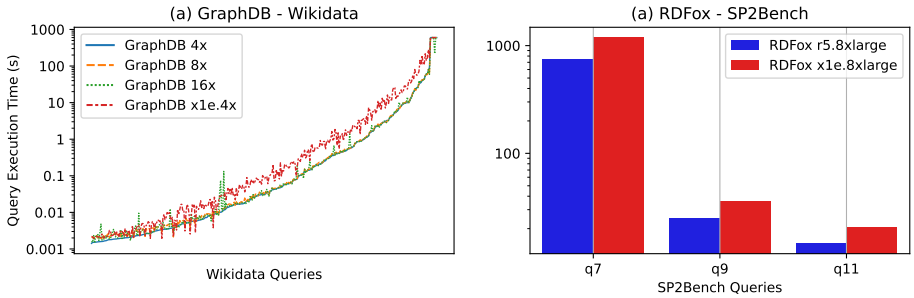

Fig. 11. Performance of (a) GraphDB on Wikidata queries and (b) RDFox on SP^{2}Bench for different EC2 instances. Accordingly, the performance of GraphDB and RDFox on the `x1e` machines was clearly slower than the ones on `r5` instances. On average, GraphDB had approximately 50% performance degrade on `x1e.4xlarge` machine. Similarly, the performance of RDFox on SP^{2}Bench decreased about 59.75% on `x1e.8xlarge` machine.

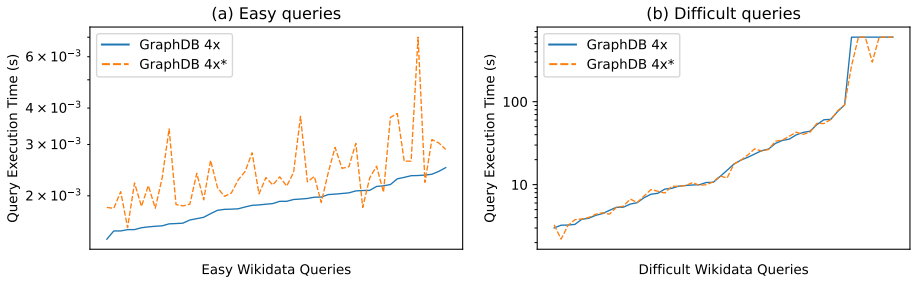

Fig. 12. Wikidata query execution time of GraphDB for top 50 easy and difficult queries on two different SPARQL client-server deployment settings: on one machine (GraphDB 4x) and two different machines (GraphDB 4x*). As easy queries usually have small results, the latency is approximate 5 milliseconds. For difficult queries, the number is expected to be larger. However, compared to the actual executing time of the triplestore, latency may be insignificant for those difficult queries. On average, the latency amounted to 100 milliseconds.

ordered the triple patterns. Accordingly, GraphDB identified the subject results (variable `?s`) by executing the triple pattern `?s wdt:P31/wdt:P279* wd:Q729` first while RDFox chose to collect the properties (variable `?pd`). Amazon Neptune and Stardog, on the other hand, determined the object results (variable `?o`) before executing the (?,?.?) pattern. As Stardog and Amazon Neptune executed this query within 30 minutes while RDFox could not finish after more than 2 hours, the execution plan of Stardog and Amazon Neptune may be a better strategy. However, for such a big dataset, the best practice is to avoid such (?,?,?)

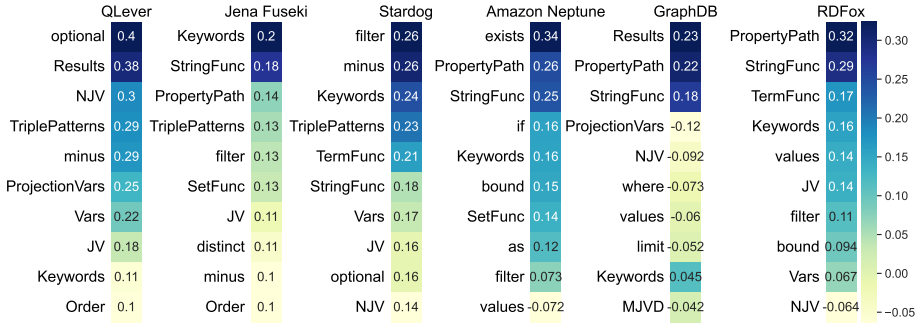

Fig. 13. Pearson correlation between top 10 query features/keywords and query execution times of the triplestores on Wikidata benchmark. There are several features such as property path syntax (ProperyPath), number of results (Results), total number of query SPARQL keywords or constructs (Keywords), number of triple patterns (TriplePatterns), number of projection variables (ProjectionVars), number of variables in the BGP (Vars), built-in functions (set functions, string functions), number of join vertices (JV), number of non join vertices (NJV), mean join vertex degree (MJVD), etc. are in the top features with high correlation values for most triplestores. Overall, there is no single feature that has a strong correlation with the query runtimes. According to [26], the query runtime is affected by a combination of different features.

```
select ?p (count (*) as ?count) {
  ?s ?pd ?o .
  ?p wikibase:directClaim ?pd .
  ?s wdt:P31/wdt:P279* wd:Q729 .
  ?o wdt:P31/wdt:P279* wd:Q43501 .
} group by ?p order by desc(?count)
```

Fig. 14. Wikidata - Query 45: Properties connecting items of type zoo (Q43501) with items of type animal (Q729).

pattern, for example, by rewriting the pattern with UNION of (?,pd,?) patterns if there is a limited number of the property `pd`. By doing so, the query optimizer can be able to estimate the optimal execution plan for the query. Furthermore, this query contains two sequence paths with arbitrary length (i.e., with "/" and "*") which can be transformed into undefined number of triple patterns with the same property. Typically, the triplestores define a looping mechanism of scanning for the same property to handle such arbitrary path length. Alternatively, if the length can be estimated (e.g., from the statistics of the dataset), the triple with property path syntax can be rewritten with equivalent sequence of triple patterns. As can be seen from Figure 13, Property Path may contribute considerably to the execution of the query due to an undetermined loop. Therefore, in order to avoid such overhead, it is better to explicitly write the sequence of triple patterns instead of using property path syntax if the length of the path is known.

```
SELECT ?doi (COUNT (?entry) as ?entries)
{
  ?entry ?p ?statement .
  ?statement prov:wasDerivedFrom/pr:P248/wdt:P356 ?doi .
}
GROUP BY ?doi
ORDER BY DESC(?entries)
```

Fig. 15. Wikidata - Query 234: The number of statements by DOI.

```
SELECT (COUNT (?statement) as ?statements)
WHERE
{
  ?entry ?p ?statement .
  ?statement prov:wasDerivedFrom/
       <http://www.wikidata.org/prop/reference/P248>/
       wdt:P356 ?doi .
}
```

Fig. 16. Wikidata - Query 235: Number of statements backed by a reference with a DOI.

Query 234 and Query 235 (Figure 15, 16) are also the most difficult queries where all triplestores either timed out or encountered "Out Of Memory" error. These two queries have only two triple patterns, but the second one has a sequence path of three properties. Most triplestores managed to rewrite this property path into a sequence of three triple patterns. These two queries also have a triple with the (?,?,?) pattern. As discussed earlier, this triple pattern should not be the first one in the execution plan. Similar in Query 45, Jena Fuseki did not either reorder the triple patterns while other triplestores put this triple pattern last in their execution plan. According to the statistics of the query execution plans, RDFox managed to finish Query 234 in 130 seconds (using the CLI) while Stardog finished in 647 seconds. For Query 235, the figures are 105 seconds and 1135 seconds for RDFox and Stardog respectively. Amazon Neptune failed to return the execution plans for the two queries within 30 minutes. As these two queries do not have any additional operator other than COUNT, the most expensive operations could be scanning indexes and joining results. According to the execution plan of Query 234, Stardog spent 44.5% of its execution time on scanning the indexes for the (?,?,?) pattern and 34.8% to `BindJoin` the results of the property path. For Query 235, the figures are 30.9% for indexes scanning and 37.3% for `HashJoin`. Although RDFox does not have similar statistics for each triple pattern, based on the total execution time mentioned earlier, RDFox seems to be very efficient in executing scanning and joining operations. This could be explained by the advantages of the in-memory solution over the persistent approach of the other triplestores. Stardog follows the pipelined query execution model[1] which has different implementations of joining algorithms that could be pipeline breakers (i.e., requiring to process all intermediate results before sending output) and therefore becoming the bottle-

[1] https://www.stardog.com/blog/how-to-read-stardog-query-plans/

neck in the pipeline. `HashJoin` is an example of a pipeline breaker in Stardog execution plan and should be avoided. As we rewrite the property path of the two queries with the sequence of three triple patterns explicitly, Stardog replaced its `BindJoin` and `HashJoin` in the execution plan with `MergeJoin` and `Sort` operations, which results in 2.6 and 4.2 times faster for the execution of Query 234 and 235. There was no difference as we executed the rewritten queries on the other triplestores. However, this observation suggests that using the usual joining syntax instead of the property path syntax to design the queries may help the query optimizer to find a better execution plan.

```
SELECT DISTINCT ?item ?itemLabel ?_PubMed_ID
WHERE
{
  ?item wdt:P31 wd:Q13442814 ;
        rdfs:label ?itemLabel .
  OPTIONAL { ?item wdt:P698 ?_PubMed_ID. }
  FILTER(CONTAINS(LCASE(?itemLabel), "zika"))
  FILTER (LANG(?itemLabel)="en")
}
```

Fig. 17. Wikidata - Query 343: Scholarly articles with "Zika" in the item label.

Query 343 (Figure 17) is another difficult query where most triplestores timed out. RDFox is the only successful triplestore that executed this query in 30 seconds. The query has three triple patterns, one of which is OPTIONAL. The query also has two FILTERS with string functions such as CONTAINS, LCASE, and LANG. The triplestores have a similar execution plan where filtering operations were performed after joining the results of the three triple patterns. Although filtering and those built-in string functions are not pipeline breakers, these operations may become time-consuming for large datasets. As can be seen in Figure 13, String Function appears in the top three expensive features for most triplestores. In this query, the filters and string functions were applied on 77.6M results, which took Stardog nearly 92% (46 minutes) of its execution time.

```
SELECT ?item ?itemLabel ?_PubMed_ID
WHERE
{
  ?item wdt:P31 wd:Q13442814 ;
        rdfs:label ?itemLabel .
  FILTER(CONTAINS(LCASE(?itemLabel), "suriname"))
  OPTIONAL { ?item wdt:P698 ?_PubMed_ID. }
}
```

Fig. 18. Wikidata - Query 350: Suriname citation corpora.

Another query with similar patterns is **Query 350** (Figure 18) which has 3 triple patterns and one FILTER with CONTAINS and LCASE. Only RDFox

can finish this query in 30 seconds while the others timed out. The filter and built-in string functions are also the most expensive operations which took 97% (160 minutes) of Stardog execution time. RDFox with all data available in RAM memory seems to suffer less from such string functions. Therefore, the usage of built-in functions should be considered carefully, especially for triple patterns that may return a large number of results.

```
SELECT DISTINCT ?pmid ?doi
WHERE
{
  ?item wdt:P698 ?pmid ;
              wdt:P356 ?doi .
}
```

Fig. 19. Wikidata - Query 233: PMID-DOI mappings.

Another difficult query that was analyzed is **Query 233** (Figure 19) which has only two triple patterns and a DISTINCT operator. Only RDFox and GraphDB managed to execute this query in 30 seconds and 217 seconds. Stardog, when executing the query profiler, returned the execution plan in 22.6 minutes. Although the query is simple, it returned about 22M results. According to the execution plan, Stardog spent 93.7% (21 minutes) of the execution time on post-processing the results (i.e., transforming the results in its internal data structure into the format that can be sent to the client). Thus, the number of results has a significant influence on the query execution time, especially for GraphDB which has result size as the feature most related to execution time as shown in Figure 13.

```
SELECT ?occupation (count(DISTINCT ?author) as ?count)
WHERE
{
    ?object wdt:P31 wd:Q13442814;
            wdt:P50 ?author .
    ?author wdt:P106 ?occupation .
}
GROUP BY ?occupation
ORDER BY DESC(?count)
```

Fig. 20. Wikidata - Query 225: Authors of scientific articles by occupation.

The next difficult query in this evaluation is **Query 225** (Figure 20) which has three triple patterns together with GROUP BY, ORDER BY modifiers, and COUNT DISTINCT operations. According to the query execution plan of Stardog and RDFox, the number of intermediate results of the triple patterns is 20M. The most expensive operation in Stardog execution plan is the `HashJoin`. It took 26 seconds (33% of the total execution time). Grouping and counting distinct results also accounted for 16% of the total time. As `HashJoin` and grouping are pipeline breakers, these operations can take a longer time, especially for such a large number of results. It seems that Stardog used `HashJoin` instead of other

joining algorithms in order to support grouping and counting distinct operations. As a result, Stardog had better performance on this query than RDFox and GraphDB. The execution time for Stardog, GraphDB, and RDFox are 68, 80, and 143 seconds respectively while Amazon Neptune and Jena Fuseki timed out.

```
select (count(?work) as ?count) ?subject where {
  ?work wdt:P31 wd:Q13442814;
        wdt:P921 ?subject .
}
group by ?subject
order by desc(?count)
limit 200
```

Fig. 21. Wikidata - Query 239: Most popular subjects of scientific articles.

Query 239 (Figure 21) has very similar operations as in Query 225. However, this query does not have the DISTINCT construct. As a result, Stardog did not `HashJoin` the results. The most expensive operation in Stardog plan was scanning for the `(?work, wdt:P31, wd:Q13442814)` pattern which took 16 seconds (41.4% of the total execution time) and returned 37.4M results. The pipeline breaker GROUP BY was the second most expensive operation accounting for 23% of the total time. Stardog, GraphDB, and RDFox were also the only three triplestores returning results. As in Query 225, Stardog and GraphDB also outperformed RDFox for this query. Amazon Neptune and Jena Fuseki also timed out. These two triplestores seem to suffer significantly from scanning and joining large number of results.

```
SELECT ?work ?title
WHERE
{
  ?work wdt:P31/wdt:P279* wd:Q838948;
        wdt:P1476 ?title.
  FILTER(REGEX(?title, "^\\w*(\\w{3})(\\W+\\w*\\1)+$", "i")
        && !REGEX(?title, "^(\\w+)(\\W+\\1)+$", "i")).
}
ORDER BY STR(?title)
```

Fig. 22. Wikidata - Query 327: Works of art where the name might be a rhyme.

Query 327 (Figure 22) is another difficult query with two triple patterns and a FILTER with two REGEX conditions. The first triple pattern also contains a sequence path with arbitrary length. Amazon Neptune, GraphDB, and RDFox were the three triplestores managing to execute this query before time out in 8, 10, and 105 seconds. As discussed earlier, the filtering operation is usually executed as early as possible in order to reduce the search space. Accordingly, Amazon Neptune, RDFox, and Stardog pushed the filter up to the first scanning operation, resulting in applying the filter on 41.41M results. As REGEX can

be an expensive string operation, this filtering operation is time-consuming. It took Stardog 11 minutes to perform the filtering operation. Amazon Neptune and GraphDB, on the other hand, performed joining results of the two triple patterns before applying the filter on only 1.1M results. Therefore, for this query, Amazon Neptune and GraphDB were superior to the other triplestores.

```
SELECT * WHERE {
  {
    SELECT ?cell_line ?cell_line_name WHERE {
      ?cell_line wdt:P31 wd:Q21014462;
                 rdfs:label ?cell_line_name.
          FILTER(LANG(?cell_line_name) = "en").
          Filter REGEX(STR(?cell_line_name),
                       "^[\\w\\-\\.]+\\.[A-z]+$")
    }
  }
  ?tld wdt:P31/wdt:P279* wd:Q14296;
       rdfs:label ?tld_name.
  FILTER(LANG(?tld_name) = "en").
  FILTER REGEX(STR(?cell_line_name),
    CONCAT(REPLACE(?tld_name, "\\.", "\\\\."), "$"), "i")
  BIND(URI(CONCAT("http://", ?cell_line_name)) as ?url)
}
```

Fig. 23. Wikidata - Query 192: Cell lines with names that could also be URLs (Internet of Cell Lines).

Query 192 (Figure 23) is one of the difficult queries that have one nested SELECT query inside another SELECT query, each of which has two FILTER operators. The filters also use string functions such as LANG, CONCAT, REPLACE, and REGEX. The outer SELECT also has a property path with arbitrary length. Amazon Neptune, Stardog, and GraphDB managed to execute the query in 5, 22, and 41 seconds while the others timed out. For such complicated queries, most triplestores kept the execution plan the same as the order in the original query. Specifically, the sub SELECT query was executed first before joining with the results of the outer query. However, Amazon Neptune had a different plan where the triple patterns and the first filter of the outer query were executed first and joined with the result of the subquery. This plan seems to be the optimal plan. According to Amazon Neptune profiler, there were only 130K intermediate results for the filter while it is 16.6M intermediate results in Stardog plan. Therefore, Amazon Neptune executed the query much faster than the others.

Another difficult query is **Query 48** (Figure 24) with nested SELECT query. The inner SELECT query has three MINUS operations, a FILTER NOT EXISTS, and a LIMIT of 1000 results. Due to the LIMIT, this query was executed by RDFox, GraphDB, and Neptune in only 3, 13, and 18 seconds and returned 1000 results. However, Stardog and Jena Fuseki timed out. According to Stardog profiler, the three MINUS operations required 81.7% (281 seconds) of the total execution time of Stardog, although they are applied on only 472K results.

```
SELECT ?item ?cnt WHERE {
{
  SELECT ?item (COUNT(?sitelink) AS ?cnt)
  WHERE {
  ?item wdt:P27|wdt:P205|wdt:P17 wd:Q16 .
  minus {?item wdt:P106 wd:Q488111 .}
  minus {?item wdt:P106 wd:Q3286043 .}
  minus {?item wdt:P106 wd:Q4610556 .}
  ?sitelink schema:about ?item .
  FILTER NOT EXISTS {
    ?article schema:about ?item .
    ?article schema:isPartOf
            <https://en.wikipedia.org/> .
  }
  } GROUP BY ?item ORDER BY DESC (?cnt) LIMIT 1000
}
} ORDER BY DESC (?cnt)
```

Fig. 24. Wikidata - Query 48: Canadian subjects with no English article in Wikipedia.

Therefore, Stardog may not have an optimal implementation of this MINUS operation.

```
SELECT ?author (COUNT(?publication) AS ?count)
WHERE
{
    ?item wdt:P2860 ?publication . #citations
    ?publication wdt:P50 ?author . #authors
    ?author wdt:P21 wd:Q6581072. #females
}
GROUP BY ?author
ORDER BY DESC(?count)
```

Fig. 25. Wikidata - Query 230: Most cited female authors.

Query 230 (Figure 25) is a simple query with only 3 triple patterns with GROUP BY and ORDER BY operations. Amazon Neptune (40s), GraphDB (56s), RDFox (66s), and Stardog (66s) managed to complete the execution while Jena Fuseki failed to finish this query. All successful triplestores followed the same query plan where they prioritized the execution of the most selective triple patterns.

Query 237 (Figure 26) is another difficult query with 5 triple patterns and 3 FILTERS, one of which has a triple pattern with a property path of arbitrary length inside. RDFox (0.6s), Stardog (21s), GraphDB (38s), and Jena Fuseki (138s) were the successful triplestores while Amazon Neptune timed out. All triplestores have the same strategy that moves the filters to an earlier phase in order to reduce the number of intermediate results, but the order and combination of triples and filters are quite different. According to Stardog profiler, Stardog spent 77.9% of the execution time on the property path inside the FILTER EXISTS and returned approximately 6.8M results. This operation is expected to be the most expensive for the other triplestores. RDFox seems to be

```
SELECT ?English ?language ?label WHERE {
  ?disease wdt:P699 "DOID:399";
             rdfs:label ?English;
             rdfs:label ?label .
  BIND(LANG(?label) as ?languageCode)
  ?wdLanguage wdt:P424 ?languageCode;
              rdfs:label ?language .
    FILTER EXISTS {?wdLanguage wdt:P31?/wdt:P279+ wd:Q17376908}
  FILTER (LANG(?English)="en")
  FILTER (LANG(?language)="en")
} ORDER BY ?language
```

Fig. 26. Wikidata - Query 237: Translations of the Disease Ontology term DOID:399 (Tuberculosis).

more optimal for scanning large results. Therefore, it outperformed the others in executing such property path syntax.

```
SELECT ?event ?eventLabel ?date
WHERE
{
  # find events
  ?event wdt:P31/wdt:P279* wd:Q1190554.
  # with a point in time or start date
  OPTIONAL { ?event wdt:P585 ?date. }
  OPTIONAL { ?event wdt:P580 ?date. }
  # but at least one of those
  FILTER(BOUND(?date)
    && DATATYPE(?date) = xsd:dateTime).
  BIND(NOW() - ?date AS ?distance).
  FILTER(0 <= ?distance && ?distance < 31).
  OPTIONAL {
    ?event rdfs:label ?eventLabel.
    FILTER(LANG(?eventLabel) = "en").
  }
}
LIMIT 10
```

Fig. 27. Wikidata - Query 13: Recent events.

The final difficult query considered in the analysis is **Query 13** (Figure 27) where only RDFox can successfully finish in 143 seconds, although the query has a LIMIT of 10 results. Fuseki and Stardog timed out while Amazon Neptune and GraphDB encountered runtime errors due to invalid date-time format. This query has 4 triple patterns with 3 OPTIONAL and 3 FILTER operations. RDFox combined the two Datetime filters into one while Stardog separated the expressions of those filters and made 4 different filtering operations. According to Stardog and RDFox profilers, these filtering operations are the most expensive in this query as they were applied on 14M results. As we removed those filters, all triplestores managed to return the results within 0.1 second due to the LIMIT operation. Therefore, RDFox shows its efficiency for date-time filtering operation compared with the other triplestores.

Queries with large variation in execution times Regarding the queries with large variation on execution times, the following interesting queries were analyzed.

```
select distinct ?item ?itemLabel ?image where {
  ?item wdt:P31/wdt:P279* wd:Q234460.

  ?author ?label 'Bram Stoker'.
  ?item wdt:P50 ?author.

  ?item rdfs:label ?itemLabel.
  filter contains(lcase(?itemLabel), 'dracula').

  optional {?item wdt:P18 ?image.}
} limit 50
```

Fig. 28. Wikidata - Query 286: Text by author containing case-insensitive title with optional cover image.

Query 286 (Figure 28) has 5 triple patterns with one pattern having arbitrary length properly path and one with OPTIONAL. The query also has a LIMIT of 50 results while all successful triplestores returned no result for this query. Except for Jena Fuseki and Amazon Neptune which timed out, the others executed the query in less than 0.1 second. Regarding the execution plan, all successful triplestores rewrote the property path pattern into two separate triple patterns and re-ordered all triple patterns in order to reduce the intermediate results for joining operations. However, Amazon Neptune and Jena Fuseki had very inefficient order of triple patterns which resulted in joining 13K, 50M, and 20M results sequentially compared to only 7 intermediate results from GraphDB execution plan. As we rewrite the query explicitly according to GraphDB plan, Jena Fuseki can execute the query in 0.03 seconds while Amazon Neptune still had the same execution plan and timed out.

```
SELECT ?item
WHERE
{
  ?item (wdt:P31/wdt:P279*) wd:Q191067.
  ?item wdt:P921 wd:Q2013.
}
LIMIT 100
```

Fig. 29. Wikidata - Query 084: Papers about Wikidata.

Query 84 (Figure 29) has only two triple patterns, one of which also has an arbitrary length property path. The query also has a LIMIT of 100 results. Similar to Query 286, Jena Fuseki and Amazon Neptune also timed out while the others finished in less than 1 second. Rewriting the property path and re-ordering the triple patterns are the strategy to execute this query. However, Jena Fuseki and Amazon Neptune also had very inefficient execution plan, which results

in joining 12M intermediate results while it is only 474 results from the plan of GraphDB. After rewriting the query manually according to GraphDB plan, Jena Fuseki executed in 0.06 second while Amazon Neptune had no change.

```
SELECT DISTINCT ?Person ?NobelPrize ?AcademyAward WHERE {
  ?NobelPrize wdt:P279?/wdt:P31? wd:Q7191 .
  ?AcademyAward wdt:P279?/wdt:P31? wd:Q19020 .
  ?Person wdt:P166? ?NobelPrize .
  ?Person wdt:P166? ?AcademyAward .
}
```

Fig. 30. Wikidata - Query 64: People that received both Academy Award and Nobel Prize.

Query 64 (Figure 30) has 4 triple patterns, two of which also have property path syntax. For this query, RDFox and GraphDB timed out while the other finished within 1 second. As this query has no complex construct, GraphDB, Fuseki, and Stardog had very similar order of triple patterns while RDFox and Amazon Neptune had the same execution plan. As we do not have the profiling reports from GraphDB and RDFox, there is not enough information to analyze the execution of those two triplestores for this query.

```
SELECT DISTINCT ?pathway ?pwpart ?variant ?disease WHERE {

   VALUES ?predictor {p:P3354 p:P3355 p:P3356 p:P3357 p:P3358 p:P3359}
   VALUES ?predictorQualifier {pq:P2175}
   VALUES ?wpID {"WP2828"}

   ?pathway wdt:P2410 ?wpID ;
          wdt:P527 ?pwpart .

   ?disease wdt:P279+ wd:Q504775 .

   ?variant wdt:P3329 ?civicID ;
             ?predictor ?node ;

             wdt:P3433 ?pwpart .

   {?node ?predictorStatement ?drug_label ;
              ?predictorQualifier ?disease .}
   UNION
   {
      ?node ?predictorStatement ?disease  .
   }
}
```

Fig. 31. Wikidata - Query 184: Get known variants reported in CIViC database (Q27612411) of genes reported in a Wikipathways pathway: Bladder cancer (Q30230812).

Query 184 (Figure 31) has 9 triple patterns and UNION operation. One of the triple patterns of this query has the form (?,?,?) inside the UNION which

may cause the triplestores to scan over the entire 16B triples in the dataset:

$$< A > JOIN(<?, ?, ? > UNION < B >)$$

As we look closer into the query, such an exhausted scan can be avoided by prioritizing the joining operation instead of the UNION operation as follow:

$$< A > JOIN <?, ?, ? > UNION < A > JOIN < B >$$

For this query, the successful triplestores managed to rearrange the operands of UNION while Jena Fuseki and RDFox kept the original execution plan. As a result, Jena Fuseki and RDFox timed out while the other finished in less than 1 second.

```
SELECT ?item ?reference ?referenceType
WHERE
{
  ?item wdt:P31 wd:Q13442814
  { ?item wdt:P921 wd:Q202864 }
  UNION
  { ?item wdt:P921 wd:Q8071861 }
  ?reference ?referenceType ?item

}
```

Fig. 32. Wikidata - Query 345: Scientific articles that have subject Zika virus or fever and that are used as a reference in another item.

Query 345 (Figure 32) is another query having pattern (?,?,?) together with UNION:

$$< C > JOIN(< A > UNION < B >)JOIN <?, ?, ? >$$

Obviously, the UNION operation should be executed before joining in order to avoid exhausted scans for the pattern (?,?,?). However, Jena Fuseki did not re-order the triple patterns and resulted in timeout. Amazon Neptune also timed out for this query while the other had a better execution plan and finished in less than 3 seconds.

Query 326 (Figure 33) is a long query with 18 triple patterns, one sub SELECT query, and 7 OPTIONAL constructs. For such a complicated query, all triplestores do not re-order any triple patterns, which seems to be the optimal execution plan. According to Amazon Neptune plan, the numbers of intermediate results in each step were not more than 105K results. However, Stardog reported 934.8M results in one of the steps in its execution plan and resulted in a memory error eventually. For this query, only Jena Fuseki and Stardog timed out. The others can finish in 2 seconds.

Query 176 (Figure 34) is also another complex query with 3 OPTIONAL constructs, two inside the other, and a MINUS operation. Similar to Query 326, all triplestores keep the original execution order. RDFox (6s), Neptune (14s), and

```
SELECT
  ?item
  (SAMPLE (?titleL) AS ?title)
  (group_concat(distinct ?creatorL ; separator = ", ") as ?creator)
  (group_concat(distinct ?genreL ; separator = ", ") as ?genre)
  (group_concat(distinct ?placeL ; separator = ", ") as ?place)
  (group_concat(distinct ?arr ; separator = ", ") as ?arrondissement)
  (SAMPLE (?img) AS ?image)
  (SAMPLE (?coord) AS ?coordinates) {

    {
      SELECT DISTINCT ?item { {
        ?item wdt:P136 wd:Q557141 ;     # genre: public art
              wdt:P131 wd:Q90           # located in: Paris
      } UNION { # or
        ?item wdt:P136 wd:Q557141 ;     # genre: public art
              wdt:P131/wdt:P131* wd:Q90 # located in an arrondissement of Paris
      } }
    }

    OPTIONAL { ?item rdfs:label ?titleL filter (lang(?titleL) = "fr") }
    OPTIONAL { ?item wdt:P170 [rdfs:label ?creatorL] filter (lang(?creatorL) = "fr") }
    OPTIONAL {
      {
        ?item wdt:P136 ?g filter (STR(?g) != 'http://www.wikidata.org/entity/Q557141')
      } UNION {
        ?item wdt:P31 ?g .
      }
      ?g rdfs:label ?genreL filter (lang(?genreL) = "fr") .
    }
    OPTIONAL {
      ?item wdt:P276 [rdfs:label ?placeL] filter (lang(?placeL) = "fr") .
    }
    OPTIONAL {
      ?item wdt:P131 [wdt:P131 wd:Q90 ; rdfs:label ?arrL] filter (lang(?arrL) = "fr").
      BIND(REPLACE(?arrL, '^([0-9]+).*$', "$1", "si") AS ?arr)
    }
    OPTIONAL { ?item wdt:P18 ?img }
    OPTIONAL { ?item wdt:P625 ?coord }
} GROUP BY ?item
```

Fig. 33. Wikidata - Query 326: Public art in Paris.

```
SELECT ?compound ?species ?source ?doi ?wpid WHERE {
  ?compound wdt:P31 wd:Q11173.
  MINUS { ?compound wdt:P31 wd:Q8054. }
  ?compound p:P703 ?statement.
  ?statement rdf:type wikibase:BestRank.
  ?statement ps:P703 ?species.
  OPTIONAL {
    ?statement (prov:wasDerivedFrom/pr:P248) ?source.
    OPTIONAL { ?source wdt:P2410 ?wpid. }
    OPTIONAL { ?source wdt:P356 ?doi. }
  }
}
ORDER BY ASC(?compound)
```

Fig. 34. Wikidata - Query 176: Metabolites and the species where they are found in.

GraphDB (25s) are the successful triplestores for this query while Jena Fuseki and Stardog timed out. Similarly, Stardog also reported a very large number of

intermediate results (486M results) while RDFox and Amazon Neptune had very consistent reported numbers from their profilers.

Queries where the numbers of results are not consistent The third category of queries investigated is the one where the numbers of results are not consistent. Accordingly, the following queries are considered.

```
SELECT ?item ?pic ?linkTo
WHERE
{
  OPTIONAL { ?item wdt:P171 ?linkTo }
  OPTIONAL { ?item wdt:P18 ?pic }
}
```

Fig. 35. Wikidata - Query 178: Asterophryinae parent taxon reverse graph.

Query 178 (Figure 35) has two OPTIONAL patterns. GraphDB reported 1.4M results while QLever, as discussed earlier, has a limitation of only 10K results. The other triplestores reported the same number with 3.4M results. As GraphDB does not have profiling information from the execution plan, there is not enough information to analyze the difference in the results for this triplestore.

Query 233 ((Figure 19)) is also a very simple query with only 2 triple patterns. For this query, RDFox and Stardog reported the same number of results (22.9M) while GraphDB had only 1.7M. Again, it is not possible to analyze this inconsistency due to not enough information from GraphDB.

```
SELECT ?property ?propertyDescription ?count WHERE {
  {
    select ?propertyclaim (COUNT(*) AS ?count) where {
      ?item wdt:P106 wd:Q1028181 .
      ?item wdt:P31 wd:Q5 .
      ?item ?propertyclaim [] .
    } group by ?propertyclaim
  }
  ?property wikibase:propertyType wikibase:ExternalId .
  ?property wdt:P31 wd:Q19595382 .
  ?property wikibase:claim ?propertyclaim .

} ORDER BY DESC (?count)
LIMIT 100
```

Fig. 36. Wikidata - Query 319: Authority control properties usage for painters.

Query 319 (Figure 36) is another query where GraphDB reported differently from the other triplestores. This query has a sub-SELECT query which returned 656 intermediate results according to the execution plans of RDFox and Amazon Neptune. As the query has a LIMIT of 100 results, all other successful triplestores returned 100 results while GraphDB reported no result.

```
SELECT ?continent ?river
WHERE
{
  {
    SELECT ?continent (MAX(?length) AS ?length)
    WHERE
    {
      ?river wdt:P31/wdt:P279* wd:Q355304;
             wdt:P2043 ?length;
             wdt:P30 ?continent.
    }
    GROUP BY ?continent
  }
  ?river wdt:P31/wdt:P279* wd:Q355304;
         wdt:P2043 ?length;
         wdt:P30 ?continent.
}
ORDER BY ?continent
```

Fig. 37. Wikidata - Query 109: Longest river of each continent.

Query 109 (Figure 37) also has a nested SELECT query. Again, GraphDB returned 0 result while the others had 11 results.

```
SELECT ?animal ?scientific_names ?common_names ?status where
{
   {
      SELECT DISTINCT ?animal
      (GROUP_CONCAT(distinct ?scientific_name; separator=", ") as ?scientific_names)
      (GROUP_CONCAT(distinct ?common_name; separator=", ") as ?common_names)
      WHERE
      {

        ?animal wdt:P141 ?status;
                wdt:P225 ?scientific_name;
                wdt:P1843 ?common_name.
        filter( ?status
           IN (
               wd:Q11394,  #Endangered
               wd:Q219127, #critcally endangered
               wd:Q278113  #vulnurable
              )
          ).

        FILTER(LANGMATCHES(LANG(?common_name), "en"))
      }
      group by ?animal
   }.
   ?animal wdt:P141 ?status.
}
```

Fig. 38. Wikidata - Query 195: Threatened Species of Animals as per IUCN Classification.

Query 195 (Figure 38) also has a nested SELECT query. Except for Stardog which returned only 139K results, the other successfully triplestores reported 9689 results. When investigating the execution plan of Stardog, we noticed that the triplestore seemed to have an issue with the sub-SELECT query and was

missing the profiling report for this sub-query. Instead, Stardog estimated this sub-query has 20K results while the actual number is 9.7K results.

```
SELECT DISTINCT ?city ?population ?country ?loc WHERE {
  {
    SELECT (MAX(?population) AS ?population) ?country WHERE {
      ?city wdt:P31/wdt:P279* wd:Q515 .
      ?city wdt:P1082 ?population .
      ?city wdt:P17 ?country .
    }
    GROUP BY ?country
    ORDER BY DESC(?population)
  }
  ?city wdt:P31/wdt:P279* wd:Q515 .
  ?city wdt:P1082 ?population .
  ?city wdt:P17 ?country .
  ?city wdt:P625 ?loc .
}
```

Fig. 39. Wikidata - Query 82: Largest cities per country.

Query 82 (Figure 39) is another query with a nested SELECT query. Similar to Query 193, Stardog also reported a very different number of results from the others.

Query 327 (Figure 22) was already discussed above as a difficult query. This query has a FILTER with two REGEX functions with very complex string patterns which can be its most difficult operation. Accordingly, Amazon Neptune returned an empty response while GraphDB and RDFox returned 1242 results.

```
SELECT ?item ?itemLabel ?coord
WHERE
{
  ?item wdt:P31/wdt:P279* wd:Q484170;
        wdt:P17 wd:Q142;
        rdfs:label ?itemLabel;
        wdt:P625 ?coord;
  FILTER (lang(?itemLabel) = "fr").
  FILTER regex (?itemLabel, "ac$").
  FILTER not exists { ?item wdt:P131 wd:Q33788 }
}
```

Fig. 40. Wikidata - Query 133: French communes with names ending in ac.

Query 133 (Figure 40) is another query with FILTER having string REGEX. Again, Amazon Neptune returned nothing while the other successfully triple-stores reported 1132 results.

Query 93 (Figure 41) also has a FILTER with string REGEX. Similarly, Amazon Neptune filtered out all of the results while the other successfully triple-stores reported 17.8M results.

Query 284 (Figure 42) also has a date-time FILTER. For this query, Stardog reported 2765 results while the others returned 2759 results. As noted from

```
SELECT DISTINCT ?settlement ?name ?coor
WHERE
{
   ?subclass_settlement wdt:P279+ wd:Q486972 .
   ?settlement wdt:P31 ?subclass_settlement ;
               wdt:P625 ?coor ;
                rdfs:label ?name .
   FILTER regex(?name, "Antwerp", "i")
}
```

Fig. 41. Wikidata - Query 93: Where in the world is Antwerp.

```
SELECT ?item ?label ?coord ?place
WHERE
{
  VALUES ?type {wd:Q571 wd:Q7725634}
  ?item wdt:P31 ?type .
  ?item wdt:P577 ?date FILTER (?date < "1830-01-01T00:00:00Z"^^xsd:dateTime) .
  ?item rdfs:label ?label filter (lang(?label) = "en")

  OPTIONAL {
    ?item (wdt:P291|wdt:P840) ?place .
    ?place wdt:P625 ?coord
  }
}
```

Fig. 42. Wikidata - Query 284: Books or literary works published before 1830 with place of publication or narrative location coordinates.

Query 298, Wikidata may have inconsistent formats for date-time data. Thus, filtering such data may result in different numbers of results from the triple-stores.