Analysis of the Effect of Query Shapes on Performance over LDF Interfaces

Gabriela Montoya, Ilkcan Keles, and Katja Hose

Aalborg University, Aalborg, Denmark
{gmontoya,ilkcan,khose}@cs.aau.dk

Abstract. The adoption of Semantic Web technologies, and in particular the Open Data initiative, has contributed to the steady growth of the number of datasets and triples accessible on the Web. Most commonly, queries over RDF data are evaluated over SPARQL endpoints. Recently, however, alternatives such as TPF have been proposed with the goal of shifting query processing load from the server running the SPARQL endpoint towards the client that issued the query. Although these interfaces have been evaluated against standard benchmarks and testbeds that showed their benefits over previous work in general, an evaluation of the effects of the query shapes on query performance of the different available interfaces has never been done. In this paper, we present the results of our in-depth evaluation of query shapes impact on the performance of existing LDF interfaces. Using representative and diverse query loads that are designed to include relevant query shapes and are based on the query log of a public SPARQL endpoint, we stress test the different interfaces and identify their strengths and weaknesses.

1 Introduction

With the adoption of the Open Data initiative by many institutions and companies, the amount of data offered on the Web in RDF is growing on a daily basis. While some of these datasets, such as DBpedia [10], offer information extracted from unstructured sources, such as Wikipedia, other datasets focus on factual information from a specific domain, such as life science, geography, government, publications, etc.

The simplest way to make such datasets available to others is publishing them on the Web as downloadable data dumps, typically encoding information in RDF formats such as N-triples or Turtle. The dump can then be downloaded through HTTP and the user can process the data according to his/her needs. Whereas this is very low effort for the data provider, the problem is that the user cannot simply query the information he/she is looking for directly at the data provider but instead has to download the entire dataset and process the query locally.

On the other hand, a data provider can choose to run a SPARQL endpoint (server) to provide access to the data. In this way, a user (client) can send a SPARQL query to the endpoint, which processes it and returns the answer to

the query. The advantage for the user in this setting is that it requires very little effort from the user. The price, however, is a relatively high load at the server running the endpoint as it has to process the entire query. If too many clients send queries concurrently or if the server is processing complex queries, query response time increases and/or the endpoint might even become unavailable for some time.

To address this bottleneck, the Triple Pattern Fragments (TPF) interface [18] was proposed. To achieve the goal of better sharing the query load between clients and servers, the server is stripped from any higher-level query functionality and is only able to process single triple pattern requests. Any other query processing tasks, in particular processing joins, filters, grouping, are exclusively handled by the client. In doing so, the TPF interface increases availability and throughput at the server side. brTPF [8] then extends TPF by allowing the client to not only send a single triple pattern to the server but also include a sequence of bindings for the variables in the triple pattern. This makes it possible to send bindings obtained from intermediate results of a SPARQL query at the client and to reduce the number of HTTP requests that need to be sent to the server.

The trade-offs of using different LDF interfaces have been explored within the Linked Data Fragment (LDF) framework [18]. LDF interfaces are interfaces that provide access to RDF data, such as data dumps, SPARQL endpoints, TPF and brTPF. Even if a formal framework for comparing LDF interfaces in terms of their expressiveness and complexity is proposed by Hartig et al. [9], there are only very few empirical evaluations that compare LDF interfaces, and they are not extensive enough. Although the literature [8, 9, 13, 18] provides some analysis of the general behavior of available interfaces for RDF data, in particular SPARQL endpoints, TPF, and brTPF, none of them provides a detailed analysis that tries to find out which interface performs best for a specific query shape and what the advantages of an interface over another interface are for processing a specific query shape.

In this paper, we therefore provide an extensive empirical evaluation of available LDF interfaces (SPARQL endpoint, TPF, and brTPF) using a real query load sent to the DBpedia SPARQL endpoint [11]. In contrast to existing analyses, we decided to use a real query log instead of a synthetic benchmark to reflect real user behavior. In summary, this paper makes the following contributions: (i) A survey of coverage and shortcomings of existing evaluations of LDF interfaces, (ii) Definition of representative and diverse query loads that facilitates an in-depth analysis of the effect of query shapes on LDF interfaces, and (iii) Extensive evaluation of LDF interfaces analyzing how much the query shapes influence the performance of available LDF interfaces.

This paper is organized as follows. Section 2 summarizes existing evaluations of LDF interfaces and highlights their shortcomings. Section 3 defines the experimental setup, including the characterization of representative queries from logs of public endpoints. Section 4 presents our experimental results and an extensive discussion, and finally Section 5 concludes the paper with a summarization of our most important findings.

2 Existing Interfaces and Evaluations

In this paper, we focus on the most popular interfaces proposed for querying RDF datasets: SPARQL endpoints, Triple Pattern Fragments (TPF) [18], and bindings-restricted Triple Pattern Fragments (brTPF) [8]. SPARQL endpoints are most convenient for clients as they can submit complete SPARQL queries and simply receive the answer to the query – the complete query load is on the server side (SPARQL endpoint). Furthermore, SPARQL endpoints support the complete SPARQL specification [5]. On the other hand, allowing clients to issue complex SPARQL queries might require considerable resources in terms of CPU and main memory at the server. Processing multiple such queries concurrently might result in considerable delays or in the worst case non-availability of the server. A survey of public SPARQL endpoints [2] shows that only 32.2% of endpoints are capable of providing 99% to 100% availability during the 27-month long monitoring.

The TPF interface [18] was proposed to address the availability issue of SPARQL endpoints by better sharing the query processing load between server and clients. A TPF server is only capable of handling triple pattern requests. In other words, it receives a triple pattern from a client and returns the triples of the hosted knowledge graph matching the input triple pattern. The client then takes care of all other query processing tasks, such as joining, filtering, grouping, query optimization and decomposition, and sending triple pattern requests to the servers. The TPF interface has been evaluated against SPARQL endpoints based on Jena Fuseki and Virtuoso [6] using an instance of the Berlin SPARQL Benchmark (BSBM) dataset [3] that contains 100 millions triples [18]. The experiments show that the CPU load on the server is lower and the CPU load on the client is higher for TPF interfaces compared to SPARQL endpoints. Moreover, the network load between the server and the client increases since the client has to issue several HTTP requests to process a single SPARQL query. Verborgh et al. [18] also provide an experiment to assess the performance of TPF on a real-world knowledge graph by executing different queries obtained from the DBpedia SPARQL benchmark (DBSB) [14] on three knowledge bases containing 14 million triples, 52 million triples, and 377 million triples, respectively. In this last experiment, TPF was the only interface assessed and no results regarding the execution of these queries against SPARQL endpoints were provided. This experiment shows that the query processing time of TPF has a high variance between queries with different keywords. Queries with keywords like UNION, FILTER, and OPTIONAL are quite expensive using TPF since the TPF client implementation used does not provide a good query plan for such queries. Moreover, the experiments presented do not pay any particular attention to the shape of the issued query. For this reason, the effect of the shape of the issued query on TPF remains unknown.

brTPF [8] extends TPF by adding a sequence of bindings to the triple pattern requests to reduce the overall number of HTTP requests necessary to answer a query. brTPF was evaluated against TPF using the WatDiv dataset and queries generated by the associated stress testing tools [1]. Specifically, a syn-

thetic knowledge graph with 10 million triples (published also on the project website) is used for evaluation. A total of 145 BGP queries and a total of 12,400 queries are used for single-client experiments and multi-client experiments, respectively. Up to 64 clients are used for multi-client experiments. The experimental evaluation demonstrates that brTPF has a better query throughput and less network overhead compared to TPF in both settings.

Aside from proposing WatDiv dataset and stress testing tools, Aluc et al. [1] also present an experimental evaluation of SPARQL endpoints including Virtuoso and 4store. The experimental evaluation shows that the query processing performance of the endpoints differs a lot with respect to the queries. In order to see the effect of query characteristics on the performance of the endpoints, they group queries with respect to their selectivity and their structure. They only consider linear and star/snowflake structures. Selectivity-based characterization of queries requires a dynamic analysis and it is quite expensive. Moreover, a systematical evaluation of the effect of shapes on the query processing performance is missing in the literature. For these reasons, we choose to consider query shapes that can be determined via static analysis in this work.

Existing evaluations between TPF and brTPF are limited to WatDiv and do not analyze the influence of particular query shapes. Instead, only average times over sets of queries are reported. However, a solution that works well on average does not necessarily work best on all types of queries.

3 Evaluation Setup

In this section, we present our experimental setup covering datasets and queries, query loads, interfaces, hardware setup, and evaluation metrics.

Dataset and Queries. For our study, we use the USEWOD 2016 research dataset [11] [1] that contains query logs from the public DBpedia interfaces: SPARQL endpoint and TPF server. We use SPARQL queries sent to the DBpedia SPARQL endpoint. The USEWOD dataset covers the query logs of 20 randomly selected days between July, 2015 and November, 2015 (43 GBs) containing nearly 10 million unique select queries. We do not use existing benchmarks that generate synthetic queries such as [1, 3] since the generated queries do not sufficiently reflect the characteristics of queries executed by actual users of the SPARQL endpoints. This can be observed for instance in a recent fine-grained evaluation done by Saleem et al. [16]. Moreover, we also do not use existing benchmarks based on user query logs such as [14,15] since they focus on generating query loads that cover all queries and SPARQL keywords present in the query log, while our focus is studying the different query shapes found in the query logs. Moreover, some of the studied systems (e.g., current implementation of TPF) do not support all the SPARQL keywords present in the query loads

[1] As indicated in the USEWOD 2016 dataset, we use the DBpedia 3.9 dataset. It is available at `http://downloads.dbpedia.org/3.9/en/`. We loaded DBpedia 3.9 dataset to all of the endpoints and removed the triples that created problems in any of the endpoints we use. At the end, our dataset contains 351,590,668 triples.

generated by these benchmarks. Therefore, we target BGPs, OPTIONALs and FILTERs in order to compare all three interfaces (SPARQL, TPF, and brTPF).

Table 1: Description of the Query Shapes

Query Shape	**Description**	**Example**
EDGE	single triple pattern	?x p ?y
CHAIN	triple patterns are chained together with object-subject joins	?x p1 **?y** . **?y** p2 ?z
CYCLE	as CHAIN but with an additional join between the first and last triple pattern	**?x** p1 *?y* . *?y* p2 **?x**
STAR	all triple patterns share a join variable either as subject or as object	**?x** p1 ?y . **?x** p2 ?z . **?x** p3 ?w
TREE	no EDGE, CHAIN or STAR, without cycles	**?x** p1 ?y . **?x** p2 **?z** . **?z** p3 ?w . **?z** p4 ?v
FLOWER	CHAINs, TREEs and PETALs with a common join variable. A petal includes multiple disjoined CHAINs between the same pair of variables	**?x** p1 **?y** . **?y** p2 **?z** . **?x** p3 **?w** . **?w** p4 **?z** . **?x** p5 **?v** . **?v** p6 ?u

A recent study [4] provides structural and shape analysis for all the queries in the USEWOD 2016 research dataset [11]. According to this analysis, six shapes are the most common shapes in query logs: EDGE, CHAIN, CYCLE, STAR, TREE, and FLOWER. Query shapes are described in Table 1. For our study, we only consider unique select queries of these types that do not have any syntactical errors according to the SPARQL specification. Moreover, we also consider structural characteristics such as the use of operators JOIN, OPTIONAL, and FILTER, use of variables as predicates, whether the used filters are safe and simple and whether the used OPTIONAL clauses are well-designed and tractable in line with Bonifati et al. [4]. A safe filter only includes variables used in its graph pattern, while simple filters include only one variable or correspond to $X = Y$ with X, Y being variables. Well designed OPTIONAL clauses only join graph patterns using variables that are always bound (in the left operand), while tractable OPTIONAL clauses include at most one join variable between their operands. In this study, we focus on the queries that do not have any variables as predicates, and that contains only safe simple FILTER clauses and well-designed and tractable OPTIONAL clauses. In other words, we focus on queries with tractable graph patterns and there are $4,337,181$ such queries contained in the USEWOD 2016 dataset.

Table 2: Shapes of Queries

Query Shape	**Total**	**Relevant**
CHAIN	832,873	171,244
CYCLE	73	31
EDGE	3,189,874	1,275,313
FLOWER	3,209	5
STAR	274,678	8,657
TREE	36,474	358

For these $4,337,181$ queries, we examined their shapes and the findings are listed in Table 2. The total number corresponds to the number of queries with

this shape. We exclude queries with empty answers and queries that are not supported by existing implementations to allow for a more interesting performance study of existing LDF interface implementations using these queries. Existing implementations of TPF and brTPF do not support features such as VALUES, subqueries, REGEX expressions with three arguments, aggregations, functions on RDF terms (e.g., isLiteral), and functions on strings (e.g., UCASE). Moreover, some predicates such as *bif:contains* are only supported by Virtuoso. We refer to the remaining queries as *relevant queries*. The number of relevant queries for our study is shown in the rightmost column of Table 2. Some query shapes had considerably fewer queries that have answers and are supported by existing implementations. An example is queries with FLOWER shape, where 3,108 out of 3,209 queries include the predicate bif:contains that is only supported by Virtuoso. After we determine the set of relevant queries, we remove modifiers DISTINCT, ORDER BY, LIMIT and OFFSET from the queries with these modifiers. This is needed to focus on the evaluation of the graph patterns and to have a fair comparison between different interfaces. Since both TPF and brTPF are not optimized for these modifiers and use post-processing on the client-side to process queries with modifiers, we think it would be unfair to compare brTPF and TPF with SPARQL endpoints using such queries.

Query Loads. After determining the set of relevant queries, we continue with creating query loads for single-client and multiple-clients experiments. In line with [8], we include experimental evaluation with multiple clients to assess how the number of clients concurrently accessing the interface affects the performance of the interface. Moreover, this set of experiments makes it possible to evaluate the interfaces under high load.

For the single-client experiments, we consider query loads of at most 100 random queries for each shape[2]. In total, we have 436 queries distributed into 6 query loads, 1 for each shape: CHAIN, CYCLE, EDGE, FLOWER, TREE, and STAR.

For multiple-client experiments, we consider up to 64 clients as done by Hartig and Aranda in [8]. Experimental results, e.g. [8, 18], demonstrate that the advantages of TPF and brTPF become visible even with 16 clients. For these experiments, instead of creating a separate query load for each shape, we create two query loads (Equal and Proportional) for each client that combine queries with different shapes. Both query loads are constructed by randomly drawing queries from different query shapes. The queries are drawn with respect to the uniform distribution for the Equal query load and with respect to the frequency distribution for the Proportional query load. In the uniform distribution, every shape has the same probability to be drawn, while in the frequency distribution the shape probability is proportional to the frequency of that shape in the relevant queries.

We want to make sure that the intersection of the query loads for different number of clients is empty since we do not want interfaces take advantage of

[2] For shapes with less than 100 queries, all the available queries are included in the query load.

caching during the experiments. For this reason, we considered only shapes with at least 6,400 queries as we aimed to have 64 query loads with 100 queries to have experiments with 64 clients.

Interfaces and Implementations. In this paper, we focus on three interfaces for accessing RDF datasets: TPF [18], brTPF [8], and SPARQL endpoints. All the interfaces require a server and a client implementation. We use popular triple stores that rely on different data representations (binary RDF, RDMS, native graph database) as concrete server implementations for the SPARQL endpoint interface: Fuseki (HDT)[3], Virtuoso [6] 7.2.5.3229-pthreads, and Blazegraph [17] 2.1.5 Release Candidate version. Fuseki is used with default configuration[4]. The file sizes for different triple stores are: 5.3G (HDT), 1.7G (HDT Index File), 21G (Virtuoso DB File), 34G (Blazegraph Journal File).

Because the brTPF server is only available in Java and to exclude the possibility that our results could be attributed to different implementations, we integrated brTPF into the latest available (and bug-free) TPF server (Java)[5]. We use that as the server implementation for TPF and brTPF. This implementation is based on the use of HDT files [7, 12].

We use the nodeJS client from [8] as brTPF client, and the nodeJS client from [18] as TPF client[6]. We use a nodeJS client from [13] for SPARQL endpoints since we want to have a fair comparison between interfaces by relying on the same client infrastructure.

Table 3: Machine configurations

Machine	Cores	RAM	OS	Network Speed
Small	8 x 2294.250 MHz	64GB	Ubuntu 16.04.1 LTS	up to 10,000Mb/s
Big	64 x 2294.176 MHz	516GB	Ubuntu 14.04.6 LTS	up to 1,000Mb/s

Hardware Configuration. We use two machines with different configurations that are described in Table 3. For the experiments with a single client, the servers are deployed using docker[7] containers in the big server and configured so that they will use up to 8GB of RAM and three cores, while the clients are run in the small machine and each client is set to use one core and up to 3GB of RAM. For the experiments with multiple clients, the servers are deployed using docker containers in the small machine and configured to use up to 21GB of RAM[8] and three cores, while up to 64 clients are run in the big machine and each client is set to use one core and up to 3GB of RAM. Given the machine configurations,

[3] part of hdt-java, available at `https://github.com/rdfhdt/hdt-java` latest development version, February 17th, 2019

[4] The configuration files for Virtuoso and Blazegraph are available in our project website: `http://qweb.cs.aau.dk/evaluation`

[5] The code of latest TPF server in Java is available at `https://github.com/LinkedDataFragments/Server.java`.The code of the extended server is available at our project website

[6] To the best of our knowledge, this is the only client publicly available

[7] `https://www.docker.com/`

[8] Virtuoso was not able to handle 64 clients with less RAM and lower bounds set by the configuration file failed to have any impact on restricting the RAM usage.

cf. Table 3, there are enough resources available for all the clients and servers configured as described above. We use a low-latency network (<1ms).

Evaluation Metrics. In the experimental evaluation we refer to SPARQL endpoints, TPF server and brTPF server as *server*. To evaluate the different approaches, we use the following measures:

- Execution Time (ET): the time elapsed between issuing the query and getting the query results (with a timeout of five minutes),
- Number of HTTP requests (NH): the number of HTTP requests sent to the the server,
- Server Load (SL): the CPU percentage used by the TPF server, brTPF server, and SPARQL endpoints during query processing. The CPU percentage is measured using the statistics docker provides regarding docker containers via *docker stats* command with a frequency of 30 seconds. It might go up to 300% (as each server has 3 cores) and the reported load is the average CPU percentage throughout processing all queries within a query load,
- Number of Retrieved kBs (NRKB): the number of kilobytes transferred from the server to the client during query execution,
- Number of Sent kBs (NSKB): the number of kilobytes transferred from the client to the server during query execution,
- Number of Timed out Queries (NTQ): the number of queries that do not complete their execution within five minutes.

4 Evaluation Results

In this section, we present the results of the single-client and multiple-client experiments[9]. We performed experiments using: Blazegraph endpoint (e_B), Fuseki endpoint (e_F), and Virtuoso endpoint (e_V), brTPF server (brtpf), and TPF server (tpf).

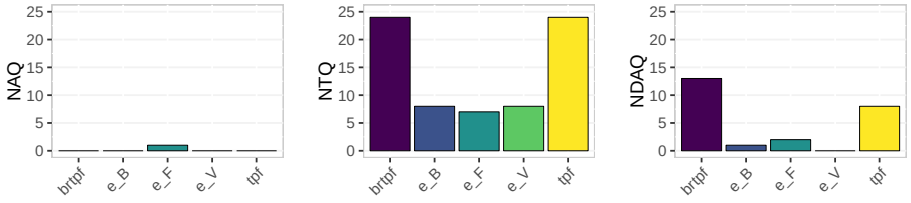
4.1 Preliminary Experiments

Surprisingly, we encountered several problems in our preliminary experiments when executing the generated query loads: queries that aborted with errors, queries with inconsistent results across systems, and timed-out queries. Figure 1 shows an overview of such queries. Including aborted and timed-out queries in our results can negatively impact the performance, data transfer, and server usage metrics of the systems that completed the execution of the queries without any problems[10]. Hence, we present the metrics obtained by excluding the problematic queries.

We assessed the reasons why we obtain different results across systems. Problems include incorrect evaluation of queries with OPTIONALs involving BGPs

[9] The complete evaluation results are available at our project website: `http://qweb.cs.aau.dk/evaluation`

[10] Figures showing the effect of including queries with consistent answers are available in our project website.

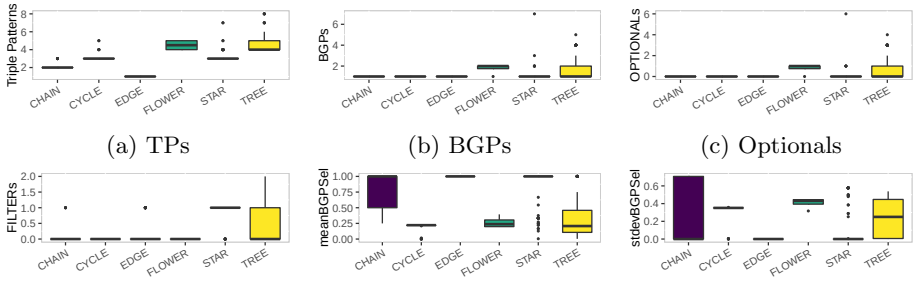


(a) Errors (b) Timeouts (c) Answers

Fig. 1: Number of queries aborted with error (NAQ), timed out (NTQ), or with different number of answers (NDAQ)

with more than one triple pattern (TPF), incorrect evaluation of queries with nested OPTIONALs (TPF and brTPF), fragment pages missing control elements that prevent accessing the second page of a fragment (the brTPF client implementation), evaluation of property paths that do not follow the standard set semantics (Virtuoso).

It is important to note that queries with the lowest number of triple patterns, e.g., EDGE-shaped queries, have no queries with different answers across the systems, while queries with higher numbers of triple patterns, e.g., TREE-shaped or STAR-shaped queries, amount to 14 of the total 16 queries (across all query loads) with different answers across systems. Therefore, studying queries with a higher numbers of triple patterns and diverse shapes allows for identifying some limitations of existing implementations of the different interfaces.



(a) TPs (b) BGPs (c) Optionals

(d) Filters (e) Mean BGP Sel (f) Stdev BGP Sel

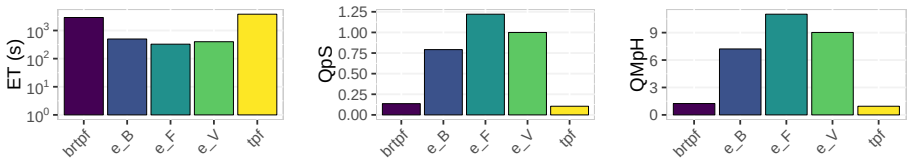
Fig. 2: Number of Triple Patterns (TPs), Basic Graph Patterns (BGPs), Optionals, and Filters, Mean and Standard Deviation (Stdev) BGP-restricted Triple Pattern Selectivity (Sel)

After removing the 16 queries mentioned above, Figure 2 shows some structural and data-driven characteristics of the queries in each query load. The query loads with higher diversity for these characteristics are CHAIN and TREE. TREE includes the higher number of OPTIONAL clauses and consequently the higher number of BGPs. CHAIN and TREE include queries with more diverse BGP-restricted triple pattern selectivity. For a BGP $bgp = \{tp_1, tp_2, ..., tp_n\}$, the BGP-restricted triple pattern selectivity of bgp for tp_i indicates the pro-

portion of solutions for tp_i that are compatible with solutions for bgp. A high BGP-restricted triple pattern selectivity value indicates that most intermediate results contribute to the solution of bgp, while a low value indicates that there are many intermediate results that do not contribute to the solution of bgp.

4.2 Single-Client Experiments

Performance. Existing benchmarks for SPARQL endpoints [3,14,15] use metrics such as queries per second (QpS) and query mixes per hours (QMpH) for performance evaluation. Existing TPF and brTPF studies [8,18] employ queries per hour or throughput (QpH) metrics. All these metrics provide information with a very coarse granularity, i.e., just one number to describe how a system performed a query load. Figure 3 shows some of these metrics for processing all single-client query loads. According to these results, the system that perform the best is the Fuseki endpoint (e_F). Moreover, the performances of brTPF and TPF are very similar.



(a) ET (b) QpS (c) QMpH

Fig. 3: Total ET, Queries per Second (QpS), and Query Mixes per Hour (QMpH) for different interfaces

However, having a single number that summarizes the performance of the systems across the query loads may hide some interesting facts. In particular, we have no information about how each system performs queries with specific shapes. Figure 4 therefore shows the query execution time (ET) represented with a boxplot for each query shape and system. EDGE- and CHAIN-shaped queries are performed faster by Blazegraph, Fuseki, and brTPF. Even if EDGE-shaped queries have no binding-restricted requests, brTPF exhibits a slightly better performance than TPF. This shows that the brTPF client also includes further optimizations besides the binding-restricted requests, e.g., variables in the triple pattern are replaced by *?s*, *?p*, *?o* to reduce the data transfer. On the other hand, Virtuoso's performance for EDGE- and CHAIN-shaped queries is very low. This is evidenced by an execution time that is considerably higher than the ones of other endpoints for half of the queries. For CYCLE-shaped queries, endpoints perform better than TPF and brTPF. For STAR-shaped queries, Virtuoso performs better than others. However, it is also worth noting that TPF and Blazegraph have quite a high number of outliers for this shape. For TREE-shaped queries, Fuseki performs the best. TPF and brTPF have a quite large variance between execution times for this shape. For this reason, we conclude that one should not use TPF and brTPF for processing TREE-shaped queries.

For FLOWER-shaped queries, Fuseki provides the most efficient query processing. If we want to execute queries with characteristics as diverse as the ones in query load TREE (see Figure 2), one would not choose the Blazegraph endpoint even if it has the best overall performance according to Figure 3. Figure 4 also shows that the shape of the issued queries affects the query processing performance for each system.

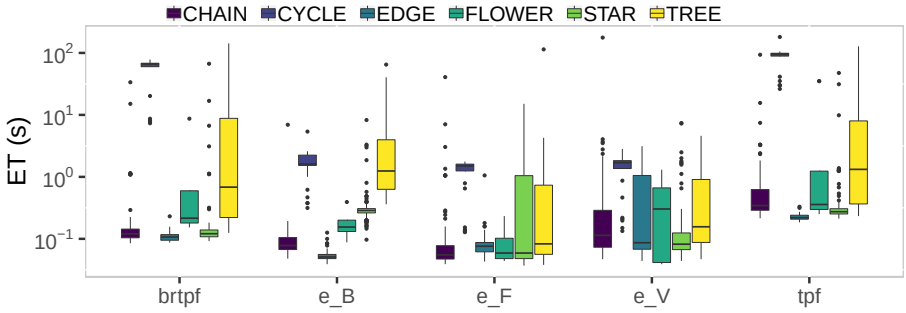
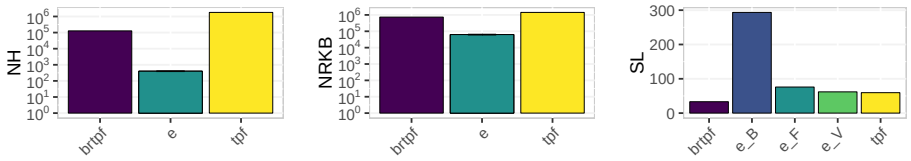


Fig. 4: ET per query shape for interfaces

Network Load. Figure 5 shows the average number of requests (NH) and average amount of data transferred from the servers to the clients (NRKB) per interface as studied earlier [8, 18]. NH and NRKB are independent from the endpoint used; the TPF interface has the higher NH and NRKB, while the endpoint interface has the lowest.



(a) NH (b) NRKB (c) SL

Fig. 5: NH, NRKB and SL

Figure 6 shows the number of transferred kBs from the server (NRKB) and from the client (NSKB) represented as a boxplot for each system and query load. Each query load has very different values for NRKB and NSKB. Naturally, the endpoints transfer considerably less kBs for both metrics. Relative values across interfaces are consistent except for the FLOWER- and STAR-shaped queries. While the endpoints have similar NSKB and higher NRKB values for STAR-shaped queries, the TPF and brTPF interfaces end up having considerably more data transfer for the FLOWER-shaped queries. This suggests that such queries result in a high number of intermediate results consistently with the BGP-restricted triple pattern selectivity reported for this query load in figures 2e and 2f. The number of HTTP requests (NH) is constant and amounts to one for the endpoint interface as expected, while it is higher for the brTPF and TPF interfaces. Moreover, it increases with respect to the number of triple patterns included in the query for brTPF and TPF interfaces. In general, an endpoint

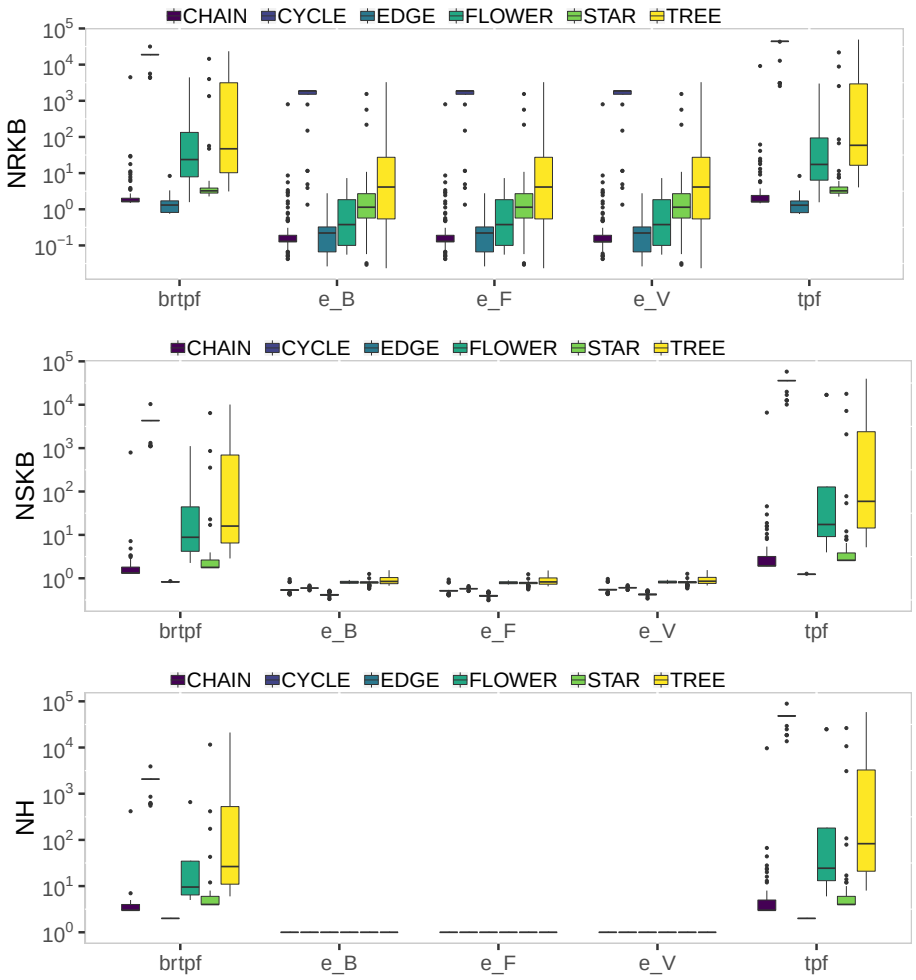


Fig. 6: NRKB, NSKB, and NS per query load

is the best performing interface and TPF is the worst performing interface for metrics related to the network load.

CPU Load. Figure 5c shows the CPU usage by the servers. While, overall, the endpoints use more CPU, using a Virtuoso endpoint uses nearly the same CPU as using a TPF server. A comparable CPU usage for Virtuoso suggests that by utilizing indexes and efficient query plans, it might be possible to achieve good CPU utilization while supporting complete SPARQL specification.

4.3 Multiple Clients

To stress the systems with multiple concurrent clients, we have executed experiments with 1, 16, 32, and 64 clients. We have processed Equal and Proportional query loads and report execution time (ET) that is the time elapsed since the beginning of processing the query loads until all the clients are done. In the figures

with box-plots, we show the distribution of the metrics in the query load. The results illustrate the trade-offs of using the different interfaces and endpoints.

Performance. While TPF and brTPF are designed to reduce the server load, they also considerably increase the execution time of queries. brTPF and TPF result in higher numbers of query timeouts than the endpoints (see Figure 7a).

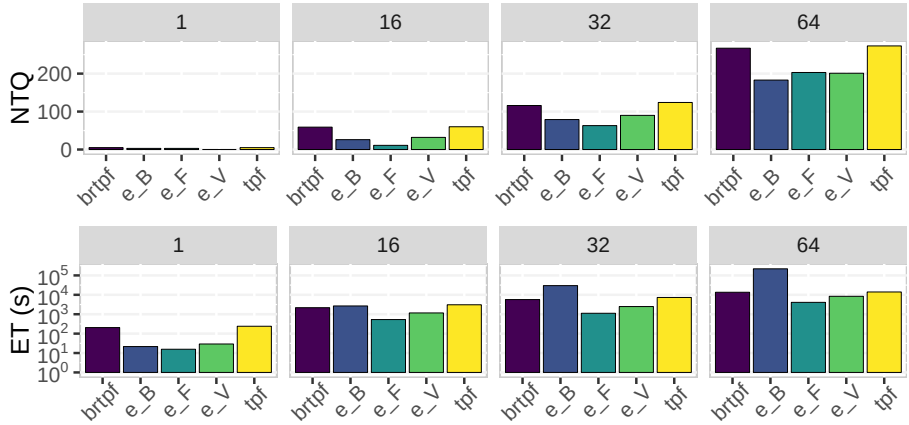


Fig. 7: Total Number of Timed out Queries (NTQ) and Execution Time (ET) for 1, 16, 32, and 64 clients

Figure 7b shows the execution time as traditionally presented in existing studies [8, 18]. For the endpoint interface, Fuseki achieves the best performance while Blazegraph shows the worst performance. Moreover, we can see that there are no changes in the relative performance of the endpoints for different number of clients except for 1 client, where Blazegraph performs slighly better than Virtuoso.

Figure 8 illustrates the execution time when each server is allocated 8GB of main memory instead of 21GB to assess whether the allocated memory to the server makes any difference. The execution times have a very similar trend compared to the execution times presented in Figure 7b. The only difference is that Virtuoso cannot handle 32 and 64 concurrent clients with 8GB of main memory.

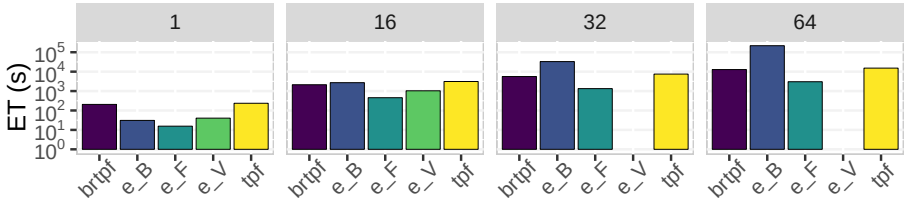


Fig. 8: Total ET for 1, 16, 32, and 64 clients (8GB of main memory)

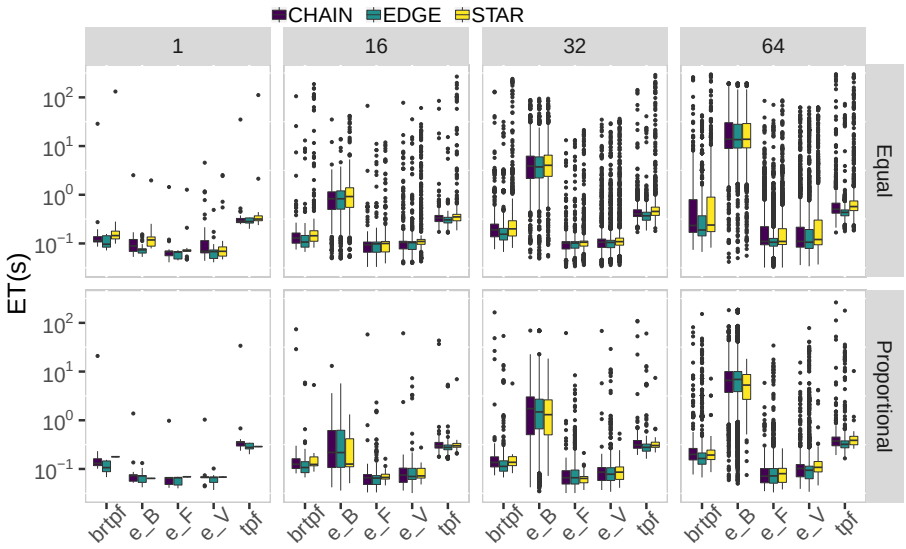


Fig. 9: ET per approach and query load for 1, 16, 32, and 64 clients

Figure 9 presents the execution time of the interface and backend combinations in a finer granularity for each query load and query shape. As shown in the figure, the endpoint interface produces the best performance when there is only a single client, which is in line with what we have seen with our single-client evaluation. However, when there are at least 16 clients, Fuseki and Virtuoso perform better than the other interfaces. Moreover, the advantages of Fuseki and Virtuoso are greater for the Equal query load, which shows that more complex queries, e.g., with more triple patterns, are processed more efficiently by these systems.

Network Load. Our experimental evaluation shows that there is a great difference in the number of bytes the clients receive (NRKB) for each of the different interfaces (Fig. 10). While NRKB naturally increases as the number of clients increases, the increment is more considerable for the TPF and brTPF interfaces.

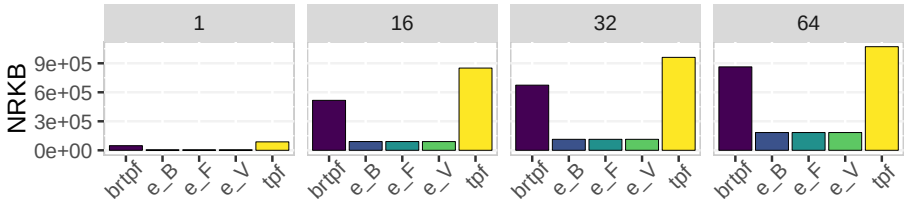


Fig. 10: NRKB for 1, 16, 32, and 64 clients

CPU Load. Figure 11 shows the average CPU loads of the servers. All the systems have more CPU load as the number of clients increases, the Blazegraph endpoint is the one that is affected the most. From using slighly more CPU than

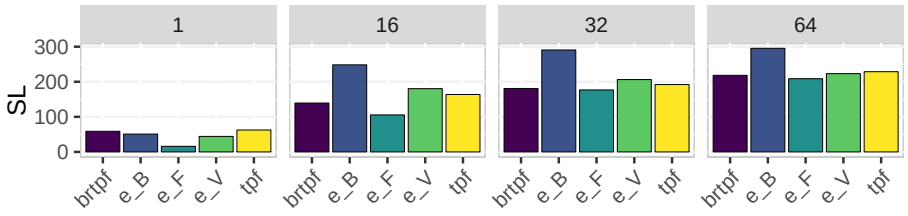


Fig. 11: Average CPU usage for the servers and their backends for 1, 16, 32, and 64 clients

Virtuoso and slighly less CPU than the TPF and brTPF interfaces for 1 client, it ends up using considerably more CPU than any other system. The difference between the CPU loads of endpoints is quite significant between 1 client and 16 clients, but not so much when the number of clients is further increased. For Virtuoso, later increases are so small that they are even smaller than the increases experienced by TPF and brTPF. For 32 and 64 clients, the CPU usage for all the systems, except Blazegraph, is quite similar.

5 Conclusion

In this paper, we presented an in-depth experimental evaluation of the state-of-the-art interfaces for querying linked data based on real query logs. We assessed the effect of query shapes on the performance of these interfaces.

The single-client evaluation results suggest that the shape of the query has a non-negligible effect on the performance of the interfaces. In addition, for complex query shapes like FLOWER and TREE, the Fuseki endpoint provides the best performance in terms of execution time, network load, and CPU load.

Our experiments clearly demonstrate that if the expected number of concurrent clients is not high, all the endpoints perform similarly well. However, if we examine the query loads and query shapes, we notice that such similarly good performance is due to Virtuoso processing the most complex query shapes more efficiently, while Fuseki processes the most simple query shapes more efficiently. While Fuseki handles the increase in the number of clients well, Blazegraph's performance deteriorates fast and Virtuoso aborts if less than 21GB of RAM are available. Differently from previous evaluations, our evaluation shows evidence that SPARQL endpoints can scale better than the TPF and brTPF interfaces, as is the case for Fuseki. For complex shapes, the difference in performance is considerably higher, and in such cases Fuseki represents a clearly better choice than the other systems. As future work we plan to study the performance of existing LDF interfaces for more complex configurations, which would include higher number of clients, more diverse query loads, other triple stores, networks with varying and controlled delays, and use of HTTP cache.

Acknowledgments. This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-4093-00301B and Aalborg University's Talent Management Programme.

References

1. G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212, 2014.
2. C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. SPARQL web-querying infrastructure: Ready for action? In *ISWC*, pages 277–293, 2013.
3. C. Bizer and A. Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
4. A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *PVLDB*, 11(2):149–161, 2017.
5. K. Clark, L. Feigenbaum, G. Williams, and E. Torres. SPARQL 1.1 protocol. W3C recommendation, W3C, Mar. 2013. `http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/`.
6. O. Erling and I. Mikhailov. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer, 2009.
7. J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF representation for publication and exchange (HDT). *J. Web Sem.*, 19:22–41, 2013.
8. O. Hartig and C. B. Aranda. Bindings-restricted triple pattern fragments. In *OTM Conferences*, pages 762–779, 2016.
9. O. Hartig, I. Letter, and J. Pérez. A formal framework for comparing linked data fragments. In *ISWC*, pages 364–382, 2017.
10. J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.
11. M. Luczak-Roesch, Z. A. Saud, B. Berendt, and L. Hollink. Usewod 2016 research dataset, 2016.
12. M. A. Martínez-Prieto, M. A. Gallego, and J. D. Fernández. Exchange and consumption of huge RDF data. In *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*, pages 437–452, 2012.
13. G. Montoya, C. Aebeloe, and K. Hose. Towards efficient query processing over heterogeneous RDF interfaces. In *DeSemWeb@ISWC*. CEUR-WS.org, 2018.
14. M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. DBpedia SPARQL benchmark - performance assessment with real queries on real data. In *ISWC*, pages 454–469. Springer, 2011.
15. M. Saleem, Q. Mehmood, and A. N. Ngomo. FEASIBLE: A feature-based SPARQL benchmark generation framework. In *ISWC*, pages 52–69, 2015.
16. M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, and A. N. Ngomo. How representative is a SPARQL benchmark? an analysis of RDF triplestore benchmarks. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 1623–1633, 2019.
17. B. B. Thompson, M. Personick, and M. Cutcher. The bigdata® RDF graph database. In *Linked Data Management*, pages 193–237. Chapman and Hall/CRC, 2014.
18. R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert. Triple pattern fragments: A low-cost knowledge graph interface for the web. *J. Web Sem.*, 37-38:184–206, 2016.