SUMMARY

The Semantic Web community has over recent years seen an ever increasing amount of data published as Linked Open Data (LOD) made available through services such as public SPARQL endpoints, dereferenceable URIs, or downloadable data dumps. Today, published LOD datasets span a broad range of different topics, such as geography, life sciences, government data, and general knowledge, and they usually contain several billions of triples. However, the availability of these datasets relies on the data providers to maintain access to the datasets. Creating and maintaining such services requires considerable resources and represents a huge burden on the part of the data providers. Without any kind of monetary incentive, this often leads to downtime and the data simply being unavailable.

Multiple recent advances propose to decentralize either the query processing load or data storage in order to lift the aforementioned burden off the data providers. For example, Peer-to-Peer (P2P) based interfaces attempt to remove the central point of failure altogether by putting data on participating clients that also function as servers and allowing such clients to communicate together. On the other hand, interfaces such as Triple Pattern Fragments (TPF) and derivatives like bindings-restricted TPF (brTPF) propose to lower the overall load on the server by making the client process the queries while the server only provides answers to individual triple patterns and, in the case of brTPF, bulks of bindings obtained from previously evaluated triple patterns. This means that the bulk of the query processing effort, such as expensive joins and SPARQL operators like `UNION` and `FILTER`, lies with the client rather than the server, lowering the load on the server.

In both cases, however, query processing relies on transferring individual triple patterns and resulting bindings over the network. This results in significant network traffic, creating a large overhead on query time. Such approaches completely ignore that evaluating small conjunctive subqueries on the server can be done with linear time complexity and can reduce the network traffic significantly since fewer intermediate results have to be transferred. This work therefore proposes a novel approach that reduces the network load and query time overhead significantly. We propose a novel RDF interface called Star Pattern Fragments (SPF) which builds upon TPF to processes star-shaped subqueries (star patterns) rather than individual triple patterns on the server while it still processes joins and SPARQL operators on the client. While this work focuses on TPF-based approaches, SPF is orthogonal to P2P systems and could be applied to such systems to provide the same benefits as highlighted in this paper for them as well.

We describe how SPF adapts the brTPF selector on the server to enable processing star patterns with bindings obtained from previously evaluated star patterns. Furthermore, we discuss the implications of such an adaptation of the selector on client-side query processing techniques. We propose an approach to process queries over an SPF server and implement both the client and server as an extension of an existing implementation of TPF.

We conduct an extensive study on the effects of evaluating such conjunctive star-shaped subqueries on the server rather than individual triple patterns. We therefore compare SPF to TPF, brTPF, and a SPARQL endpoint using several metrics such as query throughput, number of bytes transferred between the server and client, and the CPU load on the server. To conduct the study, we use a well-known benchmark suite to obtain three synthetic datasets containing 10 million, 100 million, and 1 billion triples as well as numerous queries. We split the query workload into several categories depending on the number of star patterns in the query (from 1 star pattern up to 3 star patterns) and include a workload that consists of path queries (i.e., queries with no star patterns). We run each workload over several configurations of up to 128 clients concurrently issuing queries to the server.

Our experiments clearly show that SPF is able to find an improved tradeoff for the distribution of the workload between the server and client. SPF reduces the network usage significantly both in terms of transferred bytes and number of performed server requests. In turn, SPF significantly increases performance. This is most apparent for the 1-star query workload where SPF is up to an order of magnitude faster than state of the art interfaces. This is a result of the fact that such queries can often be processed with a single server request. Moreover, even in the worst case where a query does not contain any star patterns, SPF is still as good as brTPF. Lastly, our experiments show that SPF maintains a relatively low CPU load on the server compared to brTPF and TPF. The experimental results thus show that SPF is able to increase performance under load while keeping the server load comparatively low. This motivates further research into obtaining a perhaps even more optimal balance between network and server load. Future research directions further include also processing object-based star patterns on the server and adapting P2P systems to use the approach as well.

This page was intentionally left blank.

Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns

Christian Aebeloe
caebel@cs.aau.dk
Aalborg University, Dept. of Computer Science
Selma Lagerlöfs Vej 300, DK-9220, Aalborg Øst, Denmark

ABSTRACT

The Semantic Web offers access to a vast Web of interlinked information accessible via SPARQL endpoints. Such endpoints offer a well-defined interface to retrieve results for complex SPARQL queries. The computational load for processing such queries, however, lies entirely with the server hosting the SPARQL endpoint, which can easily become overloaded and in the worst case not only become slow in responding but even crash rendering the data temporarily unavailable. Recently proposed interfaces, such as Triple Pattern Fragments, have therefore shifted the query processing load from the server to the client. For queries involving triple patterns with low selectivity, this can easily result in high network traffic and slow execution times. In this paper, we therefore present a novel interface, Star Pattern Fragments (SPF), which decomposes SPARQL queries into star-shaped subqueries and combines a lower network load with a higher query throughput and a comparatively low server load. Our experimental results show that our approach does not only significantly reduce network traffic but is also up to an order of magnitude faster in comparison to the state of the art interfaces under high query processing load.

KEYWORDS

SPF, star patterns, decentralization, query processing, semantic web, SPARQL, RDF, triples

1 INTRODUCTION

Over the previous few years, there has been a rapid increase in the amount of data published as Linked Open Data (LOD) made available through services such as public SPARQL endpoints, dereferenceable URIs, or downloadable data dumps. Several knowledge graphs have previously been published spanning multiple different topics, such as general knowledge (e.g., DBpedia [11]), government data (e.g., US Government LOD [17]), and geography (e.g., LinkedGeoData [36]). Knowledge graphs today can contain several billions of data triples. For instance, Wikidata [40] contains over 12 billion triples and Bio2RDF [12] contains over 10 billion triples.

Access to such datasets, however, relies completely on the data providers to maintain aforementioned services like a public SPARQL endpoint; however, this requires considerable resources on the part of the data providers. As highlighted in previous studies [2, 38], this often means that the provided services quickly become slow and unresponsive under high load, and in the worst case can become unavailable [5, 37].

Despite recent efforts to speed up SPARQL query processing under high querying load [15, 26, 38], answering SPARQL queries is still an expensive task. In fact, deciding whether a set of bindings is an answer to a query has been shown to be at least NP-complete [30]. Still, Triple Pattern Fragments (TPF) [38] have provided interesting insights into the problem and a novel way to approach it. TPF limits the load on the server by sharing the computational load between the server and the client. While the server evaluates individual triple patterns, the client handles remaining query processing tasks. This increases the availability of the server and ensures more efficient query processing during periods with high load.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
select distinct * where {
 ?p1 dbo:country dbr:Germany. # tp1: 18,174 matches
 ?p1 dbo:award ?a .           # tp2: 90,933 matches
 ?p1 dbo:birthDate ?bd1 .     # tp3: 1,740,614 matches
 ?p2 dbo:country dbr:Norway . # tp4: 5,520 matches
 ?p2 dbo:award ?a .           # tp5: 90,933 matches
 ?p2 dbo:birthDate ?bd2       # tp6: 1,740,614 matches
}
```

Listing 1: Find Germans and Norwegians that have won the same award and their birth dates.

Nevertheless, there are cases where TPF is significantly less efficient than SPARQL endpoints. Consider, for example, the SPARQL query shown in Listing 1 over the DBpedia dataset [11]. Executing this query using TPF requires transferring a huge number of intermediate results. In addition, the TPF client sends a server request for each binding obtained from the previously evaluated triple patterns, which results in a high number of calls to the server and thus creates a large overhead when processing the query, decreasing its performance.

TPF-based derivatives, such as Bindings-Restricted Triple Pattern Fragments (brTPF) [15], have different ways to address this issue. For instance, brTPF uses block nested loop-like joins where a triple pattern is evaluated once per a group of N bindings obtained from the previously evaluated triple patterns ($5 \leq N \leq 50$ in [15]). While this results in significantly fewer calls to the server, it still incurs relatively high network traffic.

These approaches ignore the potential of evaluating larger conjunctive subqueries. Such subqueries can (i) be computed relatively efficiently on the server [30] and (ii) reduce the network traffic since fewer intermediate results are transferred. Subqueries, such as `{tp1.tp2.tp3}` and `{tp4.tp5.tp6}` in Listing 1, do not require full SPARQL expressiveness. While there could potentially be several ways to decompose SPARQL queries, we focus on decomposition into star-shaped subqueries.

In this paper, we investigate the effects of evaluating star-shaped subqueries on the server, while still processing queries on the client, in terms of the network usage, client load, and server load under high query load. We propose a novel interface that is able to combine

a lower network load with a comparatively low server load by decomposing SPARQL queries into star-shaped subqueries, and in doing so improve the overall query processing performance while also ensuring high availability.

In summary, this paper makes the following contributions:

- A definition of Star Pattern Fragments (SPF), a novel RDF interface that reduces network usage while maintaining a low server load.
- A formalization and an implementation of an SPF server that processes star-shaped subqueries.
- Client-side query processing strategies to efficiently compute answers to SPARQL queries using an SPF server to process star-shaped subqueries and process queries with any SPARQL operator.
- A thorough evaluation of the SPF interface using three different synthetic datasets with up to 1 billion triples and several diverse stress testing query workloads from a well-known benchmark suite [4].

This paper is organized as follows. Section 2 discusses related work, Section 3 introduces preliminary information used throughout the paper, Section 4 formally defines Star Pattern Fragments, Section 5 describes query processing over SPF servers, Section 6 discusses experimental results, and finally Section 7 concludes the paper.

2 RELATED WORK

One of the most popular interfaces for querying RDF data is SPARQL endpoints. However, several studies [5, 37] have previously highlighted the fact that such endpoints are often unavailable, meaning that accessing data can sometimes be impossible. High availability of RDF datasets has been achieved by decentralizing the storage of data and distributing query processing tasks between clients and servers [38]. Decentralization of the data storage has previously been achieved by using Peer-to-Peer (P2P) architectures [2, 8, 21] and federated query engines [27, 35]. In this section, we provide a brief overview over related decentralized systems. We discuss in detail decentralized approaches, how they attempt to solve the availability problem, and their shortcomings in Appendix A.

P2P systems remove the central server altogether by placing a limited local datastore on each node. The nodes thus act like both clients and servers and communicate with each other to obtain query answers. Structured P2P systems [8, 21, 22] apply a structured overlay over the network. For instance, UniStore [22] uses Dynamic Hash Tables (DHTs) to impose on which nodes have to store which data. Unstructured P2P systems, on the other hand, do not apply such an overlay. For example, Piqnic [2] defines an architecture where connections between nodes are random. However, even if P2P systems increase the availability of datasets, they are vulnerable to churn (when nodes frequently leave and join the network) [8, 21, 22] or can cause very high network traffic [3] (when queries have large numbers of intermediate results). In any case, Star Pattern Fragments (SPF) are orthogonal to P2P systems. For example, [3] could be extended with SPF to achieve a similar reduction of intermediate results as described in Section 6.

Federated query engines [27, 35] divide SPARQL query processing over multiple SPARQL endpoints. Nevertheless, they sometimes fail to generate optimal query plans that transfer the minimum amount of data from the endpoints to the engine and therefore increase the load on SPARQL endpoints. Query optimization techniques for federated engines, such as [28], consider decomposing SPARQL queries into star-shaped groups that can be evaluated by a single SPARQL endpoint. Star-shaped query decomposition has also been used in [39] to improve the query execution time. These techniques are similar to the approach presented in this paper since they also utilize star-shaped decomposition. However, these approaches execute star-shaped subqueries on SPARQL endpoints on which much more complex queries can be executed concurrently.

Triple Pattern Fragments (TPF) [38] were proposed to improve the server availability under heavy load. TPF servers only process individual triple patterns and therefore have a lower processing burden than SPARQL endpoints. TPF clients rely on either left-deep join trees [38], a metadata based strategy [18], or adaptive query processing techniques and star-shaped decomposition [1] to determine the execution order of the triple patterns. While TPF reduces the load on the server in general, it puts much more load on the client and incurs more network traffic. Furthermore, Heling et al. [16] found that the performance of TPF is heavily affected by variables such as the triple pattern type[1] and the fragment cardinality. Bindings-Restricted TPF (brTPF) [15] was proposed to reduce the network traffic by coupling triple patterns and bindings obtained from previously evaluated triple patterns. Despite improving the availability of RDF data, all these approaches cause a large number of calls to the server during query processing. On the other hand, hybridSE [26] relies on both SPARQL endpoints and brTPF servers to process queries more efficiently than the TPF-based interfaces. SPARQL subqueries with a large number of intermediate results are evaluated using SPARQL endpoints to overcome limitations of TPF clients. However, since hybridSE may send complex subqueries to the endpoint, and endpoints have downtime [5], this leaves the approach vulnerable to downtime.

Other systems use more complex techniques to address some of the issues posed by TPF. SaGe [25], for example, uses a preemptive model that suspends queries after a fixed time quantum, so as to not starve simpler queries from system resources, whereafter they can be resumed upon client request. This, however, often leads to a large amount of requests to the server, which is exactly what we are addressing in this paper. Smart-KG [6], on the other hand, ships star-shaped partitions to the client during query processing. This decreases the amount of requests to the server, since partitions already shipped to the client can be evaluated directly on the client. However, this can in some cases lead to unnecessary data transfer during query processing, since the entire partition is shipped regardless of previously obtained object bindings.

In this paper, we instead propose an interface that provides a novel distribution of query processing tasks where, differently from the approaches discussed above, joins in star-shaped subqueries are evaluated by the server. Such computations do not significantly increase the query load because star-shaped subqueries can be answered in linear complexity [30]. Our approach therefore achieves a reduction on the data transfer and the execution time without having a negative impact on the data availability.

[1]The type of a triple pattern is defined with respect to the position of variables in the triple pattern.

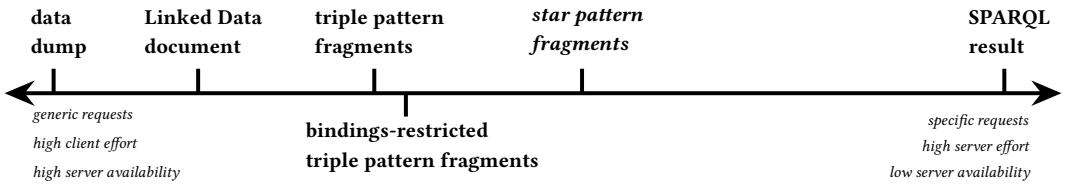


Figure 1: HTTP interfaces for RDF data (adapted from [15, 38]).

3 PRELIMINARIES

Star Pattern Fragments (SPF) build upon TPF [38] and brTPF [15] to provide more efficient access to large knowledge graphs under load. In this section, we present preliminaries for both large-scale knowledge graphs and decentralized query processing techniques.

3.1 Resource Description Framework

The recommended format for storing semantic data is the Resource Description Framework (RDF)[2].

Definition 1 (RDF Triple). *Given the infinite and disjoint sets U (set of all URIs), B (set of all blank nodes), and L (set of all literals), an RDF triple is a triple of the form $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$, where s, p, o are called subject, predicate, and object.*

A knowledge graph (RDF graph) $\mathcal{G}$ is a finite set of RDF triples. Today, SPARQL[3] is the standard language for querying RDF data. A SPARQL query contains a set of *triple patterns* which, given the additional infinite set V (disjoint with U, B and L) of all variables, are triples of the form $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$.

Definition 2 (Star Pattern). *A star pattern is a set of n triple patterns, $\{(s_1, p_1, o_1), \ldots, (s_n, p_n, o_n)\}$ such that either the subjects or objects of all these triple patterns are the same, i.e., $s_i = s_j$ for all $1 \leq i, j \leq n$ (subject-based star patterns) or $o_i = o_j$ for all $1 \leq i, j \leq n$ (object-based star patterns).*

Since subject-based stars are much more common in real datasets [34] and considering both types of star patterns requires complex adaptation of possible query optimization strategies that are outside the scope of this paper, we will for the remainder of the paper focus only on subject-based star patterns.

3.2 Linked Data Fragments

Linked Data Fragments (LDFs) [38] consider only blank-node-free RDF triples. An LDF consists of the following three elements:

- **Data**: A subset of $\mathcal{G}$'s triples
- **Metadata**: RDF triples that describe the data
- **Controls**: Links and forms to retrieve other LDFs of the same or other knowledge graphs

Any knowledge graph made available on the Web, in any format, can be described as an LDF.

Definition 3 (Selector Function [38]). *Given $\mathcal{T}^* = U \times U \times (U \cup L)$, the set of all blank-node-free RDF triples, a selector function s is a function such that $s : 2^{\mathcal{T}^*} \rightarrow 2^{\mathcal{T}^*}$.*

That is, a selector function takes as input a set of blank-node-free RDF triples, and outputs a set of blank-node-free RDF triples. Note that the output could in principle contain triples that are not in the input, e.g., `CONSTRUCT` queries. However, in most cases, the output corresponds to a subset of the input.

Definition 4 (Hypermedia Controls [38]). *A hypermedia control is a function that maps from some set to U.*

A URI is a zero-argument hypermedia control, i.e., a constant function, and a form is a multi-argument hypermedia control. In the case of LDF, the domain of a hypermedia control is a set of selector functions, encoded as URLs.

Definition 5 (Linked Data Fragment [38]). *Given a knowledge graph $\mathcal{G}$, a Linked Data Fragment (LDF) of $\mathcal{G}$ is a 5-tuple $f = \langle u, s, \Gamma, M, C \rangle$, with*

- *a source URI u,*
- *a selector function s,*
- *the result of applying s to $\mathcal{G}$, $s(\mathcal{G}) = \Gamma$,*
- *a set of additional triples M that describes metadata, and*
- *a finite set of hypermedia controls C.*

An LDF server should divide each fragment $f = \langle u, s, \Gamma, M, C \rangle$ into reasonably sized LDF pages $\phi = \langle u', u_f, s_f, \Gamma', M', C' \rangle$ containing (i) the URI u' from which ϕ could be obtained and $u' \neq u$, (ii) $u_f = u$, (iii) $s_f = s$ (iv) $\Gamma' \subseteq \Gamma$, (v) $M' \supseteq M$, and (vi) $C' \supseteq C$. M' and C' are supersets of M and C, since they also contain additional metadata and controls that are specific to the LDF page. Having additional metadata and controls makes it possible for clients to avoid downloading very large chunks of data accidentally [38].

4 STAR PATTERN FRAGMENTS

In between SPARQL endpoints, which handle all the query processing load on the server, and TPF, which processes only triple patterns on the server and handles the rest of query processing load on the client, there is a lot of potential for other interfaces that provide a better way of sharing query processing load between server side and client side. For instance, processing conjunctive subqueries (e.g., star patterns) on the server can result in less network traffic while it does not impose a high additional server load, which is evident from our experiments in Section 6.

In this section, we formally define Star Pattern Fragments (SPF) as an extension of brTPF [15] that exposes an HTTP interface for processing star pattern queries in addition to processing individual triple pattern queries. This increases the server load slightly; however, for queries with large intermediate results (such as Listing 1), this is preferable to ensure fewer requests to the server, which results in lower network traffic and faster query processing. The

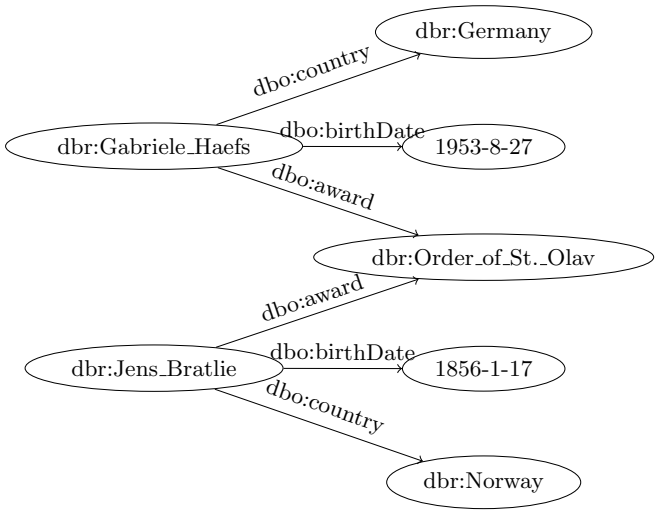
[2] https://www.w3.org/TR/rdf11-concepts/
[3] https://www.w3.org/TR/sparql11-query/

```
sp = {(?p2, dbo:country, dbr:Norway),
      (?p2, dbo:award, ?a),
      (?p2, dbo:birthDate, ?bd2)}

μ(?p2)=dbr:Jens_Bratlie
μ(?db2)=1856−1−17
μ(?a)=dbr:Order_of_St._Olav

s(sp,∅)(G)={(dbr:Jens_Bratlie, dbo:country, dbr:Norway),
      (dbr:Jens_Bratlie, dbo:award, dbr:Order_of_St._Olav),
      (dbr:Jens_Bratlie, dbo:birthDate, 1856−1−17)}
```

(a) sp **and** $s_{(sp,\emptyset)}(\mathcal{G})$



(b) RDF Graph

Figure 2: Star Pattern, Star Pattern-Based Selector Function, and RDF Graph.

relative position of SPF between different RDF interfaces is shown in Figure 1.

Logically, an SPF over a given knowledge graph $\mathcal{G}$ has the following properties:

- **Data**: All RDF stars in $\mathcal{G}$ that match a given star pattern
- **Metadata**: An estimate of the number of stars that match the given star pattern
- **Controls**: A hypermedia form that allows the client to retrieve any SPF of the same knowledge graph

Let $[[sp]]_{\mathcal{G}}$ be the answer to a star pattern sp over a knowledge graph $\mathcal{G}$. $[[sp]]_{\mathcal{G}}$ is a set of *solution mappings*, i.e., partial mappings $\mu : V \mapsto (U \cup L)$. A set of blank-node-free RDF triples T is said to be *matching triples* for a star pattern sp, denoted $T[sp]$, if there exists a solution mapping μ in $[[sp]]_{\mathcal{G}}$ such that $T = \mu[sp]$ where $\mu[sp]$ denotes the triples (or triple patterns) obtained by replacing the variables in sp with values according to μ. As mentioned earlier, we do not consider star patterns that join on objects instead of subjects since they are relatively rare [34]. It could, however, be interesting to consider as perspective work.

Similar to how brTPF [15] couples bindings and triple patterns, we couple bindings obtained from previously evaluated star patterns with subsequent star patterns to decrease the network traffic.

Definition 6 (Star Pattern-Based Selector Function). *Given a star pattern sp and a finite sequence of solution mappings Ω, the star pattern-based selector function for sp and Ω, $s_{(sp,\Omega)}$, is the selector function that, for every knowledge graph $\mathcal{G}$, is defined as follows:*

$$s_{(sp,\Omega)}(\mathcal{G}) = \begin{cases} \{t \in T \mid T \subseteq \mathcal{G} \wedge T[sp] & \text{if } \Omega = \emptyset \\ \{t \in T \mid T \subseteq \mathcal{G} \wedge T[sp] \wedge & \\ \quad \exists \mu \in [[sp]]_{\mathcal{G}}, \mu' \in \Omega : & \\ \quad \mu[sp] = T \wedge \mu' \subseteq \mu\} & \text{otherwise.} \end{cases}$$

The simplest star pattern consists of a single triple pattern. For this reason, SPF is backwards compatible with both TPF [38] and brTPF [15]. A star pattern request with a single triple pattern corresponds to a single triple pattern request for TPF and brTPF. As such, applying the star pattern-based selector function in this case would be equivalent to applying either the triple pattern-based selector or the bindings-restricted triple pattern-based selector.

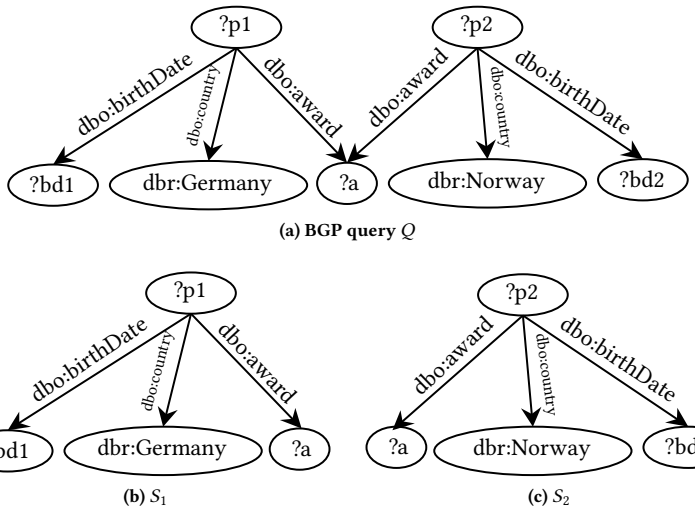
Consider the star pattern sp and the knowledge graph $\mathcal{G}$ given in Figure 2. The star pattern-based selector function $s_{(sp,\emptyset)}(\mathcal{G})$ retrieves the three triples from $\mathcal{G}$ that include dbr:Jens_Bratlie as subject, as shown in Figure 2a.

In order to formally define SPF, we adapt the general definition of LDF given in [38]. An SPF is defined as follows:

Definition 7 (Star Pattern Fragment). *Given a control c, a c-specific LDF collection F is called a Star Pattern Fragment collection if, for every possible star pattern sp and any finite sequence Ω of distinct solution mappings, there exists one LDF $\langle u, s, \Gamma, M, C\rangle \in F$, called a Star Pattern Fragment, that has the following properties:*

1. *s is the star pattern-based selector function for sp and Ω.*
2. *There exists a triple <u, void:triples, cnt> $\in M$ with cnt representing an estimate of the cardinality of Γ, that is, cnt is an integer that has the following two properties:*
   (a) *If $\Gamma = \emptyset$, then cnt = 0.*
   (b) *If $\Gamma \neq \emptyset$, then cnt > 0 and $abs(|\Gamma| - cnt) \leq \epsilon$ for some F-specific threshold ϵ.*
3. *$c \in C$.*

Notice that SPF, like TPF and brTPF, is hypermedia and therefore contains hypermedia controls (cf. Definition 7). An SPF can be obtained by forming a request from a star pattern and including already bound values (e.g. object values). Since we focus on assessing the applicability of our approach in general, we will not include a full hypermedia description of SPF in this paper; nevertheless,



(a) BGP query Q

(b) S_1

(c) S_2

Figure 3: Star decomposition of Q (Listing 1) into S_1 and S_2.

such a description can be formalized similarly to the hypermedia controls provided by TPF.

The definition of SPF, and its hypermedia controls, allows for both subject-based and object-based star patterns to be evaluated on the server. This allows the client to employ a complex decomposition strategy that can utilize both types of star patterns. However, in order to investigate the applicability of our model independently of possibly complex query decomposition strategies that would be necessary on the client if we consider both types of star patterns, and since subject-based star patterns are much more common in real query loads [34], for the rest of the paper we will focus on subject-based star patterns only.

5 QUERY PROCESSING

The SPF interface processes queries using resources from both the server and the client. The server provides fragments as answers to requests whereas the client processes all other SPARQL operators. Differently from RDF interfaces such as TPF and brTPF, SPF does not define fragments based on triple patterns but rather based on star patterns.

Query processing using SPF relies on a server and a client, each managing different tasks. The general outline of how query processing works for a given SPARQL query Q is as follows:

(1) For each BGP $B \in Q$, decompose B into star-shaped subqueries and determine the join order.
(2) Obtain intermediate results for each of B's subqueries by applying the star pattern-based selector on the server.
(3) Compute the final query result combining the intermediate bindings of the subqueries, and process all the remaining SPARQL operators in Q on the client side.

5.1 Client-Side Query Processing

To process a SPARQL query, an SPF client first decomposes the query into star-shaped subqueries. This decomposition is necessary to process more complex SPARQL queries than star-shaped queries using an SPF server. In the rest of this section, we focus on Basic Graph Pattern (BGP)[4] queries. Nevertheless, our approach can be used for full SPARQL specification including queries with one or more BGPs combined using operators such as `OPTIONAL` and `UNION` and queries with `FILTER` constraints.

Definition 8 (Subject-Based Star Decomposition). *Given a BGP query $Q = \{tp_1, \ldots, tp_n\}$, the star decomposition of Q is $\mathcal{S}(Q) = \{S_1, \ldots, S_m\}$ such that (i) $m \leq n$, (ii) all $S_j \in \mathcal{S}(Q)$ are subject-based star patterns (Definition 2), (iii) for all $1 \leq i \leq n$, there exists exactly one j such that $1 \leq j \leq m$ and $tp_i \in S_j$, and (iv) for all $1 \leq j \leq m$ and $tp_i \in S_j$, $tp_i \in Q$.*

Using Definition 8, a BGP query can be partitioned into a set of star patterns where each corresponds to a specific variable on subject position. All triple patterns are then part of a specific star pattern with a shared subject. This definition ensures that the query is decomposed into non-overlapping star patterns. However, for some queries, e.g., queries shaped as a chain, this definition can result in many star patterns with only a single triple pattern. Query processing is identical to brTPF in this case, since it requires evaluating one triple pattern at a time.

An example of using Definition 8 to partition the BGP query Q Listing 1) can be seen in Figure 3. The star decomposition of Q results in one star pattern per variable on subject position. In this example, variables ?p1 and ?p2 are both positioned as the subject of at least one triple pattern, and so the resulting star patterns are rooted in these variables. Figures 3b and 3c show the output star patterns S_1 and S_2, respectively.

When processing a BGP B, the SPF client performs the following steps:

(1) Obtain the star decomposition (Definition 8) of Q, i.e., $\{S_1, \ldots, S_n\}$.

[4] A BGP is a set of triple patterns, https://www.w3.org/TR/rdf-sparql-query/#BasicGraphPatterns

(2) Determine the join order of $S_1, \ldots, S_n$. More selective star patterns (star patterns with lower cardinality) are evaluated first. To determine the join order of $S_1, \ldots, S_n$, we send an SPF request for each S_i to retrieve its first page of results. The first page of a star pattern fragment contains the estimated cardinality of it (as described in Definition 7), and we use the estimated cardinality to determine the order.
(3) Send requests to the SPF server for each $S_1, \ldots, S_n$ with the solution mappings obtained from the already processed star patterns.

An example of processing a SPARQL query over an SPF server is shown in Appendix C.

5.2 Server-Side Query Processing

An SPF server is able to answer any syntactically valid star pattern. Upon receiving a request for a star pattern, the SPF server matches the star pattern to the knowledge graph using the star pattern-based selector function. An SPF request includes a star pattern sp, a finite sequence of distinct solution bindings Ω, and a page number p. The server processes such a request for over a knowledge graph $\mathcal{G}$ using the following two steps:

(1) Find the set of triples $s_{(sp,\Omega)}(\mathcal{G})$ (Definition 6).
(2) Return an LDF page ϕ that corresponds to the requested page p such that $\phi.\Gamma'$ consists of sets of matching triples $\mu[sp]$ where $\forall t \in \mu : t \in s_{(sp,\Omega)}(\mathcal{G})$

These results are then processed by the client, which joins them with results to other star patterns in the query, thereby computing the query answer.

An SPF server supports both the TPF and brTPF selectors in addition to the SPF selector. The server chooses which method to invoke based on the received request. For instance, the SPF method is invoked only if the request contains an SPF selector. In practice, the TPF and brTPF selectors would only be rarely used with an SPF client. However, having all three methods available in the server has two advantages. First, it makes the server compatible with both the TPF and brTPF clients. Second and more importantly, in the worst case where all star patterns have exactly one triple pattern, SPF still performs as good as brTPF as evident from our experiments in Section 6.

5.3 Implementation Details

We implemented the SPF server using Java 8 and the SPF client using Node.js[5].

Server. The SPF server is implemented as an extension of the Java implementation of the TPF server[6]. Our server implementation uses `HDT` [13] as backend. `HDT` is originally proposed to process a single triple pattern over a knowledge graph efficiently. However, we extended this implementation to also be able to process the star pattern requests over the `HDT` backend. The SPF server uses Characteristic Sets [29] to provide cardinality estimations.

Client. We extended the TPF Node.js client[7] to accommodate not only SPF requests but also brTPF requests. Thus, and in line with TPF [38] and brTPF [15], the SPF client uses a pipeline of iterators that represent a left-deep join tree. However, TPF and brTPF define the join operations on triple patterns, whereas SPF defines join operations on star patterns. The star patterns within a query are ordered based on the cardinality estimations for the star patterns provided by the server.

[5]Implementation is available on our GitHub https://github.com/Chraebe/StarPatternFragments.
[6]https://github.com/LinkedDataFragments/Server.java
[7]https://github.com/LinkedDataFragments/Client.js

6 EXPERIMENTAL EVALUATION

In order to investigate whether SPF, which processes SPARQL queries using star decomposition, increases query throughput by combining a lower network load with a comparatively low server load, we ran experiments where we compared SPF, TPF [38], brTPF [15], and a SPARQL endpoint.

6.1 Experimental Setup

Dataset and Queries: We used the WatDiv benchmark [4] to generate three datasets with 10 million, 100 million, and 1 billion triples respectively. This allows us to also test the scalability of SPF. The characteristics of the three datasets can be seen in Table 1.

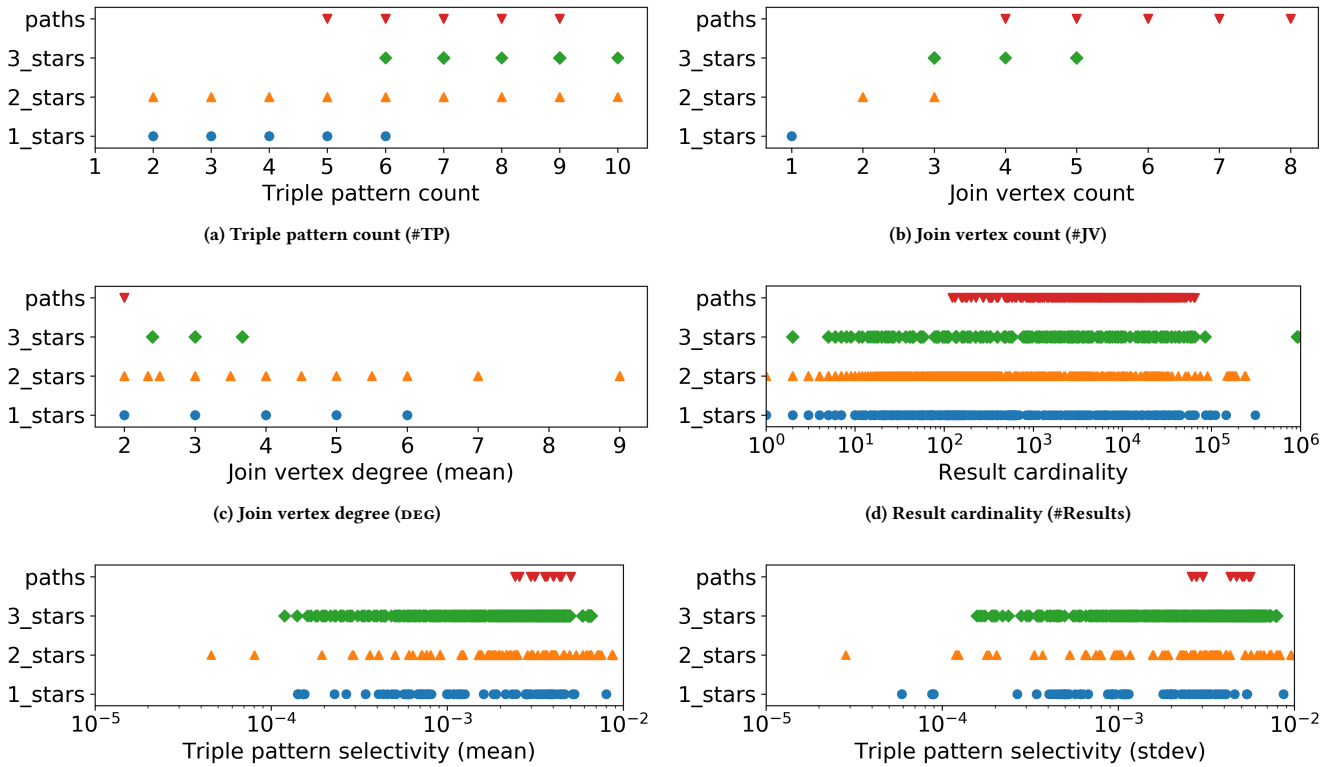
Table 1: Characteristics of used datasets

Dataset	#triples	#subjects	#predicates	#objects
`watdiv10M`	10,916,457	521,585	86	1,005,832
`watdiv100M`	108,997,714	5,212,385	86	9,753,266
`watdiv1000M`	1,092,155,948	52,120,385	86	92,220,397

To study the impact of the number of star-shaped subqueries, and to stress-test our approach, we used the WatDiv query generator to obtain query loads with 0-3 star patterns[8]. Query loads only include queries with at least one answer and the queries with zero star patterns consist of chained triple patterns with object-subject joins (path patterns). In total, we generated 25,600 distinct SPARQL queries, i.e., 200 queries for each client divided into four distinct and evenly sized groups: 50 queries consisting of one star pattern (`1-star`), 50 queries consisting of two star patterns (`2-stars`), 50 queries consisting of three star patterns (`3-stars`), and finally 50 queries consisting of a path pattern (`paths`). Figure 4 shows the following characteristics for each query load over `watdiv100M` (presented in [4]): Triple pattern count (#TP, Figure 4a), join vertex count (#JV, Figure 4b), join vertex degree (DEG, Figure 4c), i.e., the number of triple patterns incident on a join vertex, result cardinality (#Results, Figure 4d), and triple pattern selectivity ($\text{SEL}_{\mathcal{G}}(tp)$), i.e., the ratio of cardinality of a triple pattern to the size of the knowledge graph, mean (Figure 4e) and standard deviation (Figure 4f). We furthermore add the `union` query load that contains the combined queries from `1_star`, `2_stars`, `3_stars` and `paths`. All query loads (except `1_star`) include queries with subject-object joins. Queries contain 0-2 triple patterns with bound objects.

Experimental Setup: To assess how the interfaces perform under different loads, we ran experiments over eight configurations with 2^i clients concurrently issuing queries to the server in each configuration ($0 \leq i \leq 7$), i.e., up to 128 clients. In the configuration with 2^i clients, a total of 200×2^i queries are executed and at most

[8]We considered star patterns with two or more triple patterns with subject-subject joins



(a) Triple pattern count (#TP)

(b) Join vertex count (#JV)

(c) Join vertex degree (DEG)

(d) Result cardinality (#Results)

(e) TP selectivity ($\text{SEL}_{\mathcal{G}}(tp)$) mean

(f) TP selectivity ($\text{SEL}_{\mathcal{G}}(tp)$) stddev

Figure 4: Characteristics of all query loads (WatDiv query loads over `watdiv100M`).

2^i queries are executed concurrently as each client executes one query at a time. Each query load was run separately to assess the impact of it on the performance of the interfaces.

Hardware Setup: To run the clients, a virtual machine (VM) running all 128 clients concurrently was used. The VM had 128 vCPU cores with a clock speed of 2.5GHz, 64KB L1 cache, 512KB L2 cache, 8192KB L3 cache, and 2TB main memory. Each client was limited to use just one vCPU core and 15GB RAM. The LDF server and the SPARQL endpoint were run, at all times, on a server with 32 vCPU cores, with a clock speed of 3GHz, 64KB L1, 4096KB L2, and 16384KB L3 cache, and a main memory of 128GB.

Evaluation Metrics: We used the following evaluation metrics in our experimental study.

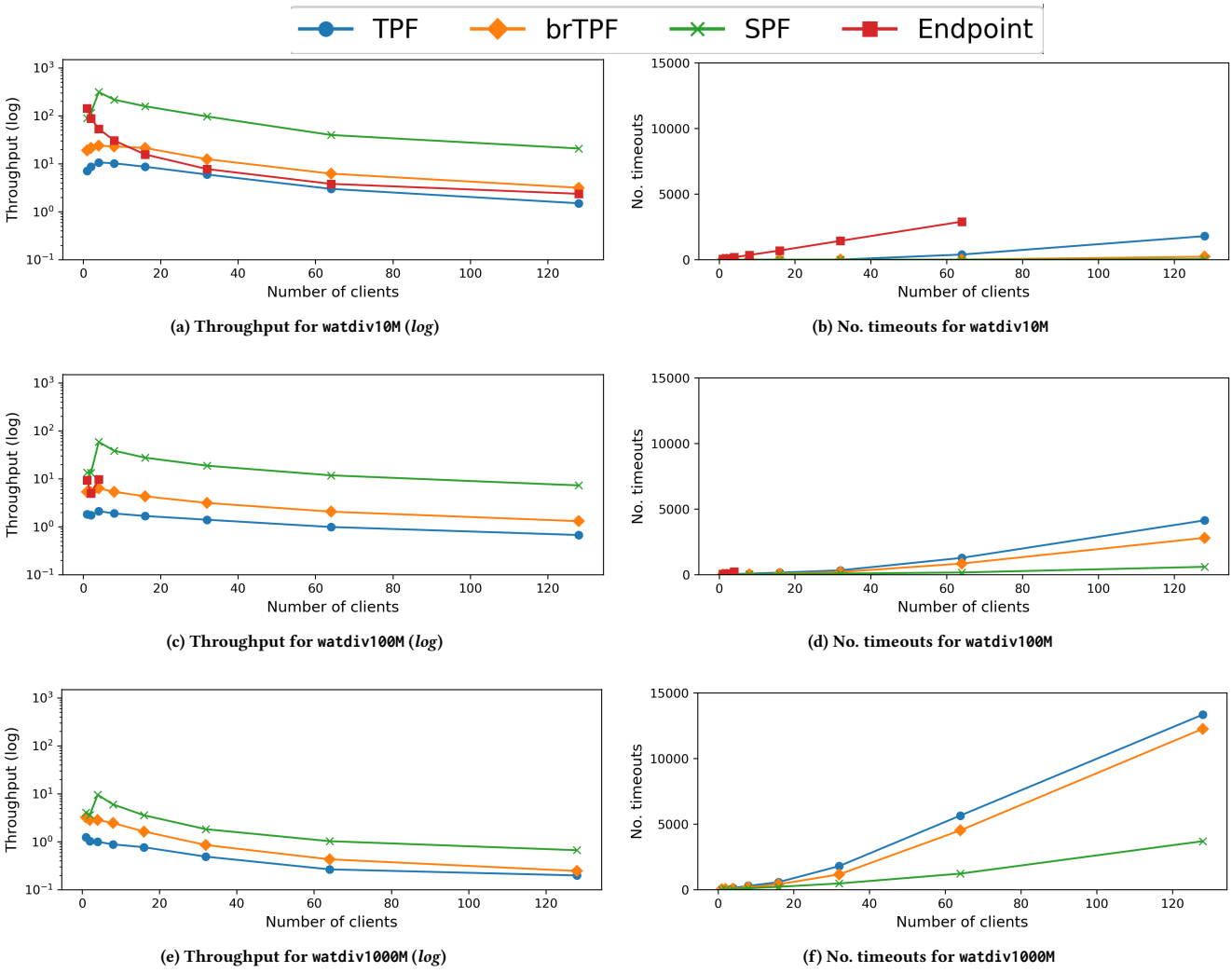
- *Number of Requests to the Server (NRS)*: The number of requests the client issues to the server while processing a query.
- *Throughput*: The number of queries processed per minute.
- *Query Execution Time (QET)*: The amount of time (in milliseconds) elapsed since a query is issued until its processing is finished.
- *Query Response Time (QRT)*: The amount of time (in milliseconds) elapsed since a query is issued until the first result is computed.
- *Number of Transferred Bytes (NTB)*: The amount of data transferred (in bytes) between the client and the server while processing a query (both from and to the server).
- *CPU Load (CPU)*: The average CPU load on the server (in percentage).

Software configuration: We used Virtuoso Open-Source version 7.2.5 to run the SPARQL endpoint, configured to use up to 32 threads at a time (one per vCPU core on the server) with the following variables set[9]:

- `NumberOfBuffers` = 9735000
- `MaxDirtyBuffers` = 7301250
- `ResultSetMaxRows` = 2097150
- `MaxQueryCostEstimationTime` = 60000

We chose Virtuoso as the SPARQL endpoint since it is the endpoint that performs the best under high query loads in terms of throughput and CPU usage [38]. The LDF page size was set to 100, and the maximum number of elements in Ω was set to 30 for both brTPF and SPF. The timeout was set to 600 seconds (10 minutes).

[9] `NumberOfBuffers` and `MaxDirtyBuffers` uses the recommended configuration from http://vos.openlinksw.com/owiki/wiki/VOS/VirtRDFPerformanceTuning given the server resources. `ResultSetMaxRows` was set to the maximum amount of rows Virtuoso allows in a 64-bit system, and `MaxQueryCostEstimationTime` was set to a large number to avoid rejection of requests by the server.



(a) Throughput for `watdiv10M` (*log*)

(b) No. timeouts for `watdiv10M`

(c) Throughput for `watdiv100M` (*log*)

(d) No. timeouts for `watdiv100M`

(e) Throughput for `watdiv1000M` (*log*)

(f) No. timeouts for `watdiv1000M`

Figure 5: Throughput (#queries/m), CPU load and no. timeouts for `union` over the different WatDiv datasets.

6.2 Experimental Results

Our objective is to asses whether or not SPF can execute SPARQL queries that contain star patterns more efficiently by greatly reducing the network traffic, but without incurring too much additional load on the server. Furthermore, we want to investigate if SPF is, in the case of path queries, still as good in terms of performance as brTPF. In this section, we will show the most important results, while Appendix C contains the remaining experimental results.

The SPARQL endpoint became unresponsive (i.e. all queries timed out) for certain configurations due to high server load. We therefore leave out the values for the configurations where it became unresponsive. Unless otherwise specified, we present results for `watdiv10M` and 64 clients since this was the configuration with the largest number of concurrent clients for which all approaches completed.

Performance under Load

The main contribution of SPF is improved query processing performance, even under high querying load. This is due to an expected decrease in network traffic, since SPF should send fewer requests to the server, and the server should return fewer intermediate results, reducing the overhead from the network traffic. While the server load is expected to increase slightly, the reduction in network traffic is expected to improve performance for queries that include star-shaped subqueries.

Figure 5 shows the throughput and number of timeouts of the four approaches for different numbers of concurrent clients for `union` over each WatDiv dataset (`watdiv10M`, `watdiv100M`, `watdiv1000M`). The throughput and number of timeouts for the different query loads over each dataset are shown in Appendix C.1. Although the throughput (Figure 5a, 5c, and 5e) of all the interfaces deteriorates as the number of concurrent clients increases, SPF maintains 4-7 times higher throughput compared to brTPF for the

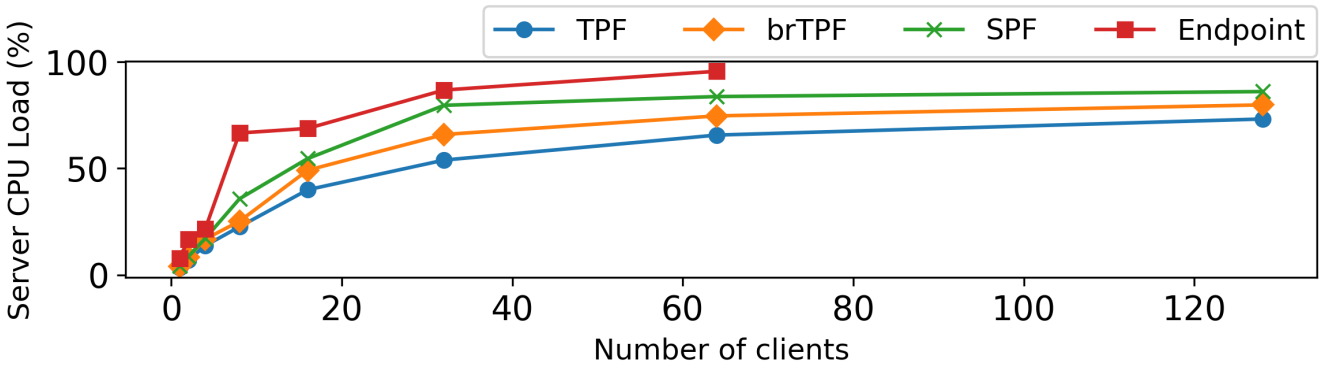


Figure 6: CPU load for the `union` query load and each configuration over `watdiv10M`.

WatDiv datasets in the most challenging case with 128 clients. Even if the SPF server computes the star patterns, Figure 6 shows that SPF only incurs 1.08 times as much CPU load as brTPF for the same configuration. Moreover, due to more efficient query processing, SPF has significantly fewer timeouts than all other approaches (Figure 5b, 5d, and 5f). Even for the largest dataset size, SPF experiences almost 3 times less timeouts than brTPF. This shows, that SPF is not only able to process queries faster, but is also able to process queries that brTPF and TPF can not process within the timeout. The endpoint is the best performing interface for only few concurrent clients. However, its performance deteriorates much faster when the number of concurrent clients increases. Furthermore, according to our experiments, the endpoint is not able to process queries over large datasets when a large number of clients issue queries concurrently. SPF, TPF and brTPF are able to handle the increased load more efficiently than the endpoint (Figures 5a and 5c). This is in line with the experiments shown in [38], and shows that SPF seems to be a suitable alternative to handling large query loads.

Figure 6 shows the server CPU load of each approach for the `union` query load and all the configurations over `watdiv10M`. The CPU load for all datasets are shown in Appendix C.2 and show the same tendencies as Figure 6. Clearly, the endpoint has the highest CPU load throughout our experiments. SPF has a slightly higher CPU load than brTPF and TPF; however, since SPF maintains a higher throughput in our experiments (Figure 5), this is not significant enough to affect availability.

Overall, our experiments seem to confirm our hypothesis that SPF increases query throughput while maintaining relatively low server load even in the presence of high querying load. The performance results, and the fact that the SPF server was able to successfully process queries issued by 128 clients concurrently, show that SPF is able to maintain high availability of the server while also increasing the query processing performance significantly.

Network Traffic

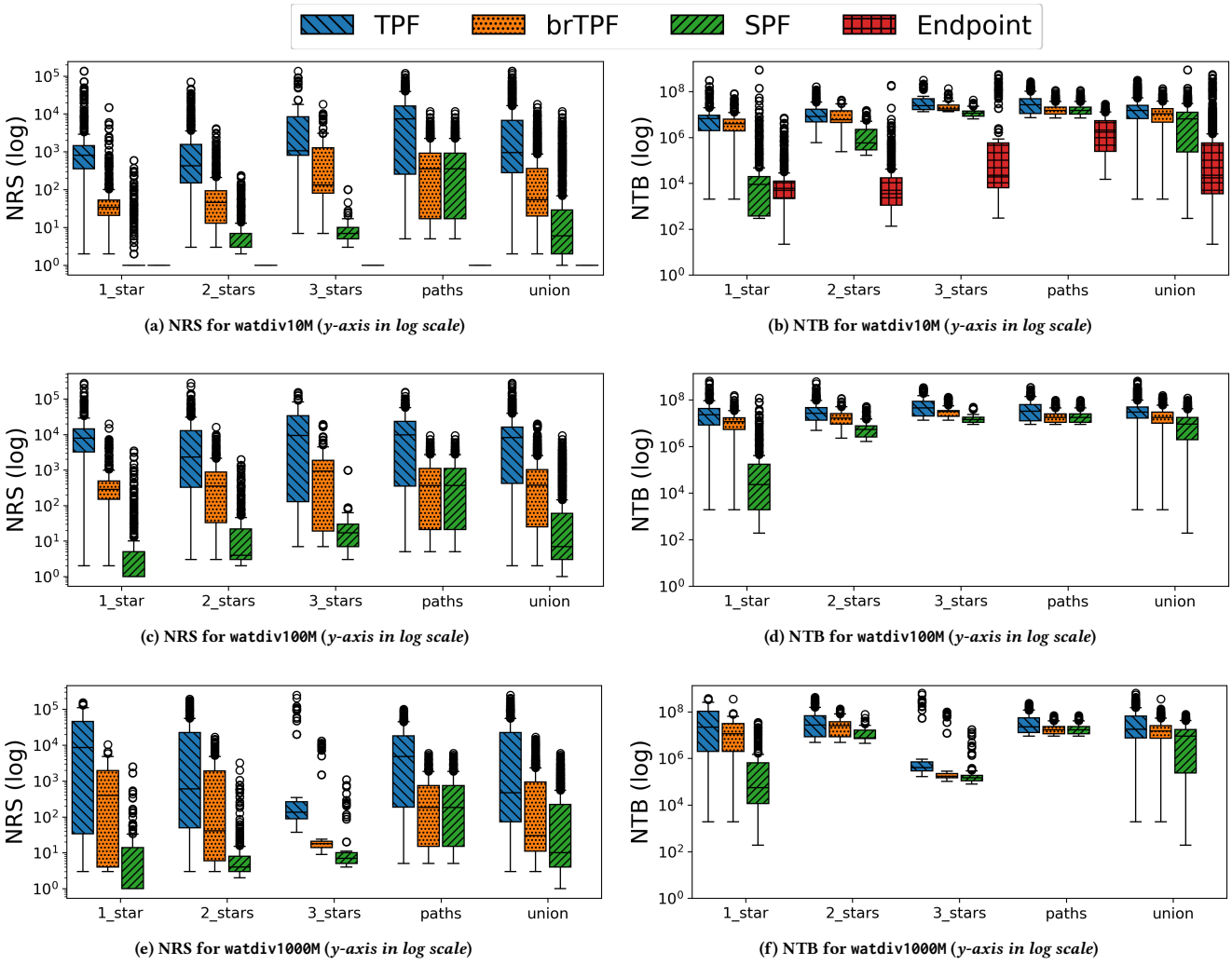
As previously highlighted, we want to assess whether sending more selective requests, i.e. subqueries that may be composed of more than one triple pattern, has an impact on the network traffic. Especially for queries with large star patterns, we expect that utilizing such subqueries results in fewer server requests and less data transfer (i.e., intermediate results) between the server and client. Figure 7 shows the network usage in the configuration with 64 concurrent clients. We include the results for all other configurations in Appendix C.3.

Figures 7a, 7c, and 7e show the number of requests to the server (NRS) for the experiments with 64 clients over each dataset for all approaches. Clearly, SPF sends significantly fewer requests to the server than both brTPF and TPF. This is due to the fact that in order to process a triple pattern, TPF sends one request for each intermediate binding while brTPF sends one request per 30 intermediate bindings (since $|\Omega| = 30$). SPF, however, sends considerably fewer requests since the intermediate results for the triple patterns within a star pattern are processed by the server. The lower network usage is most significant for the `1-star` query load, since SPF only sends one request per 100 results (cf. the page size of 100) and avoids intermediate results altogether. As the queries include more star patterns, SPF sends more requests, although at all times fewer than brTPF and TPF. SPF sends the same amount of requests to the server as brTPF for the `paths` query load as SPF's query processing is the same as brTPF when no stars are included in the query.

Similarly, since the SPF server processes larger parts of the queries, fewer of the intermediate results are sent back to the clients, resulting in a lower number of transferred bytes (NTB) (Figures 7b, 7d, and 7f). Similar to NRS, NTB is significantly lower for SPF in comparison to both TPF and brTPF throughout all query loads except `paths`, where the results are similar for SPF and brTPF. This shows that compared to TPF and brTPF, SPF signicantly reduces the network traffic. Naturally, the endpoint has the lowest NTB and NRS since only one request per query is sent to the server and only the final results are transferred back to the client.

Scalability and Applicability

Figure 5 shows that SPF is able to maintain a higher throughput than the state of the art interfaces, even as the dataset size increases. In fact, even for the largest dataset (`watdiv1000M`), SPF has 4 times higher throughput than brTPF for 128 concurrent clients. Even if the gain in throughput decreases slightly as the size of the dataset



(a) NRS for `watdiv10M` (*y-axis in log scale*)

(b) NTB for `watdiv10M` (*y-axis in log scale*)

(c) NRS for `watdiv100M` (*y-axis in log scale*)

(d) NTB for `watdiv100M` (*y-axis in log scale*)

(e) NRS for `watdiv1000M` (*y-axis in log scale*)

(f) NTB for `watdiv1000M` (*y-axis in log scale*)

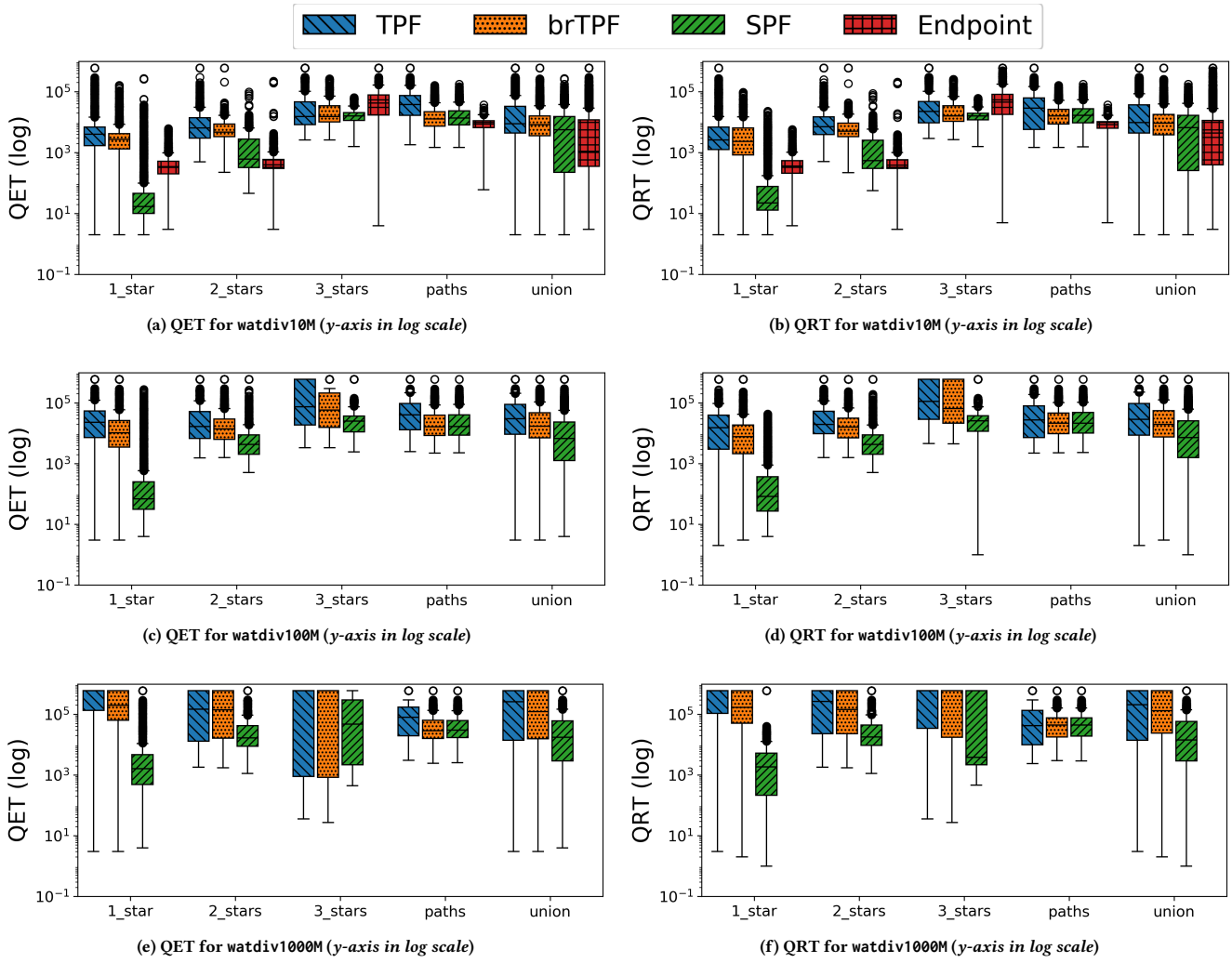
Figure 7: Network traffic for 64 clients for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.

increases, SPF still experiences almost 3 times less timeouts than brTPF for the largest dataset; this number is similar for `watdiv100M`, and SPF does not experience any timeouts for `watdiv10M`. This is due to the increased sizes of intermediate results (Figure 7) that TPF and brTPF have to deal with and the further limited amount of intermediate results of the server-side star join that reduces network traffic and only SPF can benefit from. In fact, for `watdiv1000M`, SPF still has significantly less server requests (Figure 7e) and data transfer (Figure 7f) compared to brTPF and TPF.

Our experiments suggest that SPF is able to achieve high availability of large datasets under high querying load while being able to process queries faster than state of the art LDF interfaces. While the gain in performance that SPF provides decreases slightly for the largest dataset, SPF is still significantly faster and has fewer timeouts compared to both brTPF and TPF. This shows that SPF is able to process more queries under high query load, and that SPF is able to scale to large datasets and still provide the benefits of improved performance and lowered network usage.

Impact of the Query Load

Figure 8 shows the performance for each query load in the configuration with 64 concurrent clients. We include results for all other configurations in Appendix C.4. Figures 8a, 8c, and 8e show query execution time (QET) for all five query loads in the configuration with 64 concurrent clients over the different datasets. For queries with star patterns, it is clear that SPF has better performance than both TPF and brTPF. The difference between SPF and other interfaces is quite significant for the `1-star` query load. This is expected since fewer requests are made for these queries. In fact, as previously described, some queries in the `1-star` query load can be answered with just a single call to the server. As shown in Figure 8, SPF outperforms other interfaces more significantly for the `1-star` and `2-stars` query loads. These two query loads have larger star patterns than the other query loads (Figure 4c) and therefore TPF and brTPF have to make significantly more requests to the server for these queries, whereas SPF still only makes one request to the



(a) QET for `watdiv10M` (*y-axis in log scale*)

(b) QRT for `watdiv10M` (*y-axis in log scale*)

(c) QET for `watdiv100M` (*y-axis in log scale*)

(d) QRT for `watdiv100M` (*y-axis in log scale*)

(e) QET for `watdiv1000M` (*y-axis in log scale*)

(f) QRT for `watdiv1000M` (*y-axis in log scale*)

Figure 8: Query execution time (in ms) and query response time (in ms) of each query load and the configuration with 64 clients over `watdiv10M`, `watdiv100M`, and `watdiv1000M`.

server per 100 bindings to each star pattern (cf. the page size was set to 100).

For queries with no star patterns, we only expected to show that SPF does not have a worse performance than brTPF. This is in line with our experimental results, as SPF has similar performance as brTPF for the `paths` query load. Figure 8a shows that SPF is quite comparable to the endpoint in performance; however, Figure 5 illustrates that the endpoint does not scale as well as SPF when the size of the dataset or number of concurrent clients increases.

Response time

Figures 8b, 8d, and 8f show response times (QRT) for all query loads for the configuration with 64 concurrent clients over each dataset. These experimental results show that all approaches have response times quite similar to execution times. They all receive their first result only slightly earlier than obtaining the full result. For TPF, brTPF, and SPF this is most likely due to the fact that most of the query is already processed upon receiving the first result. For the endpoint, QRT and QET are the same since it processes the entire query on the server before returning the result.

Like QET, the improvement in QRT is more significant for queries with fewer star patterns since fewer calls to the server are needed. Moreover, SPF and brTPF have quite similar QRT for the `paths` query load, as expected. Last, Figure 8b shows that SPF is comparable to the endpoint in terms of response time for a small dataset, although SPF scales better (Figure 5).

6.3 Summary

Overall, our experimental evaluation shows that SPF achieves a novel, and in most cases better, tradeoff between performance and server load than TPF, brTPF and a SPARQL endpoint. SPF does this by significantly reducing the network traffic without incurring too much extra load on the server. Our experiments show that SPF is able to increase the query throughput by up to an order of

magnitude compared to brTPF. Even for the largest dataset, SPF is able to achieve 4 times higher throughput than brTPF and 3 times less timeouts under high query load. For queries without star patterns, SPF still performs as good as brTPF, both in terms of the execution time and the network traffic. While SPF has slightly higher CPU load on the server (1.08 times higher than brTPF), it is still significantly more efficient than TPF and brTPF in the presence of high load. This confirms that SPF is able to combine a lower network load with a higher query throughput and a comparatively low server load.

7 CONCLUSIONS

In this paper, we presented Star Pattern Fragments (SPF), a new RDF interface that exploits a different trade-off for distribution of the workload between the server and client. The SPF client processes queries by processing SPARQL operators and decomposing each BGP into star-shaped subqueries and sends these subqueries, along with intermediate bindings, to the server. This is similar to other state of the art approaches that process only individual triple patterns on the server. We implemented an SPF server that is able to answer HTTP requests containing star patterns as well as an SPF client that is able to answer SPARQL queries. Our experimental results show that SPF reduces the network traffic, both in terms of the number of requests to the server and the amount of transferred data between the client and server, while it increases the query throughput with up to an order of magnitude compared to brTPF. Our evaluation also demonstrates that SPF increases the overall performance while only incurring 1.08 times more CPU load on the server than brTPF and 1.18 times more than TPF when 128 clients issue queries concurrently.

Future Work

SPF provides new insight into the distribution of the query processing workload between the client and the server. As part of the future research directions for SPF, it could be interesting to investigate the tradeoffs between the benefits of including support for object-based star patterns and the cost of the more complex query decomposition strategies on the client (and the overhead of such strategies) that supporting object-based star patterns would entail. Furthermore, decomposing queries by first ensuring the optimal join order on the triple pattern level could increase performance for an SPF client. Therefore, such a decomposition strategy that takes into account the join order of the triple patterns is part of our future work. Moreover, our future work includes assessing the impact of adding an SPF-specific cache to the server. Lastly, it could be interesting to integrate SPF into P2P systems, such as [3, 26], in order to provide a better distribution of the query processing tasks between nodes to speed up query processing for such a system.

REFERENCES

[1] Maribel Acosta and Maria-Esther Vidal. 2015. Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data. In *ISWC 2015*. 111–127. https://doi.org/10.1007/978-3-319-25007-6_7

[2] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2019. A Decentralized Architecture for Sharing and Querying Semantic Data. In *ESWC 2019*. 3–18. https://doi.org/10.1007/978-3-030-21348-0_1

[3] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2019. Decentralized Indexing over a Network of RDF Peers. In *The Semantic Web - ISWC 2019*. 3–20. https://doi.org/10.1007/978-3-030-30793-6_1

[4] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *ISWC 2014*. 197–212. https://doi.org/10.1007/978-3-319-11964-9_13

[5] Carlos Buil Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. 2013. SPARQL Web-Querying Infrastructure: Ready for Action?. In *ISWC 2013*. 277–293. https://doi.org/10.1007/978-3-642-41338-4_18

[6] Amr Azzam, Javier D. Fernández, Maribel Acosta, Martin Beno, and Axel Polleres. 2020. SMART-KG: Hybrid Shipping for SPARQL Querying on the Web. (2020), 984–994. https://doi.org/10.1145/3366423.3380177

[7] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970). https://doi.org/10.1145/362686.362692

[8] Min Cai and Martin R. Frank. 2004. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW*. https://doi.org/10.1145/988672.988760

[9] Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos. 2015. SemaGrow: optimizing federated SPARQL queries. In *SEMANTICS 2015*. 121–128. https://doi.org/10.1145/2814864.2814886

[10] Arturo Crespo and Hector Garcia-Molina. 2002. Routing Indices For Peer-to-Peer Systems. In *ICDCS*. https://doi.org/10.1109/ICDCS.2002.1022239

[11] DBpedia. 2016. DBpedia version 2016-04. https://wiki.dbpedia.org/develop/datasets/dbpedia-version-2016-04 [Online; accessed October-2018].

[12] Michel Dumontier, Alison Callahan, Jose Cruz-Toledo, Peter Ansell, Vincent Emonet, François Belleau, and Arnaud Droit. 2014. Bio2RDF Release 3: A larger, more connected network of Linked Data for the Life Sciences. In *ISWC 2014 Posters & Demonstrations Track*, Vol. 1272. 401–404.

[13] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF Representation for Publication and Exchange (HDT). *J. Web Semantics* 19 (2013), 22–41. https://doi.org/10.1016/j.websem.2013.01.002

[14] Olaf Görlitz and Steffen Staab. 2011. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *(COLD2011)*.

[15] Olaf Hartig and Carlos Buil Aranda. 2016. brTPF: Bindings-Restricted Triple Pattern Fragments (Extended Preprint). *CoRR* abs/1608.08148 (2016).

[16] Lars Heling, Maribel Acosta, Maria Maleshkova, and York Sure-Vetter. 2018. Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study. In *ISWC 2018*. 86–102. https://doi.org/10.1007/978-3-030-00668-6_6

[17] James A. Hendler, Jeanne Holm, Chris Musialek, and George Thomas. 2012. US Government Linked Open Data: Semantic.data.gov. *IEEE Intell. Syst.* 27, 3 (2012), 25–31. https://doi.org/10.1109/MIS.2012.27

[18] Joachim Van Herwegen, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. [n.d.]. Query Execution Optimization for Clients of Triple Pattern Fragments. In *ESWC 2015*. 302–318. https://doi.org/10.1007/978-3-319-18818-8_19

[19] Anders Langballe Jakobsen, Gabriela Montoya, and Katja Hose. 2019. How Diverse Are Federated Query Execution Plans Really?. In *ESWC 2019 Satellite Events*. 105–110. https://doi.org/10.1007/978-3-030-32327-1_21

[20] Mark C. Jeffrey and J. Gregory Steffan. 2011. Understanding Bloom Filter Intersection for Lazy Address-set Disambiguation. In *SPAA*. https://doi.org/10.1145/1989493.1989551

[21] Zoi Kaoudi, Manolis Koubarakis, Kostis Kyzirakos, Iris Miliaraki, Matoula Magiridou, and Antonios Papadakis-Pesaresi. 2010. Atlas: Storing, updating and querying RDF(S) data on top of DHTs. *J. Web Sem.* 8, 4 (2010), 271–277. https://doi.org/10.1016/j.websem.2010.07.001

[22] Marcel Karnstedt, Kai-Uwe Sattler, Martin Richtarsky, Jessica Müller, Manfred Hauswirth, Roman Schmidt, and Renault John. 2007. UniStore: Querying a DHT-based Universal Storage. In *ICDE*. https://doi.org/10.1109/ICDE.2007.369054

[23] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. A Demonstration of the Solid Platform for Social Web Applications. In *WWW 2016*. 223–226. https://doi.org/10.1145/2872518.2890529

[24] Edgard Marx, Muhammad Saleem, Ioanna Lytra, and Axel-Cyrille Ngonga Ngomo. 2018. A Decentralized Architecture for SPARQL Query Processing and RDF Sharing: A Position Paper. In *ICSC 2018*. 274–277. https://doi.org/10.1109/ICSC.2018.00049

[25] Thomas Minier, Hala Skaf-Molli, and Pascal Molli. 2019. SaGe: Web Preemption for Public SPARQL Query Services. In *WWW*. 1268–1278. https://doi.org/10.1145/3308558.3313652

[26] Gabriela Montoya, Christian Aebeloe, and Katja Hose. 2018. Towards Efficient Query Processing over Heterogeneous RDF Interfaces. In *DeSemWeb@ISWC 2018*.

[27] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. 2017. The Odyssey Approach for Optimizing Federated SPARQL Queries. In *ISWC 2017*. 471–489. https://doi.org/10.1007/978-3-319-68288-4_28

[28] Gabriela Montoya, Maria-Esther Vidal, and Maribel Acosta. 2012. A Heuristic-Based Approach for Planning Federated SPARQL Queries. In *COLD 2012*.

[29] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*. 984–994. https://doi.org/10.1109/ICDE.2011.5767868

[30] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45. https://doi.org/10.1007/11926078_3
[31] Axel Polleres, Maulik R. Kamdar, Javier D. Fernández, Tania Tudorache, and Mark A. Musen. 2018. A More Decentralized Vision for Linked Data. In *DeSemWeb@ISWC 2018*.
[32] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. 2014. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In *ESWC 2014*. 176–191. https://doi.org/10.1007/978-3-319-07443-6_13
[33] Muhammad Saleem, Alexander Potocki, Tommaso Soru, Olaf Hartig, and Axel-Cyrille Ngonga Ngomo. 2018. CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation. In *SEMANTICS 2018*. 163–174. https://doi.org/10.1016/j.procs.2018.09.016
[34] Muhammad Saleem, Gábor Szárnyas, Felix Conrads, Syed Ahmad Chan Bukhari, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2019. How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks. In *WWW 2019*. 1623–1633. https://doi.org/10.1145/3308558.3313556
[35] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. FedX: A Federation Layer for Distributed Query Processing on Linked Open Data. In *ESWC 2011*. 481–486. https://doi.org/10.1007/978-3-642-21064-8_39
[36] Claus Stadler, Jens Lehmann, Konrad Höffner, and Sören Auer. 2012. LinkedGeoData: A core for a web of spatial open data. *Semantic Web* 3, 4 (2012), 333–354. https://doi.org/10.3233/SW-2011-0052
[37] Pierre-Yves Vandenbussche, Jürgen Umbrich, Luca Matteis, Aidan Hogan, and Carlos Buil Aranda. 2017. SPARQLES: Monitoring public SPARQL endpoints. *Semantic Web* 8, 6 (2017), 1049–1065. https://doi.org/10.3233/SW-170254
[38] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *J. Web Sem.* 37-38 (2016), 184–206. https://doi.org/10.1016/j.websem.2016.03.003
[39] Maria-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. [n.d.]. Efficiently Joining Group Patterns in SPARQL Queries. In *ESWC 2010*. 228–242. https://doi.org/10.1007/978-3-642-13486-9_16
[40] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. https://doi.org/10.1145/2629489

A EXTENDED RELATED WORK

Recent studies [2, 23, 24, 31, 38] have proposed decentralization to improve the availability of such datasets. In this section, we discuss different decentralized approaches, how they attempt to solve the availability problem, and their shortcomings. Specifically, we discuss Linked Data Fragments (LDF) interfaces [38], Peer-to-Peer (P2P) systems, federated query engines, and other decentralized systems.

A.1 Linked Data Fragments

LDF interfaces [38] lift the burden on the data providers by shifting some of the bulk of the query processing burden from the server to the clients such that the server only has to process small subqueries. By doing so, LDF interfaces are generally able to remain efficient during high query load.

Triple Pattern Fragments (TPF) [38], for instance, only process individual triple patterns on the server while expensive joins and SPARQL operators like `UNION` and `FILTER` are processed by the clients. TPF servers therefore have a lower processing burden than SPARQL endpoints, making them more reliable under load. Since a TPF server only processes individual triple patterns, the clients can use various approaches to process the queries. Verborgh et al. [38] proposed a greedy algorithm based on a pipeline of iterators (i.e., left-linear join order) in which the order of the triple patterns are determined based on their estimated cardinality. While TPF reduces the load on the server in general, it puts much more load on the client and incurs more network traffic. To improve query processing performance over TPF servers, Acosta et al. [1] proposed to use an adaptive query processing approach that routes each intermediate triple through the query operators in a variety of orders to simulate different execution plans. On the other hand, Herwegen et al. [18] attempted to predict the optimal join order based on a combination of all the metadata made available by the server and the intermediate results.

While both approaches for optimizing query processing over TPF clients described above were able to significantly reduce the number of requests made to the server, they still incur in relatively high network usage. Furthermore, Heling et al. [16] found that the performance of TPF is affected by variables such as the triple pattern type[10] and the cardinality. Bindings-Restricted TPF (brTPF) [15] was proposed to reduce the network traffic by coupling triple patterns and bindings obtained from previously evaluated triple patterns. Despite improving the availability of RDF data, all these approaches cause a large number of calls to the server. On the other hand, hybridSE [26] relies on both SPARQL endpoints and brTPF servers to process queries more efficiently than the TPF-based interfaces. SPARQL subqueries with a large number of intermediate results are evaluated using SPARQL endpoints to overcome limitations of TPF clients. However, since hybridSE may send complex subqueries to the endpoint, and endpoints have downtime [5], this leaves the approach vulnerable to downtime.

All approaches mentioned above attempt to increase the server availability while providing efficient access to knowledge graphs. Star Pattern Fragments (SPF), as proposed in this paper, significantly improves performance for LDF interfaces by processing larger conjunctive subqueries on the server thus incurring significantly less network traffic while keeping the server load comparatively low.

A.2 Peer-to-Peer Systems

P2P systems [2, 8, 21, 22] attempt to increase the availability of knowledge graphs by removing the central server altogether. Instead, each node acts like both a client and server and contains a limited local datastore. Replication of datasets across multiple participating nodes can ensure that the data will remain available even if the original node fails. P2P systems can be divided into two separate types: structured and unstructured P2P networks.

Structured P2P systems [8, 21, 22] apply a specific structured overlay over the entire network. For instance, UniStore [22] and Atlas [21] use Dynamic Hash Tables (DHTs) to impose on which nodes to store which data. This can benefit query processing performance since nodes only have to perform a DHT lookup to determine which node carries the relevant information. Moreover, determining which nodes to replicate new data on requires only a lookup into the hash table. RDFPeers [8] instead imposes a ring structure to the network, i.e., nodes are connected in a ring based on their hash values. Triple patterns are then placed in the ring based on their hash value. This structure has the advantages that the number of routing hops for insertions and processing most queries is logarithmic to the number of nodes within the network. However, structured P2P systems usually require some level of global knowledge over the network. In situations where nodes frequently leave or join the network (high churn), obtaining such global knowledge

[10] The type of a triple pattern is defined with respect to the position of variables in the triple pattern.

can be impossible. Moreover, the overlay has to recalculated and data redistributed each time a node leaves or joins the network.

Unstructured P2P systems therefore impose no such overlay. Piqnic [2], for instance, defines an architecture where connections between nodes are random. This makes the network more robust under churn; however, it has to rely on flooding to process queries. That is, a node sends a request to each of its immediate neighbors which forward the request to their immediate neighbors, and so on. This creates an excessive amount of requests at query processing time. Though, decentralized indexes [3, 10] have removed the need for flooding, such networks still incur somewhat high network usage. Routing indexes [10] allow nodes to determine which of their neighbors, or combination of neighbors, can provide the most complete result to a query. However, this still results in at least one chain of requests being formed per request, limiting the effect of the index. PPBF indexes [3] instead estimate exactly on which nodes relevant data to a subquery is located at. This is done by summarizing the constituents of data fragments using Bloom Filters [7] and using Bloom Filter intersections [20] to filter out nodes with non relevant data.

In any case, the query processing effort in P2P systems is usually distributed across participating nodes and otherwise follows quite similar to processing queries over LDF servers; queries are processed locally and subqueries are sent to other nodes in the network for evaluation rather than a centralized server. However, differently from traditional SPF servers, the set of nodes over which to evaluate subqueries in P2P systems are not fixed. Still, extending nodes in a P2P network with Star Pattern Fragments (SPF) could result in a similar reduction of intermediate results as described in Section 6.

A.3 Federated Query Engines

Federated query engines [9, 14, 19, 27, 32, 35] divide SPARQL query processing over multiple SPARQL endpoints. This decreases the overall load on each endpoint significantly. While they ideally only send subqueries to endpoints containing relevant data, thus lowering the overall server load, finding the optimal query execution plan that limits the size of the intermediate results is a difficult problem to solve. They therefore sometimes fail to generate optimal query plans that transfer the minimum amount of data from the endpoints to the engine and therefore increase the load on SPARQL endpoints.

Several federated systems propose different ways to determine a query execution plan. Several approaches [9, 14] use metadata from the federated datasources to provide cost estimations and cardinality estimations of each triple pattern over relevant sources in order to optimize the query execution plan. HiBISCuS [32] proposes to use graph summaries and to determine relevant sources for each part of the query. Montoya et al. [28] instead consider decomposing SPARQL queries into star-shaped groups that can be evaluated by a single SPARQL endpoint. This is similar to the approach presented in this paper since it also utilizes star-shaped decomposition; however, [28] uses star decomposition to process queries over federations of SPARQL endpoints which are subject to failure. Other optimization techniques for federated query engines attempt to estimate the selectivity of joins in queries to produce better execution plans [27, 33].

However, federated query engines typically rely on a fixed set of SPARQL endpoints. As previously mentioned, such endpoints suffer from unavailability [5, 37], and the engines can therefore not guarantee higher availability of knowledge graphs. In any case, the query processing and optimization techniques discussed above could be applied over federations of SPF server to utilize the benefits of SPF highlighted in this paper.

A.4 Other Decentralized Systems

Other systems use more complex techniques to address some of the issues posed by TPF. SaGe [25] uses a preemptive model that suspends queries after a fixed time quantum, so as to not starve simpler queries from system resources, whereafter they can be resumed upon client request. This, however, often leads to a large amount of requests to the server, which is exactly what we are addressing in this paper. Smart-KG [6], on the other hand, ships star-shaped partitions to the client during query processing. This decreases the amount of requests to the server, since partitions already shipped to the client can be evaluated directly on the client. However, this can in some cases lead to unnecessary data transfer during query processing, since the entire partition is shipped regardless of object bindings obtained from previously evaluated triple patterns.

A.5 Summary

Previous studies have proposed different solutions to alleviate the burden from the data providers. LDF interfaces [15, 26, 38], for instance, improve the availability of knowledge graphs by shifting a large part of the query processing effort from the server to the client. P2P systems [2, 8, 21, 22], on the other hand, attempt to remove the central server altogether by making nodes act like both clients and servers, and allowing nodes to communicate with each other. Several other systems either split up the query processing across multiple endpoints [27, 35], ship star-shaped partitions [6] to the client, or suspend queries from being executed after a fixed time [25]. In most of these systems, however, query processing tends to follow a similar approach; queries are executed locally on the clients and small subqueries are processed remotely.

In this paper, we propose an interface that builds on this notion and provides a novel distribution of query processing tasks where, differently from the approaches discussed in this section, joins in star-shaped subqueries are evaluated by the server. Such computations do not significantly increase the query load because star-shaped subqueries can be answered in linear complexity [30]. Our approach therefore achieves a reduction on the data transfer and the execution time without having a negative impact on the data availability. While SPF is as an LDF interface, other systems could be extended with our approach to provide similar benefits as well.

B QUERY PROCESSING EXAMPLE

In this section, we will provide an example of processing a SPARQL query over an SPF server and client. This section shows how the client applies the procedure outlined in Section 5.1, and provides the intuition of how queries are processed with SPF. In this example, $|\Omega| = 30$ and the page size is 100.

Consider the query Q from Listing 1 processed over DBpedia [11]. Processing Q with SPF is done by following the approach outlined in Section 5.1. The first step is to obtain the star decomposition of Q. Figure 3 shows the star decomposition of Q into stars S_1 and S_2.

The second step is to determine the join order, i.e., which of the two star patterns should be processed first. To do this, SPF relies on the cardinality estimations from the server. The client therefore sends a request for the first page of each star pattern. The cardinalities are as follows.

- $cardinality(S_1) = 13$
- $cardinality(S_2) = 71$

Since S_1 has the lowest cardinality and is thus the more selective star pattern, it is evaluated first. However, the first page for S_1 has already been retrieved, and since the 13 intermediate results are fewer than the page size of 100, there is no need to call the server once more. Instead, the 13 bindings are bulked together as Ω_{S_1} and appended to the request for S_2. The client then forwards the request containing S_2 and Ω_{S_1} to the server.

Upon receiving the request for S_2 and Ω_{S_1}, the server continuously replaces variables in S_2 with values according to Ω_{S_1} and appends results of the corresponding 13 star patterns to the output. This happens 13 times since there are 13 bindings in Ω_{S_1}. The server thus responds with the 8 bindings that are the result of joining S_1 and S_2. These 8 bindings are the output result of Q.

C ADDITIONAL EXPERIMENTAL RESULTS

In this section, we show additional experimental results including a brief discussion on the findings. Specifically, we show the throughput and number of timeouts for each individual query load over each dataset. Moreover, we show the CPU load on the server for each datasets. Last, we show the network traffic as well as the performance for all configurations except for 64 clients since they are shown in Figures 7 and 8.

C.1 Performance over the Query Loads

Figure 9 shows the throughput over each dataset for `1-star` (Figures 9a-9c), `2-stars` (Figures 9d-9f), `3-stars` (Figures 9g-9i), and `paths` (Figures 9j-9l). Figure 10 shows the number of timeouts over each dataset for `1-star` (Figures 10a-10c), `2-stars` (Figures 10d-10f), `3-stars` (Figures 10g-10i), and `paths` (Figures 10j-10l).

It is clear from the experimental results in this section that SPF has higher throughput for all query loads with star patterns under high query load and for large-scale datasets. As pointed out in Section 6, this is more significant for the `1-star` and `2-stars` query loads. This is due to a comparatively lower number of intermediate results that SPF has to process. Moreover, for the `1-star` query load, SPF does not process any intermediate results since it only receives the final results. Furthermore, SPF is as good as brTPF in terms of throughput for the `paths` query load that does not contain any star patterns. This shows that SPF performs at least as well as brTPF, and if the queries contain star patterns it performs better.

In terms of timeouts, it is clear that SPF is able to finish more queries compared to TPF, brTPF, and the endpoint; especially for `watdiv10M` where SPF did not experience a single timeout in any query load. SPF has significantly fewer timeouts than TPF, brTPF, and the endpoint for all query loads, datasets, and configurations. The only exception to this is for the `paths` query load, where SPF had as many timeouts as brTPF.

C.2 CPU Load

Figure 11 shows the CPU load over `watdiv10M` (Figure 11a), `watdiv100M` (Figure 11b), and `watdiv1000M` (Figure 11c) for the `union` query load. While SPF has a higher CPU load on the server compared to TPF and brTPF throughout the experiments, it is not enough to significantly affect performance during high load. In fact, as pointed out in Section C.1, SPF maintains a higher throughput even when 128 clients concurrently query the server. For `watdiv100M`, SPF maintains a CPU load on the server that is almost identical to that of brTPF (Figure 11b). These results are consistent with the ones described in Section 6 and show that SPF is able to achieve a better distribution of the load on the server and client than state of the art LDF interfaces.

C.3 Network Traffic

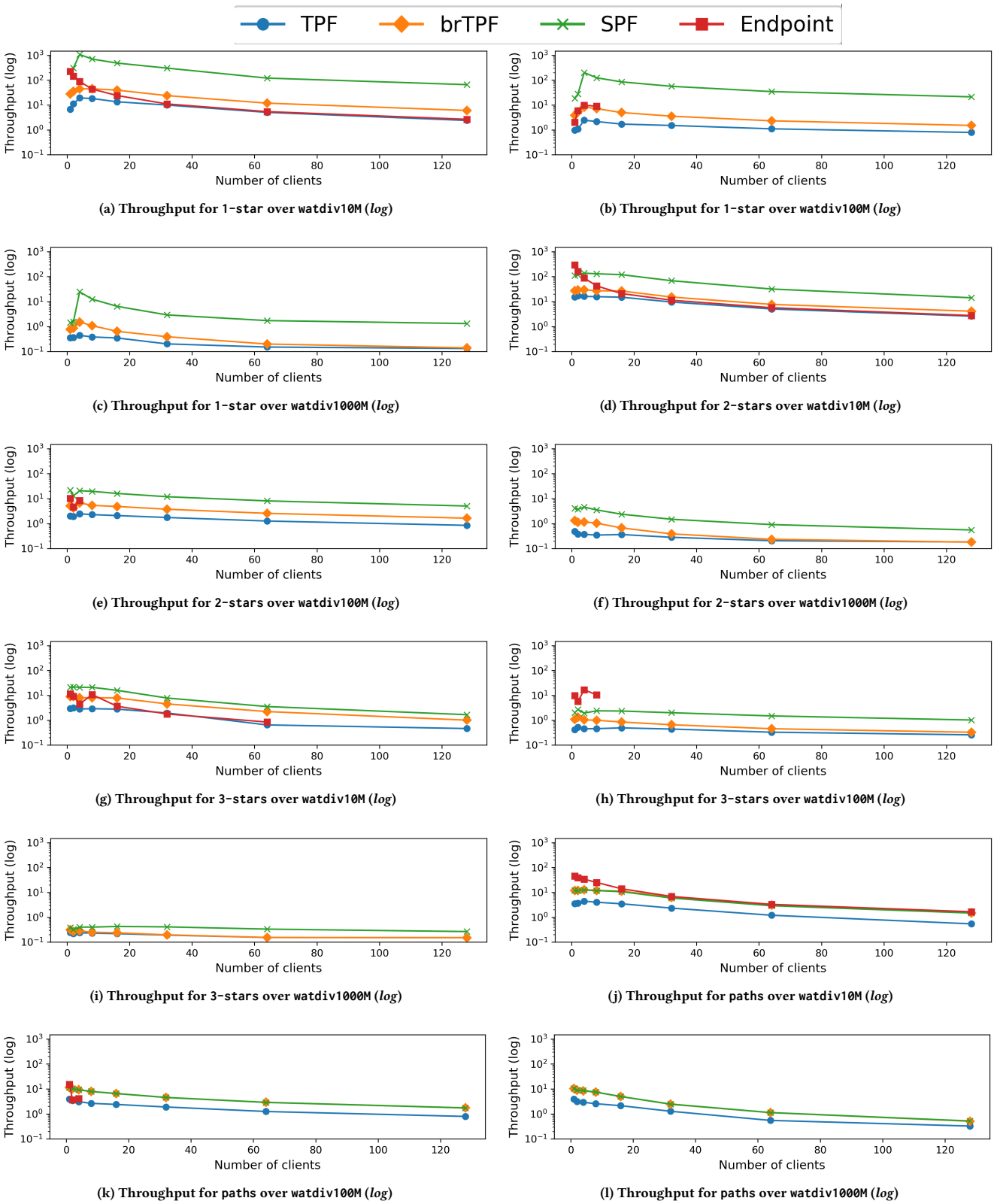
Figures 12-15 show the number of requests to the server (NRS) and the number of transferred bytes (NTB) for all numbers of clients over all datasets. These results are consistent with the results shown in Section 6 and show that SPF sends significantly less requests and transfers significantly less data than both brTPF and TPF. This is most significant for `1-star` since SPF only has to send one request per 100 results (cf. the page size of 100) and avoids intermediate results altogether. As the number of star patterns in the query increases, the difference in NRS and NTB between SPF and brTPF becomes less significant. The only exception to this is for `paths` where SPF still performs as good as brTPF. These results are consistent across all configurations.

C.4 Execution and Response Time

Figures 16-19 show the query execution time (QET) and query response time (QRT) for all numbers of clients over all datasets. These results show that SPF significantly increases performance compared to brTPF and TPF. This is more significant for `1-star` and `2-star` and is due to each individual star being a larger part of the query, thus lowering the amount of intermediate results SPF has to process. In the worst case where queries do not contain any star patterns, i.e., `paths`, SPF still performs as good as brTPF. The results shown in this section are consistent across all configurations and with the results presented in Section 6.

C.5 Summary

Overall, the additional experiments presented in this section are consistent with the ones presented in Section 6. The results show that SPF is able to increase query throughput by up to an order of magnitude compared to brTPF. Even though the load on the server is slightly higher for SPF than for brTPF and TPF, SPF significantly improves the query execution time and query response time for queries with star patterns. This indicates that SPF is a suitable alternative to accessing large-scale knowledge graphs under high query processing load.



(a) Throughput for `1-star` over `watdiv10M` (*log*)

(b) Throughput for `1-star` over `watdiv100M` (*log*)

(c) Throughput for `1-star` over `watdiv1000M` (*log*)

(d) Throughput for `2-stars` over `watdiv10M` (*log*)

(e) Throughput for `2-stars` over `watdiv100M` (*log*)

(f) Throughput for `2-stars` over `watdiv1000M` (*log*)

(g) Throughput for `3-stars` over `watdiv10M` (*log*)

(h) Throughput for `3-stars` over `watdiv100M` (*log*)

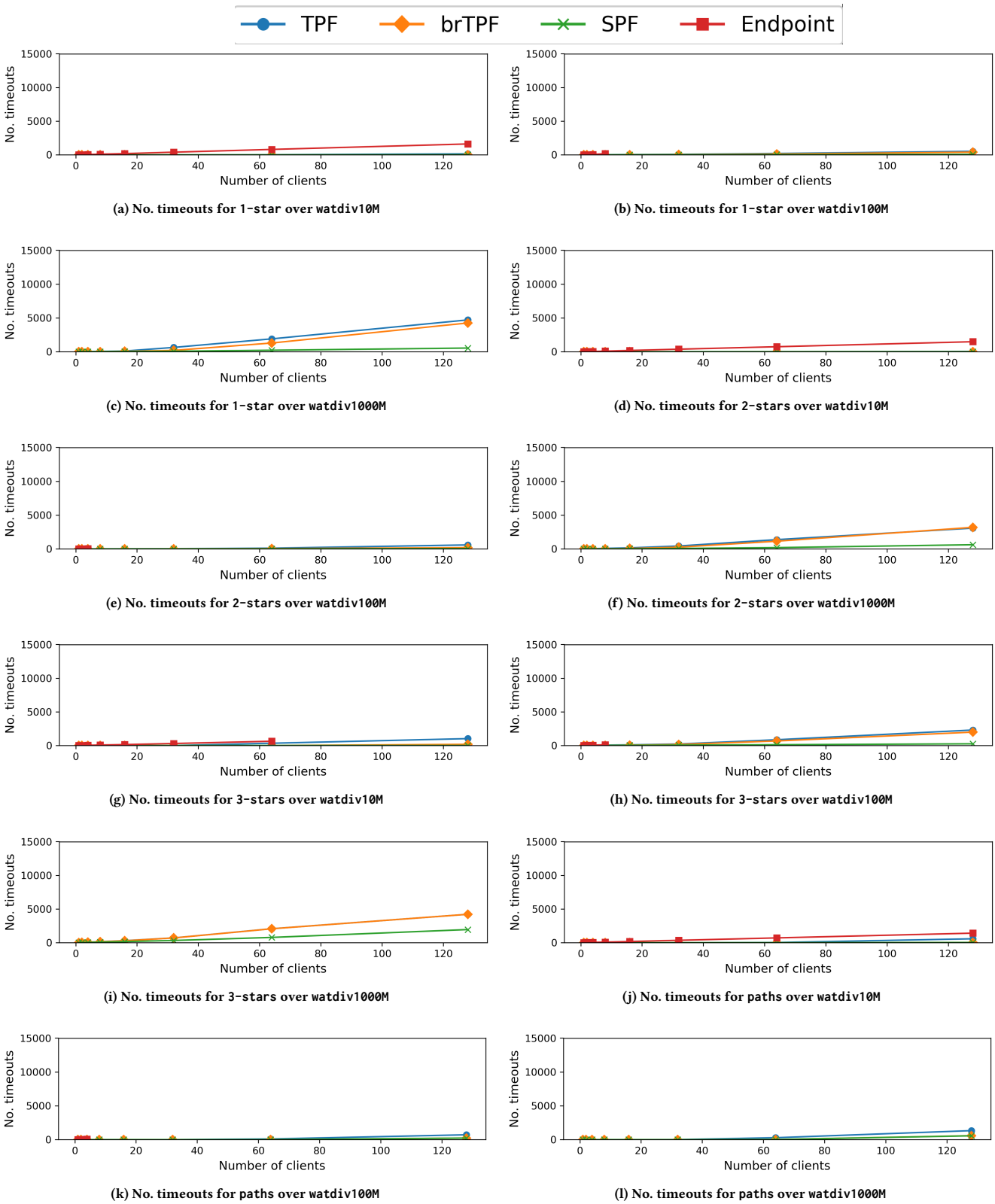
(i) Throughput for `3-stars` over `watdiv1000M` (*log*)

(j) Throughput for `paths` over `watdiv10M` (*log*)

(k) Throughput for `paths` over `watdiv100M` (*log*)

(l) Throughput for `paths` over `watdiv1000M` (*log*)

Figure 9: Throughput (#queries/m) for each query load over the different WatDiv datasets.



(a) No. timeouts for `1-star` over `watdiv10M`

(b) No. timeouts for `1-star` over `watdiv100M`

(c) No. timeouts for `1-star` over `watdiv1000M`

(d) No. timeouts for `2-stars` over `watdiv10M`

(e) No. timeouts for `2-stars` over `watdiv100M`

(f) No. timeouts for `2-stars` over `watdiv1000M`

(g) No. timeouts for `3-stars` over `watdiv10M`

(h) No. timeouts for `3-stars` over `watdiv100M`

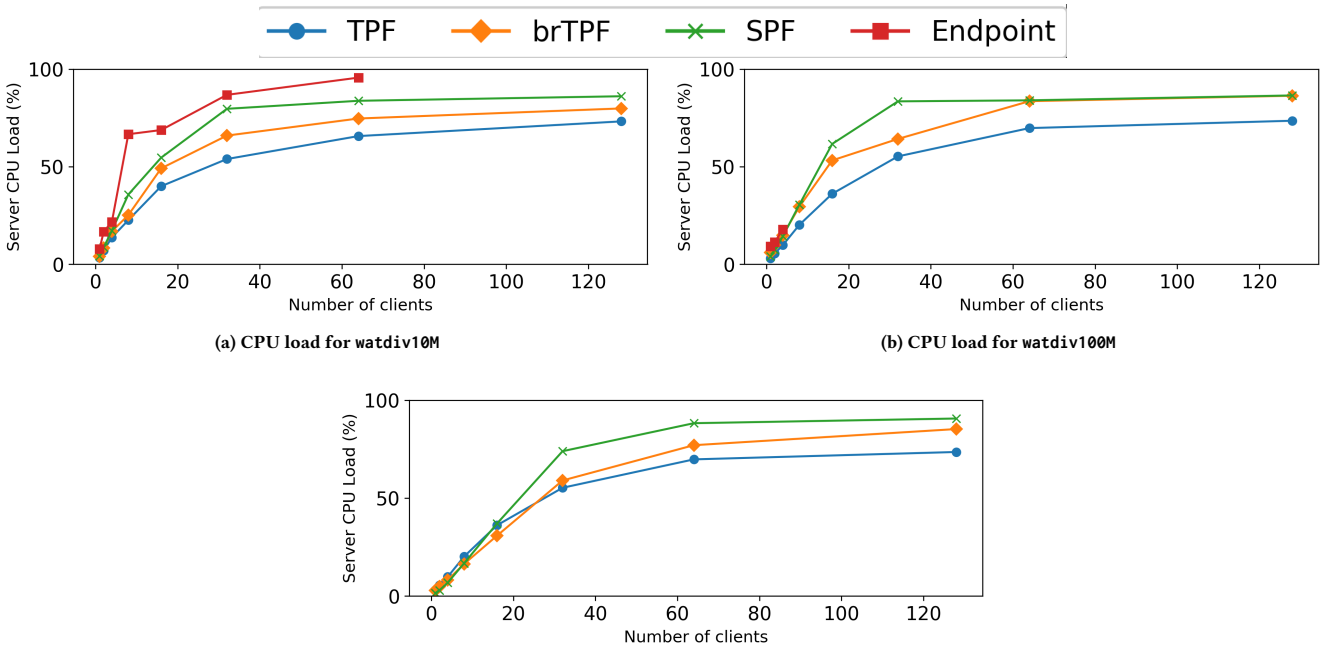
(i) No. timeouts for `3-stars` over `watdiv1000M`

(j) No. timeouts for `paths` over `watdiv10M`

(k) No. timeouts for `paths` over `watdiv100M`

(l) No. timeouts for `paths` over `watdiv1000M`

Figure 10: Number of timeouts for each query load over the different WatDiv datasets.

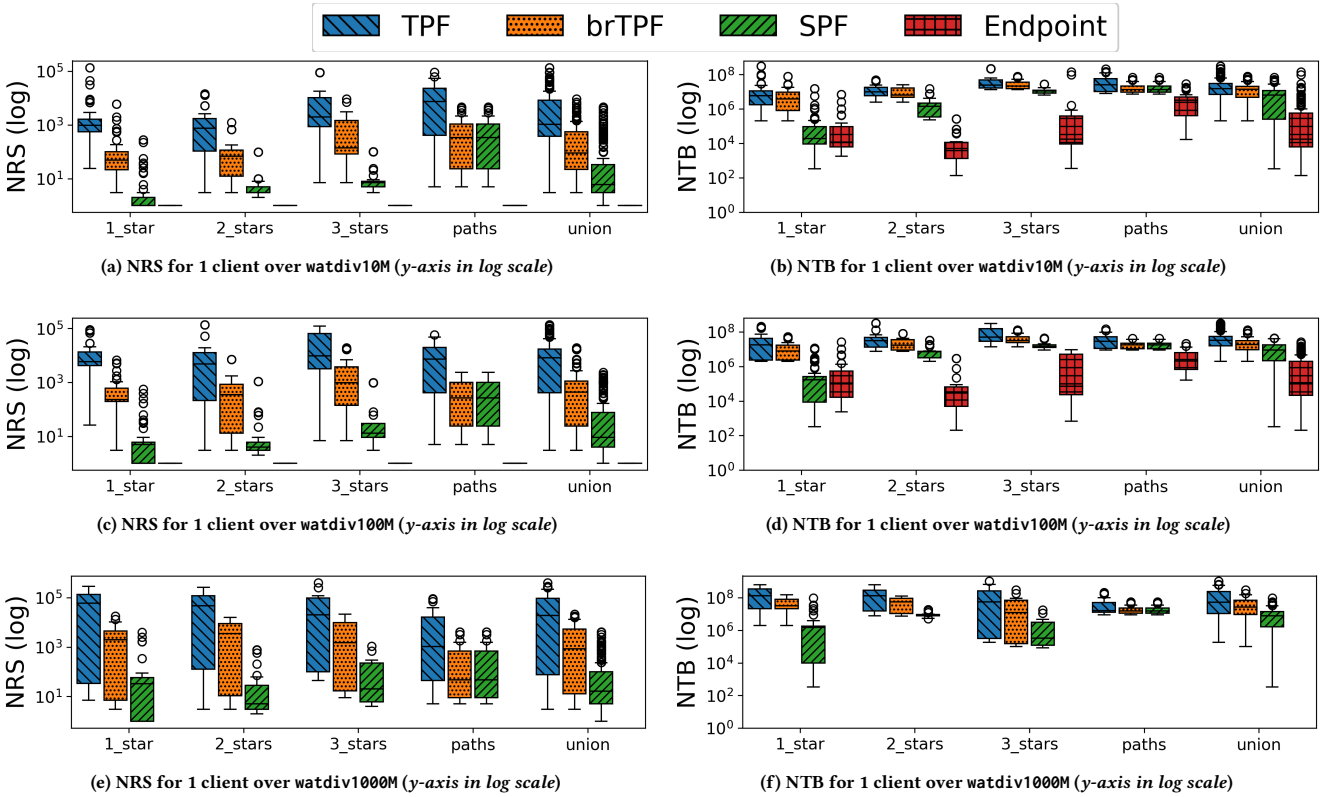


(a) CPU load for `watdiv10M`

(b) CPU load for `watdiv100M`

(c) CPU load for `watdiv1000M`

Figure 11: CPU load for `union` over the different WatDiv datasets.



(a) NRS for 1 client over `watdiv10M` (*y-axis in log scale*)

(b) NTB for 1 client over `watdiv10M` (*y-axis in log scale*)

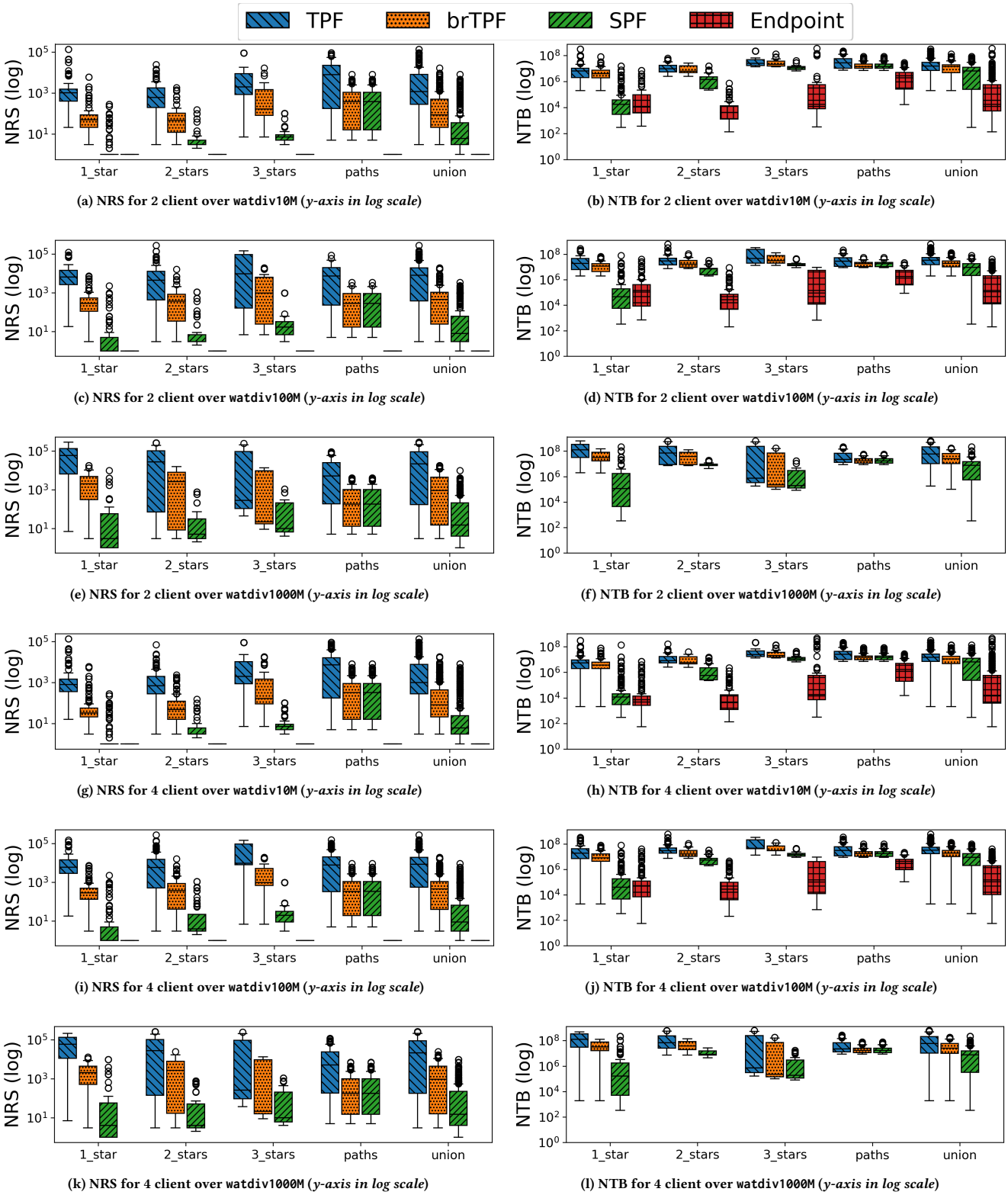
(c) NRS for 1 client over `watdiv100M` (*y-axis in log scale*)

(d) NTB for 1 client over `watdiv100M` (*y-axis in log scale*)

(e) NRS for 1 client over `watdiv1000M` (*y-axis in log scale*)

(f) NTB for 1 client over `watdiv1000M` (*y-axis in log scale*)

Figure 12: Network traffic for 1 client for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.



(a) NRS for 2 client over `watdiv10M` (*y-axis in log scale*)

(b) NTB for 2 client over `watdiv10M` (*y-axis in log scale*)

(c) NRS for 2 client over `watdiv100M` (*y-axis in log scale*)

(d) NTB for 2 client over `watdiv100M` (*y-axis in log scale*)

(e) NRS for 2 client over `watdiv1000M` (*y-axis in log scale*)

(f) NTB for 2 client over `watdiv1000M` (*y-axis in log scale*)

(g) NRS for 4 client over `watdiv10M` (*y-axis in log scale*)

(h) NTB for 4 client over `watdiv10M` (*y-axis in log scale*)

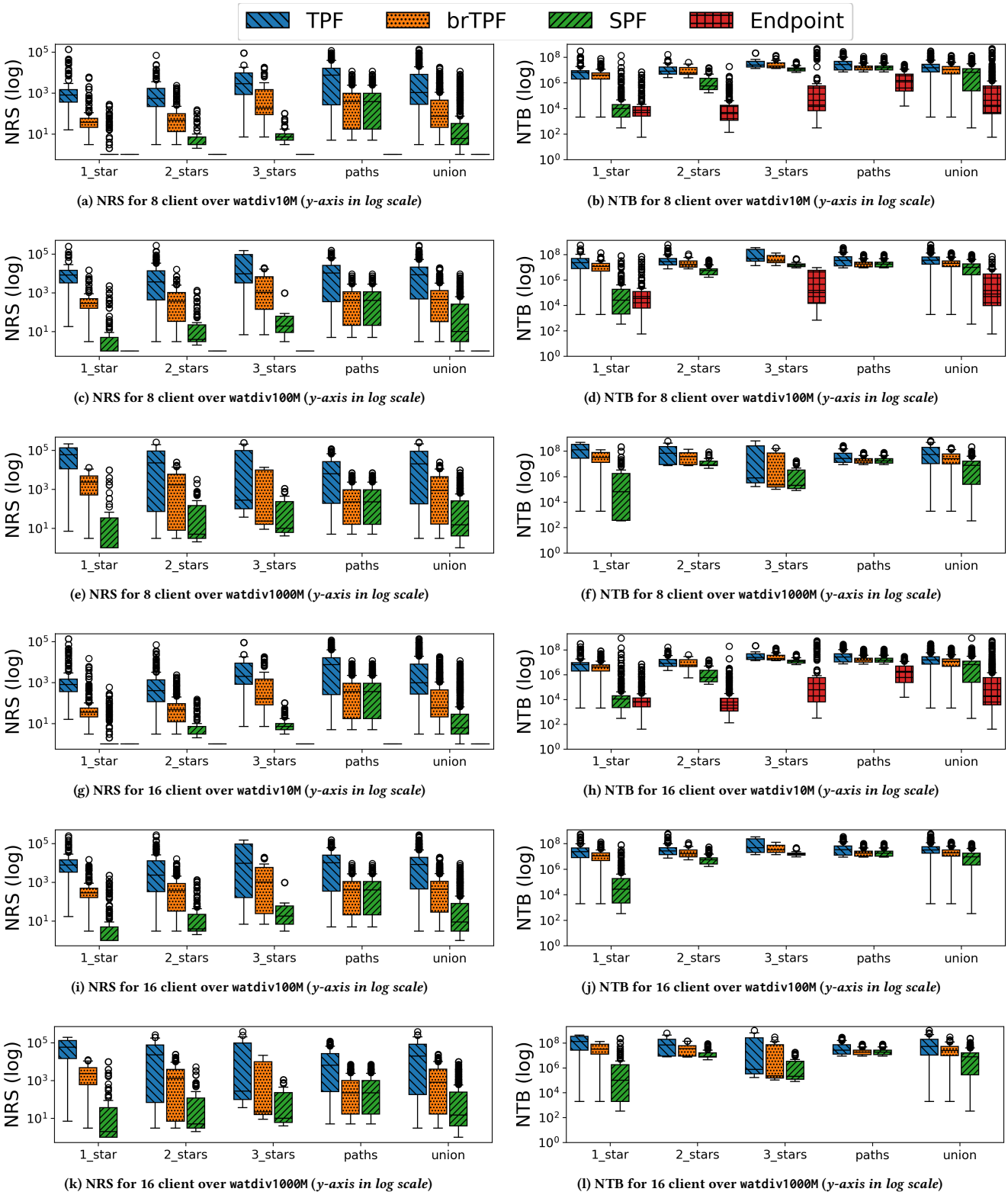
(i) NRS for 4 client over `watdiv100M` (*y-axis in log scale*)

(j) NTB for 4 client over `watdiv100M` (*y-axis in log scale*)

(k) NRS for 4 client over `watdiv1000M` (*y-axis in log scale*)

(l) NTB for 4 client over `watdiv1000M` (*y-axis in log scale*)

Figure 13: Network traffic for 2 and 4 clients for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.



(a) NRS for 8 client over `watdiv10M` (*y-axis in log scale*)

(b) NTB for 8 client over `watdiv10M` (*y-axis in log scale*)

(c) NRS for 8 client over `watdiv100M` (*y-axis in log scale*)

(d) NTB for 8 client over `watdiv100M` (*y-axis in log scale*)

(e) NRS for 8 client over `watdiv1000M` (*y-axis in log scale*)

(f) NTB for 8 client over `watdiv1000M` (*y-axis in log scale*)

(g) NRS for 16 client over `watdiv10M` (*y-axis in log scale*)

(h) NTB for 16 client over `watdiv10M` (*y-axis in log scale*)

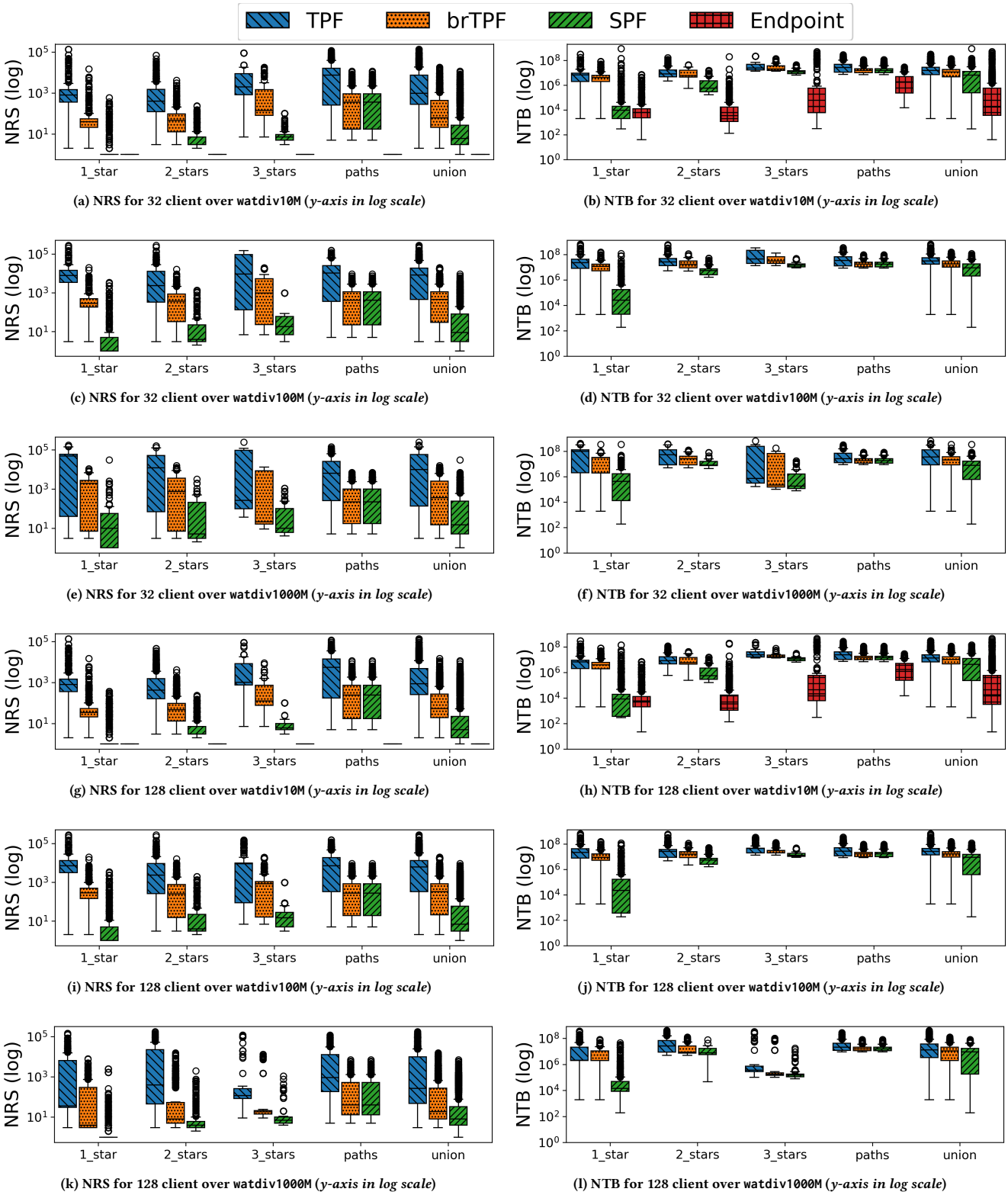
(i) NRS for 16 client over `watdiv100M` (*y-axis in log scale*)

(j) NTB for 16 client over `watdiv100M` (*y-axis in log scale*)

(k) NRS for 16 client over `watdiv1000M` (*y-axis in log scale*)

(l) NTB for 16 client over `watdiv1000M` (*y-axis in log scale*)

Figure 14: Network traffic for 8 and 16 clients for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.



(a) NRS for 32 client over `watdiv10M` (*y-axis in log scale*)

(b) NTB for 32 client over `watdiv10M` (*y-axis in log scale*)

(c) NRS for 32 client over `watdiv100M` (*y-axis in log scale*)

(d) NTB for 32 client over `watdiv100M` (*y-axis in log scale*)

(e) NRS for 32 client over `watdiv1000M` (*y-axis in log scale*)

(f) NTB for 32 client over `watdiv1000M` (*y-axis in log scale*)

(g) NRS for 128 client over `watdiv10M` (*y-axis in log scale*)

(h) NTB for 128 client over `watdiv10M` (*y-axis in log scale*)

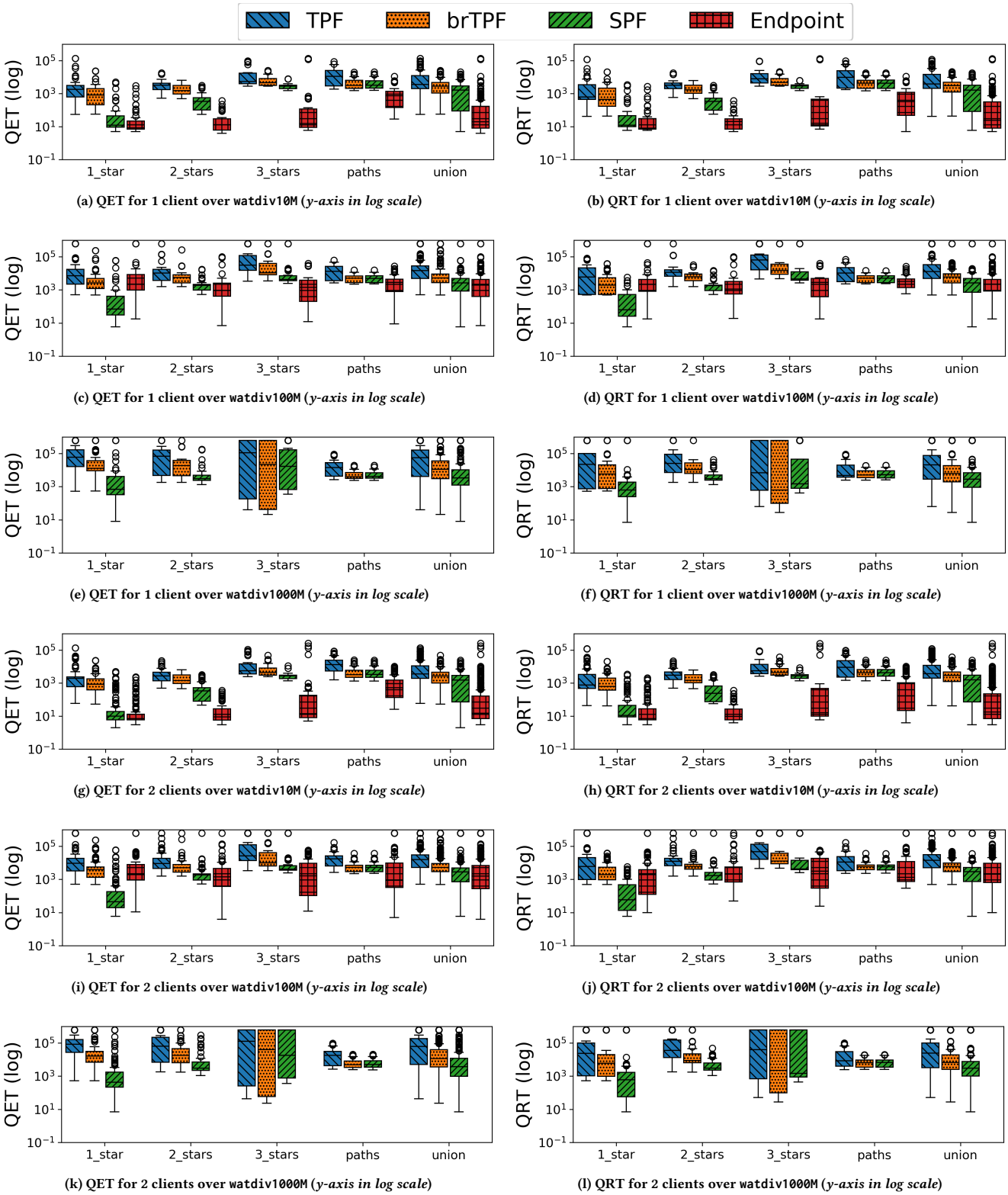
(i) NRS for 128 client over `watdiv100M` (*y-axis in log scale*)

(j) NTB for 128 client over `watdiv100M` (*y-axis in log scale*)

(k) NRS for 128 client over `watdiv1000M` (*y-axis in log scale*)

(l) NTB for 128 client over `watdiv1000M` (*y-axis in log scale*)

Figure 15: Network traffic for 32 and 128 clients for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.



(a) QET for 1 client over `watdiv10M` (*y-axis in log scale*)

(b) QRT for 1 client over `watdiv10M` (*y-axis in log scale*)

(c) QET for 1 client over `watdiv100M` (*y-axis in log scale*)

(d) QRT for 1 client over `watdiv100M` (*y-axis in log scale*)

(e) QET for 1 client over `watdiv1000M` (*y-axis in log scale*)

(f) QRT for 1 client over `watdiv1000M` (*y-axis in log scale*)

(g) QET for 2 clients over `watdiv10M` (*y-axis in log scale*)

(h) QRT for 2 clients over `watdiv10M` (*y-axis in log scale*)

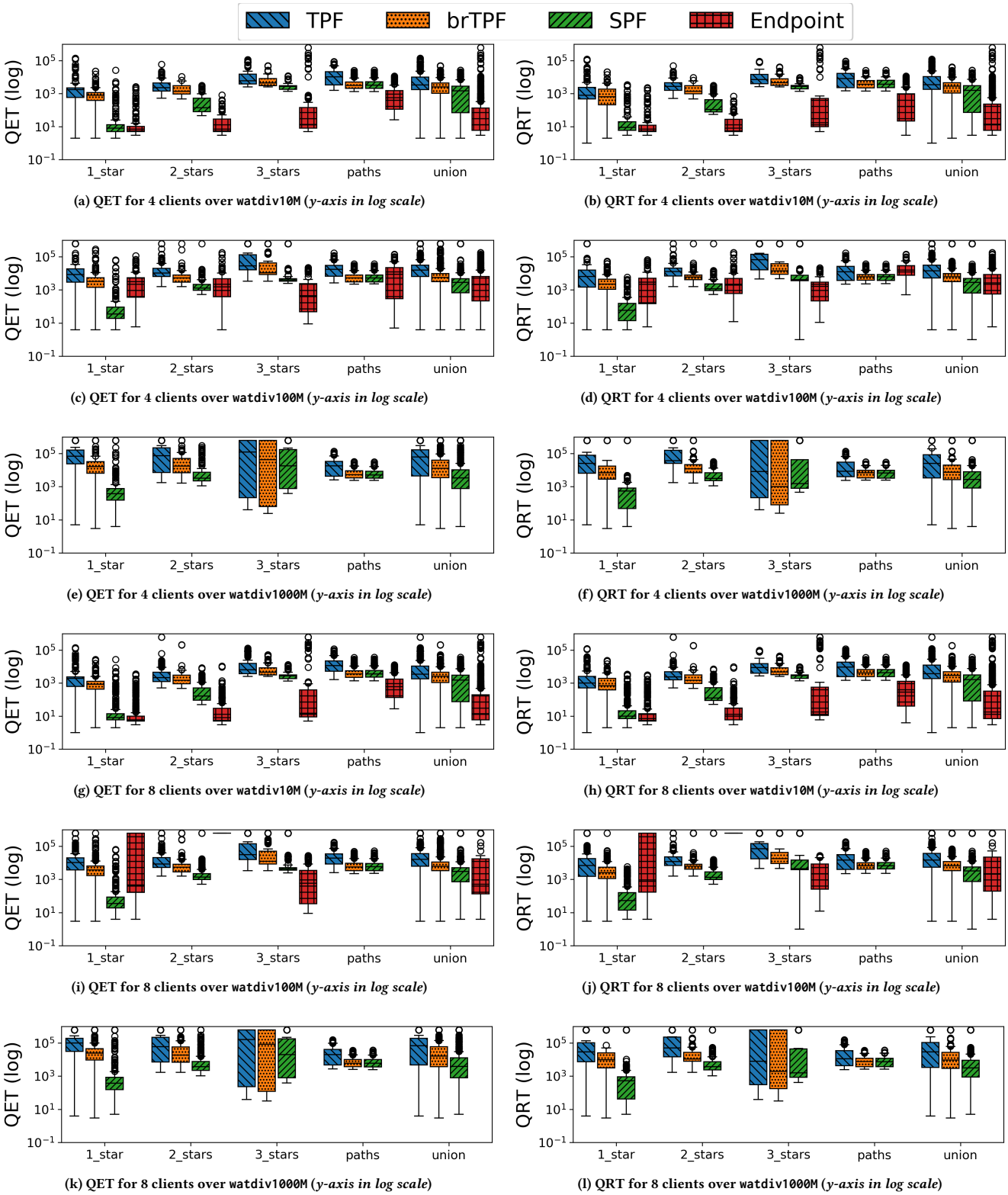
(i) QET for 2 clients over `watdiv100M` (*y-axis in log scale*)

(j) QRT for 2 clients over `watdiv100M` (*y-axis in log scale*)

(k) QET for 2 clients over `watdiv1000M` (*y-axis in log scale*)

(l) QRT for 2 clients over `watdiv1000M` (*y-axis in log scale*)

Figure 16: Query execution time (QET) and query response time (QRT) for 1 and 2 clients for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.



(a) QET for 4 clients over `watdiv10M` (*y-axis in log scale*)

(b) QRT for 4 clients over `watdiv10M` (*y-axis in log scale*)

(c) QET for 4 clients over `watdiv100M` (*y-axis in log scale*)

(d) QRT for 4 clients over `watdiv100M` (*y-axis in log scale*)

(e) QET for 4 clients over `watdiv1000M` (*y-axis in log scale*)

(f) QRT for 4 clients over `watdiv1000M` (*y-axis in log scale*)

(g) QET for 8 clients over `watdiv10M` (*y-axis in log scale*)

(h) QRT for 8 clients over `watdiv10M` (*y-axis in log scale*)

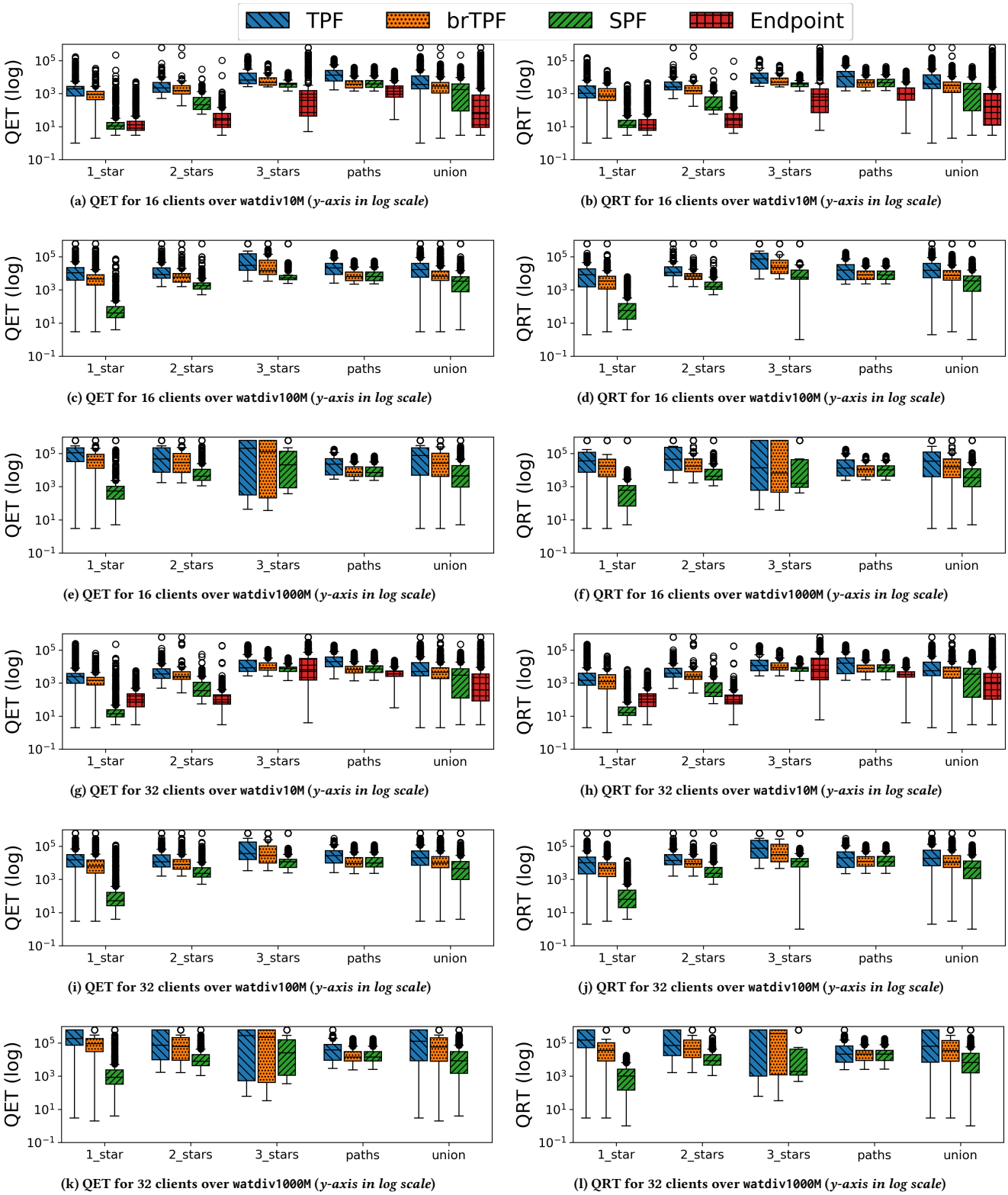
(i) QET for 8 clients over `watdiv100M` (*y-axis in log scale*)

(j) QRT for 8 clients over `watdiv100M` (*y-axis in log scale*)

(k) QET for 8 clients over `watdiv1000M` (*y-axis in log scale*)

(l) QRT for 8 clients over `watdiv1000M` (*y-axis in log scale*)

Figure 17: Query execution time (QET) and query response time (QRT) for 4 and 8 clients for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.



(a) QET for 16 clients over `watdiv10M` (*y-axis in log scale*)

(b) QRT for 16 clients over `watdiv10M` (*y-axis in log scale*)

(c) QET for 16 clients over `watdiv100M` (*y-axis in log scale*)

(d) QRT for 16 clients over `watdiv100M` (*y-axis in log scale*)

(e) QET for 16 clients over `watdiv1000M` (*y-axis in log scale*)

(f) QRT for 16 clients over `watdiv1000M` (*y-axis in log scale*)

(g) QET for 32 clients over `watdiv10M` (*y-axis in log scale*)

(h) QRT for 32 clients over `watdiv10M` (*y-axis in log scale*)

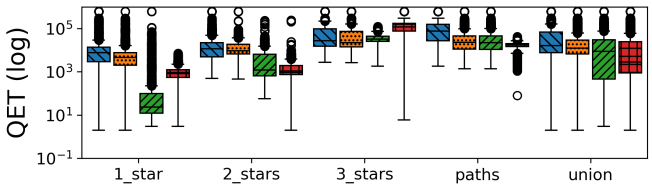
(i) QET for 32 clients over `watdiv100M` (*y-axis in log scale*)

(j) QRT for 32 clients over `watdiv100M` (*y-axis in log scale*)

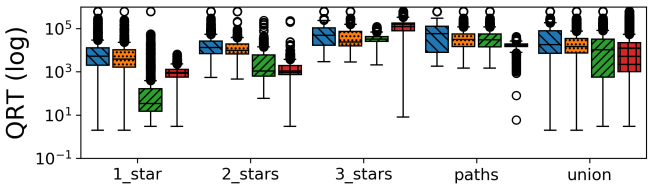
(k) QET for 32 clients over `watdiv1000M` (*y-axis in log scale*)

(l) QRT for 32 clients over `watdiv1000M` (*y-axis in log scale*)

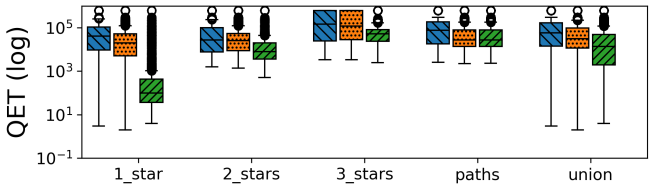
Figure 18: Query execution time (QET) and query response time (QRT) for 16 and 32 clients for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.



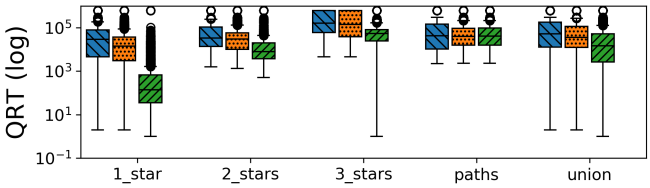
(a) QET for 128 clients over `watdiv10M` (*y-axis in log scale*)



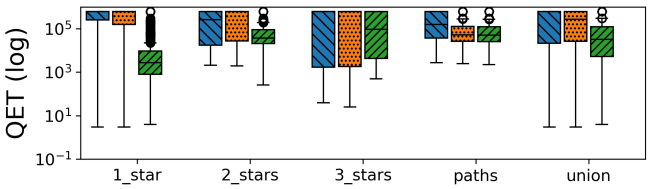
(b) QRT for 128 clients over `watdiv10M` (*y-axis in log scale*)



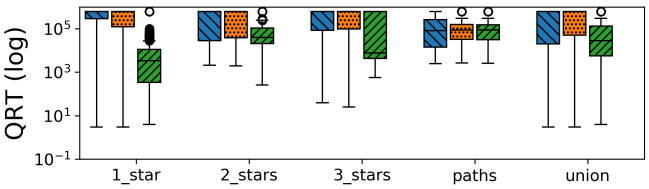
(c) QET for 128 clients over `watdiv100M` (*y-axis in log scale*)



(d) QRT for 128 clients over `watdiv100M` (*y-axis in log scale*)



(e) QET for 128 clients over `watdiv1000M` (*y-axis in log scale*)



(f) QRT for 128 clients over `watdiv1000M` (*y-axis in log scale*)

Figure 19: Query execution time (QET) and query response time (QRT) for 128 clients for all approaches over each query load for `watdiv10M`, `watdiv100M`, and `watdiv1000M`.