Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns

Christian Aebeloe, Ilkcan Keles, Gabriela Montoya, and Katja Hose

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

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 so that the data becomes 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 can combine 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 at least an order of magnitude faster in comparison to the state-of-the-art interfaces under high query processing load.

1 Introduction

Despite recent efforts to speed up SPARQL query processing under high querying load [9,13,18], answering SPARQL queries is still an expensive and difficult task. In fact, deciding whether a set of bindings is an answer to a query has been shown to be at least NP-complete [16]. Still, Triple Pattern Fragments (TPF) [18] 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, in turn, increases the availability of the server and ensures more efficient query processing during periods with high query loads.

Nevertheless, there are cases where TPF is significantly less efficient than SPARQL endpoints. Consider, for example, the SPARQL query shown in Listing 1.1 over the DBpedia dataset [7]. Executing this query using TPF requires transferring a huge number of intermediate results. In addition, the TPF client sends an HTTP request to the server for each binding obtained from the previously evaluated triple patterns, which results in a high number of calls to the

```
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 1.1: Find Germans and Norwegians that have won the same award and their birth dates

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) [9], have different ways to address this issue. brTPF, for instance, 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 [9]). While this results in significantly fewer calls to the server, it still incurs relatively high network traffic.

All of these approaches ignore the potential of exploiting and evaluating star-shaped subqueries. Such subqueries (i) can be computed relatively efficiently on the server [16] and (ii) have a potential of reducing the network traffic since fewer intermediate results are transferred. Star-shaped subqueries, such as subqueries {`tp1 . tp2 . tp3`} and {`tp4 . tp5 . tp6`} in Listing 1.1, do not require full SPARQL expressiveness. In this paper, we therefore propose a novel interface that decomposes SPARQL queries into star-shaped subqueries and is able to combine a lower network load with a comparatively low server load, and in doing so improves the overall query processing performance.

In summary, this paper makes the following contributions:

- A definition of Star Pattern Fragments (SPF), a novel RDF interface to efficiently answer star-shaped graph patterns.
- A formalization and an implementation of an SPF server.
- Query processing strategies to efficiently compute answers to SPARQL queries with star-shaped subqueries in an SPF setup.
- An extensive evaluation of the SPF interface using WatDiv [4].

This paper is organized as follows. Section 2 discusses related work, Section 3 introduces the terminology used through the paper, Section 4 presents a formal characterization of the Star Pattern Fragments interface, Section 5 describes the SPF server and client details, Section 6 discusses our 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, a recent study [5] found that more than half of all public SPARQL endpoints have less than 95% availability, 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 [18]. Decentralization has been achieved by using Peer-to-Peer (P2P) based architectures [2,6,12] and federated engines [14,17].

However, even if P2P systems increase the availability of datasets, they are vulnerable to churn (when nodes frequently leave and join the network) [6,12] 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 [14,17] 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 [15], 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 [19] 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. Therefore, they cannot increase the availability of the servers.

Triple Pattern Fragments (TPF) [18] were proposed to improve the server availability under load. TPF servers only process individual triple patterns and therefore have a lower processing burden than SPARQL endpoints. TPF clients rely on either a greedy algorithm [18], a metadata based strategy [11], 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. [10] 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) [9] was proposed to address 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. hybridSE [13], on the other hand, relies on both SPARQL endpoints and TPF 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 as well. In this paper, we instead propose an interface that provides a different 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

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

be answered in linear complexity [16]. Our approach therefore achieves a reduction on the data transfer and the execution time without having a negative impact on the data availability.

3 Background

The de facto standard 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 are triples of the form $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$. 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 the subjects of all these triple patterns are the same, i.e., $s_i = s_j$ for any $1 \leq i, j \leq n$. In the following, we recall the definition of Linked Data Fragments (LDFs) [18].

Definition 2 (Selector Function [18]). *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 $s : 2^{\mathcal{T}^*} \to 2^{\mathcal{T}^*}$.*

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 3 (Hypermedia Controls [18]). *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 4 (Linked Data Fragment [18]). *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*

[2] http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
[3] http://www.w3.org/TR/rdf-sparql-query/

- *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 [18].

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, Vidal et al. [19] show that the decomposition of queries into star-shaped subqueries and subsequent evaluation of these subqueries by a SPARQL query engine leads to good query processing performance. Such decomposition does not impose a very high 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 [9] 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 1.1), this is preferable to ensure fewer requests to the server, which results in lower network traffic and faster query processing. The relative position of SPF between different RDF interfaces is shown in Figure 1.

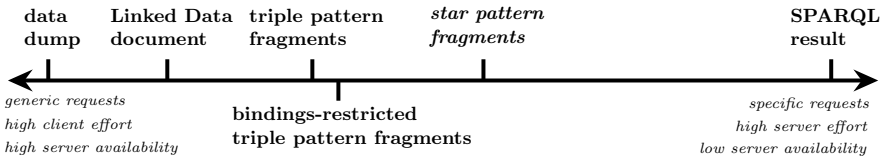


Fig. 1: HTTP interfaces for RDF data (adapted from [9,18]).

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 RDF triples T is said to be *matching triples* for a star pattern 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.

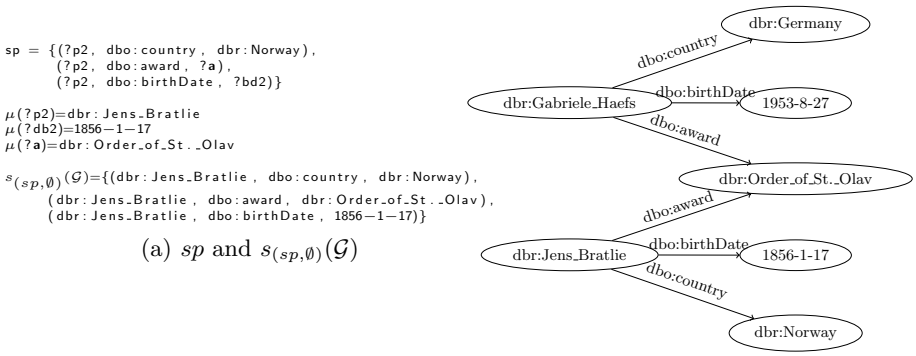
Similar to how brTPF [9] couples bindings and triple patterns, we couple bindings obtained from previously evaluated star patterns with subsequent star patterns to decrease the network traffic.

Definition 5 (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:*

$$s_{(sp,\Omega)}(\mathcal{G}) = \begin{cases} \{T \mid T \subseteq \mathcal{G} \wedge T \text{ are matching triples to } sp\} & \text{if } \Omega \text{ is empty,} \\ \{T \mid T \subseteq \mathcal{G} \wedge T \text{ are matching triples to } sp \wedge \\ \quad \exists \mu \in [[sp]]^{\mathcal{G}}, \mu' \in \Omega : \mu[sp] = T \wedge \mu' \subseteq \mu\} & \text{otherwise.} \end{cases}$$

The simplest star pattern consists of a single triple pattern. For this reason, SPF is backwards compatible with both TPF [18] and brTPF [9]. 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 function or the bindings-restricted triple pattern-based selector function.

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 (when Ω is empty) the three triples from $\mathcal{G}$ that include dbr:Jens_Bratlie as subject, as shown in Figure 2a.



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

(b) RDF Graph

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

In order to formally define SPF, we adapt the general definition of LDF given in [18]. An SPF is defined as follows:

Definition 6 (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. $c \in C$*.*

5 Query Processing

The SPF interface processes queries using resources from both the server and the client. The server provides separate 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. Decompose Q into star-shaped subqueries and determine the join order such that the subqueries with the highest selectivity (lowest cardinality) are evaluated first.
2. Obtain intermediate results for each of Q's subqueries by applying the star pattern-based selector on the server side. In other words, an SPF server returns the set of matching triples for a given star pattern.
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.

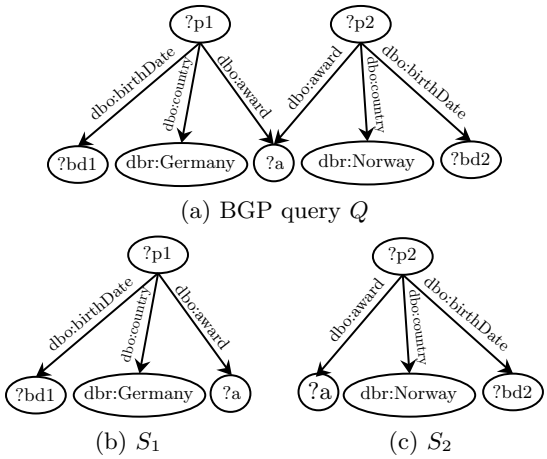
5.1 Client-Side Query Processing

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 7 (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 star patterns (in other words, for all* $1 \leq j \leq m$*, if* $(s_a, p_a, o_a) \in S_j$ *and* $(s_b, p_b, o_b) \in S_j$*,* $s_a = s_b$*), (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* $tp_i \in S_j$*,* $tp_i \in Q$*.*

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

Using Definition 7, 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.



(a) BGP query Q

(b) S_1 (c) S_2

Fig. 3: Star decomposition of Q (Listing 1.1) into S_1 and S_2.

An example of using Definition 7 to partition the BGP query Q (Listing 1.1) 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 query Q, the SPF client performs the following steps:

1. Obtain the star decomposition (Definition 7) of Q, i.e., $\{S_1, \ldots, S_n\}$.
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 6), 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.

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 $s_{(sp,\Omega)}(\mathcal{G})$ applying sp and Ω to $\mathcal{G}$.
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 $\mu \in s_{(sp,\Omega)}(\mathcal{G})$

These results are then processed by the client, which operates 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, SPF performs as good as brTPF in the worst case where all star patterns have exactly one triple pattern.

5.3 Implementation Details

We implemented the SPF server and the SPF client using Java 8[5]. The client uses an approach to process queries that is based on a block nested loop join.

Server. The SPF server is implemented as an extension of the Java implementation of the brTPF server[6]. Our server implementation uses `HDT` [8] as backend. `HDT` is originally proposed to process a single triple pattern over a knowledge graph efficiently. However, the `HDT` Java library includes an Apache Jena[7] implementation to issue SPARQL queries over the `HDT` backend. We use this implementation to process the star pattern requests.

Client. We implemented our own brTPF client in line with [9], and extended this client with the SPF-based query processing method described in Section 5.1. Like TPF [18] and brTPF clients [9], SPF client uses a pipeline of iterators that represent a left-deep join tree. However, they define the join operations on triple patterns, whereas we define join operations on star patterns.

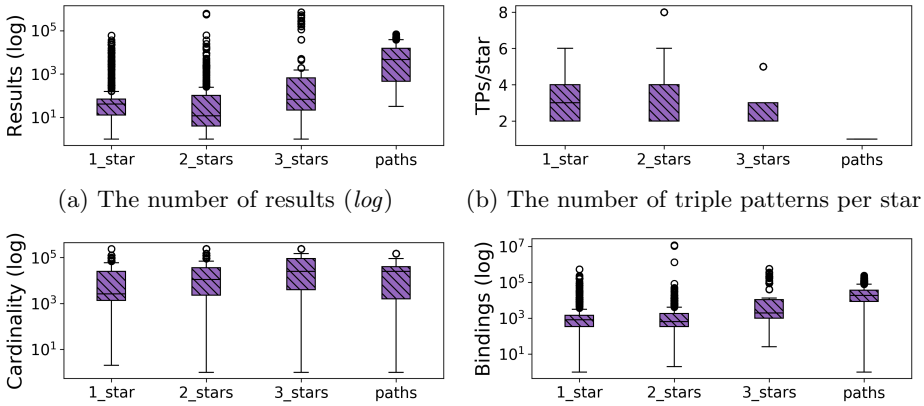
6 Evaluation

In order to confirm the hypothesis that SPF, which processes SPARQL queries using star-shaped decomposition, increases query throughput by combining a lower network load with a comparatively low server load, we ran experiments where we compared SPF, TPF [18], brTPF [9], and a SPARQL endpoint.

Dataset and Queries: We used the WatDiv benchmark [4] to generate a dataset with 10 million triples. To study the impact of the number of star-shaped subqueries, we used the WatDiv query generator to obtain query loads

[5] http://github.com/Chraebe/StarPatternFragments
[6] http://olafhartig.de/brTPF-ODBASE2016/
[7] https://jena.apache.org/

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`). We also considered an additional query load that is the union of the four query loads described above (`union`). Figure 4 shows the number of results per query, the number triple patterns per star pattern, the estimated number of relevant triples per triple pattern[9], and the number of bindings transferred during query processing with TPF. The average path length for queries in the `paths` query load was 6.89 triple patterns, while the longest path had 9 triple patterns.



(a) The number of results (*log*)

(b) The number of triple patterns per star

(c) Fragment cardinalities (*log*)

(d) The number of bindings (*log*)

Fig. 4: Statistics for each query load: the number of results, the number of triple patterns per star pattern, estimated fragment cardinalities, and the number of intermediate bindings for each query.

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

Hardware Setup: To run the clients, we used four identical virtual machines (VMs), each running up to 32 clients concurrently. All 4 VMs had 32 vCPU cores with a clock speed of 3GHz, 64KB L1 cache, 4096KB L2 cache, and 16384KB

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

[9] It is the estimated cardinality included in the metadata of the relevant fragments

L3 cache, and a main memory of 264GB. Each client was limited to use just one vCPU core and 7.8GB RAM. The LDF server and the SPARQL endpoint were run, at all times, on a server with 16 vCPU cores with the same specifications as the cores on VMs used for the clients, and a main memory of 128GB.
Evaluation Metrics:

- *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 16 threads at a time (one per vCPU core on the server) with `NumberOfBuffers` = 9735000 and `MaxDirtyBuffers` = 7301250. The LDF page size was, throughout our experiments, set to 50 results, and the maximum number of elements in Ω was set to 30 for both brTPF and SPF, i.e., they can send up to 30 bindings with each request. The timeout was set to 600 seconds, i.e., 10 minutes.

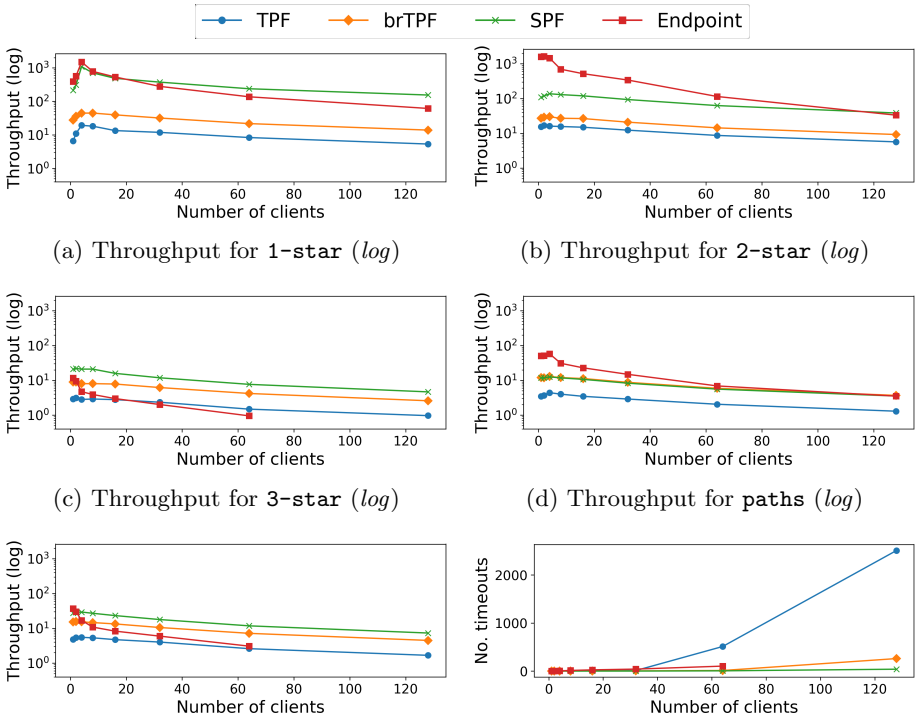
6.1 Experimental Results

Our objective is to verify that 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 confirm that SPF is, in the case of path queries, still as good in terms of performance as brTPF. Due to space restrictions, we will only show the most important results in this section. We provide all source code, experimental setup (queries, datasets, etc.), as well as the full experimental results on our website[10].

The SPARQL endpoint crashed at 128 clients for `3-stars` and `union` due to high load. We therefore only show the results of the endpoint up until 64 clients for these query loads. For the remainder of this section, we will focus on the setup with 64 concurrent clients and the `union` query load unless otherwise specified, since this was the highest number of clients all approaches were able to execute successfully.

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

[10] http://relweb.cs.aau.dk/spf



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

(b) Throughput for `2-star` (*log*)

(c) Throughput for `3-star` (*log*)

(d) Throughput for `paths` (*log*)

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

(f) Number of timeouts

Fig. 5: Throughput (# queries/m) for each query load and configuration and the number of timeouts for the `union` query load. Endpoint lacks values in `3-stars` and `union` for 128 clients because it crashed.

slightly, the reduction in network traffic is expected to improve performance for queries that include star-shaped subqueries.

Figure 5 shows the throughput of the four approaches for different numbers of concurrent clients, as well as the total number of timeouts for the `union` load. Note that for the `1-star` query load, the throughput increases for all approaches until four concurrent clients, but it decreases afterwards. This is due to the fact that when running more concurrent clients, more queries are processed in total. However, the increased server load did not significantly affect the execution time until we have eight concurrent clients.

Clearly, SPF has a significantly higher throughput compared to TPF and brTPF for queries with at least one star pattern. The only query load where SPF does not outperform brTPF is the `paths` query load (Figure 5d). This is expected since the query processing using SPF is identical to the query processing using brTPF for a path query. Overall, the throughput of all the interfaces deteriorates as the number of concurrent clients increases. Even if the SPF server computes the joins in the star patterns, there is only a slight increase in the CPU load. For this reason, SPF remains better than both TPF and brTPF for all the configurations. In addition, due to more efficient query processing, SPF has significantly fewer timeouts than all other approaches (Figure 5f). The endpoint

is clearly the best performing interface if there are only few concurrent clients. However, its performance deteriorates much faster than the other approaches when the number of concurrent clients increases. Moreover, `3-stars` contains generally larger queries (Figure 4), and thus puts more load on the server. SPF, TPF and brTPF are able to handle this increased load more efficiently than the endpoint (Figure 5c). This is in line with the experiments shown in [18], and shows that SPF seems to be a suitable alternative to handle large query loads.

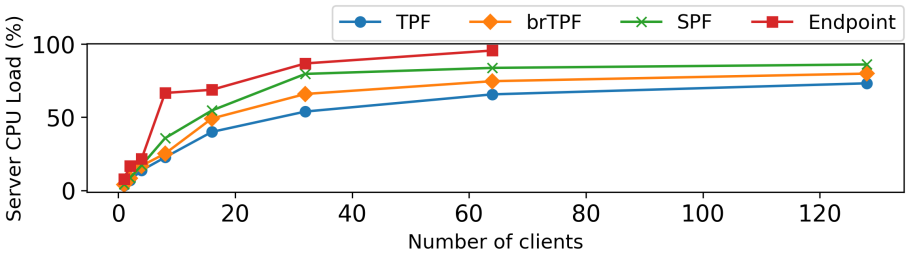


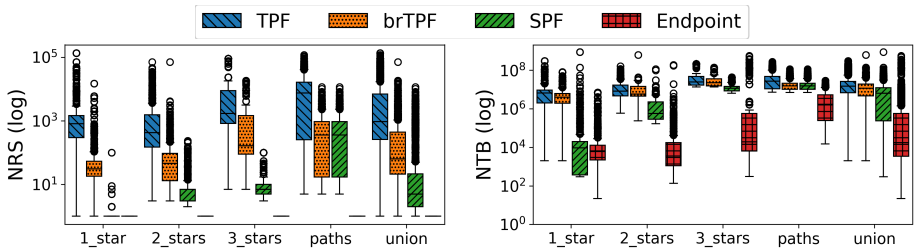
Fig. 6: CPU load for the `union` query load and each configuration.

Figure 6 shows the server CPU load of each approach for the `union` query load and all the configurations. Clearly, the endpoint has the highest CPU load throughout our experiments. SPF has a slightly higher CPU load than brTPF and TPF; however, this is not significant enough to affect availability.

Overall, our experiments confirm the hypothesis that SPF increases query throughput while maintaining relatively low server load even in the presence of high querying load. Moreover, our experiments show that even in the worst-case where the queries do not contain any star patterns, SPF is as performant as brTPF. 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.

Network traffic One of the main advantages of SPF highlighted in previous sections is that more selective requests are sent to the server, i.e., subqueries that may be composed of more than one triple pattern. Especially for queries with large star patterns, this should result in fewer requests to the server and less data (i.e., intermediate results) transfer between the server and the client.

Figure 7a shows NRS for the experiments with 64 clients. 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. 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's query processing when no stars are included in the query.



(a) NRS (*y-axis in log scale*). (b) NTB (*y-axis in log scale*).

Fig. 7: Network traffic for 64 clients for all approaches over each query load.

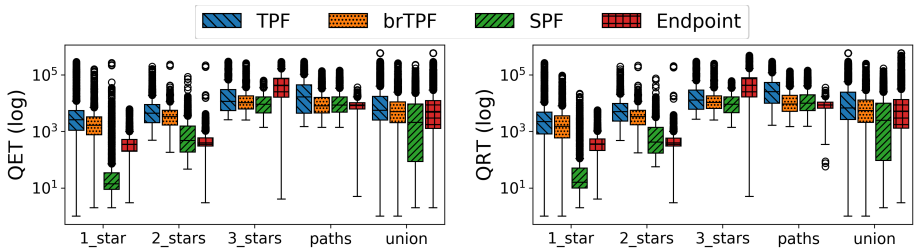
Similarly, since the SPF server processes larger parts of the queries, fewer of the intermediate results are sent back to the clients, resulting in a lower NTB (Figure 7b). Similar to NRS, NTB is significantly lower for SPF in comparison to both TPF and brTPF throughout all query loads except `paths`, 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.

Impact of the query load Figure 8a shows QET for all five query loads in the configuration with 64 concurrent clients. 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, some queries in the `1-star` query load can be answered with just a single call to the server. As shown in Figures 5 and 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 4b) and therefore TPF and brTPF have to make more requests to the server for these queries, whereas SPF still only makes one request to the server.

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. Our results thus confirm that SPF is not slower for queries with no star patterns than brTPF, and is faster for queries with star patterns.

While the endpoint is the fastest overall according to Figure 8, Figure 5 illustrates that its performance decreases much faster under high query processing load than it does for all other RDF interfaces.

Response time Figure 8b shows QRT for all query loads for the configuration with 64 concurrent clients. 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.



(a) QET (*y-axis in log scale*). (b) QRT (*y-axis in log scale*).

Fig. 8: Query execution time (in ms) and query response time (in ms) of each query load and the configuration with 64 clients.

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.

6.2 Summary

Overall, our experimental evaluation shows that SPF achieves a novel, and in some cases better, tradeoff between performance and server load than TPF and brTPF. SPF does this by significantly reducing the network traffic without incurring too much extra load on the server. 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 does have slightly higher CPU load on the server side, it is still significantly more efficient than TPF and brTPF in the presence of high querying load. This confirms our hypothesis 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 is based on star patterns. In order to process SPARQL queries, an SPF client decomposes the query into star shaped subqueries and sends these subqueries, along with intermediate bindings, to the server. The client also takes care of processing other SPARQL operators. 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 by decomposing them into star patterns and sending them to the server. 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 25 times compared to brTPF and up to 55 times compared to TPF. Our evaluation also demonstrates that SPF increases the overall performance without incurring significantly more CPU load on the server, even when a large number of clients issue queries concurrently. As future work, it could be interesting to include an SPF-specific cache on the server and to assess its impact on the performance of SPF, as well as evaluating SPF using query loads with other SPARQL operators.

Acknowledgments. This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B and Aalborg University's Talent Management Programme.

References

1. Acosta, M., Vidal, M.: Networks of linked data eddies: An adaptive web query processing engine for RDF data. In: ISWC 2015. pp. 111–127 (2015)
2. Aebeloe, C., Montoya, G., Hose, K.: A decentralized architecture for sharing and querying semantic data. In: ESWC 2019. pp. 3–18 (2019)
3. Aebeloe, C., Montoya, G., Hose, K.: Decentralized indexing over a network of rdf peers. In: ISWC 2019 (2019)
4. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: ISWC 2014. pp. 197–212 (2014). https://doi.org/10.1007/978-3-319-11964-9_13
5. Aranda, C.B., Hogan, A., Umbrich, J., Vandenbussche, P.: SPARQL Web-Querying Infrastructure: Ready for Action? In: ISWC'13. pp. 277–293 (2013)
6. Cai, M., Frank, M.R.: RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In: WWW (2004)
7. DBpedia: DBpedia version 2016-10 (2016), https://wiki.dbpedia.org/develop/datasets/dbpedia-version-2016-10, [Online; accessed October-2018]
8. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary rdf representation for publication and exchange (hdt). J. Web Semantics **19**, 22–41 (2013)
9. Hartig, O., Aranda, C.B.: brtpf: Bindings-restricted triple pattern fragments (extended preprint). CoRR **abs/1608.08148** (2016)
10. Heling, L., Acosta, M., Maleshkova, M., Sure-Vetter, Y.: Querying large knowledge graphs over triple pattern fragments: An empirical study. In: ISWC 2018. pp. 86–102 (2018)
11. Herwegen, J.V., Verborgh, R., Mannens, E., de Walle, R.V.: Query execution optimization for clients of triple pattern fragments. In: ESWC 2015. pp. 302–318
12. Kaoudi, Z., Koubarakis, M., Kyzirakos, K., Miliaraki, I., Magiridou, M., Papadakis-Pesaresi, A.: Atlas: Storing, updating and querying RDF(S) data on top of DHTs. J. Web Sem. **8**(4), 271–277 (2010)
13. Montoya, G., Aebeloe, C., Hose, K.: Towards efficient query processing over heterogeneous RDF interfaces. In: DeSemWeb@ISWC 2018 (2018)
14. Montoya, G., Skaf-Molli, H., Hose, K.: The odyssey approach for optimizing federated SPARQL queries. In: ISWC 2017. pp. 471–489 (2017)
15. Montoya, G., Vidal, M., Acosta, M.: A heuristic-based approach for planning federated SPARQL queries. In: COLD 2012 (2012)
16. Pérez, J., Arenas, M., Gutiérrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16:1–16:45 (2009)
17. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: A federation layer for distributed query processing on linked open data. In: ESWC 2011. pp. 481–486 (2011)
18. Verborgh, R., Sande, M.V., Hartig, O., Herwegen, J.V., Vocht, L.D., Meester, B.D., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. J. Web Sem. **37-38**, 184–206 (2016)
19. Vidal, M., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A.: Efficiently Joining Group Patterns in SPARQL Queries. In: ESWC 2010. pp. 228–242 (2010)