Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns

Christian Aebeloe a,*, Ilkcan Keles a,b, Gabriela Montoya a and Katja Hose a
a Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, DK-9220 Aalborg Øst, Denmark
E-mails: caebel@cs.aau.dk, ilkcan@cs.aau.dk, gmontoya@cs.aau.dk, khose@cs.aau.dk
b Turkcell, Istanbul, Turkey
E-mail: ilkcan.keles@turkcell.com.tr

Abstract. SPARQL endpoints offer access to a vast amount of interlinked information. While they offer a well-defined interface for efficiently retrieving results for complex SPARQL queries, complex query loads can easily overload or crash endpoints as all the computational load of answering the queries resides entirely with the server hosting the endpoint. Recently proposed interfaces, such as Triple Pattern Fragments, have therefore shifted some of the query processing load from the server to the client at the expense of increased network traffic in the case of non-selective triple patterns. This paper therefore proposes Star Pattern Fragments (SPF), an RDF interface enabling a better load balancing between server and client by decomposing SPARQL queries into star-shaped subqueries, evaluating them on the server side. Experiments using synthetic data (WatDiv), as well as real data (DBpedia), show that SPF does not only significantly reduce network traffic, it is also up to two orders of magnitude faster than the state-of-the-art interfaces under high query load.

Keywords: SPF, star patterns, Linked Data Fragments, decentralization, knowledge graphs

1. Introduction

Over the past decade, the Semantic Web community has seen a rapid increase in the volume of data available as Linked Open Data (LOD). Multiple LOD datasets have been released spanning a broad range of different topics, such as geography (e.g., LinkedGeoData [1]), life sciences (e.g., Bio2RDF [2]), government data (e.g., US Government LOD [3]), and general knowledge (e.g., DBpedia [4]). Today, such open datasets can have several billions of triples, for example DBpedia [4] where the English language dataset alone has over a billion triples, Wikidata [5] with around 12 billion triples, and Bio2RDF [2] with over 10 billion triples.

Such datasets are often made available through public endpoints, dereferenceable URIs, or downloadable data dumps. However, this kind of access relies totally on the individual data providers to provide access to their data. As multiple previous studies have highlighted [6, 7], this presents a huge burden for the data providers and often results in the performance of such public endpoints deteriorating quickly as the load increases [6], and worst case can at times be simply unavailable [8, 9].

Despite recent efforts to speed up SPARQL query processing under high query load [6, 10–13], answering SPARQL queries remains 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 [14]. Still, Triple Pattern Fragments (TPF) [6] 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 the remaining query processing tasks. This increases the availability of the server and ensures more efficient query processing during periods with high load.

Nevertheless, there are cases where TPF is significantly less efficient than SPARQL endpoints. Consider, for example, the Spar...
Triple Pattern Fragments (brTPF) [10] and hybridSE [11] present different ways of addressing this issue. brTPF uses block nested loop-like joins, where a triple pattern is evaluated once per group of $N$ issues. brTPF uses block nested loop-like joins, where a triple pattern is evaluated once per group of $N$ issues. While this results in significantly fewer calls to the server, it still incurs a relatively high network traffic (364 calls to process tp2 given the bindings found for tp1 in Listing 1).

What all these approaches ignore though is the potential of evaluating conjunctive subqueries. Such subqueries can (i) be computed relatively efficiently on the server [14] and (ii) reduce the network traffic since fewer intermediate results are transferred. Subqueries, such as subqueries \{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 (e.g., based on shared variable between triple patterns [6]), decomposition into star-shaped subqueries is the focus of this paper.

This paper therefore investigates what effects evaluating star-shaped subqueries on the server while still processing queries on the client has on the network usage, client load and server load under high query load. The paper introduces a novel interface that improves the overall query processing performance, while also ensuring high availability by combining a lower network load with a comparatively low server load through decomposing SPARQL queries into star-shaped subqueries.

In summary, this paper makes the following contributions:

- Definition of Star Pattern Fragments (SPF), a novel RDF interface that reduces network usage while keeping the server load comparatively low.
- Formalization and implementation of an SPF server.
- 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.

To assess the effects of processing star-shaped subqueries on the server while executing the queries on the client, a thorough evaluation of SPF using three different sized WatDiv [15] datasets with up to 1 billion triples, using large query loads for stress testing, is provided. Moreover, SPF is evaluated against DBpedia [4] using queries posed by real users [16] to evaluate how the approach performs in real-world scenarios.

This paper is organized as follows. Section 2 discusses related work, Section 3 introduces the terminology used in this 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 experimental results, and Section 7 concludes the paper and provides a perspective on future work.

### 2. Related Work

One of the most popular interfaces for querying RDF data is SPARQL endpoints. SPARQL endpoints are Web services that implement the SPARQL protocol and usually provide an HTTP interface that accepts SPARQL queries. However, several studies [8, 9] have previously highlighted the fact that such endpoints are often unavailable, meaning that accessing data can sometimes be impossible.

Decentralization of the data storage and distribution of query processing between clients and servers is often referred to when discussing solutions to dataset availability [6, 7, 17–19]. For example, the Solid project [17] uses decentralized Personal Online Datastores (PODs) to separate personal information from

1 Code is available on the SPF website http://relweb.cs.aau.dk/spf
applications. Users can decide for themselves where their POD is stored, and which application have access to it. Thus when loading a Solid application, it must query data from multiple sources located around the Web. However, Solid focuses mostly on the security of personal datasets, whereas this paper focuses on efficient query processing during high server loads.

While Solid was designed for large-scale open datasets, the decentralization of data storage has been achieved by using Peer-to-Peer (P2P) based architectures [7, 20–23], federated engines [24–29], and Linked Data Fragments (LDFs) [6, 30, 31]. This section contains an analysis of each approach to decentralization for Linked Data, and an explanation of the pitfalls of such approaches.

2.1. Peer-to-Peer Systems

P2P systems can generally be divided into two separate groups: structured [20, 21, 32] and unstructured [7]. Structured P2P systems impose a structured overlay over the connections within the network, e.g., Dynamic Hash Tables (DHTs). However, such systems are generally vulnerable to situations where nodes frequently leave or join the network (churn), or they frequently upload new data. The main reason is that the overlay has to be recomputed upon a change in the network. Unstructured P2P systems, on the other hand, impose no such overlay, and are thus more robust under churn. In fact, [7] showed that such networks, relying on replication of data on multiple nodes, can indeed improve the availability of data, even if a large portion of the nodes fail. However, since data relevant to a query can generally be located on nodes throughout the network, efficient query processing is a big challenge. To this end, [33] showed that finding where exactly in a neighborhood relevant data is located through indexes can negate the need to flood the entire network to process a query, and thus improve query processing. Such indexes are based on summaries of local fragments using Bloom Filters [34] and intersections hereof [35], as well as on routing indexes previously proposed for other P2P systems [36, 37]. However, such networks can still cause very high network traffic (when queries have large numbers of intermediate results). In any case, Star Pattern Fragments (SPF) are orthogonal to P2P systems. For example, [33] could be extended with SPF to achieve a similar reduction of intermediate results as described in Section 6.

2.2. Federated Systems

Federated query engines [24–29] divide SPARQL query processing over multiple SPARQL endpoints. Nonetheless, they sometimes fail to generate optimal query plans that transfer the minimum amount of data from endpoints to the engine and therefore increase the load on SPARQL endpoints. This means that they sometimes still suffer from relatively high server load. Query optimization techniques for federated engines, such as [38], 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, such approaches execute star-shaped subqueries over SPARQL endpoints. As mentioned earlier, such endpoints suffer from unavailability [8, 9], and they can therefore not increase the availability. Instead, other optimization techniques for federated engines [29, 40] focus on estimating the selectivity of joins to produce better query execution plans. These approaches could be combined with SPF and provide the benefits highlighted in this paper to federated systems as well.

2.3. Linked Data Fragments

Triple Pattern Fragments (TPF) [6] 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 [6], a metadata based strategy [30], or adaptive query processing techniques and star-shaped decomposition [31] 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. [41] found that the performance of TPF is heavily affected by aspects such as the triple pattern type and the fragment cardinality. Bindings-Restricted TPF (brTPF) [10] was proposed to reduce 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 [11]...
combines 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 [8], which leaves the approach vulnerable to downtime.

2.4. Other Systems

Other systems use more complex techniques to address some of the issues posed by TPF. SaGe [12], for example, uses a preemptive model that suspends queries after a fixed time quantum, as to not starve simpler queries of system resources, after which they can be resumed upon client request. This, however, often leads to a large amount of requests to the server, since long running queries will have to be resumed several times. Reducing the network traffic is exactly what SPF aims to address. Furthermore, SaGe targets SPARQL endpoints, which suffer from unavailability [8, 9]. Smart-KG [13], on the other hand, ships star-shaped partitions to the client during query processing. This decreases the number of requests issued 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.

This paper proposes an interface that provides a different distribution of query processing tasks where, different from the approaches discussed in this section, joins in star-shaped subqueries are evaluated by the server. Such computations do not significantly increase the server load because star-shaped subqueries can be answered in linear complexity [14]. SPF therefore achieves a reduction on the data transfer and the execution time without having a negative impact on the data availability.

3. Preliminaries

The recommended format for storing semantic data is the Resource Description Framework (RDF)

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) \( G \) is a finite set of RDF triples. Today, SPARQL 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 \( (s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V) \).

In the following, a star pattern is defined to be one of two types of star patterns: subject-based star patterns or object-based star patterns.

Definition 2 (Subject-Based Star Pattern). A subject-based 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 all \( 1 \leq i, j \leq n \).

Definition 3 (Object-Based Star Pattern). An object-based 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 objects of all these triple patterns are the same, i.e., \( o_i = o_j \) for all \( 1 \leq i, j \leq n \).

Given the definition of subject-based star patterns and object-based star patterns, a star pattern is defined as follows.

Definition 4 (Star Pattern). A star pattern \( S \) is a set of \( n \) triple patterns, \( \{ (s_1, p_1, o_1), \ldots, (s_n, p_n, o_n) \} \), such that \( S \) is either a subject-based star pattern or an object-based star pattern.

Corollary, an RDF star is a set of RDF triples that has the same properties as in Definition 4.

3.1. Linked Data Fragments

A Linked Data Fragment (LDF) of a knowledge graph \( G \) consists of a subset of \( G \)’s triples (a fragment) coupled with metadata about the fragment and controls to retrieve similar LDFs. The following description of LDFs follows [6]. LDFs consider only blank-node-free RDF triples. An LDF is defined as follows.

Definition 5 (Linked Data Fragment [6]). Given a knowledge graph \( G \), a Linked Data Fragment (LDF) consists of the following three elements:

- **Data**: A subset of \( G \)’s triples
- **Metadata**: RDF triples that describe the data
- **Controls**: Links and forms to retrieve other LDFs of the same or other knowledge graphs

---

3. \( \text{https://www.w3.org/TR/rdf11-concepts/} \)

4. \( \text{https://www.w3.org/TR/sparql11-query/} \)
Any knowledge graph made available on the Web, in any format, can be described as an LDF. For example, a data dump can be described as a single LDF with the following components [6]:

- **Data:** All triples in the data dump
- **Metadata:** Data about the dump, e.g., version number, author, etc.
- **Controls:** No controls, since the entire data dump is given in the LDF. It could, however, contain controls to other versions of the data dump.

Given a knowledge graph $G$, each LDF of $G$ contains triples that somehow belong together. To obtain triples from $G$ to form a fragment, a selector function is used, and defined as follows.

**Definition 6 (Selector Function [6]).** Given $T^* = U \times U \times (U \cup \mathcal{L})$, the set of all blank-node-free RDF triples, a selector function $s$ is a function such that $s : 2^T \to 2^T$.

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.

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 7 (Hypermedia Controls [6]).** A hypermedia control is a function that maps from some set to $U$.

Any knowledge graph made available on the Web, in any format, can be described as an LDF. For example, a data dump can be described as a single LDF with the following components [6]:

- **Data:** All triples in the data dump
- **Metadata:** Data about the dump, e.g., version number, author, etc.
- **Controls:** No controls, since the entire data dump is given in the LDF. It could, however, contain controls to other versions of the data dump.

Given a knowledge graph $G$, each LDF of $G$ contains triples that somehow belong together. To obtain triples from $G$ to form a fragment, a selector function is used, and defined as follows.

**Definition 6 (Selector Function [6]).** Given $T^* = U \times U \times (U \cup \mathcal{L})$, the set of all blank-node-free RDF triples, a selector function $s$ is a function such that $s : 2^T \to 2^T$.

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.

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 7 (Hypermedia Controls [6]).** 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 8 (Linked Data Fragment [6]).** Given a knowledge graph $G$, a Linked Data Fragment (LDF) of $G$ is a 5-tuple $f = (u, s, \Gamma^s, M, C)$, with

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

An LDF server should divide each fragment $f = (u, s, \Gamma^s, M, C)$ into reasonably sized LDF pages $\phi = (u', u_f, s_f, \Gamma^s, M', C')$, containing (i) the URI $u'$ from which $\phi$ could be obtained and $u' \neq u$, (ii) $u_f = u$, (iii) $s_f = s$ (iv) $\Gamma^s \subseteq \Gamma$, (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 [6].

**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 and client. 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 the experiments in Section 6.

This section contains a formal definition of Star Pattern Fragments (SPF), as an extension of brTPF [10], 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 relative position of SPF between different RDF interfaces is shown in Figure 1.

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

- **Data:** All RDF stars in $G$ 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

As with TPF [6], an SPF server should divide each fragment into reasonably sized pages to avoid unnecessary downloads of large amounts of stars. This page size could, for instance, be dynamically based on the amount of triple patterns in the star pattern.

The remainder of this section formalizes SPF by adapting the general formalizations of TPF [6] and brTPF [10], and provides a response format for an SPF request in the form of a Hydra formalization [42].
4.1. Formal Definition

Let \([S]_\mathcal{G}\) be the answer to a star pattern \(S\) over a knowledge graph \(\mathcal{G}\). \([S]_\mathcal{G}\) 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 \(S\), denoted \(T[S]\), if there exists a solution mapping \(\mu\) in \([S]_\mathcal{G}\) such that \(T = \mu[S]\) where \(\mu[S]\) denotes the triples (or triple patterns) obtained by replacing the variables in \(S\) with values according to \(\mu\).

Similar to how \(\text{brTPF}\) [10] couples bindings and triple patterns, SPF couples bindings obtained from previously evaluated star patterns with subsequent star patterns to decrease the network traffic.

**Definition 9 (Star Pattern-Based Selector Function).**

Given a star pattern \(S\) and a finite sequence of solution mappings \(\Omega\), the star pattern-based selector function for \(S\) and \(\Omega\), \(s_{(S,\Omega)}\), is the selector function that, for every knowledge graph \(\mathcal{G}\), is defined as follows.

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

The simplest star pattern consists of a single triple pattern. For this reason, SPF is backwards compatible with both TPF [6] and \(\text{brTPF}\) [10], as a star pattern request with a single triple pattern corresponds to a single triple pattern request for TPF and \(\text{brTPF}\). 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, for example, the star pattern \(S\) and the knowledge graph \(\mathcal{G}\) given in Figure 2. The star pattern-based selector function \(s_{(S,\mathcal{G})}(\mathcal{G})\) retrieves the three triples from \(\mathcal{G}\) that include \texttt{db:Jens_Brattie} as subject, as shown in Figure 2a.

Formally, SPF adapts the general definition of LDF given in [6]. Given a maximum number of distinct solution mappings that can be sent to the server \(\maxMpR\), an SPF is defined as follows:

**Definition 10 (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 \(S\) and any finite sequence \(\Omega\) of at most \(\maxMpR\) distinct solution mappings, there exists one LDF \((u, s, \Gamma, M, C) \in F\), called a Star Pattern Fragment, that has the following properties:

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

Notice that SPF, like TPF and \(\text{brTPF}\), is hypermedia and therefore contains hypermedia controls (Definition 7). An SPF can be obtained by forming a request from a star pattern and including already bound values (e.g., object values). Furthermore, by obtaining an arbitrary SPF from the server, it is possible to directly reach all other SPF’s spanning all triples in the knowledge graph and all possible star patterns.

The query semantics of a BGP over an SPF collection follows logically from the query semantics of TPF and \(\text{brTPF}\). Given that the answer to a BGP \(B\) over a knowledge graph \(\mathcal{G}\) is denoted \([B]_\mathcal{G}\), the answer to \(B\) over an SPF collection \(F\) over \(\mathcal{G}\) is determined by the following query semantics.

**Definition 11 (Query Semantics [6]).**

Given a knowledge graph \(\mathcal{G}\) and some SPF collection \(F\) over \(\mathcal{G}\), the evaluation of a BGP \(F\) over \(\mathcal{G}\), denoted by \([B]_F\), is \([B]_F = [B]_\mathcal{G}\).

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 the model independently of possibly complex query decomposition strategies that would be necessary on the client if both types of star patterns are considered, and since subject-based star patterns are much more common in real query loads [43], the rest of the paper will focus on subject-based star patterns only.

4.2. Hypermedia Controls

As previously mentioned, SPF is hypermedia, and a response to a star pattern request must thus contain controls to access other SPFs of the same collection.

The response to an SPF request consists of three fields: data, metadata and controls.

Data. The data field of an SPF response is \( \Gamma \), i.e., the result of applying \( s_{(p,\Omega)}(\mathcal{G}) \) to \( \mathcal{G} \), however, it should be paged according to Section 4.1. The metadata should thus contain pointers to other pages within the same SPF collection (i.e., next and previous pages). The data field in an SPF request consists of triples that is part of an answer to the star pattern. Triples that answer the star pattern can be grouped into resulting stars in the response do allow for faster interpretation on the client.

Metadata. The metadata field contains a set of RDF triples that are not part of the data field of the response. Given that an SPF \( f \) is obtained by the URI \( \iota \), the estimated total number of stars in the entire fragment is represented, in each page, as the triple \(<u, \text{void:triples}, \text{cnt}>\) where \( \text{cnt} \) is the cardinality estimation of the star pattern and has the type \text{xsd:integer}.

Controls. The controls of an SPF is described with the Hydra Core Vocabulary [42] as templated URIs [44]. An example of such controls, as well as an example of an SPF request applying the template obtained from the controls to the star pattern \( S \) in Figure 2a, can be seen in Listing 2. The template for an SPF request has the following fields:

- subject (line 4): Since the paper focuses on subject-based star patterns, the subject of each triple pattern is the same, and thus just has one field in the template. Accommodating for object-based star patterns can easily be done by renaming this field to vertex and adding a field describing whether the vertex is a subject or object.
- triples (line 5): The number of triple patterns in the star pattern.
- star (line 6): Grouped predicate/object values. In the case of object-based star patterns, this would instead be subject/predicate values.
- values (line 7): In the case of already bound variables, this field can be set with the same syntax as the VALUES field in a SPARQL query.

While this section contained the most important aspects of an SPF response, the full SPF specification with examples can be found on the SPF website\(^5\).

\(^5\)http://relweb.cs.aau.dk/spf

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

\[S = \{(\text{?p2, dbo:country, dbr:Norway}),
(\text{?p2, dbo:award, ?a}),
(\text{?p2, dbo:birthDate, ?bd2})\}\]

\[\mu[\text{?p2}=\text{dbr:Jens_Bratlie}
\mu[\text{?bd2}]=1856-1-17
\mu[\text{?a}]=\text{dbr:Order_of_St._Olav}\]

\[s_{(S,\Omega)}(\mathcal{G}) = \{(\text{dbr:Jens_Bratlie, dbo:country, dbr:Norway}),
(\text{dbr:Jens_Bratlie, dbo:award, dbr:Order_of_St._Olav}),
(\text{dbr:Jens_Bratlie, dbo:birthDate, 1856-1-17})\}\]
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. Find the first page of the SPF for each of \( B \)'s subqueries and select the subquery with the lowest cardinality estimation.
3. Compute the BGP result by, for each star pattern \( S \) in \( B \), incrementally updating the set of bindings by processing \( S \) on the server, and using these bindings for subsequent star patterns.
4. Compute the query result by processing all the remaining SPARQL operators in \( Q \) on the client.

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. The rest of this section focuses on Basic Graph Pattern (BGP)\(^6\) queries. Nevertheless, SPF 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 (experiments in Section 6.2 includes queries with the OPTIONAL, UNION and FILTER operators). However, this section will not go into detail on such queries since BGPs are the focus of SPF.

**Definition 12** (Subject-Based Star Decomposition). Given a BGP \( B = \{ tp_1, \ldots, tp_n \} \), the subject-based star decomposition of \( B \) is \( S(B) = \{ S_1, \ldots, S_m \} \) such that (i) \( m \leq n \), (ii) all \( S_j \in S(B) \) are subject-based star patterns (Definition 4), (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 \( tp_j \in S_j \) such that \( 1 \leq j \leq m \) and \( tp_i \in S_j \).

Using Definition 12, 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 decomposition ensures that the query is decomposed into non-overlapping star patterns. Paths in a SPARQL query then result in multiple stars that possibly consist of a single triple pattern. Processing star patterns with a single triple pattern is in line with processing the triple pattern individually.

An example of using Definition 12 to partition a BGP query \( Q \) (Listing 1) is illustrated 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.

Let \( \text{dom}(\mu) \) be a function that returns the domain of \( \mu \) (i.e., the set of variables that are bound in \( \mu \)) and \( \text{vars}(S) \) be a function that returns the set of all variables in a star pattern \( S \).

Given a control \( c \) obtained from an arbitrary fragment on the SPF server and a BGP \( B \), the general approach to processing a BGP is shown in Algorithm 1. This algorithm, while similar to the general approach

---

\(^6\)A BGP is a set of triple patterns, https://www.w3.org/TR/rdfsparql-query/#BasicGraphPatterns

---

Listing 2 Example of the controls of an SPF request and an example SPF request applying the template provided in the controls to \(<http://example.org/dbpedia#dataset>\)

```
<http://example.org/dbpedia#dataset>
  a void:Dataset , hydra:Collection ;
  void:subset [ hydra:mapping [ hydra:property rdf:subject ; hydra:variable "s" ] ;
      hydra:mapping [ hydra:property xsd:string ; hydra:variable "values" ] ;
      hydra:template "http://example.org/dbpedia/?s,triples,star,values" ] .

http://example.org/dbpedia?triples=3&star=[p1,dbo:country;ol,dbr:norway;p2,dbo:award;p3,dbo:birthDate]
```

---
for TPF (Listing 3 in [6]), has several key differences to account for due to the nature of star patterns compared to triple patterns as well as coupling bindings with the star patterns sent to the server. The approach outlined in Algorithm 1 is an illustration of how to adapt the general approach outlined by TPF to process queries over SPF recursively with a divide-and-conquer strategy. The $\maxMpR$ value (Definition 10) is therefore ignored in this algorithm. A concrete approach using iterators is shown later in this section.

First, applying the subject-based star decomposition (Definition 12) in line 4 is similar to splitting the BGP into sub-BGPs as TPF does. However, since SPF evaluates star patterns on the server, passing each individual triple pattern into sub-BGPs to process them individually is unnecessary. Instead, the entire set of star patterns is recursively evaluated (line 17), continuously expanding the set of solution mappings according to the evaluated star pattern (line 13), while sending the incrementally updated set of bindings to the server with the request (line 6). Since the set of obtained bindings can contain bindings for variables not present in the star pattern to be evaluated, and to avoid unnecessary data transfer to the server, $c(S_i, \Omega)$ on line 6 ensures that only bindings for the variables in $S_i$ are attached to the request. Second, since SPF couples previously obtained bindings with the star pattern before sending it to the server, Algorithm 1 takes an additional argument, $\Omega$, being the set of currently obtained bindings. The result of the algorithm is thus the accumulated set of bindings over each recursive call of the function (one recursive call per star pattern).

The algorithm starts by finding the first page of the corresponding SPFIs for each star pattern in the BGP (lines 5-9), and selects the star pattern with the lowest cardinality estimation (line 10). To assess the appli-
cability of the approach regardless of potentially complex join order strategies, SPF uses the same join order strategy as TPF (i.e., based on cardinality estimations provided by the server). Then, the algorithm finds all relevant bindings for the selected star pattern given the bindings \( \Omega \) through consecutive GET requests to the server using controls obtained from each page to find the next page (line 11). The bindings found for the star pattern are joined with \( \Omega \) in order to incrementally update the resulting bindings (line 13). Last, if there are any remaining star patterns within the BGP, a recursive call is made, giving as argument the remaining BGP (minus the selected star pattern) and the newly obtained bindings (line 17).

Take, as an example, the BGP \( B \) in Listing 1, and assume a control \( c \) was obtained from an SPF server giving access to DBpedia version 2016-04 [4]. Applying subject-based star decomposition (Definition 12) to \( B \) results in \( S_1 \) and \( S_2 \) from Figure 3. While the cardinalities of each individual triple pattern are large, the cardinality of \( S_1 \) is 13 and the cardinality of \( S_2 \) is 71.

When calling \( \text{evaluateBGP}(B, c) \), the first step is to obtain the first pages of the SPFs for both star patterns and select the star pattern with the lowest cardinality; in this case \( S_1 \). The 13 resulting bindings from \( S_1 \), \( \Omega_{S_1} \), are then joined with the (currently empty) set \( \Omega \). Then, the function is called recursively with \( S_1 \) removed from \( B \) (i.e., \( S_2 \)), \( \text{evaluateBGP}(S_2, c, \Omega_{S_1}) \). The bindings obtained from \( S_2 \), \( \Omega_{S_2} \), are then joined with \( \Omega_{S_1} \) and returned as the result to the BGP query.

The following presents a concrete approach to process a BGP with an SPF client that follows the general approach outlined in Algorithm 1 and uses the iterator pattern presented by Verborgh et al. [6]. A \text{RootIterator} returns an empty binding on the first call and nil on subsequent calls.

SPF provides a \text{StarPatternIterator} (Algorithm 2), similar to Listing 5 in [6], which, distinctly from the iterator provided in [6], finds a set of solution mappings rather than a single solution mapping. This is due to the fact that SPF bulks obtained bindings into groups of \( \text{maxMpR} \) bindings and forwards those to the server along with the next star patterns to obtain. A \text{StarPatternIterator} has two member variables: \( \phi \), the current SPF page, and \( \Omega_{s} \), the most recently read set of \( \text{maxMpR} \) solution mappings. If the iterator has already read one or more SPF pages, the next page will be obtained using the controls from the previous page (line 6). However, if there is no such control, or the first page has not yet been read, the iterator will retrieve the next set of at most \( \text{maxMpR} \) solution mappings (this is the case since all iterators are restricted to return sets of at most \( \text{maxMpR} \) solution mappings) from the source iterator \( I_s \) (line 8) and use those to obtain the next page (line 10). After a page has been found, the iterator will attempt to return solution mappings. Instead of finding one solution mapping, it will iterate through the current page until \( \text{maxMpR} \) solution mappings have been found, and return those as a set (lines 12-16).

Consider, for example, if the star pattern is \( S_1 \) from Figure 3 with a \( \text{maxMpR} \) of 50. In this case, since it is the first evaluated star pattern, the source iterator \( I_s \) would be a \text{RootIterator} and return an empty set of bindings. The iterator will therefore request \( S_1 \) with an empty set of bindings, and thus retrieve the 13 resulting stars. Since \( 13 < \text{maxMpR} \), all these bindings will be grouped together and returned as a set.

SPF defines a \text{BasicGraphPatternIterator} (Algorithm 3), similarly to Listing 6 in [6], which in a similar fashion to the \text{StarPatternIterator} returns a set of at most \( \text{maxMpR} \) solution mappings rather than a single solution mapping at a time. If the BGP contains no subject-based star pattern (i.e., is empty), the \text{BasicGraphPatternIterator} constructor creates a \text{RootIterator}. If, instead, the BGP consists of only a single subject-based star pattern, the constructor creates a \text{TriplePatternIterator}. Given a BGP \( B \), the \text{BasicGraphPatternIterator} creates a chained pipeline of iterators, which will incrementally call each other to obtain a set of solution mappings. It has two member variables: \( I_p \), the current iterator pipeline, and \( \Omega_{s} \), the most recently read set of \( \text{maxMpR} \) solution mappings. The \text{BasicGraphPatternIterator} creates the pipeline by, at each step, selecting the star pattern with the lowest cardinality (lines 6-10). For the selected star pattern, a \text{StarPatternIterator} is created (line 11), and for the remaining BGP a new \text{BasicGraphPatternIterator} is created (line 12). The solution mappings returned from this pipeline are then returned (line 13).

Consider again the BGP query \( B \) from Figure 3. Creating a \text{BasicGraphPatternIterator} with \( B \) as its BGP will require first to look up the cardinalities of each star pattern in \( B \). In this case, \( S_1 \) has the lowest cardinality of 13, so a \text{StarPatternIterator} \( I_s \) is created with \( S_1 \) as its star pattern and an empty solution mapping. This iterator is then used as the source iterator of the pipeline created on line 12. However, since the remaining BGP (after removing \( S_1 \)
Algorithm 2 Star pattern iterator on an SPF client

Input: A source iterator $I_1$; a star pattern $S$; a control $c$ of a $c$-specific SPF collection $F$; a maximum amount of distinct solution mapping per request $\text{maxMpR}$

Output: The next set of solution mappings $\Omega'$ such that $|\Omega'| \leq \text{maxMpR}$, or $\text{nil}$ if no such mappings are left

function $\text{StarPatternIterator}\hspace{1pt}\text{GetNext}()$

1. if $self.\phi$ has not been assigned to previously then
2.     self.\phi ← an empty page with no stars or controls;
3.     while self.\phi does not contain unread stars do
4.         if self.\phi has a control to a next page with URI $u_{\phi'}$ then
5.             self.\phi ← GET $u_{\phi'}$;
6.         else
7.             self.\phi ← GET $c(S, self.\Omega_s)$ resulting in page 1 of the SPF for $S$ and self.($\Omega_s$);
8.     end
9.     $\Omega_s$ ← $\phi$;
10.    while $|\Omega_s| < \text{maxMpR}$ and self.\phi contains unread stars do
11.        $s$ ← an unread star from self.\phi;
12.        $\mu$ ← a solution mapping such that $\text{dom}(\mu) = \text{vars}(S)$ and $\mu[S] = s$;
13.        $\Omega_s$ ← $\Omega_s \cup \{\mu\}$
14.    end
15.    return $\Omega_s$, $\Rightarrow$ self.\phi;

Algorithm 3 BGP iterator on an SPF client

Input: A source iterator $I_1$; a BGP $B$ with $|S(B)| \geq 2$; a control $c$ of a $c$-specific SPF collection $F$; a maximum amount of distinct solution mapping per request $\text{maxMpR}$

Output: The next set of solution mappings $\Omega'$ such that $|\Omega'| \leq \text{maxMpR}$, or $\text{nil}$ if no such mappings are left

function $\text{BasicGraphPatternIterator}\hspace{1pt}\text{GetNext}()$

1. if self.$I_p$ has not been assigned to previously then self.$I_p$ ← nil;
2. while self.$I_p$ = nil do
3.     self.$\Omega_s$ ← $I_p$.GetNext();
4.     if self.$\Omega_s$ = nil then return nil;
5.     for all star patterns $S_j \in S(B)$ do
6.         if self.$\Omega_s$ = nil then return nil;
7.         $\phi_i = (u_j, s_j, \Gamma_j, M'_i, C'_i) \leftarrow \text{GET}c(S_j, self.\Omega_j)$ resulting in page 1 of that SPF;
8.         cnt$_j$ ← cnt where $\langle u_j, \text{void}:\text{triples}, \text{cnt}\rangle \in M'_i$;
9.         if $\forall j : \text{cnt}_j > 0$ then
10.            $\epsilon$ ← $j$ such that $\text{cnt}_j \leq \text{cnt}_k \forall S_k \in S(B)$;
11.            $I_p \leftarrow \text{StarPatternIterator}(\text{RootIterator}(), S_j, c, \text{maxMpR})$;
12.            self.$I_p$ ← BasicGraphPatternIterator($I_p, B \setminus S_j, c, \text{maxMpR}$);
13.        end
14.     self.$I_p$.GetNext();

from $B$ only consists of a single star pattern ($S_2$), a StarPatternIterator is also created for $S_2$ as the pipeline $I_2$ ($I_2 = I_2$). This means, that when calling $I_2$.GetNext() on line 13, $I_2$ will effectively call $I_1$.GetNext(), ensuring that the 13 bindings from $S_1$ will be used to obtain the bindings for $S_1 \Rightarrow S_2$.

The QueryIterator, Algorithm 4, creates a BasicGraphPatternIterator (line 3) and iterates over the sets of bindings obtained by calling the GetNext() function on the iterator (lines 4-6). However, if a non-empty set of solution mappings has already been obtained from the iterator, the QueryIterator instead returns one of those mappings and removes it from the set (lines 7-9). Consider again the example with the BGP query $B$ from Figure 3, the QueryIterator will create a BasicGraphPatternIterator with $B$ as its BGP. This creates a pipeline with $I_2$ from above and $I_1$ as its source iterator. When calling $I_2$.GetNext() on line 5, $I_2$ will call $I_1$.GetNext(), which will find the
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 \( S \), a finite sequence of distinct solution bindings \( \Omega \), and a page number \( p \). The server processes such a request over a knowledge graph \( G \) using the following steps:

1. Given the star pattern \( S \), find the set of corresponding stars \( \lambda(S,\Omega)(G) \) (Definition 9).
2. Return an LDF page \( \phi \) that corresponds to the requested page \( p \) (LDF pages do not overlap) such that \( \phi^{\prime} \) consists of sets of matching stars.

These results are then processed by the client, which combines them with results from other star patterns in the query, thereby computing the query answer. To process star patterns, the SPF server uses similar left-deep join trees as the client. Therefore, star patterns are as efficiently processed by the SPF server as the client.

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 rarely be used with an SPF client. However, having all three methods available in the server has two advantages. First, it makes the server compatible with TPF and brTPF. 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

The SPF server was implemented using Java 8 and the SPF client using Node.js. The source code is available at http://relweb.cs.aau.dk/spf.

Server. The SPF server is implemented as an extension of the Java implementation of the TPF server\(^7\). The server implementation uses HDT [45, 46] as backend. HDT is originally proposed to process a single triple pattern over a knowledge graph efficiently. However, this implementation was extended to also be able to process the star pattern requests over the HDT backend. The SPF server uses Characteristic Sets [47] to provide cardinality estimations.

Client. The TPF Node.js client\(^8\) was extended to accommodate not only SPF requests but also brTPF requests. Thus, and in line with TPF [6] and brTPF [10], 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.

6. Experimental Evaluation

The experimental evaluation compares SPF, TPF [6], brTPF [10], and a SPARQL endpoint to investigate whether the prototype implementation of SPF, which processes SPARQL queries using subject-based star decomposition, increases query throughput by combining a lower network load with

\(^7\)https://github.com/LinkedDataFragments/Server.java
\(^8\)https://github.com/LinkedDataFragments/Client.js
a comparatively low server load. All source code, experimental setup (queries, datasets, etc.), as well as the full experimental results is provided on the SPF website\(^9\).

### 6.1. Experimental Setup

This section contains a description of the experimental setup, including a characterization of the datasets and queries used, hardware and software setup, and the measured metrics for the evaluation.

**Dataset and Queries:** The experiments were run using both synthetic datasets from the WatDiv benchmark [15] and a real-world dataset with DBpedia [4]. To test the scalability of SPF, three different sized WatDiv datasets were used. Furthermore, to test SPF in a real-world setting, the English part of DBpedia 2016-04 [4] was used. The characteristics of the used datasets can be seen in Table 1.

To study the impact of the number of star-shaped subqueries, and to stress-test the approach, the WatDiv template and query generators were used to obtain query loads with no star patterns (i.e. path queries), as well as query loads with up to 3 subject-based star patterns. Each above mentioned query load contains 6400 queries. Furthermore, the setup was tested with queries derived from the basic testing templates (BTT) provided by WatDiv\(^10\). The WatDiv basic testing templates provides 20 query templates with relatively diverse characteristics (Figure 4). For the DBpedia dataset, user-issued queries were obtained from the LSQ [16] query log. As most LSQ queries contain a single triple pattern or return an empty result set, we selected a challenging set of 24 representative queries with diverse characteristics. The complete set of queries is available on the website\(^11\). This query load, called dbpedia-lsq, was executed in random order by all clients concurrently in each configuration and include queries with the SPARQL operators FILTER, UNION and OPTIONAL. Such queries are processed by first processing the BGPs then combining them according to operators in the query.

Figure 4 shows an overview of the characteristics of the query loads [15]: Triple pattern count (#TP), join vertex count (#JV), join vertex degree (DEG), i.e., the number of triple patterns incident on a join vertex, result cardinality (#Results), and triple pattern selectivity (SelG(tp)), i.e., the average ratio of cardinality of each triple pattern to the size of the knowledge graph. High selectivity thus means that the triple patterns in a query have high cardinalities (high ratio of triples).

**Table 2** Join vertex types over query loads

<table>
<thead>
<tr>
<th>Query load</th>
<th>SS</th>
<th>SO</th>
<th>OO</th>
</tr>
</thead>
<tbody>
<tr>
<td>watdiv-1_star</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>watdiv-2_stars</td>
<td>55.38%</td>
<td>34.31%</td>
<td>10.31%</td>
</tr>
<tr>
<td>watdiv-3_stars</td>
<td>57.69%</td>
<td>30.77%</td>
<td>11.53%</td>
</tr>
<tr>
<td>watdiv-paths</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>watdiv-union</td>
<td>53.27%</td>
<td>41.27%</td>
<td>5.46%</td>
</tr>
<tr>
<td>watdiv-btt</td>
<td>56%</td>
<td>34%</td>
<td>10%</td>
</tr>
<tr>
<td>dbpedia-lsq</td>
<td>64.71%</td>
<td>26.47%</td>
<td>8.82%</td>
</tr>
</tbody>
</table>

Table 2 shows the relative distribution of the types of joins in each query load (SS is subject-subject joins, SO is subject-object or object-subject joins, and OO is object-object joins). The watdiv-union query load that contains the combined queries from watdiv-1_star, watdiv-2_stars, watdiv-3_stars and watdiv-paths was added as well. All query loads (except watdiv-1_star) include queries with subject-object joins. All queries in the watdiv-paths query load contain only subject-object joins.

**Experimental Configuration:** To assess how each approach performs under different loads, experiments were run 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 \(244 \times 2^i\) queries are executed and at most \(2^i\) queries are executed concurrently.

---

\(^9\)http://relweb.cs.aau.dk/spf  
\(^10\)https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml  
\(^11\)http://relweb.cs.aau.dk/spf
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. For the watdiv-1_star, watdiv-2_stars, watdiv-3_stars, and watdiv-paths query loads, 50 distinct queries were executed by each client in the configuration (24 distinct queries for watdiv-btt). For dbpedia-lsq, the 24 queries in the query load were run on all clients in the configuration in a distinct, random order on each client.

**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:**

- **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 over all clients.
- **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:** Virtuoso Open-Source version 7.2.5 was used 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:

- 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.
We chose Virtuoso since Verborgh et al. [6] showed that is the endpoint that performed best with respect to high throughput and low CPU usage. Only one LDF server implementation was used, which was a combined TPF, brTPF, and SPF server. The LDF page size was, throughout the experiments, set to 100 results, and the maximum number of elements in $\Omega$ was set to 30 for both brTPF and SPF, i.e., they can send up to 30 bindings with each request. The original TPF and brTPF Node.js clients were used. The timeout was set to 600 seconds, i.e., 10 minutes.

**Experimental Results:** The objective is to assess whether SPF can execute SPARQL queries containing star patterns more efficiently in terms of response time and network traffic without incurring too much additional load on the server. Furthermore, the experiments investigate if SPF is, in the case of path queries, still as good in terms of performance as brTPF.

The SPARQL endpoint became unresponsive (i.e., all queries timed out) for certain configurations due to high server load.

### 6.2. Performance Under Load

Figure 5 shows the throughput, CPU load, and number of timeouts of the four approaches for different numbers of concurrent clients for waterdiv-union over each WatDiv dataset and for dbpedia-lsq over dbpedia, including queries that timed out. Figures 5a-5d include queries that timed out for any approach. This means that the figures include the entire execution time for the approaches that did not time out (e.g., SPF) while they include only partial execution times for the approaches that did time out (e.g., TPF).

Note that for waterdiv-union over waterdiv-100M and waterdiv-1000M, the throughput increases for SPF, brTPF, and TPF 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 does not significantly affect the execution time until after the four concurrent clients. Although the throughput (Figures 5a-5d) of all the interfaces decreases as the number of concurrent clients increases, SPF maintains between 4-7 times higher throughput compared to brTPF for the WatDiv datasets, and 96 times higher throughput for dbpedia.

The relative gain in performance provided by SPF is slightly lower for waterdiv-1000M compared to waterdiv-100M (4 times higher throughput compared to 7 times higher throughput for 128 clients with respect to brTPF). The same tendencies are also supported by the throughput, CPU load, and number of timeouts for the remaining query loads (see Appendix A, Figures 8-10). Intuitively, larger datasets that share characteristics (e.g., larger WatDiv datasets) should mean that more intermediate results for each star pattern have to be processed by the server. However, this is also explained by the higher number of timeouts that especially TPF and brTPF incurs, which mean that more partial execution times are included than for SPF. Moreover, DBpedia is roughly the same size in terms of number of triples as waterdiv-1000M and presents a much larger relative gain in performance for SPF (96 times for 128 clients in comparison to brTPF). Section 6.4 shows that the query loads with larger join vertex degrees (Figure 4c) and higher selectivity (Figure 4e), e.g., waterdiv-3_stars, result in lower relative gain in performance for SPF. Nevertheless, SPF has higher throughput even for the largest WatDiv dataset.

Even though SPF servers compute star patterns, Figures 5e-5g show that SPF only incurs up to 1.08 times as much CPU load in comparison to brTPF for 128 clients. The CPU load is relatively similar across all WatDiv datasets. This is due to the fact that each client makes at most one server request at a time, meaning at most 128 requests at a time have to be processed concurrently. The CPU load, for all configurations, is below 100% for SPF, brTPF, and TPF. While SPF has a slightly higher CPU load than brTPF even under high load (1.08 times higher for 128 clients), the relative increase in throughput for SPF remains the same for increased numbers of clients for all WatDiv datasets. Therefore, the slight difference in CPU load is small enough to not impact performance significantly.

Due to the more efficient query processing, SPF has significantly fewer timeouts than all other approaches (Figure 5h-5j). Comparing SPF to brTPF, it is clear that the number of timeouts rises faster for brTPF than for SPF for the larger WatDiv datasets. As the dataset grows larger, brTPF becomes more similar to TPF in

---

13The combined server implementation is available at [http://relweb.cs.aau.dk/spf](http://relweb.cs.aau.dk/spf).
14https://github.com/LinkedDataFragments/Client.js
15http://olafhartig.de/brTPF-ODBASE2016/
the number of timeouts, while it stays relatively low for SPF. This is due to the increased sizes of intermediate results 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 that only SPF can benefit from.

Comparing the throughput to the number of timeouts over the different sized datasets, it is not immediately clear why the relative gain in throughput that SPF provides decreases, while the ratio of timed out queries gets relatively better for SPF compared to brTPF for the larger datasets. However, when look-
at the individual results, this becomes more clear. The increased size in the dataset means that it takes longer time to process each star pattern on the server (since each triple pattern has more intermediate bindings). This is mostly mitigated by the limited amount of intermediate bindings for each star pattern. However, for queries high selectivity this can cause processing subqueries on the server to have slightly lower performance. Though, due to the limited server requests, SPF is able to process more queries within the timeout. This also helps explaining why SPF has such an improved performance for DBpedia compared to watdiv-1000M (Figure 5c-5d); brTPF and TPF actually have quite similar throughputs for both datasets given that they are roughly the same size. However, Figure 5d shows that SPF increases throughput by up to two orders of magnitude for DBpedia (SPF has 96 times higher throughput for 128 clients). The generally lower number of results (Figure 4d) and lower selectivity (Figure 4e) for dbpedia-lsq means that SPF is able to decrease the number of intermediate results more significantly compared to brTPF and thus improve the throughput.

The endpoint is the best performing interface for only few concurrent clients and a small dataset. However, its performance deteriorates much faster when the number of concurrent clients increases. SPF, TPF, and brTPF are able to handle the increased load more efficiently than the endpoint (Figures 5a and 5b). This is in line with the experiments shown in [6] and shows that SPF seems to be a suitable alternative to handle large query loads.

6.3. Network Traffic

As previously highlighted, this section assesses whether or not 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, it was expected that utilizing such subqueries results in fewer requests to the server and less data transfer (i.e., intermediate results) between server and client.

Figures 6a-6c show NRS for the experiments with 64 clients (since 64 was the highest number of clients all approaches were able to finish for watdiv-10M) over all WatDiv datasets for all WatDiv query loads.
Fig. 7. QET (in ms) and QRT (in ms) with 64 clients including queries that timed out for any approach.

Similarly, since the SPF server processes larger parts of the queries, fewer intermediate results are returned to the clients, resulting in a lower NTB (Figures 6d-6f). Similar to NRS, NTB is significantly lower for SPF in comparison to both TPF and brTPF throughout all query loads except watdiv-paths, where the results are similar for SPF and brTPF. This shows that compared to TPF and brTPF, SPF significantly reduces the network traffic. 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. However, as shown in Figure 5, this results in higher CPU usage on the server and lower throughput under load overall. Given that SPF clearly has a lower network usage, SPF seems to be a suitable alternative to handle large query loads.

6.4. Impact of Query Pattern

Figure 7 shows QET and QRT for all WatDiv query loads over all WatDiv datasets in the configuration with 64 concurrent clients, and includes queries that timed out. For queries with star patterns, it is clear that SPF has better performance than both TPF and brTPF.
over all configurations. The difference between SPF and other interfaces is more significant for the 1-star query load. This is expected since fewer requests are made for these queries. In fact, some queries in the watdiv-1_star query load can be answered with just a single call to the server. As shown in Figure 7, SPF outperforms other interfaces more significantly for the watdiv-1_star and watdiv-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 more requests to the server for these queries, whereas SPF still only makes one request to the server per 100 bindings to each star pattern (cf. the page size was set to 100). For watdiv-3_stars over watdiv-1000M, while the mean QET and QRT is lower for SPF than brTPF and TPF, few queries have a slightly higher QET and QRT. This supports the earlier point that for the larger datasets and with queries with high selectivity, each star pattern request takes a little longer to process. This means that queries with more star patterns and higher selectivity are more heavily affected. Note also that while some queries in the watdiv-3_stars query load are affected by this, SPF was able to finish many more queries (since fewer queries timed out, Figure 5), and thus SPF still outperforms brTPF et al. (TPF).

For queries with no star patterns, it was expected that SPF does not have a worse performance than brTPF. This is in line with the experimental results, as SPF has similar performance as brTPF for watdiv-paths and better performance for all other query loads. Figure 7 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 the number of client increases.

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 joins in the query are 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.

6.5. Summary

Overall, the 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. For queries without star patterns, SPF still performs as good as brTPF, both in terms of execution time and network traffic. While SPF does have slightly higher CPU load on the server (SPF increases server usage by up to 1.08 times compared to brTPF and 1.18 times compared to TPF), it is still significantly more efficient than TPF and brTPF in the presence of high load (SPF increases throughput by up to 96 times compared to brTPF and 136 times compared to TPF for 128 clients). This is true both for large-scale synthetic datasets and real-world datasets, and suggests that SPF is able to combine a lower network load with a higher query throughput at a comparatively low server load.

7. Conclusions

In this paper, Star Pattern Fragments (SPF), a new RDF interface that exploits a different tradeoff for distribution of the workload between the server and client, was presented. The SPF client processes queries by processing SPARQL operators and decomposing each BGP into star-shaped subqueries and sending these subqueries, along with intermediate bindings, to the server. An SPF server that is able to answer HTTP requests containing star patterns was implemented as well as an SPF client that is able to answer SPARQL queries. The 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 by a factor of up to 96 times compared to brTPF and up to 137 times compared to TPF. The evaluation also demonstrates that SPF increases the overall performance while only increasing the CPU load on the server by a factor of 1.08 compared to brTPF and 1.18 compared to TPF when 128 clients issue queries concurrently.

Future Work

While a novel distribution of the workload between the client and server was presented, SPF presents an
opportunity to explore different ways to utilize this distribution of workload. While relatively few queries include many object-based star patterns (Table 2), investigating the tradeoff between including such star patterns on the server and the expense of a more complex query decomposition strategy on the client (and overhead of such a strategy) is part of the future work for SPF. Furthermore, in some cases performance could be increased by sending star patterns to the server that do not contain all triple patterns for the given star pattern, in order to ensure the optimal join order on the triple pattern level. In that sense, it could also be interesting to assess whether other query decomposition techniques, not focused on star patterns, could provide any benefits. Furthermore, it could be interesting to include an SPF-specific cache on the server and to assess its impact on the performance of SPF. Lastly, it could be interesting to integrate SPF into systems, such as [11, 33] that rely on the different strengths of different RDF interfaces to process SPARQL queries more efficiently.

Acknowledgments. This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, Aalborg University’s Talent Programme, and the Poul Due Jensen Foundation.

Appendix A. Additional Experimental Results

This appendix contains additional experimental results for query loads left out in Section 6. Figure 8 shows the throughput for watdiv-1_star, watdiv-2_stars, watdiv-3_stars, and watdiv-paths over all WatDiv datasets for all configurations, and Figure 9 shows the number of timeouts for the same configurations. Figure 10 shows throughput, CPU load and number of timeouts for watdiv-btt over all WatDiv datasets and all configurations, as well as the CPU load and number of timeouts for dbpedia-lsq. Last, Figure 11 shows the network usage including queries that timed out.

References

Fig. 8. Throughput (# queries/m) for `watdiv-1_star` over `watdiv10M (log)`

(a) Throughput for `watdiv-1_star` over `watdiv10M (log)`

(b) Throughput for `watdiv-1_star` over `watdiv100M (log)`

(c) Throughput for `watdiv-1_star` over `watdiv1000M (log)`

(d) Throughput for `watdiv-2_stars` over `watdiv10M (log)`

(e) Throughput for `watdiv-2_stars` over `watdiv100M (log)`

(f) Throughput for `watdiv-2_stars` over `watdiv1000M (log)`

(g) Throughput for `watdiv-3_stars` over `watdiv10M (log)`

(h) Throughput for `watdiv-3_stars` over `watdiv100M (log)`

(i) Throughput for `watdiv-3_stars` over `watdiv1000M (log)`

(j) Throughput for `watdiv-paths` over `watdiv10M (log)`

(k) Throughput for `watdiv-paths` over `watdiv100M (log)`

(l) Throughput for `watdiv-paths` over `watdiv1000M (log)`

Fig. 8. Throughput (# queries/m) for `watdiv-1_star`, `watdiv-2_stars`, `watdiv-3_stars` and `watdiv-paths` over the different WatDiv datasets. Includes queries that timed out.
Fig. 9. Number of timeouts for \texttt{watdiv-1\_star}, \texttt{watdiv-2\_stars}, \texttt{watdiv-3\_stars} and \texttt{watdiv-paths} over the different WatDiv datasets.
Fig. 10. Throughput, number of timeouts and CPU load for \textit{watdiv-btt} over the different WatDiv datasets and \textit{dbpedia-lsq} over \textit{dbpedia}.
Fig. 11. NRS and NTB with 64 clients including queries that timed out for any approach.