Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web

Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert

In: Journal of Web Semantics, 37–38, 184–206, 2016.

http://linkeddatafragments.org/publications/jws2016.pdf

Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web

Ruben Verborgh[a,*], Miel Vander Sande[a], Olaf Hartig[b], Joachim Van Herwegen[a], Laurens De Vocht[a], Ben De Meester[a], Gerald Haesendonck[a], Pieter Colpaert[a]

[a] *Ghent University – iMinds, Sint-Pietersnieuwstraat, 9000 Gent, Belgium*
[b] *Hasso Plattner Institute, University of Potsdam, Prof.-Dr.-Helmert-Straße 2-3, 14482 Potsdam, Germany*

Abstract

Billions of Linked Data triples exist in thousands of RDF knowledge graphs on the Web, but few of those graphs can be queried live from Web applications. Only a limited number of knowledge graphs are available in a queryable interface, and existing interfaces can be expensive to host at high availability. To mitigate this shortage of live queryable Linked Data, we designed a low-cost Triple Pattern Fragments interface for servers, and a client-side algorithm that evaluates SPARQL queries against this interface. This article describes the Linked Data Fragments framework to analyze Web interfaces to Linked Data and uses this framework as a basis to define Triple Pattern Fragments. We describe client-side querying for single knowledge graphs and federations thereof. Our evaluation verifies that this technique reduces server load and increases caching effectiveness, which leads to lower costs to maintain high server availability. These benefits come at the expense of increased bandwidth and slower, but more stable query execution times. These results substantiate the claim that lightweight interfaces can lower the cost for knowledge publishers compared to more expressive endpoints, while enabling applications to query the publishers' data with the necessary reliability.

Keywords: Linked Data, Linked Data Fragments, querying, SPARQL

This is the revised manuscript of an article that has been accepted for publication in *Journal of Web Semantics*.
DOI: doi:10.1016/j.websem.2016.03.003
URL: `http://www.sciencedirect.com/science/article/pii/S1570826816000214`

This article is the main reference for Linked Data Fragments and Triple Pattern Fragments.
Please cite as follows:
Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. Journal of Web Semantics. (2016). DOI: doi:10.1016/j.websem.2016.03.003

*Corresponding author
Email address: `ruben.verborgh@ugent.be` (Ruben Verborgh)

1. Introduction

Before the Linked Data initiative [1], the Semantic Web suffered from a *chicken-and-egg* situation: there were no applications because there was no data, and there was no data because no applications were using it. Fortunately, Tim Berners-Lee's credo "Raw data now" has caught on, and now more knowledge graphs exist as Linked Data than ever before [2]. Statistics from the LODstats project [3] indicate that, as of March 2015, there are over 88 billion Linked Data triples distributed over 9,960 knowledge graphs.[1] Thus, the ball is now back in the Semantic Web's court: given this tremendous amount of data in various domains, we should be able to build the envisioned intelligent applications [4].

Unfortunately, the availability of live queryable knowledge graphs on the Web still appears to be low. With "live queryable", we mean Linked Data that can be queried without first downloading the entire knowledge graph. With "low availability", we mean the two-sided problem the Semantic Web is currently facing: *i)* the majority of knowledge graphs is not published in queryable form [3] and *ii)* knowledge graphs that *are* published in a public SPARQL endpoint suffer from frequent downtime [5]. This unavailability becomes all the more problematic if we consider queries over multiple, *distributed* knowledge graphs. It is therefore understandable that many publishers choose the safe option, avoiding the responsibility of hosting a SPARQL endpoint by offering a data dump instead. Yet, this does not bring us closer to the Semantic *Web* because such data dumps need to be downloaded and stored locally so that the actual querying can happen *offline*. Furthermore, their consumption is only possible on sufficiently powerful machines—not on mobile devices, whose popularity continues to increase—and requires a technical background to set up. A significant amount of Linked Data knowledge graphs is therefore not reliably queryable, nor easily accessible, on the Web.

If we want Semantic Web applications on top of live knowledge graphs to become a reality, we must reconsider our options regarding Web-scale publication of Linked Data. Between the two extremes of data dumps and SPARQL endpoints lies a whole spectrum of possible Web interfaces, which has remained largely unexplored. The challenge is to methodically analyze the benefits and drawbacks an interface brings for clients and servers. In particular, we aim for solutions with minimal server complexity (minimizing the *cost* for data publishers) while still enabling live querying (maximizing the *utility* for Semantic Web applications).

[1] http://stats.lod2.eu/

In this article, we present and extend our ongoing work on *Linked Data Fragments* [6], a framework to analyze Linked Data publication interfaces on the Web, and *Triple Pattern Fragments* [7], a low-cost interface to triples. Novel contributions include in particular:

- an extended formalization (Sections 4 and 5) that details the response format and its considerations (Section 5.3);
- a detailed discussion of Triple Pattern Fragments publication and their relationship to existing interfaces (Sections 4.3 and 5.4);
- an extension of the query execution algorithm toward other SPARQL constructs (Section 6.3) and toward a federation of interfaces (Section 6.4);
- additional experimental results that
  - measure queries on the real-world knowledge graph DBpedia, revealing that the type of queries has a stronger influence than knowledge graph size (Section 7.2);
  - assess the impact of different serialization formats, which reveals a limited gain for specialized binary formats, likely due to the small page size (Section 7.3);
  - extend the application from one knowledge graph to multiple knowledge graphs, where we find that our proposed solution performs well for certain types of queries with regard to precision, recall, and/or execution time, but less so for other types (Section 7.4).

The remainder of this paper is structured as follows. Section 2 derives the research questions and hypotheses underlying this work, based on quantifiable characteristics for Web APIs. In Section 3, we describe existing solutions and highlight their advantages and disadvantages, focusing especially on the potential for live query execution. Section 4 introduces the Linked Data Fragments conceptual framework, followed by the definition of the Triple Pattern Fragments interface in Section 5. Section 6 details a client-side query algorithm for basic graph patterns, and extends it toward both general SPARQL queries and federations of knowledge graphs. We present an experimental evaluation in Section 7. Finally, Section 8 concludes the article with lessons learned and starting points for further research.

2. Problem Statement

For querying knowledge graphs on the Web, there exist interfaces with powerful query capabilities (e.g., SPARQL endpoints) and interfaces with low server-side CPU cost (e.g., data dumps). The task of query evaluation currently happens either fully on the server side, or fully on the client side. However, there is a lack of options that balance these and other trade-offs in different ways. In this article, we aim to define and analyze an interface that distributes the load of query evaluation between a client and a server. To this end, we first define different characteristics relevant to Web Application Programming Interfaces (Web APIs) in Section 2.1. Using these characteristics, we then formulate a research question and associated hypotheses in Section 2.2.

2.1. Characteristics for Web APIs

A crucial architectural choice is the definition of the interface, which in turn reflects on characteristics such as *performance*, *cost*, *cache reuse*, and *bandwidth*. Each of these can be considered from the perspective of either servers in general, or clients performing a specific task [8].

performance Performance is the rate at which tasks can be completed. For the server, performance can be measured as the number of requests it can handle per time interval, i.e., the inverse of the average request processing time. However, we need to take into account that one API might offer more granular requests than another. Therefore, to solve the same task, a client might require a different number of requests on different APIs.

cost Cost is the amount of resources a request consumes. For the server, the resources typically involve CPU, RAM, and IO consumption. From the client perspective, the cost consists of processing one or multiple server responses into the desired result for a given task.

cache reuse Cache reuse is the ratio of items that are requested multiple times from the cache versus the number of items it stores. The server might offer responses that can be reused by multiple clients, which can then subsequently be served from a cache instead, saving on server cost.

bandwidth The bandwidth for the client is the product of the number of retrieved responses and the average response size (ignoring the relatively small request size). This is the same for the server, except that a portion of the responses might be cached and thus involve cache bandwidth instead of server bandwidth.

Given a task T a client needs to complete, two Web APIs I and I' might exhibit different behavior. For instance, a client might be able to complete T using a single large operation o with server-side cost c against I, whereas n smaller operations $o'_1, \ldots, o'_n$ with costs $c'_1, \ldots, c'_n$ might be needed in the case of I'. We assume here that the cost c'_i of each individual smaller operation o'_i is less than the cost of the large operation, c, but the total server cost for I' is $\sum_1^n c'_i$, which may be greater than c. If, however, some of these n smaller operations are cacheable, multiple clients executing tasks similar to T could reuse already generated responses from a cache, lowering the total number of requests—and thus the cost—for the server. Which of these factors dominates depends on the number of clients, the cost per request, the cache reuse ratio, the similarity of tasks, and other factors. In general, if costs increase to a certain level, a server might become fully occupied and unable to fulfill new incoming requests, and hence start a period of unavailability. Due to its impact on availability, it is thus important to examine how choosing a specific interface influences the cost for the server.

Lastly, we introduce an important practical characteristic. When designing an API, we need to consider the restrictions the interface places on clients. For this reason, we also assess the overhead for clients, which we express as follows:

efficiency Efficiency for the client is the fraction of data retrieved from the server during the execution of a task over the amount of data required to execute that task.

2.2. Balancing trade-offs for publishing and querying Linked Data on the Web

Our goal is to enable reliable applications on top of knowledge graphs on the Web. This requires Linked Data that is *i)* available for a high percentage of time *ii)* in a queryable form. Given that the interface is the aspect of cost we have most control over, and that the interface directly determines queryability, we define our main research question as follows.

Q1 To what extent can a restricted Web API for knowledge graphs provide live querying with low server cost?

Note that we are not looking for an "optimal" interface, as such an optimum (if it exists) is likely application and use-case dependent. Instead, we specifically aim to investigate the following question.

Q2 What trade-offs are involved in clients' execution of SPARQL queries over knowledge graphs accessible through Web APIs whose expressiveness is limited to a small subset of that of SPARQL?

Here, we consider the notion of the *expressiveness* of a Web API as the (possibly infinite) set of different query expressions a client can send to the interface. Note that, in comparison to other typical Web APIs, SPARQL endpoints offer an interface with a relatively high expressiveness; it goes beyond the expressive power of the Relational Algebra (which has been shown to have an expressive power equivalent to a core fragment of SPARQL [9]). Hence, the answer to question Q2 will indicate whether it makes sense to investigate other means of publishing knowledge graphs besides SPARQL endpoints and data dumps.

Concretely, we propose an interface I, for which we formulate the following hypotheses. First, we expect the overall server cost to be lower, as we improve the usage of CPU, caching and concurrency.

H1 In comparison to the state-of-the-art in single-machine SPARQL endpoints, I reduces the server-side costs to publish knowledge graphs in a queryable form.

- H1.1 The CPU usage is smaller and increases more slowly with an increasing number of clients that access the interface concurrently.
- H1.2 The number of requests answered by an HTTP proxy cache increases with the number of clients.
- H1.3 The average response time is less affected by increasing client numbers.

Next, we predict that this approach is sufficiently fast for scenarios with real-world queries, i.e., randomly selected queries from query logs from knowledge graphs such as DBpedia. We set 1 second as a target response time, which is generally accepted as maximum waiting time without additional feedback to the user [10, 11].

H2 The majority of typical real-world queries execute over I in less then 1 second.

Then, we study the required bandwidth of client and server communication by experimenting with different RDF serialization formats. A lower response size per request could reduce transfer time or advise us on optimal page sizes, likely improving the overall query execution time.

H3 Serialization formats that result in a lower response size of I, compared to N-Triples, decrease query execution times.

We anticipate that an approach with a low-cost server-side interface will form an efficient architecture for query evaluation over a federation of interfaces. We use the term "federation" in the context of Sheth and Larson, who define a federated database system as "a collection of cooperating database systems that are autonomous and possibly heterogeneous" [12]. With "query evaluation over a federation of interfaces", we mean that a whole of multiple interfaces is considered for the evaluation of a query, that is, the result of evaluating a given query over these multiple interfaces is correct if and only if it is identical to the result that would be obtained when the query would be evaluated (under set semantics) over one interface that combines the data of all considered interfaces.

H4 When a client evaluates queries over a federation of interfaces of type I under public network latency, performance is similar to that of state-of-the-art SPARQL query federation frameworks.

- H4.1 For the majority of queries, result completeness is similar to that of the state-of-the-art.
- H4.2 For the majority of queries, average query execution time is of the same magnitude compared to the state of the art.

We test these hypotheses by the definition of a concrete interface (Sections 4 and 5), an associated query execution algorithm (Section 6), and an experimental evaluation thereof (Section 7).

3. Related Work

In this section, we first give a brief introduction to Web APIs in general. This is followed by detailed discussions for three common interface types to RDF triples [13]: data dumps, SPARQL endpoints, and Linked Data documents. We finish with an overview of miscellaneous approaches. For each interface type, we specifically address the possibilities for query execution by clients and/or servers using the SPARQL language [14], which is the W3C standard to express declarative queries over collections of RDF triples.

3.1. Web APIs

The notions of "Web service" and "Web API" (Web Application Programming Interface) have been used in various contexts [15]. In the broadest sense, a Web API is any application that communicates through the HTTP protocol. Regular websites can be conceived of as APIs for humans, where the interface is determined by the HTML pages offered by a server. In the stricter sense, the term "Web API" mostly refers to interfaces that have been designed for an automated, machine-based consumption of Web content. Such an interface can range from providing access to a single resource to an entire dynamic network of interconnected resources.

In particular, we distinguish two architectural styles for Web APIs. Some APIs follow a remote-procedure calling (RPC) style, in which HTTP simply acts as a tunneling mechanism for method invocation. Other APIs use the Representational State Transfer (REST) architectural style [16], in which hyperlinked resources (as opposed to actions) form the interface building blocks. In contrast to RPC, REST captures the design of the human Web: we browse the Web by clicking links and forms (respectively `<a>` and `<form>` in HTML). Therefore, with REST Web APIs, machines similarly use such hypermedia controls to navigate from one resource to another [17, 18]. A *hypermedia control* is a declarative construct that informs clients of a hypermedia interface of possible application and/or session state changes in their interaction with a server, and explains them how to effectuate such changes. The Hydra Core Vocabulary [19] allows Web APIs to describe the equivalent of links and forms in RDF. The benefit of REST APIs is that, like websites, they are self-describing: once the basic mechanisms (HTML, Hydra, ...) are implemented, no external documentation is necessary to browse and consume the API or website.

Architectural styles have an influence on the API characteristics discussed in Section 2.1. For instance, RPC-based APIs tend to involve more individualized and uncached requests, analogous to the one-on-one communication of software APIs and system APIs on local machines. REST APIs tend to offer reusable resources that are consumed by multiple clients; for instance, a personalized webpage might still reuse images and other assets from a cache. Many individual variations are however possible, depending on the granularity of operations/resources in the API.

3.2. Data dumps

The most straightforward way to publish Linked Data knowledge graphs is to upload an archive containing one or more files in an RDF format such as N-Triples or N-Quads. Clients download an archive through HTTP, extract its files, and process them as they see fit.

This solution requires only a low-complexity file server, but potentially a lot of bandwidth and client processing cost if archives are large. The main advantage of data dumps is their universality: clients obtain the entire knowledge graph and can ingest it into an access point of choice, optimized for the kind of task they want to perform. Depending on the use case and total graph size, this also means the cost for clients to use the data is high, and thus possibly out of reach for a significant number of consumers. Furthermore, the efficiency of the approach is likely limited, unless clients need to execute many data-intensive tasks over a knowledge graph. In an extreme case, a client only needs a few triples, but has to download the entire dump to retrieve these triples. Finally, if (part of) the data becomes outdated, the client has to restart downloading and processing entirely. The above arguments question the Web-appropriateness of data dumps as only access mechanism. After all, human consumers of Web content do not need to download an entire website before being able to read one or more webpages—so why should machine clients have to?

Some initiatives have aimed to facilitate the usage of data dumps. For instance, the LOD Laundromat [20] harvests data dumps from the Web in various RDF format, cleans up quality issues relating to the serialization, and republishes the dumps in standards-conform formats. However, this effort still does not tackle the issue that data is not *live queryable*: clients have to download data dumps and ingest them in a SPARQL-aware system before SPARQL queries can be executed.

3.3. SPARQL endpoints

From the viewpoint of the client, the easiest way to execute a SPARQL query is to dispatch this query in its entirety to the server that hosts the corresponding knowledge graph. The SPARQL protocol [21] standardizes this interaction: clients send SPARQL queries through a specific HTTP interface, and the server attempts to execute these queries and responds with their results. Many triple stores, such as Virtuoso [22] and Jena TDB [23], offer a SPARQL interface. Exposing such a query interface on the public Web contrasts with most other Web APIs, whose main purpose is to expose a *less* powerful interface than the underlying database. While enabling clients to send arbitrary SPARQL queries leads to low bandwidth consumption and low client cost, the processing of individual requests is potentially very expensive in terms of server CPU time and memory consumption.

In fact, it has been shown that the evaluation problem for SPARQL is PSPACE-complete [24].

A 2013 survey revealed that the majority of public SPARQL endpoints had an uptime of less than 95% [5]. In other words, the average endpoint is down for more than 1.5 days each month. This makes developing and running live applications on top of public endpoints an unrealistic endeavor in practice. A practical use case for public SPARQL endpoints as identified by a developer of the popular Virtuoso triple store is to provide such endpoints *"for lookups and discovery; sort of a dataset demo"* [25]. If reliable access is needed, a common practice is to download a data dump and host a local endpoint; the scenario then becomes that of Section 3.2, inheriting the same drawbacks.

A characteristic of SPARQL endpoints is that, in addition to the unpredictability of the total number of users and requests, the processing cost per request can vary significantly because of the relatively high expressive power of SPARQL (with different fragments of SPARQL having a different computational complexity [24]). In most other Web APIs, the cost per request is bounded more strongly, because the interface is designed to restrict the query capabilities. Regardless of the absolute value of this cost, the high variations in the individual factors of the product make it difficult to correctly provision an infrastructure for a public SPARQL endpoint. While high availability is possible, it is potentially expensive and not within reach for most knowledge graph publishers, who rather use a more reliable but less expressive interface such as a data dump, rather than facing frequent unavailability.[2]

Another characteristic of SPARQL endpoints is that, due to the relatively high expressive power of SPARQL, clients perform very individualized requests. As a consequence, caching such requests only allows for limited reuse, since other clients likely have different requests.

Note that in the case of *private* SPARQL endpoints, the number of users, requests, and the cost per request are under stricter control, and cache reuse is possibly higher, allowing for more efficient provisioning. Therefore, highly available endpoints in enterprise contexts can be viable and cost-effective. On the public Web, however, availability of SPARQL endpoints remains a two-sided problem. Because of the high cost, few knowledge graphs are available through a SPARQL endpoint—and some of the endpoints that exist are sometimes unaccessible. For instance, in March 2015, the LODstats project [3] listed 10,059 dataset descriptions, out of which only 464 referred to a SPARQL endpoint, and only 230 of these endpoints responded within 10 seconds with an HTTP `200 OK` status code to our `GET` request for their base URL.

[2] We note that some systems allow publishers to expose only a subset of all SPARQL queries; for instance, by placing restrictions on the allowed query execution times. However, even with such restrictions, the availability of public SPARQL endpoints remains low [5].

3.4. Linked Data documents

A different category of approaches follows the Linked Data principles by Berners-Lee [1]. RDF triples are divided across several Linked Data documents, each of which contains triples related to a specific entity. Typically, the contents of each document d_e about an entity with URI e are chosen such that d_e contains (a part of) the knowledge graph's triples of the form (e, p, o) and (s, p, e). For instance, the URI `http://dbpedia.org/resource/Nikola_Tesla` denotes the inventor Nikola Tesla, and looking up this URI leads to a document with triples in which the URI is the subject or the object. The possibility of performing such a lookup is a crucial property of Linked Data: if a client does not know what entity an URI represents, it can find information by performing an HTTP `GET` request.

The server performance for responses can be high, and their cost is low, since the required data lookups to generate each document are light. Furthermore, the set of resources per knowledge graph is finite, and resources can be reused by many clients, leading to high cache reuse. These factors allow high availability of the interface at low cost.

Several approaches exist to execute queries over Linked Data documents, as surveyed by Hartig [26]. One family of approaches uses pre-populated index structures [27] and another focuses on live exploration by a traversal-based query execution [28]. Typically, such query approaches have longer query execution times than SPARQL endpoints, but—unlike data dumps—allow for *live* querying. The required bandwidth is generally smaller than that of data dumps, but the efficiency can still be low depending on the type of query. Furthermore, completeness with regard to a knowledge graph cannot be guaranteed, and certain queries are difficult or impossible to evaluate without an index [26]. In particular, queries for patterns with unbound subjects (e.g., `?s foaf:made <o>`) pose problems, since Linked Data documents are by definition subject-centric.

3.5. Other specialized Web APIs to Linked Data

Finally, several other HTTP interfaces for triples have been designed. The Linked Data Platform [29] is a W3C recommendation for a read/write Linked Data HTTP interface. It details several concepts that extend beyond

the Linked Data principles, such as containers and write access. The API has been designed primarily for consistent read/write access to Linked Data resources, not to enable reliable and/or efficient query execution. Another read/write interface is the SPARQL Graph Store Protocol [30], which describes HTTP operations to manage RDF graphs through SPARQL queries.

Additionally, several other fine-grained HTTP interfaces for triples have been proposed, such as the Linked Data API [31] and Restpark [32]. Some of them aim to bridge the gap between the SPARQL protocol and the REST architectural style underlying the Web [33]. However, none of these proposals are widely used and no query engines for them are implemented to date.

4. Linked Data Fragments

4.1. Concept and context

To understand and analyze existing Web APIs for RDF-based knowledge graphs, we need a uniform view on them. To derive such a view, we look at what all such APIs have in common: in one way or another, they publish specific *fragments* of a knowledge graph. A SPARQL endpoint response, a Linked Data document, and a data dump each offer specific parts of all triples of a given knowledge graph. Rather than presenting these APIs as fully distinct approaches, we uniformly call the response to each request against such an API a *Linked Data Fragment (LDF)*. Informally, an LDF of an RDF-based knowledge graph is a resource consisting of a specific subset of RDF triples of this graph, potentially combined with metadata, and hypermedia controls to retrieve related LDFs. Each LDF thus consists of the following three (possibly empty) parts.

data: RDF triples obtained from the knowledge graph;

metadata: RDF triples that describe the data triples;

controls: links and forms to retrieve other LDFs of the same or other knowledge graphs.

As Figure 1 illustrates, different types of LDFs differ in the granularity of the selection mechanism through which they expose the underlying knowledge graph. Depending on this granularity, the workload to execute queries is divided differently between clients and servers. The key to efficient and reliable Web querying is to find fragments that strike a balance between client and server cost. For instance, we expect a relationship between the per-request cost of an interface and the granularity of its selection mechanism.

In the remainder of this section we formalize the notion of a Linked Data Fragment (Section 4.2) and show how existing interfaces can be analyzed in terms of our conceptual framework of such fragments (Section 4.3).

4.2. Formal definition

As a basis of our formalization, we use the following concepts of the RDF data model [13]. We write $\mathcal{U}$, $\mathcal{B}$, and $\mathcal{L}$, to denote the disjoint, infinite sets of URIs, blank nodes, and literals, respectively. Then, the (infinite) set of all *RDF triples* is $\mathcal{T} = (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$.

For the sake of a more straightforward formalization, we assume without loss of generality[3] that every knowledge graph G published via some kind of fragments on the Web is a finite set of blank-node-free RDF triples; i.e., $G \subseteq \mathcal{T}^*$ where $\mathcal{T}^* = \mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$. Each fragment of such a graph contains triples that somehow belong together; they can be obtained from the graph based on some function, which we refer to as a *selector function*.

Definition 1 (selector function). A *selector function* is a function that maps from $2^{\mathcal{T}^*}$ to $2^{\mathcal{T}^*}$.

A selector function can be very precise (e.g., selecting all triples of the knowledge graph whose subject is the URI u) or loosely defined (e.g., selecting triples "related" to the entity identified by u). Moreover, a selector function may also return triples that are not contained in the dataset (as is the case with SPARQL `CONSTRUCT` queries). Usually, any type of fragment comes with a specific type of selector functions. A concrete example are the triple-pattern-based selector functions that we shall introduce to define Triple Pattern Fragments in Section 5.2 (see in particular Definition 6).

As discussed in Section 3.1, Web APIs that follow the REST architectural style equip their representations with hypermedia controls [17, 18]. All controls that do not change resource state (e.g., those used for HTTP `GET`) have in common that, given some (possibly empty) input, the activation of such a control results in a client performing a request of a specific URL. We capture the notion of these controls formally as follows.

Definition 2 (hypermedia control). A (resource-state-invariant) *hypermedia control* is a function that maps from some set to $\mathcal{U}$.

A hyperlink or a URL is a zero-argument control (i.e., a constant function), and a form is a multi-argument control whose domain is the Cartesian product of the form's possible field values. We are specifically interested in

[3] This assumption is possible because skolemization [13] enables the conversion from blank nodes to well-known URIs and back.

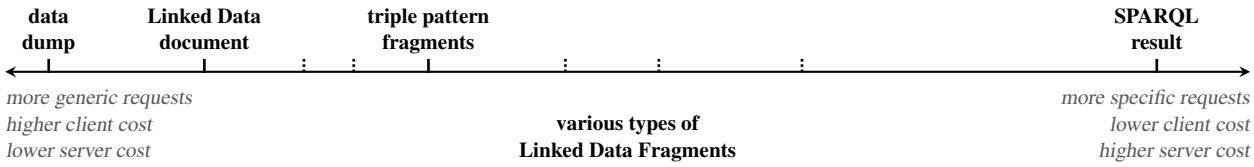

Figure 1: All Web APIs to RDF triples offer Linked Data Fragments of a knowledge graph. These fragments differ in the specificity of the data they contain, and thus the cost to create them.

hypermedia controls whose domain is a set of selector functions, because a URL generated to encode a selector function can be used to denote the data obtained by applying that selector function.

By now, we have introduced all concepts needed to define fragments of an RDF-based knowledge graph.

Definition 3 (Linked Data Fragment). Let $G \subseteq \mathcal{T}^*$ be a finite set of blank-node-free RDF triples. A *Linked Data Fragment (LDF)* of G is a tuple $f = \langle u, s, \Gamma, M, C \rangle$ with the following five elements:

- u is a URI (which is the "authoritative" source from which f can be retrieved);
- s is a selector function;
- Γ is a set of (blank-node-free) RDF triples that is the result of applying the selector function s to G, i.e., $\Gamma = s(G)$;
- M is a finite set of (additional) RDF triples, including triples that represent metadata for f;
- C is a finite set of hypermedia controls.

Any source of RDF-based data on the Web can be described as an LDF by specifying the corresponding values for u, s, Γ, M, and C. The definition of well-known interfaces is discussed in the next section. In particular, we shall distinguish different types of LDFs, each of which represents LDFs that have the same type of selector functions and the same kind of conditions on their metadata M and on their hypermedia controls C.

For some LDFs, the set Γ can be very large. For instance, a data dump may contain millions of triples. Retrieving such a large fragment completely can be undesired in certain situations. For instance, suppose we want to inspect only a part of the data in a fragment, or we are interested only in the metadata of a fragment but not its actual data. To address these use cases and avoid overly large responses, a server that hosts LDFs may partition them into pages. Formally, we capture such a page as follows.

Definition 4 (LDF page).
Let $f = \langle u, s, \Gamma, M, C \rangle$ be an LDF of some finite set of blank-node-free RDF triples $G \subseteq \mathcal{T}^*$. A *page partitioning* of f is a finite, nonempty set Φ whose elements are called *pages* of f and have the following properties:

1. Each page $\phi \in \Phi$ is a tuple $\phi = \langle u', u_f, s_f, \Gamma', M', C' \rangle$ with the following six elements: (i) u' is a URI from which page ϕ can be retrieved, where $u' \neq u$, (ii) $u_f = u$, (iii) $s_f = s$, (iv) $\Gamma' \subseteq \Gamma$, (v) $M' \supseteq M$, and (vi) $C' \supseteq C$.
2. For every pair of two distinct pages $\phi_i = \langle u'_i, u_f, s_f, \Gamma'_i, M'_i, C'_i \rangle \in \Phi$ and $\phi_j = \langle u'_j, u_f, s_f, \Gamma'_j, M'_j, C'_j \rangle \in \Phi$ it holds that $u'_i \neq u'_j$ and $\Gamma'_i \cap \Gamma'_j = \emptyset$.
3. $\Gamma = \bigcup_{\langle u', u_f, s_f, \Gamma', M', C' \rangle \in \Phi} \Gamma'$.
4. There exists a strict total order $\prec$ on Φ such that, for every pair of two pages $\phi_i = \langle u'_i, u_f, s_f, \Gamma'_i, M'_i, C'_i \rangle \in \Phi$ and $\phi_j = \langle u'_j, u_f, s_f, \Gamma'_j, M'_j, C'_j \rangle \in \Phi$ with ϕ_j being the direct successor of ϕ_i (i.e., $\phi_i \prec \phi_j$ and $\neg \exists \phi_k \in \Phi : \phi_i \prec \phi_k \prec \phi_j$), there exists a hypermedia control $c \in C'_i$ with $u'_j \in \mathrm{img}(c)$.

We emphasize that, in addition to the control c for navigating from one page to the next, each page contains *all* metadata and controls of the corresponding fragment. If a server provides paging, it should avoid sending overly large chunks by redirecting clients from a fragment to the first page of the fragment.

Additionally, we should distinguish the formalization of LDFs and LDF pages (Definitions 3 and 4) from their representation in a response to clients (e.g., Section 5.3). For instance, even though a selector function is a component of their definitions, this function is not necessarily represented inside of the response.

In many cases, a Web server that provides access to a knowledge graph exposes an interface to retrieve multiple, different fragments of the graph. As a last general concept, we define such a collection of LDFs.

Definition 5 (LDF collection). Let $G \subseteq \mathcal{T}^*$ be a finite set of blank-node-free RDF triples, and let c be a hypermedia control. A c-specific *LDF collection* over G

is a set F of LDFs such that, for each LDF $f \in F$ with $f = \langle u, s, \Gamma, M, C \rangle$, the following three properties hold: *i)* f is an LDF of G; *ii)* $s \in \mathrm{dom}(c)$; *iii)* $c(s) = u$.

4.3. Application to existing interfaces

To connect the concepts of the previous sections to practice, we now apply them to three existing interfaces discussed in Section 3: SPARQL endpoints, data dumps, and Linked Data documents.

4.3.1. A data dump as an LDF

A data dump of a knowledge graph is a single fragment that consists of the following components.

data: all triples of the graph; hence, in terms of our formalization, the selector function of data dumps is the *identity* s_{id} with $s_{\mathrm{id}}(G) = G$ for all $G \in 2^{\mathcal{T}^*}$;

metadata: data about the knowledge graph or the dump, such as version, author, or licensing details (the metadata set may be empty for some data dumps);

controls: controls to obtain other fragments of the knowledge graph are not necessary because a data dump contains the entire graph.

While some data dumps are divided across multiple archives (e.g, DBpedia 2014, see: `http://wiki.dbpedia.org/Downloads2014`), we can regard such archives as dumps of sub-graphs of a larger knowledge graph. Hence, our conceptual framework still applies.

4.3.2. A SPARQL endpoint as LDFs

Depending on the kind of SPARQL query executed, a SPARQL endpoint returns either a set of RDF triples (for `CONSTRUCT` or `DESCRIBE` clauses), a boolean value (`ASK`), or solution mappings (`SELECT`). Since our conceptual framework is concerned with interfaces that return triples from a knowledge graph, only the `CONSTRUCT` and `DESCRIBE` clauses are under consideration. Since `DESCRIBE` queries, whose behavior is implementation-specific [14], can be written more exactly as `CONSTRUCT` queries, we focus on describing only the latter. However, since it is also possible to specify a triple-based description of any other result form, such as solution mappings from `SELECT`, focusing on `CONSTRUCT` does not present a major conceptual limitation.

The response to a SPARQL `CONSTRUCT` query can be considered an LDF with the following properties.

data: all triples that are contained in the result of the query; thus, the selector function of the LDF is defined based on the query;

metadata: the metadata set is empty;

controls: often, no explicit hypermedia controls are provided in the response because it is assumed that the client using the interface can extract the endpoint URL (e.g., `/sparql`) from the request URL (e.g., `/sparql?query=...`).

The control of the LDF collection exposed by a SPARQL endpoint is the endpoint URL, on which SPARQL queries can be executed.

4.3.3. A Linked Data documents interface as LDFs

Each Linked Data document for a specific entity with URI e in a Linked Data documents interface can be considered an LDF with the following characteristics.

data: RDF triples that are related to the entity and selected with an implementation-specific selector function such as:

$$s_e(G) = \{\langle s, p, o \rangle \in G \mid s = e \text{ or } o = e\};$$

metadata may describe the provenance of the Linked Data document [34] or other types of metadata;

controls HTTP URIs in the data triples act as dereferenceable hyperlinks; in particular, links to entities belonging to the same knowledge graph as e are expected.

Note that different fragments may have different hypermedia controls for this type of interface (which is not the case for data dumps or SPARQL endpoints).

5. Triple Pattern Fragments

5.1. Concept and context

The Linked Data Fragments framework not only enables us to analyze existing Web APIs to RDF in a uniform way, it also allows us to define new APIs with a different combination of characteristics. In particular, we aim to design an interface that enables reliable applications over public RDF-based knowledge graphs. That is, such an interface must be able to counter the availability issues of SPARQL endpoints [5], while still allowing live querying of a knowledge graph. Hence, we require that the interface:

- facilitates query execution;
- limits the cost for the server;
- makes reuse of cached responses likely;
- returns responses that are hypermedia-driven and describe the interface.

The first two properties should facilitate higher availability, provided we incorporate a mechanism to reduce the number of requests needed per client. The notion of "live queryable" means in practice that clients at least must be able to execute SPARQL queries without having to download the entire knowledge graph. Finally, to avoid the inflation of custom APIs for which dedicated clients are needed [8], we aim for an interface with hypermedia controls. Such controls allow clients to identify whether they are talking to a TPF interface or some other interface, and how to access TPFs through it. A hypermedia-aware client should be able to access the API by using hypermedia controls provided in the responses [18].

To facilitate querying on the client side, clients should be able to access fragments that can be used to answer query parts. To maximize availability on the server side, servers should only offer fragments they can generate with minimal CPU and memory cost. In other words, we have to find a compromise along the axis in Figure 1. Offering *triple-pattern*-based access to RDF knowledge graphs seems an interesting compromise because *i)* triple patterns are the most basic building blocks of SPARQL queries [14], and *ii)* servers can select triples that match a given triple pattern at low processing cost [35].

Therefore, we define the Triple Pattern Fragments (TPF) interface, consisting of Linked Data Fragments with the following properties:

data: all triples of a knowledge graph that match a given triple pattern;

metadata: an estimate of the number of triples that match the given triple pattern;

controls: a hypermedia form that allows clients to retrieve any TPF of the same knowledge graph.

A TPF server should divide each fragment into reasonably sized pages (e.g., 100 data triples), such that clients cannot accidentally download very large chunks. For instance, a response for a triple pattern with three unbound variables would contain *all* triples of the knowledge graph if it is not paged. This explains the presence of the metadata: like in any paged interface on the Web, it is beneficial to know how many items there are in total since each page only shows a part of the whole.

In the following, we define TPFs formally. Thereafter, we detail a concrete response format for TPFs in Section 5.3, and explain the relation to existing Web APIs for RDF in Section 5.4.

5.2. Formal definition

We define TPFs based on our general formalization of LDFs as given in Section 4.2. As preliminaries, we first recall some standard definitions of concepts related to SPARQL. We use $\mathcal{V}$ to denote the infinite set of all variables that is disjoint from the sets $\mathcal{U}$ (the set of all URIs), $\mathcal{B}$ (all blank nodes), and $\mathcal{L}$ (all literals). Any tuple $tp \in (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V})$ is called a *triple pattern*[4] and any finite set of such triple patterns is a *basic graph pattern* (BGP), usually denoted by B. Other SPARQL *graph patterns*, usually denoted by P, combine triple patterns (or BGPs) using specific operators [14, 24]. For any such pattern P we write vars(P) to denote the set of all variables that occur in P.

The standard (set-based) query semantics for SPARQL defines the *query result* of a graph pattern P over a set of RDF triples G as a set that we denote by $[\![P]\!]_G$ and that consists of so called *solution mappings*, that is, partial mappings $\mu : \mathcal{V} \to (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$. An RDF triple t is a *matching triple* for a triple pattern tp if there exists a solution mapping μ such that $t = \mu[tp]$, where $\mu[tp]$ denotes the triple (pattern) that we obtain by replacing the variables in tp according to μ. Usage of $\mu[tp]$ can be extended to a BGP B by applying μ to every triple pattern in B, which we denote by $\mu[B]$.

We are now ready to define TPFs. We start by introducing a specific type of selector functions.

Definition 6 (triple-pattern-based selector function). Let tp be a triple pattern. The *triple-pattern-based selector function* for tp, denoted by s_{tp}, is a selector function that, for every set $G \in 2^{\mathcal{T}^*}$, is defined by $s_{tp}(G) = \{t \in G \mid t \text{ is a matching triple for } tp\}$.

We now define a Triple Pattern Fragment and a Triple Pattern Fragment collection, which we introduce together because of their close connection.

Definition 7 (Triple Pattern Fragment). Given a control c, a c-specific LDF collection F is called a *Triple Pattern Fragment collection* if, for every possible triple pattern tp, there exists one LDF $\langle u, s, \Gamma, M, C\rangle \in F$, referred to as a *Triple Pattern Fragment*, that has the following properties:

1. s is the triple-pattern-based selector function for tp.

[4] For the sake of simplicity, in this paper we ignore triple patterns that contain blank nodes, which are possible in the standard SPARQL syntax. However, the query execution algorithm that we shall introduce in Section 6.2 is easily extensible to support queries that contain blank nodes. To this end, every blank node has to be replaced by a unique fresh variable that is not used in the given query; then, for every basic graph pattern of the query (which may now include such new variables), the pattern is evaluated as described in Section 6.2 and, afterwards, these new variables have to be projected out of the query result by an additional iterator that implements a projection operator [36].

2. There exists a triple $\langle u, \texttt{void:triples}, cnt \rangle \in M$ 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 $\text{abs}(|\Gamma| - cnt) \leq \epsilon$ for some F-specific threshold ϵ.

3. $c \in C$.

The threshold ϵ accommodates for implementation differences in calculating the approximate number of triples matching a given triple pattern; interfaces are not required to return the exact number, but they should strive to minimize ϵ. By Definition 7, TPFs have to include the collection-specific hypermedia control, which makes any TPF collection a hypermedia-driven REST API [18]. Hence, by discovering an arbitrary fragment of such a collection, TPF clients can directly reach and retrieve *all* fragments of the collection, which covers the complete set of all possible triple patterns.

Finally, we define a semantics of SPARQL graph patterns when used to query a knowledge graph that is published as a collection of TPFs; that is, the following definition specifies the expected query result of evaluating a SPARQL graph pattern over a TPF collection.

Definition 8 (query semantics). Let $G \subseteq \mathcal{T}^*$ be a finite set of blank-node-free RDF triples, and let F be some TPF collection over G. The *evaluation* of a SPARQL graph pattern P over F, denoted by[5] $[\![P]\!]_F$, is defined by $[\![P]\!]_F = [\![P]\!]_G$.

Observe that, by Definition 8, an approach to execute SPARQL queries over TPF collections is sound and complete if and only if the approach returns query results that are equivalent to the results expected from evaluating the queries directly over the knowledge graphs exposed as TPF collections. While this requirement seems trivial in the context of TPF collections, we emphasize that for other types of LDF collections it is not necessarily possible to ensure such an equivalence. For instance, query evaluation over a Linked Data documents interface (which is an LDF collection) cannot be guaranteed to be complete with respect to all data in the knowledge graph that is exposed by the interface [38, 39] (unless all links between the Linked Data documents are bidirectional).

5.3. Response format and specification

The definition in the previous section brings us to the question of how servers can represent TPF resources. Since metadata and controls should not be considered data from the dataset itself, TPF resources are best represented in quad-based formats such as N-Quads [40], TriG [41], or JSON-LD [42]. Triple-based formats should only be considered if content negotiation reveals the client does not support quads. Such formats are not recommended, since a proper separation of data, metadata, and controls cannot be guaranteed. We will discuss the representation of data, metadata, and controls separately below.

The data component of a TPF (i.e., Γ) is straightforward to represent, as it is a set of triples. These triples should be serialized to the default graph in the cased of quad-based formats. The pagination of data needs to be consistent; the exact order is not determined, as long as it remains the same. Dynamic knowledge graphs can have version-specific URLs in order to achieve result consistency.

Note that we specifically opt to respond with RDF data triples, instead of only solution mappings. For instance, a representation of the TPF for the triple pattern ⟨`<s>`, `<p>`, `?o`⟩ could in theory only list solution mappings with bindings for variable ?*o*, since the predicate and subject are fixed through the request. We did not choose this option for three reasons: *i)* clients might have navigated to the TPF by following a link rather than filling in a form, unaware of the other triple pattern components and thus unable to complete them; *ii)* interpreting such a document would require information outside of the response and thus limit its interpretation to clients that know the TPF specification; *iii)* the possible overhead of component repetition is avoided anyway by common compression techniques for HTTP, such as GZIP.

The metadata, consisting of the estimated total number of data triples in an entire fragment, can also be represented as a triple. Each page ϕ of a given TPF f with URI u contains the triple $\langle u, \texttt{void:triples}, cnt \rangle$ with *cnt* an `xsd:integer` representing the count estimate. In quad-based formats, data and metadata should be separated by placing the metadata in a named graph, say g_M, that is not the default graph. The named graph g_M must explicitly relate itself to the fragment f by including a triple $\langle g_M, \texttt{foaf:primaryTopic}, u \rangle$, so clients can find the metadata of the fragment and interpret it as such.

[5] As usual when introducing multiple evaluation functions to define query semantics (e.g., [37]), we overload notation here. That is, depending on the object represented by the subscript, $[\![P]\!]_{\cdot}$ denotes the evaluation of P over a set of RDF triples or over a TPF collection.

```
<http://fragments.dbpedia.org/2014/en#metadata> {
    <http://fragments.dbpedia.org/2014/en#metadata> foaf:primaryTopic <>.
    <http://fragments.dbpedia.org/2014/en#graph> void:subset <>;
        hydra:filter [
            rdf:type hydra:IriTemplate;
            hydra:template "http://fragments.dbpedia.org/2014/en{?s,p,o}";
            hydra:mapping [ hydra:variable "s"; hydra:property rdf:subject   ],
                          [ hydra:variable "p"; hydra:property rdf:predicate ],
                          [ hydra:variable "o"; hydra:property rdf:object    ]
        ].
}
```

Listing 1: TriG and other RDF representations can use the Hydra Core Vocabulary to represent a three-field form.

Through the Hydra Core Vocabulary [19], we can represent the hypermedia control as a templated link [17] in an RDF-based representation, as illustrated in Listing 1. Such a templated link functions similarly as a form in HTML, which users regularly encounter in a browser. The listing describes a control that lets a client fill in `rdf:subject`, `rdf:predicate`, and `rdf:object` fields, which are mapped to `s`, `p`, and `o`, respectively. Note that the choice of field names depends on the interface and is thus not imposed; on the contrary, the use of an in-band hypermedia control avoids the need to standardize these field names and/or the structure of TPF URLs. Note that the hypermedia control in the description (i.e., the instance of `hydra:IriTemplate`) is attached to the *knowledge graph* rather than to the fragment or page. After all, this control is meant to search the entire knowledge graph, not the fragment. Like metadata, the description of hypermedia controls should reside in a different named graph than the data triples if the RDF serialization supports named graphs. Metadata and controls may be placed together in one named graph without interference.

Note that the number of *different* responses per TPF collection is *finite*, even though the number of possible triple patterns—and thus TPFs—is infinite. Indeed, given two TPFs $f_1, f_2 \in F$ of some TPF collection F, where tp_1 and tp_2 are the triple patterns of the selector functions of f_1 and f_2, respectively, we say that f_1 and f_2 are *equivalent* if $tp_1 = tp_2$ or if there exists a bijective mapping $r : \mathcal{V} \to \mathcal{V}$ such that tp_2 can be obtained from tp_1 by replacing all variables in tp_1 using r. Hence, the number of *mutually non-equivalent TPFs* that have a nonempty set Γ is *finite* for any TPF collection, since only a finite number of non-equivalent triple patterns result in a non-zero number of matching triples. Therefore, for a static knowledge graph, the set of all non-equivalent TPF responses with a nonempty set Γ could be generated beforehand; all empty TPFs could be represented by the same response.

The TPF specification [43] is available as part of the ongoing work in the Hydra W3C Community Group. As expected from a hypermedia-driven REST API, this specification spends "almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state" [18]. Given a built-in understanding of (generic) hypermedia controls as defined in the Hydra Core Vocabulary, this results in an evolvable interface.

5.4. Relation to existing interfaces

We now compare the TPF interface to existing interfaces. In particular, we compare it to Linked Data documents, SPARQL endpoints, and Web APIs in general.

5.4.1. Relation to Linked Data documents

The TPF interface has a strong relation to the Linked Data documents interface. In fact, TPFs can be seen as an extension of the Linked Data principles from hyperlinks to hypermedia forms, which overcomes two drawbacks of dereferencing, *single authority* and *unidirectionality*, as we explain below.

First, while multiple providers may have data about an entity, looking up a URI as per the Linked Data principles leads to data from a single provider, which is considered authoritative with regard to the URI (but not to the entity). For instance, the DBpedia project has minted the URI `http://dbpedia.org/resource/Joan_Miró` for the artist Joan Miró; looking up this URI leads to triples about Miró from the DBpedia knowledge graph only. This does not mean that DBpedia is the only provider—let alone the best or most relevant one—that has triples about Miró. In fact, museums that have Linked Data for Miró might even reuse

the DBpedia URI in their triples, yet this would not make these triples available through URI lookups. In the case of TPFs, users can query arbitrary servers for URIs from any domain in a uniform way. For example, while the official DBpedia TPF interface is hosted at `http://fragments.dbpedia.org/2014/en`, it can still show triples about `http://dbpedia.org/resource/Joan_Miró`, despite different domain names.

Second, Linked Data documents only allow for unidirectional lookups, which severely limits the kind of information clients can derive from a URI. For instance, given the aforementioned Joan Miró URI, it is easy to determine that Miró is a person. On the other hand, given the URI `http://xmlns.com/foaf/0.1/Person`, it is hard to find data about Miró or any other person. This issue stems from the fact that Web linking is unidirectional: for scalability reasons, a hyperlink is maintained only at the outgoing side, not at the incoming side. Such a fundamental limitation puts bounds on which query evaluations can yield results using the interface; typically, one or more triple patterns with non-variable subjects are necessary to obtain results with a purely traversal-based strategy. By using triple-pattern forms, queries with variable subjects become possible at roughly the same per-request cost for the server.

Most importantly, the TPF interface is compatible with the Linked Data principles. For instance, a URI lookup could be redirected using the HTTP status code `303 See Other` to the TPF with the triples that have the requested URI as a subject. As a result, Linked Data clients can look up URIs as usual, without requiring any TPF-specific extension. For example, the server for the domain `http://example.org/` may redirect requests of the URI `http://example.org/people/Joan_Miró` to `http://example.org/knowledge-graph/?subject=http%3A%2F%2Fexample.org%2Fpeople%2FJoan_Miró`, which would be the TPF with triples where Miró occurs in the subject position. This compatibility allows for a seamless integration of TPFs and Linked Data documents, reaffirming the importance of a generic triple-based format that does not require special interpretation, as argued in Section 5.3.

5.4.2. Relation to SPARQL endpoints

Conceptually, accessing a TPF interface is equivalent to accessing a SPARQL endpoint with queries that contain a single triple pattern only (disregarding metadata and controls momentarily). Such a practice would, however, lead to substantially different characteristics than those of the proposed interface.

Cache reuse would decrease, as there are many syntactical variants of such a single triple pattern query, which a simple HTTP proxy server cannot identify as the same query. Crucially, even though some clients might limit themselves to triple pattern queries, this does not stop other clients from requesting more complex queries, thereby still endangering availability. If the triple pattern restriction is not enforced on the server side, there is thus no immediate benefit for clients to use a SPARQL endpoint in that way.

Furthermore, because the TPF interface explicitly states which requests are supported, it can be extended using the same mechanism. That is, publishers are free to provide a TPF-based API with additional functionality by extending responses with additional metadata and controls. This extensibility will be discussed further in Section 8. A query protocol such as SPARQL, that depends on external specification documents instead of in-band metadata and hypermedia controls, necessarily evolves at a slower, centralized pace. While SPARQL endpoints might serve as a back-end for a TPF API, other options exist, some of which provide faster access to triple patterns and approximate counts (such as HDT [35]).

5.4.3. Relation to Web APIs in general

Finally, like most Web APIs, a TPF API is a restricted interface to the underlying database. For instance, Web applications with traditional setups usually consist of a front-end that generates HTML and/or JSON from a relational (or other) database in the back-end. It is very unlikely to find back-end databases directly exposed on the Web in a native query language such as SQL. There are many reasons for this, including security, but the possibly high per-request cost is an important factor.

In contrast to other Web APIs, however, the TPF interface does not expose domain-specific resources. Like SPARQL endpoints, the interface is model-specific, which makes it reusable across many different scenarios. This universality is important to counter the current overgrowth of heterogeneous Web APIs [8].

6. Client-side SPARQL query execution

Given the TPF interface as introduced in the previous section, we now specify how clients can use this interface to execute SPARQL queries over remote knowledge graphs. To this end, we first focus on BGPs and describe the general idea of TPF-based query execution in Section 6.1. Thereafter, in Section 6.2, we introduce a concrete algorithm to evaluate a BGP over a TPF collection. Given such an algorithm, Section 6.3 details how other SPARQL features, such as `OPTIONAL` and `UNION`,

```
SELECT ?person ?city WHERE {
    ?person a dbpedia-owl:Architect.                          # tp1 (± 2,300 matches)
    ?person dbpprop:birthPlace ?city.                         # tp2 (± 572,000 matches)
    ?city dc:subject dbpedia:Category:Capitals_in_Europe.     # tp3 (± 60 matches)
} LIMIT 100
```

Listing 2: This example SPARQL query finds architects born in European capitals.

can be supported. Finally, Section 6.4 elaborates on how TPF query clients can be extended to support queries over a federation of multiple TPF collections.

6.1. General workflow of TPF-based query execution

To start an execution of a SPARQL query over a knowledge graph exposed as a TPF collection on the Web, a TPF query client has to be given the query and the URI of an arbitrary page of some fragment of the TPF collection. As a running example, assume the client is given the query in Listing 2, and the URI `http://fragments.dbpedia.org/2015/en` to retrieves a page from a TPF interface to the DBpedia 2015 knowledge graph. By Definitions 4 and 7, any such TPF page contains a hypermedia control, informing the client inside of the response that the server supports lookups by triple pattern. After obtaining such a hypermedia control, the client can start the query execution process that is based on evaluating the BGPs in the given query.

Listing 3 outlines an approach to perform such an evaluation of BGPs over a TPF collection. We emphasize that Listing 3 is meant to illustrate our general idea of TPF-based query execution rather than being a concrete algorithm to be implemented in a TPF query client. This idea uses a divide-and-conquer strategy that splits each BGP B_i recursively into a triple pattern tp_ϵ and a BGP $B'_i = B_i \setminus \{tp_\epsilon\}$ with one less triple pattern, until a singleton BGP remains. The triple pattern tp_ϵ to split off in each step is determined based on the estimated number of matching triples for tp in the queried knowledge graph, as indicated by the metadata of the first page of the TPF for tp_ϵ. In the following, we discuss Listing 3 in more detail.

First of all, the given BGP B is split into connected sub-BGPs (line 3) since recursive invocations may lead to independent patterns that can be evaluated in parallel. For instance, two sub-BGPs might have been connected by a single variable previously, but due to a candidate mapping involving this variable, the patterns could have become independent of each other. In Listing 2, this would happen if tp_2 is resolved first, which would leave $\{tp_1\}$ and $\{tp_3\}$ as independent sub-BGPs. However, Listing 2 itself contains a single connected sub-BGP, namely the entire BGP.

For each such independent sub-BGP B_i, we aim to find solution mappings. To this end, we first collect the count metadata for each of the triple patterns in B_i by retrieving the first page of the TPFs for these triple patterns (lines 5 to 9). For Listing 2, this results in the following approximate counts: $\{(tp_1, 2\,300), (tp_2, 572\,300), (tp_3, 60)\}$. If one or more of the triple patterns have no matches, then the result of the entire sub-BGP B_i is empty.

Next, we determine the triple pattern $tp_\epsilon \in B_i$ with the smallest estimated number of matches (line 10). By selecting tp_ϵ we minimize the number of immediately following recursive calls—and thus HTTP requests—since each matching triple results in a solution mapping. We retrieve all pages of the TPF for tp_ϵ and use the obtained data triples to compute a set Ω^ϵ of solution mappings for tp_ϵ (lines 11 to 13). For the query in Listing 2 we have $tp_\epsilon = tp_3$, which results in 60 solution mappings, such as $\mu_1 = \{?city \mapsto \texttt{dbpedia:Amsterdam}\}$ and $\mu_2 = \{?city \mapsto \texttt{dbpedia:Athens}\}$.

We apply each of the solution mappings in Ω^ϵ to the BGP that remains after removing tp_ϵ from the given sub-BGP B_i (line 16), which reduces the number of variables in the pattern. Each resulting BGP B'_i is then considered by recursion to obtain a set of solution mappings Ω'_i, respectively (line 17). The mappings in these sets are joined with their respective mapping from Ω^ϵ (line 18). For instance, applying our aforementioned example mapping μ_1 to BGP $\{tp_1, tp_2\}$ results in a BGP $\{tp_1, tp'_2\}$, in which the original triple pattern tp_2 is replaced by tp'_2 that contains `dbpedia:Amsterdam` as a substitute for the `?city` variable in tp_2. By evaluating this BGP $\{tp_1, tp'_2\}$ recursively, we obtain a result that contains solution mappings such as $\mu'_1 = \{?person \mapsto \texttt{dbpedia:Erick\_van\_Egeraat}\}$, which is then joined with μ_1 to yield a complete solution mapping $\{?city \mapsto \texttt{dbpedia:Amsterdam}, ?person \mapsto \texttt{dbpedia:Erick\_van\_Egeraat}\}$.

Function `EvaluateBGP(`B, c`)`
  Input: A basic graph pattern $B = \{tp_1, \ldots, tp_n\}$; a control c of a c-specific TPF collection F
  Output: A set of solution mappings $\Omega = \{\mu_1, \ldots, \mu_m\}$ such that $\Omega = [\![B]\!]_F$
  return $\{\mu_\emptyset\}$ with $\mu_\emptyset$ the empty mapping **if** $B = \emptyset$;
  $S_B \leftarrow \{B_1, \ldots, B_k\}$ such that S_B is the largest possible partition of B for which the following property holds:
    $\forall B_i, B_j \in S_B : B_i \neq B_j \Rightarrow \text{vars}(B_i) \cap \text{vars}(B_j) = \emptyset$;
  foreach sub-BGP $B_i \in S_B$ **do**
    foreach triple pattern $tp_j \in B_i$ **do**
      $\phi_1^j = \langle u_1^j, u_j, s, \Gamma_1^j, M_1^j, C_1^j \rangle \leftarrow$ `GET` $c(\{tp_j\})$, resulting in page 1 of the TPF for tp_j;
      $cnt_j \leftarrow cnt$ where $\langle u_j,$ `void:triples`$, cnt\rangle \in M_1^j$;
      return $\emptyset$ **if** $cnt_j = 0$;
    end
    $tp_\epsilon \leftarrow tp_k$ such that $tp_k \in B_i$ and $cnt_k \leq cnt_j$ for all $tp_j \in B_i$;
    $\Phi^\epsilon \leftarrow \{\phi_1^\epsilon, \ldots, \phi_l^\epsilon\}$ through consecutive `GET` requests for each page ϕ_p^ϵ using the control on ϕ_{p-1}^ϵ;
    $\Gamma^\epsilon \leftarrow \bigcup_{\langle u_l^\epsilon, u_\epsilon, s, \Gamma_l^\epsilon, M_l^\epsilon, C_l^\epsilon\rangle \in \Phi^\epsilon} \Gamma_l^\epsilon$;
    $\Omega^\epsilon \leftarrow \{\mu \mid \text{dom}(\mu) = \text{vars}(tp_\epsilon)$ and $\mu[tp_\epsilon] \in \Gamma^\epsilon\}$;
    $\Omega_i \leftarrow \emptyset$;
    foreach $\mu \in \Omega^\epsilon$ **do**
      $B_i' = \mu[B_i \setminus \{tp_\epsilon\}]$;
      $\Omega_i' \leftarrow$ `EvaluateBGP(`B_i', c`)`;
      $\Omega_i \leftarrow \Omega_i \cup \{\mu \cup \mu' \mid \mu' \in \Omega_i'\}$;
    end
  end
  return $\{\mu_1 \cup \ldots \cup \mu_k \mid (\mu_1, \ldots, \mu_k) \in \Omega_1 \times \ldots \times \Omega_k\}$;
end

Listing 3: A recursive function that illustrates the general process of evaluating a BGP B over a c-specific TPF collection F.

6.2. Incremental algorithm

While Listing 3 is easy to understand because it presents the entire evaluation strategy in a single listing, it is not suited for actual implementation. It can only return all solution mappings at once, and they all have to remain in memory until completion. If the number of mappings is large, this might lead to excessive memory consumption, and if only a subset of the mappings is required (for instance, in combination with `LIMIT`), a significant amount of unnecessary work happens. Furthermore, such a monolithic algorithm would be difficult to extend toward other SPARQL features.

Therefore, we now introduce a concrete algorithm whose execution resembles the execution process outlined in Listing 3, but it does not have the drawbacks as mention before. We achieve this by using the well-known iterator model [44] as a basis of our algorithm.

The iterator model is widely used in query execution systems to enable incremental result computation and to increase the flexibility of implementing query operators [45]. This model introduces the concept of an *iterator* as a particular implementation of an operator that allows a consumer to obtain individual results of each operation separately, one at a time. An iterator provides three functions: `Open`, `GetNext`, and `Close`. `Open` initializes the data structures needed to perform the operation, `GetNext` returns the next result of the operation, and `Close` ends the iteration and releases allocated resources. Query execution plans are implemented as a tree of iterators. Such an iterator tree computes a query result in a *pull*-based fashion; that is, during execution, the `GetNext` function of each iterator calls `GetNext` on the child(ren) and uses the input obtained by these calls to produce the next result(s).

An iterator is said to be *blocking* if it first has to pull all input results before it can return a first output result. Hence, such a blocking iterator cannot be used to produce a query result incrementally [46, 47]. In contrast, a *non-blocking* iterator is able to compute and return output results without having to pull all input results first. Hence, if all iterators in an iterator-based query execution plan are non-blocking, then the plan can produce the query result incrementally. While some query constructs can be implemented by a non-blocking iterator (as well

```
1 Function RootIterator.GetNext()
      Output: μ∅ on first invocation; nil on all subsequent invocations
2     return nil if self.finished has been assigned;
3     self.finished ← true;
4     return μ∅;
5 end
```

Listing 4: A `RootIterator` returns the empty mapping once.

as by a blocking iterator), others can be implemented only by iterators that are blocking.

Our algorithm employs two core iterators, `TriplePatternIterator` and `BasicGraphPatternIterator`, both of which are non-blocking. The former generates solution mappings for a triple pattern using a TPF interface, and the latter generates solution mappings for a basic graph pattern using a TPF interface and a combination of multiple `TriplePatternIterators`. Additionally, a helper `RootIterator` does not depend on a source iterator and, thus, can act as the source of any other iterator without initialization.

Listing 4 describes the `GetNext` method of the `RootIterator`: it returns the empty mapping $\mu_\emptyset$ (with $\text{dom}(\mu_\emptyset) = \emptyset$) exactly once. This empty mapping can be used as a starting point for a `TriplePatternIterator` or a `BasicGraphPatternIterator`. For instance, if a `TriplePatternIterator` reads $\mu_\emptyset$ from an input `RootIterator`, it simply returns solution mappings for its triple pattern, without extending them.

A `TriplePatternIterator` (Listing 5) reads solution mappings from a source iterator I_s and combines them with possible mappings for a given triple pattern *tp*. For example, if its triple pattern is ⟨`?person`, `dbpprop:birthPlace`, `?city`⟩ and the source iterator has returned $\mu_s = \{?city \mapsto$ `dbpedia:Amsterdam`$\}$, the `TriplePatternIterator` will subsequently request ⟨`?person`, `dbpprop:birthPlace`, `dbpedia:Amsterdam`⟩ triples through the TPF interface. If the TPF response contains a triple ⟨`dbpedia:Erick_van_Egeraat`, `dbpprop:birthPlace`, `dbpedia:Amsterdam`⟩, then the iterator will return the combined solution mapping $\mu \cup \mu_s = \{?city \mapsto$ `dbpedia:Amsterdam`, $?person \mapsto$ `dbpedia:Erick_van_Egeraat`$\}$. A `TriplePatternIterator` has two member variables: `self.`ϕ to hold the current TPF page from which it is reading, and `self.`μ_s to hold the most recently read mapping from `self.`ϕ. If this page is read, the next page of the same TPF is requested if it exists (line 5). If the fragment has no more pages, the next mapping `self.`μ_s is read from the source iterator I_s, and the first page of the TPF for the mapped triple pattern `self.`$\mu_s[tp]$ is fetched (lines 7 to 9). That way, the solution mappings resulting from matching triples in the TPF's pages are compatible with the corresponding input mappings `self.`μ_s.

Finally, a `BasicGraphPatternIterator` combines the above two iterators to evaluate BGPs. For an empty BGP, the `BasicGraphPatternIterator` constructor creates a `RootIterator` instead (since the corresponding query result contains only the empty mapping $\mu_\emptyset$); for a singleton BGP, a `TriplePatternIterator` is constructed (since no further decomposition is needed). In all other cases, Listing 6 is executed.

The main principle of a `BasicGraphPatternIterator` is that it creates a *separate* iterator pipeline for each incoming solution mapping. The iterator has two member variables: `self.`I_p to hold the current iterator pipeline, and `self.`μ_s to hold the most recently read mapping from its source iterator I_s. As in Listing 3[6], upon reading a mapping `self.`μ_s, a `BasicGraphPatternIterator` identifies which of the triple patterns in $\{$`self.`$\mu_s[tp_j] \mid tp_j \in B\}$ has the lowest estimated total matches by fetching the first pages of the corresponding TPFs (lines 8 to 13). Then, a new iterator pipeline `self.`I_p is created, consisting of a `TriplePatternIterator` for the identified triple pattern and a `BasicGraphPatternIterator` for the remainder of B (lines 14 to 15). The mappings μ returned by this pipeline are then combined with the input mapping `self.`μ_s and returned (lines 18 to 21). In other words, the BGP is evaluated by splitting off the "simplest" triple pattern at each stage. For Listing 2, the `BasicGraphPatternIterator` would thus split off tp_3, and create a pipeline for $\{tp_1, tp_2'\}$. This process is *dynamic*: instead of constructing a static pipeline for a BGP upfront, a local pipeline is decided at each step.

Note in particular that only `TriplePatternIterators` read more than one page of a TPF. `BasicGraphPatternIterators` only fetch first pages, and never read actual data, only metadata. In an efficient implementation, pages are cached locally, so that the `TriplePattern-`

[6]For simplicity, Listing 6 does not display the process of splitting the BGP into connected subpatterns as shown in Listing 3 (line 3).

```
1  Function TriplePatternIterator.GetNext()
       Data: A source iterator I_s; a triple pattern tp; a control c of a c-specific TPF collection F
       Output: The next mapping μ' such that μ' ∈ [[{tp}]]_F, or nil when no such mappings are left
2      self.φ ← an empty page without triples or controls if self.φ had not been assigned to previously;
3      while self.φ does not contain unread triples do
4          if self.φ has a control to a next page with URL u_φ' then
5              self.φ ← GET u_φ';
6          else
7              self.μ_s ← I_s.GetNext();
8              return nil if self.μ_s = nil;
9              self.φ ← GET c({self.μ_s[tp]}), resulting in page 1 of the TPF for self.μ_s[tp];
10         end
11     end
12     t ← an unread triple from self.φ;
13     μ ← a solution mapping μ' such that dom(μ') = vars(tp) and μ'[tp] = t;
14     return μ ∪ self.μ_s;
15 end
```

Listing 5: A `TriplePatternIterator` incrementally evaluates a triple pattern tp over a c-specific TPF collection F.

```
1  Function BasicGraphPatternIterator.GetNext()
       Data: A source iterator I_s; a BGP B with |B| ≥ 2; a control c of a c-specific TPF collection F
       Output: The next mapping μ' such that μ' ∈ [[B]]_F, or nil when no such mappings are left
2      μ ← nil;
3      self.I_p ← nil if self.I_p has not been assigned previously;
4      while μ = nil do
5          while self.I_p = nil do
6              self.μ_s ← I_s.GetNext();
7              return nil if self.μ_s = nil;
8              foreach triple pattern tp_j ∈ B do
9                  φ_1^j = ⟨u_1^j, u_j, s, Γ_1^j, M_1^j, C_1^j⟩ ← GET c({self.μ_s[tp_j]}), resulting in page 1 of that TPF;
10                 cnt_j ← cnt where ⟨u_j, void:triples, cnt⟩ ∈ M_1^j;
11             end
12             if ∀j : cnt_j > 0 then
13                 ϵ ← j such that cnt_j ≤ cnt_k ∀tp_k ∈ B;
14                 I_ϵ ← TriplePatternIterator(RootIterator(), self.μ_s[tp_ϵ], c);
15                 self.I_p ← BasicGraphPatternIterator(I_ϵ, self.μ_s[B \ {tp_ϵ}], c);
16             end
17         end
18         μ ← self.I_p.GetNext();
19         self.I_p ← nil if μ = nil;
20     end
21     return μ ∪ self.μ_s;
22 end
```

Listing 6: A `BasicGraphPatternIterator` incrementally evaluates a BGP B over a c-specific TPF collection F.

`Iterator` need not fetch the first page again—and in general, no page should be fetched more than once.

Several improvements to this algorithm are possible. For instance, consider the following query:

```
SELECT * WHERE {
  ?s1 <p1> ?o1. # tp1 (± 10 matches)
  ?s1 <p2> ?o2. # tp2 (± 1,000,000 matches)
  ?s2 <p3> ?o2. # tp3 (± 100 matches)
}
```

The algorithm of Listing 6 will first consider tp_1, which will supply mappings for *?s1* to the remaining subpattern $B' = \{tp_2, tp_3\}$. The `BasicGraphPatternIterator` will then instantiate *?s1* with each of the supplied values, leading to instantiated subpatterns such as $B'' = \{tp'_2, tp_3\}$ with $tp'_2 = \langle s1, p2, ?o2\rangle$ for specific *s1*. If the estimated match count of tp'_2 is higher than that of tp_3 (100), then tp_3 will be selected first. This leads to a Cartesian product between tp_1 and tp_3, since they share no variables. To mitigate this problem, Acosta and Vidal [48] proposed to prefer triple patterns that share variables with the first triple pattern during initial query plan generation. In this example, this would change the execution order to (tp_1, tp_2, tp_3) rather than (tp_1, tp_3, tp_2). Combined with adaptive query execution, this typically leads to faster evaluation [48]. Other improvements include the usage of a symmetric hash join instead of a nested loop join whenever this results in fewer HTTP requests [48, 49].

6.3. Evaluating other SPARQL query constructs

The discussed query algorithm focuses on BGPs, as they are the main parts of SPARQL queries. Even though subtleties of other query constructs make it hard to cover their full usage range on a high level, it is interesting to look at common occurrences. An important consideration is whether or not the query result can be produced incrementally, in which case first solutions of the result can be presented to the user while the query execution is still running. Due to the non-blocking iterators in Section 6.2, our iterator-based implementation is able to produce results for BGPs incrementally. However, for SPARQL queries with query constructs other than BGPs, such a behavior may not be guaranteed anymore.

First, let us consider SPARQL query modifiers [14]. For `LIMIT` we use a non-blocking iterator that simply pulls the iterator tree the required number of times. `OFFSET` is equally simple, though less efficient: drop the required number of elements and continue from there, which we implemented with a non-blocking iterator. If `LIMIT` and `OFFSET` are used for paging, which is often the case, it is possible to keep the iterator open between subsequent queries. This allows paged results without extra computational cost. `DISTINCT` requires keeping the results in memory to validate whether the same mappings have occurred before; we implemented this as a non-blocking iterator. In contrast, given that a regular TPF interface does not impose an ordering, we had to implement `ORDER BY` with a blocking iterator that retrieves all input solution mappings and sorts them by using a variant of the well-known quicksort algorithm. Similarly, aggregates such as `COUNT` and `SUM` use blocking iterators.

Next, let us consider graph patterns other than BGPs. With TPFs, `FILTER` constructs can generally only be evaluated client-side, for which we use a non-blocking iterator that has to be applied after bindings for the variables in the filter have been computed. For example, a query on DBpedia looking for English labels of movies would have to include a filter such as `FILTER(LANGMATCHES(LANG(?label),"EN"))`. The client would necessarily first find bindings for `?label` and then execute the filter over them. This does not pose performance problems for filters with relatively low selectivity compared to other parts of the query. However, filters with high selectivity benefit from early scheduling in the query plan, possibly before bindings for all of its variables have been computed. For instance, a query for "movies whose title contains 'silence'" would include a filter such as `FILTER(REGEX(?label, "silence", "i"))`. Only few labels satisfy such a condition in practice, so it would be beneficial to execute the filter early in the plan. Unfortunately, with a TPF interface, we have no option but to retrieve all movies and their titles, since we need a binding for `?label`. This makes the execution of such queries significantly slower than similar queries without filters. Finally, `UNION` is also implemented as a non-blocking iterator (that, similar to `DISTINCT`, requires keeping a list of results under set semantics), and `OPTIONAL` is implemented with a blocking iterator.

In general, *all* SPARQL queries can be executed with full completeness using TPFs; some of them efficiently, some only very slowly. This is a consequence of the fact that a TPF interface allows clients to obtain all data in the knowledge graph by retrieving the `?s ?p ?o` fragment. Even though this observation might appear trivial, note that is not necessarily possible to execute all SPARQL queries over any public SPARQL endpoint: some endpoints limit the kind of queries client can execute, and/or the number of triples that can be returned and/or disallow high `OFFSET`s. Therefore, some queries can neither be executed remotely (because the server disallows them) nor locally (because not all data can be downloaded) using a SPARQL endpoint. With a TPF interface, it is by definition possible to download the entire dataset, and

hence, to execute any SPARQL query.

6.4. Mediator to query a federation of TPF interfaces

To conclude this section, we propose a *mediator* layer to query multiple TPF interface instances with a single query. A mediator is a software module that abstracts a collection of data resources for a higher software layer [50], making them independent from each other. In this case, each TPF interface acts as a data-source *wrapper*, with the mediator creating a unified TPF view over the interfaces [51]. As a result, the task of a TPF client when executing a (regular) SPARQL query over a federation of TPF interfaces is to compute results identical to those it would obtain when the query would be evaluated over a single TPF interface that combines the data of all considered interfaces. That is, a user can "query using a classical query language against the federated schema with an illusion that he or she is accessing a single system" [12].

6.4.1. Comparison with SPARQL federation frameworks

Federated query processing has been studied in the context of SPARQL endpoints, where most SPARQL endpoint federation frameworks apply a client-server architecture with multiple SPARQL endpoints and one or more clients [52]. An important step in federated query execution frameworks is *source selection* [53–55], determining which endpoints are relevant to evaluate a given query. Based on this information, a SPARQL query is divided in subqueries that are sent to the relevant endpoints. These partial results are then combined into results for the entire query.

Source selection usually involves pre-computed summaries and/or the retrieval of extra (meta-)data from the endpoints, and happens as a *separate* step before the actual execution. While this step is intended to reduce the number of requests to servers, and hence the overall query evaluation time, source selection itself also takes time. Since source selection can be performed *independently* of the actual query execution, existing source selection algorithms can be used in conjunction with different execution strategies. Therefore, in the following, we will *not* focus on source selection. Instead, we propose only a TPF-specific federation mediator, which can optionally be preceded by an existing source selection step.

6.4.2. Mediator layer

The mediator federates TPF requests to candidate sources and uses runtime *source elimination* during query execution to prevent HTTP requests we know would not result in a match. Source elimination either refines the optional source selection step at every iteration by excluding more specific patterns, or, if no prior source selection was performed, acts as a runtime optimization.

The client is given a query, and a pre-defined list of TPF interfaces (instead of a single TPF interface). This list can optionally be pre-filtered by a source selection algorithm. The mediator creates an abstraction layer to the fragment request operation: all interfaces are exposed to the query algorithm as a single interface. Initially, all interfaces are marked as possible candidates for each triple pattern (unless explicitly ruled out by the optional source selection step). Each time the client requests a TPF, the mediator consults all candidate interfaces for the same triple pattern.

When a TPF interface returns an empty fragment for a certain triple pattern, it becomes an eliminated source for that pattern, which is stored in an interface-specific elimination list. For each triple pattern requested from this interface, we check the elimination list for a possible *ancestor* pattern. A tp_a is an ancestor of tp_b if each term of tp_a is either a variable or equal to the corresponding term of tp_b. If an ancestor pattern is found, the interface is no longer marked as a candidate, since it has no matches for the ancestor pattern, nor for more specific patterns.

For all interfaces, the fragments' data and metadata are merged in a streaming way. The count metadata are combined into a single value using an aggregation function $\vartheta(tp_j)$, which is a cost function that can be optimized to the type of interface. For example, $\vartheta(tp_j)$ can be a *weighted* sum, taking into account practical differences between N servers, such as response time, page size, etc. In our implementation, we chose an ordinary sum for simplicity; i.e., $\vartheta(tp_j) = \sum_{i=1}^{N} \mathrm{cnt}_{ij}$.

The application of this algorithm is evaluated in Section 7.4, where the query evaluation over a federation of TPF interfaces is considered without prior source selection.

7. Evaluation

In this section, we experimentally test the hypotheses H1–H4 as introduced in Section 2.2. The results of all experiments are published at `https://github.com/LinkedDataFragments/TPF-Experiment-Results`.

As a basis for the experiments, we implemented the query execution approach of Section 6.2 as an open-source LDF client for SPARQL queries. This client is written in JavaScript, so it can be used either as a standalone application, or as a library for browser and server

applications. We provide all source code of the implementations, as well as the full benchmark configuration, at `https://github.com/LinkedDataFragments/`.

7.1. Experiment 1: Influence of client numbers

With our first experiment, we aim to validate hypothesis H1: "In comparison to the state-of-the-art in single-machine SPARQL endpoints, *I* reduces the server-side costs to publish knowledge graphs in a queryable form."

7.1.1. Experimental setup

In this experiment, we compare a TPF client/server setup to four SPARQL endpoint based setups. For the latter we use Virtuoso (6.1.8 and 7.1.1) [22] and Jena Fuseki [23] (TDB 1.0.1 and HDT 1.1.1), respectively; and for the TPF server we use an HDT [35] backend. Virtuoso was configured with `NumberOfBuffers` = 595000, `MaxDirtyBuffers` = 455000, and `MaxCheckintRemap` as 1/4 of `NumberOfBuffers`.

To measure the cost and performance of the TPF server and the SPARQL endpoints under varying loads, we set up an environment with one server and a variable number of clients on the Amazon AWS platform. The complete setup consists of 1 server (4 virtual CPUs, 7.5 GB RAM), 1 HTTP cache (8 virtual CPUs, 15 GB RAM) and 60 client machines (4 virtual CPUs, 7.5 GB RAM), capable of running 4 single-threaded clients each. All machines have Intel Xeon E5-2666 processors running at 2.60 GHz. The HTTP cache acts as a proxy server between servers and clients and was chosen for its bandwidth capabilities (which Amazon associates with specific CPU/RAM combinations). The maximum lifetime of cached resources is set to 5 mins.

As a benchmark for the experiment, we chose the Berlin SPARQL Benchmark (BSBM) [56], for the following reasons:

- the BSBM was designed to compare SPARQL endpoints across architectures [56], and we aim to compare our client–server architecture to such traditional server architectures;
- the BSBM aims to simulate realistic workloads with large amounts of RDF data [56];
- the BSBM contains parametrized queries to create different workloads for large numbers of clients.

In particular, we used a BSBM instance with a knowledge graph size of 100M triples. To mimic the variability of real-world scenarios, each client executes different BSBM query workloads based on its own random seed. Some of the BSBM queries use the `ORDER BY` operator, which our TPF client implements with a blocking iterator, so the first solution can only be output after all solutions have been computed. Therefore, (only) for measurements of the first solution time we use variants of these queries without `ORDER BY`, assuming the user application prefers streaming results and sorts locally. After every 1-second interval during the evaluation, we measure on the server, cache, and clients the current value of several properties, including CPU usage of each core and network IO.

7.1.2. Results

Figures 2.1 to 2.8 summarize the main measurements of the evaluation. All x-axes use a logarithmic scale, because we varied the number of clients exponentially. We measured all data points for Virtuoso 7 (as latest and best performing version) and our proposed solution. For the other alternatives, we measured the points most relevant for the analyses. Figure 2.1 shows that the performance of SPARQL endpoints decreases significantly with the number of clients. Even though a TPF setup executes SPARQL queries with lower performance, the performance decrease with a higher number of clients is significantly lower. Because of caching effects, TPF querying starts performing slightly better with a high number of clients ($n > 100$). The per-core processor usage of the SPARQL endpoints grows rapidly (Figure 2.5) and quickly reaches the maximum; in practice, this means the endpoint spends all CPU time processing queries while newly incoming requests are queued. The TPF server consumes only limited CPU, because each individual request is simple to answer, and due to the coarser granularity of the selector function, the cache can answer several requests (Figure 2.4).

At the client side, we observe the opposite (Figure 2.6): clients of SPARQL endpoints hardly use CPU, whereas TPF clients use between 20% to 100% CPU. With an increasing number of concurrent TPF clients, the networking time dominates and, thus, the per-client CPU usage decreases, whereas the memory consumption does not vary significantly (ca. 0.5 GB per client, and 8 GB on the server).

Figure 2.2 shows the outbound network traffic on the server with an increasing number of clients. This traffic is substantially higher for the TPF server, because TPF clients need to perform several requests to evaluate a single query. The cache ensures that responses to identical requests are reused; Figure 2.4 shows that caching is far more effective with TPFs.

Some of the BSBM queries perform comparably slow on TPF clients, especially those queries that depend on operators such as `FILTER`, which in a triple-pattern-based

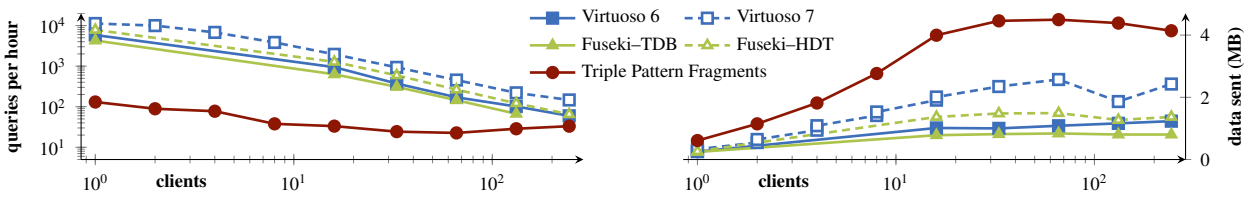

Figure 2.1: Server performance *(log-log plot)*

Figure 2.2: Server network traffic

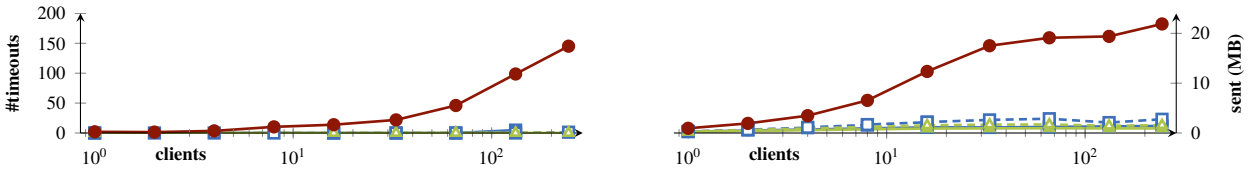

Figure 2.3: Query timeouts

Figure 2.4: Cache network traffic

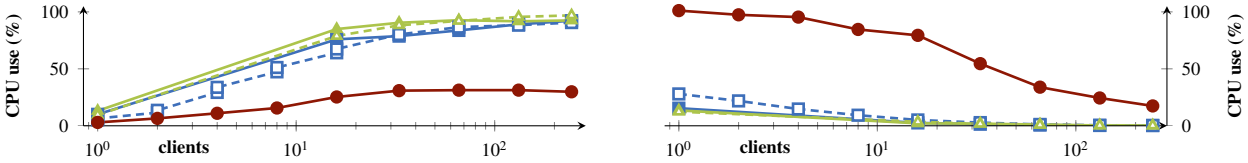

Figure 2.5: Server processor usage per core

Figure 2.6: Client processor usage per core

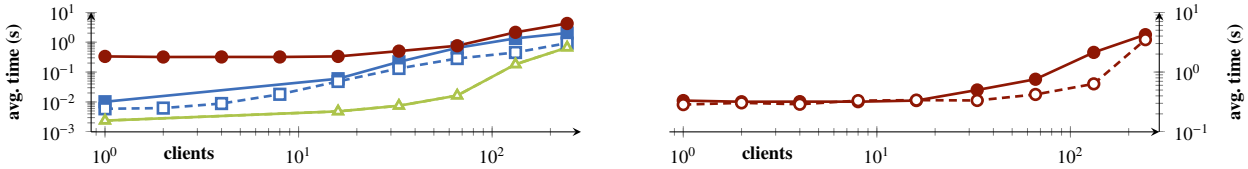

Figure 2.7: Query 3 execution time *(log-log plot)*

Figure 2.8: Query 3 execution and 1st solution time (dotted) *(log-log plot)*

interface can only be evaluated on the client. The execution times of some of these queries exceed the timeout limit of 60s (Figure 2.3). A "timeout" means that query execution was stopped before *all* results arrived; at least a partial result set was already computed. An example of such queries are instances of BSBM query template 3 (finding products that satisfy 2 numerical inequalities and an `OPTIONAL` clause). Focusing only on the queries of this template, we observe that the average execution time of these queries is comparably higher for TPFs (Figure 2.7). However, with an increasing number of clients, these times increase only very gradually in the TPF setup, whereas they rise very rapidly for the SPARQL endpoints (which have to compute the queries of all clients concurrently). Furthermore, the time to the first solution increases more slowly with increased load (Figure 2.8). Only on the TPF server, CPU usage remains low for this query at all times.

7.2. Experiment 2: Performance of queries on a real-world knowledge graph

Our second experiment aims to extend the results of Experiment 1 toward real-world knowledge graphs and different dataset sizes, as per H2: "The majority of typical real-world queries execute over *I* in less then 1 second."

7.2.1. Experimental setup

The experimental setup is the same as that of Section 7.1. However, this time, we want to validate the behavior of the TPF client for increasing real-world knowledge graph sizes. To this end, we execute the DBpedia SPARQL benchmark [57], which uses a real-world dataset. The benchmark incorporates queries from the public DBpedia SPARQL endpoint log, filtering non-relevant variations and queries with a low frequency [57]. We use three knowledge graph sizes as made available online: 14,274,065 triples and 52,323,498 triples from the DBpedia benchmark website[7], and 377,367,913 triples

[7] `http://benchmark.dbpedia.org/`

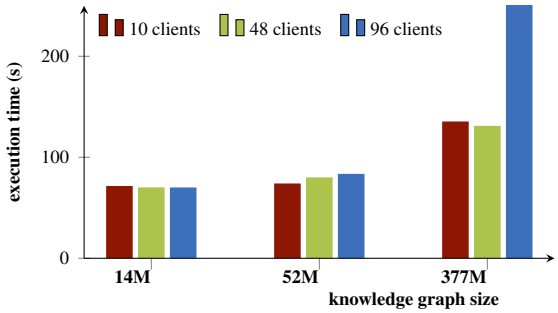

Figure 3: Average execution times per client of a DBpedia query mix with variable knowledge graph sizes

	14M	52M	377M
Q1	0.44	0.42	0.37
Q2	0.50	0.50	0.46
Q3	0.50	0.51	0.91
Q4	0.53	0.60	1.49
Q5	0.47	0.48	0.86
Q6	1.80	10.06	35.16
Q7	0.47	0.46	0.40
Q8	0.50	0.49	0.46
Q9	0.47	0.45	0.39
Q10	0.47	0.45	0.46
Q11	0.52	0.92	0.43
Q12	60.03	60.03	59.95
Q13	2.29	3.16	21.80
Q14	0.48	0.47	0.43
Q15	0.73	1.08	7.69
Total	70.19	80.09	131.27

Table 1: Average individual query execution time (in seconds) for the three DBpedia knowledge graph sizes with 48 concurrent TPF clients

from the 2014 version of DBpedia (without long abstracts). In contrast to the BSBM we used in the last experiment, the DBpedia query logs do not contain parameters. Therefore, clients execute a mix of 15 queries with different template values for each client. The generated queries are available along with the benchmark data. Query timeout was set to 60s.

7.2.2. Results

Figure 3 shows the results for 10, 48, and 96 TPF clients. We observe an increase in query execution time, which is minor when going from the 14M graph to the 52M graph, but more visible for the entire version of DBpedia 2014. The main cause for this increase is the increased number of elements in the result set, as the total number of triples influences the number of matching triples per query. To a lesser degree, the execution times are also influenced by an increased number of triples that match one pattern but cannot be used in joins.

The most important observation in this experiment is, however, the high variance in execution time across queries. Table 1 shows the average query execution times for an individual client per query, for the case of 48 concurrent clients. Note how Q12 reaches the 60s timeout even for the small knowledge graph. This is due to the presence of various `UNION`, `OPTIONAL`, and `FILTER` constructs, for which our client does not generate efficient query plans. Most of the query execution times remain at the same magnitude for the different knowledge graph sizes, with small differences accounting for factors such as caching and increasing bandwidth consumption. Queries Q6, Q13 and Q15 are affected more clearly by increasing knowledge graph sizes; at the same time, however, they also yield more results. This indicates that, more than knowledge graph size, the type of query seems to be a crucial factor for queries against real-world knowledge graphs such as DBpedia.

7.3. Experiment 3: Influence of serialization formats

With our third experiment, we aim to validate H3: "Serialization formats that result in a lower response size of I, compared to N-Triples, decrease query execution times."

The delay to transfer triples from the server into a memory-based representation of the client is much more crucial for TPF clients than for SPARQL endpoints, because the number of required HTTP requests per query can be high. Therefore, we study the impact of the RDF serialization format of TPF responses on SPARQL query execution performance. The serialization format contributes to the delay in two ways: *i) response processing delay*, i.e coding and decoding triples, and *ii) response download delay*, i.e. transporting the response over network. A serialization format offers a specific mix of both, which can be more or less suited for a certain Web application. We evaluated different existing formats to discover a balanced mix between processing and download cost in the context of our client-side querying algorithm. Subsequently, we can obtain recommendations on *i)* characteristics for new serialization formats, and *ii)* server page size configurations.

7.3.1. Serialization formats

To test with a diverse mix of formats we identified three categories—text-based, processing-oriented binary, and download-oriented binary—from which we selected an overall number of twelve different formats, and used each of them with and without additional GZIP compression, which is common with HTTP.

Text-based formats are string-encoded notations aimed at both human and machine consumption. We selected

the W3C standards N-Triples [58] and its superset Turtle [59]. The Apache Jena RIOT library[8] was used for both formats, using stream processing when possible. For Turtle, we tested the three configurations *pretty* (pre-sorted by subject with maximal grouping; non streaming), *flat* (not pre-sorted and not grouped; triple-based streaming), and *block* (pre-sorted by subject and grouped in fixed windows; window-based streaming). In addition, we tested with the Sesame RIO[9] library to include a possible implementation difference.

Processing-oriented binary formats are binary notations optimized to reduce processing delay, which often comes with a response size penalty. Fortunately, such formats tend to work well with common compression, e.g., GZIP. This compensates response size increase at the cost of processing time. In this category, we selected RDF Thrift [60], implemented by Jena RIOT, and Sesame Binary RDF [61], implemented by Sesame RIO. For RDF Thrift we tested both the default configuration and the *Values* configuration, where literals are encoded more efficiently. For Sesame Binary we used both a buffer size of 800 and of 1,600 triples.

Download-oriented binary formats are optimized to reduce download delay. They offer compression techniques that greatly reduce response size, often sacrificing processing time. We added the recent ERI format [62] using a custom implementation by the authors. We tested with the recommended block sizes of 1,024, 2,048, and 4,096 triples.

7.3.2. Experimental setup

To obtain collections of TPF responses for which we could measure the impact of each of the selected serialization formats, we instructed our TPF client implementation to store each TPF page retrieved during a query execution as a local RDF file. Then, to take the measurements for this experiment we implemented a single-threaded Java application that loads such a local RDF file into main memory and serializes and deserializes the loaded data using the different serialization formats mentioned before. This application was deployed on an Intel Xeon CPU (E5-2640 2.50 GHz) with 1 TB HDD RAID storage and 64 GB DDR3 1333 MHz RAM.

To ensure diversity of the collections of TPF responses as considered for this experiment, we used queries and a knowledge graph of the Waterloo SPARQL Diversity Test Suite (WatDiv) [63]. In contrast to the previous two experiments, which focus on realistic loads, we here specifically want to assess the impact of different query patterns, which WatDiv is designed for. We set up our TPF server with the WatDiv 10 million triple knowledge graph[10] and queried it using 20 BGP queries that we generated from the 20 query templates in the basic testing use case[11] of WatDiv. These query templates span a wide spectrum of various graph pattern structures (cf. Table 2). For each of the 20 queries, Table 2 shows the number of TPF pages requested by our client-side query execution algorithm.

template name	triple patterns	variables	join vertices	*TPF pages*	template name	triple patterns	variables	join vertices	*TPF pages*
L1	3	3	2	*9,159*	**S1**	9	9	1	*251*
L2	3	2	2	*14,815*	**S2**	4	3	1	*1,415*
L3	2	2	1	*59,062*	**S3**	4	4	1	*455*
L4	2	2	1	*357*	**S4**	4	3	1	*67*
L5	3	3	2	*315*	**S5**	4	3	1	*304*
F1	6	5	2	*900*	**S6**	3	3	1	*33*
F2	8	8	2	*184*	**S7**	3	3	1	*594*
F3	6	6	2	*1,793*	**C1**	8	9	4	*869*
F4	9	8	2	*869*	**C2**	10	11	6	*431*
F5	6	6	2	*216*	**C3**	6	7	1	*4*

Table 2: Properties of our WatDiv queries (template from which the query is generated, number of triple patterns, number of variables, join vertex count [63]), and the number of TPF pages requested during their execution. WatDiv distinguishes linear queries (**L**), snowflake-shaped queries (**F**), star queries (**S**), and complex queries (**C**).

7.3.3. Results

First, we analyze how much data the collected TPF pages contain (which is independent of serialization formats). Figure 4 shows a distribution of the number of triples per page. The histogram starts with 25 triples, because a response always includes at least 26 triples for metadata and controls, and ends with 130 triples because a page size of 100 triples was used. All queries show a similar bimodal distribution with a peak at a triple count of 25 to 35 triples and a small peak at 125 to 130 triples. The former indicates the presence of many requests that are used to verify the presence of a single triple, which results in 0 or 1 data triples. The latter denotes the initial phase of query execution where many non-selective triple patterns are requested. Most frequent, these are the triple patterns as they occur in the query's BGP and some of their early bound derivatives. These patterns contain the highest number of variables, more likely to

[8] https://jena.apache.org/documentation/io/

[9] http://rdf4j.org/sesame/2.8/docs/programming.docbook?view#Parsing_and_Writing_RDF_with_Rio

[10] http://dsg.uwaterloo.ca/watdiv/watdiv.10M.tar.bz2

[11] http://dsg.uwaterloo.ca/watdiv/basic-testing.shtml

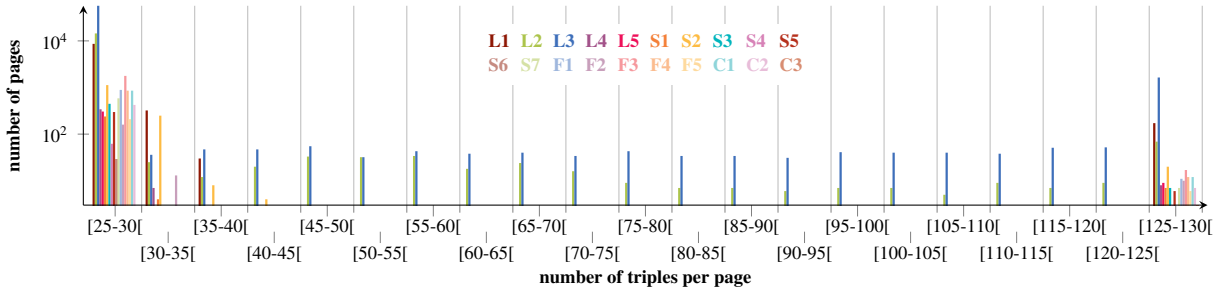

Figure 4: The distribution of average number of triples per used TPF page for different queries shows that most fragments either contain either few data triples (20–30 triples are metadata), or the maximum number (page size).

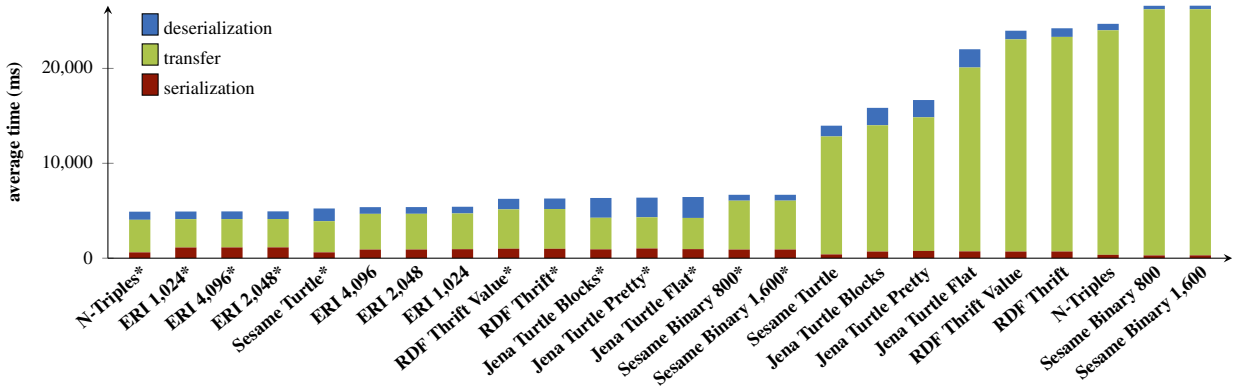

Figure 5: The chosen RDF format impacts the time to serialize, transfer, and deserialize TPF pages. Additionally applying GZIP compression and decompression (indicated by *) has an important effect, except on ERI, which is already small.

exceed the given page size. Examples are the triple patterns tp_1 and tp_2 from Listing 2, that have a cardinality of 2 300 and 572 300, respectively. On average, only 2% of all responses contain between 35 and 125 triples. This means HTTP responses are in most cases very small, in a few cases very big (depending on the page size), and almost never in-between. Two queries (L2 and L3) have more fragment pages in the in-between groups, which is directly related to the fact they have more pages overall.

Next, we executed the response serializer for each combination of query and format, with and without applied GZIP compression. For each response, we measured its serialization time, deserialization time and transfer time. We estimate the transfer time by dividing the response size when serialized with an average network speed of 1MB/s (as in other evaluations [62]). The sum of all three measurements over all HTTP responses give the total overhead per query.

Figure 5 shows the average overhead over all queries. A first observation is that transfer time clearly dominates over serialization and deserialization time. Therefore, the small-size ERI format results in roughly 3 times less overhead compared to Turtle and even 5 times less compared to N-Triples, RDF Thrift, and Sesame Binary. However, since text-based formats and processing-oriented formats can be compressed effectively, the response size difference can be reduced significantly by applying GZIP compression. On average, text-based formats can be reduced to 110% of the ERI response size. Nonetheless, the processing penalty by applying compression does not compensate the dominance of transfer delay for most formats, as ERI keeps an average gain of 1s per query. For N-Triples though, the combination of its high compression rate and lower processing cost match the low overhead of ERI.

In general, the impact of binary formats is rather limited. This is likely because TPF pages are small, while such formats are designed to be effective on large amounts of triples. All responses are serialized as a single block or buffer, which explains the insignificant difference between block sizes for ERI and Sesame Binary. A larger page size could result in a potential overhead decrease. However, for all tested queries, only 0.24% of all fragments requested more than one page and only 0.001% more than two. As a result, the benefit of a larger page size would be negligible in the current algorithm.

Still, future algorithms could optimize more in this direction. For instance, Van Herwegen et al. [49] reduce the number of requests by downloading all pages first to perform a local join, which could justify a larger page size.

7.4. Experiment 4: Querying federated knowledge graphs

With this final experiment, we aim to validate H4: "When a client evaluates queries over a federation of interfaces of type *I* under public network latency, performance is similar to that of state-of-the-art SPARQL query federation frameworks." In accordance with the sub-hypotheses, we study whether the evaluation of SPARQL queries over a federation of TPF interfaces results in similar recall and query execution time.

7.4.1. Experimental setup

We implemented the mediator approach from Section 6.4 in the TPF client, which now accepts a *set* of TPF interfaces and a SPARQL query. As the previously considered benchmarks only use a single knowledge graph, we chose the popular FedBench benchmark [64]. The original datasets[12] were first cleaned through the LOD Laundromat service [20], since some of them contained invalid RDF. Then, each dataset was published through a TPF API on a dedicated Amazon EC2 machine (2 virtual CPUs, 7.5 GB RAM) located in the US, each running their own HTTP cache.

The client can execute the same (regular) SPARQL queries as in the single-server scenario, i.e., queries without the `SERVICE` keyword. As a query-mix, we selected the Linked Data (LD), Life Science (LS), and Cross Domain (CD) queries from FedBench, appended with the complex queries (C) by Montoya et al. [65]. The complete query-mix was ran 20 times in sequence on the public Web, accessed from a desktop computer in Belgium in order to represent realistic long-distance latency. Per executed query, the client cache was cold and the timeout was set to 5 minutes.

We compare our measurements with numbers reported by Castillo et al. [66], who tested the following SPARQL endpoint federation systems: ANAPSID [53], ANAPSID EG (ANAPSID using Exclusive Groups), FedX [67] (with a warmed-up cache), and SPLENDID [68]. Castillo et al. obtained their measurements with one client and 10 SPARQL endpoints on different machines in a fast local network. We opted to perform our tests on a public network in order to validate the federated TPF solution within the context of the Web. Measuring recall and execution time on the Web gives an indication of the practical feasibility of real-world TPF-based federation, and a comparison with measurements of the state-of-the-art on a closed network positions TPF relative to an ideal baseline without connection delays.

[12] `https://code.google.com/p/fbench/wiki/Datasets`

7.4.2. Results: recall

Table 3 shows the average result recall for each query. For the queries LD, LS, and CD, the TPF setup reaches a recall of 99% or 100% for all queries except LS3, which reaches only 24% before it timeouts. Most C queries have low recall (6 queries have close to 0%), except for C3 and C5 that reach 100%. In total, 13 queries have less than 100% recall, the causes of which can be assigned into three categories. First, 6 queries (L3, LS5, C2, C4, C7, C9) time out before all results have been computed. Especially for the complex queries, this is due to the amount of data needed by the client to complete the algorithm. Second, 3 queries (CD1, LD2, LS2) reach 0.99% recall on average: they achieve 100% in some runs, but less than 100% in others. This is likely due to the client sending many requests out at once, which can sometimes not all be answered in time. Third, 4 queries (C1, C6, C8, C10) stop before the timeout without full recall. They cause the client to send out so many simultaneous requests that these do not all complete in time, causing the client to abort the execution. The last two causes indicate the client's vulnerability to single request time-outs, leading to the omission of (partial) results. In general, the low recall on the C queries is explained by large numbers of joins (C1) and/or the presence of complex `optional` statements (C6, C7, C8, C9, C10). This requires a large number of triples from all servers, making request time-outs more likely.

Comparing these (real-Web-based) recall measurements to Castillo et al.'s results for state-of-the-art SPARQL endpoint federation systems in a fast local network [66], we observe the following. Overall, TPF and SPLENDID have the highest number of queries with $> 0\%$ and $\geq 10\%$ recall (32 and 30 queries, respectively), TPF has the highest number of queries with $\geq 90\%$ recall (27), and only SPLENDID has more queries with complete results (24). For the LD queries, the TPF setup is the only one not obtaining 100% recall for LD2. On the other hand, all other systems except FedX have less than 100% recall for some of the other LD queries that the TPF setup evaluates with full completion. For the LS queries, none of the systems obtain 100% recall for LS2. The LS3 query, for which TPF only has 24% recall, is evaluated completely by all others. For the CD queries, TPF has the highest recall in all cases and only has less

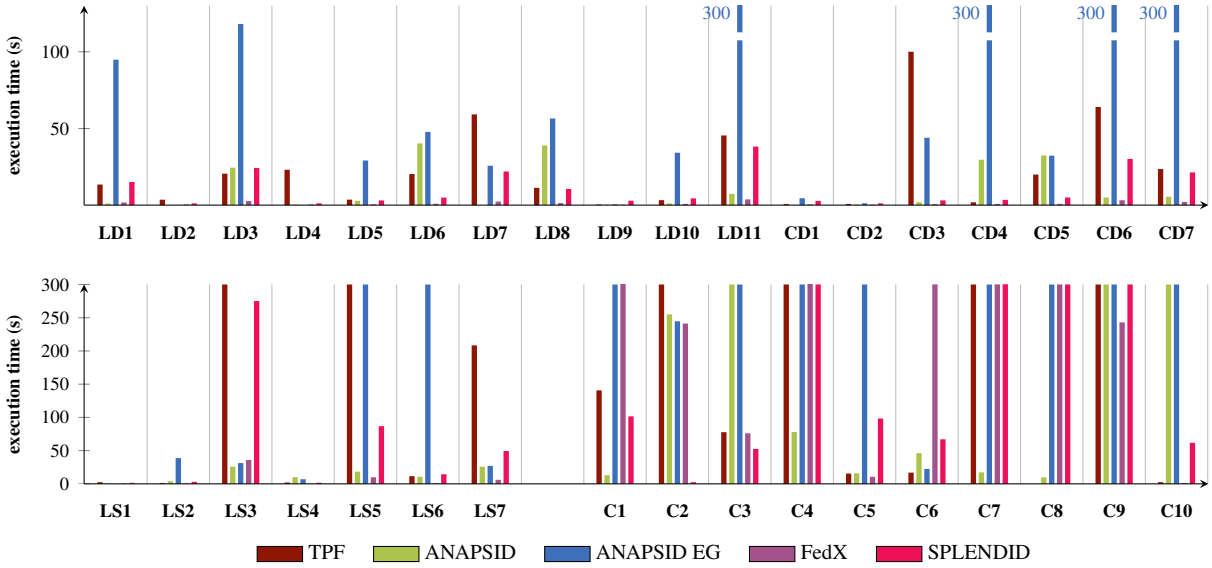

Figure 6: Evaluation times of FedBench query execution on the TPF client/server setup compared to SPARQL endpoint federation systems (timeout of 300s). These measurements should be considered together with the recall for each query (Table 3). The TPF-related measurements were performed in the context of this article; the numbers for the four SPARQL endpoint federation systems are adopted from [66].

than 100% average recall for CD1. Especially ANAPSID and ANAPSID EG score badly on CD queries. In contrast, the C queries are problematic for most systems *except* ANAPSID, which achieves 100% recall for 7 out of 10 queries. This contrast for FedX (and ANAPSID EG) with results of the other queries is related to their usage of so-called *exclusive groups* [67], which is not always the best strategy for the complex queries [69]. Remarkably, TPF and SPLENDID are the only ones to achieve 100% recall for C3, for which ANAPSID and FedX do not find results. Finally, none of the systems seem to obtain significant recall for C7–C10, with the exception of ANAPSID for C7 and C8.

7.4.3. Results: execution time

Next, we study the total query execution times. Our intention is to relate current query execution over TPF collections to the state-of-the-art in SPARQL endpoint federation systems. Hence, we compare to recently published results for the same queries [66]. In general, we notice that the performance gap observed with the single-server experiments in Sections 7.1 and 7.2 becomes smaller in the case of federation. This indicates that the native query decomposition of TPF, combined with light requests and metadata, is more effective in federated environments, and should be examined further.

Figure 6 presents the execution times for all FedBench queries, measured in seconds. The general trend is that the TPF client performs in between ANAPSID (lower bound) and ANAPSID EG (upper bound). TPF is occasionally faster than SPLENDID (LD1, LD3, and notably C5), but sometimes several times as slow (LD7, CD6, LS5, LS7). We should, however, recall that we compare against a TPF setup on a public network, so the TPF measurements include network delay. FedX with a warmed-up cache outperforms all systems for most queries, with a few notable exceptions such as C6. The timings in Figure 6 should, however, be considered together with the recall in Table 3, since not all results might have been obtained in a shorter executions time. FedX, for instance, does not find results for the C queries. A likely explanation for this difference is again FedX's usage of exclusive groups.

Considering the added network delay, a number of queries show promise: CD1, CD2, CD4, LD2, LD5, LD6, LD8, LD9, LD10, LS1, LS2, LS4, and LS6. These queries are answered in less than 4 seconds. This execution time is comparable to that of ANAPSID and SPLENDID, and even approximates FedX. A likely explanation is the presence of highly selective triple patterns in the query, for which the simplicity of TPF requests seems to compensate the overhead of query planning in other systems. For other queries (CD5, CD7, LD1, LD3), the TPF client performs worse than ANAPSID and SPLENDID, but remains within comparable bounds. A probable cause is the presence of patterns like `?x rdf:type ?y`, which can be answered by all data sources. Thus, other systems clearly benefit from prior

	TPF	ANAPSID	ANAPSID EG	FedX (warm)	SPLENDID
LD1	1.00	1.00	1.00	1.00	1.00
LD2	**0.99**	1.00	1.00	1.00	1.00
LD3	1.00	1.00	1.00	1.00	**0.98**
LD4	1.00	1.00	1.00	1.00	1.00
LD5	1.00	**0.00**	**0.00**	1.00	1.00
LD6	1.00	**0.00**	**0.00**	1.00	1.00
LD7	1.00	**0.00**	**0.09**	1.00	1.00
LD8	1.00	1.00	1.00	1.00	1.00
LD9	1.00	1.00	1.00	1.00	1.00
LD10	1.00	1.00	1.00	1.00	1.00
LD11	1.00	1.00	**0.00**	1.00	1.00
LS1	1.00	1.00	1.00	1.00	1.00
LS2	**0.99**	**0.88**	**0.88**	**0.88**	**0.88**
LS3	**0.24**	1.00	1.00	1.00	1.00
LS4	1.00	1.00	1.00	1.00	1.00
LS5	**0.99**	1.00	**0.00**	1.00	1.00
LS6	1.00	**0.00**	**0.00**	1.00	1.00
LS7	1.00	1.00	1.00	**0.09**	1.00
CD1	**0.99**	**0.97**	**0.97**	**0.95**	**0.97**
CD2	1.00	1.00	1.00	1.00	1.00
CD3	1.00	**0.60**	1.00	**0.80**	**0.60**
CD4	1.00	**0.00**	**0.00**	1.00	1.00
CD5	1.00	**0.00**	**0.00**	1.00	1.00
CD6	1.00	**0.00**	**0.00**	1.00	1.00
CD7	1.00	**0.00**	**0.00**	**0.50**	**0.50**
C1	**0.02**	1.00	**0.00**	**0.00**	**0.02**
C2	**0.93**	1.00	**0.00**	**0.00**	1.00
C3	1.00	**0.00**	**0.00**	**0.00**	1.00
C4	**0.00**	1.00	**0.00**	**0.00**	**0.00**
C5	1.00	1.00	**0.00**	**0.00**	1.00
C6	**0.77**	1.00	1.00	**0.00**	1.00
C7	**0.00**	1.00	**0.00**	**0.00**	**0.00**
C8	**0.01**	1.00	**0.00**	**0.00**	**0.00**
C9	**0.01**	**0.00**	**0.00**	**0.00**	**0.00**
C10	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
# queries					
= 1.00	22	21	14	20	24
≥ 0.90	27	23	15	21	26
≥ 0.10	30	24	16	24	30
> 0.00	32	24	17	25	32

Table 3: Recall of FedBench query execution on the TPF client/server setup compared to SPARQL endpoint federation systems (timeout of 300s). All occurrences of incomplete recall are highlighted. The TPF-related measurements were performed in the context of this article; the numbers for the other four systems are adapted from [66].

source selection, which the current TPF client does not use.

For the remaining regular FedBench queries, the TPF client distinctly reveals its limitations. These queries (LS3, LS5) time out, or execute significantly slower (CD3, CD6, LD7, LS7) than SPARQL endpoint federation systems. They contain common predicates like `owl:sameAs` or `foaf:name` that trigger requests to all interfaces. Additionally, a high number of subject joins causes inefficiencies, since they potentially produce many "membership requests" to all sources, checking whether a triple is present or not. A possible enhancement is to include metadata that prevents this, at the expense of a more costly server-side interface [70]. Another cause is the presence of a `FILTER` statement (LS7), which is currently executed client-side. This indicates room for interface extensions for such clauses [71].

The limitations of TPF become more apparent with the C queries, 4 of which time out and another 4 end prematurely (as discussed above). The high number of produced HTTP requests (3,692 on average) caused by the many triple patterns, contributes significantly to this delay. SPLENDID, FedX, and ANAPSID show similar results, but fail on different queries. For the queries where TPF reaches complete recall (C3, C5), the total execution time is comparable and even outperforms ANAPSID. These findings, measured on the public Web, motivate a more in-depth study of complex queries for TPF, to discover possible client or server enhancements.

8. Conclusions and future work

How useful are *limited* Linked Data APIs for the Semantic Web? If we design them the right way, they allow more with higher performance than intuition would predict—if we are willing to accept the combination of trade-offs they bring. Then again, each Web API to Linked Data comes with its own trade-off mix, so the question becomes: is it worthwhile to look at *other* trade-offs than those that the currently existing APIs already bring? The Triple Pattern Fragments API introduced in this article is a deliberately simple one: clients can only ask for `?s ?p ?o` patterns. Its responses explicitly describe the interface, so clients can discover without external documentation what the interface supports and how to access it. The aim of the interface is to lower the cost for knowledge publishers to offer live queryable data on the open Web, the availability of which is currently low [3, 5], hindering application development. Have we found a candidate interface with TPFs, either for direct use or as the base of other APIs, and is this direction viable for future research and development?

8.1. Discussion of the results

The results of the first experiment (Section 7.1) indicate that TPF query execution succeeds in reducing server usage, at the cost of increased query times. TPF servers cope better with increasing numbers of clients than SPARQL endpoints. They have a generally low and regular CPU load (H1.1), accompanied by less variation in response time (H1.3). Furthermore, querying benefits strongly from regular HTTP caching (H1.2), which can be added at any point in the network. These three facts validate the hypothesis H1 that the interface reduces the server-side cost to publish knowledge graphs. This is all the more remarkable since, to allow comparisons with other work, these results were obtained with an *existing* SPARQL benchmark that focuses on performance, not server cost. Even though certain queries make it difficult for an LDF client to find *all* results within the timeout window the *first* results to all queries arrive before the timeout period. In that sense, TPF query execution challenges the traditional query paradigm of *"send the query – wait for the server – act on all results"* and proposed instead *"start the query – act on each incoming result"*. In other words, since we know that we will be waiting longer for results anyway, we should focus on handling individual results as they arrive rather than on processing the whole result set at the end. Concretely, instead of waiting for queries containing constructs TPF clients cannot implement with a non-blocking iterator (e.g., `ORDER BY`), an application might prefer to use queries without such constructs and then perform additional result transformations itself as part of a self-updating interface.

The second experiment (Section 7.2) generalizes these findings and validates the hypothesis H2 that this behavior extends to real-world knowledge graphs such as DBpedia. A vast majority of queries stays well below the 1 second limit, despite being affected by the knowledge graph size. We note a strong influence of the type of query, especially when non-BGP SPARQL constructs are involved.

Experiment 3 (Section 7.3) falsifies the hypothesis H3. Although more compact formats show a decrease in query execution time, these findings no longer apply when responses are compressed by GZIP, commonly used within the HTTP protocol. Also, the serialization and deserialization costs can be decisive, especially if they involve relatively few triples—which is the case for typical page sizes (e.g., 100) of a TPF interface. The experiment shows the importance of carefully considering serializations. Even though removing or shortening metadata and control triples would work for specialized TPF clients, the applicability of the application would be narrowed.

The extension to federated querying in Experiment 4 (Section 7.4) shows the scalability of the approach toward multiple knowledge graphs on the Web. The measurements show a competitive recall compared to the state-of-the-art, validating H4.1. Even though low server cost—not performance—is the main concern of TPFs, certain queries perform comparably as with state-of-the-art SPARQL federation systems. Therefore, we validate hypothesis H4.2—and, together with H4.1, thus also H4. Some of the queries still execute more slowly or time out, however, we emphasize that the TPF setup was tested on the public Web, whereas the other federation engines were considered in a fast local network. Hence, if we focus on improving some of the remaining bottlenecks, either through query optimizations [49] or limited interface extensions [70, 71], TPF might serve as a realistic solution to federations of knowledge graphs on the Web. Moreover, the experiments show a TPF client *without* a prior source selection step; the runtime source elimination of sources with zero-result ancestor patterns thus seems adequate for a number of cases. Any of the existing source selection algorithms could be incorporated beforehand in order to reduce the number of considered sources, but doing so might involve additional metadata and/or computations, and thereby influence the measured parameters.

8.2. Usage potential

An important concern is whether we have practical evidence to assume that the TPF interface is a possible candidate for adoption. To address this concern, we can point to the already existing TPF interfaces on the Web. Since October 2014, the DBpedia Foundation provides the popular DBpedia knowledge graph as TPFs[13] in addition to data dumps, Linked Data documents, and SPARQL endpoint. The TPF interface had an uptime of 99.99% during its first 9 months, handling 16,776,170 requests [72]. Since February 2015, the LOD Laundromat [20] publishes more than 650,000 knowledge graphs from the Web not only as data dumps, but also as TPFs [73]. While these knowledge graphs are currently published on a single machine, the LOD Laundromat source code[14] is publicly available for a straightforward setup of large-scale TPF servers. In addition, a few independent data publishers have made knowledge graphs available as TPFs through our software or other

[13] `http://fragments.dbpedia.org/`
[14] `https://github.com/LODLaundry/`

libraries.[15] Because the number of fragments per knowledge graph is finite, TPFs of smaller knowledge graphs can be pregenerated and published on the Web through free hosting platforms, while enabling live querying [74].

8.3. Future directions

The TPF interface proposed in this article is not the final solution to querying knowledge graphs on the Web. Instead, our main goals have been to *i)* gather evidence that low-cost interfaces can enable the efficient execution of common SPARQL queries; *ii)* provide a starting point for further interface development. While there is still room to improve query evaluation efficiency over TPFs (e.g., [48, 49]), this will eventually reach a limit. Therefore, TPF responses explicitly describe the interface, allowing clients to find out dynamically if this and other features are supported [8]. If a new feature is proposed to filter data in a different way, this can be implemented on top of TPFs. Clients that support TPFs will still be able to execute their algorithm; clients that also support the additional feature can make use of it if the interface indicates so.

In future work, we aim to explore such additional features and other possible interface along the axis of Figure 1—and invite others to do the same. In particular, we consider features such as sorting (in order to optimize `ORDER BY` clauses) and substring search (in order to support `FILTER` more efficiently) [71]. Another possibility is the incorporation of additional metadata in the interface, which allows clients to further reduce the number of requests [70]. Each feature can have an impact on server cost and client efficiency, sometimes in very subtle ways. For instance, even though a certain feature might introduce extra server complexity, this feature could reduce the number of client requests for a certain task, perhaps making the one costly operation more efficient than several inexpensive ones. These are the kind of questions that future research should answer. All of these supported features must take into account real-world concerns, such as rapidly changing data, a direction we recently started to explore [75]. Additionally, we consider the creation of SPARQL benchmarks that focus on other aspects besides performance, such as cost and cache reuse.

Interestingly, query evaluation over TPFs reveals a pattern in which clients and servers engage in a dialog to tackle a problem, asking questions such as "What kinds of fragments do you offer?" and "How can I access them?" This dialog becomes possible because servers *enable* clients to perform complex behavior on the Web, rather than centralizing the complexity. As different interfaces in addition to TPF evolve, explicitly describing their functionality inside their responses, we come closer to the original vision of the Semantic Web [4] in which *clients*—not servers—are intelligent. An important challenge to arrive at such a queryable Web is to find realistic balances of client/server trade-offs, to which TPFs are an initial step.

[15] `http://linkeddatafragments.org/data/`

Acknowledgments

The research activities in this article were funded by Ghent University, iMinds, the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), and the European Union. R. Verborgh is a postdoctoral fellow of the Research Foundation Flanders.

References

[1] C. Bizer, T. Heath, T. Berners-Lee, Linked Data – the story so far, International Journal on Semantic Web and Information Systems 5 (3) (2009) 1–22.

[2] M. Schmachtenberg, C. Bizer, H. Paulheim, Adoption of the linked data best practices in different topical domains, in: P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, C. Goble (Eds.), Proceedings of the 13th International Semantic Web Conference, Vol. 8796 of Lecture Notes in Computer Science, Springer, 2014, pp. 180–196.

[3] I. Ermilov, M. Martin, J. Lehmann, S. Auer, Linked open data statistics: Collection and exploitation, in: P. Klinov, D. Mouromtsev (Eds.), Knowledge Engineering and the Semantic Web, Vol. 394 of Communications in Computer and Information Science, Springer, 2013, pp. 242–249.
URL `http://dx.doi.org/10.1007/978-3-642-41360-5_19`

[4] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Scientific American 284 (5) (2001) 34–43.

[5] C. Buil-Aranda, A. Hogan, J. Umbrich, P.-Y. Vandenbussche, SPARQL Web-querying infrastructure: Ready for action?, in: H. Alani, L. Kagal, A. Fokoue, P. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. Noy, C. Welty, K. Janowicz (Eds.), Proceedings of the 12th International Semantic Web Conference, 2013.

[6] R. Verborgh, M. Vander Sande, P. Colpaert, S. Coppens, E. Mannens, R. Van de Walle, Web-scale querying through Linked Data Fragments, in: C. Bizer, T. Heath, S. Auer, T. Berners-Lee (Eds.), Proceedings of the 7th Workshop on Linked Data on the Web, 2014.

[7] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, R. Van de Walle, Querying datasets on the Web with high availability, in: P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, C. Goble (Eds.), Proceedings of the 13th International Semantic Web Conference, Vol. 8796 of Lecture Notes in Computer Science, Springer, 2014, pp. 180–196.

[8] R. Verborgh, E. Mannens, R. Van de Walle, Bottom-up web APIs with self-descriptive responses, in: Proceedings of the 6th International Workshop on Modeling Social Media, 2015.

[9] R. Angles, C. Gutierrez, The expressive power of SPARQL, in: Proceedings of the 7th International Semantic Web Conference, Lecture Notes in Computer Science, Springer, 2008, pp. 114–129.

[10] R. B. Miller, Response time in man-computer conversational transactions, in: Proceedings of the December 9-11, 1968, fall joint computer conference, part I, ACM, 1968, pp. 267–277.

[11] S. K. Card, G. G. Robertson, J. D. Mackinlay, The information visualizer, an information workspace, in: Proceedings of the SIGCHI Conference on Human factors in computing systems, ACM, 1991, pp. 181–186.

[12] A. P. Sheth, J. A. Larson, Federated database systems for managing distributed, heterogeneous, and autonomous databases, ACM Computing Surveys 22 (3) (1990) 183–236.

[13] R. Cyganiak, D. Wood, M. Lanthaler (Eds.). RDF 1.1 concepts and abstract syntax, Recommendation, World Wide Web Consortium (Feb. 2014).
URL http://www.w3.org/TR/rdf11-concepts/

[14] S. Harris, A. Seaborne (Eds.). SPARQL 1.1 query language, Recommendation, W3C (Mar. 2013).
URL http://www.w3.org/TR/sparql11-query/

[15] R. Verborgh, Serendipitous web applications through semantic hypermedia, Ph.D. thesis, Ghent University, Ghent, Belgium (Feb. 2014).

[16] R. T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. thesis, University of California (2000).

[17] M. Amundsen, Hypermedia types, in: E. Wilde, C. Pautasso (Eds.), REST: From Research to Practice, Springer, 2011, pp. 93–116.

[18] R. T. Fielding, REST APIs must be hypertext-driven (Oct. 2008).
URL http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[19] M. Lanthaler, C. Gütl, Hydra: A vocabulary for hypermedia-driven Web APIs, in: Proceedings of the 6th Workshop on Linked Data on the Web, 2013.

[20] W. Beek, L. Rietveld, H. Bazoobandi, J. Wielemaker, S. Schlobach, LOD Laundromat: a uniform way of publishing other people's dirty data, in: P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, C. Goble (Eds.), Proceedings of the 13th International Semantic Web Conference, Vol. 8796 of Lecture Notes in Computer Science, Springer, 2014, pp. 213–228.

[21] L. Feigenbaum, G. T. Williams, K. G. Clark, E. Torres (Eds.). SPARQL 1.1 protocol, Recommendation, W3C (Mar. 2013).
URL http://www.w3.org/TR/sparql11-protocol/

[22] O. Erling, I. Mikhailov, Virtuoso: RDF support in a native RDBMS, in: R. de Virgilio, F. Giunchiglia, L. Tanca (Eds.), Semantic Web Information Management, Springer, 2010, pp. 501–519.

[23] M. Grobe, RDF, Jena, SPARQL and the Semantic Web, in: Proceedings of the 37th Annual ACM SIGUCCS Fall Conference: Communication and Collaboration, 2009.

[24] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Transactions on Database Systems 34 (3) (2009) 16:1–16:45.

[25] O. Erling, SEMANTiCS 2014 (part 3 of 3): Conversations (Aug. 2014).
URL http://www.openlinksw.com/dataspace/doc/oerling/weblog/Orri%20Erling%27s%20Blog/1815

[26] O. Hartig, An overview on execution strategies for Linked Data queries, Datenbank-Spektrum 13 (2) (2013) 89–99.

[27] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, A. Polleres, Comparing data summaries for processing live queries over linked data, World Wide Web 14 (5–6) (2011) 495–544.

[28] O. Hartig, C. Bizer, J.-C. Freytag, Executing SPARQL queries over the Web of Linked Data, in: A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, K. Thirunarayan (Eds.), Proceedings of the 8th International Semantic Web Conference, Springer, 2009, pp. 293–309.

[29] S. Speicher, J. Arwe, A. Malhotra (Eds.). Linked Data Platform 1.0, Recommendation, W3C (Feb. 2015).
URL http://www.w3.org/TR/ldp/

[30] C. Ogbuji (Ed.). SPARQL 1.1 Graph Store HTTP Protocol, Recommendation, W3C (Mar. 2013).
URL http://www.w3.org/TR/sparql11-http-rdf-update/

[31] Linked Data API, retrieved at 2015-08-02.
URL https://code.google.com/p/linked-data-api/

[32] L. Matteis, Restpark: Minimal RESTful API for retrieving RDF triples (2013).
URL http://lmatteis.github.io/restpark/restpark.pdf

[33] E. Wilde, M. Hausenblas, RESTful SPARQL? You name it! – Aligning SPARQL with REST and resource orientation, in: Proceedings of the 4th Workshop on Emerging Web Services Technology, ACM, 2009, pp. 39–43.

[34] O. Hartig, J. Zhao, Publishing and consuming provenance metadata on the Web of Linked Data, in: D. L. McGuinness, J. R. Michaelis, L. Moreau (Eds.), Proceedings of the 3rd International Provenance and Annotation Workshop, 2010.

[35] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, M. Arias, Binary RDF representation for publication and exchange (HDT), Journal of Web Semantics 19 (2013) 22–41.

[36] M. Schmidt, M. Meier, G. Lausen, Foundations of SPARQL query optimization, in: Proceedings of the 13th International Conference on Database Theory, 2010, pp. 4–33.

[37] P. Barceló, Querying graph databases, in: Proceedings of the 32nd Symposium on Principles of Database Systems (PODS), 2013.

[38] O. Hartig, SPARQL for a Web of Linked Data: Semantics and Computability, in: E. Simperl, P. Cimiano, A. Polleres, O. Corcho, V. Presutti (Eds.), Proceedings of the 9th Extended Semantic Web Conference, Springer, 2012.

[39] O. Hartig, How caching improves efficiency and result completeness for querying Linked Data, in: C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas (Eds.), Proceedings of the 4th Workshop on Linked Data on the Web, 2011.

[40] G. Carothers (Ed.). RDF 1.1 N-Quads, Recommendation, W3C (Feb. 2014).
URL http://www.w3.org/TR/n-quads/

[41] C. Bizer, R. Cyganiak. RDF 1.1 TriG, Recommendation, W3C (Feb. 2014).
URL http://www.w3.org/TR/trig/

[42] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, N. Lindström. JSON-LD 1.0, Recommendation, W3C (Jan. 2014).
URL http://www.w3.org/TR/json-ld/

[43] R. Verborgh. Linked Data Fragments, Unofficial draft, Hydra W3C Community Group.
URL http://www.hydra-cg.com/spec/latest/linked-data-fragments/

[44] G. Graefe, Query evaluation techniques for large databases, ACM Computing Surveys 25 (2) (1993) 73–169.

[45] J. M. Hellerstein, M. Stonebraker, J. Hamilton, Architecture of a database system, Foundations and Trends in Databases 1 (2) (2007) 141–259. doi:10.1561/1900000002.

[46] J. M. Smith, P. Y.-T. Chang, Optimizing the performance of a relational algebra database interface, Communications of the ACM 18 (10) (1975) 568–579.

[47] S. B. Yao, Optimization of query evaluation algorithms, ACM Transactions on Database Systems 4 (2) (1979) 133–155.

[48] M. Acosta, M.-E. Vidal, Networks of linked data eddies: An

adaptive Web query processing engine for RDF data, in: M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, K. Thirunarayan, S. Staab (Eds.), The Semantic Web – ISWC 2015, Vol. 9366 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 111–127.

[49] J. Van Herwegen, R. Verborgh, E. Mannens, R. Van de Walle, Query execution optimization for clients of triple pattern fragments, in: F. Gandon, M. Sabou, H. Sack, C. d'Amato, P. Cudré-Mauroux, A. Zimmermann (Eds.), Proceedings of the 12th Extended Semantic Web Conference, 2015.

[50] G. Wiederhold, Mediators in the architecture of future information systems, Computer 25 (3) (1992) 38–49.

[51] M. T. Özsu, P. Valduriez, Principles of distributed database systems, Springer Science & Business Media, 2011, Ch. 9.2, p. 299.

[52] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, A.-C. Ngonga Ngomo, A fine-grained evaluation of SPARQL endpoint federation systems, Semantic Web Journal Accepted for publication.

[53] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, E. Ruckhaus, ANAPSID: An adaptive query processing engine for SPARQL endpoints, in: L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Noy, E. Blomqvist (Eds.), The Semantic Web – ISWC 2011, Vol. 7031 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 18–34.

[54] M. Saleem, A.-C. N. Ngomo, Hibiscus: Hypergraph-based source selection for SPARQL endpoint federation, in: The Semantic Web: Trends and Challenges, Springer, 2014, pp. 176–191.

[55] K. Hose, R. Schenkel, Towards benefit-based RDF source selection for SPARQL queries, in: Proceedings of the 4th International Workshop on Semantic Web Information Management, SWIM '12, ACM, New York, NY, USA, 2012, pp. 2:1–2:8.

[56] C. Bizer, A. Schultz, The Berlin SPARQL benchmark, International Journal on Semantic Web and Information Systems 5 (2) (2009) 1–24.

[57] M. Morsey, J. Lehmann, S. Auer, A.-C. Ngonga Ngomo, DBpedia SPARQL benchmark – performance assessment with real queries on real data, in: Proceedings of the 9th International Semantic Web Conference, 2011.

[58] D. Beckett. RDF 1.1 N-Triples, Recommendation, W3C (Feb. 2014).
URL http://www.w3.org/TR/n-triples/

[59] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, G. Carothers. RDF 1.1 Turtle, Recommendation, W3C (Feb. 2014).
URL http://www.w3.org/TR/turtle/

[60] A. Seaborne, RDF binary using Apache Thrift.
URL http://afs.github.io/rdf-thrift/

[61] J. Broekstra, Binary RDF in Sesame (Nov. 2011).
URL http://www.rivuli-development.com/2011/11/binary-rdf-in-sesame/

[62] J. D. Fernández, A. Llaves, O. Corcho, Efficient RDF interchange (ERI) format for RDF data streams, in: P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, C. Goble (Eds.), Proceedings of the 13th International Semantic Web Conference, Springer, 2014, pp. 244–259.

[63] G. Aluç, O. Hartig, M. T. Özsu, K. Daudjee, Diversified stress testing of RDF data management systems, in: P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, C. Goble (Eds.), Proceedings of the 13th International Semantic Web Conference, Springer, 2014, pp. 197–212.

[64] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, T. Tran, Fedbench: A benchmark suite for federated semantic data query processing, in: Proceedings of the International Semantic Web Conference, Springer, 2011, pp. 585–600.

[65] G. Montoya, M.-E. Vidal, O. Corcho, E. Ruckhaus, C. Buil-Aranda, Benchmarking federated SPARQL query engines: are existing testbeds enough?, in: P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, E. Blomqvist (Eds.), Proceedings of the 11th International Semantic Web Conference, Springer, 2012, pp. 313–324.

[66] S. Castillo, G. Palma, M.-E. Vidal, G. Montoya, M. Acosta, Fed-DSATUR decompositions, retrieved at 2015-09-01.
URL http://scast.github.io/fed-dsatur-decompositions/

[67] A. Schwarte, P. Haase, K. Hose, R. Schenkel, M. Schmidt, FedX: Optimization techniques for federated query processing on linked data, in: L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Noy, E. Blomqvist (Eds.), The Semantic Web – ISWC 2011, Vol. 7031 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 601–616.
URL http://dx.doi.org/10.1007/978-3-642-25073-6_38

[68] O. Görlitz, S. Staab, SPLENDID: SPARQL endpoint federation exploiting VOID descriptions, in: O. Hartig, A. Harth, J. Sequeda (Eds.), Proceedings of the Second International Workshop on Consuming Linked Data, 2011.

[69] G. Montoya, M.-E. Vidal, M. Acosta, A heuristic-based approach for planning federated SPARQL queries, in: J. F. Sequeda, A. Harth, O. Hartig (Eds.), Proceedings of the Third International Workshop on Consuming Linked Data, 2012.

[70] M. Vander Sande, R. Verborgh, J. Van Herwegen, E. Mannens, R. Van de Walle, Opportunistic Linked Data querying through approximate membership metadata, in: M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, S. Staab (Eds.), Proceedings of the 14th International Semantic Web Conference, 2015.

[71] J. Van Herwegen, L. De Vocht, R. Verborgh, E. Mannens, R. Van de Walle, Substring filtering for low-cost Linked Data interfaces, in: M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, S. Staab (Eds.), Proceedings of the 14th International Semantic Web Conference, 2015.

[72] R. Verborgh, DBpedia's Triple Pattern Fragments: Usage patterns and insights, in: F. Gandon, C. Guéret, S. Villata, J. Breslin, C. Faron-Zucker, A. Zimmermann (Eds.), Proceedings of the 12th Extended Semantic Web Conference – Satellite Events, 2015.

[73] L. Rietveld, R. Verborgh, W. Beek, M. Vander Sande, S. Schlobach, Linked Data-as-a-Service: The Semantic Web redeployed, in: F. Gandon, M. Sabou, H. Sack, C. d'Amato, P. Cudré-Mauroux, A. Zimmermann (Eds.), Proceedings of the 12th Extended Semantic Web Conference, 2015.

[74] L. Matteis, R. Verborgh, Hosting queryable and highly available Linked Data for free, in: R. Verborgh, E. Mannens (Eds.), Proceedings of the ISWC Developers Workshop, 2014.
URL http://ceur-ws.org/Vol-1268/paper3.pdf

[75] M. Vander Sande, R. Verborgh, E. Mannens, R. Van de Walle, Updating SPARQL results in real-time with client-side fragment patching, in: A. Filipowska, R. Verborgh, A. Polleres (Eds.), Proceedings of the 11th International Conference on Semantic Systems – Posters and Demos, 2015.