Querying Datasets on the Web with High Availability

Ruben Verborgh[1], Olaf Hartig[2], Ben De Meester[1], Gerald Haesendonck[1], Laurens De Vocht[1], Miel Vander Sande[1], Richard Cyganiak[3], Pieter Colpaert[1], Erik Mannens[1], and Rik Van de Walle[1]

[1] Ghent University – iMinds, Belgium
{firstname.lastname}@ugent.be
[2] University of Waterloo, Canada
ohartig@uwaterloo.ca
[3] Digital Enterprise Research Institute, NUI Galway, Ireland
richard@cyganiak.de

Abstract. As the Web of Data is growing at an ever increasing speed, the lack of reliable query solutions for live public data becomes apparent. SPARQL implementations have matured and deliver impressive performance for public SPARQL endpoints, yet poor availability—especially under high loads—prevents their use in real-world applications. We propose to tackle this availability problem by defining triple pattern fragments, a specific kind of Linked Data Fragments that enable low-cost publication of queryable data by moving intelligence from the server to the client. This paper formalizes the Linked Data Fragments concept, introduces a client-side SPARQL query processing algorithm that uses a dynamic iterator pipeline, and verifies servers' availability under load. The results indicate that, at the cost of lower performance, query techniques with triple pattern fragments lead to high availability, thereby allowing for reliable applications on top of public, queryable Linked Data.

Keywords: Linked Data, Linked Data Fragments, querying, availability, scalability, SPARQL.

1 Introduction

The past few years, the performance of SPARQL endpoints has increased steadily. In spite of all this progress, reliable queryable access to public Linked Data datasets largely remains impossible due to the low availability percentages of public SPARQL endpoints. As of end-2013, the average SPARQL endpoint is down for more than 1.5 days *each month* [4]. This means we cannot build reliable applications on top of queryable public data. No matter how fast SPARQL implementations become, if their availability does not increase, no one will take the risk of depending on public data providers to provide querying for their applications. Availability, not performance, is currently the main threat to the success of the Semantic Web as a viable technology for today's challenges.

P. Mika et al. (Eds.) ISWC 2014, Part I, LNCS 8796, pp. 180–196, 2014.
© Springer International Publishing Switzerland 2014

To circumvent the availability issue, consumers who want to query public data typically download a data dump and host their own private SPARQL endpoint. While this seems to solve the issue, it has the following drawbacks:

- Hosting an endpoint requires (possibly expensive) infrastructural support and involves (often manual) set-up and maintenance.
- The data in the endpoint is not guaranteed to be up-to-date.
- Each dataset required by any of the desired queries must be fully loaded into the endpoint, even if only a small part of that dataset is actually needed.

Furthermore, querying a local machine can hardly be considered *Web* querying, as everything happens offline. Making the Semantic Web vision scalable by downloading and querying all data locally seems an unsatisfactory paradox.

In order to advance towards a solution for high-availability Web querying, *Linked Data Fragments (LDFs)* [27] were proposed as a framework to analyze all possible ways of publishing parts of a Linked Data dataset, ranging from SPARQL endpoints with highly specific results to data dumps that contain the entire dataset. In particular, this framework allows to define specific types of fragments that can be generated with minimal effort by servers, while still enabling efficient client-side querying. One such type are *triple pattern fragments* (formerly called *basic* Linked Data Fragments [27]), which offer triple-pattern-based access to a dataset.

In this paper, we show that client-side query processing using triple pattern fragments allows live querying with high availability and scalability of public datasets. This result demonstrates that this enables reliable query execution on the Web of Data, with minimal server-side cost. First, the next section discusses related work on querying RDF-based datasets on the Web. We then provide a formalization of Linked Data Fragments in Section 3, followed by a client-side, iterator-based query execution algorithm in Section 4. Section 5 contains the availability evaluation and discussion. We conclude the paper in Section 6.

2 Related Work

On the current Web, several HTTP interfaces that provide access to triple-based data are available. We will discuss public SPARQL endpoints, Linked Data publishing, and other HTTP interfaces for triples, as well as their querying methods.

2.1 Public SPARQL Endpoints

The SPARQL language [12] is the W3C standard to query a collection of RDF triples [16]. Many triple stores, such as Virtuoso [5] and Jena TDB [11], offer a SPARQL interface. Even though current SPARQL interfaces offer high performance [3, 19, 22], individual queries can consume a significant amount of server processor time and memory. In fact, it has been shown that the evaluation problem for SPARQL is PSPACE-complete [21]. Like any high-performance database server, SPARQL servers with high demand are generally expensive to host, which is further complicated for public servers because of unpredictable loads.

The current de-facto way for providing queryable access to triples on the Web is the SPARQL protocol [6]: clients send SPARQL queries through a specific HTTP interface; the server executes these queries and responds with their results. This contrasts with the majority of machine-to-machine HTTP interactions on the Web, where the server implements a rigidly structured API through which clients access the data. Such APIs purposely limit the kind of queries a client can ask, as it allows those servers to place a bound on the computation time needed for each API request [27]. With SPARQL endpoints, clients can demand the execution of arbitrarily complicated queries[1] [6]. Furthermore, since each client requests unique, highly specific queries, regular HTTP caching mechanisms are ineffective, since they can only optimize repeated identical requests.

These factors contribute to the low availability of public SPARQL endpoints, which is documented extensively [4]. It is important to note that this low availability is *not* the result of poor performance: as indicated by multiple benchmarks [3, 19, 22], many SPARQL implementations deliver very high performance. Instead, it is the consequence of the architectural decision of the current SPARQL protocol, which demands the server responds to highly complex requests [27]. This makes providing reliable public SPARQL endpoints an exceptionally difficult challenge, incomparable to hosting any other public HTTP server.

2.2 Linked Data

Perhaps the most well-known alternative interface to RDF triples is described by the Linked Data principles [2] which, not coincidentally, align with the Web's architectural constraints [8]. The principles require servers to publish documents ("subject pages") with triples about specific entities, which a client can access through their entity-specific URI, a practice which is called *dereferencing*. Each of these Linked Data documents contains triples that mention URIs of other entities, which can be dereferenced in turn. Serving such documents is like serving HTML files, which does not require much processor time or memory, so hosting them at low cost is straightforward. Several Linked Data querying techniques [14] aim to use dereferencing to solve SPARQL queries over the Web of Data. This process happens client-side, so the availability of servers is not impacted.

The Linked Data publishing and querying strategy has two main drawbacks. First, query execution times are high, and many queries cannot be solved (efficiently). For example, it is nearly impossible to directly answer the following seemingly simple query for any given dataset:

```
SELECT ?person WHERE { ?person a <http://xmlns.com/foaf/0.1/Person> }
```

A client could try to fetch the URL `http://xmlns.com/foaf/0.1/Person` but, because of the Web's unidirectional linking structure, the document at that URL cannot possibly link to all instances of `foaf:Person`. In fact, it does not link to any, so the query execution yields an empty result.

[1] Many endpoints allow to only expose a subset of all SPARQL queries, for instance, by limiting the allowed execution time. However, even under those circumstances, the availability of public SPARQL endpoints remains low [4].

Second, documents about an entity are looked up through dereferencing, and the URI of an entity only points to the single document on the server that hosts the domain of that URI. For example, the URI `http://dbpedia.org/resource/Barack_Obama` leads to triples about Barack Obama on the DBpedia server, not to the triples hosted on other sources that also have data about Barack Obama, such as the BBC or the New York Times. And even though DBpedia could link to those sources, this is entirely up to the server's discretion. While anybody can reuse the DBpedia URI to add triples about an entity, it is highly unlikely that those triples are considered by Linked Data querying. This contrasts with SPARQL endpoints, which can provide data about resources with *any* URI.

2.3 Other HTTP Interfaces to RDF Triples

Finally, several other HTTP interfaces for triples have been designed. Strictly speaking, the most trivial HTTP interface is a data dump, which is a single-file representation of a (part of a) dataset. As discussed in Section 1, this allows consumers to set up a private query endpoint. Typical HTTP APIs offer more granular access, albeit still far less flexible than SPARQL endpoints.

The Linked Data Platform [23] is a read/write HTTP interface for Linked Data, scheduled to become a W3C recommendation. It details several concepts that extend beyond the Linked Data principles, such as containers and write access. However, 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 [20], 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 [17] and Restpark [18]. Some of them aim to bridge the gap between the SPARQL protocol and the REST architectural style underlying the Web [28]. However, none of these proposals are widely used at the moment and no query engines for them are implemented to date.

3 Linked Data Fragments

3.1 Concept and Context

What all of the above interfaces have in common is that, in one sense or another, they publish certain *fragments* of a Linked Data dataset. A SPARQL endpoint response, a Linked Data document, and a data dump each offer specific parts of all triples of a given collection. Rather than presenting them as fully distinct approaches, we uniformly call the result of each request to such interfaces a *Linked Data Fragment (LDF)* [25, 27]. As Fig. 1 shows, each kind of fragment mainly differs in its specificity. Depending on this, the workload to compute answers to queries is divided differently between clients and servers. The key to efficient and reliable Web querying is to find fragments that strike an optimal balance between client and server effort. Before we examine particular options, let us define formally what LDFs are.

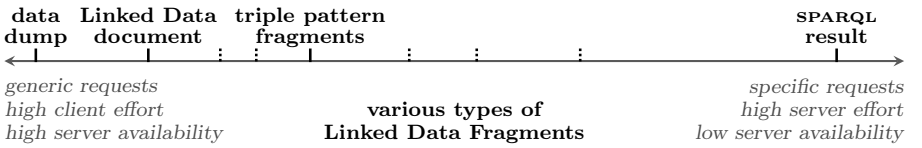


Fig. 1. All HTTP triple interfaces offer Linked Data Fragments of a dataset. They differ in the specificity of the data they contain, and thus the effort needed to create them.

3.2 Formal Definitions

As a basis for our formalization, we use the following concepts of the RDF data model [16] and the SPARQL query language [12]. We write $\mathcal{U}$, $\mathcal{B}$, $\mathcal{L}$, and $\mathcal{V}$ to denote the sets of all URIs, blank nodes, literals, and variables, respectively. Then, $\mathcal{T} = (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is the (infinite) set of all *RDF triples*. Any tuple $tp \in (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V})$ is a *triple pattern*. Any finite set of such triple patterns is a *basic graph pattern* (BGP). Any more complex SPARQL *graph pattern*, typically denoted by P, combines triple patterns (or BGPs) using specific operators [12,21]. The standard (set-based) query semantics for SPARQL defines the *query result* of such a graph pattern P over a set of RDF triples $G \subseteq \mathcal{T}$ as a set that we denote by $[\![P]\!]_G$ and that consists of partial mappings $\mu : \mathcal{V} \to (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$, which are called *solution mappings*. 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.

For the sake of a more straightforward formalization, in this paper, we assume without loss of generality that every dataset 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 dataset contains triples that somehow belong together; they have been selected based on some condition, which we abstract through the notion of a selector:

Definition 1 (selector). *A* selector *is a partial function* $s : 2^{\mathcal{T}} \to \{\texttt{true}, \texttt{false}\}$.

A more concrete type of this abstract notion are triple pattern selectors, which select triples that match a certain triple pattern:

Definition 2 (triple pattern selector). *Given a triple pattern tp, the* triple pattern selector *for tp is the selector* s_{tp} *that, for any singleton set* $\{t\} \subseteq 2^{\mathcal{T}}$*, is defined by*

$$\mathrm{s}_{tp}(\{t\}) = \begin{cases} \texttt{true} & \text{if } t \text{ is a matching triple for } tp, \\ \texttt{false} & \text{else.} \end{cases}$$

When publishing data on the Web, we should equip its representations with hypermedia controls [1,8,9]. We encounter them on a daily basis when browsing HTML pages; they are usually present as hyperlinks or forms. What all these controls have in common is that, given some (possibly empty) input, they result in our browser performing a request for a specific URL.

Definition 3 (control). *A* control *is a function that maps from some set to* $\mathcal{U}$.

In particular, we are interested in controls whose domain is a set of selectors, as they allow to create URLs that correspond to data matching those selectors.

By now, we have introduced all elements necessary to define fragments of an RDF-based dataset.

Definition 4 (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;*
- *Γ is a set consisting of all subsets of G that match selector s, that is, for every $G' \subseteq G$ it holds that $G' \in \Gamma$ if and only if $G' \in \mathrm{dom}(s)$ and $s(G') = \texttt{true}$;*
- *M is a finite set of (additional) RDF triples, including triples that represent metadata for f; and*
- *C is a finite set of 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. For example, the result of a SPARQL `CONSTRUCT` query is an LDF where the selector is the query, the metadata set is empty, and the control set contains a SPARQL endpoint URL [6].

Informally, we distinguish different types of LDFs, each of which represents LDFs that have the same type of selector and the same kind of conditions on their metadata M and on their controls C. Section 3.3 will show a specific LDF type.

Some LDFs can be quite large; for instance, a data dump typically contains millions of triples. Downloading such a large fragment can be undesired in certain situations, for instance, if we just want to inspect part of the data in the fragment, or if we are only interested in a fragment's metadata but not its actual data. Therefore, a server that hosts LDFs can segment them into smaller pages. Formally, we capture such a page as follows:

Definition 5 (LDF page). *Let $f = \langle u, s, \Gamma, M, C \rangle$ be an LDF of some finite set of blank-node-free RDF triples. A* page partitioning *of f is a finite, nonempty set Φ consisting of so-called* pages *of f such that the following properties hold:*

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 with $u' \neq u$, (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 control $c \in C'_i$ with $u'_j \in \mathrm{img}(c)$.*

Note in particular that each page contains *all* metadata and controls of the corresponding fragment, in addition to the controls that allow to navigate from one page to the next. If paging is available, servers should automatically redirect clients from the fragment to its first page, to avoid sending overly large chunks.

The collection of all LDFs of a certain dataset provided by a server is captured formally as follows:

Definition 6 (LDF collection). *Let $G \subseteq \mathcal{T}^*$ be a finite set of blank-node-free RDF triples, and let c be a control. The 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: 1. f is an LDF of G; 2. $s \in \mathrm{dom}(c)$; 3. $c(s) = u$.*

Finally, we define a query semantics for evaluating SPARQL queries over a dataset that is published as a collection of LDFs.

Definition 7 (query semantics). *Let $G \subseteq \mathcal{T}^*$ be a finite set of blank-node-free RDF triples, and let F be some LDF collection over G. The* evaluation *of a SPARQL graph pattern P over F, denoted by $[\![P]\!]_F$, is defined by $[\![P]\!]_F = [\![P]\!]_G$.*

3.3 Triple Pattern Fragments

The current HTTP interfaces for RDF, as discussed in Section 2 and summarized in Fig. 1, have limitations for query evaluation over live data with high availability. To facilitate querying on the client side, clients should be able to access those fragments that correspond to important parts of the query. To maximize availability on the server side, servers should only offer those fragments they can generate with minimal effort. In other words, we have to search for a compromise along the axis in Fig. 1. Offering triple-pattern-based access to datasets seems an interesting compromise because *a)* graph patterns, the main building blocks for SPARQL queries, consist of triple patterns, so they are important query parts for clients; and *b)* servers can select triples corresponding to a certain triple pattern at low processing cost [7]. For this reason, we introduced a triple-pattern-based HTTP interface for data access [26,27], which we formalize as follows.

Definition 8 (triple pattern fragment and collection). *Given a control c, a c-specific LDF collection F is called a* triple pattern fragment collection *if, for any possible triple pattern tp, there exists an LDF $\langle u, s, \Gamma, M, C \rangle \in F$, referred to as a* triple pattern fragment*, such that the following three properties hold:*

1. *s is the triple pattern selector for triple pattern tp (as per Definition 2).*
2. *There exists a (metadata) RDF 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 $[\![tp]\!]_G = \emptyset$, then $cnt = 0$.*
   (b) *If $[\![tp]\!]_G \neq \emptyset$, then $cnt > 0$ and $\mathrm{abs}\big(|[\![tp]\!]_G| - cnt\big) \leq \epsilon$ for some F-specific threshold ϵ.*
3. *$c \in C$.*

Since the selector s of a triple pattern fragment $f = \langle u, s, \Gamma, M, C \rangle$ is a triple pattern selector, all elements of Γ are singleton sets: $|G'| = 1$ for all $G' \in \Gamma$. Large fragments would usually be paged as in Definition 5; so while a single page would not contain all matching triples of the fragment, it would contain the *cnt* estimate metadata for the entire fragment, together with the collection's control.

Furthermore, any triple pattern fragment collection over some set of RDF triples G consists of the *complete* set of triple pattern fragments of G, which in practice means the server can provide any of them when requested, i.e., it does not need to have materialized versions for all of them. Each of these fragments includes the collection-specific hypermedia control (e.g., using the Hydra Core Vocabulary [26]), making triple pattern fragment collections hypermedia-driven REST APIs [9]. Consequently, by discovering an arbitrary fragment of a collection, a client can directly reach and retrieve *all* fragments of the collection. In particular, this includes all fragments with a selector for one of the triple patterns of a given SPARQL graph pattern. Therefore, clients can compute a complete query result for such a pattern over the collection after obtaining *any* of its fragments. In the following section, we discuss an efficient approach for performing this.

4 SPARQL Queries over Triple Pattern Fragments

4.1 High-Level Algorithm

Triple pattern fragments offer triple-pattern-based access to a dataset on the Web. If a client wants to evaluate a SPARQL query over this dataset, it should thus transform this query into a sequence of triple pattern queries. To optimize the performance of the execution, the number of HTTP requests should be minimized, and they should execute in parallel to the extent possible. Reducing the number of expensive operations is possible by selecting a suitable order in which query parts are evaluated. Therefore, database systems use a *query planner* to create an optimized order, based on statistical information about the data [10]. Since such information is usually not available for data on the Web, query planners have to resort to heuristics [13]. To mitigate this, triple pattern fragments contain metadata, i.e., the number of triples matching a certain pattern.

We previously introduced a recursive algorithm to efficiently evaluate basic graph patterns (BGPs) over a triple pattern collection [27], since BGPs form the main building blocks of SPARQL queries. We summarize the algorithm here:

1. For each triple pattern tp_i in the BGP $B = \{tp_1, \ldots, tp_n\}$, fetch the first page ϕ_1^i of the LDF f_i for tp_i, which contains an estimate cnt_i of the total number of matches for tp_i. Choose ϵ such that $cnt_\epsilon = \min(\{cnt_1, \ldots, cnt_n\})$.
2. Fetch all remaining pages of f_ϵ. For each triple t in the LDF, generate the solution mapping μ_t such that $\mu_t[tp_\epsilon] = t$. Then compose the subpattern $B_t = \{tp \mid tp = \mu_t[tp_j] \wedge tp_j \in B\} \setminus \{t\}$. If $B_t \neq \emptyset$, find mappings Ω_{B_t} by calling the algorithm for B_t. Else, $\Omega_{B_t} = \{\mu_\emptyset\}$ with $\mu_\emptyset$ the empty mapping.
3. Return all solution mappings $\mu \in \{\mu_t \cup \mu' \mid \mu' \in \Omega_{B_t}\}$.

By recursively fetching those fragments with the lowest number of matches, and applying their mappings to the graph pattern, we narrow down the number of HTTP requests that are subsequently needed.

While this algorithm finds all matches for the BGP in the collection, its recursive calling structure returns all results at once, i.e., we have to wait for the first result until all other results have been computed. Furthermore, adding support for additional SPARQL operators to such a monolithic algorithm is impractical.

4.2 Dynamic Iterator Pipelines

A common approach to implement query execution in database systems is through *iterators* that are typically arranged in a tree or a *pipeline*, based on which query results are computed recursively [10]. Such a pipelined approach has also been studied for Linked Data query processing [13,15]. In order to enable *incremental* results and allow the straightforward addition of SPARQL operators, we implement a triple pattern fragments client using iterators.

The previous algorithm, however, cannot be implemented by a *static* iterator pipeline. For instance, consider a query for architects born in European capitals:

```
SELECT ?person ?city WHERE {
  ?person a dbpedia-owl:Architect.                # tp1
  ?person dbpprop:birthPlace ?city.               # tp2
  ?city dc:subject dbpedia:Capitals_in_Europe.    # tp3
} LIMIT 100
```

Suppose the pipeline begins by finding `?city` mappings for tp_3. It then needs to choose whether it will next consider tp_1 or tp_2. The optimal choice, however, differs depending on the value of `?city`:

- For `dbpedia:Paris`, there are ±1,900 matches for tp_2, and ±1,200 matches for tp_1, so there will be less HTTP requests if we continue with tp_1.
- For `dbpedia:Vilnius`, there are 164 matches for tp_2, and ±1,200 matches for tp_1, so there will be less HTTP requests if we continue with tp_2.

With a static pipeline, we would have to choose the pipeline structure in advance and subsequently reuse it.

In order to generate an optimized pipeline for each (sub-)query, we propose a divide-and-conquer strategy in which a query is decomposed *dynamically* into subqueries depending on partial solution mappings. The main function of an iterator is `next()`, which either returns a mapping or `nil` if no mappings are left.

We first introduce a trivial *start iterator*, which outputs the empty mapping μ_0 on the first call to `next()`, and `nil` on all subsequent calls.

Next, we implement a previously defined *triple pattern iterator* [15] for triple pattern fragments. This iterator I_{tp} is initialized with a predecessor iterator I_p, a triple pattern tp, and a page ϕ_0 of an arbitrary triple pattern fragment of a collection F. The iterator then extends mappings from its predecessor by reading triples from the LDF corresponding to triple pattern tp. The URL of this LDF is retrieved through the collection control in the start page ϕ_0. Each call to I_{tp}`.next()` results in mappings for tp in F, depending on the predecessor's mappings.

To solve BGPs of SPARQL queries, we introduce a triple pattern fragment *BGP iterator*. Such a *BGP iterator* is initialized with a predecessor I_p, a BGP $B = \{tp_1, \ldots, tp_n\}$, and an arbitrary triple pattern fragment page ϕ_0 of a collection F. For an empty pattern ($n = 0$), a BGP iterator is equal to a start iterator. For a pattern length $n = 1$, it is constructed by creating a triple pattern iterator for (I_p, tp_1, ϕ_0). For $n \geq 2$, a BGP iterator uses Algorithm 1.

BGP iterators evaluate a BGP by recursively decomposing it into smaller iterators. For each triple pattern in the BGP mapped by each result of I_p, the iterator

Data: (predecessor I_p, BGP $B = \{tp_1, \ldots, tp_n\}$ with $n \geq 2$, start page ϕ_0)
$I \leftarrow$ `nil`; $c \leftarrow$ the triple pattern control in the control set C_0 of ϕ_0;
Function `BasicGraphPatternIterator.next()`
  $\mu \leftarrow$ `nil`;
  while $\mu =$ `nil` **do**
    while $I =$ `nil` **do**
      $\mu_p \leftarrow I_p$`.next()`;
      return `nil` **if** $\mu_p =$ `nil`;
      $\Phi \leftarrow \{\phi_1^i \mid \phi_1^i =$ HTTP `GET` first fragment page using URL $c(\mu_p[tp_i])\}$;
      $\epsilon \leftarrow i$ such that $cnt_{\phi_1^i} = \min(\{cnt_{\phi_1^1}, \ldots, cnt_{\phi_1^n}\})$;
      $I_\epsilon \leftarrow$ `TriplePatternIterator(StartIterator(),` $\mu_p[tp_\epsilon], \phi_1^\epsilon$`)`;
      $I \leftarrow$ `BasicGraphPatternIterator(`$I_\epsilon, \{\mu[tp] \mid tp \in B \setminus \{tp_\epsilon\}\}, \phi_1^\epsilon$`)`;
    $\mu \leftarrow I$`.next()`;
  return $\mu \cup \mu_p$;

Algorithm 1. For all mappings μ_p of a predecessor I_p, a BGP iterator for a pattern $B = \{tp_1, \ldots, tp_n\}$ creates a triple pattern iterator I_ϵ for the least frequent pattern tp_ϵ, passed to a BGP iterator for the remainder of P.

fetches the first page of the corresponding LDF. This page contains the *cnt* metadata, which tells us how many matches the dataset has for each triple pattern. The pattern is then decomposed by evaluating it using *a)* a triple pattern iterator for the triple pattern with the smallest number of matches, and *b)* a new BGP iterator for the remainder of the pattern. This results in a *dynamic* pipeline for *each* of the mappings of its predecessor, as visualized in Fig. 2. Each pipeline is optimized *locally* for a specific mapping, reducing the number of requests.

To evaluate a SPARQL query over a triple pattern fragment collection, we proceed as follows. For each BGP of the query, a BGP iterator is created. Dedicated iterators are necessary for other SPARQL constructs such as `UNION` and `OPTIONAL`, but their implementation need not be LDF-specific; they can reuse the triple pattern fragment BGP iterators. The predecessor of the first iterator is a start iterator. We continuously pull solution mappings from the last iterator in the pipeline and output them as solutions of the query, until the last iterator responds with `nil`. This pull-based process is able to deliver results incrementally.

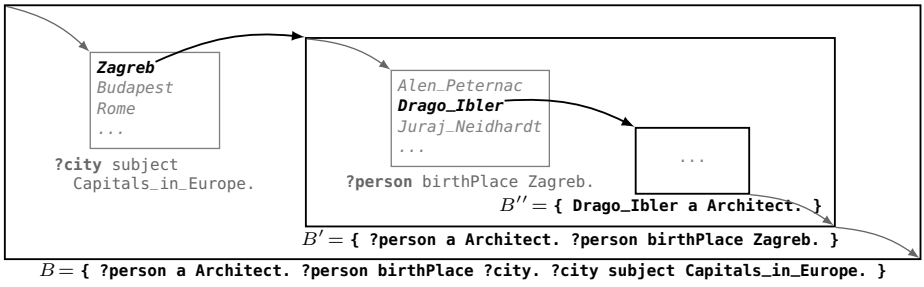


Fig. 2. A BGP iterator decomposes a BGP $B = \{tp_1, \ldots, tp_n\}$ into a triple pattern iterator for an optimal tp_i and, for each resulting solution mapping μ of tp_i, creates a BGP iterator for the remaining pattern $B' = \{tp \mid tp = \mu[tp_j] \wedge tp_j \in B\} \setminus \{\mu[tp_i]\}$

As most time of an LDF client is spent waiting on HTTP requests, the process can be sped up by buffering the individual iterators. A major advantage of our dynamic pipelines is that, because each element of a BGP iterator uses its own separate sub-pipeline, multiple pipelines can run in parallel. E.g., given the context of Fig. 2, the pipelines for `Zagreb`, `Budapest`, and `Rome` can run in parallel, and so can those for the `Alen_Peternac`, `Drago_Ibler`, and `Juraj_Neidhardt`. This results in more concurrent HTTP requests and thus a lower average waiting time per request. Since triple pattern fragment APIs are deliberately designed to allow high throughput, clients are not bound by crawler politeness rules [14].

5 Evaluation

The goal of the evaluation is to compare the availability–performance relationship of triple-pattern-based query execution to query execution over other LDFs, SPARQL endpoints in particular. *Performance* in this case refers to the query response time (i.e., time until the client reports a first solution of the query result) and total execution time. We measure *availability* of a server as the fraction of cases in which the client receives a response within a specified amount of time after sending a request. For this evaluation, we use a timeout of 60 seconds.

Since overloaded servers are a major cause of unavailability, we also monitor processor, memory, and bandwidth usage of servers. The assumption is that servers with high resource usage will be more prone to low availability, i.e., a temporal inability to process responses in a reasonable time.

5.1 Experimental Setup

We implemented the triple pattern fragments query execution approach of Section 4 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. While we also implemented an LDF client as an adapter for the popular Java framework Jena [11], it was not included in the comparison, because it uses the existing Jena ARQ querying infrastructure instead of our algorithm. The used LDF server is an open-source Java server with the compressed HDT format [7] as back-end. We provide all source code of the implementations, as well as the full benchmark configuration, at `https://github.com/LinkedDataFragments/`. The triple pattern fragments client/server setup is compared to four SPARQL endpoint infrastructures: Virtuoso (6.1.8 and 7.1.1) [5] and Jena Fuseki [11] (TDB 1.0.1 and HDT 1.1.1).

To measure the availability and performance of triple pattern fragment servers and SPARQL endpoints under varying loads, we set up an environment with one server and a variable number of clients. In order to obtain repeatable experiments, the benchmarks were executed on virtual machines 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. We purposely chose a modest server machine to show the impact for low-budget scenarios. 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). It caches HTTP requests for a maximum of 5 minutes.

To date, no SPARQL availability benchmark exists; however, several performance benchmarks exist. We chose the Berlin SPARQL Benchmark (BSBM) [3] because of its wide-spread use, with a dataset size of 100 million triples. To mimic the variability of real-world scenarios, each client executes different BSBM query workloads based on its own random seed. As existing work on Linked Data querying focuses exclusively on BGP queries [14], this paper is the first to use a SPARQL benchmark on a Linked Data publishing method with a non-SPARQL HTTP interface. We do *not* aim for best performance with triple pattern fragments; instead, we strive to improve the availability/performance balance.

For our experiments, we have extended BSBM with support for parsing streaming Turtle results, and added the possibility to measure the *response time* (reception of first solution) in addition to the *total query execution time* (reception of all solutions). Some of the BSBM queries use the `ORDER BY` operator, which has to be implemented in a blocking way; i.e., the first solution can only be sent after all solutions have been computed. Therefore, (only) for measurements of the response time, we use variants of these queries without `ORDER BY`, assuming the user application prefers streaming results and performs sorting itself.

After every 1-second interval during the evaluation, we measure on the server, cache, and client the current value of several properties, including CPU usage of each core, memory usage, and network IO. These measurements are obtained using PerfMon, while distributed testing happens using JMeter.[2]

5.2 Results and Discussion

Figs. 3.1 to 3.10 summarize the main measurements of the evaluation. All x-axes use a logarithmic scale, because we varied the number of clients exponentially.

Fig. 3.1 shows that the performance of SPARQL endpoints significantly decreases with the number of clients. Even though a triple pattern fragments setup executes SPARQL queries with lower performance, the performance decrease with a higher number of clients is significantly lower. Because of caching effects, triple pattern fragments querying starts performing slightly better with a high number of clients ($n > 100$). The per-core processor usage of the SPARQL endpoints grows rapidly (Fig. 3.5) and quickly reaches the maximum; in practice, this means the endpoint spends all CPU time processing queries while newly incoming requests are queued. The triple pattern fragments server consumes only limited CPU, because each individual request is simple to answer, and due to their finer granularity, the cache can answer several requests (Fig. 3.4).

At the client side, the opposite happens (Fig. 3.7): clients of SPARQL endpoints hardly use CPU, whereas triple pattern fragments clients do use between 20% and 100% CPU. This percentage decreases with higher numbers of clients, because the networking time dominates. Memory consumption remains fairly constant and low (Fig. 3.8). On the server (Fig. 3.6), memory usage remains constantly high; however, each considered implementation could be configured to use less memory.

[2] `http://jmeter.apache.org/` and `http://jmeter-plugins.org/wiki/PerfMonAgent/`

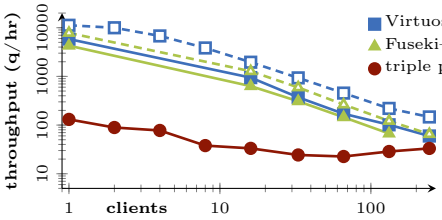


Fig. 3.1. Server performance *(log-log plot)*

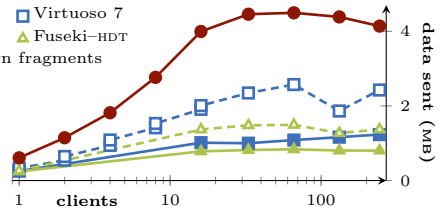


Fig. 3.2. Server network traffic

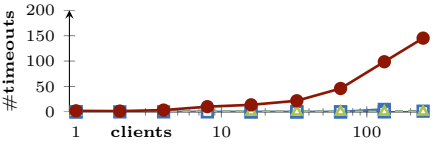


Fig. 3.3. Query timeouts

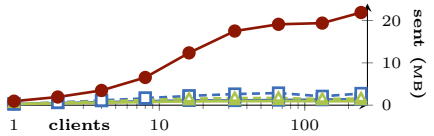


Fig. 3.4. Cache network traffic

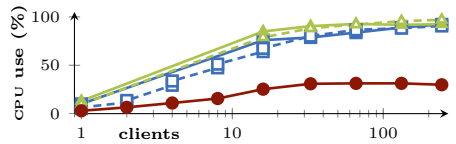


Fig. 3.5. Server processor usage per core

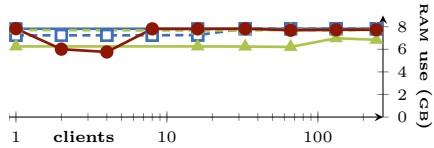


Fig. 3.6. Server memory usage

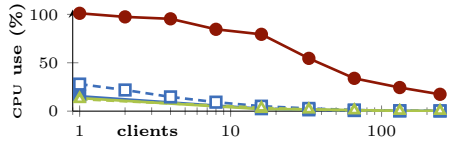


Fig. 3.7. Client processor usage per core

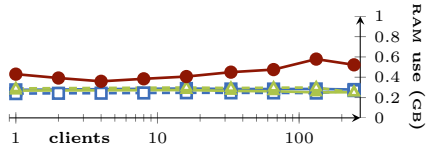


Fig. 3.8. Client memory usage

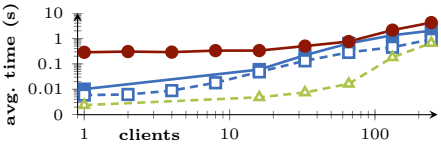


Fig. 3.9. Query 3 execution time *(log-log plot)*

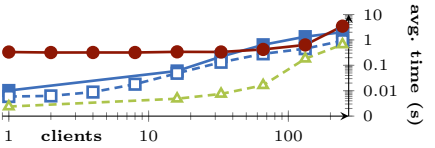


Fig. 3.10. Query 3 response time *(log-log plot)*

Fig. 3.2 shows the outbound network traffic on the server with an increasing number of clients. This traffic is substantially higher with triple pattern fragment servers, because clients need to ask for several responses to evaluate a single query. The cache ensures that responses to identical requests are reused; Fig. 3.4 indeed shows that caching is far more effective with triple pattern fragments.

Some of the BSBM queries execute slowly on triple pattern fragments clients, especially those queries that strongly depend on operators such as `FILTER`, which in a triple-pattern-based interface can only be evaluated on the client. The execution times of these queries exceed the timeout limit of 60s (Fig. 3.3).

Therefore, we separately study BSBM query template 3 (finding products that satisfy 2 numerical inequalities and an `OPTIONAL` clause), which is one of the templates whose queries cause few timeouts. Note how its execution time (Fig. 3.9) for triple pattern fragments starts high, but only increases very gradually, whereas the execution time on SPARQL endpoints rises very rapidly. Furthermore, the response times increase more slowly with increased load (Fig. 3.10). Only on the triple pattern fragments server, CPU usage remains low for this query at all times.

These results indicate that triple pattern fragments query execution succeeds in reducing server usage, at the cost of increased query times. Triple pattern fragments servers cope better with increasing numbers of clients than SPARQL endpoints. Furthermore, querying benefits strongly from regular HTTP caching, which can be added at any point in the network. 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 availability. Even though certain queries make it difficult for an LDF client to find *all* results within the timeout window (especially with blocking operators such as `ORDER BY`), the *first* results to all queries arrive before the timeout period. In the future, the development of an availability-focused SPARQL benchmark could stimulate availability improvements of the considered systems. The full results of our experiments are published as LDFs at `http://data.linkeddatafragments.org/`.

6 Conclusions

Publishers of Linked Data strive to host their data reliably at minimal cost. Applications, on the other hand, need to query data in the most flexible way. The three well-known RDF interfaces on the Web—SPARQL endpoints, Linked Data documents, and data dumps—are just a fraction of all possible ways to transfer Linked Data from a server to a client. Since SPARQL endpoints offer the most flexibility, they are not coincidentally the most expensive to host with high availability. The *Linked Data Fragments* framework captures the search for alternative HTTP interfaces to RDF data, trying to balance the server's desire for maximum reliability and the client's need for maximum flexibility.

In this paper, we have shown that triple pattern fragments, which additionally contain count metadata and hypermedia controls, can reduce the load on servers to less than 30% of the load on SPARQL endpoints. This happens by shifting the query-specific tasks to clients, at the cost of slower query execution. Instead of sending one complex query, clients use a dynamic iterator pipeline to combine the results of several simpler queries, thereby also vastly improving the effectiveness of HTTP caching. This captures the spirit of *Web* querying: clients browse pages and iteratively extract bits of information to find complex answers. The goal of triple pattern fragments is to provide those bits that are helpful for clients to evaluate queries, yet inexpensive to generate by servers.

Triple pattern fragments are definitely not the final answer to querying RDF datasets on the Web. In fact, there will probably never be such a final answer. By definition, each API on the Web that publishes RDF triples (which, through

JSON-LD [24], can include those APIs that publish JSON) offers its own kind of LDFs. The challenge for future clients is to find answers to queries through all kinds of *different* fragments along the axis of LDFs. The results indicate the potential of this querying strategy, as we have shown they allow executing complex queries of common SPARQL benchmarks over live data on the Web with high availability. Whereas the Linked Data principles emphasize *hyperlinks* between data documents [2], triple pattern fragments add *forms* that let clients control what data they request. Those forms allow custom access, but at the same time limit the possible kind of queries in order to save on server processing resources.

Especially in cases where there are limited financial resources to publish data, triple pattern fragments could make a significant difference: data can be hosted at low cost, in a way that allows live querying, with high availability. In addition to our own open source implementations of LDF servers, two third-party implementations are available. The Belgian Crossroads Bank for Enterprises recently published their data as triple pattern fragments (`http://data.kbodata.be/`) using their own open-source server. The open-source data management system The DataTank (`http://thedatatank.com/`) now also supports triple pattern fragments. This lowers the entry barrier for publishers even further. Implementers of clients and servers can follow the triple pattern fragments specification [26].

Improving the performance is possible if clients can query more specific fragments. In particular, support for certain `FILTER` expressions would speed up several queries, as triple pattern fragments only allow for exact matches. Interesting future work is therefore to define new classes of LDFs that support such features, where we always need to keep in mind that minimizing the server's processing cost for each fragment is the key to maximizing its availability. Part of this work includes the *description* of such fragments, so a client can dynamically discover what fragment types a server offers, and thus how it can execute a query in an optimal way. For instance, if a server supports (a subset of) SPARQL, clients need to ask fewer queries than when only triple patterns are supported. This trade-off between server cost/availability and client performance will continue to exist.

Finally, LDF querying shows us that we perhaps need to re-evaluate the way we develop applications on top of Linked Data. The dominant paradigm so far has been: *"ask a complex question to a server; wait; act on the results"*, where the "waiting" part can be long if the server has low availability. As the response times of the evaluation indicate, new applications might prefer not to wait, evolving towards a real-time, distributed paradigm: *"ask simple questions to many servers, acting on results as they arrive"*. A major benefit of clients that solve queries by fetching fragments of Linked Data, in addition to incrementally updating results, is that they can support distributed querying by asking fragments from different servers. In other words, limiting the HTTP interfaces of RDF servers does not only lead to higher availability, it encourages clients to solve complex queries themselves—and for that, they have the entire Web of Data at their disposal.

References

1. Amundsen, M.: Hypermedia types. In: REST: From Research to Practice, pp. 93–116. Springer (2011)
2. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data – the story so far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
3. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. International Journal on Semantic Web and Information Systems 5(2), 1–24 (2009)
4. Buil-Aranda, C., Hogan, A.: SPARQL Web-querying infrastructure: Ready for action? In: Proceedings of the 12th International Semantic Web Conference (November 2013)
5. Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: Semantic Web Information Management, pp. 501–519. Springer (2010)
6. Feigenbaum, L., Williams, G.T., Clark, K.G., Torres, E.: SPARQL 1.1 protocol Recommendation, W3C (March 2013), http://www.w3.org/TR/sparql11-protocol/
7. Fernández, J.D., Martínez-Prieto, M.A., Gutirrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Journal of Web Semantics 19, 22–41 (2013)
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California (2000)
9. Fielding, R.T.: REST APIs must be hypertext-driven (October 2008), http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
10. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surveys 25(2), 73–169 (June 1993)
11. Grobe, M.: RDF, Jena, SPARQL and the Semantic Web. In: Proceedings of the 37th Annual ACM SIGUCCS Fall Conference: Communication and Collaboration (2009), doi:10.1145/1629501.1629525
12. Harris, S., Seaborne, A.: SPARQL 1.1 query language. Recommendation, W3C (March 2013), http://www.w3.org/TR/sparql11-query/
13. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 154–169. Springer, Heidelberg (2011)
14. Hartig, O.: An overview on execution strategies for Linked Data queries. Datenbank-Spektrum 13(2), 89–99 (2013)
15. Hartig, O., Bizer, C., Freytag, J.-C.: Executing SPARQL Queries over the Web of Linked Data. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 293–309. Springer, Heidelberg (2009)
16. Klyne, G., Carrol, J.J.: Resource Description Framework (RDF): Concepts and abstract syntax. Rec., W3C (February 2004), http://www.w3.org/TR/rdf-concepts/
17. Linked Data API, https://code.google.com/p/linked-data-api/
18. Matteis, L.: Restpark: Minimal RESTful API for retrieving RDF triples (2013), http://lmatteis.github.io/restpark/restpark.pdf
19. Morsey, M., Lehmann, J.: DBpedia SPARQL benchmark performance assessment with real queries on real data. In: Proceedings of the 9th International Semantic Web Conference (2011)
20. Ogbuji, C.: SPARQL 1.1 Graph Store HTTP Protocol. Recommendation, W3C (March 2013), http://www.w3.org/TR/sparql11-http-rdf-update/

21. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems 34(3), 16:1–16:45 (2009)
22. Schmidt, M., Hornung, T., Meier, M., Pinkel, C., Lausen, G.: SP^{2}Bench: A SPARQL performance benchmark. In: Semantic Web Information Management (2010)
23. Speicher, S., Arwe, J., Malhotra, A.: Linked Data Platform 1.0. Candidate recommendation, W3C (June 2014), http://www.w3.org/TR/2014/CR-ldp-20140619/
24. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: JSONLD 1.0. Recommendation, W3C (January 2014), http://www.w3.org/TR/json-ld/
25. Verborgh, R.: Linked Data Fragments. Unofficial draft, Hydra WTHREEC Community Group, http://www.hydra-cg.com/spec/latest/linked-data-fragments/
26. Verborgh, R.: Triple Pattern Fragments. Unofficial draft, Hydra W3C Community Group, http://www.hydra-cg.com/spec/latest/triple-pattern-fragments/
27. Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., Van de Walle, R.: Web-scale querying through Linked Data Fragments. In: Proceedings of the 7th Workshop on Linked Data on the Web (2014)
28. Wilde, E., Hausenblas, M.: RESTful SPARQL? You name it! – Aligning SPARQL with REST and resource orientation. In: Proceedings of the 4 Workshop on Emerging Web Services Technology, pp. 39–43. ACM (2009)