Web-Scale Querying through Linked Data Fragments

Ruben Verborgh Miel Vander Sande Pieter Colpaert
Sam Coppens Erik Mannens Rik Van de Walle

Multimedia Lab – Ghent University – iMinds
Gaston Crommenlaan 8 bus 201
B-9050 Ledeberg-Ghent, Belgium
{firstname.lastname}@ugent.be

ABSTRACT

To unlock the full potential of Linked Data sources, we need flexible ways to query them. Public SPARQL endpoints aim to fulfill that need, but their availability is notoriously problematic. We therefore introduce Linked Data Fragments, a publishing method that allows efficient offloading of query execution from servers to clients through a lightweight partitioning strategy. It enables servers to maintain availability rates as high as any regular HTTP server, allowing querying to scale reliably to much larger numbers of clients. This paper explains the core concepts behind Linked Data Fragments and experimentally verifies their Web-level scalability, at the cost of increased query times. We show how trading server-side query execution for inexpensive data resources with relevant affordances enables a new generation of intelligent clients.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

Keywords

Linked Data, querying, availability, scalability, SPARQL

1. INTRODUCTION

Whenever there is a large amount of data, people will want to query it—and nothing is more intriguing to query than the vast amounts of Linked Data published over the last few years [2]. With over 800 million triples in the widely used DBpedia [3], only one of the many datasets in a large ecosystem, the need for various specialized information searches has never been this high before. SPARQL has been specifically designed [30] to fulfill this requirement for reliable and standardized access to data in the RDF triple format. Consisting of a query language [15] and a protocol [9], SPARQL is the *de facto* choice to publish RDF data in a flexible way, and allows to select with high precision the data that interests us.

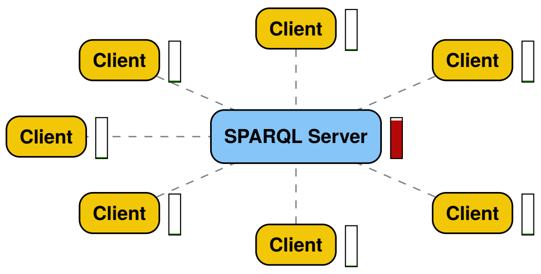
There is one issue: it appears to be very hard to make a SPARQL endpoint available reliably. A recent survey examining 427 public endpoints concluded that only one third of them have an availability rate above 99%; not even half of all endpoints reach 95% [6]. To put this into perspective: 95% availability means the server is unavailable for one and a half days every month. These figures are quite disturbing given the fact that availability is usually measured in "number of nines" [5, 25], counting the number of leading nines in the availability percentage. In comparison, the fairly common three nines (99.9%) amounts to 8.8 hours of downtime *per year*. The disappointingly low availability of public SPARQL endpoints is the Semantic Web community's very own *"Inconvenient Truth"*.

More precisely, practice reveals that the following three characteristics are irreconcilable for SPARQL endpoints:

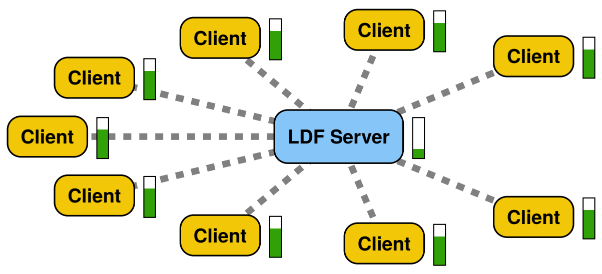
a) being publicly available;
b) offering unrestricted queries;
c) having many concurrent users.

This is because the load of a server is proportional to the product of the variety, complexity, and amount of requests, the first two of which remain virtually unbounded for SPARQL. Any endpoint's availability can be considerably improved by sacrificing one of these three characteristics: private SPARQL endpoints perform well because the server load can be predicted more reliably, limiting query possibilities eliminates slow queries that can bring down the server, and low demand of course contributes positively to availability. HTTP servers on the other hand have no problem combining these characteristics, as the complexity of each request can be limited because the server restricts what "queries" a client can execute by determining the offered HTTP resources [13]. We do not claim by any means this comparison is fair, as SPARQL servers have to perform significantly more work per request. On the contrary, it is exactly *because* SPARQL requests require more processing that SPARQL endpoints do not scale well compared to HTTP servers.

This paper challenges the idea that servers should spend their CPU cycles on expensive queries, and proposes a model in which the client solves a complex query by only asking the server for simple data retrieval operations. Instead of answering a complex SPARQL query, the server sends a Linked Data Fragment that corresponds to a specific triple pattern. This fragment then contains metadata that allows the client itself to execute the complex query. While this leads to an increased number of HTTP requests between clients and servers, each request is answered easily and also fully cacheable. Therefore, this is the scalable and sustainable approach to Web querying: with SPARQL, each new client requires additional processing power from the server, whereas with Linked Data Fragments, clients take care of their own processing. We effectively trade fast answers but low scalability for increased (yet manageable) query times with Web-level scalability. Most importantly, this makes it possible to fully query datasets of publishers who cannot invest in hosting and maintaining an expensive SPARQL endpoint—which is most of us.



(a) SPARQL endpoints perform all processing on the server, leading to fast query execution with low data bandwidth, and a rapidly overloaded server.



(b) LDF servers only support simple requests and can thus handle far higher loads. Clients perform the querying, so they need more (cacheable) data.

Figure 1: A comparison of required processing *(filled bars)* and data transfer *(dotted lines)* shows why LDF scales significantly better.

In the next section, we critically examine the scalability problems of SPARQL. Next, we introduce Linked Data Fragments, followed by the implementation of a server (Section 4) and a client (Section 5). Section 6 evaluates the improved scalability. We then discuss our method and its context in Section 7, and conclude in Section 8.

2. RELATED WORK

At its core, the SPARQL query language [15] allows clients to find triples based on basic graph patterns. For instance, consider the following SPARQL query:

```
SELECT ?p ?c WHERE {
  ?p a <http://dbpedia.org/ontology/Artist>.
  ?p <http://dbpedia.org/ontology/birthPlace> ?c.
  ?c <http://xmlns.com/foaf/0.1/name> "York"@en.
}
```

Listing 1: Search for artists born in places named "York".

Such queries facilitate searching relevant information in datasets that can contain hundreds of millions of triples. Most triple stores, such as Virtuoso and Sesame, offer a SPARQL interface, which is referred to as a "SPARQL endpoint" when exposed through HTTP.

Before we dive into the details of SPARQL endpoints, let us first briefly recapitulate the architectural properties of the Web and why they enable the Web to scale the way it does. The Web is a distributed hypermedia application that conforms to the constraints of the Representational State Transfer architectural style (REST, [11]). The main building blocks of the Web are *resources*, which are identified by URLs through the *uniform interface* offered by HTTP [13]. Resources can be represented in *hypermedia formats*, which can link to other resources. These links remove the need for the server to maintain the application state between different interactions, as each representation (and not the server) retains the next steps a client can take [12]. The combination of the uniform interface and statelessness makes it possible for intermediaries to *cache* server responses, which significantly improves scalability.

SPARQL endpoints essentially implement a protocol on top of HTTP through a strictly standardized set of constraints [9]. The client sends a SPARQL query to a server, which executes it and sends back the results. Given the amount of data involved and the arbitrary complexity of queries, the server possibly needs to execute a significant amount of work to obtain the results of each query. In contrast to regular HTTP servers, a SPARQL endpoint does not expose resources on an application-specific level, but rather one "endpoint" resource that acts as a data handling process [13], and an unlimited set of "query answer" resources that correspond to all queries [9, 11]. Therefore, regular HTTP caching strategies for resources *below* query level cannot be applied; each unique query still needs full execution. This query-based partitioning of resources gives SPARQL poor scaling properties, as illustrated in Figure 1a. The inherent problem with such an endpoint architecture is that the required time to generate each query answer resource is potentially very high, and that all processing needs to happen at the server side. While this makes querying rather convenient for clients, it puts an enormous burden on providers of SPARQL endpoints, as SPARQL engines can strain CPU and RAM intensively even for common queries [4]. It should thus not surprise us that maintaining high availability rates for public SPARQL endpoints is exceptionally challenging [6]. This problem seldom occurs with regular HTTP servers, as the granularity of offered resources can be adjusted such that each individual resource does not require excessive processing time. Additionally, this finer granularity allows those resources to be cached efficiently [11].

The performance of SPARQL has been the subject of multiple benchmarks [4, 29]. Several caching strategies have been proposed on various levels, for instance, by placing a proxy in front of a SPARQL endpoint [24], or by integrating caching information into the triple store itself to allow HTTP caching [35]. However, these techniques consider caching of results for entire queries, which means related but non-identical queries do not benefit. Syntax-agnostic approaches can cache based on the algebraic representation and allow subquery caching [36], also enabled by other specific techniques [22, 23, 31].

A category of approaches for executing SPARQL queries over Linked Data [18] is based on link traversal [19] and relies on the principle of dereferencing [2]. Link traversal strongly benefits from caching [16] because the granularity is refined to the level of data needed for queries—as opposed to the full result set of a single query. This technique resembles the querying method we will introduce in this paper, because of the active role clients play in fetching and evaluating data, as well as the potential of pipelining through non-blocking iterators [19]. However, our method does not rely on one primary data source per URI (a consequence of dereferencing) and we use additional information to reduce the execution time of typical queries by more than an order of magnitude. While optimizing planning heuristics exist [17], our planning strategy employs more reliable indicators. Furthermore, the present initial paper focuses on vastly improving the scalability of *individual* endpoints, even though the method is generalizable to distributed querying.

Closely tied to the publication of Linked Data is the specification of a standard read/write interface, which is the goal of the Linked Data Platform (LDP, [32]). While the definitions in Section 3 will seemingly demand a comparison with LDP, it is crucial to note that LDP and Linked Data Fragments are orthogonal, i.e., a server can choose to support either or both of them independently. More specifically, LDP proposes a subject-centric read/write interface, while the goal of Linked Data Fragments is to offer scalable query execution. Our design permits any resource to additionally implement LDP.

3. LINKED DATA FRAGMENTS

3.1 Motivation

As indicated above, the concept of querying through endpoints entails serious availability issues [6], the root cause of which is the non-scalability of the expensive component, the SPARQL server (Figure 1a). While all client–server interactions on the Web can lead to server overloading, SPARQL is especially vulnerable to this because of its partitioning in (potentially expensive) query answer resources. For instance, compare DBpedia access through its SPARQL endpoint[1] versus its subject pages[2]. The former provides access to the unlimited set of query answers, whereas the latter provides the *same* data through a limited set of subject resources listing all triples per subject. It it straightforward to understand that, regardless of the used technology, the latter demands less server usage because the underlying queries are answered easily by simple index lookups. In fact, the second case does not even require an on-demand query processor: because the subject set is finite, a static file server could serve pre-generated subject pages, which are updated periodically by another process. Furthermore, such a finite set can be cached efficiently by regular HTTP caches as several clients reuse the same pages, whereas a large amount of SPARQL queries are client-specific.

Admittedly, even though the same data is exposed in both cases, the SPARQL endpoint is more powerful when available, because it offers custom client-centric views on specific parts of the data. In contrast, the server-driven partitioning in subjects might or might not be helpful for a specific client's goal. Yet this is exactly the reason why hosting a SPARQL endpoint is such a risky endeavor: the scalability of HTTP and thus the whole Web are based on effective partitioning of resources. It is only natural that a server goes down if it commits itself to serving an unlimited set of expensive resources.

3.2 Definitions and examples

To solve these availability and scalability issues, we need to create a compromise between offering a very limited partitioning and allowing unrestricted SPARQL queries. Each of the offered resources should additionally contain the necessary information for clients to execute SPARQL queries efficiently themselves. To that end, we introduce *Linked Data Fragments*, offering a hybrid solution between limited subject-based Linked Data dereferencing and the difficultly scalable server-side SPARQL execution.

Definition 1. *A Linked Data Fragment (LDF) of a Linked Data dataset is a resource consisting of those elements of this dataset that match a specific selector, together with their metadata and the controls to retrieve related Linked Data Fragments.*

We will first discuss the selector aspect, before we detail the metadata and control constraints. The concept is not unlike that of a *media fragment* [33], which selects a part of a media resource. We instead select parts of Linked Data resources, without *a priori* restricting the kind of selector. Therefore, the data of an LDF could for instance be that of a subject page (which would have the triple pattern selector `{ <s> ?p ?o }` for a specific `<s>`) or even a SPARQL result resource (which would have the SPARQL query as a selector). However, we are primarily interested in those LDFs that *a*) are useful for client-side query answering *b*) only require a low server processing cost. Therefore, we define the following:

Definition 2. *A basic Linked Data Fragment (basic LDF) is a Linked Data Fragment with a triple pattern as selector, count metadata, and the controls to retrieve any other basic LDF of the same dataset, in particular other fragments the matching elements belong to.*

[1] *http://dbpedia.org/sparql*

[2] e.g., *http://dbpedia.org/page/Pete_Townshend* (via Pubby [7])

Linked Data subject pages offer only a subset of all basic LDFs, namely those triple patterns with a fixed subject and variable predicates and objects. However, to avoid exhaustive searches when solving queries with variable predicates or objects, a partitioning into basic LDFs includes all combinations of `{ ?s ?p ?o }` with each component either a variable or a specific URI or literal. For instance, the basic LDFs for DBpedia include "triples with *Pete Townshend* as subject", "triples with *The Who* as object", as well as "triples with *Pete Townshend* as subject and *birth place* as predicate". In other words, as each component can either be variable or fixed, each triple in a dataset belongs to exactly $2^3 = 8$ basic LDFs.

This data partitioning is only one of the aspects that sets LDFs apart from alternatives. Definition 1 also mentions *metadata* and *controls*, which are defined in the same open way as the possible selectors. Together, they transform the LDF into an *affordance* [12] that enables the client to perform actions, in particular SPARQL query execution. Each LDF thereby provides the client with context on how this fragment relates to the dataset and other fragments. The metadata on the one hand includes information such as the fragment's selector, since the client might have received this fragment from a third party, unaware of the precise selector used. Controls on the other hand include links to other fragments, allowing clients to discover more data. Providing affordance is a required part of any REST interface [12], as it allows statelessness and reduces client–server coupling [27].

Which metadata and controls we need is constrained by Definition 2. Since LDFs can be very long (for instance, DBpedia counts more than 60 million matches for `{ ?s rdf:type ?o }`), they sometimes need *pagination* [26]. To compensate, each basic LDF should provide the (estimated) total number of triples that match the pattern. As we will see in Section 5, this is crucial for efficient querying.

Furthermore, each basic LDF should provide the (hypermedia) controls to access other basic LDFs. A concrete implementation could be that the basic LDF for the pattern `{ ?s rdf:type ?o }` links to the LDF for `{ ?s ?p foaf:Person }` (if this fragment contains `foaf:Person` triples). Additionally, each basic LDF representation must contain a form or similar control that allows to retrieve any basic LDF with a triple pattern selector of choice. This is necessary for the independent evolution of LDF clients and servers: a client should not need to know how a server exposes its LDF resources. Concretely, servers are free to choose the URLs of the LDF resources they offer. This makes LDF compatible with a partitioning that does require a specific URL structure such as the SPARQL protocol, which demands a "`?query=`" parameter [9].

The next section will discuss the design and implementation of LDF servers, followed by a section on the design of LDF clients that can execute SPARQL queries through LDFs.

4. SERVER

4.1 Architecture

The desired architectural characteristics for a server of LDFs are *availability*, *scalability*, and *performance*, in that order. This means that, at any time, its top priority is to ensure clients can reach the server and retrieve a response within a time interval that is similar to other HTTP servers. As typical HTTP response time is of the order of a few hundreds of milliseconds, this is what we aim for (and preferably less). In addition, the infrastructure should scale with the number of clients. We define an LDF server as follows.

Definition 3. *A Linked Data Fragments server (LDF server) is an HTTP server that offers Linked Data Fragments covering one or more datasets in at least one triple-based representation.*

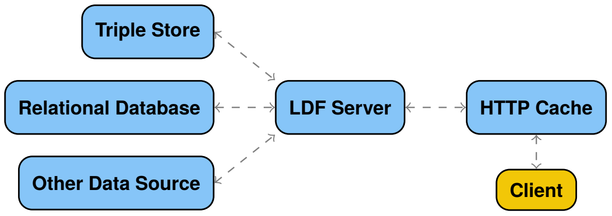


Figure 2: An LDF server sends out simple queries to underlying data sources, which result in fast and cacheable answers.

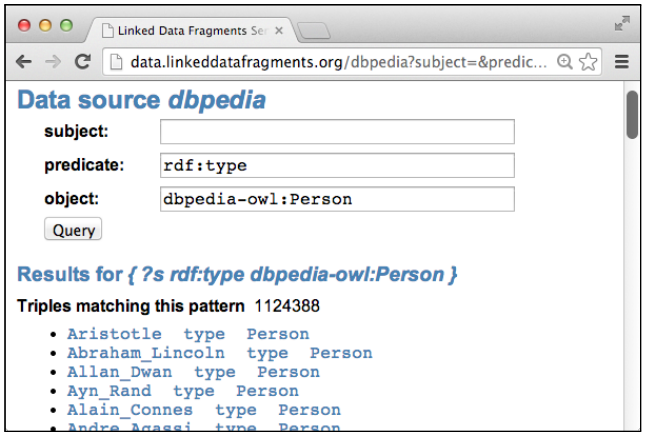
Note that under this definition, SPARQL endpoints, subject page servers, and HTTP servers with data dumps are also LDF servers. While servers choose which specific LDF partitionings they use, those servers that offer *basic* LDFs strike an optimal balance between low server-side complexity and efficient client-side querying (Section 5); we call these *basic LDF servers*. In addition to triple-based representations (such as Turtle or HTML with RDFa), servers can offer others (such as text or regular HTML) and/or more LDFs with more complex selectors. However, these features are optional, as complex selectors might negatively impact server availability and performance.

Figure 2 shows a schematic display of how an LDF server interacts with its environment. On the back-end side, the server fetches data from an underlying data source to construct LDF fragments. Such a data source could be a triple store, perhaps through a SPARQL endpoint, but even a relational database or an RDF source file. The fact that we would still use SPARQL endpoints after criticizing them for low availability seems a contradiction, but it is not: as stated in Section 1, when the complexity of queries can be limited, endpoints can perform very well. Since LDF servers are only required to obtain results for single triple patterns, the endpoint is not stressed in any way. Alternatively, regular relational databases can also perform well because of the simplicity of the lookup patterns.

The main task of the LDF server is offering a REST interface [11] to its LDFs by providing and maintaining a URL space. It translates each request into a specific query for the appropriate data server that collects the needed data and metadata. This is then combined into an LDF and represented in a media type the client understands (such as Turtle or HTML) by adding the data as well as the metadata and controls that provide the affordance towards next steps [12].

On the front-end side, the LDF server can be proxied through a regular HTTP cache [13], restricting the load on the LDF server. Furthermore, the LDF server can restrict load on the underlying data sources by caching responses as well, for instance, using existing SPARQL HTTP caching mechanisms [31, 35].

This architecture maximizes availability and performance by two key decisions. First, the offered resources consist of LDFs that are simple to generate, minimizing processing time for each resource. In contrast to endpoints offering an unlimited SPARQL interface, this places an upper bound on the execution time of each request; and lower server loads directly lead to higher availability. Second, when partitioning in basic LDFs, the entire dataset is exposed in a way that maximizes reuse across clients, and hence enables efficient caching. Furthermore, since the set of basic LDFs of a dataset is finite, substantial parts can be pre-generated and pre-cached, leading to lower server load and faster response times. The scalability is then guaranteed through the properties induced by the REST architectural style [11]. Caching at the front-end can happen hierarchically, and load-balancing between multiple LDF servers is possible. Back-end caching and load-balancing can happen as well, but synchronization might be required to ensure consistency between different servers. However, as the load caused by the front-end server is predictable, a sufficient infrastructure for synchronization can be planned.



Data source *dbpedia*

subject:

predicate: rdf:type

object: dbpedia-owl:Person

Results for *{ ?s rdf:type dbpedia-owl:Person }*

Triples matching this pattern 1124388

- Aristotle type Person
- Abraham_Lincoln type Person
- Allan_Dwan type Person
- Ayn_Rand type Person
- Alain_Connes type Person

Figure 3: Each basic LDF has a triple pattern selector, count metadata, and controls towards any other basic LDFs.

4.2 Example implementation

We have implemented an example LDF server, the source code of which is available at *http://linkeddatafragments.org/software/*. A public instance of this server with several datasets is running at *http://data.linkeddatafragments.org/*. We will demonstrate the discussed features of LDFs through this public instance. Note that every step in the following discovery process happens entirely through the affordance supplied by the LDFs, i.e., by using links and forms, indicating the client's decoupling from any server's URL structure.

When you open *http://data.linkeddatafragments.org/* in a browser, you will see links to different datasets. This start resource is in fact an LDF that allows to browse all datasets on the server. One of them is DBpedia, which is located at */dbpedia*. This initial DBpedia LDF lists some triples of the dataset to allow browsing. Using the provided links, we can click through to see related fragments. For example, when we click an `rdf:type` link of a triple, we arrive at the basic LDF of all triples with the `rdf:type` predicate, located at */dbpedia?predicate=rdf%3Atype*. We can also use the form to navigate to a specific basic LDF. For example, Figure 3 shows the LDF for the pattern `{ ?s rdf:type dbpedia-owl:Person }`, located at */dbpedia?predicate=rdf%3Atype&object=dbpedia-owl%3APerson*. While both URLs follow a convention adopted by this particular server, they remain opaque identifiers that servers can assign freely to LDF resources as long as they provide the necessary controls.

The HTML representation of LDFs generated by this server contains RDFa markup to enable interpretation by automated clients. All triples and metadata are annotated. However, parsing HTML involves an overhead that can be avoided by directly parsing Turtle. For that reason, our implementation also offers Turtle representations of each LDF through HTTP content negotiation [13]. For example:

```
curl http://data.linkeddatafragments.org/dbpedia \
     -H "Accept: text/turtle"
```

This results in a Turtle representation of the LDF we retrieved earlier. In contrast to HTML, which has `<a>` and `<form>` elements, RDF offers no native support for hypermedia controls [20]—except for the URIs of its triple components, which only allow dereferencing (cf. DBpedia subject pages). Since basic LDFs must contain controls towards all other basic LDFs of the dataset, we have to describe them declaratively. This happens in three ways. First, for each of the non-variable parts of the basic LDF's triple pattern, `rdfs:seeAlso` links are provided to the LDFs that have these parts in subject or object position. Second, the representation provides an alternative to the HTML form using the Hydra hypermedia API vocabulary [21], allowing the client to query any basic LDF of the dataset. Finally, it offers a dataset description using the VOID vocabulary [8], which defines properties such as triple count. These annotations give the Turtle representation the same affordance as its HTML counterpart.

4.3 Dereferencing

At first sight, it might appear that LDF voids the Linked Data principles that enable dereferencing [2]. After all, the fact that the identifier of a concept (URI) also serves as its address (URL) forms the foundation of Linked Data. For example, not only does *http://dbpedia.org/resource/Pete_Townshend* uniquely identify the musician *Pete Townshend*, it also affords retrieving information about him. In contrast, this information on the previous LDF server is located at */dbpedia?subject=dbpedia%3APete_Townshend* and */dbpedia?object=dbpedia%3APete_Townshend*. Yet, dereferencing and LDFs actually play complementary roles, as indicated below.

First, the use of LDFs does not break dereferencing. Since LDF servers are not bound by URL constraints, they can choose to serve the LDF about the resource at its own URL. In fact, *dbpedia.org is* an LDF server: it could host the LDF with Pete Townshend as subject at *http://dbpedia.org/resource/Pete_Townshend*, and could in principle also offer support for basic LDFs. This shows that dereferencing and LDF can work in conjunction seamlessly.

Second, the Web is founded on the idea that "anyone can say anything about anything". While dereferencing is fast and easy, it only leads to the source that happens to host the identifier, which does not mean this source also has the information we are looking for. Compare this to regular Web browsing: the best source for objective information about a certain company is likely not that company's homepage. It would be unpractical to assign a new identifier every time another party wants to add statements about a resource. Furthermore, no single representation can contain all facts; for instance, *http://dbpedia.org/ontology/Person* does not contain a list of all people on the Web. A basic LDF server instead lets us indicate what triples we want to obtain about a certain resource, differentiating between subject, predicate, and object positions. While a basic LDF server would also not represent millions of people on a single page, it allows to retrieve a list of them page by page through the paginated basic LDF resource for the pattern `{ ?s a dbpedia-owl:Person }`. That way, we can ask to obtain all `dbpedia-owl:Person` instances from any dataset, even when not hosted on the DBpedia URL space. Additionally, dereferencing only works with URLs, whereas LDF affordances also function with generic URIs.

Third, the fourth Linked Data principle demands to include links to other resources [2]. This means representations of resources such as *http://dbpedia.org/resource/Pete_Townshend* could link to LDFs that contain more data. This closes the circle, as LDFs themselves link to *a)* the concept's URL (through the data) and *b)* related LDFs (through the metadata). The main difference between dereferencing and LDFs is that the former uses the implicit affordance in the URL, whereas LDFs explicitly provide multiple affordances.

5. CLIENT

5.1 Querying basic Linked Data Fragments

The scalability of LDFs as depicted in Figure 1b can be achieved when the server offers a partitioning that is inexpensive to generate but still allows efficient client-side querying. When using basic LDFs, leading to a partitioning in basic triple patterns, clients can solve queries for basic graph patterns autonomously and efficiently. Each resource operation requires only minimal cost from the server, is fully cacheable, and likely to be reused.

```
1  Function FindVariableBindings($Q$, $F$)
       Input: basic graph pattern query $Q$ and start fragment $F$
       Output: possible variable bindings (nil if none needed)
2      possible variable bindings $B \leftarrow \{\}$;
3      split pattern $Q$ in connected subpatterns $S = \{S_1, \ldots, S_n\}$;
4      foreach subpattern $S_i \in S$ do
5          foreach triple pattern $t_j$ in subpattern $S_i$ do
6              $F_j \leftarrow$ GET first page of basic LDF for $t_j$ through $F$;
7          end
8          return $\varnothing$ if any fragment $F_j$ has 0 matching triples;
9          if all fragments $F_j$ have exactly 1 matching triple then
10             $B[S_i][b_j] \leftarrow$ nil $\forall t_j$ where $b_j :=$ binding of $t_j$ to $F_j$;
11             return $\varnothing$ if $\bigcup b_j$ is inconsistent;
12         else
13             $F_m := F_j$ with minimal total number of matches;
14             $F'_m \leftarrow F_m \cup \{$ GET remaining pages of $F_m$ $\}$;
15             foreach binding $b_k$ of pattern $t_m$ in fragment $F'_m$ do
16                 $S'_i \leftarrow$ apply binding $b_k$ to subpattern $S_i$;
17                 $B[S_i][b_k] \leftarrow$ FindVariableBindings($S'_i$, $F'_m$);
18             end
19         end
20     end
21     return $B' \leftarrow B$ where $B[s_i][b_k] \neq \varnothing$;
22 end
```

Algorithm 1: An LDF client can efficiently find the possible variable bindings of any basic graph pattern through basic LDFs.

The main task of an LDF client is to find possible variable bindings of queries such as the one in Listing 1. Algorithm 1 details this process for basic LDFs. Count metadata is used to ensure an efficient solution path (line 13). `FindVariableBindings` has three possible types of output for queries and their subqueries:

the empty set $\varnothing$ if no valid binding exists. For instance, given a query `{ ?person foaf:name "Fake Name"@en }`, no value for `?person` exists, so the set of possible bindings is empty.

nil if no binding is necessary to satisfy the query. For instance, `{ dbpedia:Keith_Moon foaf:name "Keith Moon"@en }` is satisfied if (and only if) the corresponding triple exists.

a hierarchical list of bindings in all other cases. For instance, a solution to the query in Listing 1 could have bindings for `?c`, each of which can have one or multiple bindings for `?p`.

We will now run through a possible execution of the algorithm for the query in Listing 1, assuming a basic LDF server with the DBpedia dataset. We invoke `FindVariableBindings` with $Q \leftarrow$ Listing 1 and $F \leftarrow$ the basic LDF at *http://data.linkeddatafragments.org/dbpedia*. As the entire query pattern is connected (i.e., there exists a path from any triple to any other by following shared variables), there is only one subpattern: $S = \{S_1\} = \{Q\}$. There are three triple patterns in S_1, so we use the controls in F to `GET` the corresponding basic LDFs. For each of them, we receive the first 100 matches and metadata:

	fragment selector	matches
F_1	`{ ?p a dbpedia-owl:Artist }`	68,237
F_2	`{ ?p dbpedia-owl:birthPlace ?c }`	469,849
F_3	`{ ?c foaf:name "York"@en }`	12

As each fragment has more than 1 match, lines 8 to 11 of the algorithm are skipped and we go straight to line 13 where F_3 is selected as smallest fragment F_m. Since the 12 triples fit on one result page, $F'_m = F_m$. The possible bindings of `?c` in F'_m include `dbpedia:York_(album)`, `dbpedia:York`, `dbpedia:York,_Ontario`, `dbpedia:York,_New_York`, `dbpedia:28220_York`, and seven others. Each of those is in turn bound to S_1 ($= Q$), which gets stored in S'_1.

We now follow the recursive invocation of `FindVariableBindings` with $Q \leftarrow S'_1 = S_1$ bound to `?c` = `dbpedia:York` and $F \leftarrow F'_m$. The graph pattern query Q thus becomes:

t_1 `?p a dbpedia-owl:Artist.`
t_2 `?p dbpedia-owl:birthPlace dbpedia:York.`
t_3 `dbpedia:York foaf:name "York"@en.`

This time, there are two connected subpatterns: $S_1 = \{t_1, t_2\}$ with the first two triple patterns containing variable `?p`, and $S_2 = \{t_3\}$ with the last triple without variables. For S_2, only 1 matching triple exists, so this results in `nil` as no binding is necessary. For each triple pattern in S_1, basic LDFs are retrieved (F_1 was cached):

	fragment selector	**matches**
F_1	`{ ?p a dbpedia-owl:Artist }`	68,237
F_2	`{ ?p dbpedia-owl:birthPlace dbpedia:York }`	75

Since F_2 has the lowest number of matches, it is used for $F'_m = F_m$. Possible bindings for `?p` include `dbpedia:Paul_Banks_(musician)`, `dbpedia:Eddie_Robson`, `dbpedia:Thomas_Turton`, and 72 others.

For each of them, `FindVariableBindings` is executed again. We will follow the execution with the following Q parameter:

t_1 `dbpedia:Eddie_Robson a dbpedia-owl:Artist.`
t_2 `dbpedia:Eddie_Robson …:birthPlace dbpedia:York.`

Both corresponding basic LDFs are retrieved (F_2 can be generated from cache). Essentially, we verify whether the triple t_1 exists, i.e., whether Eddie Robson is an artist according to the DBpedia dataset. F_1 and F_2 both have 1 matching triple, so the check at line 9 is successful; no further bindings are necessary.

By contrast, if we would follow the execution for the binding `?p` = `Thomas_Turton`, who was a mathematician, F_1 would have 0 matches, which results in the empty binding on line 8.

When all of the 75 possibilities have been scanned for artists, control is returned to the earlier `FindVariableBindings` invocation, which is now at line 21. Out of 75 matches for people with York as birthplace, 12 are artists. They are returned as part of B, which also contains the binding to `dbpedia:York`.

Execution continues similarly for the 11 other matches for `?c` at the highest level, most of which have no matches for `birthPlace`. Finally, the returned bindings B from this level are the following:

- `?c` = `dbpedia:York`
    - `?p` = `dbpedia:Eddie_Robson`
    - `?p` = `dbpedia:Dustin_Gee`
    - `?p` = `dbpedia:Paul_Banks_(musician)`
    - `?p` = `dbpedia:Johnny_Leeze`
    - `?p` = `dbpedia:Joe_Van_Moyland`
    - `?p` = `dbpedia:Mark_Simpson_(journalist)`
    - `?p` = `dbpedia:David_Reed_(comedian)`
    - `?p` = `dbpedia:Andrew_Martin_(novelist)`
    - `?p` = `dbpedia:Sam_Forrest`
    - `?p` = `dbpedia:Seebohm_Rowntree`
    - `?p` = `dbpedia:Peter_John_Allan`
    - `?p` = `dbpedia:John_Barry_(composer)`
- `?c` = `dbpedia:York,_Ontario`
    - `?p` = `dbpedia:Dawn_Langstroth`

From these bindings, the result set can be generated unambiguously. They are the same results we get when executing the query on the DBpedia SPARQL endpoint (given the same version of the dataset).

This algorithm has been implemented in the LDF client, which has been made available at *http://linkeddatafragments.org/software/*. Caching is added where possible, so the same LDF is only retrieved once—even though the algorithm might need it multiple times.

5.2 Querying other Linked Data Fragments

In Section 5.1, we explained how any basic graph pattern can be solved at the client side by retrieving basic LDFs. However, not all Linked Data servers will be partitioned (only) in *basic* LDFs; some will support more detailed LDFs (e.g., SPARQL endpoints), others will merely support less detailed fragments (e.g., Pubby subject pages [7]). While we believe that basic LDFs strike a fair balance between server effort and client effort, Algorithm 1 can be extended to optimally query servers with *any* LDF partitioning.

The main difference would be how a subpattern S_i is divided in fragment selectors (line 3). Since each basic LDF corresponds to a single triple pattern selector t_j, the original algorithm retrieves fragments F_j for each triple pattern. If the partitioning in LDFs is different, the subpattern can be divided in other fragment selectors to minimize the number of needed requests. However, each of those requests might be more (or less) expensive to a server, so the server should carefully consider which partitioning it offers.

Below are examples of possible LDF partitionings.

a (limited) SPARQL endpoint – If the server offers SPARQL, each SPARQL query corresponds to an LDF with that query as selector. While a full SPARQL endpoint would be able to solve any basic graph pattern directly, it would suffer from the aforementioned scalability issues. It is therefore beneficial to limit the possible query forms. For instance, if an endpoint would only allow SPARQL queries containing up to two triple patterns, the query discussed in Section 5.1 could be solved faster, since the `Artist`/`birthPlace` subpattern could be retrieved in one request instead of having to test for `Artist` 75 times.

basic LDFs with extra data – Any basic LDF server is free to send extra triples along that might be helpful to a client. For instance, a server could decide to always send the `rdfs:label` of any triple pattern component. That way, if the query in Listing 1 would additionally ask for artists' labels, no extra requests would be necessary.

only subject pages – When the server offers only subject-based dereferencing (such as DBpedia subject pages), triple patterns with variable subjects cannot be retrieved easily. In that case, a lot more requests are needed; the algorithm in fact becomes regular Linked Data querying with link traversal [18, 19].

Before a client can decide how a subpattern can be divided, it must know the available partitioning(s) of the server. They can be advertised in RDF using VOID and/or other vocabularies.

6. EVALUATION

6.1 Experimental design

The main characteristic of basic LDFs is that they allow a much higher availability and scalability than other LDF partitionings such as SPARQL result sets. The primary purpose of this evaluation is thus to verify whether the availability and scalability of LDF client/server setups is significantly higher than that of SPARQL endpoints. To this end, we will execute a series of SPARQL queries against a SPARQL endpoint and through clients of a basic LDF server.

We built a prototype implementation of a basic LDF client that can execute SPARQL queries consisting of basic graph patterns against a basic LDF server. Since existing benchmark suites [4, 29] use additional features such as filters, we could not meaningfully reuse them here. An examination of logs from popular endpoints such as DBpedia revealed that these are presently not the best sources of varied, non-trivial queries consisting of only basic graph patterns. We therefore developed a generator of basic graph pattern queries, available at *https://github.com/LinkedDataFragments/Benchmarks*, which can provide us with varied queries for any given dataset. The algorithm generates basic graph pattern queries $Q = \{q_1, \dots, q_n\}$, where each query q_i consists of triple patterns, using the following:

1. Select a random type `<t>` from the dataset (`{_:s a ?t}`) and add the triple pattern `{?s1 a <t>}` to the query.
2. Select a random subject `<s>` with this type (`{?s a <t>}`).
3. Select a random property `<p>` of this subject (`{<s> ?p _:o}`) and add the pattern `{?s1 <p> ?o1}`.
4. Select matching objects `<o1>` and `<o2>` (`{<s> <p> ?o}`).
5. For non-literal `<o1>` and `<o2>`, find triples `{<o> ?p3 ?o3}` and possibly `{<o3> ?p4 ?o4}`, using the results to augment the query with further triple patterns.

Below is an example query for DBpedia generated by this algorithm:

```
SELECT * WHERE {
  ?s1 a dbpedia-owl:Agent.
  ?s1 dbpedia-owl:associatedMusicalArtist ?o1.
  ?o1 dbpedia-owl:genre ?o2.
  ?o1 dbpedia-owl:recordLabel ?o3.
  ?o2 a dbpedia-owl:Genre.
  ?o3 rdfs:label "Paramount Records"@en.
}
```

We can see this is representative for the kind of queries we would be interested in on DBpedia. The algorithm generated 275 such queries.

6.2 Experimental setup

For this experiment, we installed Virtuoso 7 and our basic Linked Data Fragments server on a Ubuntu Linux machine (four 6-core processors at 2.4 GHz, 24 GB RAM). Virtuoso was configured with the recommended optimal settings, but result caching was disabled to ensure the results were served from the database and not from memory. The English DBpedia 3.8 was then ingested (427,670,470 triples).

Two different data sources were configured on the basic LDF server (as in Figure 2). The first one is the Virtuoso 7 server described above. The basic LDF server will execute two types of queries: 1) `CONSTRUCT` queries for basic graph patterns; 2) `COUNT` queries for the same. While Virtuoso can execute the former really fast, counts for large result sets are inherently slow. Therefore, we configured a second data source with DBpedia in the HDT format (Header, Dictionary, Triples) [10]. HDT is a compressed format for RDF that allows fast triple patterns queries and fast (approximate) counts. The fact that SPARQL only allows exact counts—even though approximations are sufficient for basic LDFs—is a major advantage for HDT.

The HTTP load testing tool JMeter was used to test the throughput of queries with a distributed setup, alternating between 1, 2, and 4 physical client machines that each attempted to execute 10 queries per second. If no response was received within 60 seconds, this was considered a timeout. A monitor on the server sampled the CPU and RAM usage of the data source processes every second.

6.3 Availability/scalability results

Table 1 shows the averages of the measured CPU and RAM usage for the Virtuoso process and the HDT process. When we execute the queries against the SPARQL endpoint, the CPU load on the Virtuoso process is high and increases linearly with the number of clients; RAM usage also increases steadily. Extrapolating the CPU usage, our 24-core server could handle at most ±20 clients at the same time. Beyond that, availability would become compromised.

The basic LDF server with Virtuoso as back-end handles an increasing number of clients with less CPU load; CPU deltas are also lower. Remarkably, RAM usage remains constant, likely due to the fact that no join operations need to be performed but only basic selections and counts. Extrapolation reveals the server handles ±46 clients.

However, if we choose a data source that is optimized for basic triple patterns and counts, such as HDT, we see that the scalability and resulting availability could be improved drastically. HDT is not CPU-bound or RAM-bound, as it basically streams the needed segments from disk. We hardly see the influence with a low number of clients. Note however that these numbers only include the data access part and not the cost of handling the HTTP interactions; they will probably be the first bottleneck in most scenarios.

The above indicates basic LDF servers scale better than SPARQL endpoints and thus can guarantee a much higher availability, certainly with data sources optimized for triple pattern access and counts.

6.4 Performance results

Table 2 summarizes the media and average query times for our test set, and the percentage of queries that time out (time > 60 s). Note how the median is in all cases far lower than the average, indicating that there are outliers with a high query time.

Without any doubt, a SPARQL endpoint such as Virtuoso solves SPARQL queries much faster under availability. However, solutions generated by basic LDF clients do not require excessive time: results generally arrive in a matter of seconds. The query time for the basic LDF server with 4 clients increased in our tests, yet this was not due to the data process, as Table 1 reveals, but due to the HTTP server process, to which more CPU cycles could be allocated. Additionally, regular HTTP caching would allow major performance improvements for the LDF server—and unlike SPARQL, even across different queries.

server type *data source*	**number of clients**	**average CPU usage**	**average RAM usage**
SPARQL endpoint	1 client	121.62%	3.81 GB
Virtuoso 7	2 clients	241.51%	4.52 GB
	4 clients	477.96%	5.18 GB
basic LDF server	1 client	66.58%	3.32 GB
Virtuoso 7 back-end	2 clients	82.35%	3.32 GB
	4 clients	116.30%	3.41 GB
basic LDF server	1 client	0.60%	0.98 GB
HDT back-end	2 clients	0.67%	1.09 GB
	4 clients	0.48%	0.88 GB

Table 1: Virtuoso's CPU load increases steadily with the number of clients; the LDF server slows this down. HDT is not CPU-bound at all.

server type *data source*	**number of clients**	**median time**	**average time**	**time-outs**
SPARQL endpoint	1 client	753 ms	2,338 ms	1.09%
Virtuoso 7	2 clients	837 ms	2,544 ms	1.45%
	4 clients	902 ms	2,623 ms	1.82%
basic LDF server	1 client	1,539 ms	6,136 ms	4.73%
Virtuoso 7 back-end	2 clients	1,551 ms	6,275 ms	5.09%
	4 clients	1,743 ms	6,214 ms	3.73%
basic LDF server	1 client	907 ms	3,460 ms	2.18%
HDT back-end	2 clients	922 ms	3,520 ms	2.18%
	4 clients	1,333 ms	5,044 ms	2.55%

Table 2: The SPARQL endpoint offers faster query execution times under availability, but LDF querying times remain acceptable.

7. DISCUSSION

7.1 Linked Data Fragments in the Semantic Web context

The SPARQL language and protocol have always been important to the Semantic Web's infrastructure, and we do not see a necessity for this to change. However, we do question the scalability of public endpoints that aim to offer unrestricted queries to a large number of users. The main strength of the endpoint philosophy is also its Achilles' heel: the fact that one server accepts the responsibility of answering arbitrarily complex requests inevitably leads to availability problems, as evidenced by recent statistics [6]. It is important to understand that this cannot be solved by building more efficient SPARQL servers—the problem is inherent to the *concept* of such a powerful endpoint. The resource partitioning of regular HTTP servers on the Web can be chosen by its developers in such a way that each resource can be delivered within acceptable bounds; the resources of SPARQL endpoints are query results that can be unpredictably complex. After all, for every 100 distinct queries a certain SPARQL server can answer in one second, there exists at least one query it cannot: the union of those queries.

We do see several important roles for SPARQL endpoints. First, as a private or internal data source, since the load is predictable; for instance, as a back-end of Web or desktop applications, similar to how relational databases are used. Second, when the query forms are somehow constrained; for instance, by limiting the allowed number of triple patterns or the execution time. Third, in environments where the number of users is limited; for instance, for highly specialized datasets. In those cases, the product of query variety, complexity, and access rate, which correlates with endpoint load, is minimized because one of its factors is controlled. For public SPARQL endpoints with a high number of users, the only option to guarantee high availability is to limit query complexity, but this often conflicts with the motivations for offering a queryable endpoint in the first place.

This paper pleads to move the intelligence that enables querying from the server to the client side. As clients have become increasingly powerful compared to servers—even mobile devices now exceed older laptops' capabilities—a model in which the client performs most of the work is realistic. This results in a significant increase in scalability, as depicted in Figure 1b. Even though clients have to issue many more requests, each of those requests 1) requires minimal server processing cost—and the server decides how much effort it is willing to spend; 2) can be cached and reused across different clients, as the granularity of responses is much finer. As the Web has been designed with per-resource access and caching [11, 13], ensuring that each resource can be rapidly generated and subsequently reused contributes more to availability and scalability than offering highly specific and expensive resources.

Performance-wise, the LDF querying approach cannot outperform SPARQL; availability-wise, it certainly can. This has a considerable impact on average query times, as shown in Table 3. If we look at the **SPARQL *a*** scenario in which the server has 99% availability (which is only the case with one third of SPARQL endpoints [6]), and assuming the average episode of downtime lasts 15 minutes, then an average query time of 0.2 seconds *under availability* comes down to an average query time of 4.70 seconds *in general*:

$$0.2\,\mathrm{s} + (1 - 0.99) \times (15\,\mathrm{min} / 2) = 4.70\,\mathrm{s}$$

If we look at those endpoints with 95% availability (**SPARQL *b***, less than half of all endpoints [6]), the generalized average query time increases to 45 seconds. In contrast, maintaining a 99.9% availability level for a basic LDF server is reasonable; with increased query times of 5 and 10 seconds, the generalized average query times become respectively 5.15 and 10.15 seconds (Table 3).

use case	average query time	server availability	average downtime	adjusted query time
basic LDF *a*	5.0 s	99.9%	5 min	5.15 s
basic LDF *b*	10.0 s	99.9%	5 min	10.15 s
SPARQL *a*	0.2 s	99.0%	15 min	4.70 s
SPARQL *b*	0.2 s	95.0%	30 min	45.20 s

Table 3: Average query times, adjusted according to availability.

So while SPARQL is certainly an order of magnitude faster under availability, actual availability percentages are sufficiently low that, when considering them in the average query time calculation, the difference with LDF querying becomes much smaller. The trade-off is the increased usage of bandwidth, which might be acceptable for desktop devices but perhaps difficult for mobile devices on slow connections. The improved caching can partly compensate for this.

7.2 Linked Data Fragments and Linked Data

Above all, Linked Data Fragments are a publishing strategy for Linked Data, with basic LDFs offering a partitioning that allows client-side querying at low server-side cost. Implementing basic LDFs can be seen as adding additional constraints to the Linked Data principles [2]: each basic LDF with a fixed subject (`{ <s> ?p ?o }`) has its own HTTP URI, represents triples about a certain subject, and includes links to other documents that allow to discover more things (all related basic LDFs). As discussed in Section 4.3, the URI-based dereferencing concept is retained, and actually augmented with hypermedia controls that allow to retrieve *different* fragments about a topic. For instance, while dereferencing *http://dbpedia.org/ontology/Artist* only leads to DBpedia's metadata of `Artist`, the same URI allows basic LDF servers to show 1) their *own* metadata of `Artist`; 2) all resources that have type `Artist`. Dereferencing a topic's URI on a basic LDF server might lead to the fragment of those triples that have the topic as subject, and this fragment contains controls towards all other basic LDFs of that dataset.

Additionally, the way we have defined LDFs in Definition 1 allows to consider all existing published Linked Data sets as Linked Data Fragments; an LDF is literally any "fragment" of a Linked Data source. All of the following are LDF partitionings, from coarse- to fine-grained: a single-file data dump in Turtle format, a dataset exposed as subject pages, a collection of basic LDFs, a SPARQL endpoint. Furthermore, the algorithm discussed in Section 5 and its generalization allow to query any LDF partitioning, thereby providing a means to evaluate which partitioning is best for efficient client-side querying—while still guaranteeing server availability.

Basic LDFs are not the only way of partitioning, but they set an example for novel ways to publish Linked Data, with a focus on enabling more intelligent clients through added metadata and hypermedia controls. It would be interesting to see which other LDF partitionings emerge and how they influence client capabilities.

Of crucial importance is the independence of clients and servers. While SPARQL is an expressive language, its use in a contract between a client and a server determines to a certain extent the way a client operates and behaves. Basic LDF servers impose a much less strict contract. The resources they offer *can* be used to solve SPARQL queries, but not that is not their only purpose. They can be used for browsing, to solve queries in other languages or even without a specific query language, to solve SPARQL queries partially (if not all results are needed), and for several other purposes. In that way, LDFs can enable a more serendipitous reuse [34] of Linked Data that is able to transcend individual data silos. The key to such an approach is that the client is in control of recombining individual pieces of data, and inter-fragment links aid this combination process.

7.3 Towards a new querying paradigm

In addition to improving the availability of queryable Linked Data sources, we believe that client-side querying can contribute to a new querying paradigm. SPARQL approaches querying in the traditional, non-Web-specific way: a client asks a question, the server computes the answer while the client waits, and finally, the client receives the whole answer at once. However, we should ask ourselves how realistic and desirable such a single delineated answer is in the context of an open and unpredictable Web. SPARQL endpoints of course never pretend to offer complete answers (they cannot, because no data source is ever complete); but each query is answered with a finite-length response, and the entire query needs to be asked again to check whether there are any changes.

Therefore, when we say "Web-scale querying", we do not only mean our method of querying can technically scale with an increasing number of clients; we also mean that LDFs are able to embrace the open nature of the Web. Even though we presented the querying algorithm in Section 5 in a synchronous way, its steps can actually be completed asynchronously and iteratively, streaming intermediate results as soon as they become available. This is not unlike other Linked Data querying strategies [18], but on a smaller time scale because of the more efficient partitioning of the data source.

Concretely, partial results can be communicated as soon as they are known. Revising the artists query example discussed in Section 5, the URI of a person born in York could be sent directly after it has been determined that he/she is an artist, without having to wait until all other 74 York inhabitants have been checked. This improves the latency of applications on behalf of which the requests are made, as the results can either be shown iteratively as they arrive (for instance, visualized on a map), or the first incoming results might already be sufficient to make a decision. It makes sense to trade the idea that delineated queries demand delineated answers for a more fluid way of querying answering. In theory, the artist query could even run indefinitely, returning new answers as DBpedia (or any other data source) gets updated. In some cases, tentative answers might also be useful, e.g., "this person is a potential match because she lives in a city named York, but the verification whether she is an artist is pending".

Another aspect of being Web-scale is the use of more than one data source. While SPARQL federated query [28] enables querying data from multiple SPARQL endpoints, low SPARQL endpoint availability makes the mechanism brittle. After all, if two endpoints each have an availability of 95%, the *a priori* probability of both endpoints being available decreases to $95\% \times 95\% \approx 90\%$.

LDFs allow querying of distributed sources in a transparent way. Since the mechanism of basic LDFs is based on hyperlinks, each LDF can link to LDFs on the same or another server. At no point in Sections 4 and 5 have we used any knowledge about the server's URI structure, because only links and forms were followed. If, for any reason, those links lead to another server, the querying algorithm can be completed as usual. Furthermore, the client can decide to have multiple starting fragments; for instance, it might ask birth place information from DBpedia and use BBC MusicBrainz to verify whether somebody is an artist. Interestingly, in sharp contrast to SPARQL federation, LDF querying actually becomes *faster* when using different data sources, because the HTTP requests are distributed across different servers. The use of different data sources also fits well with iterative results: DBpedia might not contain the birthplace of a certain person, while Freebase does.

We end up with an information-gathering process that bears more similarities with the way human consumers would answer questions. Instead of posing the question to an omniscient oracle, we consult targeted data sources to refine an answer iteratively.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the concept of Linked Data Fragments, discussed the development of LDF servers and clients, and made example implementations of a server and a client available at *http://linkeddatafragments.org/software/*. We thereby aim to facilitate further experiments with more intelligent clients, starting with a different resource publication strategy at the server side. Below are various directions for future work.

Above all, this paper strives to encourage research into offering datasets as fragments in addition to traditional partitionings such as data dumps, subject pages, and SPARQL endpoints. Even though basic LDFs already improve scalability and illustrate the powerful architectural properties of fragments, they are likely not the final destination of the quest for scalability. Other specifically designed partitionings could reduce bandwidth, which would significantly improve performance. The example query in Section 5 required 75 artist type checks; if they could somehow be bundled into fewer requests, the entire query can execute much faster. One way to do this is at the protocol level, for instance using HTTP 2.0 [1], which allows to send multiple requests to a single server more efficiently. Another way is a more granular partitioning than basic LDFs, so that multiple similar triple patterns can be queried at once. For instance:
`dbpedia:{Dustin_Gee,Thomas_Turton} a dbpedia-owl:Artist.`
This would decrease the number of needed requests, but each individual request would become more expensive. Furthermore, caching efficiency would be reduced. It is again up to the server to decide how much processing time it is willing to spend on each resource.

This brings us to another important research topic, namely how servers can indicate what kind of resource partitioning they support. A straightforward approach would be to create a vocabulary for different types, such as "single-file data dump", "subject pages", "basic LDFs", and "limited/full SPARQL". However, we envision that different kinds of LDF partitionings will emerge, and that these might even vary dynamically depending on server load. Perhaps a semantic way to express the data, metadata, and hypermedia controls of each fragment will be necessary.

A next technological step is the implementation of a streaming client. At the moment, the current algorithm and implementation follow a bottom-up approach, where each iteration downloads all pages from the smallest fragment. A top-down approach with a data pipeline would read fragment data one page at a time. This would make partial results available earlier, and thus allow faster decisions.

This paper has focused on querying basic graph patterns. In time, the full expressivity of the SPARQL query language could be supported efficiently as well. This would involve support for filters; one way to implement them is to offer LDFs with regular expression selectors. Such features would then also be indicated by a server.

In order to enable LDF querying in an uniform way, we should look at standardizing basic LDFs and related technologies. A first effort is our website *http://linkeddatafragments.org/*, which offers documentation and example source code, as well as LDF sources.

Finally, we are eager to explore links between LDFs and other technologies and standards. In particular, we see an important role for provenance [14] to explain how a client obtained an answer and what data sources were used in the process.

With Linked Data Fragments, we have introduced a novel way to look at Linked Data querying. By adjusting the granularity of information and equipping each fragment with metadata and the controls needed to find others, clients become able to consume Linked Data in more flexible ways. We believe the best way to make intelligent clients happen is to stop creating intelligent servers. The ultimate objective of Linked Data Fragments is therefore to build servers that foster intelligent clients.

9. ACKNOWLEDGMENTS

Ruben wishes to thank Richard Cyganiak for insightful discussions, Mario Arias for his help with HDT, and Johannes Lorey for suggestions on related work.
The described research activities were funded by Ghent University, the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research Flanders (FWO Flanders), and the European Union.

10. REFERENCES

[1] Belshe, M., Peon, R., Thomson, M., Melnikov, A.: Hypertext Transfer Protocol version 2.0. Internet draft, Internet Engineering Task Force (Dec 2013), *http://tools.ietf.org/html/draft-ietf-httpbis-http2-09*

[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 (Mar 2009), *http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf*

[3] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia – a crystallization point for the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web 7(3), 154–165 (2009), *http://www.websemanticsjournal.org/index.php/ps/article/view/164*

[4] Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. International Journal on Semantic Web and Information Systems 5(2), 1–24 (2009), *http://wifo5-03.informatik.uni-mannheim.de/bizer/pub/Bizer-Schultz-Berlin-SPARQL-Benchmark-IJSWIS.pdf*

[5] Bottomley, J.E.J.: Implementing clusters for high availability. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. USENIX Association (2004), *http://dl.acm.org/citation.cfm?id=1247415.1247459*

[6] Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: SPARQLWeb-querying infrastructure: Ready for action? In: Proceedings of the 12th International Semantic Web Conference (Nov 2013), *http://link.springer.com/chapter/10.1007/978-3-642-41338-4_18*

[7] Cyganiak, R., Bizer, C.: Pubby – a Linked Data frontend for SPARQL endpoints, *http://wifo5-03.informatik.uni-mannheim.de/pubby/*

[8] Cyganiak, R., Zhao, J., Alexander, K., Hausenblas, M.: Vocabulary of Interlinked Datasets (VOID). Interest group note, World Wide Web Consortium (Mar 2011), *http://www.w3.org/TR/media-frags/*

[9] Feigenbaum, L., Williams, G.T., Clark, K.G., Torres, E.: SPARQL 1.1 protocol. Recommendation, World Wide Web Consortium (Mar 2013), *http://www.w3.org/TR/sparql11-protocol/*

[10] Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Journal of Web Semantics 19, 22–41 (Mar 2013), *http://dx.doi.org/10.1016/j.websem.2013.01.002*

[11] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California (2000), *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*

[12] Fielding, R.T.: REST APIS must be hypertext-driven. Untangled – Musings of Roy T. Fielding (Oct 2008), *http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven*

[13] Fielding, R.T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol (HTTP). Request For Comments 2616, Internet Engineering Task Force (Jun 1999), *http://tools.ietf.org/html/rfc2616*

[14] Groth, P., Moreau, L.: PROV overview. Working group note, World Wide Web Consortium (Apr 2013), *http://www.w3.org/TR/prov-overview/*

[15] Harris, S., Seaborne, A.: SPARQL 1.1 query language. Recommendation, World Wide Web Consortium (Mar 2013), *http://www.w3.org/TR/sparql11-query/*

[16] Hartig, O.: How caching improves efficiency and result completeness for querying Linked Data. In: Proceedings of the 4th Workshop on Linked Data on the Web (Mar 2011), *http://ceur-ws.org/Vol-813/ldow2011-paper05.pdf*

[17] Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: Proceedings of the 8th Extended Semantic Web Conference on The Semantic Web. pp. 154–169. Springer (2011), *http://dl.acm.org/citation.cfm?id=2008892.2008906*

[18] Hartig, O.: An overview on execution strategies for linked data queries. Datenbank-Spektrum 13(2), 89–99 (2013), *http://dx.doi.org/10.1007/s13222-013-0122-1*

[19] Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL queries over the Web of Linked Data. In: Proceedings of the 8th International Semantic Web Conference. pp. 293–309. Springer (2009), *http://www2.informatik.hu-berlin.de/~hartig/files/HartigEtAl_QueryTheWeb_ISWC09_Preprint.pdf*

[20] Kjernsmo, K.: The necessity of hypermedia RDF and an approach to achieve it. In: Proceedings of the Workshop on Linked APIS for the Semantic Web (May 2012), *http://lapis2012.linkedservices.org/papers/1.pdf*

[21] Lanthaler, M., Gütl, C.: Hydra: A vocabulary for hypermedia-driven Web APIS. In: Proceedings of the 6th Workshop on Linked Data on the Web (May 2013), *http://ceur-ws.org/Vol-996/papers/ldow2013-paper-03.pdf*

[22] Lorey, J., Naumann, F.: Caching and prefetching strategies for SPARQL queries. In: Proceedings of the 3rd International Workshop on Usage Analysis and the Web of Data (May 2013)

[23] Lorey, J., Naumann, F.: Detecting SPARQL query templates for data prefetching. In: Proceedings of the 10th Extended Semantic Web Conference (May 2013)

[24] Martin, M., Unbehauen, J., Auer, S.: Improving the performance of Semantic Web applications with SPARQL query caching. In: The Semantic Web: Research and Applications, Lecture Notes in Computer Science, vol. 6089, pp. 304–318. Springer (2010), *http://dx.doi.org/10.1007/978-3-642-13489-0_21*

[25] Marwah, M., Maciel, P., Shah, A., Sharma, R., Christian, T., Almeida, V., Araújo, C., Souza, E., Callou, G., Silva, B., Galdino, S., Pires, J.: Quantifying the sustainability impact of data center availability. SIGMETRICS Performance Evaluation Review 37(4), 64–68 (Mar 2010), *http://doi.acm.org/10.1145/1773394.1773405*

[26] Nottingham, M.: Feed paging and archiving. Request For Comments 5005, Internet Engineering Task Force (Sep 2007), *http://tools.ietf.org/html/rfc5005*

[27] Pautasso, C., Wilde, E.: Why is the Web loosely coupled? – A multi-faceted metric for service design. In: Proceedings of the 18th International Conference on World Wide Web. pp. 911–920. ACM, New York (2009), *http://www2009.eprints.org/92/1/p911.pdf*

[28] Prud'hommeaux, E., Buil-Aranda, C.: SPARQL 1.1 federated query. Recommendation, World Wide Web Consortium (Mar 2013), *http://www.w3.org/TR/sparql11-federated-query/*

[29] Schmidt, M., Hornung, T., Meier, M., Pinkel, C., Lausen, G.: SP^{2}Bench: A SPARQL performance benchmark. In: Semantic Web Information Management, pp. 371–393. Springer (2010), *http://dx.doi.org/10.1007/978-3-642-04329-1_16*

[30] Shadbolt, N., Berners-Lee, T., Hall, W.: The Semantic Web revisited. Intelligent Systems 21(3), 96–101 (Jul 2006), *http://eprints.soton.ac.uk/262614/*

[31] Shu, Y., Compton, M., Müller, H., Taylor, K.: Towards content-aware SPARQL query caching for Semantic Web applications. In: Web Information Systems Engineering, Lecture Notes in Computer Science, vol. 8180, pp. 320–329. Springer (2013), *http://dx.doi.org/10.1007/978-3-642-41230-1_27*

[32] Speicher, S., Arwe, J., Malhotra, A.: Linked Data Platform 1.0. Working draft, World Wide Web Consortium (Jul 2013), *http://www.w3.org/TR/2013/WD-ldp-20130730/*

[33] Troncy, R., Mannens, E., Pfeiffer, S., Van Deursen, D.: Media fragments URI 1.0 (basic). Recommendation, World Wide Web Consortium (Sep 2012), *http://www.w3.org/TR/media-frags/*

[34] Vinoski, S.: Serendipitous reuse. Internet Computing 12(1), 84–87 (Jan 2008), *http://steve.vinoski.net/pdf/IEEE-Serendipitous_Reuse.pdf*

[35] Williams, G.T., Weaver, J.: Enabling fine-grained HTTP caching of SPARQL query results. In: Proceedings of the 10th International Conference on The Semantic Web. pp. 762–777. Springer (2011), *http://www.cs.rpi.edu/~weavej3/papers/iswc2011.pdf*

[36] Wu, G., Yang, M.d.: Improving SPARQL query performance with algebraic expression tree based caching and entity caching. Journal of Zhejiang University SCIENCE C 13(4), 281–294 (2012), *http://dx.doi.org/10.1631/jzus.C1101009*