Substring Filtering for Low-Cost Linked Data Interfaces

Joachim Van Herwegen(✉), Laurens De Vocht, Ruben Verborgh, Erik Mannens, and Rik Van de Walle

Multimedia Lab, Ghent University – iMinds,
Gaston Crommenlaan 8 Bus 201, 9050 Ledeberg-Ghent, Belgium
joachim.vanherwegen@ugent.be

Abstract. Recently, Triple Pattern Fragments (TPFS) were introduced as a low-cost server-side interface when high numbers of clients need to evaluate SPARQL queries. Scalability is achieved by moving part of the query execution to the client, at the cost of elevated query times. Since the TPFS interface purposely does not support complex constructs such as SPARQL filters, queries that use them need to be executed mostly on the client, resulting in long execution times. We therefore investigated the impact of adding a literal substring matching feature to the TPFS interface, with the goal of improving query performance while maintaining low server cost. In this paper, we discuss the client/server setup and compare the performance of SPARQL queries on multiple implementations, including Elastic Search and case-insensitive FM-index. Our evaluations indicate that these improvements allow for faster query execution without significantly increasing the load on the server. Offering the substring feature on TPF servers allows users to obtain faster responses for filter-based SPARQL queries. Furthermore, substring matching can be used to support other filters such as complete regular expressions or range queries.

Keywords: Linked data · SPARQL · String matching · Regular expressions

1 Introduction

The publication of RDF data on the Web is often presented as a dichotomy [6]: either the entire dataset is made available in a downloadable data dump, or fine-grained query-level access is provided through a public SPARQL endpoint [7]. In the first case, users need to download the entire dataset—even if they are only interested in a specific part of it—and are then free to use it locally in any way they see fit. Commonly, this means ingesting the triples in a local triple store and setting up a private SPARQL endpoint. While this approach is most straightforward for data publishers and offers most freedom for data consumers, it also comes with a high burden in terms of needed bandwidth, device capacity, and technical ability of the data consumer. Especially given the rise of mobile devices to browse the Web, offline querying of datasets is not viable as the only

M. Arenas et al. (Eds.): ISWC 2015, Part I, LNCS 9366, pp. 128–143, 2015.
DOI: 10.1007/978-3-319-25007-6_8

solution. Furthermore, if datasets change often, synchronizing them with the latest version becomes a challenge.

In the second case, public SPARQL endpoints allow users to execute all of their queries on the server against live data, without having to worry about huge downloads or updates. However, since all work is shifted to the server, the endpoint's load increases drastically, causing an increase in server cost and potential overloads and subsequent downtime if many complex queries are executed. Even though most public servers pose certain limits to query execution, the low availability of SPARQL endpoints is a prominent issue [3], and high load is a potential availability risk. Furthermore, from an economic perspective, not all data publishers on the public Web are willing and/or able to pay for a computational infrastructure that allows third parties to execute complex queries free of charge, given that they already provide the data itself for free. Whether or not the duties of a publisher include live queryable access, the fact is that only a minority attempts to host a public SPARQL endpoint [6,20].

Recently, the Triple Pattern Fragments (TPFs) interface was introduced as an alternative to bridge the gap between the low server load of datasets and the functionality of a full SPARQL endpoint. TPFs limit the functionality of the server to triple pattern requests; more complex SPARQL queries are executed by a *client-side* query processor that decomposes them into triple patterns. Clients combine intermediary results received from the server locally to find results to larger queries. This greatly reduces the server load by shifting querying partially to the client, at the expense of increased query times and bandwidth. While basic graph pattern queries can be executed efficiently with TPFs, other query constructs potentially lead to slow queries. The TPF interface, however, is self-describing, meaning we can transparently add new features to it. While they might make individual requests more expensive, the number of requests—and thereby the total sum of all request costs—might be significantly reduced.

In this paper, we investigate a server-side interface feature that allows SPARQL queries with certain `FILTER` patterns to be executed more efficiently. With the regular TPF interface, all filters have to be evaluated client-side, since only exact triple patterns have server-side support. Concretely, this means that clients first have to download all triples that match the remainder of the query, after which the filter can be applied to these results. This works sufficiently for filters with low selectively, which SPARQL endpoints typically also execute at the end. However, if the filter is the only part of a query that has a high selectivity, the client cannot solve this query efficiently, since the filter can only be used on triples that were already downloaded from the server. For this reason, we introduce *substring matching* as a server-side interface, allowing the user to request all literal objects that contain a given string pattern.

In Section 2 we discuss related work, especially regarding query interfaces and pattern matching, and show how it is related to our current work. Section 3 explains the benefit of adding a substring feature to servers that publish LDFs, and Section 4 presents two implementations. We then extend the client-side query algorithm in Section 5 to make use of the substring feature and examine its impact on performance and server load in Section 6. We conclude in Section 7 and look at how this work can be extended in the future.

2 Related Work

This paper defines a server-side interface feature for substring search on RDF object literals. To place the contribution in context, we first discuss the multiple available interfaces to access Linked Data, followed by an introduction to the indexing algorithm used later on.

2.1 Web APIs to Linked Data

Linked Data can be published on the Web using different Application Programming Interfaces (APIs). The Linked Data Fragments conceptual framework [25,27] enables the analysis and comparison of Web APIs by abstracting each API according to how it provides access to parts of a given dataset. Each such part is called a *Linked Data Fragment* (LDF), which consists of data, metadata, and controls. The *data* is a set of those triples of the dataset that match a given interface-dependent selector. The *metadata* set consists of triples that describe the dataset and/or the current fragment or related fragments. Finally, the *controls* are hypermedia links and/or forms that allow clients to retrieve other fragments of the same or other datasets.

In addition to describing existing interfaces, LDFs also allow defining new interfaces with different characteristics. Below, we discuss three types of interfaces using the LDF conceptual framework.

Data Dumps. A data dump of a dataset is an LDF whose *data* consists of all triples in that dataset, usually in a compressed archive. The *metadata* set typically contains data such as publication date and/or license. No *controls* are present, because all available data is contained within the archive. Data dumps are prevalent on the Web: LODstats mentions more than 1,700 data dumps [6], and the LOD Laundromat contains more than 600,000 datasets crawled from the Web [20]. Their main drawback is that they cannot be queried "live", i.e., they need to be downloaded in their entirety before typical SPARQL queries can be evaluated over them. Since data dumps can become quite large (several gigabytes are not an exception), they are impractical for most use cases—especially if data changes often and needs to remain up-to-date. Furthermore, setting up and maintaining a query interface on top of a data dump requires a technical background and significant computational power, so this is out of reach for typical desktop users as well as all mobile users.

SPARQL Endpoints. The SPARQL protocol [7] exposes RDF graphs on the Web using the SPARQL query language [13]. Each response to a `CONSTRUCT` or `DESCRIBE` query can be seen as an LDF, where the *data* consists of the RDF triples in the dataset that match the query. The *metadata* and *control* sets are empty; controls are given implicitly through the SPARQL protocol. The main advantage of SPARQL endpoints is their expressiveness: clients can ask very specific questions about a dataset and retrieve only the results they are interested in. However, public SPARQL endpoints suffer from a two-sided availability problem: the majority of datasets is not published as a SPARQL endpoint [6], and those endpoints that are on the Web experience frequent downtime [3]. Furthermore, SPARQL endpoints have a high per-request cost [25] and are thus relatively expensive to host.

Triple Pattern Fragments. The Triple Pattern Fragments (TPF) interface [24,25] was designed to combine the desirable characteristics of data dumps (low server-side cost) and SPARQL endpoints (live queryable). Clients can ask a server for triple patterns; in response, the server sends a TPF, consisting of *data* triples matching the triple pattern (paged to reduce response sizes), *metadata* expressing the total number of matching triples, and *controls* to retrieve all other TPFs of the same dataset. Complex SPARQL queries are evaluated by clients, which split a query into triple patterns and use the metadata in fragments to determine an efficient execution order. The advantage of TPFs is that they only require low processing power on the server side, and are thus less expensive to host with high availability [25]. The drawback is that SPARQL queries have longer (but more consistent) query times than on a SPARQL endpoint. More than 600,000 datasets are available as TPFs through the LOD Laundromat [20]. DBpedia, the most well-known dataset on the Semantic Web, has an official TPF interface with 99.999% availability [26].

TPFs move the query planning problem to the client. It is up to the client to make optimal use of all metadata exposed by the TPF server, which in the default case consists of the estimated amount of matched triples for a triple pattern. Since a TPF server only supports triple patterns, complex SPARQL structures such as filters also have to be computed client-side. This can be done by checking all resulting triples against the filters in the SPARQL query. The originally proposed query planning algorithm is a greedy algorithm [25]. Assuming a Basic Graph Pattern (BGP) query, the client starts by downloading results for the smallest triple pattern in the BGP, based on the count estimate metadata sent by the server. The values of each resulting triple are bound to each remaining pattern. The client then requests the smallest of these bound patterns from the server, and continues binding results to unbound patterns until all patterns have been bound. This process is started multiple times if there are multiple unconnected BGP's. Computationally this method is quite fast: most of the time the client is simply waiting on the server response. The downside is that the client can become stuck in local optima, causing it to execute more requests than a theoretically optimal solution.

Van Herwegen et al. proposed an improvement to the greedy algorithm [22], which tries to find a solution using a minimum number of HTTP calls, avoiding local optima. This is achieved by downloading two triple patterns separately from the server and joining them on the client if this requires fewer HTTP calls. Multiple estimation techniques, based on the intermediate results of the algorithm, are used to predict which query path is least expensive. If the current path is suboptimal, the algorithm will change it at runtime and continue from the new path. This decrease in HTTP requests results, however, in more computational work for the client because of the more complex local joining process.

2.2 Burrows-Wheeler Transform and FM-Index

The Burrows-Wheeler Transform (BWT) [4] was created to transform data so that it can be compressed more easily without any loss [19]. It is used in multiple fields, such as bio-informatics [15].

An FM-index is an index on a BWT-transformed dataset to find substrings in the data by adding some additional metadata [10]. Brisaboa et al. show how an FM-index can even be used to perform substring matching in a list of strings instead of a single string, returning all strings that contain a given substring [2]. In Section 4.3, we extend this technique to also allow for case-insensitive matches.

Ferguson shows that it is even possible to execute regular expressions on data stored using an FM-index [8]. In his paper, he describes a system called FEMTO, which can index large datasets while still maintaining adequate performance.

2.3 HDT (Header Dictionary Triples)

HDT [9] is a data storage format that optimizes the space required to store large RDF datasets by storing its URIs and literals separately in a compressed dictionary and storing identifier triplets that reference this compressed data. Multiple dictionaries are supported by HDT; we are particularly interested in the dictionary that uses an FM-index to store the object literals, thus allowing full-text search on triple objects [1,16].

2.4 Full-Text Search in Triple Stores

HDT is not the only way to support full-text search on triples: there exist multiple other implementations supporting the same functionality or even more [5,17]. Minack et al. [18] performed an extensive comparison of multiple of these implementations in complete triple stores (unfortunately not including HDT since it is a data structure and not a triple store). They concluded that most standard triple stores have sufficient support for full-text search, but that there are still areas where performance is inadequate.

3 Problem Statement

A common use case for substring matching is when we know the name of an entity we are looking for, but we do not know its URI. As an example, Listing 1

```
SELECT ?movie
WHERE {
  ?movie dbpedia-owl:starring [ rdfs:label ?name ].
  FILTER REGEX(?name, "Matthias Schoenaerts", "i")
}
```

Listing 1. SPARQL query to find all movies with Matthias Schoenaerts

contains a query that returns all movies starring the actor Matthias Schoenaerts. We assume the user did not know the exact URI that was necessary, and used string matching to find the URI that corresponds to the person called "Matthias Schoenaerts". Since TPF servers only support exact triple pattern lookups, a client-side algorithm would need to execute the filter locally on all triples joined by the previous two triple patterns. The first pattern has 200,000 matches and the second one more than 12,000,000, meaning that any solution would need at least 200 calls (assuming a page size of 100) to obtain results for the first pattern, and then another 200,000 to map them to their label.

However, if it were possible to filter all literals in the dataset first, we would only obtain 20 results that would then need to be mapped to the previous patterns, resulting in a total of 40 HTTP requests instead of 200,000. In general, the ability to solve substring matches is especially useful when all of the triple patterns have a low selectivity and the string pattern selectivity is quite high. For this example, the string pattern in Listing 1 is highly selective since there are only 20 results. These observations lead us to the following research question.

Question 1: How does the performance of queries with `FILTER` patterns improve if the TPF interface is extended with substring search on literals?

Extending the interface means that clients are able to send more complex requests, which could mean a higher per-request cost for servers. At the time, however, an increased expressivity of requests could lead to a reduction in the number of requests needed to evaluate a particular SPARQL query. This brings us to a second research question:

Question 2: What is the server-side impact of adding support for substring search to the interface, i.e., can we still maintain the low-cost properties associated with the TPF interface?

Ultimately, the results of this research should help data publishers decide whether the costs of adding substring search are worth the expressivity and the possible improvement in query performance they bring.

Question 3: For which scenarios and types of queries is it beneficial to add a substring search interface to the server?

Based on the above research questions, we propose the following hypotheses:

Hypothesis 1: The HTTP requests required to solve typical queries with highly selective `REGEX FILTER`s can be greatly reduced when a substring search is present in addition to a TPF interface.

```
PREFIX void:  <http://rdfs.org/ns/void#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX hydra: <http://www.w3.org/ns/hydra/core#>

<#about> {
    <#about> foaf:primaryTopic <#fragment>.
    <#fragment> void:subset <http://example.org/mydataset>;
    <http://example.org/mydataset> hydra:search [
        hydra:template "http://example.org/mydataset{?substring}";
        hydra:mapping [ hydra:variable "substring";
                        hydra:property hydra:freetextQuery ]
    ].
}
```

Listing 2. Self-descriptive hypermedia controls in a TPF fragment explain how substring matching can be accessed (TriG syntax).

Hypothesis 2: Queries with `REGEX FILTER`s that are *not* highly selective are unaffected by the presence of a substring search interface.

Hypothesis 3: The cost to offer substring matching is limited such that substring requests can be executed at an acceptable time cost on a typical server.

4 Server-Side Interface

To support substring search, Section 4.1 defines an interface feature that can work in conjunction with the TPF interface. We evaluated two different techniques to filter through the data: an internal FM-index and an external Elasticsearch[1] index, which can be found in Section 4.2 and 4.3, respectively.

4.1 Extension of the TPF Interface

The TPF interface has been designed in a self-descriptive and extensible manner, so clients can discover what capabilities are supported [25]. Given a single URI to any resource of the interface, the client can fetch it with HTTP `GET`, asking for an RDF representation. In this representation, the clients will find hypermedia controls, which explain that "this interface can be queried by triple pattern" [24]. This avoids a hard-wired client/server contract. For example, if we visit the resource http://fragments.dbpedia.org/2014/en in a Web browser, we will see an HTML form with fields *subject*, *predicate*, *object*, and this form instructs the browser how to create HTTP requests against the interface. This same information is conveyed in the RDF-based representation of the same resource, using the Hydra Core Vocabulary [14, 24].

Since we aim to provide a substring search interface feature, we should similarly inform human and machine clients of this functionality and how they can

[1] http://www.elastic.co/

perform such requests. Listing 2 shows an example form in RDF for an example fragment of the dataset http://example.org/mydataset/. In this particular case, it states that substring search is supported on this dataset, and that it can be performed by appending the search string to http://example.org/mydataset?substring=. That way, with this resource as a starting point of their query process, clients can decide to use substring search during query execution. Should this not be supported, clients can decide to fall back to other features supported by the server, such as possibly TPF.

In order to explain the exact type of support for substring matching as implemented by the server, subproperties of `hydra:freetextQuery` might be defined, such as `ex:substringQuery` or `ex:caseInsensitiveSubstringQuery`. This then requires the client to understand such extensions; therefore, it might be beneficial to always additionally list the base property `hydra:freetextQuery` so that more generic clients can still interpret the hypermedia controls and thus use the interface. Note that due to this self-descriptive mechanism, no hard-coded contract between clients and servers is necessary, and conventions (such as `?substring=`) need not be standardized but can transparently vary between servers. Implementers can consult the LDF substring feature specification[2] for a detailed explanation on how to optimally describe the interface.

4.2 Elasticsearch

Elasticsearch is a search server based on the text indexing engine Lucene [12]. It provides a full-text search engine with an HTTP interface and schema-free JSON documents. The fact that it has a Web interface out-of-the-box and is particularly designed for use in such scenarios, made it an obvious choice as a back-end. Elasticsearch being a versatile search solution and inheriting Lucene's extensive capabilities, it is not trivial to tweak its configuration. It is very strict about data types and forces developers to think from the beginning about how text queries will be performed against the underlying data.

The use case mentioned in this paper—searching for arbitrary-length exact substrings in texts of varying sizes—is not provided by ElasticSearch by default. For example, when searching for the actor name "Will Smith", ElasticSearch' standard tokenizer would match sentences such as *"Will Mr. and Ms. Smith be an awesome movie?"*, even though this is not an exact substring match and thus not a desired result in our use case.

Therefore, it is necessary to force ElasticSearch' analyzer to keyword-tokenize each text literal and apply an n-gram analyzer to each of them. The result is a huge index file, about the size of the original dataset. Furthermore, generic n-gram matching is very costly: instead of query times in the order of milliseconds, query times were in the order of dozens of seconds. Configuring ElasticSearch to work with only prefixes using edge-n-grams, effectively dropping certain results, mitigated the issue of extremely long query processing times. One could argue that most users would input prefix queries and expect a text search engine to

[2] http://www.hydra-cg.com/spec/latest/linked-data-fragments/substring-search/

behave in that way. Whatever the assumption might be, for use in a query scenario as explained before, results need to be exact and complete.

In ElasticSearch, important choices have to be made to optimize query time, index size, the number of desired matches, and the nature of the returned matches given the text search query and the use case. This makes choosing it for a generic use case where the end-user or application can not be reckoned with at least more debatable than initially expected. The evaluation in Section 6 shines some light on this aspect.

4.3 Case-Insensitive FM-Index

HDT already included support for substring search by using an FM-index. One simply has to edit the config file to make sure the correct type of dictionary is used when generating the HDT file. The problem here was that default FM-index only supports case-sensitive substring search, while we wanted case-insensitive searching as well. We will start by briefly explaining the existing algorithms, followed by our changes to FM-index to make it case-insensitive.

Burrows-Wheeler Transform. As mentioned in Section 2.2, BWT is a technique developed to transform a string in such a way that identical characters appear next to each other more often after transformation, while still allowing for a reverse transformation to the original string [4]. Strings that have sequences of identical characters are a lot easier to compress when using methods like *move-to-front transform* and *run-length encoding*.

The BWT creates n permutations of a string with length n by cyclically shifting the characters in the string. These strings are sorted and placed in a matrix, with each row corresponding to one of the permuted strings. The first column then corresponds to a sorted list of all characters in the original string. Since it is impossible to generate the original string from this list, the BWT actually stores the last column of the matrix, assuming certain characters are more likely to precede certain other characters, causing the BWT result to have multiple identical characters next to each other. From this string it actually is possible to go back to the original.

FM-Index. An FM-index adds metadata to a BWT-transformed string so that it can be used for full-text search without actually reverting back to the original string. The extra metadata consists of two parts: an array C containing for each character the amount of characters that precede it in the sorted string, and a matrix Occ containing for every character c and every position i how many times c occurs up to position i in the BWT string. Note that this metadata can be calculated at runtime and does not need to be stored with the BWT string.

Using these additional elements it is possible to count the number of times a pattern occurs in $\mathcal{O}(p)$ time with p being the length of the pattern. Actually locating the pattern matches in the string takes $\mathcal{O}(p + occ \log_{\epsilon} u)$ with occ being the number of occurrences and u the length of the string.

FM-Index as a Dictionary. As mentioned before, it is possible to generate HDT files that use an FM-index to store the literal objects. Brisaboa et al. [2] describe an adaptation of FM-index to store a list of n strings instead of a single string, by concatenating all strings and separating them with $n+1$ occurrences of a separator character s_1 (corresponding to the ASCII value 1). Since the BWT sorts all the prefix strings, the first $n+1$ characters will be s_1, with the first one being the last s_1 of the concatenation and the remaining occurrences of s_1 sorted based on the strings they precede, which is the same order of the concatenated strings if these are sorted in advance. HDT uses the positions of each s_1 to internally assign IDs to the object literals, meaning that ID 1 corresponds to string 1 after sorting the strings.

When converting a part of a BWT string to an original string, it is only possible to do this backwards since for every character we only know which character precedes it. Since the IDs correspond to the position of the *first* character of a string, going backwards would give us the wrong result. As the strings are sorted in the same order as the IDs, simply starting at position ID + 1 will give us the result for a given ID.

Case-Insensitive BWT. Sadakane [21] introduces a way to use case-insensitive searching in a BWT string by changing the algorithm in such a way that lower- and upper-case characters are interpreted as identical. Once the suffix substrings have been sorted, substrings starting with the same character will be next to each other, even if the character has a different casing. This method can even be extended to treat all kinds of symbols as identical, such as accented characters.

Case-Insensitive FM-Index. Some changes are needed to adapt a case-insensitive BWT for an FM-index. After concatenating the (case-insensitively) sorted strings, we replace the last s_1 with s_{255} instead to make sure that if the last two strings are identical (when ignoring casing) they remain in the same order after generating the BWT string. Previously this was not a problem since it was impossible to have identical strings in the concatenation, but now, if we did not append this character, the ordering of their IDs would be reversed.

Some changes also had to be made to the lookup algorithm to take the different casings into account. The values in the C table are also combined: $C(\text{ A }) = C(\text{ a })$, which now corresponds to the number of characters preceding the A and a characters. Similarly, the *Occ* matrix values were also merged to count the number of occurrences of A and a characters.

Since we do not actually change the casing of the strings, no information is lost and all objects can still be obtained from the triple store. We do need to introduce an extra step to still support case-sensitive searching by effectively verifying if the resulting strings match the case of the pattern.

5 Client-Side Query Algorithm

To make use of the substring functionality of the TPF server we updated the query execution algorithm as described by Verborgh et al. [25]. This is a greedy algorithm focusing on BGP queries and evaluating all other query constructs (filters, unions, etc.) after the BGP parts are resolved. Since evaluating `FILTER`s on the server is only advantageous if these are executed *before* the corresponding BGPs, it was necessary to change this ordering.

To make use of this new feature to solve SPARQL queries, we adapted how the client handles regular expression filters. Regular expression filters obviously support much more than simple string matching, so we first check if the query we want to execute contains an expression that can be translated into a pattern matching problem. If the query contains such a filter, we evaluate whether it would be efficient to solve that filter before the BGP parts are executed. The standard greedy algorithm starts by finding the triple pattern with the lowest number of results, then binding these results to the next smallest pattern and so on until all results are bound. For our implementation, we also check if one of the substring expressions has at least 100 times less results than the smallest pattern (100 being the page size). This is still a greedy implementation, albeit one that takes advantage of the server-side substring feature.

6 Evaluation

6.1 Experimental Setup

We compare multiple situations to evaluate the impact of the substring feature. We want measure how the performance changed on both client and server on substring queries, we want to make sure we did not hurt the performance on non-substring queries, and we also want to compare the FM-index implementation against adding an external index such as Elasticsearch.

We executed both client and server on the same machine (Intel Core i5-3230M CPU at 2.60GHz with 8GB of RAM) while the Elasticsearch index was located on a different server (12 Intel Xeon E5-2640 CPU cores at 2.50GHz with hyperthreading, 64GB of RAM) in the same network with a ping time of < 1ms with the DBpedia2014 dataset without abstracts[3]. The Elasticsearch index is on a different server due to the extra memory requirements for the index. All results can be found online[4] as well as the server[5] and client[6] code.

We performed the following tests:

1. With both Elasticsearch and FM-index, request all objects containing a specific keyword through the TPF interface, meaning these have to be requested one page at a time. The keywords are sampled from the list created by Freitas et al.[11] in such a way that their number of results are spread out.

[3] http://wiki.dbpedia.org/Downloads2014
[4] http://github.com/LinkedDataFragments/TPF-Substring-Results/
[5] http://github.com/LinkedDataFragments/Server.js/tree/feature-substring-search
[6] http://github.com/LinkedDataFragments/Client.js/tree/substring

```
SELECT ?person ?city WHERE {
  ?club a dbpedia-owl:SoccerClub;
        dbpedia-owl:ground ?city.
  ?player dbpedia-owl:team ?club;
          dbpedia-owl:birthPlace ?city.
  ?city dbpedia-owl:country dbpedia:Spain.
}
```

Listing 3. SPARQL query: Spanish soccer players

2. Evaluate the query in Listing 1 against a regular TPF server and a TPF server with a substring feature using an FM-index. We used the actor "Johnny Depp", who has many substring matches, to have more robust results.
3. Perform a query not containing any substring requirements on both the original as the FM-indexed version of TPF. For this we used the query in Listing 3.

6.2 Results

Elasticsearch and FM-Index. The results of the keyword test can be seen in full in Table 1 and visualized in Figure 1. The differences in results are due to the fact that our Elasticsearch configuration uses prefix matching. We did not configure the Elasticsearch index for full substring search since it performs a lot worse there, both in lookup speed and index size. It is clear that FM-index performs really well here, scaling linearly with the amount matches because an increase in matches also results in an increase in pages that need to be requested from the server.

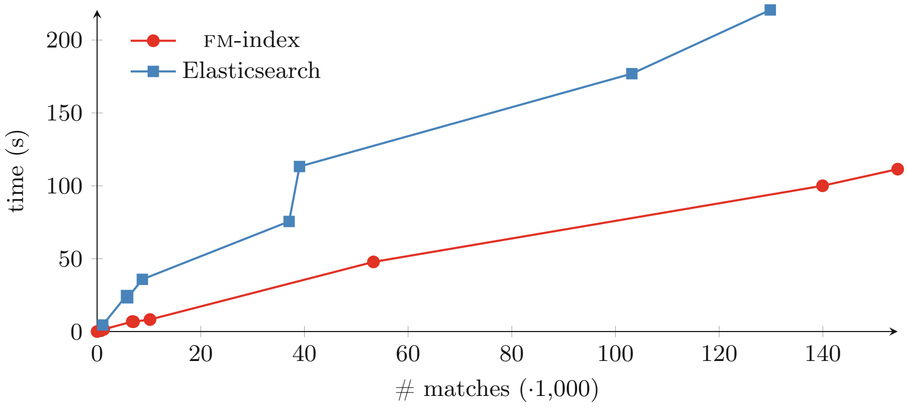

Fig. 1. Number of matches and query time for the keywords in Table 1.

Table 1. FM-index and Elasticsearch comparison.

	FM-index		**Elasticsearch**	
keyword	**# matches**	**time (ms)**	**# matches**	**time (ms)**
laptop	1,280	1,552	1,089	4,536
tools	6,795	6,900	5,679	24,758
photography	7,030	6,802	5,903	23,209
landing	10,211	8,302	8,735	35,863
computer	53,316	47,817	39,052	113,262
politician	139,952	100,052	103,159	176,938
sun	154,439	111,451	129,868	220,656
car	965,159	785,423	627,020	1,225,449

Table 2. Original implementation and FM-index comparison. CPU and memory strictly refer to the server process.

	"Johnny Depp"		***Spanish soccer***	
	original	**FM-index**	**original**	**FM-index**
http calls	304,154	174	91,658	91,694
time (ms)	1,189,706	1,352	329,287	319,065
average cpu (%)	58.98	40.67	42.34	45.12
average ram (mb)	3,664	3,563	440	3,290

Original Implementation and FM-Index. The results of the two implementations are found in Table 2 for both the query where having substring search is an advantage and the query where it has no influence at all. The results for the "Johnny Depp" query speak for themselves: where the original implementation needs to iterate over all the actors, the FM-index can first find the correct actor and iterate over his movies, having more than 100 times fewer HTTP calls.

The results of the second query are also positive: although more memory is required to store the FM-index during execution, there is no loss in performance, both implementations have the same results.

Data Size. Besides those evaluations there is also a difference in storage required: the original HDT file is 6.4GB while the updated HDT file with FM-index is 8.2GB in size. The Elasticsearch index for the original HDT file is 52.7GB. FM-index obviously has a big advantage here due to the fact that it is embedded in the data itself while Elasticsearch has to duplicate the data.

7 Conclusions

In this paper we discuss how the Triple Pattern Fragments interface can be extended by adding a substring pattern matching feature to the server, either by an internal FM-index or external Elasticsearch index. We extended the FM-index implementation of HDT to also support case-insensitive searches and updated the TPF server interface to make use of this functionality.

Our evaluations show that the new interface greatly increases the performance for certain queries (validating Hypothesis 1) without harming unrelated query results (validating Hypothesis 2). Also, the server can execute these substring queries quite fast (validating Hypothesis 3). The FM-index also performed better than the external Elasticsearch index, but since the functionality of both systems is not identical—because of the extra features in Elasticsearch—this can not be used to conclude that one is strictly better than the other.

The type of SPARQL queries that benefit from the substring interface are obviously those with a text-centric component. One possible application are *auto-completion* widgets, in which terms are suggested to an end user based on preliminary text input. Other applications include so-called *reconciliation* tasks, in which Linked Data identifiers are sought for a large corpus of text strings [23]. Such tasks currently rely on public SPARQL endpoints, on which they need to launch relatively costly queries, yet they can be implemented with the substring interface feature at lower cost and thus improved reliability.

In the future we would like to improve these results by using the FEMTO system described in Section 2.2 to also allow regular expressions besides simple pattern matching. It would also be interesting to see what the effect would be when the updated interface was used in the improved query algorithm described by Van Herwegen et al.[22] to further enhance query results. Besides partially replacing regular expression filters, we would also like to use the FM-index to support other filters, such as date ranges by finding all literals that match a specific date template. This would follow the TPF principle of solving complex problems by using simple building blocks.

Another interesting direction to explore is the combination of the substring interface feature with other interfaces than TPF. As we explained in Section 4, clients can dynamically discover support for substring matching, analogous to how they can dynamically discover support for TPF. Suppose a dataset is offered through a SPARQL interface, then the TPF interface can still be helpful. After all, support for substring search in SPARQL engines is not always available with high performance (even though derived features, such as prefix search or `bif:contains` might be supported). Therefore, SPARQL queries that rely on text filters might be evaluated by decomposing them on the client side into a substring search (evaluated on the substring interface) and a regular SPARQL query (evaluated on the SPARQL endpoint) in which the filter has been replaced by a list of matching substring values. An alternative is that the SPARQL endpoint is configured as a client of the substring interface; i.e., that it has access to this interface to evaluate complex text filters.

References

1. Arias Gallego, M., Corcho, O., Fernández, J.D., Martínez-Prieto, M.A., Suárez-Figueroa, M.C.: Compressing semantic metadata for efficient multimedia retrieval. In: Bielza, C., Salmerón, A., Alonso-Betanzos, A., Hidalgo, J.I., Martínez, L., Troncoso, A., Corchado, E., Corchado, J.M. (eds.) CAEPIA 2013. LNCS, vol. 8109, pp. 12–21. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-40643-0_2
2. Brisaboa, N.R., Cánovas, R., Claude, F., Martínez-Prieto, M.A., Navarro, G.: Compressed string dictionaries. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 136–147. Springer, Heidelberg (2011)
3. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: SPARQL web-querying infrastructure: ready for action? In: Proceedings of the 12th International Semantic Web Conference, November 2013. http://link.springer.com/chapter/10.1007/978-3-642-41338-4_18
4. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. Rep. SRC-RR-124, Digital Equipment Corporation (1994)
5. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: Pellegrini, T., Auer, S., Tochtermann, K., Schaffert, S. (eds.) Networked Knowledge - Networked Media. SCI, vol. 221, pp. 7–24. Springer, Heidelberg (2009)
6. Ermilov, I., Martin, M., Lehmann, J., Auer, S.: Linked open data statistics: collection and exploitation. In: Klinov, P., Mouromtsev, D. (eds.) KESW 2013. CCIS, vol. 394, pp. 242–249. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-41360-5_19
7. Feigenbaum, L., Williams, G.T., Clark, K.G., Torres, E.: SPARQL 1.1 protocol. Recommendation, World Wide Web Consortium, March 2013. http://www.w3.org/TR/sparql11-protocol/
8. Ferguson, M.P.: FEMTO: fast search of large sequence collections. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 208–219. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-31265-6_17
9. 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, March 2013. http://dx.doi.org/10.1016/j.websem.2013.01.002
10. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, pp. 390–398 (2000)
11. Freitas, A., Curry, E., O'Riain, S.: A distributional approach for terminological semantic search on the linked data web. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp. 384–391 (2012)
12. Gormley, C., Tong, Z.: Elasticsearch: The Definitive Guide. O'Reilly (2014)
13. Harris, S., Seaborne, A.: SPARQL 1.1 query language. Recommendation, World Wide Web Consortium, March 2013. http://www.w3.org/TR/sparql11-query/
14. 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
15. Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics **25**(15), 1966–1967 (2009)

16. Martínez-Prieto, M.A., Fernández, J.D., Cánovas, R.: Querying rdf dictionaries in compressed space. ACM SIGAPP Applied Computing Review **12**(2), 64–77 (2012)
17. Minack, E., Sauermann, L., Grimnes, G., Fluit, C., Broekstra, J.: The sesame lucenesail: RDF queries with full-text search. NEPOMUK Consortium, Technical Report 1 (2008)
18. Minack, E., Siberski, W., Nejdl, W.: Benchmarking fulltext search performance of RDF stores. In: Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 81–95. Springer, Heidelberg (2009)
19. Nelson, M.: Data compression with the Burrows-Wheeler transform. Dr. Dobb's Journal **9**, 46–50 (1996)
20. Rietveld, L., Verborgh, R., Beek, W., Vander Sande, M., Schlobach, S.: Linked data-as-a-service: the semantic web redeployed. In: Gandon, F., Sabou, M., Sack, H., d'Amato, C., Cudré-Mauroux, P., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9088, pp. 471–487. Springer, Heidelberg (2015)
21. Sadakane, K.: A modified Burrows-Wheeler transformation for case-insensitive search with application to suffix array compression. In: Proceedings of the Data Compression Conference, p. 548 (1999)
22. Van Herwegen, J., Verborgh, R., Mannens, E., Van de Walle, R.: Query execution optimization for clients of triple pattern fragments. In: Gandon, F., Sabou, M., Sack, H., d'Amato, C., Cudré-Mauroux, P., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9088, pp. 302–318. Springer, Heidelberg (2015)
23. van Hooland, S., Verborgh, R., De Wilde, M., Hercher, J., Mannens, E., Van de Walle, R.: Evaluating the success of vocabulary reconciliation for cultural heritage collections. Journal of the American Society for Information Science and Technology **64**(3), 464–479 (2013). http://freeyourmetadata.org/publications/freeyourmetadata.pdf
24. Verborgh, R.: Triple Pattern Fragments. Unofficial draft, Hydra w3c Community Group. http://www.hydra-cg.com/spec/latest/triple-pattern-fragments/
25. Verborgh, R., et al.: Querying datasets on the web with high availability. In: Mika, P., et al. (eds.) The Semantic Web – ISWC 2014. LNCS, vol. 8796, pp. 180–196. Springer, Heidelberg (2014). http://linkeddatafragments.org/publications/iswc2014.pdf
26. Verborgh, R., Mannens, E., Van de Walle, R.: Initial usage analysis of DBpedia's triple pattern fragments. In: Proceedings of the 5th Workshop on Usage Analysis and the Web of Data, June 2015
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, April 2014. http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_04.pdf