Reducing the Network Load of Triple Pattern Fragments by Supporting Bind Joins

Olaf Hartig[1] and Carlos Buil-Aranda[2]

[1] Department of Computer and Information Science (IDA), Linköping University, Sweden
olaf.hartig@liu.se
[2] Informatics Department, Universidad Técnica Federico Santa María, Chile
cbuil@inf.utfsm.cl

1 Introduction

The recently proposed Triple Pattern Fragment (TPF) interface aims at increasing the availability of Web-queryable RDF datasets [3]. To this end, the TPF proposal trades off an increased client-side query processing effort for a significant reduction of server load. However, an additional aspect of this trade-off is a very high network load. To mitigate this drawback we propose to extend the interface by augmenting TPF requests with an optional VALUES clause as introduced in SPARQL 1.1. This extension enables clients to attach intermediate results to triple pattern requests. The response to such a request is expected to contain triples from the underlying dataset that do not only match the given triple pattern, but that also are guaranteed to contribute in a join with the given intermediate result. Hence, given such an extended interface—which we call *Bindings-Restricted Triple Pattern Fragments* (*brTPF*)—the execution of joins can be distributed between client and server by using the well-known bind join strategy [1].

In an ongoing research project we study the trade-offs of the brTPF interface and compare it to the pure TPF interface. With a poster in the conference we aim to present initial results of this research. In particular, we would like to present a series of experiments showing that distributed, bind-join-based query executions using the brTPF interface reduce the network load drastically. In the remainder of this extended abstract, we describe these experiments, discuss their results, and elaborate on the open research questions that will drive the next steps of our ongoing work regarding brTPF. All digital artifacts for our experiments are available online at http://olafhartig.de/brTPF-ISWC2016.

2 Experimental Setup

We begin by describing the metrics and the setup of our experiments. The goal of these experiments is to compare TPF and brTPF in terms of the network load that the interfaces may cause when accessed by clients that execute SPARQL queries.

Metrics: For the comparison we focus on two metrics: First, the *number of requests* (#req) that a client sends to the server during the execution of a query. Since both the TPF interface and the brTPF interface split fragments into pages, the measurements for #req do not correspond to the number of fragments requested during query executions but to the number of pages requested for the fragments that the client choses to access. The second metric that we focus on is the *amount of data received* (dataRecv) by the client during query executions. We measure dataRecv in terms of the number of RDF triples contained in all fragment pages that the client receives during a query execution.

Evaluation Prototypes: As the *server* component in our experiments, we used an established Java servlet implementation of the TPF interface and extended it with the

functionality to also support brTPF. The actual implementation approach used for the latter is not relevant for the results that we present below. However, we emphasize that the brTPF server specifies an upper bound on the number of solution mappings that can be sent with any brTPF request. Hereafter, we refer to the upper bound as *maxM/R*.

As a pure *TPF client* implementation, we used a JavaScript implementation of the TPF-based query execution algorithm for SPARQL basic graph patterns (BGP) as proposed by Verborgh et al. [3]. This algorithm is based on iterators that are arranged in pipelines. Query results are computed recursively by executing the pipelines. Each of these pipelines is generated for a subquery obtained from a decomposition of the initial BGP. Each iterator executes one of these subqueries returning as well an estimation of the size of its response. The algorithm uses this estimation to adapt its execution dynamically so that subqueries with a smaller result are executed first.

For the *brTPF client* we implemented a simple bind-join-based query execution algorithm. In contrast to the comparably sophisticated adaptive algorithm used by the TPF client, the brTPF algorithm is kept deliberately straightforward. That is, this algorithm simply choses a fixed query execution plan upfront. This plan represents a left-deep join tree that is implemented using a fixed pipeline of iterators such that each of these iterators is responsible for a different triple pattern of the query. The join order is decided based on result cardinality estimates for every triple pattern of the query. During query execution, every iterator receives chunks of solution mappings from its predecessor. The size of these chunks corresponds to the value of maxM/R as specified by the brTPF server. Given such a chunk, the iterator issues a brTPF request consisting of the triple pattern that the iterator is responsible for and the solution mappings from the chunk. Upon arrival of the data for the requested brTPF, the iterator uses this data to generate chunks of solution mappings for the next iterator in the pipeline.

Benchmark: For the experiments we used the DBpedia 3.5.1 dataset and a sequence of 100 BGP queries that we generated for this dataset by using the FEASIBLE benchmark generator [2]. FEASIBLE generated these queries by mimicking features that were extracted from real user queries in the log files of the DBpedia SPARQL endpoint [2].

Experimental Environment: We conducted the experiments using a single-machine setup with a single client. That is, the combined TPF/brTPF Java servlet (with DBpedia) is deployed on the same machine on which the client implementation performs the sequence of query executions (using either the TPF algorithm or the brTPF algorithm).

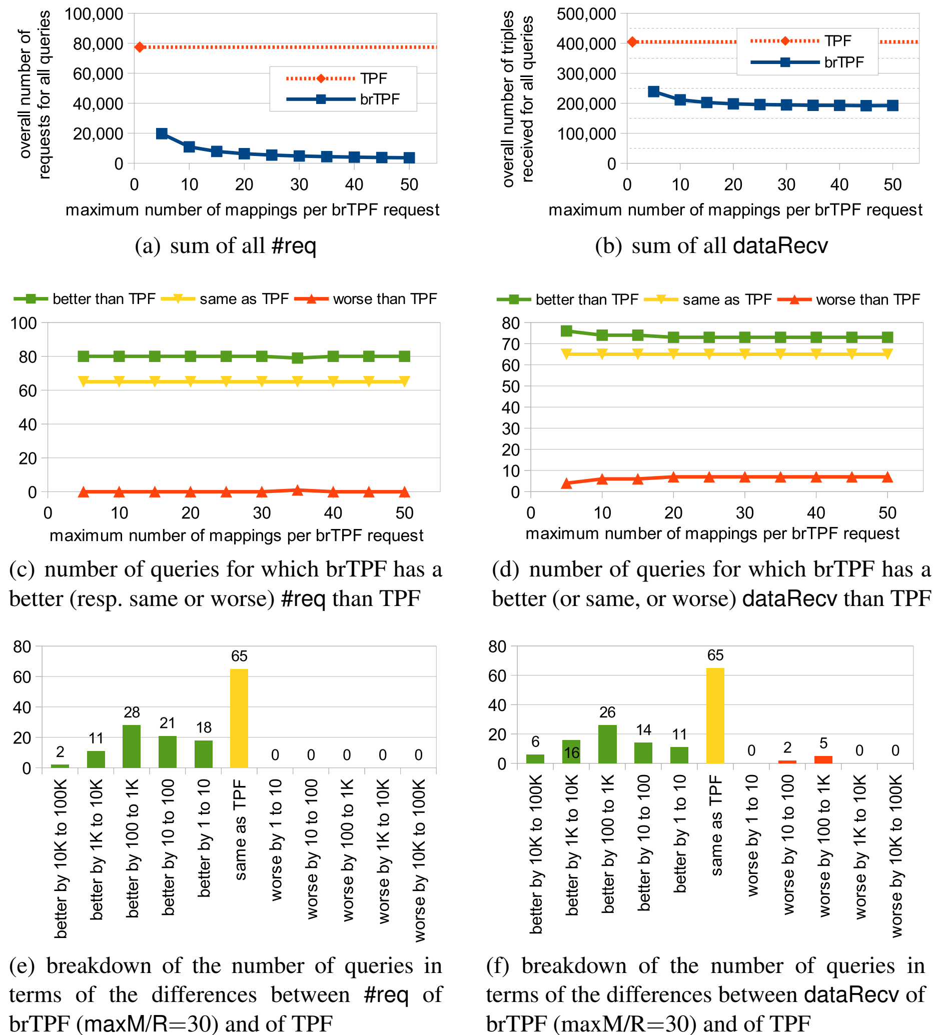
3 Experimental Results

For our first experiment we use a page size of 100 data triples per fragment page and execute the query sequence using the TPF client and the brTPF client, respectively. For the latter we repeat the execution of the query sequence using a maxM/R of 5, 10, ... , 45, and 50. The charts in Figure 1(a) and 1(b) provide an aggregated view on the resulting measurements. In particular, Figure 1(a) illustrates the overall #req summed up for each client over the whole sequence of queries, respectively; similarly, Figure 1(b) illustrates the sums of the dataRecv measurements obtained for all queries, respectively.

Regarding brTPF, we observe that the overall #req decreases with an increasing value for maxM/R, and so does the overall dataRecv. While for #req this observation is not surprising (if the fraction of any large intermediate result that can be sent with each request is smaller, the brTPF client has to send more such requests), for dataRecv we explain the observation by the fact that each fragment page contains not only data

triples but also additional metadata triples [3]. Therefore, if the number of fragment pages requested and received is greater (as is the case for a smaller maxM/R), then so is the overall number of these additional triples that have to be received with each page.

By now comparing the behavior of TPF vs. brTPF in the charts in Figures 1(a)–1(b), we notice that for both the overall #req and the overall dataRecv, brTPF achieves significantly smaller values. At this point, we have to recall that these charts only show aggregated measurements. Hence, it might still be possible that the vastly superior behavior of brTPF as shown in these charts is actually only due to a small number of outliers. We can verify that this is not the case by drilling into the measurements: For the different values of maxM/R, Figure 1(c) illustrates the number of queries for which brTPF has a smaller (i.e., better) or greater (i.e., worse) #req than TPF. Figure 1(d) presents a corresponding comparison for dataRecv. In Figures 1(e)–1(f), we drill in even deeper for maxM/R=30 (corresponding charts for the other values of maxM/R look similar) and



(a) sum of all #req

(b) sum of all dataRecv

(c) number of queries for which brTPF has a better (resp. same or worse) #req than TPF

(d) number of queries for which brTPF has a better (or same, or worse) dataRecv than TPF

(e) breakdown of the number of queries in terms of the differences between #req of brTPF (maxM/R=30) and of TPF

(f) breakdown of the number of queries in terms of the differences between dataRecv of brTPF (maxM/R=30) and of TPF

Figure 1. Measurements of network-related metrics using FEASIBLE queries over DBpedia.

report the number of queries for which the difference between the #req (resp. dataRecv) of brTPF vs. TPF is between 100K to 10K, between 10K to 1K, etc. These charts show that, in terms of both #req and dataRecv, brTPF is not only better than TPF in an impressively high number of cases, but for a large majority of these cases in which brTPF is better, the differences are significant.

To investigate whether these results are different for a different page size we conducted another experiment in which we varied the page size (number of data triples per fragment page). That is, with both the TPF client and the brTPF client, we repeated the execution of the query sequence for different page sizes (up to 2000). Due to space limitations, we do not include charts for this experiment in this paper. However, we highlight that the measurements obtained by this experiment show that, for both brTPF and TPF, the page size does not have any considerable impact on #req or on dataRecv. In other words, the relative differences between brTPF and TPF as identified by the first experiment are independent of the page size (and so are the relative differences between the different maxM/R configurations for brTPF). Hence, our main conclusion from these experiments is that, *independent of the page size (and the value of maxM/R), brTPF typically achieves a significantly smaller #req and dataRecv than TPF*.

4 Future Directions

While brTPF can be used as an alternative to TPF to achieve a drastic reduction of network load, this advantage does not come for free: In comparison to a TPF request, responding to a brTPF request imposes more server-side work, which might have a negative impact on the throughput of a brTPF server. On the other hand, when accessed by a brTPF-aware client, the number of requests that a brTPF server has to answer is significantly smaller than the throughput that a TPF server has to deliver (recall Figures 1(a), 1(c), and 1(e)). As a consequence, considering the whole client-server system, both approaches, brTPF and TPF, might achieve a comparable overall throughput (in terms of full SPARQL queries executed within a given time frame). However, this is speculation. Therefore, in the next step of our ongoing investigation of the trade-offs of TPF versus brTPF, we will focus on throughput-related properties. To this end, we will run large-scale experiments with an increasing number of concurrent clients.

An important aspect of considering the throughput of approaches such as brTPF and TPF is the degree to which the server load may be reduced by HTTP caching. We assume that TPF would benefit more from caching than brTPF because it seems more likely that different TPF-based query executions issue the same TPF requests than it is for different brTPF-based executions to issue some identical brTPF requests. We will conduct experiments to test this assumption and to identify how caching affects the performance of both approaches.

References

1. Haas, L.M., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing Queries Across Diverse Data Sources. In: Proc. of the 23rd Int. Conference on Very Large Data Bases (VLDB) (1997)
2. Saleem, M., Mehmood, Q., Ngomo, A.N.: FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In: Proc. of the 14th Int. Semantic Web Conf. (ISWC) (2015)
3. Verborgh, R., Vander Sande, M., Hartig, O., et al.: Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. Journal of Web Semantics 37–38, 184–206 (2016)