brTPF: Bindings-Restricted Triple Pattern Fragments

Olaf Hartig[1,2] and Carlos Buil-Aranda[3]

[1] Hasso Plattner Institute, University of Potsdam, Germany
[2] Department of Computer and Information Science (IDA), Linköping University, Sweden
olaf.hartig@liu.se

[3] Informatics Department, Universidad Técnica Federico Santa María, Chile
cbuil@inf.utfsm.cl

Abstract The Triple Pattern Fragment (TPF) interface is a recent proposal for reducing server load in Web-based approaches to execute SPARQL queries over public RDF datasets. The price for less overloaded servers is a higher client-side load as well as a dramatic increase in network load (in terms of both the number of HTTP requests and data transfer). In this paper, we propose a slightly extended interface that allows 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 (as in the case of TPF), but that are guaranteed to contribute in a join with the given intermediate result. Our hypothesis is that a distributed query execution using this extended interface can reduce the network load (in comparison to a pure TPF-based query execution) without increasing the server load significantly. Our main contribution in this paper is twofold: we empirically verify the hypothesis and provide an extensive experimental comparison of our proposal and TPF.

1 Introduction

Recent years have witnessed a large and constant growth in the amount of data that is structured based on the data model of the Resource Description Framework (RDF) [5] and made available on the Web through HTTP interfaces [3,13,17]. A prevalent (and standardized) type of such interfaces that provides query-based access to RDF data are SPARQL endpoints [6]; that is, Web services that accept queries written in the SPARQL query language [9]. While a SPARQL endpoint enables users to query its RDF dataset by using the full potential of SPARQL, providing such a comparably complex functionality presents a serious challenge (the evaluation problem of a core fragment of SPARQL has been shown to be PSPACE complete [15]). As a consequence, many public endpoints suffer from frequent downtime; for instance, by monitoring over 400 such endpoint for 27 months, Buil-Aranda et al. show that only 32.3% of the endpoints offer an availability of more than 99%, and 50.4% have an availability of less than 95% [4]. Furthermore, if many client applications start to access a SPARQL endpoint concurrently, then the performance of the endpoint (in terms of average query execution times and per-client query throughput) drops significantly [18].

To address these problems, Verborgh et al. recently proposed the Triple Pattern Fragment (TPF) interface [18,19]. This proposal restricts the type of queries supported by the server to single triple patterns. Then, to support arbitrary SPARQL queries over a dataset exposed by such a TPF server, a major part of the query processing effort has to be shifted to the clients. As a result, the server load is reduced and query execution times are more stable. However, everything comes at a cost and, thus, the price for the aforementioned benefits of the TPF approach is not only a higher client-side load but also a significant increase in network load. More precisely, to execute a given SPARQL query, a TPF-based client usually sends many more HTTP requests than a client that sends the whole query with a single request to a SPARQL endpoint. Additionally, the overall amount of data returned in response to these TPF requests is much greater than what a SPARQL endpoint returns (namely, just the query result).

To mitigate this drawback of the TPF approach (without losing the benefits) we propose a slightly extended interface that supports so-called *Bindings-Restricted Triple Pattern Fragments* (*brTPF*). That is, in addition to pure TPF requests, the brTPF interface allows clients to attach intermediate results to TPF requests. The response to such a brTPF request is expected to contain RDF triples from the underlying dataset that do not only match the given triple pattern (as in the case of TPF), but that are guaranteed to contribute in a join with the given intermediate result. Hence, given the brTPF interface, it becomes possible to distribute the execution of joins between client and server by using the well-known bind join strategy [8]. Our hypothesis is given as follows:

Hypothesis. *In comparison to pure TPF-based query executions, query executions that are based on the brTPF interface can reduce the network load without increasing the server load significantly.*

Our main contribution in this paper is twofold: First, we empirically verify the aforementioned hypothesis. Second, we also provide an extensive experimental comparison of the brTPF approach and the TPF approach. The queries that we focus on in this study are expressed using SPARQL basic graph patterns (BGPs). We chose this focus because we believe that achieving a comprehensive understanding of how TPF and brTPF behave for this fundamental fragment of SPARQL is essential before attempting to focus on more expressive fragments. All digital artifacts related to our study (e.g., software, test data, etc.) are available on the Web page for this paper.[4]

The remainder of the paper is organized as follows: Section 2 provides an overview of the related work on existing HTTP interfaces to access RDF data on the Web. Section 3 presents a formalization of the concepts used in this paper, Section 4 describes the implementation of the propotypes for the brTPF server and client, as well as a brief description of the TPF server and client implementations. Sections 5, 6 and 7 present the experiments that we have done to evaluate TPF and brTPF in terms of the network load, server load, and cache use, respectively. Finally, we conclude in Section 8.

2 Related Work

One of the most common ways for accessing RDF data on the Web is through HTTP interfaces. Verborgh et al. classified such interfaces along an axis as illustrated in Figure 1 [18]. In the following we describe the currently most common of these interfaces.

[4] http://olafhartig.de/brTPF-ODBASE2016

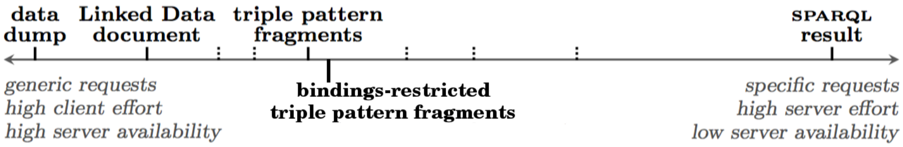


Figure 1. HTTP interfaces to RDF data (adapted from [18]).

2.1 SPARQL Endpoints

SPARQL endpoints are Web services that implement the SPARQL Protocol [6]. A SPARQL endpoint is usually an HTTP interface that accepts SPARQL queries (like Jena Fuseki[5]) and the query processing is done by the endpoint's triple store (e.g., Virtuoso[6], Jena TDB[7]). As it was shown in [4], public SPARQL endpoints offering interfaces with SPARQL query capabilities may suffer from low availability and poor performance. To address this problem, Triple Pattern Fragments (TPF) was proposed [18].

2.2 Triple Pattern Fragments

Verborgh et al. [18] proposed a novel HTTP interface to access RDF data which relies on the client processing power to execute the SPARQL queries while the SPARQL endpoint only provides the data for the triple patterns in the query. Relying on the client's capabilities to execute the actual operations on the data it is possible to maximize availability on the server side, since servers only perform operations that require minimal effort. The TPF client algorithm is based on a decomposition of the original query into triple patterns, which are ordered according to a selectivity estimation and executed recursively in a pipeline. The downside of this approach is a higher client-side load as well as an increase in network load (in terms of both the number of HTTP requests and data transfer). In order to solve this problem appeared some new approaches to query TPF servers. For improving the overall performance of the Triple Pattern Fragments original algorithm the authors in [11] extend the original TPF work by injecting every partial solution into the next subquery with which it shares variables. The main advantage of the two algorithms described in [18] and in [11] is that the remote query server does not perform any complex operations; it just scans the database for matching a single triple pattern while the client perform most of the query execution work.

Acosta et al. [1] improved the original TPF client algorithm by developing a query optimizer that generates new query plans using the result size metadata provided by the TPF server. That query optimizer focuses on reducing the intermediate amount of results and requests posed to the TPF server and it also implements an adaptive routing query engine which is able to dynamically adapt the query plan according to execution conditions. Again, this work only focuses in optimizing the client algorithm while the

[5] https://jena.apache.org/documentation/serving_data/

[6] http://virtuoso.openlinksw.com/

[7] https://jena.apache.org/documentation/tdb/

server remains unchanged, and even though the authors improve the TPF query execution considerably, we argue that there is room for improvement at server side.

There have been works that extended the TPF server instead of focusing on the client improvement. One of them is [12] in which the authors extended the original TPF controller to deal with substring filtering. The authors extended the server capabilities using an Elastic Search service, allowing the server to execute FILTER queries besides the usual triple patterns. Another server improvement is [16], in which the TPF metadata is extended by using using approximate membership functions (Bloom filters and Golomb-coded sets). The result of using these statistical techniques was that the amount of HTTP requests was reduced by 25%, barely increasing the server load. Even though the amount of HTTP requests was fewer, query times were not improved.

From a SPARQL query federation point of view, Montoya et al. [14] proposed a federation system that is aware of existing replicated TPF servers. Given a set of endpoints with having replicated TPFs, the system is able to identify which fragments will contribute to the final query result and create a query plan to efficiently solve the query. This work focused mainly on the client, which tries to find the TPF servers efficiently in order to minimize the number of tuples to be transferred from the server to the client.

2.3 Linked Data Querying

Hartig et al. [10] proposed a Linked Data traversal approach, in which the follow your nose concept is applied for following links between linked RDF datasets on the Web. The implementation consists first in a source selection process for next traversing the links obtained from the initial query to the selected data sources. The link traversal operator implements an asynchronous pipeline of iterators executing first the most selective iterator. The query engine is also able to adapt the execution to source availability by detecting whenever an HTTP server stops responding. This is one of RDF data access approaches that imposes less load on the server, however it is among the slowest.

3 Formal Definition of brTPF

In this section, we provide a formal definition of the brTPF interface. For this formalization we use the general formal framework for defining interfaces to RDF datasets as provided by Verborgh et al. [18]. For the following definitions we assume that the reader is familiar with the fundamental concepts of RDF [5] and SPARQL [9,15].

As the basis for formalizing any type of interface to RDF datasets Verborgh et al. introduce the notion of a *selector* that captures conditions for selecting sets of triples from datasets (cf. [18, Definition 1]). We define such a selector for brTPFs as follows.

Definition 1. *Given a triple pattern tp and a finite sequence of solution mappings Ω, the **bindings-restricted triple pattern selector** for tp and Ω is the selector $\mathrm{s}_{(tp,\Omega)}$ that,* *for any singleton set* $\{t\} \subseteq 2^{\mathcal{T}}$*, is defined by*

$$\mathrm{s}_{(tp,\Omega)}(\{t\}) = \begin{cases} \texttt{true} & \textit{if } \Omega \textit{ is empty and there exists a solution} \\ & \textit{mapping } \mu \textit{ such that } \mu[tp] = t, \\ \texttt{true} & \textit{if there exists a solution mapping } \mu \textit{ such that} \\ & \mu[tp] = t \textit{ and } \mu \textit{ is compatible with some } \mu' \textit{ in } \Omega, \\ \texttt{false} & \textit{else.} \end{cases}$$

Given the notion of a bindings-restricted triple pattern selector, we now define brTPFs.

Definition 2. *Given a positive integer* maxM/R *and a control* c *(as per [18, Definition 3]), a* c*-specific LDF collection* F *(cf. [18, Definition 6]) is called a* ***bindings restricted triple pattern fragment collection*** *for* maxM/R *if, for any possible triple pattern* tp *and any finite sequence* Ω *of at most* maxM/R *distinct solution mappings, there exists an LDF* $\langle u, s, \Gamma, M, C\rangle \in F$ *(as per [18, Definition 4]), referred to as a* ***bindings restricted triple pattern fragment (brTPF)****, that has the following three properties:*

1. *Selector* s *is the bindings-restricted triple pattern selector for* tp *and* Ω*;*
2. *There exists a (metadata) RDF triple* $\langle u, \texttt{void:triples}, cnt\rangle \in M$ *with* cnt *being an integer that represents an estimate of the cardinality of* Γ*; that is, if* $\Omega' = \{\mu \in [\![tp]\!]_G \mid \mu \sim \mu'$ *for some* μ' *in* $\Omega\}$*, then* cnt *has the following two properties:*
   (a) If $\Omega' = \emptyset$*, then* $cnt = 0$.
   (b) If $\Omega' \neq \emptyset$*, then* $cnt > 0$ *and* $\mathrm{abs}\big(|\Omega'| - cnt\big) \leq \epsilon$ *for some* F*-specific threshold* ϵ.
3. $c \in C$.

Observe that our definition assumes a positive integer maxM/R that presents a well-defined restriction on the number of distinct solution mappings that can be attached to any brTPF request supported by any specific brTPF interface.

4 Evaluation Prototypes

In this section, we describe the implementations that we used for our experiments.

4.1 Server Implementation

As a basis for the server, we used an established Java servlet implementation of the TPF interface[8] and extended it with the functionality to also support the brTPF interface. As a result, both interface implementations coexist within the Java servlet, which choses which of them it invokes depending on the HTTP GET request it receives: If the request contains a bindings-restricted triple pattern selector, then the brTPF implementation is used to generate the response; if the HTTP request just contains a TPF selector, the TPF implementation is used. Having both implementations in a single software component has the advantage that commonly used basic functionality (such as serializing RDF triples for a response) is shared and, thus, experimental results are not affected by potential differences in how efficient the implementation of such basic functionality is.

[8] https://github.com/LinkedDataFragments/Server.Java

TPF Server Implementation: The TPF server implementation is a Java servlet that accepts HTTP GET and POST requests. The servlet identifies the SPARQL triple pattern enclosed within such a request and evaluates the triple pattern using an internal storage component that contains the dataset exposed via the TPF interface. The particular storage component used by the servlet in our experiments is based on an RDF-HDT back-end [7], which manages a highly compressed, in-memory representation of RDF data. The TPF server returns to the client an HTTP response containing the RDF triples from the evaluation of the triple pattern divided in pages (whose size is configurable) and an estimation of the entire result set size. These result size estimations are obtained by just asking the RDF-HDT back-end how many matching triples there are for the given triple pattern. The TPF client uses such result set size estimation in its query execution algorithm.

brTPF Server Implementation: The brTPF server implementation extends the TPF implementation servlet by enabling it to process brTPF requests as follows. Given such a request, the servlet internally generates a stream of data triples for the requested (brTPF) fragment and processes this stream in the same way as the TPF implementation processes the stream of matching triples returned by the RDF-HDT backend. To generate the data triples for a brTPF request, the servlet iterates over the sequence of solution mappings that comes with the request. For each such mapping, the servlet applies this mapping to the triple pattern of the request by replacing variables in the triple pattern according to the mapping. From the resulting sequence of potentially more concrete triple patterns, the servlet removes all duplicates. Next, the servlet uses each remaining triple pattern, one after the other, to query the RDF-HDT backend. The resulting streams of matching triples are then concatenated into the desired stream of all data triples for the brTPF request.

4.2 TPF Client Implementation

For our experiments we use a TPF client that is implemented using Node.js and that uses the TPF-based query execution algorithm for SPARQL basic graph patterns (BGP) as proposed by Verborgh et al. [18]. 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 the subqueries with a smaller result are executed first. In this way it is possible to rapidly return a first subset of the query result.

4.3 brTPF Client Implementation

To develop a brTPF client we used the aforementioned Node.js-based TPF client and added a brTPF-based query execution algorithm to it. Hence, as for the servers, all basic functionality required for both client implementations is based on the same source code. This way, we ensure the comparability of our experimental results and allow for a fair comparison of both approaches. For the same purpose, the brTPF-based query execution algorithm that we developed is kept deliberately simple. In fact, we did not attempt to integrate any sophisticated query optimization technique such as the adaptive, intermediate result based approach to generate subplans at query execution runtime as used by the TPF client. Instead, our brTPF 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 intermediate result cardinality estimates for every triple pattern of the query. These estimates can be obtained from the server by requesting the first TPF page for each of the triple patterns.

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 interface. 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.

5 Experimental Comparison of Network Load

Our first group of experiments focuses on comparing TPF and brTPF in terms of the network load that the interfaces may cause when accessed by clients that execute SPARQL queries. In this section, we first introduce the metrics and the experimental setup that we use for these experiments, and, thereafter, we present the results of the experiments.

5.1 Metrics

For this 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. Recall that both the TPF interface and the brTPF interface split fragments into pages. Therefore, the measurements for #req do not correspond to the number of fragments (TPF or brTPF) requested during query executions but to the number of pages requested for the fragments that the client choses to access. Notice furthermore that such requests do not necessarily have to reach the server if there exists an HTTP cache in the network between the client and the server. Nonetheless, unless the cache is located directly in front of the client (in which case it may not be very effective), the requests are sent into the network.

The second network-related metric that we focus on is the *amount of data received* (dataRecv) by the client during query executions. We measure this metric in terms of the number of RDF triples contained in all fragment pages that the client receives during a query execution. Observe that this metric is independent of whether the data comes directly from the server or from an HTTP cache that acts as a proxy server.

5.2 Setup

For this group of experiments we use a single-machine setup with a single client. That is, the combined TPF/brTPF Java servlet is deployed (using Jetty 9.2.5) on the same machine on which the client implementation performs the query executions (using either the TPF algorithm or the brTPF algorithm). This machine runs the Ubuntu 12.04.5 LTS operating system with Oracle Java 1.8.0_92 and Node.js 0.10.37, and it is equipped with an Intel Core i7-2620M CPU (2.7GHz) and 8 GB of main memory. In all of our experiments, every query execution is performed by using a separate operating system process (that is, we stop and restart the client software between any two executions).

As a basis for the experiments we used the 10M triples dataset provided on the project page[9] of the Waterloo SPARQL Diversity Test Suite (WatDiv) [2], and we used a sequence of 145 BGP queries that we selected uniformly at random from the WatDiv stress test query workload[10]. This workload has been shown to be a challenging benchmark that is very diverse in terms of various structural and data-driven features [2].

5.3 Results

For our first experiment we use a page size of 100 data triples per fragment page (which is the default configuration of the TPF server implementation that we use as a basis for our evaluation) 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 each of the following values for maxM/R: 5, 10, 15, . . . , 45, and 50.[11] The charts in Figure 2(a) and 2(b) provide an aggregated view on the resulting measurements for WatDiv. In particular, Figure 2(a) illustrates the overall #req summed up for each client over the whole sequence of queries, respectively; similarly, Figure 2(b) illustrates the sums of the dataRecv measurements obtained for all queries, respectively.

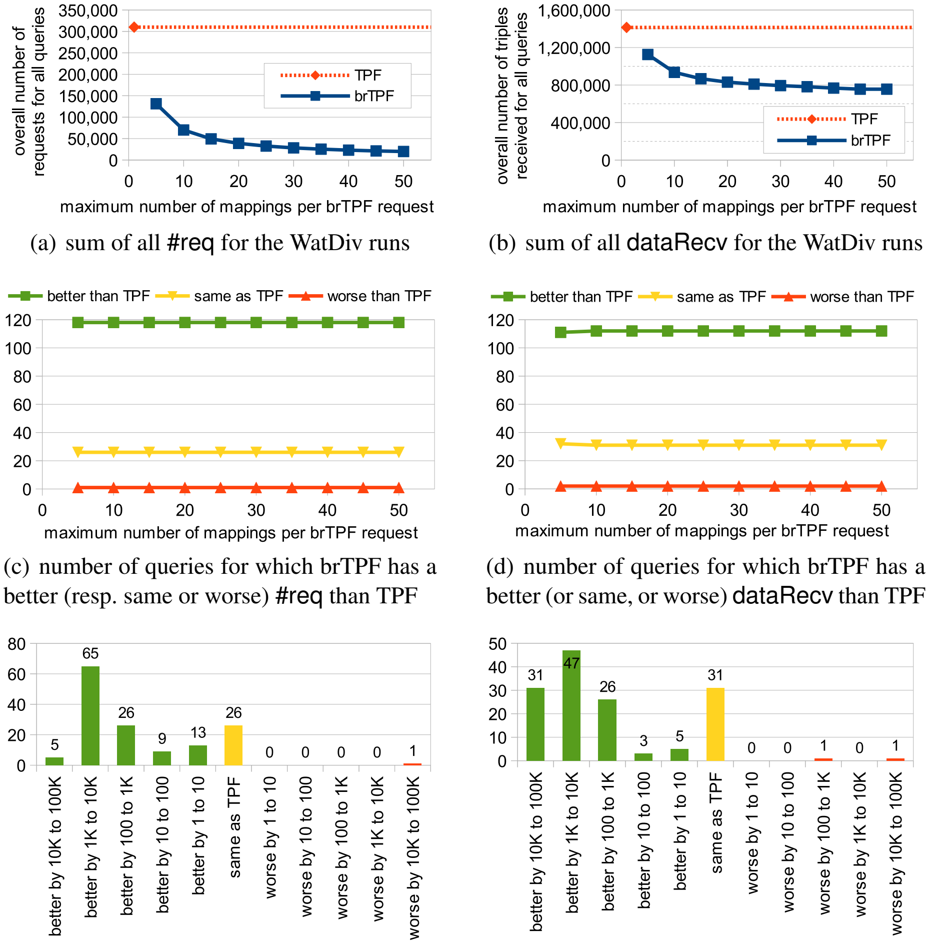
Regarding these measurements, we make the following observations. By first focusing on brTPF only, we observe that the overall #req decreases with an increasing value for maxM/R (from about 131K for maxM/R=5 to about 20K for maxM/R=50), and so does the overall dataRecv (from about 1,126K for maxM/R=5 to about 756K for maxM/R=50). 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 a greater number of such requests), for dataRecv we explain the observation as follows: Each fragment page contains not only data triples but also additional metadata triples that refer to the next and the previous page, describe the controls of the LDF interface, etc. Therefore, when the number of fragment pages requested and received is greater (as is the case for smaller maxM/R; cf. Figure 2(a)), then so is the overall number of these additional triples that have to be received with each fragment page.

By now comparing the behavior of TPF vs. brTPF in the charts in Figures 2(a)–2(b), we notice that for both the overall #req and the overall dataRecv, brTPF achieves significantly smaller values. More specifically, regarding dataRecv, brTPF achieves between 53.5% (maxM/R=50) and 79.6% (maxM/R=5) of the overall dataRecv of TPF (which is about 1,414K), and for #req, it even goes down to 6.5% (maxM/R=50) of the overall #req of TPF (310K). 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

[9] http://dsg.uwaterloo.ca/watdiv/

[10] https://cs.uwaterloo.ca/~galuc/watdiv/stress-workloads.tar.gz

[11] We do not increase maxM/R beyond a value of 50 because the client prototype uses the HTTP method GET, which causes problems for our server if maxM/R>50; that is, in preliminary tests with greater values we observed server responses with status code 414 (URI Too Long).



(a) sum of all #req for the WatDiv runs

(b) sum of all dataRecv for the WatDiv runs

(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 2. Measurements of network-related metrics using WatDiv.

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 2(c) illustrates the number of queries for which brTPF has a smaller (i.e., better) or greater (i.e., worse) #req than TPF. Figure 2(d) presents a corresponding comparison for dataRecv. In Figures 2(e)–2(f), we drill in even deeper for maxM/R=30 (corresponding charts for the other values of maxM/R look very similar) and 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 (using maxM/R=15 and maxM/R=30 as exemplars), we repeated the execution of the query sequences for each of the following page sizes: 100, 250, 500, 1000, and 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 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*.

6 Experimental Comparison of Server Performance under Load

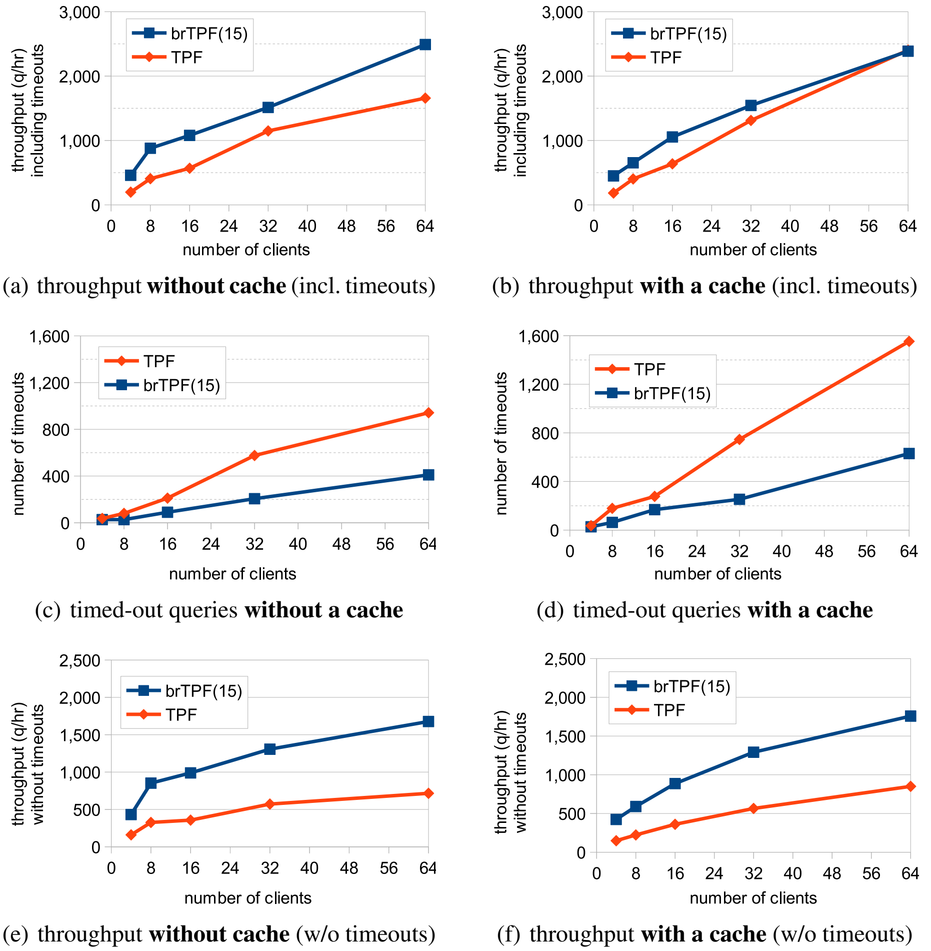
With our second group of experiments we aim to compare the behavior of TPF and brTPF when multiple clients access a server concurrently, and we aim to analyze how the two approaches scale to an increasing number of concurrent clients. To this end, we deliberately ignore the possibility of having an HTTP cache that may reduce the server load (we consider caching in our third group of experiments as presented in Section 7).

6.1 Setup

For our multi-client experiments we use a cluster of machines that are connected via a 10 Gigabit Ethernet network. The cluster includes 17 identical machines, each of which has an Intel Core i7 processor with 4 cores at 2.6Ghz and 8 GB of main memory. The operating system running on all machines is an Ubuntu 14.04 LTS.

One of these machines we used as the server and deployed the combined TPF/brTPF Java servlet on an Apache Tomcat 8 application server running with Java 1.8. The page size in this setup we fix to 100 data triples per fragment page.

The other 16 machines we used to simulate clients. For this purpose, we configured each of these machines to run on each of its 4 cores a thread with a single brTPF/TPF client, which gives us a total of up to 64 brTPF/TPF clients that we used for executing different WatDiv query sequences in parallel. To this end, we used the aforementioned WatDiv stress test query workload which consists of a set of 12,400 different queries; we split this set into 64 distinct sets, and distributed these sets over our experiment cluster such that each of the 64 CPU cores had available a total of 193 queries that they always executed as a sequence in the same order. We also configured each client to terminate any query execution that did not complete within 5 minutes. These query executions were recorded in our experiments as 'timed out.' Once a client stopped a query execution after 5 minutes, the client starts executing the next query from its sequence.



(a) throughput **without cache** (incl. timeouts)

(b) throughput **with a cache** (incl. timeouts)

(c) timed-out queries **without a cache**

(d) timed-out queries **with a cache**

(e) throughput **without cache** (w/o timeouts)

(f) throughput **with a cache** (w/o timeouts)

Figure 3. Measurements of server load in the multi-client setup using WatDiv.

6.2 Metrics

The metric that we consider to study the performance of the approaches under load is *query throughput* (throughput), which we measure in terms of the overall number of queries that all concurrent clients manage to execute within one hour. We also measured the average execution times, presenting it in the Appendix.

6.3 Results

The three charts on the left hand side of Figure 3 illustrate the measurements obtained by the experiment (ignore the figures on the right hand side for the moment).

Before we focus on the throughput values measured, we need to mention that we observed a varying, but often very high number of timed out queries across all runs.

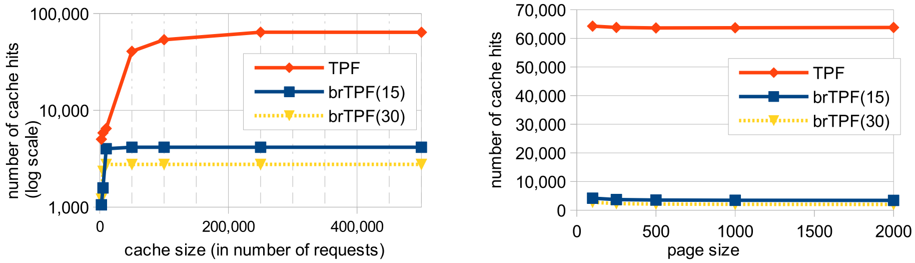
More specifically, as illustrated in Figure 3(c), for both TPF and brTPF, the number of timeouts increases with an increasing number of clients; and in all cases, the TPF clients run into a substantially higher number of timeouts than the brTPF clients. We identified the following cause of this behavior: Both client implementations asynchronously issue multiple HTTP requests in parallel. The TPF server has comparably less work to do to answer each of these multiple HTTP requests, returning the data as fast as possible. However both clients at this stage still have to do some data processing to join intermediate results. In the case of TPF, this client-side processing is significantly more work due to the larger amounts of data that the TPF client receives from the TPF server. As a result of this extra work that the TPF client has to do, there are more query executions that exceed the 5 mins timeout threshold before they complete.

Given the high amount of timeouts, it is important to report not only the throughput of all query execution attempts, including the ones the timed out, but also the throughput in terms of query executions that terminated normally after computing their complete query result. Therefore, Figure 3(a) illustrates the former and Figure 3(e) the latter. In these charts we make two main observations: First, the brTPF approach achieves a greater throughput than TPF, and, second, both approaches are able to achieve higher throughput values for an increasing number of clients. However, regarding the latter we also observe that the brTPF approach scales better if we look at the throughput of completed query executions, whereas, by comparing only the throughput in terms of all query execution attempts, both approaches seem to exhibit a similar scaling behavior.

We also tried to conduct a comparison between brTPF and a Virtuoso-based SPARQL endpoint in our experimental setup. To this end, we installed the latest version of the Open Source edition of Virtuoso (v.7.2.4) using the same configuration as for the brTPF and TPF servers (data and server). For the clients we developed a simple Python script that submits the sequence of WatDiv queries of the respective client using the SPARQL endpoint REST API, one query after another. We also configured the Virtuoso server with a result size limit of 1,000,000 results per query so Virtuoso was able to to return complete results to the Watdiv queries. However Virtuoso stopped its execution when there were 16 clients concurrently accessing the server. To understand such bad performance we configured Virtuoso to return partial results (i.e. a maximum result size of 10,000 results). This time the Virtuoso server performed faster finishing the query execution for 16 clients, but returning incomplete answers to the Watdiv queries, unlike brTPF and TPF. Thus, we decided not to include Virtuoso in our experimental evaluation since the query result sizes are not comparable.

7 Experimental Comparison of Server Performance with Caching

For our analysis in the previous section we ignore the possibility of HTTP caches. We recall that such caches are designed to reduce the load of Web servers by serving requests that are identical to earlier requests (instead of requiring the server to recompute the response for such identical requests over and over again). Therefore, our previous setup without such a cache can be conceived of as some kind of a worst-case scenario for both TPF and brTPF, and it might be more worse for one than it is for the other depending on the respective likelihood for observing identical requests from different



(a) different cache sizes (for page size of 100)

(b) different page sizes (unlimited cache)

Figure 4. Comparison of the cacheability of TPF requests vs. brTPF requests (using WatDiv).

query executions. In fact, it seems reasonable to assume that this likelihood is higher for TPF than it is for brTPF. More precisely, it seems more likely that different TPF-based query executions request the same triple pattern than it is for different brTPF-based executions to request an identical pair of the same triple pattern and the same sequence of solution mappings. To verify this assumption and to identify how caching affects the performance of both approaches we conducted another set of experiments. In this section, we describe these experiments and present their results.

7.1 Cache Hit Potential

To systematically study and to compare the potential for TPF requests and for brTPF requests to be served from a cache we use the same single-machine setup as used for the network-related experiments (cf. Section 5.2). As a metric for this analysis, we use the *number of cache hits* (#hits) as could be achieved by a possible HTTP cache when executing the test sequence of WatDiv queries. We instrumented our combined TPF/brTPF server implementation to measure this number assuming either an unlimited cache or a cache whose size is limited to a given number of distinct requests (using LRU as replacement policy). For our analysis we first used the latter and varied the cache size from 2.5K, 5K, 10K, 50K, 100K, 250K, to 500K.

The chart in Figure 4(a) illustrates our measurements for these different cache sizes when executing the WatDiv query sequence with either the TPF client or the brTPF client (using maxM/R=15 and maxM/R=30 as exemplars, and a page size of 100). First and foremost, we observe that TPF always achieves a significantly higher #hits than brTPF (note that the y-axis is log scale). This observation verifies the aforementioned assumption that the likelihood for observing identical requests from different query executions is higher for TPF than it is for brTPF. By focusing on brTPF, we notice that a smaller value for maxM/R results in a higher #hits. More precisely, the #hits for maxM/R=15 in this experiment is always about 150% of the #hits for maxM/R=30, which is not surprising given that brTPF requests with a greater number of solution mappings attached to them are more specialized than brTPF requests with a smaller number of solution mappings. As a final noteworthy observation regarding the measurements in Figure 4(a) we mention that the curves flatten out completely at some point and this point is different for each curve. The respective cache size at each of these points correlates with the overall number of requests issued during the respective WatDiv run (see the #req as reported for these runs in Figure 2(a)). Unsurprisingly, for cache sizes that are large enough to cover all distinct requests issued during such a run, the #hits is equivalent to the #hits as achieved by an unlimited cache.

In a second experiment we assumed an unlimited cache and executed the query sequences with different page sizes to investigate whether the cache hit potential of both TPF and brTPF is affected by the page size. Our measurements, as reported in Figure 4(b), show that the page size has no impact in the #hits. Therefore, from these experiments we conclude that, *independent of the page size and the value of maxM/R, TPF has a significantly higher potential to benefit from HTTP caches than brTPF*.

7.2 Impact of Caching on Performance

After verifying that TPF is more likely to benefit from an HTTP cache than brTPF, the obvious question that arises is whether the use of a cache allows TPF to gain a measurable advantage in performance. To answer this question we extended our multi-machine setup (cf. Section 6.1) with an additional machine on which we run an Nginx proxy server (1.4.6) as an HTTP cache that is located between the client machines and the machine with the TPF/brTPF server. This additional machine has an Intel Core i7 CPU with 8 cores and 16 GB of main memory, and it also runs an Ubuntu 14.04 LTS operating system.

Given this extended setup, we repeated the same set of multi-client executions as in the corresponding experiment without the cache (cf. Section 6). The three charts on the right hand side of Figure 3 illustrate the resulting measurements for the number of timeouts and the throughput (with and without queries whose execution timed out).

As a first observation, by comparing Figures 3(c) and 3(d), we note that the number of timeouts for TPF has increased substantially in comparison to the experiment without the cache. Our explanation of such an increased number of timed out queries is similar to what we saw in Section 6.3, however more exacerbated. That is, for many of the additional query execution attempts, the amount of data that the TPF clients receive results in client-side processing work that exceeds the 5 minutes timeout threshold. Figures 3(a) and 3(b) show the query throughput by brTPF and TPF when not having a cache server configured and in the scenario in which such cache server is present. By looking at Figure 3(b) it is possible to see how the TPF query throughput is almost the same than brTPF's query throughput when 64 clients access the servers concurrently. This high TPF throughput is due to the large amount of timed out queries as Figure 3(d) shows and as Figures 3(e) and 3(f) confirm. By comparing Figures 3(e) and 3(f), we observe that, in the case of a high number of concurrent clients (i.e., the 64-clients runs in our experiments), the performance *of both approaches* benefits from the cache, whereas for smaller number of concurrent clients we do not observe any significant impact for either approach. To explain the latter we revisit the measurements of our single-machine experiments: Given the #hits achieved by the WatDiv runs discussed in the previous section, cf. Figure 4(b), and the corresponding values for the overall number of requests during these runs, cf. Figure 2(a), we can compute hit rates. For TPF we obtain a hit rate of about 20.7% and for brTPF it is about 10.7% for maxM/R=15 and 11.8% maxM/R=30.[12] It appears that these hit rates are too small to allow for the cache to take significant load from the server in the case of a smaller number of concurrent clients. Only when the overall system becomes more busy with a higher number of concurrent clients, the availability of the cache reduces the server load and the throughput increases.

However, the perhaps most surprising finding is that TPF cannot benefit enough from the cache to gain an advantage over brTPF. We explain this finding by the network load that is significantly higher for TPF (as shown in Section 5). In particular, the increased amounts of data that need to be transferred and processed prove to be the primary weakness of TPF, even if some of the data comes from the cache.

In conclusion, this experiment shows that *for both approaches, TPF and brTPF, caching helps to increase the overall performance in a multi-client setting (in particular for greater numbers of clients), but it does not help TPF to outperform brTPF.*

8 Conclusions

In this paper we present an interface to access RDF datasets that slightly extends the Triple Pattern Fragments interface. Our extended interface, which we call Bindings Restricted Triple Pattern Fragments (brTPF), allows clients to attach intermediate results to triple pattern requests. By an extensive evaluation of brTPF and TPF we obtain the following results:

- Our main conclusion from the experimental comparison of network load is that, independent of the page size (and the value of maxM/R), brTPF typically achieves a significantly smaller #req and dataRecv than TPF.
- From the experimental comparison of server performance under load our conclusions are twofold: first, the brTPF approach achieves a greater throughput than TPF, and, second, both approaches are able to achieve higher throughput values for an increasing number of clients. Regarding the latter we also observe that the brTPF approach scales better if we look at the throughput in terms of completed query executions, whereas, by comparing only the throughput in terms of all query execution attempts, both approaches exhibit a similar scaling behavior.
- From the experimental comparison of server performance with cache the experiment shows that for both approaches, TPF and brTPF, caching helps to increase the overall performance in a multi-client setting (in particular for greater numbers of clients), but it does not help TPF to outperform brTPF.

In this paper we improved the idea of TPFs by significantly reducing the amount of HTTP requests and query processing times. We did this by keeping a simple client to show such an improvement, however we believe that there is more room to improve the overall query execution performance by exploring different combinations of interfaces, controls and selectors in the overall client and server configuration.

[12] Although the #hits is higher for maxM/R=15 than for maxM/R=30 (cf. Section 7.1), the former achieves a smaller hit rate because it has a higher #req (as we observe in Section 5.3).

Acknowledgements

Olaf Hartig's work has been funded partially by the German Government, Federal Ministry of Education and Research under the project number 03WKCJ4D. Carlos Buil-Aranda's work has been supported by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004.

References

1. Acosta, M., Vidal, M.: Networks of linked data eddies: An adaptive web query processing engine for RDF data. In: The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I. pp. 111–127 (2015)
2. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: Proceedings of the 13th International Semantic Web Conference (ISWC) (2014)
3. Bizer, C., Eckert, K., Meusel, R., Mühleisen, H., Schuhmacher, M., Völker, J.: Deployment of RDFa, Microdata, and Microformats on the Web – A Quantitative Analysis. In: Proceedings of the 12th International Semantic Web Conference (ISWC) (2013)
4. 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 (ISWC) (2013)
5. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (Feb 2014)
6. Feigenbaum, L., Williams, G.T.: SPARQL protocol for RDF. W3C Recommendation (2013)
7. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Web Semantics: Science, Services and Agents on the World Wide Web 19, 22–41 (2013)
8. Haas, L.M., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing Queries Across Diverse Data Sources. In: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB) (1997)
9. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation (2013)
10. Hartig, O., Bizer, C., Freytag, J.: Executing SPARQL queries over the Web of Linked Data. In: Proceedings of the 8th International Semantic Web Conference. pp. 293–309. ISWC'09, Springer-Verlag, Berlin, Heidelberg (2009)
11. Herwegen, J.V., Verborgh, R., Mannens, E., de Walle, R.V.: Query execution optimization for clients of triple pattern fragments. In: The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings. pp. 302–318 (2015)
12. Herwegen, J.V., Vocht, L.D., Verborgh, R., Mannens, E., de Walle, R.V.: Substring filtering for low-cost linked data interfaces. In: The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I. pp. 128–143 (2015)
13. Mika, P., Potter, T.: Metadata Statistics for a Large Web Corpus. In: Proceedings of the 5th Linked Data on the Web Workshop (LDOW) (2012)
14. Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.: Federated SPARQL queries processing with replicated fragments. In: The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I. pp. 36–51 (2015)

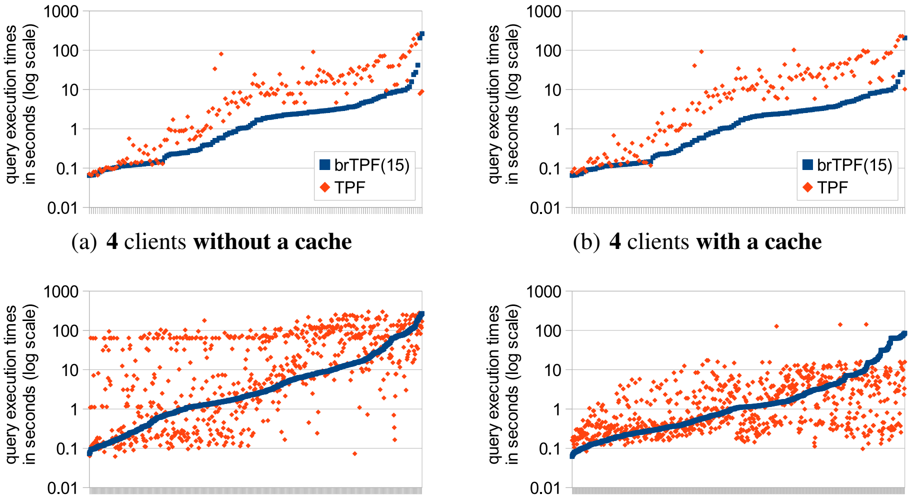
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems 34(3) (2009)
16. Sande, M.V., Verborgh, R., Herwegen, J.V., Mannens, E., de Walle, R.V.: Opportunistic linked data querying through approximate membership metadata. In: The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I. pp. 92–110 (2015)
17. Schmachtenberg, M., Bizer, C., Paulheim, H.: Adoption of the Linked Data Best Practices in Different Topical Domains. In: Proceedings of the 13th International Semantic Web Conference (ISWC) (2014)
18. Verborgh, R., Hartig, O., De Meester, B., Haesendonck, G., De Vocht, L., Vander Sande, M., Cyganiak, R., Colpaert, P., Mannens, E., Van de Walle, R.: Querying Datasets on the Web with High Availability. In: Proceedings of the 13th International Semantic Web Conference (ISWC) (2014)
19. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. Journal of Web Semantics 37–38, 184–206 (2016)

Appendix: Query Execution Times in the Multi-Client Experiments

The average query execution time (average QET) across all WatDiv queries that were executed completely (no timeouts) in the 4-clients setup without cache was 17.9 s for TPF (st.dev. 33.2) and 16.5 s for brTPF (st.dev. 44.9). For the 64-clients setup this average QET increased to 45.3 s for TPF (st.dev. 61.0) and to a much smaller 33.1 s for brTPF (st.dev. 57.7). For the setup with a cache, the relative differences are similar.

To drill into these numbers, the charts in Figure 5 detail the individual execution times for all queries that were executed completely by both TPF and brTPF in the respective setup. More precisely, in each chart, the queries are organized along the x-axis by sorting them from left to right based on their respective QET in the brTPF case. Then, at the x-axis position of such a query, the chart contains two measurement points that indicate the brTPF-based QET and the TPF-based QET of the query, respectively.

For the 4-clients setups without cache (Figure 5(a)) and with cache (Figure 5(b)) we observe that for almost all queries, brTPF achieved a QET that is either similar or even smaller than the QET achieved by using TPF. For the 64-clients setups (Figures 5(c) and 5(d)), there is more variation, which we attribute to fact that, for both approaches, TPF and brTPF, individual QETs are more affected by the higher load of the whole system. By interpreting these charts, the reader should keep in mind that these charts only consider queries executed completely in both the corresponding TPF and brTPF runs. What these charts do not show is that the brTPF approach achieved more than twice as many complete query executions as TPF as we recall from Figures 3(e) and 3(f).



(a) **4** clients **without a cache**

(b) **4** clients **with a cache**

(c) **64** clients **without a cache**

(d) **64** clients **with a cache**

Figure 5. Execution times of all the WatDiv queries that were executed completely (without timeout) by *both* the respective TPF setup and the respective brTPF setup (maxM/R=15). Hence, these charts ignore all queries that were executed completely by only one of the approaches (due to a timeout or a smaller throughput of the other approach). In each chart the measurements are ordered from left to right by the time that brTPF required to execute the corresponding query.