Intelligent clients for replicated Triple Pattern Fragments

Thomas Minier, Hala Skaf-Molli, Pascal Molli, Maria-Esther Vidal

Intelligent clients for replicated Triple Pattern Fragments

Thomas Minier[1], Hala Skaf-Molli[1], Pascal Molli[1], and Maria-Esther Vidal[2]

[1] LS2N, University of Nantes, Nantes, France
firstname.lastname@univ-nantes.fr
[2] TIB Leibniz Information Centre For Science and Technology
University Library & Fraunhofer IAIS, Germany
Maria.Vidal@tib.eu

Abstract. Following the Triple Pattern Fragments (TPF) approach, intelligent clients are able to improve the availability of the Linked Data. However, data availability is still limited by the availability of TPF servers. Although some existing TPF servers belonging to different organizations already replicate the same datasets, existing intelligent clients are not able to take advantage of replicated data to provide fault tolerance and load-balancing. In this paper, we propose Ulysses, an intelligent TPF client that takes advantage of replicated datasets to provide fault tolerance and load-balancing. By reducing the load on a server, Ulysses improves the overall Linked Data availability and reduces data hosting cost for organizations. Ulysses relies on an adaptive client-side load-balancer and a cost-model to distribute the load among heterogeneous replicated TPF servers. Experimentations demonstrate that Ulysses reduces the load of TPF servers, tolerates failures and improves queries execution time in case of heavy loads on servers.

Keywords: Semantic Web, Triple Pattern Fragments, Intelligent client, Load balancing, Fault tolerance, Data Replication

1 Introduction

The Triple Pattern Fragments (TPF) [16] approach improves Linked Data availability by shifting costly SPARQL operators from servers to intelligent clients. However, data availability is still dependent on servers' availability, *i.e.*, if a server fails, there is no failover mechanism, and the query execution fails too. Moreover, if a server is heavily loaded, performances can be deteriorated.

The availability of TPF servers can be ensured by cloud providers and consequently servers' availability are depended on the budget of data providers. An alternative solution is to take advantage of datasets replicated by different data providers. In this case, TPF clients can balance the load of queries processing among data providers. Using replicated servers, they can prevent a single point of failure server-side, improves the overall availability of data, and distributes the financial costs of queries execution among data providers.

Some data providers already replicate RDF datasets produced by other data providers [11]. Replication can be total, *e.g.*, both DBpedia [3] and LANL Linked Data Archive [4] publish the same versions of DBpedia datasets. Replication can also be partial, *e.g.*, LOD-a-lot [5] [4] gathers all LOD Laundromat datasets [6] into a single dataset, hence each LOD Laundromat dataset is a partial replication of LOD-a-lot.

Existing TPF clients allow to process a federated SPARQL query over a federation of TPF servers replicating the same datasets. However, *existing TPF clients do not support replication nor client-side load balancing* [16]. Consequently, the execution time of queries are severely degraded in presence of replication. To illustrate, consider the federated SPARQL query Q_1, given in Figure 1, and the TPF servers S_1 and S_2 owned respectively by DBpedia and LANL. Both servers host the DBpedia dataset 2015-10. Executing Q_1 on S_1 alone takes **7s** in average, and returns 222 results. Executing the same query as a federated SPARQL query on both S_1 and S_2 also returns 222 results, but takes **25s** in average.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?software ?company WHERE {
  ?software dbo:developer ?company. # tp1
  ?company dbo:locationCountry ?country . # tp2
  ?country rdfs:label "France"@en . # tp3
}
```

Fig. 1: Federated SPARQL query Q_1 finds all software developed by French companies, executed on S_1 : `http://fragments.dbpedia.org/2015-10/en` and S_2 : `http://fragments.mementodepot.org/dbpedia_201510`

Moreover, in the first setting, S_1 received **442 HTTP calls** while, in the federated setting, S_1 received **478 HTTP calls** and S_2 received **470 HTTP calls**. Thus, there was unnecessary transfer of data between the client and servers which increased the global load on servers without producing new results.

Distributing the load of Q_1 processing across S_1 and S_2 requires to know servers capabilities. As TPF servers are heterogeneous, *i.e*, they do not have the same processing capabilities and access latencies, poorly distributed load further deteriorates query processing.

In this paper, we propose Ulysses, a replication-aware intelligent TPF client providing load balancing and fault tolerance over heterogeneous replicated TPF servers. Managing replication in Linked Data has been already addressed in [10,11,13]. These approaches consider SPARQL endpoints and not TPF servers. Moreover, they focus on minimizing intermediate results and do not address the

[3] `http://fragments.dbpedia.org/`
[4] `http://fragments.mementodepot.org/`
[5] `http://hdt.lod.labs.vu.nl/?graph=LOD-a-lot`
[6] `http://lodlaundromat.org/wardrobe/`

problems of load-balancing and fault-tolerance. The load balancing problem with replicated datasets is addressed in [9] but without considering heterogeneous servers. The main contributions of this paper are:

- A replication-aware source selection for TPF servers.
- A light-weighted cost-model for accessing heterogeneous TPF servers.
- A client-side load balancer for distributing SPARQL query processing among heterogeneous TPF servers hosting replicated datasets.

The paper is organized as follows. Section 2 summarizes related works. Section 3 presents Ulysses approach and key contributions. Section 4 presents our experimental setup and details experimental results. Finally, conclusions and future works are outlined in Section 5.

2 Related Work

Triple Pattern Fragments The Triple Pattern Fragments approach (TPF) [16] proposes to shift complex query processing from servers to clients to improve availability and reliability of servers, at the cost of performance. In this approach, SPARQL query processing is distributed between a TPF server and a TPF client: the first only evaluates single triple patterns, issued to the server using HTTP requests, while the latter performs all others SPARQL operations [12]. Queries are evaluated using dynamic Nested Loop Joins that minimize the selectivity of each join, using metadata provided by the TPF server. The evaluation of SPARQL queries by TPF clients could require a great number of HTTP requests. For example, when processing $tp_3 \bowtie tp_2$ (of Q_1 in Figure 1), each solution mapping of tp_3 is applied to tp_2 to generate subqueries, which are then evaluated against a TPF server. As a TPF server delivers results in several pages, the evaluation of one triple pattern could require several HTTP requests. In our example, the client downloads 429 triples in 5 requests, and generates 429 new subqueries when joining with tp_1.

Federated SPARQL query processing with replication Federated SPARQL query engines [1,6,7,14] are able to evaluate SPARQL queries over a set of data sources, *i.e.* a federation. However, if the federated query engine is not aware of replicated data, computing complete results will degrade performance: the query engine has to contact every relevant source and will transfer redundant intermediate results. This is an issue for federations of SPARQL endpoints, as pointed in [10,13], and also for federations of TPF servers, as pointed in Section 1.

The Fedra [10] and LILAC [11] approaches address this issue in the context of SPARQL endpoints. Both prune redundant sources and use *data locality* to reduce data transfer during federated query processing. Fedra is a source selection algorithm that finds as many sub-queries as possible that can be answered by the same endpoint, increasing the number of joins evaluated locally by SPARQL endpoints. LILAC is a replication-aware decomposer that further reduces intermediate results by allocating the same triple patterns to several endpoints. In the context of TPF servers, data locality cannot be exploited and consequently

Fedra and LILAC are not pertinent. Moreover, Fedra and LILAC do not address problems of load-balancing and fault tolerance using replicated datasets.

PeNeLoop [9] makes available a replication-aware parallel join operator for federated SPARQL queries. PeNeLoop parallelizes join processing over SPARQL endpoints or TPF servers hosting replicated data by distributing bindings among available servers. However, PeNeLoop approach does not address the issue of heterogeneous servers. As the load of join processing is equally distributed across the federation, servers with poor performance or latency could deteriorate query execution time.

Client-side load balancing with heterogeneous servers Client-side load balancing is well suited for heterogeneous servers [3]. In this context, strategies for selecting servers can be classified into three categories: (i) *random*; (ii) *statistical*, by selecting the server with the lowest estimated latency; (iii) *dynamic*, using probing requests to select the fastest server. Dynamic probes perform better for selecting servers, but they add a communication overhead, as additional messages need to be exchanged between clients and servers. Dynamic probes reduce retrieval time for objects replicated on different servers but are not designed to distribute the cost among servers. Thus, powerful servers could receive all the load, and cost, of query processing. Smart clients [18] provide client-side load balancing and fault tolerance using a weighted random distribution algorithm: the probability of choosing a server is inversely proportional to its response time. Smart clients rely on probing to gather load information about servers but they do not propose an accurate and low-overhead load estimator that can be used to estimate the load of TPF servers.

3 Ulysses approach

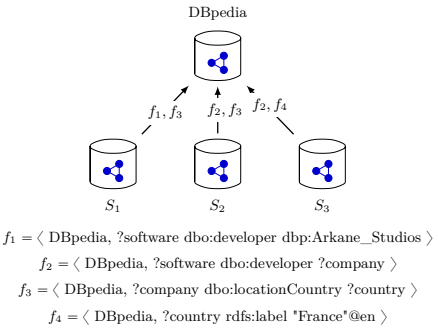
Ulysses proposes intelligent clients for replicated TPF servers. In order to balance the load on heterogeneous servers hosting replicated datasets, Ulysses relies on 3 key ideas:

First, it uses a *replication-aware source selection* algorithm to identify which TPF servers can be used to distribute evaluation of triple patterns during SPARQL query processing. The source selection algorithm relies on the replication model introduced in [10,11].

Second, Ulysses uses each call performed to a TPF server during query processing as a probe to compute the *processing capabilities* of the server. Since triple pattern queries can be resolved in constant time by TPF servers [5], observing HTTP responses times allows to compute an accurate load-estimation of TPF servers. Such estimation is updated in real-time during query processing and allows Ulysses to dynamically react to failures or heavy load of TPF servers.

Last, Ulysses uses an *adaptive load-balancer* to perform load balancing among replicated servers. Instead of simply selecting the server with the best access latency, Ulysses performs its selection using a weighted random algorithm: the probability of selecting a server is proportional to its processing

(a) Replicated fragments



(b) Triple patterns in Q_1 and their relevant fragments and servers

Triple pattern	**Relevant fragment**	**Relevant server(s)**
tp_1	f_1	S_1
	f_2	S_2, S_3
tp_2	f_3	S_1, S_2
tp_3	f_4	S_3

Fig. 2: Relevant replicated fragments for query Q_1 (from Figure 1)

capabilities. Thus, the load of query processing will be distributed across all replicated servers, *minimizing* the individual cost of query processing for each data provider. This load-balancer also provides *fault-tolerance*, by re-scheduling failed HTTP requests using available replicated servers.

3.1 Replication model

For replication, we follow the approach of replicated fragments introduced in [10,11] and recall related definitions, adapted for TPF servers.

Definition 1 (Fragment). *A* fragment *is a tuple $f = \langle u, tp_f \rangle$ where u is the authoritative source of the fragment,* e.g.*, DBpedia, and tp_f is the triple pattern met by the fragment's triples.*

Figure 2a shows a federation with three TPF servers S_1, S_2 and S_3, each of them exposing fragments replicated from DBpedia. For example, S_3 replicated the fragment f_4, which correspond to all triples matched by the triple pattern `?country rdfs:label "France"@en`. Notice that a total replication can be easily expressed using a fragment defined by the triple pattern `?s ?p ?o`.

Definition 2 (Fragment mapping). *A* fragment mapping *is a function $\mathcal{L}$ that maps a fragment to a set of TPF servers.*

For example, in Figure 2a, the fragment mapping of f_2 is $\mathcal{L}(f_2) = \{S_2, S_3\}$.

Definition 3 (Containment mapping). *A* containment mapping *is a containment relation $\sqsubseteq$ defined as follows:*

Let $tp(\mathcal{D})$ denote the evaluation of the triple pattern tp over an RDF dataset $\mathcal{D}$. Let tp_1 and tp_2 be two triple patterns. We say that tp_1 is contained in tp_2, denoted by $tp_1 \sqsubseteq tp_2$, if, $\forall$ RDF dataset $\mathcal{D}, tp_1(\mathcal{D}) \subseteq tp_2(\mathcal{D})$.

Computing triple pattern containment has a complexity of $O(1)$, as demonstrated in [10].

Algorithm 1: Ulysses Source Selection algorithm

Input: Q: SPARQL query, S: set of TPF servers, F: set of fragments
Output: *selection*: map from Triple Pattern to set of set of TPF servers

```
1  Function SourceSelection(Q, S, C):
2      selection ← empty map
3      patterns ← get triple patterns in Q
4      for each tp ∈ patterns do
5          selection[tp] ← ∅
6          R(tp) ← RelevantFragments(tp, F)
7          R(tp) ← CheckContainment(R(tp), F)
8          for each f ∈ R(tp) do
9              L(f) ← FragmentLocations(f, S)
10             selection[tp] ← selection[tp] ∪ {L(f)}
11     return selection
```

For example, in Figure 2a, we have $f_1 \sqsubseteq f_2$, as all triples matching f_1 pattern are included in the triples matching f_2 pattern.

Definition 4 (Fragment relevance). *Let f be a fragment defined by a triple pattern tp_1. Let tp_2 be a triple pattern of a SPARQL query Q. f is* relevant *to Q if $tp_1 \sqsubseteq tp_2$ or $tp_2 \sqsubseteq tp_1$.*

Figure 2b shows the relevant fragments of query Q_1, from Figure 1, using the fragments defined in Figure 2a. For example, f_1 and f_2 are relevant to tp_1, as $tp_{f_1} \sqsubseteq tp_1$ and $tp_{f_2} \sqsubseteq tp_1$.

3.2 Replication-aware source selection for Triple Pattern Fragments

When processing a SPARQL query, Ulysses loads a *catalog* that describes fragments and the servers that provide access to them, *i.e.*, the fragment localities. In this paper, we made the following assumptions:

- We do not address how the catalog is obtained. It could be provided as an input by the user, any server in the federation or by a dedicated service that record replication between online datasets.
- For simplicity, we consider that replicated fragments are synchronized, *i.e.* there are no updates. Managing consistency between replicated datasets with updates is addressed in [8]. Most TPF servers address this issue by hosting *versioned datasets* [15].

Algorithm 1 presents Ulysses replication-aware source selection algorithm. This algorithm identifies the TPF servers that can be used to evaluate each triple pattern of a query. The following example illustrates how this algorithm works.

Example 1. Consider Algorithm 1 with the following inputs: the SPARQL query Q_1 from Figure 1, the set of TPF servers $S = \{S_1, S_2, S_3\}$ and the fragments from

Figure 2. First, relevant fragments are computed for each triple pattern (line 6): $R(tp_1) = \{f_1, f_2\}$, $R(tp_2) = \{f_3\}$ and $R(tp_3) = \{f_4\}$. Notice that triple patterns with more than one relevant fragments require the retrieval of all relevant fragments to get complete results. Next, we compute triple pattern containment to remove redundant fragments (line 7), f_1 is removed because $f_1 \sqsubseteq f_2$, and then fragments are localized on TPF servers (line 9): $L(f_2) = \{S_2, S_3\}$, $L(f_3) = \{S_1, S_2\}$ and $L(f_4) = \{S_3\}$. Finally, the source selection of each triple pattern is computed (line 10): $selection[tp_1] = \{\{S_2, S_3\}\}$, $selection[tp_2] = \{\{S_1, S_2\}\}$ and $selection[tp_3] = \{\{S_3\}\}$.

The results of the source selection algorithm are used to identify the TPF servers that replicate the same relevant fragments. These servers can be used to distribute the evaluation of triple patterns. However, as TPF servers are heterogeneous, *i.e.*, they do not exhibit the same processing capabilities, we must ensure that servers with weaker processing capabilities are less requested in favor of more powerful servers, to maintain good query processing performance. To this end, we define a cost-model to compute and evaluate servers capabilities.

3.3 A cost-model for evaluating TPF servers capabilities

The cost model uses server capability factors to evaluate servers capabilities. A *server capability factor* depends on: (i) The access latency of the TPF client for this server. (ii) The processing capabilities of the server, *i.e.*, in terms of CPU, RAM, etc. (iii) The impact of the server loads on its processing capabilities.

As a triple pattern is evaluated in constant time [5], a server capability factor can be deduced from its access time. If an HTTP request is not resolved in the server cache, a *server access time* is the time to receive one page of RDF triples matching a triple pattern from the TPF server [7]. However, as the size of pages could be different among servers, two servers with the same access times do not necessarily produce results at the same rate. Thus, we choose to rely on a *server throughput*, *i.e.*, the number of results served per unit of time, to evaluate more precisely its processing capabilities.

Definition 5 (Server throughput). *Given a set of TPF servers* $S = \{S_1, \ldots, S_n\}$, $\Delta = \{\delta_1, \ldots, \delta_n\}$ *where* δ_i *is the access time of* S_i*, and* $P = \{p_1, \ldots, p_n\}$ *where* p_i *is the number of results served per access to* S_i*.*

$\forall S_i \in S$*, the server throughput* w_i *of* S_i *is* $w_i = \dfrac{p_i}{\delta_i}$

Example 2. Consider three TPF servers S_1, S_2 and S_3, with their access times be $\delta_1 = \delta_2 = 100\ ms$ and $\delta_3 = 500\ ms$, and the number of results they serve per access be $p_1 = 100$ and $p_2 = p_3 = 400$, respectively. Using Definition 5, we compute the servers throughput as $w_1 = 1$, $w_2 = 4$ and $w_3 = 0.8$, respectively. Notice that S_1 and S_2 have the same access times, but using their throughput, we observe that S_2 delivers more triples per unit of time than S_1.

[7] We suppose that an HTTP client is able to detect if an HTTP request has been resolved in the cache.

Next, we use the throughput of a TPF server to estimate its capability. The capability is normalized with respect to other servers used to evaluate the query.

Definition 6 (Server capability). *Given a set of TPF servers* $S = \{S_1, \ldots, S_n\}$ *and* $W = \{w_1, \ldots, w_n\}$ *where* w_i *is the throughput of* S_i*.*

$\forall S_i \in S$*, the capability* ϕ_i *of* S_i *is* $\phi_i = \left\lfloor \frac{w_i}{\min W} \right\rfloor$

Example 3. Consider the set of servers $S = \{S_1, S_2, S_3\}$ and their throughputs $W = \{w_1 = 1, w_2 = 4, w_3 = 0.8\}$ from Example 2. Using Definition 6, we compute the capability of S_1, S_2 and S_3 as $\phi_1 = 1$, $\phi_2 = 5$ and $\phi_3 = 1$, respectively. We observe that S_1 and S_3 have similar capabilities, even if S_3 access times is higher than S_1, and that S_2 is five times more powerful than both S_1 and S_3.

3.4 Accessing TPF servers based on capabilities

We follow a load distribution approach similar to Smart Clients [18], with a random algorithm weighted by the servers capabilities. This allows for quick adaptation to variations in server loads: if a server throughput is deteriorated, its capability will decrease and it will be less frequently accessed. Definition 7 states how to access a set of TPF servers in such way.

Definition 7 (Weighted random access). *Given a set of TPF servers* $S = \{S_1, \ldots, S_n\}$ *and* $\Phi = \{\phi_1, \ldots, \phi_n\}$ *where* ϕ_i *is the capability of* S_i*.*

When selecting a TPF server $S_i \in S$ *to evaluate a triple pattern tp, the probability of selecting* S_i *is:* $\mathcal{A}(S_i) = \frac{\phi_i}{\sum_{j=1}^{n} \phi_j}$*, such as: (i)* $\sum_{S_i \in S} \mathcal{A}(S_i) = 1$*; (ii)* $\forall S_i \in S, 0 \leqslant \mathcal{A}(S_i) \leqslant 1$.

Example 4. Consider again the set of TPF servers $S = \{S_1, S_2, S_3\}$ and the set of capabilities $\Phi = \{\phi_1 = 1, \phi_2 = 5, \phi_3 = 1\}$ computed in Example 3.

According to Definition 7, the probability of selecting S_1, S_2 and S_3 for evaluating a triple pattern are $\mathcal{A}(S_1) = \frac{1}{7}$, $\mathcal{A}(S_2) = \frac{5}{7}$ and $\mathcal{A}(S_3) = \frac{1}{7}$, respectively.

Next, we define how Ulysses uses the cost-model to effectively distribute the evaluation of triples patterns across replicated TPF servers.

3.5 Ulysses adaptive client-side load balancing with fault tolerance

Ulysses defines an adaptive client-side load balancer that acts as a *transparent component* between the client and the set of replicated TPF servers. When the TPF query engine evaluates a triple pattern, it uses the load balancer to perform the evaluation. The load balancer distributes accesses to relevant TPF servers according to servers capabilities as defined in Section 3.4.

Algorithm 2 describes the load balancing algorithm used by Ulysses. First, the sources selected by the source selection algorithm are used to find S_{tp}, a set of set of TPF servers (line 3). Each $S' \in S_{tp}$ is a set of servers that replicates one

Algorithm 2: Ulysses Load Balancing algorithm

Data: $S = \{S_1, \ldots, S_n\}$: set of TPF servers, *selection*: map of Triple Pattern to set of set of TPF servers, $\Phi = \{\phi_1, \ldots, \phi_n\}$: set of servers capabilities.

```
1  Function EvaluatePattern(tp: triple pattern):
2      μ ← ∅
3      S_tp ← selection[tp]
4      for each S' ∈ S_tp do
5          if |S'| > 1 then // replicated servers available
6              s' ← a TPF server selected using Φ, such as s' ∈ S'
7          else
8              s' ← the only server s' ∈ S'
9          μ ← μ ∪ { Evaluate tp at s' }
10     return μ
11 Event OnHTTPResponse(S_i: TPF server, δ': request execution time):
12     Update access times of S_i and recompute Φ using it
13 Event OnHTTPFailure(q: HTTP request, S_i: TPF server, tp: triple pattern):
14     ∀S' ∈ selection[tp], remove S_i from S'
15     if selection[tp] = ∅ then // all servers have failed
16         FailQuery()
17     Retry q using another relevant TPF server, selected using Φ
```

relevant fragment of tp. Ulysses has to evaluate tp using at least one server in each S' to get complete results (line 4). For each $S' \in S_{tp}$, if replicated servers are available, the set of servers capabilities is used to select a server to evaluate tp (line 6). Otherwise, the unique server in S' is used to evaluate tp (line 8).

Additionally, Ulysses load balancer *adapts* to changes in network conditions and provides *fault tolerance*. If a valid HTTP response is received from a server (line 11), its access time is updated in Ulysses cost-model and the set of servers capabilities is recomputed to be kept up-to-date. Furthermore, if a server has failed to process a request (line 13), it is removed from the cost-model and the request is re-scheduled using an alternative server (lines 14-17).

4 Experimental study

The goal of the experimental study is to evaluate the effectiveness of Ulysses: (i) Ulysses produces complete results and does not deteriorate query execution time; (ii) The load distribution is done conforming to the cost model; (iii) Ulysses speed-up query execution when servers are loaded; (iv) Ulysses is able to tolerate faults and adapts to the load of servers in real time.

We compare the performance of the reference TPF client alone (denoted as TPF) and the same TPF client with the addition of Ulysses (denoted as Ulysses) [8]

4.1 Experimental setup

Dataset and Queries: We use one instance of the Waterloo SPARQL Diversity Test Suite (WatDiv) synthetic dataset [2] with 10^7 triples, encoded in the HDT format [5]. We generate 50,000 DISTINCT queries from 500 templates (STAR, PATH, and SNOWFLAKE shaped queries). Next, we eliminate all duplicated queries, and then pick 100 random queries to be used in our experiments. Queries that failed to deliver an answer due to a query engine internal error with the regular TPF client are excluded from all configurations.

Type of replication: We consider two types of replication: (i) *total replication*: our WatDiv dataset is replicated by all servers in the experimentation; (ii) *partial replication*: fragments are created from the 100 random queries and are replicated up to two times. Each replica is assigned randomly to a server in the experimentation.

Servers and client configurations: We use the Amazon Elastic Compute Cloud (Amazon EC2) to host our WatDiv dataset with the latest version of the TPF server. RDF triples are served per page of 100 triples. Each server use *t2.micro* instances (one core virtual CPU, 1GB of RAM), with 4 workers and no HTTP web cache. HTTP proxies are used to simulate network latencies and special conditions, using two configurations: (i) *Homogeneous*: all servers have access latencies of 300ms. (ii) *Heterogeneous*: The first server has an access latency of 900ms, and other servers have access latencies of 300ms.

The Ulysses TPF client is hosted on a machine with Intel Core i7-4790S 3.20GHz and 2BG of RAM and implemented as an extension of the reference TPF client.

Evaluation Metrics: (i) *Execution time (ET)*: is the elapsed time since the query is posed until a complete answer is produced. (ii) *HTTP response time (HRT)*: is the elapsed time since a HTTP request is submitted by the client to a TPF server until a complete HTTP response is received. (iii) *Number of HTTP requests per server (NHR)*: is the number of HTTP requests performed by the client against each TPF server for a query. Thus, it represents the load that each query injected on each server. (iv) *Answer Completeness (AC)*: is the ratio between the answers produced by the evaluation of a query using the reference TPF client and the evaluation by Ulysses; values ranges between 0.0 and 1.0.

Results presented for all metrics correspond to the average obtained after three successive evaluation of our queries.

[8] The datasets, queries, code and results relative to the experiment are available the companion web site `https://callidon.github.io/ulysses-tpf` as long as with an online demo `http://ulysses-demo.herokuapp.com`.

4.2 Experimental Results

Query execution time and Answer Completeness: First, we check that Ulysses preserves answer completeness. We executed our 100 random queries with Ulysses using one, two and three homogeneous TPF servers. As a baseline, we also executed our queries with the reference TPF client, using one TPF server. In all configurations, Ulysses is able to produce the same answers as the baseline for all queries.

Next, to confirm that Ulysses does not deteriorate query execution time, we run a *Wilcoxon signed rank test* [17] for paired non-uniform data for the query execution time results obtained by Ulysses, using up to three servers, with the following hypothesis: H_0: Ulysses does not change SPARQL query execution time compared to the reference TPF client; H_1: Ulysses does change SPARQL query execution time compared to the reference TPF client.

We obtained p-values of $2.83019e^{-17}, 9.0472e^{-12}$ and $5.05541e^{-12}$ for configurations with one, two and three servers, respectively. These low p-values allow for rejecting the null hypothesis and support that Ulysses do change SPARQL query execution times compared to the reference TPF client.

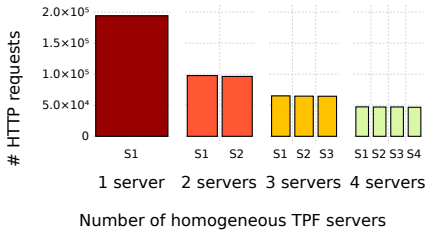
Next, we validate that Ulysses is able to distribute the load of query processing according to servers' capabilities both in a total replication settings and in a partial replication settings.

Load distribution with total replication: Figure 3a shows the number of HTTP requests per server (*NHR*) after the evaluation of our workload of 100 queries, with up to four homogeneous servers that totally replicate the dataset. The configuration with one server runs with the reference TPF client, others with Ulysses. As all servers have the same capabilities according to Ulysses cost-model, the requests are equally distributed among servers. Ulysses reduces the number of HTTP requests received per server. Consequently, each server receives fewer loads during query processing and servers availabilities are potentially increased.

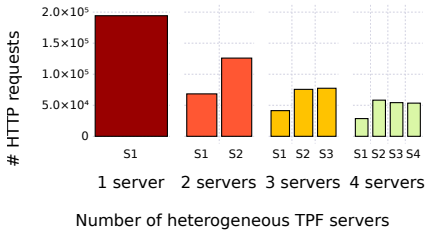
Figure 3b shows the same experiment with heterogeneous servers. Again, Ulysses is able to distribute the load according to servers capabilities: as S_1 is three times slower than other servers, therefore it receives less requests.

Load distribution with partial replication: Figure 3c shows for the five queries (from our 100 queries) that generate the most HTTP requests, the number of HTTP requests per server, grouped by triple patterns in a query. We consider four homogeneous servers and partial replication. Results are similar to those obtained previously: the HTTP requests required to evaluate a triple pattern are distributed across servers that replicate relevant fragments. As the load of a query processing is distributed at triple pattern level, we conclude that the shape of a SPARQL query does not influence the load distribution.

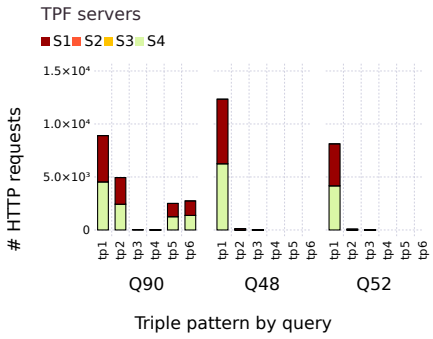
Execution time under load: We study the impact of Ulysses load balancing on query execution time when servers experience heavy load. We separately study

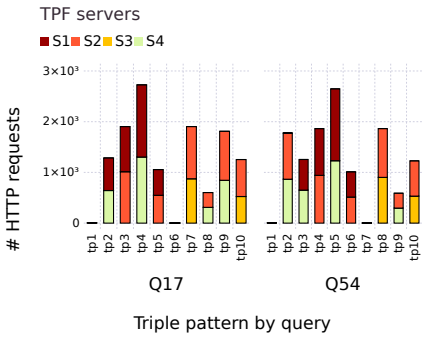


(a) All queries with **homogeneous servers** and **total replication** (All servers have same capabilities)



(b) All queries with **heterogeneous servers** and **total replication** (S_1 is three times slower than the others)





(c) The five queries that generate the most HTTP requests (**Homogeneous servers** and **partial replication**)

Fig. 3: Average number of HTTP requests received by servers after evaluation of WatDiv queries, using several configurations

the query 72 (from our set of 100 queries) of the template query from WatDiv that generates an average load of requests (590 HTTP requests). Figure 4 shows the execution time of this query, using up to twenty homogeneous servers. The servers load is generated using several TPF clients, up to a hundred, that evaluate the same query against the servers.

With only one server, results are similar to those obtained in [16]: as the load increases, the server is less available and the query execution time is deteriorated. Using several replicated servers, Ulysses distributes the load among servers and improves availablity, so query execution time is significantly improved. This improvement is not proportional to the number of replicated servers available: for example, gains are more important between one and two servers than between three and twenty servers.

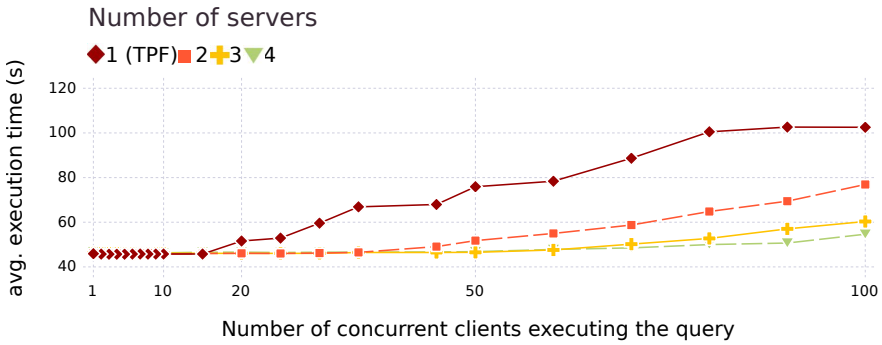


Fig. 4: Average query execution time with an increasing number of concurrent clients and available servers, using Ulysses client

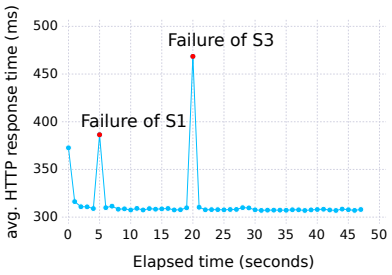


Fig. 5: Average HTTP response time when evaluating query 72 using three homogeneous servers (S_1, S_2, S_3) in presence of failures: S_1 fails at 5s and S_3 fails at 20s

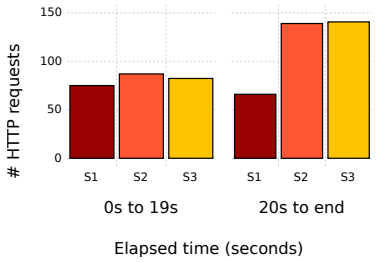


Fig. 6: Average number of HTTP requests received by servers S_1, S_2, S_3 during evaluation of query 72. Servers start homogeneous, then S_1 access latency is tripled at 20s

Fault tolerance: We examine how Ulysses reacts to faults. Figure 5 shows the average HTTP responses times recorded client-side, when Ulysses evaluates the query 72 using three homogeneous servers S_1, S_2 and S_3 in presence of failure: after 5 seconds, S_1 becomes unavailable, and after 20 seconds, S_3 also becomes unavailable. We observe that Ulysses is able to tolerate servers failure and evaluates the query with 100% answer completeness. When a failure is detected, Ulysses distributes failed requests among available servers and resumes query execution in less than a second. Tolerating faults involves a slight overhead, as failed requests need to be re-executed.

Load adaptivity: We examine the evaluation of query 72 in a context where servers load vary during query execution. Figure 6 shows the average number of HTTP requests received by servers S_1, S_2 and S_3 during the evaluation of

query 72 by Ulysses. Replication is total and servers start homogeneous, after 20s the access latency of S_1 is tripled. Before S_1 becomes loaded, requests are evenly distributed between the three servers, as they have the same processing capabilities. Passed the 20 seconds, Ulysses detects that processing capabilities of S_1 have been deteriorated and adapts the load distribution in consequence: S_2 and S_3 receive more requests until the end of the query processing. Ulysses is able to quickly adapt to changes in servers conditions.

5 Conclusion and Future Works

In this paper, we presented Ulysses, a replication-aware intelligent TPF client providing load balancing and fault tolerance over heterogeneous replicated TPF servers. Ulysses accurately evaluates processing capabilities of TPF servers using only HTTP responses times observed during query processing. Experimental results demonstrate that Ulysses reduces the individual load per server, speeds up query execution time under heavy load, tolerates faults, and adapts to the load of servers in real-time. Moreover, by distributing the load among different data providers, Ulysses distributes the financial costs of queries execution among data providers without impacting query execution times for end-users.

Ulysses opens several perspectives. First, we do not address how the catalog of replicated fragments can be acquired. It could be provided by TPF servers as additional metadata or built collaboratively by TPF clients as they evaluate SPARQL queries. Building and maintaining the catalog of replicated data over the web is challenging. Another perspective is to consider divergence over replicated data. Executing queries over weakly-consistent replicated datasets raises interesting issues about the correctness of results.

Acknowledgments This work is partially supported through the FaBuLA project, part of the AtlanSTIC 2020 program.

References

1. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: Anapsid: an adaptive query processing engine for sparql endpoints. ISWC 2011 pp. 18–34 (2011)
2. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: International Semantic Web Conference. pp. 197–212. Springer (2014)
3. Dykes, S.G., Robbins, K.A., Jeffery, C.L.: An empirical evaluation of client-side server selection algorithms. In: INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. vol. 3, pp. 1361–1370. IEEE (2000)
4. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: LOD-a-lot. In: International Semantic Web Conference. pp. 75–83. Springer (2017)
5. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semantics: Science, Services and Agents on the World Wide Web 19, 22–41 (2013)

6. Görlitz, O., Staab, S.: Federated data management and query optimization for linked open data. In: New Directions in Web Data Management 1, pp. 109–137. Springer (2011)
7. Görlitz, O., Staab, S.: Splendid: Sparql endpoint federation exploiting void descriptions. In: Proceedings of the Second International Conference on Consuming Linked Data-Volume 782. pp. 13–24. CEUR-WS. org (2011)
8. Ibáñez, L.D., Skaf-Molli, H., Molli, P., Corby, O.: Col-graph: Towards writable and scalable linked open data. In: International Semantic Web Conference. pp. 325–340. Springer (2014)
9. Minier, T., Montoya, G., Skaf-Molli, H., Molli, P.: Parallelizing federated SPARQL queries in presence of replicated data. In: The Semantic Web: ESWC 2017 Satellite Events, Revised Selected Papers. pp. 181–196 (2017)
10. Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Federated SPARQL queries processing with replicated fragments. In: International Semantic Web Conference. pp. 36–51. Springer (2015)
11. Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Decomposing federated queries in presence of replicated fragments. Web Semantics: Science, Services and Agents on the World Wide Web 42, 1–18 (2017)
12. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems (TODS) 34(3), 16 (2009)
13. Saleem, M., Ngomo, A.N., Parreira, J.X., Deus, H.F., Hauswirth, M.: DAW: duplicate-aware federated query processing over the web of data. In: The Semantic Web - ISWC 2013. pp. 574–590 (2013)
14. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: Optimization techniques for federated query processing on linked data. In: International Semantic Web Conference. pp. 601–616. Springer (2011)
15. Vander Sande, M., Verborgh, R., Hochstenbach, P., Van de Sompel, H.: Toward sustainable publishing and querying of distributed linked data archives. Journal of Documentation 74(1), 195–222 (2018)
16. 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. Web Semantics: Science, Services and Agents on the World Wide Web 37, 184–206 (2016)
17. Wilcoxon, F.: Individual comparisons by ranking methods. In: Breakthroughs in Statistics, pp. 196–202. Springer (1992)
18. Yoshikawa, C., Chun, B., Eastham, P., Vahdat, A., Anderson, T., Culler, D.: Using smart clients to build scalable services. In: Proceedings of the 1997 USENIX Technical Conference. p. 105. CA (1997)