Extracting Basic Graph Patterns from Triple Pattern Fragment Logs

Georges Nassopoulos, Patricia Serrano-Alvarado, Pascal Molli, Emmanuel Desmontils

Extracting Basic Graph Patterns from Triple Pattern Fragment Logs

Georges Nassopoulos, Patricia Serrano-Alvarado, Pascal Molli, and Emmanuel Desmontils

LS2N Laboratory - Université de Nantes – France
{firstname.lastname}@univ-nantes.fr

Abstract. The Triple Pattern Fragment (TPF) approach is de-facto a new way to publish Linked Data at low cost and with high server availability. However, data providers hosting TPF servers are not able to analyze the SPARQL queries they execute because they only receive and evaluate queries with one triple pattern. In this paper, we propose LIFT: an algorithm to extract Basic Graph Patterns (BGPs) of executed queries from TPF server logs. Experiments show that LIFT extracts BGPs with good precision and good recall generating limited noise.

Keywords: Linked Data, Triple Pattern Fragments, log analysis, Basic Graph Patterns

1 Introduction

The Triple Pattern Fragment (TPF) approach is de-facto a new way to publish Linked Data at low cost and with high availability for data providers [15]. WarDrobe [1] provides more than 38 billions of triples distributed over 65 datasets. Following the TPF approach, most of the SPARQL query processing is now executed on the client-side, *TPF servers only receive and evaluate queries with one triple pattern*. Consequently, data providers of TPF servers do not know the executed SPARQL queries and cannot analyze them as data providers do with queries of SPARQL endpoints.

Knowing executed SPARQL queries is fundamental for data providers. Mining logs of SPARQL endpoints allows to detect recurrent patterns in queries for prefetching [4] or for benchmarking [8]. It provides the type of queries issued, the complexity and the used resources/predicates [6, 10]. It allows also to distinguish between man or machine made queries [11, 12]. Currently, such analysis cannot be done on logs of TPF servers because they only contain information about single triple patterns.

In this paper, we propose LIFT (LInked data Fragment Tracking): an algorithm to extract Basic Graph Patterns (BGPs) from logs of TPF servers. Compared to the state of art, [14] reported statistics from the logs of the DBpedia's TPF server. In previous work [9], we proposed an algorithm to extract BGPs of *federated SPARQL queries* from logs of a *federation of SPARQL endpoints*.

Here, we address a similar scientific problem but in the context of a single TPF server.

The main challenge to extract BGPs is the concurrent execution of SPARQL queries on one TPF server. If we find a function f, to extract BGPs from isolated traces of one SPARQL query, is f able to extract the same BGP from traces of concurrent SPARQL queries? LIFT faces this problem by tracking the bindings among different triple pattern queries to detect joins. We experimented LIFT with different levels of concurrency. We demonstrate in which conditions, it extracts BGPs with good precision and good recall generating limited noise. Thanks to LIFT, we were able to extract the frequent BGPs from the TPF log published in the USEWOD 2016 dataset [5].

Next section introduces a motivating example and our problem statement. Section 3 presents LIFT and Section 4 shows our experiments. Section 5 presents related work. Finally, conclusions and future work are outlined in Section 6.

2 Motivating example and problem statement

In Figure 1, two clients, c_1 and c_2, execute concurrently queries Q_1 and Q_2 over the DBpedia's TPF server. Q_1 asks for movies starring Brad Pitt and Q_2 for movies starring Natalie Portman.[1] Both queries have one BGP composed of several triple patterns (tp_n).

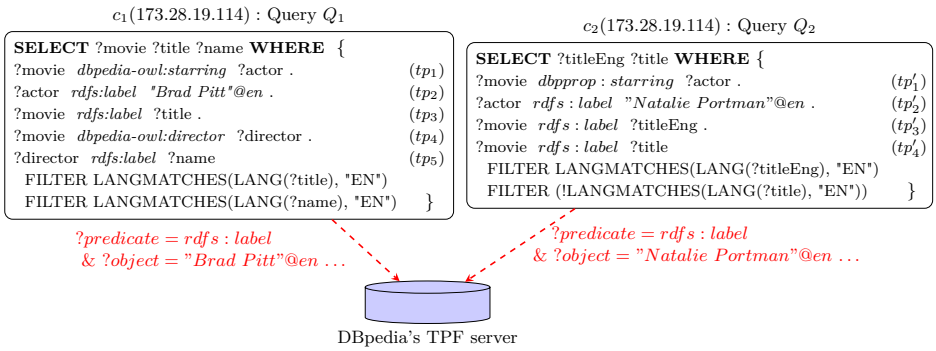


Fig. 1: Concurrent execution of queries Q_1 and Q_2.

TPF clients decompose SPARQL queries into a sequence of triple pattern queries. Table 1 presents some traces of the TPF server for query Q_1. Odd-numbered lines represent received triple pattern queries and even-numbered ones represent sent triples after evaluation on the RDF graph.

Lines 1 and 3, correspond to triple pattern queries for tp_2 and tp_1 of Q_1.[2] We can observe that the object in Line 3, comes from a mapping seen in Line

[1] These queries come from `http://client.linkeddatafragments.org/`.

[2] TPF clients only request bound parts of a triple patterns, variables are omitted.

	IP	Time	Asked triple pattern/TPF
1	172...	11:24:19	?predicate=rdfs:label & ?object="Brad Pitt"@en
2	172...	11:24:23	**dbpedia:Brad_Pitt** rdfs:label "Brad Pitt"@en ,
3	172...	11:24:24	?predicate=dbpedia-owl:starring & ?object=**dbpedia:Brad_Pitt**
4	172...	11:24:27	**dbpedia:A_River_Runs_Through_It_(film)** dbpedia-owl:starring dbpedia:Brad_Pitt dbpedia:Troy_(film) dbpedia-owl:starring dbpedia:Brad_Pitt ...
5	172...	11:24:28	?subject=**dbpedia:A_River_Runs_Through_It_(film)** &?predicate=rdfs:label

Table 1: Excerpt of log of the DBpedia's TPF server for query Q_1.

2. This *injection* of a mapping obtained from a previous triple pattern query, is clearly a join implemented in a nested-loop from tp_2 towards tp_1.

As the TPF server only sees triple pattern queries, the original queries e.g., Q_1 and Q_2 are unknown to the data provider. In this work, we address the following research question: *Can we extract the BGPs from a TPF server log?*

In our example, we aim to extract two BGPs from the TPF server log, one corresponding to Q_1, BGP[1]= $\{tp_1.tp_2.tp_3.tp_4.tp_5\}$ and another corresponding to Q_2, BGP[2]= $\{tp'_1.tp'_2.tp'_3.tp'_4\}$. Before presenting our scientific problem, we introduce the following definitions.

Definition 1 (BGP). *A BGP (Basic Graph Pattern) is a set of triple patterns. Any tuple* $\in (\mathcal{I} \cup \mathcal{L} \cup V) \times (\mathcal{I} \cup V) \times (\mathcal{I} \cup \mathcal{L} \cup V)$ *is a triple pattern* $\langle s, p, o \rangle$*, where* $\mathcal{I}$ *is the set of all IRIs,* $\mathcal{L}$ *the set of all literals and* V *the set of all variables disjoint from* $\mathcal{L}$ *and* $\mathcal{I}$*.*[3]

Definition 2 (TPF server log). *A TPF server log is a totally ordered sequence of execution traces structured in tuples* $\langle ip, ts, tp, \mu_o \rangle$ *where* ip *is the IP address of the client,* ts *is the timestamp of the http request,* tp *is a triple pattern, and* μ_o *is the set of RDF triples returned by the TPF server transformed in mappings.*

We denote by $E(Q_i)$, the log produced by a TPF server when evaluating the SPARQL query Q_i and by $E(Q_1 \parallel ... \parallel Q_n)$ the log of n *concurrent queries.*

Definition 3 (Approximation $\approx$ of BGPs). *A BGP approximates another* ($\approx$) *if, to some extent, both contain same triple patterns and same joins.*

To measure such approximation we can use *precision* and *recall* of triple patterns and joins of one BGP against another. The average of precision and recall can be used as a measure of global *quality* of the approximation.

Definition 4 (Problem of BGP reversing). *Given a log corresponding to the execution of one query,* $E(Q_i)$*, find a function* $f(E(Q_i))$ *producing a set of BGPs* $\{BGP_1, ..., BGP_n\}$*, such that:*

[3] We do not consider blank nodes in this paper.

Property 1. $f(E(Q_i))$ *approximates* ($\approx$) *the BGPs existing in the original query. If we consider that* $BGP(Q_i)$ *returns the set of BGPs of* Q_i *then* $f(E(Q_i)) \approx BGP(Q_i)$.

Property 2. $f(E(Q_i))$ *guarantees* resistance to concurrency*, i.e., BGPs obtained from the log of* isolated *queries, approximate* ($\approx$) *results obtained from the log of* concurrent *queries:* $f(E(Q_1)) \cup ... \cup f(E(Q_n)) \approx f(E(Q_1 \parallel ... \parallel Q_n))$.

We evaluate the BGPs extracted by f with the precision, recall and quality of triple patterns and joins returned by f against those existing in original queries. If $f(E(Q_1))$ extracts the BGP= $\{tp_1.tp_2.tp_3.tp_4.tp_5\}$, then precision, recall and quality of triple patterns and joins are perfect according to the BGP present in Q_1, even if variables have different names. But if $f(E(Q_1))$ misses one triple pattern (e.g., tp_5), then precision is 4/4, recall is 4/5 and quality is $(1 + 4/5)/2$.

In Figure 1, if c_1 and c_2 have different IP addresses, it is possible to separate $E(Q_1 \parallel Q_2)$ into $E(Q_1)$, $E(Q_2)$ and apply the reversing function to each trace. However, in the worst case, c_1 and c_2 have the same IP address, i.e., a web application running on the cloud that executes queries Q_1 and Q_2 in parallel. Thus, we expect that $f(E(Q_1 \parallel Q_2)) \approx f(E(Q_1)) \cup f(E(Q_2))$.

3 LIFT: a reversing function

We propose LIFT as a system of heuristics to implement f. The idea is to detect nested-loop joins. In Table 1, the mappings returned in Line 2 are reused in the next triple pattern query at Line 3. We track such bindings in order to link variables of different triple pattern queries. In this paper, we make the following hypothesis: (i) we consider only bound predicates, (ii) we do not consider the server's web cache (this information can be easily obtained by data providers), and (iii) we do not consider the client's TPF cache. The first hypothesis can be omitted, but we kept it because analysis of SPARQL queries show that predicates are frequently bound [2].

ts	tp	μ_o	
1	?s p2 toto	?s	**c1 c2**
3	**c1** p1 ?o	?o	a
5	**c2** p1 ?o	?o	b

(a) Log $E(Q_3)$.

ts	tp	μ_o	
1	?s p2 toto	?s	c1 c2
2	?s p3 titi	?s	c3 c4
3	c1 p1 ?o	?o	a
4	c3 p4 tata		
5	c2 p1 ?o	?o	b
6	c3 p1 ?o	?o	c
7	c4 p4 tata		
8	c4 p1 ?o	?o	d

(b) Log $E(Q_3 \parallel Q_4)$.

ts	tp	μ_o	
2	?s p3 titi	?s	**c3 c4**
4	**c3** p4 tata		
6	**c3** p1 ?o	?o	c
7	c4 p4 tata		
8	c4 p1 ?o	?o	d

(c) Log $E(Q_4)$.

Fig. 2: Examples of simplified TPF logs.

Figure 2 presents a simplified log of $E(Q_3)$, $E(Q_4)$ and $E(Q_3 \parallel Q_4)$ where:
$Q_3 = SELECT\ *\ WHERE\ \{?x\ p2\ toto\ .\ ?x\ p1\ ?y\}$ and
$Q_4 = SELECT\ *\ WHERE\ \{?x\ p3\ titi\ .\ ?x\ p1\ ?y\ .\ ?x\ p4\ tata\}$.

For the sake of simplicity, timestamps are transformed into integers. The IP address of the TPF client is the same for Q_3 and Q_4, so we removed the *ip* column. Variables are named $?s$ or $?o$. μ_o represents the mappings of variables resulting from the evaluation of *tp*. We call them *output-mappings*. Observe the client first requests more selective triple patterns, i.e., $\langle ?x\ p2\ toto\rangle$ for Q_3 and $\langle ?x\ p3\ titi\rangle$ for Q_4, leaving at the end less selective ones i.e., $\langle ?x\ p1\ ?y\rangle$ for both queries. Then mappings returned by selective patterns are bound into less selective ones producing a nested-loop. See that mappings $c1$ and $c2$ are bound in the variable $?x$ of the second triple pattern of Q_3. Similarly, mappings $c3$ and $c4$ are bound in the variable $?x$ of the other triple patterns of Q_4. We call *input-mappings* these injected mappings. Modified triple patterns are the inner part of the nested-loop that we call *inner loop*. We call *outer loop* the triple patterns whose mappings are used to bound variables, e.g., $\langle ?x\ p2\ toto\rangle$.

The basic intuition of LIFT is to detect if mappings obtained in a request are bound in next requests. This can be challenging because mappings can be : (i) bound several times (e.g., in star queries), (ii) bound partially as a side-effect of LIMIT and FILTER clauses, (iii) or bound into a different concurrent query.

As a real log can be huge, LIFT analyzes the log on a sliding window defined by a *gap*, i.e., a time interval. When LIFT reads an entry e in the log with a timestamp ts, it considers only entries reachable within the gap i.e., $ts \pm gap$. Algorithm 1 shows the three phases of LIFT.

Algorithm 1: Global algorithm of LIFT

```
1 Function LIFT(log, gap) is
    input  : a TPF server log; a gap in time units (seconds)
    output: a set of BGPs
    data   : CTP a list of ctps, DTP a graph of dtps
2   CTP ← ctpExtraction (log, gap)
3   DTP ← nestedLoopDetection (CTP, gap)
4   return BGP ← bgpExtraction(DTP)
```

1. First, LIFT **merges** triple pattern queries having same characteristics into *candidate triple patterns (ctp)*. This allows to gather triple pattern queries that seem to be part of the same inner loop.
2. Next, LIFT looks for an inclusion relationship among output-mappings and input-mappings of ctps. If such an inclusion exists a *deduced triple pattern (dtp)* is created. If instead of inclusion an intersection exists, LIFT **splits** ctps to obtain a dtp with inclusion. If neither inclusion nor intersection exists an isolated dtp is created. This produces a DTP Graph where nodes are dtps and edges are inclusion relationships between dtps.

3. Finally, LIFT extracts BGPs from the DTP graph. Ideally, LIFT($E(Q_3 \parallel Q_4), gap$) should compute the 2 BPGs of Q_3 and Q_4:
$\{?s\ p2\ toto\ .\ ?s\ p1\ ?o\}, \{?s\ p3\ titi\ .\ ?s\ p1\ ?o\ .\ ?s\ p4\ tata\}$.

Section 3.1 details the CTP extraction. Section 3.2 describes the nested-loop detection. Finally, Section 3.3 presents the phase of extraction of BGPs.

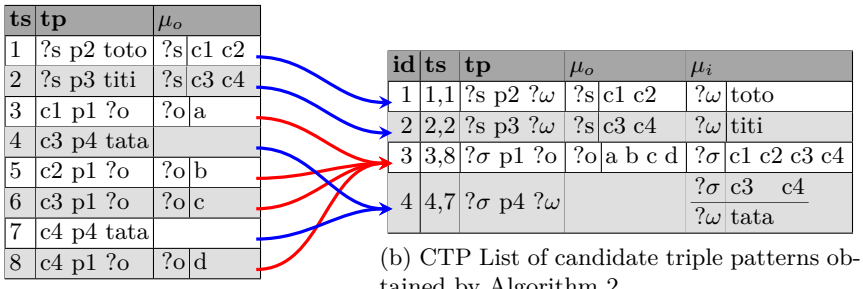
3.1 Extraction of candidate triple patterns

The objective of *ctpExtraction* is to aggregate together log entries that seem to participate in the same outer or inner loop. Aggregated entries are ctps. A ctp is a tuple $\langle ip, ts, tp, \mu_o, \mu_i \rangle$ where ip is an IP address, ts is a pair of timestamps $(s.min, ts.max)$ representing a range; when creating a ctp both timestamps are identical and correspond to the timestamp of the corresponding entry in the log. tp is a triple pattern query, μ_o (output-mappings) is the list of solution mappings for variables of tp. μ_i (input-mappings) is a set of mappings built during the *ctpExtraction*. Basically, we replace any constant of tp by a variable, we use σ for subject and ω for object. Replaced constants are regrouped in μ_i.

Algorithm 2: Extraction of Candidate Triple Patterns

```
1  Function ctpExtraction (log, gap) is
       input  : a TPF server log; a gap in time units (seconds)
       output: a list CTP of ctp
2      CTP ←[ ]
3      foreach e ∈ log do
4          c ← read(e) as (ip,(ts,ts), tp, μo, μi) switch c.tp do
5              case ?s p o: do c.tp ← ?s p ?oin ; c.μi ← ?ω|o
6              case s p ?o: do c.tp ← ?sin p ?o ; c.μi ← ?σ|s
7              case s p o: do c.tp ← ?sin p ?oin ; c.μi ← ?σ|s, ?ω|o
8          if ∃ ck ∈ CTP / ingap(c,ck,gap) ∧ ck.ip = c.ip ∧ c.tp = ck.tp then
9              ck. μo ∪ c. μo ; ck. μi ∪ c. μi ; ck.ts.max=c.ts.max;
10         else CTP.add(c)
11     return CTP
```

Algorithm 2 outlines the extraction of a CTP List from a TPF log with a particular *gap*. Figure 3 illustrates the effect of executing Algorithm 2 on log $E(Q_3 \parallel Q_4)$ with gap=8. The log is processed in sequential order. Lines 5 to 7 initialize input-mappings by replacing constants by variables σ or ω. Next, Lines 9 to 10 merge (i.e., aggregate) current ctp with an existing and compatible one. An existing ctp is compatible if it has the same tp, it is produced by the same ip address, and fits in the gap. The $ingap(c, c_k, gap)$ function returns true if $c.ts.min - c_k.ts.max \leq gap$. If the current ctp is compatible with an existing one, output/input-mappings and timestamps are merged. When updating timestamps, the lower timestamp remains always the same, only the upper timestamp can grow. A variable of tp cannot belong to μ_o and μ_i simultaneously.



ts	tp	μ_o
1	?s p2 toto	?s c1 c2
2	?s p3 titi	?s c3 c4
3	c1 p1 ?o	?o a
4	c3 p4 tata	
5	c2 p1 ?o	?o b
6	c3 p1 ?o	?o c
7	c4 p4 tata	
8	c4 p1 ?o	?o d

(a) Log of a TPF server.

id	ts	tp	μ_o	μ_i
1	1,1	?s p2 ?ω	?s c1 c2	?ω toto
2	2,2	?s p3 ?ω	?s c3 c4	?ω titi
3	3,8	?σ p1 ?o	?o a b c d	?σ c1 c2 c3 c4
4	4,7	?σ p4 ?ω		?σ c3 c4 ?ω tata

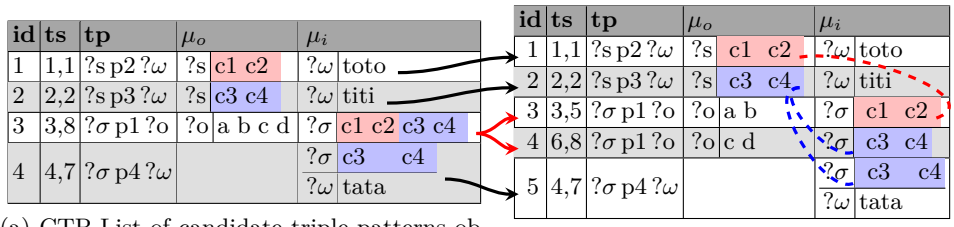
(b) CTP List of candidate triple patterns obtained by Algorithm 2.

Fig. 3: TPF log and CTP List produced by Algorithm 2 with $E(Q_3 \parallel Q_4)$ and $gap = 8$.

This algorithm can aggregate triple patterns that do not belong to the same nested-loop as it is the case in our example of Figure 3, where CTP[3] aggregates triple patterns of Q_3 and Q_4. We suppose that this case is not likely, especially when the gap is small but if it is the case, next algorithm splits ctps to separate these nested-loops.

3.2 Nested-loop join detection

Algorithm 3 describes how to link variables of different ctps produced by Algorithm 2. It builds a DTP Graph of *deduced triples patterns (dtp)* where nodes have the same structure as ctps and edges represent a relation of inclusion between input-mappings (μ_i) and output-mappings (μ_o) of 2 different dtps. Figure 4b presents the DTP Graph produced by Algorithm 3 with the CTP List of Figure 4a. Dashed links represent linked variables deduced by Algorithm 3.



id	ts	tp	μ_o	μ_i
1	1,1	?s p2 ?ω	?s c1 c2	?ω toto
2	2,2	?s p3 ?ω	?s c3 c4	?ω titi
3	3,8	?σ p1 ?o	?o a b c d	?σ c1 c2 c3 c4
4	4,7	?σ p4 ?ω		?σ c3 c4 ?ω tata

(a) CTP List of candidate triple patterns obtained by Algorithm 2.

id	ts	tp	μ_o	μ_i
1	1,1	?s p2 ?ω	?s c1 c2	?ω toto
2	2,2	?s p3 ?ω	?s c3 c4	?ω titi
3	3,5	?σ p1 ?o	?o a b	?σ c1 c2
4	6,8	?σ p1 ?o	?o c d	?σ c3 c4
5	4,7	?σ p4 ?ω		?σ c3 c4 ?ω tata

(b) DTP Graph of deduced triple patterns obtained by Algorithm 3.

Fig. 4: CTP List and DTP Graph produced by Algorithm 3 with $gap = 8$.

If the μ_i of a ctp is a subset of the μ_o of a previous ctp, then we consider that the 2 corresponding variables can be linked. This happens in the example described in Figure 4a with CTP[2] and CTP[4]. We consider that ?σ of CTP[4] is linked to ?s of CTP[2]. We formalize this behavior at Lines 6 to 7 of Algorithm 3.

Algorithm 3: Detection of nested-loop joins

```
1  Function nestedLoopDetection (CTP, gap) is
       input  : a list CTP of ctps; a gap in time units (seconds)
       output: An edge-labelled DTP Graph of dtps
2      foreach c ∈ CTP do
3          if split(c) ≠ ∅ then CTP.insertAfter(c.id,split(c));
4          else DTP.addnode(c); foreach v_o ∈ vars(c.μ_o) do
5              foreach (c_k, v_i) ∈ { (c_k, v_i) / c_k ∈ CTP ∧ c_k.id > c.id ∧
                 ingap(c_k,c,gap) ∧ ∃ v_i ∈ vars(c_k.μ_i) / c_k.μ_i(v_i) ∩ c.μ_o(v_o) ≠ ∅ } do
6                  if c_k.μ_i(v_i) ⊆ c.μ_o(v_o) then
7                      DTP.addnode(c_k) ; DTP.addEdge(c,c_k,(v_o,v_i));
8                  else DTP.addnode(s=split(c_k, v_i, c, v_o)) ; DTP.addEdge(c,s,(v_o,v_i));
9
10     return DTP;
```

A direct inclusion does not occur if Algorithm 2 aggregated too many log entries as it is the case with CTP[3]. Q_3 and Q_4 have a common triple pattern $\langle ?x\, p1\, ?y \rangle$ and Algorithm 2 aggregates them. We solve this problem by splitting a ctp. The idea is to produce a dtp from a ctp c_k if it exists an intersection between the μ_o of a ctp c_l and the μ_i of c_k. In the example described in Figure 4a, CTP[3] produces two splits: one when analyzing CTP[1] (DTP[3] is produced) and another when analyzing CTP[2] (DTP[4] is produced) because both μ_o intersect the μ_i of CTP[3]. Splitting affects input-mappings, but also timestamps and output-mappings. After splitting, we obtain input-mappings that are subsets of previous output-mappings. Intersection and splitting is shown in Lines 5 and 8 of Algorithm 3.

For lack of space, we do not detail the function split. It basically groups, from the TPF log, values that belong to the intersection, this generates correct timestamps, output-mappings and input-mappings. We register the split relationship with a *split* predicate that links a ctp with its produced dtps. In our example, we have 2 *split* relations; $split(3,3)$ and $split(3,4)$. Splitting has an effect on CTP traversal that we see in Line 3. Output-mappings of produced dtps must be analyzed so when the nested-loop detection analyzes a splitted ctp, it inserts in the CTP List, produced dtps. $split(c)$ returns the set of dtps produced by splitting c.

3.3 BGP Extraction

Figure 5 represents the connected components of the DTP Graph shown in Figure 4b. From this representation, it is easy to compute the final BGPs with a variable renaming and restitution of an IRI/literal in place of ω when there is only one input mapping, e.g., $toto, titi$ and $tata$. Detected joins are underlined.

In our example, LIFT rebuilds perfectly BGPs of queries Q_3 and Q_4. This example is executed with $gap = 8$. If we reduce the gap, then some joins are not

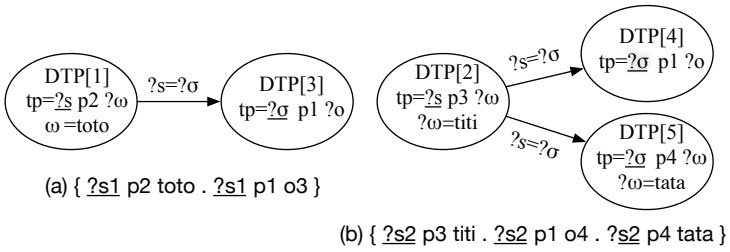


Fig. 5: Connected components of the DTP Graph produced by the execution of Algorithm 3 for $gap = 8$.

detected and recall decreases. If we execute concurrently more queries having same triple patterns, then LIFT can deduce joins that do no exist in original queries and consequently precision will decrease. In Section 4, we measure experimentally the precision and recall of LIFT in different situations.

4 Experiments

The goals of the experiments are twofold, (i) to evaluate precision and recall of LIFT's results and (ii) to show that LIFT extracts meaningful BGPs from a real TPF log.

First, we analyze a small set of queries taken from the TPF web site. We evaluate precision and recall of LIFT deductions from logs of queries executed *in isolation* in Section 4.1, and from logs of queries executed *concurrently* in Section 4.2. These two sections evaluate to which extent LIFT guarantees Properties 1 and 2 of our problem statement (cf. Definition 4), correspondingly.

Then, we analyze LIFT with an important number of real user queries taken from logs of USEWOD 2016 [5]. In Section 4.3, we evaluate precision and recall of 14,259 queries coming from logs of the DBpedia's SPARQL endpoint. In Section 4.4, we analyze 4,720,874 single triple pattern queries coming from logs of the DBpedia's TPF server.

4.1 Evaluation of LIFT with queries in isolation

We used 29 over 30 queries of the TPF web site[4]. Concerned datasets are DBpedia 2015-04, UGhent, LOV or VIAF. We captured http requests and answers of queries using the *webInspector 1.2* tool[5]. Source code of LIFT is available on GitHub[6] For each query Q_i, we run LIFT$(E(Q_i), \infty)$. Figure 6 presents precision and recall of LIFT deductions against original queries[7], they show to which

[4] `http://client.linkeddatafragments.org/`
[5] `https://sourceforge.net/p/webinspector/wiki/Home/`
[6] `https://github.com/coumbaya/lift`
[7] Queries, TPF logs and LIFT results are available at:
`https://github.com/coumbaya/lift/blob/master/experiments.md`

extent $\mathsf{LIFT}(E(Q_i)) \approx BGP(Q_i)$ (cf. Definition 4, Property 1). In average, LIFT obtained 97% of recall and 75% of precision of joins. That gives a quality of 84.5%. LIFT deduces perfectly 15 of 30 BGPs: $Q_1 - Q_6$, Q_9, Q_{11}, $Q_{15} - Q_{18}$, Q_{22}, and $Q_{29} - Q_{30}$. Q_{28} was not analyzed because it needs two TPF servers.

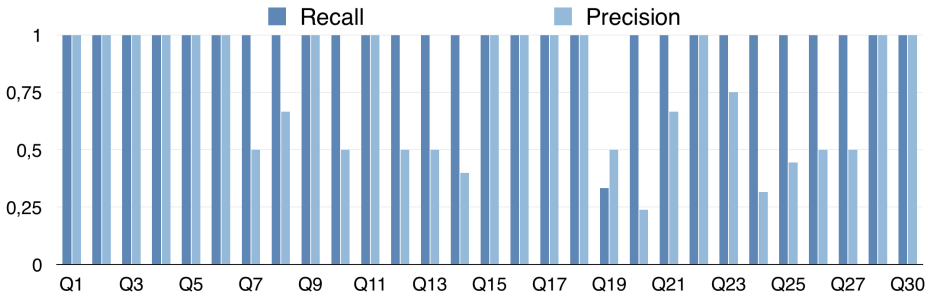


Fig. 6: Precision and recall by query of the TPF web site.

LIFT does not detect UNION queries because they are processed on the client side. Q_9 is a query with a BGP like {(tp1 UNION tp2) . tp3}. In this case, LIFT detects 2 BGPs, {tp1 . tp3} and {tp2 . tp3}. Q_{29} is also a UNION query but without join, as there is no intersection between mappings, LIFT detects two separate BGPs containing one triple pattern. We consider this behaviour correct.

Figure 7 describes Q_7 and its deduced BGPs. BGP[1] is correct, while BGP[2] is not. This is due to traces of the first request made by TPF clients for each triple pattern of each query to decide the join ordering (TPF servers send cardinalities of concerned triples in their answers). Thus, when processing Q_7, the client makes a first request of each triple pattern and then decides to begin with the first triple pattern. Then it binds resulting mappings into the ?*book* variable of the second triple pattern to retrieve corresponding authors. This nested-loop is deduced in BGP[1]. But as output mappings of the first request of the second triple pattern intersects with the values of the inner loop, LIFT deduces BGP[2] with a self-join that is very unlikely and that can be easily filtered in a post-processing. Such situation appears in 6 of 29 queries: Q_7, Q_{12-14}, Q_{21} and Q_{25}.

In some cases, LIFT deduces additional triple patterns and thus false joins with well deduced triple patterns because of the intersection of mappings. This is more challenging to filter. In Figure 7, Q_8 has an additional triple pattern, the last one, and a join with the second triple pattern. This is the case for Q_8, Q_{10}, Q_{14}, Q_{20}, Q_{23-27}.

In addition, LIFT merges triple patterns when they are very similar as it is the case in Q_{19} and Q_{20} where some triple patterns have same predicate and variables in the same position (subject/object).

ID	Original query	Deduced BGPs
Q_7	**SELECT DISTINCT** ?book ?author **WHERE** { ?book rdf:type dbpo:Book . ?book dbpo:author ?author } **LIMIT** 100	BGP[1]: {?s1 rdf:type dbpo:Book . ?s1 dbpo:author ?o2} BGP[2]: {?s3 dbpo:author ?o3 . ?s3 dbpo:author ?o4}
Q_8	{**SELECT** ?award **WHERE** { ?award **a** dbpedia−owl:Award . ?award dbpprop:country ?language . ?language dbpedia−owl:language dbpedia:Dutch_language}	{?s1 dbpedia−owl:language dbpedia:Dutch_language . ?s2 dbpprop:country ?s1 . ?s2 rdf:type dbpedia−owl:Award . ?s1 rdf:type dbpedia−owl:Award}

Fig. 7: LIFT deductions for Q_7 and Q_8. Prefix dbpo corresponds to dbpedia-owl.

4.2 Does LIFT results resist to concurrency?

We implemented a tool to shuffle several TPF logs according to different parameters. Thus, given $E(Q_1), ..., E(Q_n)$, we are able to produce different significant representations of $E(Q_1 \parallel ... \parallel Q_n)$. We grouped queries targeting the same dataset into a set of randomly chosen queries as shown in Table 2. For each query set, we evaluate how $\mathsf{LIFT}(E(Q_1), gap) \cup ... \cup \mathsf{LIFT}(E(Q_n), gap) \approx \mathsf{LIFT}(E(Q_1 \parallel ... \parallel Q_n), gap)$ in terms of precision and recall for different gap values (cf. Definition 4, Property 2). *Gap* varies from 1% to 100% of the log duration. Each query set was shuffled 4 times and we calculate the average of results by gap. In Figure 8 we report results by join, (a) and (b) show precision whereas (c) and (d) show recall.

Dataset	**Query sets**
DBpedia 2015	$DB_1 = \{Q_1, Q_8, Q_{14}, Q_{22}\}$ $DB_4 = \{Q_4, Q_{12}, Q_{24}\}$ $DB_2 = \{Q_3, Q_{11}, Q_{15}, Q_{20}\}$ $DB_5 = \{Q_7, Q_{16}, Q_{21}, Q_5\}$ $DB_3 = \{Q_6, Q_{13}, Q_{19}, Q_{27}\}$ $DB_6 = \{Q_9, Q_{10}, Q_{29}, Q_{30}\}$
UGhent	$UG = \{Q_2, Q_{23}, Q_{25}, Q_{29}, Q_{30}\}$
LOV	$LV = \{Q_{17}, Q_{18}, Q_{26}, Q_{29}, Q_{30}\}$
VIAF	$VF = \{Q_{29}, Q_{30}\}$

Table 2: Query sets executed concurrently over a TPF server.

Precision and recall globally improve when gap increases. When gap is small (less than 50%) precision decreases significantly. A small gap leads LIFT to split values of an inner loop i.e., the *ctpExtraction* algorithm cannot aggregate in one ctp all triple patterns of the inner loop.

Concerning recall, LIFT is moderately impacted by concurrency. In some cases, LIFT favours recall by producing all possible joins in the nested-loop detection.

Concerning precision, LIFT is more impacted by concurrency and results depend on concurrent executed queries. When executed queries have triple patterns that are semantically or syntactically similar, then LIFT generates many false joins that impact precision. A post-processing could filter these false joins.

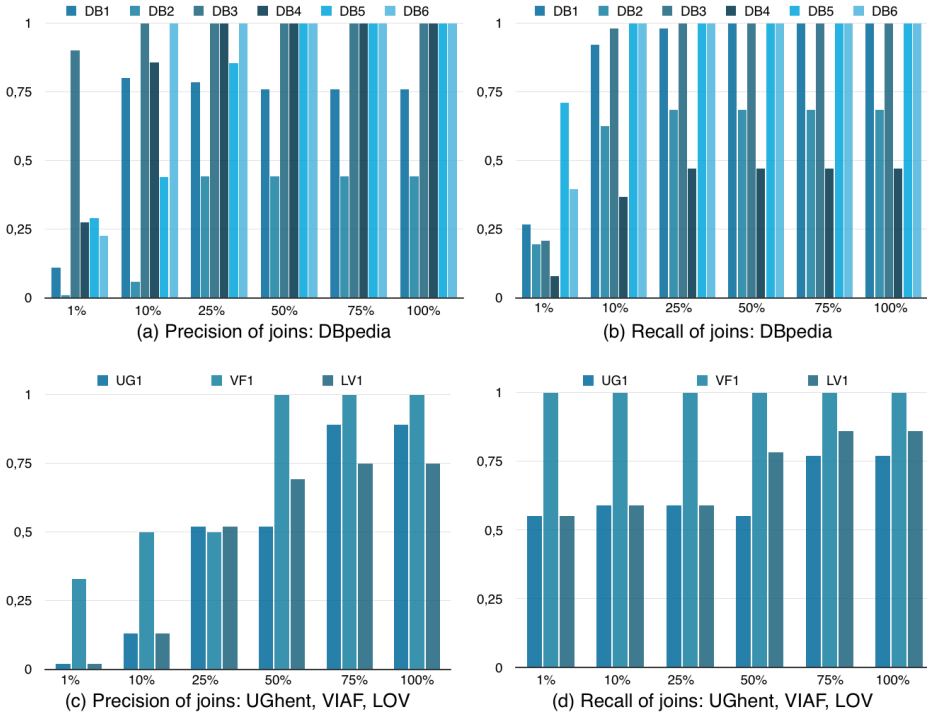


Fig. 8: Precision and recall.

4.3 Evaluation of LIFT with real SPARQL queries

The goal of this experiment is to evaluate to which extent LIFT is able to deduce BGPs of an important number of real user queries. We analyzed 10 hours of one day (2015-10-30) of the log of the DBpedia SPARQL endpoint of USEWOD 2016 dataset [5]. From 380,834 http requests containing SPARQL queries, we analyzed 14,259 queries that represent our ground truth. We kept only SELECT queries having BGPs with more than one triple pattern and that return not null results. We made an aggregation of same queries and kept only one, by hour and by user. That is because, if in a TPF log, the same query appears several times, LIFT deduces only one query, i.e., $\mathsf{LIFT}(E(Q_1) \parallel E(Q_1), gap) \approx BGP(Q_1)$. Then, we constraint to 200 the number of queries by user for each hour. The OPTIONAL

operator was transformed in a JOIN and we filter queries having a join over predicates.[8]

We installed locally a TPF server with DBpedia 3.9 to execute and collect the TPF server logs that were given as input to LIFT. Gap was set to one hour. We calculate precision, recall and quality of deduced BGPs against the ground truth. LIFT returned BGPs in xml files which were loaded in a MySQL database. This allows us to make several analysis.

Globally, we obtained 69% of precision, 64% of recall and 66% of quality. Results are better for recurrent queries (in particular for precision) for which we obtained 85% of precision, 55% of recall and 70% of quality. That means that queries executed frequently under different execution contexts (i.e., concurrent queries) are better deduced than queries requested few times. We analyzed results by user and calculate their averages, we obtained 84% of precision, 74% of recall and 78% of quality. From this analysis, we observed that there are particular users whose queries are less well deduced, but in general LIFT deductions by user are good.[9]

4.4 Analysis of a real TPF log

The goal of this experiment is to extract BGPs from the log of the DBpedia's LDF server of USEWOD 2016 dataset [5]. This log contains http requests from October 2014 to November 2015. We analyzed the first quarter of the log representing 4,720,874 single triple pattern queries (until 27th February 2015). We pruned 1% of the log corresponding to entries that are not well formed TPF requests. LIFT needs not only received triple pattern queries but also corresponding answers. To obtain answers, we executed the triple pattern queries of the log using a TPF client [10]. Then, we run LIFT with log slices of one hour with a maximum gap (one hour).

We obtained 1236 BGPs containing more than one triple pattern. Table 9 describes the most recurrent queries. Unsurprisingly, some of them are the queries available on the TPF web site. BGP[1] corresponds to Q_1, BGP[2] is like BGP[1] except that *dbpedia-owl:starring* is replaced by *dbpprop:starring*. BGP[3] corresponds to the query used as the motivation example of [13], BGP[4] corresponds to Q_3, BGP[5] to Q_6, etc. In this top 20 list, only few queries were unknown: BGP[6], BGP[7], BGP[8], BGP[12] and BGP[13]. We can analyze deduced BGPs to obtain some statistics, for instance, the most common type of join is subject-subject (2693), followed by subject-object (1063), what is coherent with analysis shown in [2]. LIFT's deductions from the TPF log of USEWOD 2016 are available in an xml document[11].

[8] All scripts to process logs of the DBpedia SPARQL endpoint are in Python and available at `https://github.com/edesmontils/BE4DBPedia/`

[9] These results are available at:
`http://documents.ls2n.fr/lift/dbpedia-sparql-results.tar`

[10] `https://github.com/LinkedDataFragments/Client.js`

[11] `http://documents.ls2n.fr/lift/dbpedia-tpf-results.xml`

BGP[1]- deduced 126 times	BGP[2] - deduced 47 times
{?s_47 rdfs:label ?o_51 . ?s_47 dbpo:director ?o_52 . ?s_46 rdfs:label "Brad Pitt"@en . ?s_47 dbpo:starring ?s_46 . ?o_52 rdfs:label ?o_53 .}	{?s_8 dbpprop:starring ?s_7 . ?s_8 rdfs:label ?o_12 . ?s_7 rdfs:label "Brad Pitt"@en . ?o_13 rdfs:label ?o_14 . ?s_8 dbpo:director ?o_13 .}
BGP[3] - deduced 43 times	BGP[4] - deduced 34 times
{?s_10 rdfs:label "York"@en . ?s_11 rdf:type dbpo:Artist . ?s_11 dbpo:birthPlace ?s_10 .}	{?s_6 dbpo:birthDate ?o_8 . ?s_6 dbpo:influencedBy dbpedia:Pablo_Picasso . ?s_6 rdf:type dbpo:Artist .}
BGP[5] - deduced 34 times	BGP[6] - deduced 20 times
{?s_9 dbpprop:cityServed dbpedia:Italy . ?s_9 rdf:type dbpo:Airport .}	{dbpo:Agent rdfs:subClassOf ?o_13 . ?o_13 rdfs:subClassOf ?o_14 .}
BGP[7] - deduced 17 times	BGP[8] - deduced 16 times
{dbpo:Activity rdfs:subClassOf ?o_29 . ?o_29 rdfs:subClassOf ?o_30 .}	{?s_283 rdf:type dbpo:Writer . ?s_282 rdfs:label "Trinity College, Dublin"@en . ?s_283 dbpo:almaMater ?s_282 .}
BGP[9] - deduced 15 times	BGP[10] - deduced 13 times
{?s_17 rdf:type dbpo:Book . ?s_17 dbpo:author ?o_18 .}	{?s_20 dbpo:team ?o_21 . ?s_20 dbpo:birthPlace dbpedia:Urbel_del_Castillo .}
BGP[11] - deduced 11 times	BGP[12] - deduced 11 times
{?s_33 dbpo:ingredient ?o_33 . ?s_33 dbpo:kingdom dbpedia:Plant .}	{?s_20 rdf:type yago:Carpenter . ?s_20 rdf:type yago:PeopleExecutedByCrucifixion .}
BGP[13] - deduced 10 times	BGP[14] - deduced 10 times
{?s_2 foaf:isPrimaryTopicOf ?o_3 . ?s_2 rdf:type foaf:Person .}	{?s_35 dbpo:ingredient ?o_36 . ?s_35 dbpo:type dbpedia:Dessert . ?o_36 dbpo:kingdom dbpedia:Plant .}
BGP[15] - deduced 9 times	BGP[16] - deduced 8 times
{?s_18 dbpo:team ?s_15 . ?s_15 dbpo:ground dbpedia:Urbel_del_Castillo . ?s_18 rdfs:label ?o_22 . ?s_15 rdf:type dbpo:SoccerClub . ?s_15 rdfs:label ?o_17 .}	{?s_23 dbpprop:starring dbpedia:Brad_Pitt . ?s_23 rdfs:label ?o_24 . ?s_23 dbpo:director ?o_25 . ?o_25 rdfs:label ?o_26 .}
BGP[17] - deduced 8 times	BGP[18] - deduced 8 times
{?s_18 dbpo:developer ?s_16 . ?s_15 rdfs:label "Belgium"@en . ?s_16 dbpo:locationCountry ?s_15 .}	{dbpedia:Raspberry_Pi dbpo:operatingSystem ?o_16 . ?s_17 dbpo:operatingSystem ?o_16 . ?s_17 rdf:type dbpo:Device .}
BGP[19] - deduced 7 times	BGP[20] - deduced 7 times
{?s_6 rdfs:label ?o_7 . ?s_6 dbpprop:starring dbpedia:Natalie_Portman .}	{dbpedia:Jesus dc:subject ?o_28 . ?s_29 dc:subject ?o_28 .}

Fig. 9: Recurrent BGPs extracted from the TPF log of USEWOD 2016.

5 Related Work

Compared to the state of art, [14] reports statistics on TPF logs that allow to verify server availability, number of evaluated requests and cache usage. These statistics are useful but can be obtained at triple pattern granularity. LIFT, is able to extract BGPs from TPF logs, that can be subsequently analyzed to retrieve other information, e.g., frequently executed BGPs as done in Sections 4.3 and 4.4.

Extracting information from logs is traditionally a data mining process [3]. Sequential pattern mining [7] focuses on discovering frequent subsequences from an ordered sequence of events. However, searching for BGPs cannot be reduced to searching for frequent episodes in a TPF log. Suppose, two queries Q_1 : $\{?x\ p1\ o1\ .\ ?x\ p2\ ?y\}$ and Q_2 : $\{?x\ p1\ ?y\ .\ ?y\ p3\ ?z\}$. The TPF query engine executes the joins with a nested-loop. So, $?x\ p1\ o1$ and $?x\ p1\ ?y$ will appear once in the log, while patterns with $IRIs\ p2\ ?y$ and $IRIs\ p3\ ?z$ will appear many times according to the selectivity of the triple patterns on $p1$. Searching for frequent episodes will raise up episodes with $p2$ and $p3$ but joins were between $p1, p2$ and $p1, p3$. Clearly, TPF logs need to be pre-processed before analysis.

In previous work [9], we focused on analyzing SPARQL logs. We proposed FETA, an algorithm to reverse BGPs of federated SPARQL queries from logs collected from a set of SPARQL endpoints. LIFT and FETA share the problem statement, but have several major differences. First, FETA analyzes a SPARQL log gathered from many data providers in order to infer BGPs of federated queries. LIFT analyzes a single TPF log of a single data provider to infer BGPs of SPARQL queries. This makes possible to run LIFT on real available logs as those published by USEWOD 2016 [5]. Second, the input log of FETA is composed of SPARQL queries that are more complex than TPF queries. On the other hand, SPARQL queries include variable names that can be used to disambiguate some situations. In TPF logs, such variables are not available. Next, TPFs contain metadata that helps clients to decide a join ordering. To obtain the cardinality of a triple pattern, a client actually requests the triple pattern making indistinguishable these requests from regular query processing. This makes BGP extraction from TPF logs more challenging.

6 Conclusions and future work

LIFT extracts BGPs from TPF server logs. It tracks joins executed with nested-loops following a strategy of merging and splitting compatible triple pattern queries. LIFT deductions have good precision and good recall mainly when evaluating frequent queries. The main challenge for LIFT is the concurrent execution of queries from the same client targeting exactly the same triples (queries sharing IRIs).

LIFT opens several perspectives. First, LIFT focused on nested-loops. Clients can also rely on symmetric hash to perform joins that can also be detected. Detecting nested-loop and symmetric hash together from the server-side is challenging. Second, in LIFT, we favoured recall, mainly by splitting input mappings

for every intersection with output mappings. We can introduce more constraints for splitting that should improve precision against recall, e.g., to separate repeated executions of identical queries which traces appear in the same TPF. Third, LIFT is currently an algorithm that post-processes logs. We can transform LIFT into a streaming algorithm able to extract BGPs in real-time. This will also allow to process very large TPF logs. Finally, LIFT is able to process logs on different datasets by just merging them. In this case, one data provider is able to detect cross-join against different datasets hosted on the same server. If different data providers are ready to collaborate by just sharing their logs, they should be able to infer federated queries running across multiple servers.

References

1. W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach. LOD Laundromat: A Uniform Way of Publishing Other People's Dirty Data. In *ISWC Conference*, 2014.
2. M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An Empirical Study of Real-World SPARQL Queries. In *USEWOD workshop*, 2011.
3. J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Elsevier, 2011.
4. J. Lorey and F. Naumann. Detecting SPARQL Query Templates for Data Prefetching. In *ESWC Conference*, 2013.
5. L.-R. Markus, A. Saud, B. Bettina, and H. Laura. USEWOD Research Dataset., 2016. http://dx.doi.org/10.5258/SOTON/385344.
6. K. Möller, M. Hausenblas, R. Cyganiak, G. Grimnes, and S. Handschuh. Learning from Linked Open Data Usage: Patterns & Metrics. In *WebSci10:Extending the Frontiers of Society On-Line*, 2010.
7. C. H. Mooney and J. F. Roddick. Sequential Pattern Mining–Approaches and Algorithms. *ACM Computing Surveys (CSUR)*, 45(2):19, 2013.
8. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL Benchmark–Performance Assessment with Real Queries on Real Data. In *ISWC Conference*, 2011.
9. G. Nassopoulos, P. Serrano-Alvarado, P. Molli, and E. Desmontils. FETA: Federated QuEry TrAcking for Linked Data. In *DEXA Conference*, 2016.
10. F. Picalausa and S. Vansummeren. What are Real SPARQL Queries Like? In *SWIM Workshop*, 2011.
11. A. Raghuveer. Characterizing Machine Agent Behavior through SPARQL Query Mining. In *USEWOD Workshop*, 2012.
12. L. Rietveld, R. Hoekstra, et al. Man vs. Machine: Differences in SPARQL queries. In *USEWOD Workshop*, 2014.
13. M. V. Sande, R. Verborgh, J. V. Herwegen, E. Mannens, and R. V. de Walle. Opportunistic Linked Data Querying Through Approximate Membership Metadata. In *ISWC Conference*, 2015.
14. R. Verborgh, E. Mannens, and R. Van de Walle. Initial Usage Analysis of DBpedia's Triple Pattern Fragments. In *USEWOD Workshop*, 2015.
15. R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. *Journal of Web Semantics*, 37–38, Mar. 2016.