FETA: Federated QuEry TrAcking for Linked Data

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

▶ **To cite this version:**

Georges Nassopoulos, Patricia Serrano-Alvarado, Pascal Molli, Emmanuel Desmontils. FETA: Federated QuEry TrAcking for Linked Data. International Conference on Database and Expert Systems Applications (DEXA), Sep 2016, Porto, Portugal. pp.0. hal-01336386

HAL Id: hal-01336386

https://hal.science/hal-01336386v1

Submitted on 23 Jun 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

FETA: Federated QuEry TrAcking for Linked Data

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

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

Abstract. Following the principles of Linked Data (LD), data providers are producing thousands of interlinked datasets in multiple domains including life science, government, social networking, media and publications. Federated query engines allow data consumers to query several datasets through a federation of SPARQL endpoints. However, data providers just receive subqueries resulting from the decomposition of the original federated query. Consequently, they do not know how their data are crossed with other datasets of the federation. In this paper, we propose FETA, a Federated quEry TrAcking system for LD. We consider that data providers collaborate by sharing their query logs. Then, from a federated log, FETA infers Basic Graph Patterns (BGPs) containing joined triple patterns, executed among endpoints. We experimented FETA with logs produced by FedBench queries executed with Anapsid and FedX federated query engines. Experiments show that FETA is able to infer BGPs of joined triple patterns with a good precision and recall.

Keywords: Linked data, federated query processing, log analysis, usage control.

1 Introduction

Linked Data (LD) interlinks massive amounts of data across the Web in multiple domains like life science, government, social networking, media and publications. Federated query engines [1–3, 5, 9, 11] allow data consumers to execute SPARQL queries over a decentralized federation of SPARQL endpoints maintained by LD providers. But, data providers are not aware of users' federated queries; they just observe subqueries they receive. Thus, they do not know when and which datasets are joined together in a single query. Consequently, the federation does not hold enough meta-information to ensure services, such as, efficient materialization to improve joins, activation of query optimization techniques, discovering data providers partnership, etc. Knowing how provided datasets are queried together is essential for tuning endpoints, justify return of investment or better organize collaboration among providers.

A simple solution for this problem is to consider that data consumers publish their federated queries. However, public federated queries cannot be considered

as representative of real data usage because they may represent a small portion of really executed queries. Only logs give evidences about real execution of queries.

Thus, in this paper, we address the following problem: if data providers share their logs, can they infer the Basic Graph Patterns (BGP) of federated queries executed over their federation? Many works have focused on web log mining [6, 7], but none has addressed reversing BGPs from a federated query log.

The main challenge is the concurrent execution of federated queries. If we find a function f, to reverse BGPs from isolated traces of one federated query, is f able to reverse the same BGP from traces of concurrent federated queries?

We propose FETA to implement f, a Federated quEry TrAcking system that computes BGPs from a federated log. Based on subqueries contained in the log, FETA deduces triple patterns and joins among triple patterns with a good precision and recall. Our main contributions are:

1. the definition of the problem of reversing BGPs from a federated log,
2. the FETA algorithm to reverse BGPs from federated logs,
3. an experimental study using federated queries of the benchmark FedBench[1]. From execution traces of these queries, FETA deduces BGPs under two scenarios, queries executed in isolation and in concurrence.

The paper is organized as follows. Section 2 introduces a motivating example and our problem statement. Section 3 presents FETA and its heuristics. Section 4 reports our experimental study. 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 data consumers, C_1 and C_2, execute concurrently federated queries $CD3$ and $CD4$ of FedBench. They use Anapsid or FedX federated query engines to query a federation of SPARQL endpoints composed of *LMDB, DBpedia InstanceTypes (IT), DBpedia InfoBox (IB)* and *NYTimes (NYT).* Data providers hosting these endpoints receive only subqueries corresponding to the execution of physical plans of $CD3$ and $CD4$. For example, $CD3$ can be decomposed into $\{tp_1^{@IT}.(tp_2.tp_3)^{@IB}.(tp_4.tp_5)^{@NYT}\}$, and NYT just observes tp_4 and tp_5: it does not know these triple patterns are joined with tp_1 from IT and (tp_2, tp_3) from IB. So, NYT does not know the real usage of data it provides.

More formally, we consider that an execution of a federated query FQ_i produces a partially ordered sequence of subqueries SQ_i represented by $E(FQ_i) = [SQ_1, ..., SQ_n]$. Subqueries are processed by endpoints of the federation at given times represented by timestamps. We suppose that endpoints' clocks are synchronized, i.e., timestamps of logs can be compared safely. Timestamps of subqueries in a federated log are partially ordered because two endpoints can receive queries at same time. Query execution with a particular federated query engine,

[1] `http://fedbench.fluidops.net/`

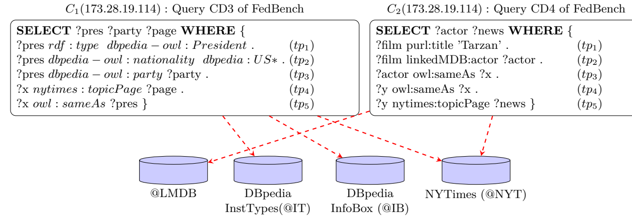

Fig. 1: Concurrent execution of FedBench queries CD3, CD4 over a federation of endpoints.

qe_i, is represented by $E_{qe_i}(FQ_i)$. In addition, we represent a concurrent execution of n federated queries by $E(FQ_1 \parallel ... \parallel FQ_n)) = [SQ_1, ..., SQ_n]$. This work addresses the following research question: *if data providers share their logs, can they rebuild the BGPs annotated with the sources that evaluated each triple pattern?* From the previous example, we aim to extract two BGPs: one corresponding to $CD3$ $\{tp_1^{@IT}.(tp_2.tp_3)^{@IB}.(tp_4.tp_5)^{@NYT}\}$ and another to $CD4$ $\{(tp_1.tp_2.tp_3)^{@LMDB}.(tp_4.tp_5)^{@NYT}\}$. Next definitions formalize this problem.

Definition 1 (BGPs' reversing). *Given a federated log corresponding to the execution of one federated query $E(FQ_i)$, find a function $f(E(FQ_i))$ producing a set of BGPs $\{BGP_1, ..., BGP_n\}$, where each triple pattern is annotated with endpoints that evaluated it, such that $f(E(FQ_i))$ approximates ($\approx$) the BGPs existing in the original federated query. Thus, if we consider that $BGP(FQ_i)$ returns the set of BGPs of FQ_i then $f(E(FQ_i)) \approx BGP(FQ_i)$.*

In our motivating example, if C_1 and C_2 have different IP addresses, then it is straightforward to apply the reversing function separately on each execution trace. However, in the worst case, if they share the same IP address, we expect that $f(E(CD3 \parallel CD4)) \approx f(E(CD3) \cup f(E(CD4)$ as defined next.

Definition 2 (Resistance to concurrency). *The reversing function f should guarantee that BGPs obtained from execution traces of* isolated *federated queries, approximate ($\approx$) results from execution traces of* concurrent *federated queries: $f(E(FQ_1)) \cup ... \cup (f(E(FQ_n)) \approx f(E(FQ_1 \parallel ... \parallel FQ_n))$.*

3 FETA, a reversing function

Finding a reversing function f requires to join IRIs, literals or variables from different SPARQL subqueries. We propose FETA as a system of heuristics to implement the reversing function f. Figures 2a, 2b present two endpoints, each providing some triples. Figure 2d has the federated log corresponding to the execution of queries $Q_1 = SELECT\ ?z\ ?y\ WHERE\ \{?z\ p1\ o1\ .\ ?z\ p2\ ?y\}$ and $Q_2 = SELECT\ ?x\ ?y\ WHERE\ \{?x\ p1\ ?y\}$. Figure 2c shows reversing results according to different *gap* values described below. Depending on the *gap*, on verifications made, and concurrent traces, reversed BGPs are different. In the example of Figure 2c, if the *gap* has no limit, we obtain the BGP of Line 1, if the

@ep1
s1 p1 o1
s2 p1 o1
s3 p1 o2

(a) EP1

@ep2
s1 p2 o3
s2 p2 o4

(b) EP2

Gap	Reversed BGPs
∞	{ (?x p1 ?y)$^{@ep1}$. (?x p2 ?y)$^{@ep2}$. (?z p1 ?y)$^{@ep1}$. (?z p2 ?y)$^{@ep2}$ }
1	{ ?x p1 ?y }$^{@ep1}$, { ?z p1 o1 }$^{@ep1}$, { s1 p2 ?y }$^{@ep2}$, { s2 p2 ?y }$^{@ep2}$
2	{ ?x p1 ?y }$^{@ep1}$, { (?z p1 o1)$^{@ep1}$. (?z p2 ?y)$^{@ep2}$ }

(c) Deducted BGPs.

Time	Subquery	Set of mappings
$1^{@ep1}$	sq_1={?x p1 ?y}	Ω1={{x, s1}{y, o1},{x, s2}{y, o1},{x, s3}{y, o2}}
$4^{@ep1}$	sq_2={?z p1 o1}	Ω2={{z, s1},{z, s2}}
$6^{@ep2}$	sq_3={s1 p2 ?y}	Ω3={{y, o3}}
$7^{@ep2}$	sq_4={s2 p2 ?y}	Ω4={{y, o4}}

(d) Federated log.

Fig. 2: Motivating example.

gap is 1, only sq_3 and sq_4 can be analyzed together (cf. Line 2). As the join on $?y$ gives no results, a join is discarded. If the *gap* is 2, then the reversed BGPs are the expected ones (cf. Line 3).

We assume pairwise disjoint infinite sets $\mathcal{B}$, $\mathcal{L}$, $\mathcal{I}$ (blank nodes, literals, and IRIs respectively). We also assume an infinite set $\mathcal{S}$ of variables. A mapping μ is a partial, non surjective and non injective, function that maps variables to RDF terms $\mu : \mathcal{S} \rightarrow \mathcal{BLI}$. A set of mappings is represented by Ω. See [8] for more explanations. Next, we present the input and output of FETA.

Given: a federation of endpoints Φ, a federated input log $\mathcal{Q} = \{\langle q, t, ep, ip\rangle\}$, a federated output log $\mathcal{A} = \{\langle\{\mu\}, t, ep, ip\rangle\}$, and a user-defined *gap*,

Find $\mathcal{G} = \{\langle g, ip\rangle\}$, the set of connected graphs corresponding to the BGPs of the federated queries processed by Φ, such that:

- $g = \langle V, E\rangle$ is an undirected connected graph where $V = \{tp\}$ is an unordered set of distinct triple patterns, (annotated with the endpoints that processed tp and $\mathcal{T}$ the set of timestamps given by the endpoints), and E is an unordered set of edges representing the joins between triple patterns.
- ip is the IP address of the client query engine that sent g.

3.1 FETA algorithms

FETA has 4 main phases. From input logs and a predefined *gap*, a graph of subqueries $\mathbb{G}$ is constructed in the first phase. Then, this graph is reduced and transformed into a graph of triple patterns $\mathcal{G}$, where, from a big set of subqueries, frequently only one triple pattern is obtained. In a third phase, joins between triple patterns executed through nested-loops are identified. Finally, symmetric hash joins, possibly made at the federated query engine, are identified. Next sections present these algorithms at high level of abstraction.

Graph Construction: This phase executes two main functions, (a) *LogPreparation* and (b) *CommonJoinCondition*. This module builds $\mathbb{G} = \{g\}$, a set of graphs, where $g = \langle V', E', ip\rangle$ is an undirected connected graph, with different semantics than $\mathcal{G}$. In $\mathbb{G}$, nodes are queries and arcs are labeled with

the number of common variables between each pair of queries. *LogPreparation*, prepares and cleans the input log. ASK queries are suppressed and identical subqueries or differing only in their offset values are aggregated in one single query. Timestamp of such aggregated query becomes an interval. *CommonJoinCondition*, incrementally constructs $\mathbb{G}$, by joining queries depending on the given *gap* and having common projected variables or triple patterns with common IRI or literal on their subjects or objects. In general, subqueries are joined on their common projected variables. However, we consider also IRIs and literals, even if they can produce some false positives. In our example, with an infinite *gap*, two graphs are constructed as shown in Figure 3: $\mathbb{G} = \{g_1, g_2\}$, where $g_1 = \langle\{sq_1, sq_3, sq_4\}, \{(sq_1, sq_3), (sq_1, sq_4), (sq_3, sq_4)\}\rangle$ and $g_2 = \langle\{sq_2\}\rangle$. To simplify, all annotations to sq_i are omitted.

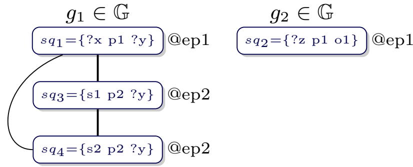

Fig. 3: Deduced graphs in $\mathbb{G}$ after *GraphConstruction*, for an infinite *gap*.

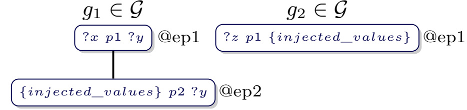

Fig. 4: Deduced graphs in $\mathcal{G}$ after *GraphReduction*, for an infinite *gap*.

Graph reduction. The graph of queries is transformed into a graph of patterns. This heuristic aggregates triple patterns, produced from mappings of the outer dataset towards the inner dataset, into one big aggregated pattern (that we call inner pattern). This pattern, for instance, has the form of $\langle$*injected_values*,*predicate*, *object*$\rangle$, if mappings of the outer dataset are injected into the subject. Graph reduction significantly reduces the size of each $g \in \mathbb{G}$, because nested-loops can be executed with hundreds of subqueries. Figure 4, illustrates $\mathcal{G}$ after the graph reduction phase, for our motivating example.

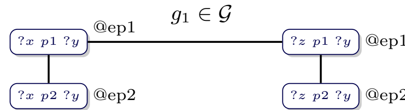

Fig. 5: Deduced graphs in $\mathcal{G}$ after *NestedLoopDetection*, for an infinite *gap*.

Nested-loop detection. This heuristic analyzes existing graphs in $\mathcal{G}$ to identify nested-loops. From n subqueries, it obtains two joined triple patterns by nested-loop. To do this, Algorithm 1, Lines 3-6, associates the pattern that pushes the outer dataset (that we call outer pattern) towards the inner pattern. This association is made by searching for a matching, between the injected values

Algorithm 1: NestedLoopDetection($\mathcal{G}, \mathcal{A}, gap$)

input : $\mathcal{G}, \mathcal{A}, gap$
output: $\mathcal{G}$

1 **foreach** $g \in \mathcal{G}$ **do**
2   **foreach** $tp_i \in g$ **do**
3   **foreach** $(tp_j \in g) \vee (tp_j \in g' : g' \in \mathcal{G}, g' \neq g)$ **do**
4   **if** $(t^{max}_{tp_j} - t^{min}_{tp_i}) \leq gap \wedge (\mu^{-1}(tp_j, \mathcal{A}) \in var(tp_i))$ **then**
5                 $dp \leftarrow$Association(tp_i, tp_j)
6                 $\mathcal{G} \leftarrow$Update$(\mathcal{G}, tp_j, dp, 'nestedLoop')$

of the inner pattern and the variable mappings of the outer, with the function of *inverse mapping* that we propose below.

Definition 3 (Inverse mapping). *We define the inverse mapping as a partial, non surjective and non injective, function* $\mu^{-1} : \mathcal{BLI} \rightarrow \mathcal{S}$ *where* $\mu^{-1} = \{(val, s) \mid val \in \mathcal{BLI}, s \in \mathcal{S}\}$ *such that* $(s, val) \in \mu$. $\mathcal{B}$ *is considered for generalization reasons even if blank nodes cannot be used for joins between datasets.*

$NestedLoopDetection$ is the most challenging heuristic of FETA because μ^{-1} may return more than one variable, when the same value was returned for more than one variable. This depends on the similarity of concurrent federated queries and the considered *gap*. Thus, some times, identifying the variable that appears in the original query is uncertain. Figure 5, illustrates $\mathcal{G}$ after $NestedLoopDetection$ for our motivating example with an infinite *gap*. We observe that graphs g_1 and g_2 are merged.

Symmetric hash detection. This heuristic verifies that *(i)* edges of $g \in \mathcal{G}$ that were not produced by an exclusive group or a nested-loop, are on same ontologically concepts for their common projected variables, and *(ii)* their join has a not null result set. From this, symmetric hash joins are identified, otherwise joins are removed. Symmetric hash detection produces false positives as it infers all possible joins that may be made at the query engine. If a star-shape set of triple patterns exists, all possible combinations of joins will be deduced instead of the subset of joins chosen by the query engine. Consequently, FETA privileges recall to the detriment of precision. For our example, this phase has no impact.

3.2 Time complexity of FETA

The computational complexity of the global algorithm of FETA is, in the worst case, $O(N^2 + N * M + M^2)$, where N is the number of queries of $\mathbb{G}$, and M is the number of triple patterns of $\mathcal{G}$. The overload produced by FETA is high but we underline that the size of the log corresponds to a *sliding window of time* and that the log analysis can be made as a batch processing.

4 Experiments

To the best of our knowledge, a public set of real federated queries executed over the LD does not exists, thus we evaluated FETA using the queries and the setup

of FedBench [10]. We used the collections of Cross Domain (CD) and Life Science (LS), each one has 7 federated queries. We setup 19 SPARQL endpoints using Virtuoso OpenLink[2] 6.1.7. We executed federated queries with Anapsid 2.7 and FedX 3.0. We configured Anapsid to use Star Shape Grouping Multi-Endpoints (SSGM) heuristic and we disabled the cache for FedX. We captured http requests and answers from endpoints with justniffer 0.5.12[3]. FETA is implemented in Java 1.7 and is available at `https://github.com/coumbaya/feta`.

The goals of the experiments are : (i) to evaluate the precision and recall of FETA with federated queries executed *in isolation* and (ii) to evaluate the precision and recall of FETA with federated queries executed *concurrently* under a worst case scenario, i.e., when BGPs of different federated queries cannot be distinguished as they share the same IP address. All results are available at: `https://github.com/coumbaya/feta/blob/master/experiments_with_fedbench.md`.

To analyze traces of federated queries in isolation, we executed CD and LS collections. We captured 28 sequences of subqueries used as input for FETA one by one. In average, we obtained 94,64% of precision and 94,64% of recall of *triple patterns deduction*. We obtained 79,40% of precision and 87,80% of recall for *joins deduction*. Deducing *sets of joined triple patterns*, i.e., BGPs, is more challenging. From Anapsid traces, BGPs deduced correspond to CD and LS queries, except for Union queries, i.e., CD1, LS1 and LS2. These queries have two BGPs but a join is possible between them locally at the query engine, and FETA deduces a symmetric hash join. All other problems of deduction come from *NestedLoopDetection*. False triple patterns are deduced from FedX traces that decreases precision. This is because μ^{-1} may return more than one variable and more than one triple pattern may be deduced. But as right triple patterns are in general well deduced, recall is good. FETA succeeds in deducing 11 out of 14 exact BGPs from Anapsid traces, and 7 out of 14 from FedX traces. It finds 18/28 exact BGPs, i.e., 64%. If we include Union queries where all triple patterns are deduced, FETA finds (18+3)/28 BGPs, i.e., 75% BGPs of FedBench.

To analyze traces of concurrent federated queries, we implemented a tool that shuffles logs of queries executed in isolation to produce different sequences of $E(FQ_1 \parallel ... \parallel FQ_n)$. These traces vary in (i) the order of queries, (ii) the number of subqueries, of the same federated query, appearing continuously (blocks of 1 to 16 subqueries), and (iii) the delay between each subquery (1 to 16 units of time). In our experiments, *gap* varies from 1% to 100% of the total time of each mix. We measured precision and recall of deductions made by FETA, from traces of federated queries in isolation against our mixes of traces of concurrent queries.

If FETA can distinguish triple patterns of concurrent federated queries, precision and recall by join are perfect when the *gap* is big enough. We analyzed a set of chosen queries having distinguishable triple patterns that we named MX: CD3, CD4, CD5, CD6, LS2 and LS3. We produced 4 different mixes of traces of these queries ($M_1, ..., M_4$) that were analyzed by FETA under 6 different *gaps* (1%, 10%, etc.) producing 6 groups of deductions. We obtained 100% of preci-

[2] `http://virtuoso.openlinksw.com/`
[3] `http://justniffer.sourceforge.net/`

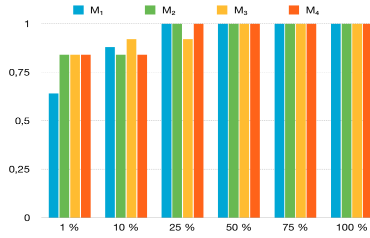

Fig. 6: Recall of joins from ANAPSID MX traces, by *gap*.

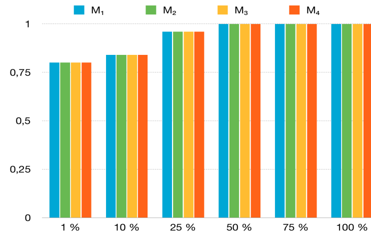

Fig. 7: Recall of joins from FedX MX traces, by *gap*.

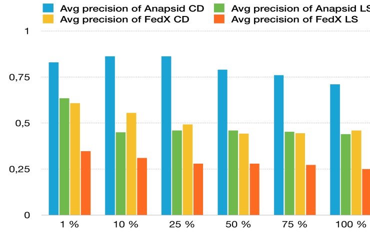

Fig. 8: Average of precision of joins, for four mixes by *gap*.

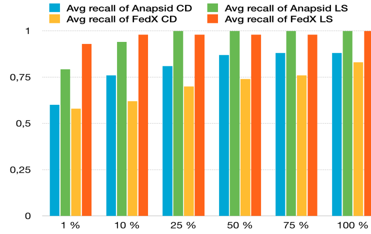

Fig. 9: Average of recall of joins, for four mixes by *gap*.

sion of joins from traces of Anapsid and FedX since the smallest *gap*. Figures 6 and 7 show recall of joins from Anapsid and FedX traces respectively. We get 100% of recall with a *gap* of 50% from traces of both query engines.

If triple patterns of concurrent queries are the same or syntactically similar, it is hard for FETA to obtain good precision and recall of joins. We produced four different and concurrent mixes by queries' collection (4 for CD and 4 for LS). We analyzed them by query engine and by *gap*. Figure 8 shows the average of precision of joins, each bar concerns 4 mixes. We can see that for FETA it is easier to analyze traces from Anapsid than from FedX. Moreover, CD queries are more distinguishable than LS ones. That is because triple patterns of LS queries vary less than those of CD queries, thus it is less evident to separate LS queries from their mixed traces. Furthermore, the bigger the *gap* the smaller the precision. That is because more false joins are detected thus reducing precision. Figure 9 shows the average of recall of joins. In general, recall of LS is bigger than recall of CD because LS queries generate lots of symmetric hash joins including the good ones. Unlike precision, the bigger the *gap*, the bigger the recall because more joins are detected thus the possibility of finding the good ones is bigger.

5 Related Work

Extracting information from logs is traditionally a data mining process [4]. As a log of subqueries is in fact a log of accessed resources, data log mining algorithms could be used to solve our problem, where each item is an accessed predicate or triple pattern. Sequential pattern mining [7] focuses on discovering frequent subsequences (totally or partially ordered) from an ordered sequence of

events. An event is a collection of unordered items, an item is a literal, and a set of items composes an alphabet. In our context, we focus on sequential pattern mining algorithms able to operate on non-transactional logs such as WINEPI or MINEPI [6]. WINEPI decomposes a temporal sequence in overlapping windows of a user-defined size n and counts the frequency of episodes in all windows. Episodes can be of size 1 to n. MINEPI instead, looks for minimal occurrences of episodes. It identifies in a sequence, the set of time intervals of minimal occurrences of episodes according to the maximum user-defined window size. The number of minimal occurrences of an episode is called support. The minimum frequency (for WINEPI), the minimal support (for MINEPI) and the maximum window size (for both), are thresholds defined by the user. The difference of these approaches, is that WINEPI can be interpreted as the probability of encountering an episode from randomly chosen windows, while MINEPI counts exact occurrences of episodes.

We think that searching for BGPs in a federated log is not like searching for frequent episodes in a temporal log. First, the alphabet of events in a federated log can be proportional to the cardinality of data in the federation. A nested-loop operator can generate thousands of different subqueries as we observed with FedX. Managing huge alphabets is challenging for sequential pattern algorithms. FETA uses heuristics to reduce the alphabet by deducing hidden variables. Second, frequency of events in a federated log is related to the selectivity of operations and can confuse sequential pattern algorithms. Suppose, two queries $Q_1 : \{?x\ p1\ o1\ .\ ?x\ p2\ ?y\}$ and $Q_2 : \{?x\ p1\ ?y\ .\ ?y\ p3\ ?z\}$. The federated 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 $p1, p3$.

Limitations of sequential pattern mining algorithms have been pointed out in process mining [12]. Process mining algorithms recompute workflow models from logs. However, queries are not workflows and federated logs are not process logs. In a process log, events corresponds to identified tasks which is not the case in our context. The number of different subqueries can be proportional to the cardinality of the federated datasets. Moreover, in a federated log, a subquery cannot be the cause of another; in general, join ordering is decided according to the selectivity of subgoals in the original query.

6 Conclusions and future work

Federated query tracking allows data providers to know how their datasets are used. In this paper we proposed FETA, a federated query tracking approach that reverses federated Basic Graph Patterns (BGPs) from a shared log maintained by data providers. FETA links and unlinks variables from subqueries of the federated log by applying a set of heuristics to decrypt behavior of physical join operators.

Even in a worst case scenario, FETA extracts BGPs that contain original BGPs of federated queries executed with Anapsid and FedX. Extracted BGPs, annotated with endpoints, give valuable information to data providers about which triples are joined, when and by whom.

We think FETA opens several interesting perspectives. First, heuristics can be improved in many ways by better using semantics of predicates and answers. Second, we can improve FETA to make it agnostic to the federated query engine. Third, FETA can be used to generate a transactional log of BGPs from a temporal log of subqueries. Analyzing frequency of BPGs in a transactional log allows to discriminate false positive deductions of FETA.

7 Acknowledgments

This work was partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003), and by the DeSceNt project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01).

References

1. M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *International Semantic Web Conference (ISWC), Part I*, 2011.
2. C. Basca and A. Bernstein. Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In *International Semantic Web Conference (ISWC)*, 2010.
3. O. Görlitz and S. Staab. SPLENDID:SPARQL Endpoint Federation Exploiting VOID Descriptions. In *International Workshop on Consuming Linked Data (COLD)*, 2011.
4. J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Elsevier, 2011.
5. O. Hartig, C. Bizer, and J. C. Freytag. Executing SPARQL Queries over the Web of Linked Data. In *International Semantic Web Conference (ISWC)*, 2009.
6. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 1(3), 1997.
7. C. H. Mooney and J. F. Roddick. Sequential Pattern Mining–Approaches and Algorithms. *ACM Computing Surveys (CSUR)*, 45(2):19, 2013.
8. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3), 2009.
9. B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *European Semantic Web Conference (ESWC)*, 2008.
10. M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fed-Bench: A Benchmark Suite for Federated Semantic Data Query Processing. In *International Semantic Web Conference (ISWC)*, 2011.
11. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *International Semantic Web Conference (ISWC), Part I*, 2011.
12. W. Van Der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Science & Business Media, 2011.