Deducing Basic Graph Patterns from Logs of Linked Data Providers

Georges Nassopoulos

UNIVERSITÉ DE NANTES

ThŁse de Doctorat

Georges NASSOPOULOS

MØmoire prØsentØ en vue de l'obtention du grade de Docteur de l'UniversitØ de Nantes

École doctorale : Sciences et technologies de l'information, et mathØmatiques

Discipline : Informatique et applications
UnitØ de recherche : Laboratoire des Sciences du NumØrique de Nantes (LS2N)

Soutenue le 22 mai 2017

Deducing Basic Graph Patterns from Logs of Linked Data Providers

JURY

Rapporteurs :	M. Philippe L AMARRE, Professeur des UniversitØs, Institut National des Sciences AppliquØes de Lyon (INSA)
	M. Olivier C URÉ, Maître de confØrences HDR, Laboratoire d'informatique Gaspard-Monge
Examinateurs :	M^me^ Pascale K UNTZ-COSPEREC, Professeur des UniversitØs, Ecole Polytechnique de l'UniversitØ de Nantes
	M^me^ Claudia R ONCANCIO, Professeur des UniversitØs, Institut Polytechnique de Grenoble - Ensimag
	M. RenØ QUINIOU, Chercheur, Institut National de Recherche en Informatique et en Automatique de Rennes (INRIA)
Directeur de thŁse :	M. Pascal M OLLI, Professeur des UniversitØs, UniversitØ de Nantes
Co-encadrants de thŁse :	M^me^ Patricia S ERRANO-ALVARADO, Maître de confØrences, UniversitØ de Nantes
	M. Emmanuel D ESMONTILS, Maître de confØrences, UniversitØ de Nantes

Acknowledgements

First of all, I would like to thank the rapporteurs, Professor Dr. Philippe LAMARRE and Associate Professor Dr. Olivier CURÉ for accepting reading my PhD thesis and writing up a report. I would also like to thank Professor Dr. Pascale KUNTZ - COSPEREC, Professor Dr. Claudia RONCANCIO and Researcher Dr. René QUINIOU, for accepting to be part of the PhD jury as examiners. I would like to thank also Professor Dr. Philippe LAMARRE and Professor Dr. Sébastien GAMBS for the follow up during all the years of my PhD as members of the Comité de Suivi de Thèse.

I owe infinite gratitude to my supervisors Professor Dr. Pascal MOLLI, Associate Professor Dr. Patricia SERRANO-ALVARADO and Associate Professor Dr. Emmanuel DESMONTILS for entrusting me with my PhD theme and giving me the opportunity to participate in this scientific journey. Their guidance and supervision were invaluable during my PhD years. All the members of the GDD team and the LS2N (former LINA) laboratory provided a nice working environment and I would like to thank them for that.

I am also very thankful to all my fellow Ph.D. students. This journey was enriched with a multi-cultural and friendly environment, that I will never forget. I would like to list the people that helped me the most sharing of-work moments, drinks... and of coarse session therapies: Pauline FOLZ, Gabriela MONTOYA, Stamatina (Matoula) PETROLIA, Mohamed Amine AOUADHI, Anicet BART, Marko BUDINICH, Amir HAZEM, Firas HMIDA, Luis Daniel IBANIEZ, Brice NEDELEC, Jonathan PEPIN, Mathieu PERRIN, Alejandro REYES-AMARO, Nicolo RIVETTI and James SCICLUNA. A special thanks to Gabriela MONTOYA for showing me the good side of the force... of query processing over the Linked Data!

Finally I would like to thank my family. Euqarist¸ gia thn ˆneu ìrwn kai ˆneu orÐwn st rix sac. Moiˆzei koinìtupo allˆ qwrÐc esˆc den ja ta katˆferna ... sÐgoura ìqi tìso kalˆ. Eutuq¸c pou upˆrqei to google translˆte kai opoiosd pote mporeÐ na diapist¸sei ìti paramènw ìpwc pˆnta eugen c :)

À Sophie, Hélène, Hypatie et Marc

Contents

List of Tables

List of Figures

1

Introduction

1.1 The Semantic Web initiative

Semantic Web[1] is an extension of the current Web, known also as Web of Data. It provides a normalized way to find, share, reuse and combine information [10, 18]. The Semantic Web is made up of Linked Data[2] i.e., the Semantic Web is the whole while Linked Data is the parts. Linked Data practices have lead to a global data space interlinking various domains. In Figure 1.1 we see published datasets of the Linked Data cloud as of February 2017 including publications (in light grey), life science (in light purple) or cross domain (in brown). The W3C[3] recommendations to store, query and update Linked Data are the Resource Description Framework (RDF) data model and the SPARQL query language.

RDF is the graph-based model to represent information in the Linked Data. RDF encodes data in triples (subject; predicate; object). Subjects and objects are both IRIs or IRI and a string literal respectively. The predicate specifies how the subject and object are related, also represented by an IRI. In Figure 1.2 on page 17, we see an example of RDF graphs concerning Bob and DBpedia[4]. A RDF triple example of Bob's dataset is (bob: me; foaf : topic_interest; wd : Q12418) which expresses the interest of Bob to the wikipedia resource "wd : Q12418" i.e., Mona Lisa. A RDF triple example of DBpedia's dataset is (wd : Q12418; dcterms : creator; dbpedia: Leonardo_da_Vinci), which expresses that "wd : Q12418" was created by Leonardo da Vinci.

SPARQL is a sql-like query language that allows to manipulate and retrieve data stored in RDF format. SPARQL is used to match RDF triples expressed in form of triple patterns, where subjects, predicates and objects are IRIs, literals or variables. Each set of joined triple patterns of a SPARQL query is called a Basic Graph Pattern (BGP). Furthermore, SPARQL allows a query to consist of triple patterns which enhanced with

[1] http://semanticweb.org
[2] http://linkeddata.org/
[3] http://www.w3.org/
[4] Example taken from: https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/

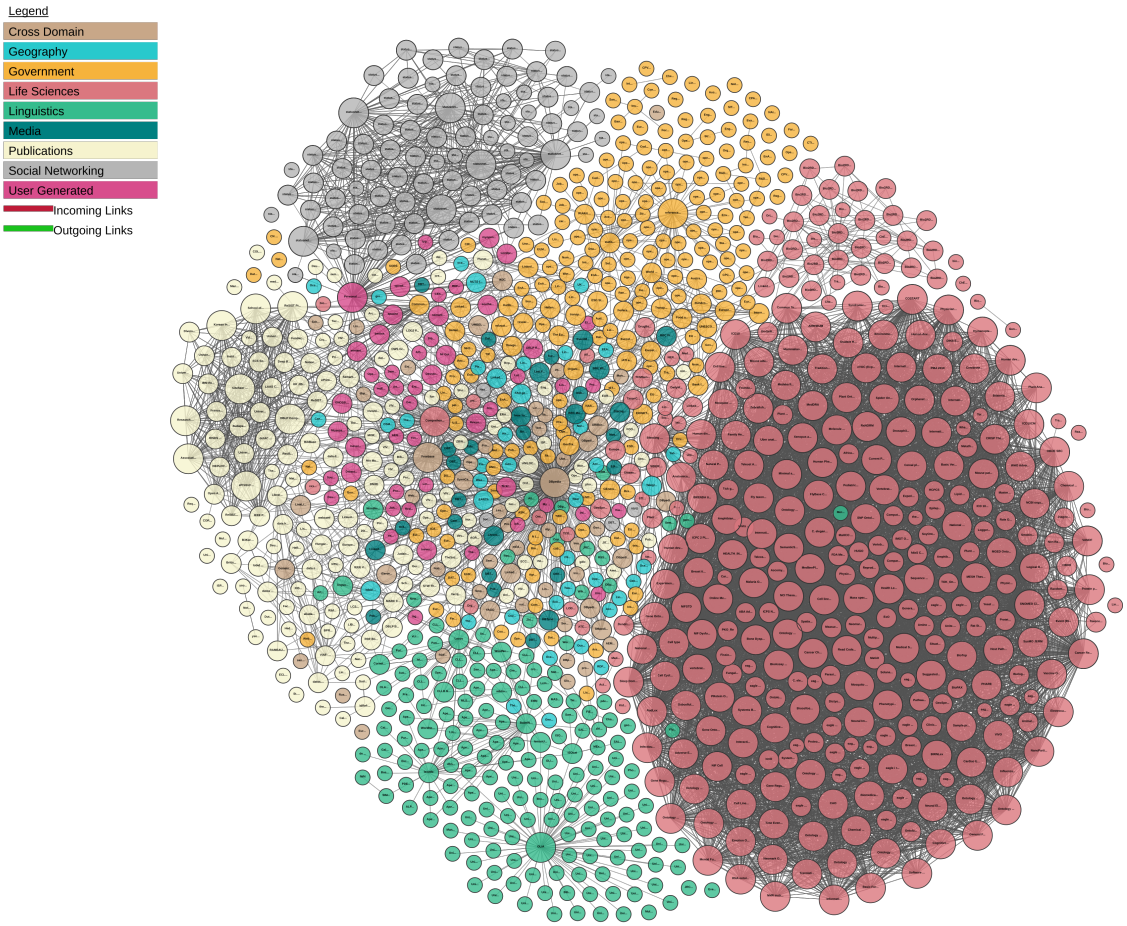


Figure 1.1 Linked Data cloud, as of February 2017 [10].

various features can express more complex expressions such as conjunctions, disjunctions or optional graph patterns.

The graph matching facility of the SPARQL language can be applied over one or several RDF datasets, residing in di erent sources. In Figure 1.3 the SPARQL query expresses a conjunction graph pattern, seeking to nd all artifacts that interest Bob and were created by Leonardo da Vinci. There exist a variety of methods and strategies to evaluate SPARQL queries. Although, the way of how Linked Data are consumed is mostly in uenced by who bears the workload of query processing, the data consumer or the data provider. Next we overview, the main approaches to query the Linked Data.

1.2 Querying the Linked Data

In the Linked Data, billions of triples are provided by autonomous providers across multiple domains. We overview below how this plethora of information is consumed [16]. Strategies for querying the Linked Data, can be hierarchized depending on (a) if the query can be answered on single or over several sources, and (b) if the query processing load is ensured by the data consumer or the data provider:

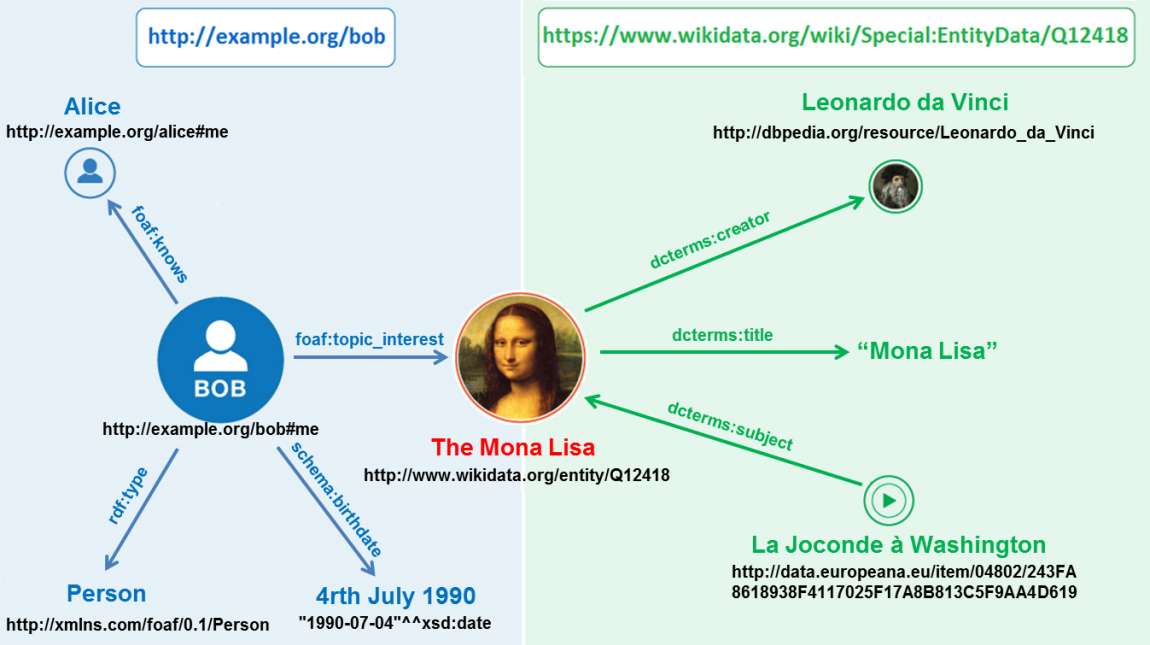


Figure 1.2 RDF graphs of Bob and DBpedia (concerning "Mona Lisa").

```
PREFIX bob:< http : ==example:org=bob#me= >
PREFIX dbpedia:< http : ==dbpedia:org=resource= >
PREFIX dcterms :< http : ==purl:org=dc=terms= >
PREFIX foaf :< http : ==xmlns:com=foaf=0:1= >
  SELECT ?artifact WHERE {
     bob:me foaf:topic_interest ?artifact .
     ?artifact dcterms:creator dbpedia:Leonardo_da_Vinci }
```

Figure 1.3 SPARQL query combining data from Bob and DBpedia.

1. Direct access to public interfaces : In the simplest case, a user accesses the public interfaces of providers in the Linked Data[5]. Although such an access provides the user with valuable data, at the same time ignores the great potential of the Web of Data that is to combine information from di erent sources.

2. Data warehouse : All providers' datasets are downloaded into a dump, creating a single local RDF store at the data consumer. Subsequently, queries are executed in a centralized way by combining data without any further communication with providers that publish them, following a data warehouse approach [50, 53, 56]. Query processing over a data dump increases the availability of providers. However, a data warehouse solution is not always practical, because of the cost to host all downloaded datasets and also the question of data freshness.

3. Federated query processing : Federated query engines [2, 9, 12, 48], evaluate a SPARQL query over a set of autonomous SPARQL endpoints. This federation is

[5] A well known example, is the public interface of the DBpedia SPARQL endpoint: http://dbpedia.org/sparql .

transparent to the end user i.e., the distributed datasets are consumed as if they were a single RDF graph. Federated query processing guarantees that data are up-to-date. On the other hand, the workload is pushed to the selected endpoints, raising the issue of servers' unavailability. Even if SPARQL endpoints put restrictions such as a limited execution time, their availability remains low [6].

4. Alternative query processing strategies : Due to the limitations that data warehouse and federated query processing have in exploiting the Web of Data, other approaches have been proposed [17, 23, 29, 45, 55, 58]. Some solutions, aim to find a trade-off between processing effort on the consumer and data availability on the provider, for instance:

   i Linked Data Documents (LDD): Consuming the Linked Data through LDDs, either uses pre-populated index structures [55], or focuses on live exploration by a traversal-based query execution [17]. Query evaluation with LDDs, has the constraint of longer query execution times, compared to federated query processing or direct access to public interfaces. On the other hand compared to data dumps, documents allow live querying.

   ii Triple Pattern Fragments (TPF): TPFs are a new way to consume Linked Data, also called basic Linked Data Fragments (LDFs) [58] Clients split queries into single triple pattern subqueries and evaluate them against providers, that publish their data as TPF servers[6]. The TPF solution can be applied over single or federations of TPF servers. Query processing of costly SPARQL features, is pushed to the client to leverage the pressure on providers.

LD provider	IP	Time	Query	Answer
Bob	ip_1	12 : 11 : 10	SELECT WHERE { Bob : me foaf : topic_interest ?artifact . }	{ ?artifact → wd:Q12418 }
DBpedia	ip_1	12 : 11 : 15	SELECT WHERE { ?artifact dcterms : creator dbpedia : Leonardo_da_Vinci . }	{ ?artifact → wd:Q12418 }

Table 1.1 Federated log of Q_1 traces, produced by a federated query engine and executed over the federation of SPARQL endpoints that are hosted by DBpedia and Bob data providers.

The usefulness of Linked Data is that it allows to evaluate queries through a distributed execution that roams from resource to resource, residing in the same or different datasets. Queries can be decomposed in many subqueries, either due to the location of their matching triples or for optimization reasons during their execution. Data providers receiving subqueries do not know the whole query they evaluate. In Table 1.1 we see a federated log of DBpedia and Bob, with execution traces of Q_1 = SELECT ?artifact WHERE { bob: me foaf : topic_interest ?artifact . ?artifact dcterms : creator dbpedia: Leonardo_da_Vinci . }. The question that emerges in this example, is how Bob data provider could know that its data were combined with data of DBpedia data provider. Next, we define the problem we aim to solve.

[6] In this thesis, we refer to LDF as TPF servers.

1.3 Problem statement

The limitation of consuming Linked Data using either federated query processing or TPFs, is that providers are not aware of the whole user queries they process. Data providers just observe subqueries of decomposed user queries and have no idea about their data usage i.e., which data are joined with their datasets, when and by whom. Knowing how datasets are queried is essential, not only for ensuring usage control but for other purposes as well. In particular, data providers need to know the queries they process in order to optimize the cost of provided services (i.e., access to their Linked Data), justify return on investment, improve their users' experience or even create business models to discover usage trends over the Semantic Web.

In the traditional model of data warehouse, the meta-information of data usage is completely hidden from data providers. But due to the distributed nature of the Semantic Web, the extraction and processing of all Linked Data locally at the data consumer seems a paradox. On the other hand, for query processing either over single, or, federations of data providers, the de ciency of ignoring how data are joined remains open to be answered.

A simple solution is to consider that data consumers inform data providers about their data usage, either: (i) *a priori*, for instance by respecting licence agreements established between both parties [51], (ii) *on the fly*, through query execution environments that inform data providers, to which original query every subquery belongs to, or, (iii) *a posteriori*, by publishing on the web their queries, once they have been executed. However, such solutions are not practical and scalable. Even worst, data providers have to verify that data are actually joined in the way public queries describe or agreements stipulate. Only logs give evidences about real execution of queries.

In this thesis we aim to infer what users are looking for on the Semantic Web, by inferring the general form of SPARQL queries, in particular over (i) single or federations of TPF servers, and, (b) over federations of SPARQL endpoints. The scienti c question we aim to answer, is the following:

How to infer Basic Graph Patterns (BGPs) of SPARQL queries executed by data consumers from logs of servers hosted by data providers?

Note that we do not aim to infer the exact queries posed by users as we are interested, in a general way, in detecting how Linked Data are crossed together.

1.4 Approach

We aim to reconstruct BGPs from logs of Linked Data providers. Extracting information from logs is traditionally a Data Mining process. A log of subqueries is in fact a log of accessed resources via the web. Thus, Data Mining algorithms [3, 15, 32] could be used to solve our problem where each predicate, triple pattern or subquery is an accessed resource on the data provider. The lacunae of Data Mining is that none of its algorithms has addressed reversing BGPs from a query log. In general, these algorithms extract sets of items and deduce rules based on occurrences of items in query logs. Unfortunately, obtained results may not be always representative of joins. In particular, frequent sets of

accessed resources do not correspond necessarily to joins and joins are not always frequent enough to be deduced as sets.

LD provider	IP	Time	Query	Answer
Bob	ip_1	12 : 11 : 10	SELECT WHERE f Bob : me foaf : topic_interest ?artifact g	f ?artifact 7! wd:Q12418 g
DBpedia	ip_1	12 : 11 : 15	SELECT WHERE f ?artifact dcterms : title ?title g	f ?artifact 7! wd:Q12418 ; ?title 7! "Mona Lisa "g
DBpedia	ip_1	12 : 11 : 15	SELECT WHERE f ?artifact dcterms : creator dbpedia : Leonardo_da_Vinci g	f ?artifact 7! wd:Q12418 g

Table 1.2 Federated log of Q_I and Q_{II} traces, produced by a federated query engine and executed concurrently over the federation of SPARQL endpoints that are hosted by DBpedia and Bob data providers.

In this thesis, we propose a BGP reversing approach to solve our problem statement. Our work aims to reveal and deduce joins between triple patterns i.e., to extract executed BGPs over the Linked Data from single or federated logs of Linked Data providers. The goal, is to link hundred or thousand subqueries that correspond to one or more user queries, based on common constants on their triple patterns or mappings of their projected variables. The main challenge is the concurrent execution of queries. Suppose an additional SPARQL query Q_{II} = SELECT WHERE f ?artifact dcterms : title ?title g, executed concurrently with Q_I from the same user ip_1. The federated log of DBpedia and Bob for a concurrent execution of Q_I and Q_{II}, is presented in Table 1.2. We observe that all queries in the log concern the same resource for the variable ?artifact i.e., wd : Q12418 If we find a function f to reverse BGPs from execution traces of one query, f is able to reverse the same BGPs from execution traces of several concurrent queries?

1.5 Organization and contributions

The contributions of this thesis are, in summary:

The definition of the scientific problem of reversing BGPs of user queries, from a log of subqueries that corresponds to their execution traces.

The analysis of Data Mining algorithms to solve our problem and their limitations.

LIFT, an ad hoc approach that reverses triple patterns and their joins, evaluated through Triple Pattern Fragments over single or federations of TPF servers. Obtained results have good recall and a precision which depends on the concurrent execution of queries and the deduction parameters of LIFT.

FETA, an ad hoc approach that reverses triple patterns and their joins, evaluated through federated query processing over federations of SPARQL endpoints. Similarly to LIFT, obtained results have good recall and a precision which depends on the concurrent execution of queries and the deduction parameters of FETA.

The thesis manuscript is organized as follows: Chapter 2 introduces the SPARQL semantics, illustrates two main procedures for querying the Linked Data and de nes formally the scienti c problem we address. The related work of Data Mining, is analysed in Chapter 3. Chapter 4 presents LIFT, our proposed reversing approach that extracts BGPs from single or federated logs of TPF servers. Chapter 5 presents FESTA, our proposed reversing approach that extracts BGPs from federated logs of SPARQL endpoints. Finally, conclusions and perspectives are outlined in Chapter 6.

2

Preliminaries: querying the Linked Data

Contents

In this chapter we illustrate how SPARQL expressions are formally de ned and then consumed over data providers using various query processing strategies, each with its own optimization techniques. Then, based on traces produced with these query processing strategies, we formally de ne the problem we aim to solve—How to infer Basic Graph Patterns (BGPs) of SPARQL queries executed by data consumers from logs of servers hosted by data providers?

First, we present how SPARQL semantics formalize the graph expressions to consume Linked Data, in Section 2.1. Then, we describe the main physical join operators, used in practice to evaluate SPARQL, in Section 2.2. Thereafter, we illustrate two main approaches to consume Linked Data based on these physical operators, each with its own optimization techniques. In particular, rst we present query processing over single or federations of TPF servers in Section 2.3, and second, query processing over federations of SPARQL endpoints in Section 2.4. Finally, we formally de ne the scienti c problem we aim to solve based on log traces generated with these approaches, in Section 2.5.

2.1 SPARQL semantics

RDF is a model that represents the Linked Data as directed labeled graphs and SPARQL is essentially a graph-matching query language. In this section, we address the formal study of SPARQL, by focusing on its graph pattern facility[1]. From its basic features, SPARQL is used to build recursively more complex expressions in order to consume data residing in one or more data providers. In order to de ne how these expressions are evaluated over RDF graphs, we adopt the formalization of [39, 47]. The elementary assumptions and de nitions we adopt, are:

> We assume pairwise disjoint in nite sets B, L, I (blank nodes, literals, and IRIs respectively). A RDF triple, tr, has the form $(s; p; o)$, where the subject $s \in (I \cup B)$, the predicate $p \in I$ and the object $o \in (I \cup B \cup L)$. A RDF graph is a set of triples, also called RDF dataset or RDF document. The nite set of all triples in a RDF graph is $G \in 2^T$, where $T = (I \cup B) \times (I) \times (I \cup B \cup L)$ is the (in nite) set of all RDF triples.
>
> We assume an in nite set S of variables. A mapping μ is a partial, non surjective and non injective function that expresses a variable-to-document binding i.e.; $S \rightarrow B \cup L \cup I$. The universe of all mappings is Ω. The domain of a mapping, $dom(\mu)$, is the subset S where μ is de ned. Two mappings μ_1, μ_2 are compatible, written $\mu_1 \sim \mu_2$, if they agree on their common domain variables[2] i.e., if $\mu_1(?x) = \mu_2(?x)$, $\forall ?x \in (dom(\mu_1) \cap dom(\mu_2))$. This is equivalent to say that $\mu_1 \cup \mu_2$ is also a mapping.

The SPARQL language and Relational Algebra have the same expressive power [5]. For this reason, SPARQL is formalized based on Relational Algebra, using set-based semantics[3] in order to evaluate algebraic operators[4]. Next de nitions, present SPARQL algebraic syntax.

De nition 1 (SPARQL expression) A SPARQL graph pattern expression P is built recursively as follows[5]:

1. A triple pattern, tp, is a graph pattern represented by a triple from $(I \cup L \cup S) \times (I \cup S) \times (I \cup L \cup S)$.
2. If P_1 and P_2 are graph patterns, then expressions $(P_1$ AND $P_2)$[6], $(P_1$ OPT $P_2)$ and $(P_1$ UNION $P_2)$ are graph patterns i.e., conjunction, optional, and union graph pattern, respectively.
3. If P is a graph pattern and R is a SPARQL built-in condition[7], then the expression $(P$ FILTER $R)$ is a graph pattern (or a lter graph pattern).

---

[1] The de nitions presented in this chapter concern the 1.0 SPARQL protocol version.

[2] Variables in SPARQL language are pre xed by a "?" symbol.

[3] Almost all aspects of set-based semantics can be carried over the o cial bag-based semantics adopted by W3C [47].

[4] Which are partially extended for SPARQL 1.1 [7].

[5] SPARQL 1.1 extends SPARQL 1.0 with graph expression keywords, such as SERVICE.

[6] Note that conjunction is also denoted with the "." symbol.

[7] A SPARQL built-in condition is constructed using a combination of elements $I \cup L \cup S$ and constants, logical connectivities $(\neg; \wedge; \vee)$, inequality or equality symbols $(<; \leq; \geq; >$ or $=)$, unary predicates (such as bound or isIRI) plus other features.

De nition 2 (SPARQL set algebra) Let $_1$, $_2$ be mapping sets, R is a lter condition and S S be a nite set of variables. The algebraic operations of join (1), union ([ ), minus (n), projection ( ), and selection ( ) are de ned as follows[8]:

$$
\begin{aligned}
&{}_1 1 \ {}_2 := f \ {}_1 [\ {}_2 \mid {}_1 2 \ {}_1;\ {}_2 2 \ {}_2 : \ {}_1 \ \ {}_2 g \\
&{}_1 [\ {}_2 := f \ \ j \ \ 2 \ {}_1 \_ \ \ 2 \ {}_2 g \\
&{}_1 n \ {}_2 := f \ {}_1 2 \ {}_1 j 8 \ {}_2 2 \ {}_2 : \ {}_1 6 \ {}_2 g \\
&{}_S( ) := \ f \ {}_1 j 9 \ {}_2 : (\ {}_1 [\ {}_2 2 \ ) \ ^ (dom(\ {}_1) \ \ S) ^ \ (dom(\ {}_2) \setminus S = ;) g \\
&{}_R( ) := \ f \ \ 2 \ \ j \ \ j= R g
\end{aligned}
$$

De nition 3 (SPARQL set semantics) Let G be a RDF graph, tp a triple pattern, P, P_1, P_2 SPARQL expressions, R a lter condition and S S be a nite set of variables. The evaluation of a graph expression [[P]]$_G$, by using the set semantics as described above, is de ned recursively as follows:

$$
\begin{aligned}
&[[tp]]_G := f \ \ j (dom(\ ) = vars(tp)) \ ^ \ (\ (tp) \ 2 G) \ g \\
&[[P_1 \text{ AND } P_2]]_G := [[P_1]]_G \ 1 \ [[P_2]]_G \\
&[[P_1 \text{ UNION } P_2]]_G := [[P_1]]_G \ [\ [[P_2]]_G \\
&[[P_1 \text{ OPT } P_2]]_G := ([[P_1]]_G \ 1 \ [[P_2]]_G) \ [\ ([[P_1]]_G \ n \ [[P_2]]_G) \\
&[[P \text{ FILTER } R \ ]]_G := \ \ _R([[P]]_R) \\
&[[\text{SELECT}_S \ (P)]]_G := \ \ _S([[P]]_G) \\
&[[\text{ASK } \ (P)]]_G := \ : (; \ = [[P]]_G)
\end{aligned}
$$

Note, that these elementary operators may be used to recursively de ne other. For instance, the full outer join is evaluated as: ($_1$ 1 $_2$) [($_1$ n $_2$) [($_2$ n $_1$).

In this section we reviewed the formal de nitions of the SPARQL language and how it is used to evaluate RDF graph expressions. Next, we describe how the evaluation of SPARQL expressions is implemented over the distributed network of the Linked Data through physical join operators, namely symmetric hash and nested-loop joins.

2.2 Physical join operators

We focus in this work on join operators in distributed environments, as we aim to deduce how data are combined. Join features de ne the evaluation of a SPARQL query at the conceptual level i.e., what needs to be done, thus called logical operators Physical operators, each associated with a cost, implement the operation described by logical operators i.e., stipulate how the join is actually done. We present next the physical operators that are mainly used to evaluate SPARQL queries, namely symmetric hash and nested-loop joins.

Consider the graph expressions R = f ?x p2 ?y g and S = f ?z p3 ?y g, which are evaluated over data providers p_A and p_B respectively. M and N are the sizes of R and S, respectively. R-matching triples are f (s1; p2; o3); (s2; p2; o1); (s3; p2; o4) g, while

[8] SPARQL 1.1 extends 1.0 also with features to evaluate the variables contained in graph expressions, such as FILTER NOT EXISTS .

S-matching triples are f (s2; p3; o1); (s2; p3; o2); (s4; p3; o3) g. We denote every id-matching triple of a graph expression P as $tr_{P_{id}}$ e.g., tr_{S_1} = (s2; p3; o1). Next we evaluate the R 1 S conjunction graph pattern, using the symmetric hash and nested-loop operators that we describe next.

Symmetric hash join [60]: In the traditional hash join [13], two phases are performed. First, inputs from the smaller dataset are partitioned i.e., built into a hash table. Then, tuples of the opposite dataset are used to probe i.e., lookup matching data from the built table. For this, its time complexity is O(M + N). This is a blocking operator as, rst, inputs from the smaller dataset must be partitioned into a table. On the other hand, the symmetric hash join uses at the same time both datasets to partition and lookup common data, independently of which dataset has the smaller cardinality. Symmetric hash is used to reduce response time because of its fully pipelined nature as build and probe phases are interleaved. That is, each tuple from either dataset is partitioned into a hash table and at the same time used to probe the hash table of the other dataset. So the complexity of this operator is O(2 (M + N)), equivalent to O(M + N). Depending on the size of retrieved datasets i.e., interim result sets, it can be a very e cient solution due to its possible parallelization. However, symmetric hash is expensive if the interim result sets are much larger than the join result size. In addition, if remote sites impose a result size limit k, where k < jSj or jRj, then join results may be lost [7].

The symmetric hash operator proceeds incrementally[9] by fetching one by one triples from both R and S datasets for our example. First, tr_{R_1} and tr_{S_1} are partitioned to their corresponding hash tables and probed to the opposite ones. Then, it is the turn to of tr_{R_2} and tr_{S_2}, and nally of tr_{R_3} and tr_{S_3}. The conjunction graph pattern R 1 S produces results when the triple pairs f tr_{R_1}, tr_{S_3} g and f tr_{R_3}, tr_{S_1} g are joined. Thus, solution mappings with this physical operator are extracted only in the last step, where tr_{R_3} and tr_{S_3} are fetched from their sources and are also used to probe the opposite triples respectively.

Nested-loop join [30]: In a double iteration, each triple of the outer dataset is used to search matching triples in the inner, so its time complexity is O(M N). In a distributed environment, in order for nested-loops to be e ective the outer is the smaller and the inner is the largest source. The main advantages of this physical operator, is that (a) it is used to avoid reaching the limit response de ned by data providers, as triples of the smaller dataset are progressively pushed to the site hosting the larger dataset, and (b) the necessary in-memory size to compare the input data, is less important than in the case of a symmetric hash join. The disadvantages of nested-loop compared to symmetric hash are: (a) its higher time complexity, and (b) the fact that it is not by default pipelined. Nested-loop joins can be evaluated either as a blocking operator [7], or, enhanced with the well known pipelined operator model described in [13]. In the former case, all triples of the outer dataset are extracted locally at the client before probing the inner dataset in the opposite site. In the latter case, triples of the outer dataset are progressively used to probe matching triples in the inner, without waiting all outer triples to be extracted

[9] The block size of triples that are incrementally extracted, is de ned using the LIMIT feature. Every following step of the symmetric hash join, fetches the next matching triple de ned with the OFFSET operator.

locally at the client.

The nested-loop operator for our example, may be implemented either in a blocking or pipelined fashion. In any case, the first solution mapping is produced when tr_{R_1} is pushed to the inner dataset while the second solution with tr_{R_3}, as the conjunction graph pattern R 1 S produces results when the triple pairs $\{tr_{R_1}, tr_{S_3}\}$ and $\{tr_{R_3}, tr_{S_1}\}$ are joined. Note that as both datasets have the same cardinality, the join ordering choice is arbitrary.

Next, we focus on two approaches to query the Linked Data. First, using Triple Pattern Fragments over single or federations of TPF servers, in Section 2.3. Second, using federated query processing over federations of SPARQL endpoints, in Section 2.4.

2.3 Querying TPF servers

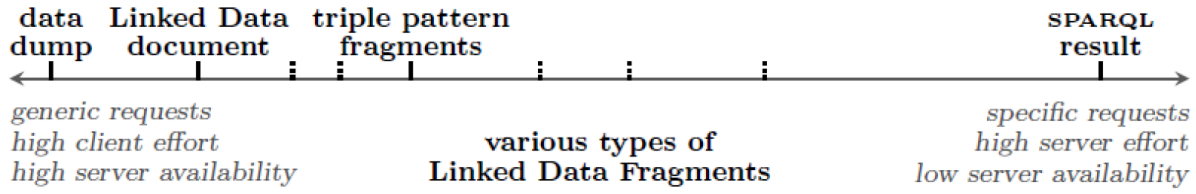


Figure 2.1 Different types of Linked Data Fragments (LDFs) and their trade-offs [57].

Triple Pattern Fragments, also called basic LDFs, were proposed to leverage the "pressure" on data providers[10] by delegating to clients the process of high cost SPARQL features [58]. As we see in Figure 2.1, all different types of LDFs are hierarchized based on a combination of characteristics of Web APIs such as performance or cache reuse, from the perspective of either servers or clients performing a specific task. For instance, data dump is the LDF that requires a high effort on the client but at the same time creates high availability on the server. These criteria are:

1. Performance : Measures the rate of completion per query processing task i.e., the number of processed requests/responses per time unit.

2. Cost : Refers to consumed resources per query processing task i.e., CPU, RAM, and IO consumption.

3. Cache reuse : Measures the ratio of items in the case that are requested multiple times versus the totality of stored items in this cache[11].

4. Bandwidth : Consists of the product of retrieved responses with the average response size, per query processing task.

[10] Linked Data providers e.g., DBpedia, publish their data both as SPARQL endpoint and TPF server.

[11] Servers use extensively caches with LDFs. As clients have the tendency to repeat the same queries, it is useful for data providers to employ practices of data shipping .

5. E ciency : Measures the fraction of data retrieved from a server during the execution of a task over the amount of data that are actually required for this task. This measure was introduced to highlight the overhead for clients when using TPFs, as they mostly bear the workload of query processing.

TPF clients decompose SPARQL queries into single triple patterns which they evaluate on TPF servers, and process locally the high cost operators of the SPARQL language. This approach introduces new semantics and de nitions, that we abstractly present in Section 2.3.1. Then, based on TPF semantics, we illustrate the query evaluation with TPFs in Section 2.3.2.

2.3.1 The TPF framework

TPF semantics describe conceptually how the client needs to evaluate complex queries, by matching only single triple patterns on the server. Next, we brie y introduce the TPF concepts. The formalization of both LDF and TPF is analytically presented in [59]. We use the simpli ed dataset of Table 2.1 to explain the de nitions of this section.[12]

@p_A
c1 p1 a
c2 p1 b
c3 p1 c
c4 p1 d
c1 p2 toto
c2 p2 toto
c3 p3 titi
c4 p3 titi
c1 p4 a
c2 p4 b

Table 2.1 Example of a simpli ed dataset of a TPF server, hosted by data provider p_A.

Triple Pattern Fragments (TPF) interface : Consists of Linked Data Fragments with the following properties: (a) data: all triples of a RDF graph that match a given triple pattern and are returned as answer to a single triple pattern subquery posed by a client, (b) metadata: estimation of the number of triples that match the given triple pattern, and (c) controls: a hypermedia form that allows clients to retrieve any TPF of the same knowledge graph.

We suppose that the dataset presented in Table 2.1 is published by the TPF server hosted by p_A data provider. Consider the triple pattern $tp = \{?y\ p1\ ?x\}$. p_A will return as answer to this triple pattern a TPF with: (a) the set of triples that matches the triple pattern i.e., $\{(c1, p1, a), (c2, p1, b), (c3, p1, c), (c4, p1, d)\}$ (b) the estimation of the number of matching triples for this set i.e., 4 triples and (c) a

[12] Note, that for simplicity we removed the pre xes from predicates.

form with how to retrieve other triple patterns of the same graph i.e., triple patterns f ?x p2 ?yg, f ?x p3 ?yg and f ?x p4 ?yg.

Triple-pattern-based selector function : Let tp be a triple pattern. The triple-pattern-based selector function fortp, denoted bysr$_{tp}$, is a selector function that for every datasetG 2 2^{T} is de ned by sr$_{tp}$ = f tr 2 G j tr is a matching triple for tp g.

For the dataset of Table 2.1, of a TPF server hosted byp$_{A}$, an instance of triple-pattern-based selector corresponding totp = f ?y p1 ?xg is sr$_{tp}$ = f ?subject = & predicate = p1 & object = g. Note, that the TPF client always renames variable names in a SPARQL query assubject, predicate or object. The TPF returned from the TPF server for tp, is described in the previous point.

Hypermedia controls : A hypermedia control is a declarative construct, that informs clients for possible application and/or session state changes in the server and explains how to e ectuate them. With this information, no external documentation is necessary to browse and consume the datasets of this server. TPF servers use on their interfaces a speci c language, namely theHydraCoreV ocabulary[24], in order to de ne the collection of links and forms in RDF.

A simpli ed set of controls, regarding forms for the dataset in Table 2.1, is the set of predicates that can be answered from this TPF server i.e.f p1; p2; p3; p4g. With this information the TPF client will know the possible triple patterns that this TPF server is able to answer, even when requesting only one triple pattern.

Triple Pattern Fragment : Let G 2 2^{T} be a nite set of blank-node-free RDF triples. A Triple Pattern Fragment (TPF) of G, denotedf , is a tuple hu; sr; ; M; C i with the following ve elements: (i) u is a URI representing the "authoritative" source from whichf can be retrieved, (ii) sr is a selector function, (iii) is a set of (blank-node-free) RDF triples that is the result of applying the selector function sr to G, (iv) M is a nite set of (additional) RDF triples, including triples that represent metadata for f, and (v)C is a nite set of hypermedia controls.

Consider that querySELECT WHERE f ?x p2 toto : ?x p1 ?yg is addressed to TPF server ofp$_{A}$. The corresponding TPFs for each triple pattern in this query, are:

f$_{tp_1}$ = h u$_{tp_1}$ = http : ==pa:com=sr$_{tp_1}$;
sr$_{tp_1}$ = f ?subject = & predicate = p2 & object = totog;
$_{tp_1}$ = f (c1; p2; toto); (c2; p2; toto)g
M$_{tp_1}$ = f (u$_{tp_1}$; void : triples; 2)g;
C = f p1; p2; p3; p4g i, and

f$_{tp_2}$ = h u$_{tp_2}$ = http : ==pa:com=sr$_{tp_2}$;

$sr_{tp_2} = f(?subject = \ \& \ predicate = p1 \ \& \ object = g;$

$_{tp_2} = f(c1;\ p1;\ a);\ (c2;\ p1;\ b);\ (c3;\ p1;\ c);\ (c4;\ p1;\ d)g;$

$M_{tp_2} = f(u_{tp_2};\ void:triples;\ 4)g;$

$C = f\,p1;\ p2;\ p3;\ p4g\ i$

Note that for f_{tp_1} and f_{tp_2} we use the same annotation C, as control elds for both fragments are the same for the dataset of the TPF server of A.

TPF page : Let $hu; sr; \ ; M; C\,i$ be f, a TPF of some nite set of blank-node-free RDF triples $G \ 2 \ 2^T$. A page partitioning of f is a nite, non-empty set [13] whose elements are called pages of f. Each page $\# \ 2$ has the form $hu_\#; u; sr_\#; \ _\#; M_\#; C_\#i$ with the following six properties: (i) $u_\#$ is the URI from which the page is retrieved, (ii) u is the source for retrieving the whole f, (iii) $sr_\#$ is a selector function to retrieve the page, (iii) $_\#$ is the set of matching triples of the page, a subset of the triples matching f (iv) M is a superset of the metadata of f, with both matching estimations of the page and the fragment f, and (v) $C_\#$ is a superset of the controls of f, enhanced with links to previous and next pages.

A TPF page is composed by the subset of matching triples of a fragment, which size is de ned by the TPF server [14]. Next we present the sets of TPF pages for triple patterns of our example query. Consider that the page size de ned by the TPF server is equal to 2. Then, the fragment that corresponds to tp_1 will not be fractioned. Instead, the corresponding set of pages for tp_2, namely $\#_{2a}$ and $\#_{2b}$, are:

$\#_{2a} = h\,u_{\#_{2a}} = http::==pa:com=sr_{tp_2}\&\ page=1;$

$u_{tp_2} = http::==pa:com=sr_{tp_2}$

$sr_{\#_{2a}} = sr_{tp_2}\&\ page=1;$

$_{\#_{2a}} = f(c1;\ p1;\ a);\ (c2;\ p1;\ b)g;$

$M_{\#_{2a}} = f(u_{\#_{2a}};void:triples;\ 2);\ (u_{tp_2};void:triples;\ 4)g;$

$C_{\#_{2a}} = f\,f\ p1;\ p2;\ p3;\ p4g;$

$f < prev\_page >: null;\ < next\_page >: u_{\#_{2b}}gg\ i$, and

$\#_{2b} = h\,u_{\#_{2b}} = http::==pa:com=sr_{tp_2}\&\ page=2;$

$u_{tp_2} = http::==pa:com=sr_{tp_2}$

$sr_{\#_{2b}} = sr_{tp_2}\&\ page=2;$

$_{\#_{2b}} = f(c3;\ p1;\ c);\ (c4;\ p1;\ d)g;$

$M_{\#_{2b}} = f(u_{\#_{2b}};\ void:triples;\ 2);\ (u_{tp_2};\ void:triples;\ 4)g;$

[13] We use the notation instead of [59] to distinguish it from the federation of data providers , a notation we used in [35].

[14] In practice, a TPF server e.g., DBpedia de nes a page size equal to 100 matching triples.

$$C_{\#2a} = \{\{p1, p2, p3, p4\},\ \{<prev\_page>: u_{\#2a},\ <next\_page>: null\}\}$$

Next, we describe the procedure of query evaluation through TPFs. In particular we present the pipelined evaluation of TPFs, where mappings from a fragment that match a triple pattern are incrementally pushed into another, through a dynamic implementation of the nested-loop join.

2.3.2 TPF query processing

TPFs can be used to evaluate SPARQL queries over both single or federations of TPF servers. The general workflow model of query execution on TPF severs, represented in Figure 2.2, consists of three steps:

1. Query decomposition: Transforms at the TPF client a SPARQL query into sets of triple patterns that are evaluated through a set of TPFs over the targeted TPF server(s). SPARQL features will be processed locally at the TPF client, during the distributed execution phase.

2. Global query optimization: Establishes at the TPF client the most suitable order of joins in the original query, in order to minimize the number of http requests using a cost estimation function. First, the TPF client sends a selector function for each triple pattern of the original query. Then, it decides the join ordering execution using the estimation of matching triples in the Metadata M, returned by the TPF server of each selector. Note that when a query is addressed to a federation of TPF servers, each selector in this phase is used for both count estimation and data localisation.

3. Distributed execution: Evaluates each triple pattern at the TPF server based on the join ordering established in the previous phase and pushes its mappings towards the next triple pattern. This procedure simulates a nested-loop implementation. The first implementation of the algorithm in [58] defines a blocking iterator to evaluate each nested-loop. More precisely, this blocking operator needs first to pull all triples of the outer dataset before pushing mappings into the inner. The evolution of this algorithm in [57], employs the pipelined iterator model [13]. This model extracts progressively triples that match a triple pattern and pushes their mappings to the next, without waiting to extract the remaining triples of the former. Next, we see an example of this evaluation.

Consider again SELECT WHERE {?x p2 to . to : ?x p1 ?y}. First, this query is decomposed in a set of triple patterns i.e. $tp_1 = \{?x\ p2\ to\}$ and $tp_2 = \{?x\ p1\ ?y\}$. Next, the TPF client sends two selectors, one for each triple pattern i.e. f_{tp1} and f_{tp2}, and uses their Metadata i.e., M_{tp_1} and M_{tp_2}, to choose the most suitable join ordering, as we see in the first two entries of Table 2.2. As $M_{f_{tp_1}} = 2$ and $M_{f_{tp_2}} = 4$, the TPF client starts with tp_1. That is, mapping results of the join variable of tp_1 i.e., ?subject $\rightarrow$ c1; c2, are pushed into the corresponding variable of tp_2, one by one. TPF client pushes these mappings by evaluating sequentially the triple pattern selectors ?subject = c1 & predicate =

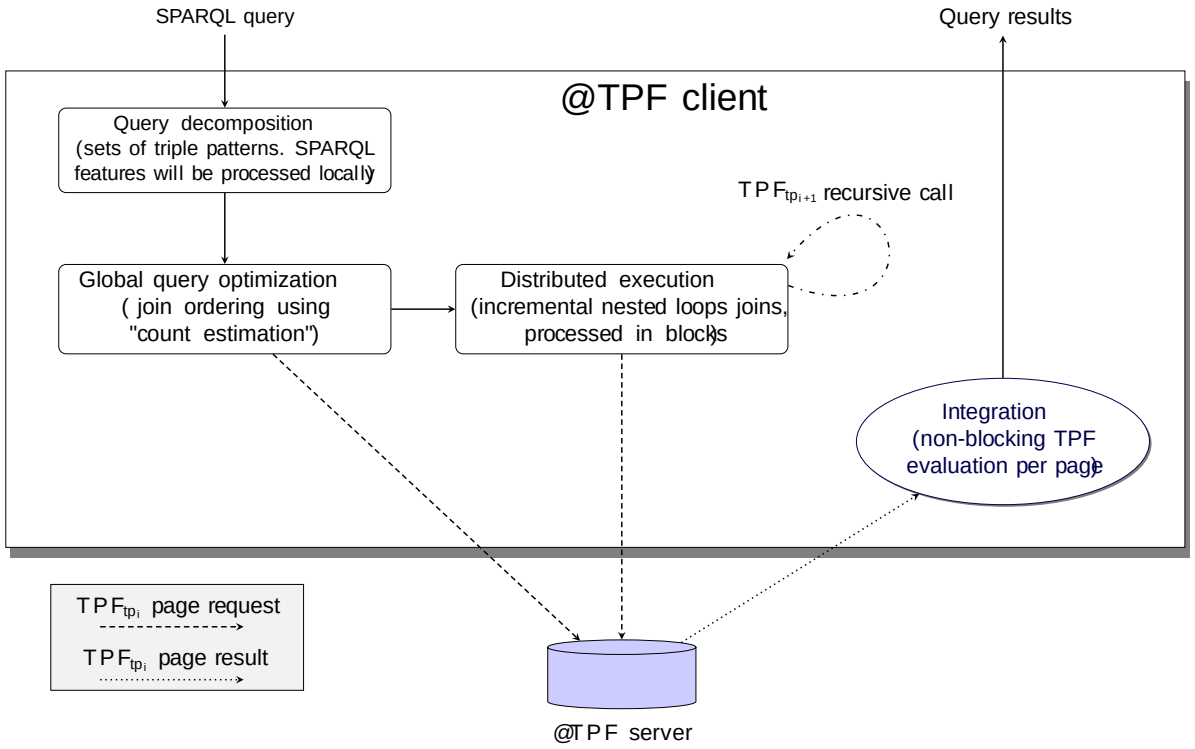


Figure 2.2 TPF query processing model. When applied over a federation of TPF servers, the selector functions for count estimation are also used for data localisation.

LD provider	IP	Time	TP selector	Answer
p_A	ip_1	09:10:01	?subject= & predicate = p1 & object=	f < :::controls::: >; f (c1; p1; a); (c2; p1; b); (c3; p1; c); (c4; p1; d)g; < :::metadata::: > g
p_A	ip_1	09:10:04	?subject= & predicate = p2 & object= toto	f < :::controls::: >; f (c1; p2; toto); (c2; p2; toto)g; < :::metadata::: > g
p_A	ip_1	09:10:07	?subject= c1 & predicate = p1 & object=	f < :::controls::: >; f (c1; p1; a); < :::metadata::: > g
p_A	ip_1	09:10:09	?subject= c2 & predicate = p1 & object=	f < :::controls::: >; f (c2; p1; b)g; < :::metadata::: > g

Table 2.2 Query log of SELECT WHERE f ?x p2 toto : ?x p1 ?yg traces, produced by a TPF client with ip_1 IP address and executed on the TPF server hosted by p_A data provider.

p1 & object = g and f ?subject = c2 & predicate = p1 & object = g to the TPF server, which results to mappings ?object 7! a and ?object 7! b respectively, as we see in the last two entries of Table 2.2. This approach is recursive, as continuous joins between multiple triple patterns are evaluated without waiting all mappings of a triple pattern to be pushed into another. For instance, suppose that this query had a third triple pattern, tp_3 = f ?y p3 ?wg. In this case, results produced when pushing progressively mappings of f_{tp_1} into tp_2 are subsequently pushed into tp_3, without waiting all mappings of f_{tp_1} to be pushed into tp_2.

In this section, rst we abstractly presented the concept of Triple Pattern Fragments. Thereafter, we illustrated the incremental procedure of TPF evaluation through nested-loops, that can be applied both over single or federations of TPF servers. Next, we de ne the procedure of consuming Linked Data over federations of SPARQL endpoints.

2.4 Querying SPARQL endpoints

As pointed in Chapter 1, data consumers query Linked Data in SPARQL endpoints, either by accessing directly their public interfaces or via query engines that access data residing in di erent sites. In this section, we focus on query processing over federations of SPARQL endpoints. Actually, query engines view SPARQL endpoints as federations of distributed and autonomous sources, sharing their data to answer complex queries [14, 40]. Next, we present the procedure of federated query processing employed by query engines, followed by the illustration of two state of art query engines, namely FedX [48] and Anapsid [1, 2].

2.4.1 Federated query processing

In federated query processing, referred also as virtual integration [14], a query is split into subqueries that can be answered from a federation of data providers. This procedure is employed by a federated query processor, named query engine. The federation is transparent to the end user i.e., the distinct data sources can be queried as if they were a single RDF graph. Challenges of federated query processing over the Linked Data, concern the conception of a query plan and its distributed execution [36].

We present below this procedure, based on distributed query processing over relational database systems [22, 37] and which is adapted in the context of the Linked Data [19]. Given a SPARQL query and a federation of SPARQL endpoints, a federated query engine performs the following tasks, as we see in Figure 2.3.

1. Query parsing/rewriting: Checks if the input query is valid regarding the SPARQL protocol and, if necessary, rewrites and normalizes it. Regardless the optimization techniques of the query engine this phase rewrites federated queries into equivalent but more e cient ones, thus producing alternative execution plans. Typical transformations are the elimination of redundant predicates or simpli cation of expressions. An example of query rewriting, is the transformation of R 1 S into (R 1 S) [(R n S). In typical SPARQL optimization based on Relational Algebra [47], query rewriting rules are used in order to de ne equivalent SPARQL expressions that minimize the execution cost for data consumers, such as lter pushing.

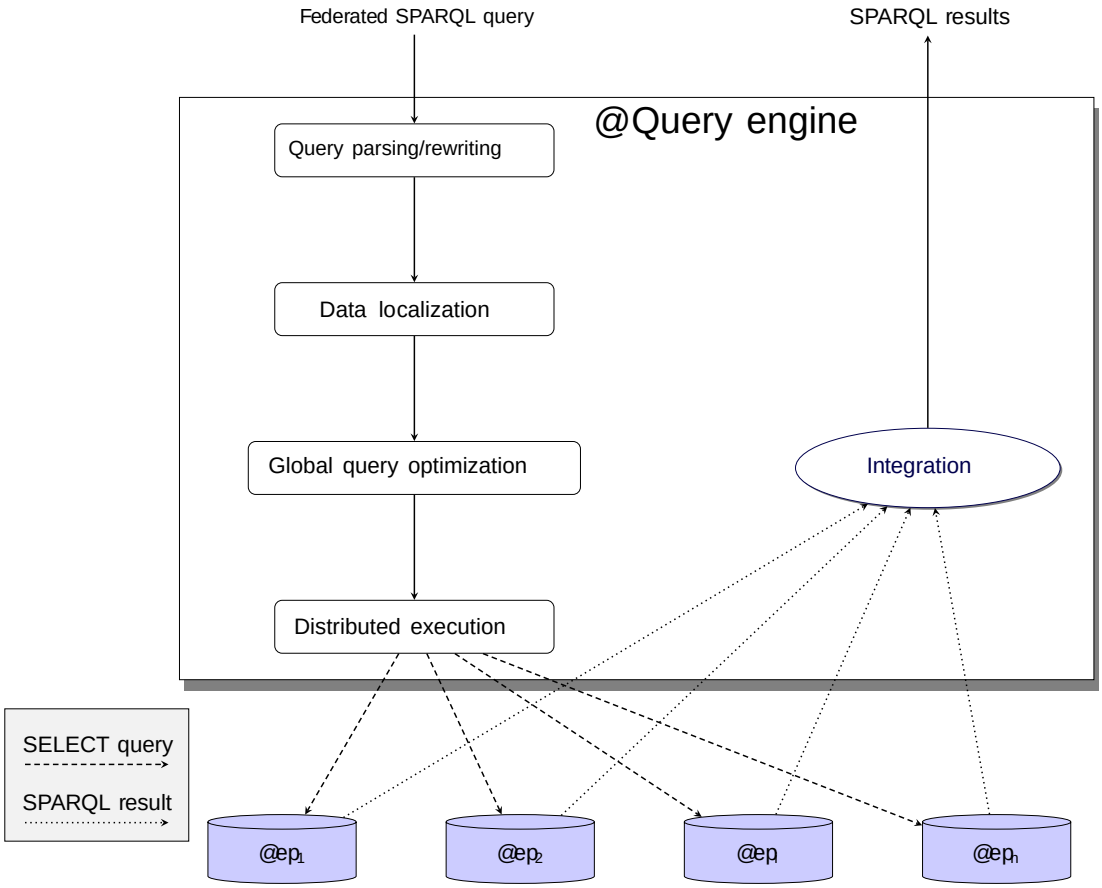


Figure 2.3 Federated query processing model [37].

2. Data localization: Performs source selection among a user-defined federation of trusted SPARQL endpoints and rewrites the query into a decomposed set of sub-queries. Most approaches are based on a Triple-Pattern-Wise Source Selection (TPWSS) [12, 34, 40, 43, 48]. In this strategy, even if the join produces a non-empty result set, some selected sources may not contribute to the retrieved data when joined with others. Consequently, a possible overestimation of data sources may decrease the performance of query processing by increasing network traffic and intermediate results. On the other hand, join-aware TPWSS strategies have been proposed to reduce this problem based on predicates of triple patterns such as [2, 42]. There exist also some other approaches that are not in the scope of this thesis [36].

3. Global query optimization: Optimizes the adopted query plan, by rewriting it using various heuristics [9, 12, 48] such as, grouping evaluation of triple patterns to the same source, minimizing intermediate results, minimizing number of calls, etc. As an extension, dynamic oriented approaches adjust their planning based on load and availability of sources [2]. Note, that the cardinality estimation of matching data is not based on statistics, as a reliable source providing this information does not exist to the best of our knowledge.

4. Distributed query execution: Deploys physicals operators in order to evaluate the plan established in the previous steps. As presented in Section 2.2, the evaluation of

a join may be either (a) *pipelined*, where results are incrementally produced by the query engine, or (b) *blocking* where intermediate results are blocked when SPARQL endpoints are temporary unavailable[15].

@p_A
s1 p1 o1
s1 p1 o2
s2 p1 o2
s3 p1 o3
c3 p3 titi
c4 p3 titi
c1 p4 a
c2 p4 b

(a) Dataset of p_A

@p_B
s1 p2 o3
s2 p2 o4
s3 p2 o1
s4 p2 o2
c1 p5 toto
c2 p5 toto

(b) Dataset of p_B

Table 2.3 Example of simplified datasets of two SPARQL endpoints, hosted by p_A and p_B data providers respectively.

In next section, we overview two state of art query engines, namely FedX and Anapsid, and illustrate how each query engine evaluates the federated query processing model with its own optimization techniques.

2.4.2 State of art query engines: FedX and Anapsid

FedX [48], is a framework that follows an on-demand approach to setup a federation of SPARQL endpoints at query time. This query engine has the advantage that it does not need any preprocessed metadata such as statistics and indices to discover and consume Linked Data, but is based only on the list of relevant SPARQL endpoints defined by the user. Figure 2.4 presents the procedure of federated query processing of FedX. We use the simplified datasets of Tables 2.3a and 2.3b, to explain the definitions of this section[16]. The set of heuristics and optimization techniques established by FedX for efficient query processing, are namely:

1. Statement sources: Discovers the relevant sources able to answer each triple pattern of a federated query through SPARQL ASK queries, given the user defined list of

[15] Note that Integration is considered as a post step of federated query processing.
[16] Note that, like in the previous section, we removed prefixes from predicates for simplicity.

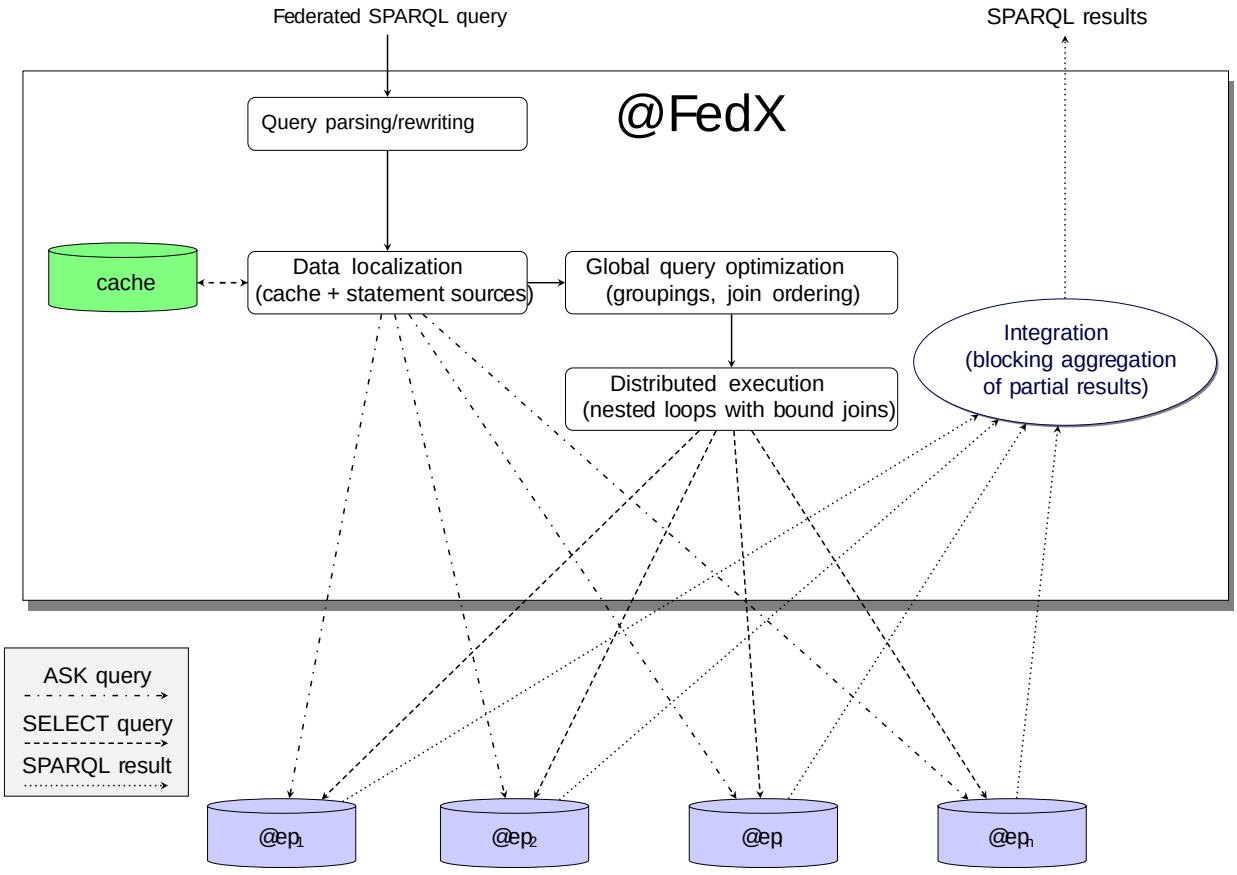


Figure 2.4 Federated query processing model of FedX.

LD provider	IP	Time	Subquery	Answer
p_A	ip_1	10:00:01	SELECT ?z WHERE { ?z p1 o2 }	f ?z 7! f s1, s2gg
p_B	ip_1	10:00:04	SELECT ?y_0 ?y_1 WHERE { { s1 p2 ?y_0 } UNION { s2 p2 ?y_1 } }	f ?y_0 7! f o3g; ?y_1 7! f o4gg

Table 2.4 Federated query log of SELECT ?z ?y WHERE f ?z p1 o2 : ?z p2 ?yg traces, produced by FedX query engine with ip_1 IP address and executed over the federation of SPARQL endpoints hosted by p_A and p_B data providers.

targeted SPARQL endpoints. This technique is used in conjunction with an adaptive cache, that learns after executing each federated query the location(s) of its triple patterns.

2. Groupings: Groups the evaluation of multiple triple patterns to the same SPARQL endpoint using the information extracted in the previous step. Thus, joins are pushed to the SPARQL endpoint hosting the largest subtotal of the triple patterns that need to be evaluated and local processing at the client is minimized. This type

of joins are called *exclusive groups*.

3. Join ordering: Reorders the joins between triple patterns by using variable counting techniques, in order to choose the most effective evaluation order. FedX implements a rule-based join function to choose iteratively the next triple pattern to be evaluated. This function is an extension of the variable counting strategy [54], where unbound variables of a triple pattern are counted by excluding those that are common with variables of previously evaluated triple patterns. The next triple pattern to be evaluated is the one with less unbound variables.

4. Nested-loops with bound joins: In conjunction with a nested-loop, it computes joins in a block to minimize requests to the targeted sites. Before applying bound joins, all matching triples of the outer dataset are retrieved. Then, mapping results of these triple patterns i.e., literals/IRIs, are grouped into subqueries using SPARQL UNION constructs. Each subquery is sent to the relevant sources and used to search matching triples in the inner dataset.

   Consider that query SELECT ?z ?y WHERE f ?z p1 o2 : ?z p2 ?yg, is executed over the federation of SPARQL endpoints of Tables 2.3a and 2.3b. FedX chooses to start the evaluation with the first triple pattern, based on the join ordering strategy we presented above. In the second entry of Table 2.4, we see an example of a bound query, where mappings of tp_1 = f ?z p1 o2g i.e., $_{tp_1}(?z)$ = f s1; s2g from the first entry of the same table are used to evaluate tp_2 = f ?z p2 ?yg through a nested-loop. The number of produced bound join queries, depends on the number of mappings of the outer dataset and the block size of bound queries which is configurable by the user. For our example, as the cardinality of mappings of $_{tp_1}(?z)$ is 2 and for a block size also equal to 2, FedX will send one bound query.

Anapsid [1, 2] is an adaptive query processing engine, that attempts to minimize the workload of SPARQL endpoints by adapting its query execution to data availability and run-time conditions. In order to do so, Anapsid provides with non-blocking implementations of physical join operations, that opportunistically produces results as quickly as they are retrieved from relevant sources. Figure 2.5 presents the procedure of federated query processing of Anapsid. The set of heuristics and optimization techniques established by Anapsid for efficient query processing are:

1. Schema alignments: Obtains the ontologies of datasets of SPARQL endpoints and stores them in form of a catalogue. This catalogue is expressed as the set of predicates that can be answered by each SPARQL endpoint, and which is exploited during data localization.

2. Adaptive source selection: Selects SPARQL endpoints that can answer a query. For this, it uses sampling techniques [31] to adapt on the execution context, namely: i) Star Shaped Group Multiple sources or SSGM, where a triple pattern is evaluated by the set of SPARQL endpoints that can give answers, ii) Star Shaped Group Single source or SSGS, where a triple pattern is evaluated according to SSGM rules but

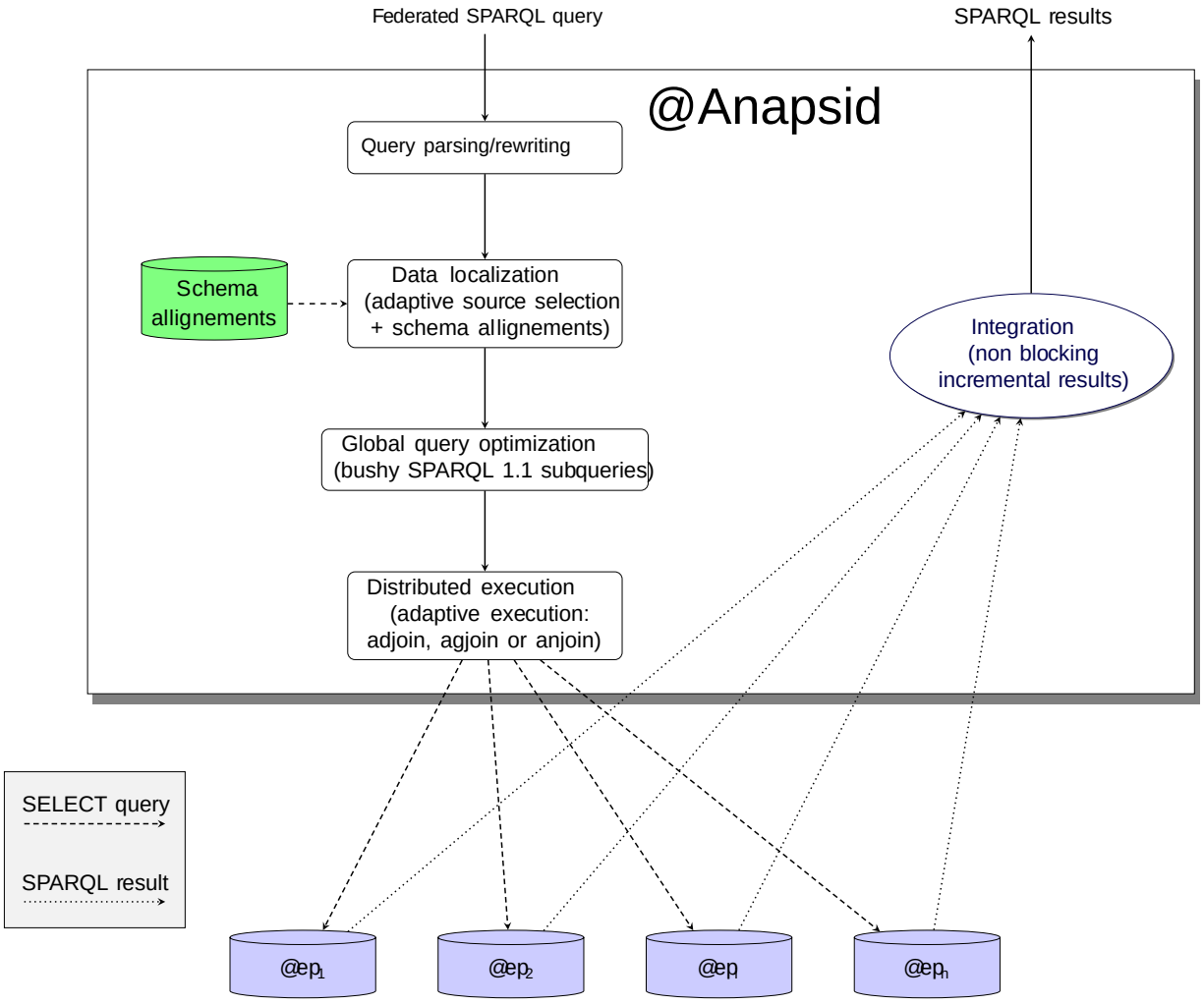


Figure 2.5 Federated query processing model of Anapsid.

LD provider	IP	Time	Subquery	Answer
p_A	ip_1	11:30:15	SELECT ?z WHERE { ?z p1 o2 }	f < ?z 7! f s1, s2g > g
p_B	ip_1	11:30:17	SELECT ?y WHERE { { ?z p2 ?y } FILTER ((?z="s1") || (?z="s2"))}	f < ?y 7! f o3; o4g > g

Table 2.5 Federated query log of SELECT ?z ?y WHERE f ?z p1 o2 : ?z p2 ?yg traces, produced by Anapsid query engine with ip_1 IP address and executed over the federation of SPARQL endpoints hosted by p_A and p_B data providers.

also by choosing the rst SPARQL endpoint which con rms that can evaluate this triple pattern, and iii) Exclusive Groups, which is not recommended but only created to be compared with FedX.

3. Bushy-tree query decomposition and rewriting. Decomposes user queries into multiple subqueries and eventually rewrites them in SPARQL 1.1. These subqueries

are produced using an estimation function cost, which is based on the hypothesis that subqueries with triple patterns sharing exactly one variable have small-sized cardinality of answers. This gives an execution plan in form of a balanced tree, compared to the left linear tree of FedX where each pattern is individually evaluated and subsequently its mappings are pushed to the next one. Thus, as the depth of the execution tree is minimized, the parallelization of the processing workload is maximized during the adaptive evaluation of the query.

4. Adaptive query execution Employs physical join operators in order to adapt on the execution context, during query evaluation. These operators are: (i) Adaptive Group Join (Agjoin) , a combination of symmetric hash and Xjoin , in order to integrate the results as they are produced, (ii) Adaptive Dependent Join (Adjoin), an extension of Agjoin where results are produced when both SPARQL endpoints are available and not asynchronously, and nally (iii) Adaptive Nested Join (Anjoin), an extension of nested-loop deployed when selectivity between triple patterns is not balanced.

Consider again query SELECT ?z ?y WHERE f ?z p1 o2 : ?z p2 ?yg. Anapsid using SSGM or SSGS, chooses to start the evaluation with the rst triple pattern based on the join ordering strategy we presented above. In the second entry of Table 2.5, we see a query with two FILTER options produced with the anjoin operator, where mappings of tp_1 = f ?z p1 o2g i.e., $_{tp_1}(?z)$ = f s1; s2g from the rst entry of the same table are used to evaluate tp_2 = f ?z p2 ?yg through a nested-loop. The number of produced FILTER join queries depends on the number of mappings of the outer triple pattern and a static upper bound for FILTER options which is employed by Anapsid. So for our example, as the cardinality of mappings of $_{tp_1}(?z)$ is 2 and for a block size equal to 2, we have one FILTER query.

In Sections 2.1 - 2.4, we presented two approaches to consume Linked Data using either Triple Pattern Fragments or the procedure of federated query processing. Next, we formally de ne the problem we aim to solve: how to reverse BGPs of user queries from logs of their execution traces.

2.5 Formal problem statement

As pointed in Chapter 1, the limitation of query processing over the Linked Data, is that data providers are not aware of queries they process; they just observe subqueries of the original user queries. In this thesis we aim to answer this limitation (a) over single or federations of TPF servers, and, (b) over federations of SPARQL endpoints. In order to do so, we use the example of Bob and DBpedia sites, as presented on page 17 in Chapter 1, which triples are presented in Tables 2.6b and 2.6c, respectively.

De nition 4 (Query log) A log of one or more Linked Data providers is a sequence of execution traces structured in tuples $\langle p; ip; ts; q; r\rangle$ where p is a data provider, ip is the ip address of the client, ts is the timestamp of the http request, q is a query, and r is the

Prefix	IRI
alice	<http://example.org/alice#me/>
bob	<http://example.org/bob#me/>
dbpedia	<http://dbpedia.org/resource/>
dcterms	<http://purl.org/dc/terms/>
europeana	<http://data.europeana.eu/item/>
foaf	<http://xmlns.com/foaf/0.1/>
schema	<http://schema.org/sameAs/>
rdf	<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
wiki	<http://www.wikidata.org/entity/>
xsd	<http://www.w3.org/2001/XMLSchema#>

(a) IRI prefixes

@Bob
(bob:me, rdf:type, foaf:Person)
(bob:me, foaf:knows, alice:me)
(bob:me, schema:birthDate, "1990-07-04"^^xsd:date)
(bob:me, foaf:topic_interest, wiki:Q12418)

(b) Dataset triples of Bob

@DBpedia
(wiki:Q12418, dcterms:creator, dbpedia:Leonardo_da_Vinci)
(europeana:243FA...4D619, dcterms:subject, wiki:Q12418)
(wd:Q12418, dcterms:title, "Mona Lisa")

(c) Dataset triples of DBpedia (concerning "Mona Lisa")

Table 2.6 Dataset triples of DBpedia and Bob data providers.

set of matching RDF triples, returned as a response to tp from data provider p. Note that for a single data provider, log traces are totally ordered, while, for a federation of data providers, log traces are partially ordered as queries may be received in different sites at same time.

We denote by $E(Q_i)$ the execution trace of query Q_i, which consists of log of sub-queries produced when a data consumer executes the SPARQL query Q_i. We represent execution traces of n concurrent queries by $E(Q_1 \,||\, \cdots \,||\, Q_n)$.

Consider again Q_I = SELECT ?artifact WHERE { bob:me foaf:topic_interest ?artifact . ?artifact dcterms:creator dbpedia:Leonardo_da_Vinci }, of Figure 1.3 on page 17. Triple patterns tp_1 and tp_2 of Q_I are evaluated at Bob's and DBpedia's sites, respectively. A federated log corresponding to the execution of Q_I is presented in Table 1.1 on page 18. Suppose that the user with ip_1 IP Address, concurrently poses another query Q_{II} = SELECT * WHERE { ?artifact dcterms:title ?title }. This query corresponds to $BGP_2 = \{tp_3\}$, where tp_3 = { ?artifact dcterms:title ?title }. The federated log of DBpedia and Bob for a concurrent execution of Q_I and Q_{II}, denoted as $E(Q_I \,||\, Q_{II})$, is presented in Table 1.2 on page 20.

De nition 5 (BGP reversing) Given a log corresponding to the execution of one query $E(Q_i)$, nd a function $f(E(Q_i))$ producing a set of BGPs $\{BGP_1, ..., BGP_n\}$, such that $f(E(Q_i))$ approximates ( ) the BGPs existing in the original query. Thus, if we consider that $BGP(Q_i)$ returns the set of BGPs of Q_i then $f(E(Q_i)) \approx BGP(Q_i)$.

We consider that a BGP approximates another ( ) if both contain same triple patterns and same joins. We evaluate the quality of f with the precision and recall of triple patterns and joins returned by f against those existing in original queries. If $f(E(Q_I))$ produces the $BGP = \{tp_1 \bowtie tp_2\}$, then precision and recall of triple patterns and joins are perfect according to the BGP present in Q_I.

Property 1 (Resistance to concurrency) The reversing function f should guarantee that BGPs obtained from execution traces of isolated queries, approximate ( ) results obtained from execution traces of concurrent queries: $f(E(Q_1)) \cup ... \cup f(E(Q_n)) \approx f(E(Q_1 \parallel ... \parallel Q_n))$.

If Q_I and Q_{II}, were sent by two di erent IP addresses, it is possible to separate $E(Q_I \parallel Q_{II})$ into $E(Q_I)$, $E(Q_{II})$ and apply the reversing function to each trace. However, in the worst case, Q_I and Q_{II} have the same IP address i.e., a web application running on the cloud that runs queries Q_I and Q_{II} in parallel. In this case, if $f(E(Q_I \parallel Q_{II}))$ produces the $BGP = \{tp_1 \bowtie tp_2 \bowtie tp_3\}$ then recall in joins is perfect. Precision of joins, although, is equal to 0.33 as $f(E(Q_I \parallel Q_{II}))$ deduce two additional false joins i.e., $\{tp_1 \bowtie tp_3\}$ and $\{tp_2 \bowtie tp_3\}$. Thus, we expect that $f(E(Q_I \parallel Q_{II})) \approx f(E(Q_I)) \cup f(E(Q_{II}))$.

Next chapters, explore if existing approaches are able to solve the problem that was formally de ned in this section and then present new ones. First, in Chapter 3 we address this problem using Data Mining algorithms to identify BGPs as frequent sets of triple patterns. Then, in Chapter 4 we propose an approach to solve the BGP reversing problem over single or federations of TPF servers. Finally, in Chapter 5 we propose an approach to solve the BGP reversing problem over federations of SPARQL endpoints.

3

State of art: Data Mining

Contents

Extracting information from raw logs is a task related to the Data Mining process, known also as Knowledge Data Discovery (KDD). Data Mining algorithms have been extensively used to extract knowledge from web logs. Therefore, Data Mining could be used to solve our scientific problem, by considering each predicate, triple pattern or subquery as a requested resource via the web on the data provider. The question is: "Can Data Mining algorithms extract BGPs of queries based on the occurrences of sequences of their triple patterns?" In this chapter, we aim to explore if this is possible.

First, we briefly present web usage mining, in Section 3.1. Then, we concentrate on sequential pattern mining, by illustrating its main approaches and presenting two state of art algorithms, WINEPI and MINEPI, in Section 3.2. Third, we apply the MINEPI algorithm as to solve our problem in Section 3.3. Thereafter, we identify the limitations of sequential pattern mining when applied over logs of Linked Data providers, in Section 3.4. Finally, we present an extension of MINEPI with either pre-processing data

transformation or post-processing constraints which we developed to solve our problem in Section 3.5.

3.1 Web usage mining

Web mining [38, 52] is the application of Data Mining techniques to find interesting and potentially useful knowledge from web data, which is divided into: (a) *web content mining*, which extracts useful information from a diversity of web content such as audio, video or text, (b) *web structure mining*, which models the web based on the topology of hyperlinks and tags and (c) *web usage mining*, which aims to understand the behaviour of users in interacting with the web or within a website, as they navigate from web resource to web resource.

We argue that our work is related to *web usage mining*. Logs of execution traces actually correspond to subqueries or simply triple patterns accessed via the HTTP protocol on data providers. Consider the abstract log of Table 3.1, corresponding to execution plans of queries Q_A = SELECT ?x ?y WHERE { ?x p1 o1 . ?x p2 ?y . ?y p3 ?z }, Q_B = SELECT ?y WHERE { ?x p2 ?y }, Q_C = SELECT ?y WHERE { ?y p3 ?z } and Q_D = SELECT ?x WHERE { ?x p1 o1 }. In this chapter, we aim to explore if the scientific problem we address, as presented on page 39 in Chapter 2, is equivalent to associate sets of triple patterns that are accessed on data providers via the web. The intuition is to explore if joins of triple patterns, which are evaluated by query engines or TPF clients in multiple blocks for optimization reasons, can be detected based on their occurrences in server logs.

Various algorithms have been proposed in web usage mining in order to find web usage patterns either based on association ruling, clustering, classification or simply statistical knowledge extraction. The approach that identifies causal relations between webpages, is association ruling. As we aim to discover which sets of triple patterns are joined together from a sequence of subqueries/triple patterns evaluated through the HTTP protocol on data providers, we overview the association rule-based approach of sequential pattern mining. In this case, frequent episodes of accessed webpages are identified, by viewing a web historic journal as a sequence of timestamped URLs.

In the next section, we overview approaches of *sequential pattern mining* and position our interest to the approach that is most suitable to answer our scientific problem, namely sequential mining over *temporal sequences*.

3.2 Sequential pattern mining

Sequential pattern mining [32] discovers frequent *episodes* from a sequence of *events*. An event is a collection of totally or partially ordered items[1], where the set of all different items composes the *alphabet*. An episode is a set of events. The aim of sequential pattern mining is to discover the sets of frequent episodes, in a log of either fixed or dynamic size. Such episodes can be represented as acyclic digraphs and are thus more general than linearly ordered sequences. Frequent episodes are identified using a threshold value which is represented either as *frequency* or *support*, in order to calculate the ratio or the number of occurrences respectively of an episode in a log.

Sequential mining algorithms follow techniques similar to *association rule mining* in

[1] Serial and parallel class of events correspond to totally or partially ordered items, respectively.

	LD provider	IP	Time	HTTP request
[1]	p_A	ip_1	11:24:19	http : ==pa:com=sparql= &query = SELECT ?x ?y { ?x p1 o1 } &format = json &timeout = 0
[2]	p_A	ip_1	11:24:23	http : ==pa:com=sparql= &query = SELECT ?y { s1 p2 ?y } &format = json &timeout = 0
[3]	p_B	ip_1	11:24:24	http : ==pb:com=sparql= &query = SELECT ?z { o3 p3 ?z } &format = json &timeout = 0
[4]	p_A	ip_1	11:24:27	http : ==pa:com=sparql= &query = SELECT ?y { s2 p2 ?y } &format = json &timeout = 0
[5]	p_B	ip_1	11:24:28	http : ==pb:com=sparql= &query = SELECT ?z { o4 p3 ?z } &format = json &timeout = 0
[6]	p_A	ip_1	11:24:30	http : ==pa:com=sparql= &query = SELECT ?y { ?x p2 ?y } &format = json &timeout = 0
[7]	p_B	ip_1	11:24:31	http : ==pb:com=sparql= &query = SELECT ?y { ?y p3 ?z } &format = json &timeout = 0
[8]	p_A	ip_1	11:24:36	http : ==pa:com=sparql= &query = SELECT ?x { ?x p1 o1 } &format = json &timeout = 0

Table 3.1 HTTP log of of Q_A - Q_D traces, produced by data consumer with ip_1 IP Address and executed over the federation of p_A and p_B data providers. SPARQL results are requested in json format with execution timeout = 0.

	LD provider	IP	Time	HTTP request
[1]	p_A	ip_1	11:24:19	URL_1
[2]	p_A	ip_1	11:24:23	URL_2
[3]	p_B	ip_1	11:24:24	URL_3
[4]	p_A	ip_1	11:24:27	URL_2
[5]	p_B	ip_1	11:24:28	URL_3
[6]	p_A	ip_1	11:24:30	URL_4
[7]	p_B	ip_1	11:24:31	URL_5
[8]	p_A	ip_1	11:24:36	URL_1

Table 3.2 HTTP log of web pages, accessed by the data consumer with ip_1 IP Address, over the federation of p_A and p_B data providers. The log is represented as a temporal sequence

order to discover causal relations between events. Although, the difference with traditional Data Mining is that sequential mining views data as a sequence. Therefore, sequential pat-

tern mining can be applied over different dataset formats such as transactional-oriented, streams, time series, etc.

Examples of raw data analyzed through sequential pattern mining include genome searching, web logs, alarm data in telecommunications networks, population health data, etc. In Table 3.2, we see an abstract example of an HTTP navigation journal. All different events composing the alphabet of this example, are URL_1; URL_2; URL_3; URL_4 and URL_5. For a threshold defined by the user as equal to 2, the deduced episodes are: episode$_A$ = { URL_1 }, episode$_B$ = { URL_2 }, episode$_C$ = { URL_3 } and finally episode$_D$ = { URL_2; URL_3 }, all with occurrences$_{episode}$ = 2.

Next, we overview the main categories of sequential pattern mining, namely apriori-based, pattern growth algorithms and temporal sequences, in Section 3.2.1. Subsequently, we focus on two state of art algorithms applied over temporal sequences, WINEPI and MINEPI, that could be used to solve our problem, in Section 3.2.2.

3.2.1 Approaches and techniques

Depending on the dataset format and the generation method of episodes, sequence mining algorithms are divided into three broad classes [32], we briefly overview below:

(A) Apriori-based: This family of algorithms discovers frequent sets of events that appear in different transactions. In particular, they transform transactions into sequences and apply on them the apriori approach [4] in order to generate association rules. These algorithms are divided depending on how data are stored, into horizontal and vertical. Horizontal e.g., AprioriAll, AprioriSome, or DynamicSome, save the data by their "Transaction Id" and sort them by "Customer Id" and "Transaction Time". Vertical e.g., SPADE, SPAM or CCSM, transform their data in event-oriented lists i.e., for each event there exist a list of pairs (sequence id; timestamp i). Vertical compared to horizontal algorithms, are used to apply depth-first approach to the mining and then employ pattern growth methods. Independently of their taxonomy, once these algorithms transform transactions into sequences, they apply two phases (1) candidate generation, where episodes are generated in different ways e.g., maximal sequences, hash trees or prefix tree, each with a particular cost in space and time, and, (2) pruning: where candidate episodes are considered as frequent, based on the user defined threshold. The limitation of apriori-based family, is the exponential number of generated episodes. Some works address this problem using constraints i.e., conditions to remove generated episodes such as episode length, time gap between events, etc.

In Figure 3.3, we see an example of transactions transformed into sequences. Note that items of a transaction are considered as non ordered i.e., parallel, when transformed into a sequence. For instance, customer with ID=1 bought on December 12, 2016 the items <a, b, c>, which are denoted in the sequence with ID=1 as (a b c) parallel events. Once transactions are transformed into sequences, apriori based algorithms are able to extract the most frequent episodes. In our example, for support = 5 and serial class of events, the most frequent episodes of maximum

Customer ID (CID)	Transaction Item	Items bought
1	December 09, 2016	< a >
1	December 12, 2016	< a, b, c >
1	December 15, 2016	< a, c >
1	December 18, 2016	< d >
1	December 20, 2016	< c, f >
2	November 5, 2016	< a, d >
2	November 7, 2016	< c >
2	November 12, 2016	< b, c >
2	November 22, 2016	< a, e >
3	November 23, 2016	< e, f >
3	December 1, 2016	< a, b >
3	December 10, 2016	< d, f >
3	December 12, 2016	< c >
3	December 14, 2016	< b >
4	November 12, 2016	< e >
4	November 15, 2016	< g >
4	November 20, 2016	< a, f >
4	December 1, 2016	< c >
4	December 10, 2016	< b >
4	December 20, 2016	< c >

(a) Transaction oriented DB, sorted by "Customer ID "

Sequence ID (SID)	Sequence
1	< a(abc)(ac)d(cf) >
2	< (ad)c(bc)(ae) >
3	< (ef)(ab)(df)cb >
4	< eg(af)cbc >

(b) Sequential oriented DB, stored in an horizontal format

Table 3.3 Transformation of transaction-oriented into sequence-oriented DB.

length is $\{a; b; c\}$ as it has 5 occurrences. The episode $\{a; b; c\}$ appears in different subsequences i.e. $a(\_bc)$ and (abc) in the first sequence, $(a\_)(bc)$ in the second, $(ab)c$ in the third and finally $(a\_)bc$ in the fourth.

Regarding our scientific problem, we could apply the Apriori based algorithms to extract BGPs of user queries by considering an execution log as a single transaction. But with this approach, we do not use any more timestamps and all events are unordered i.e., considered as parallel. Consequently, this will produce a large number of generated episodes. Apriori based algorithms would correlate triple patterns even if they originally were captured in distant timestamps in the log, or, correlate

triple patterns of the inner operand of a nested-loop that originally were captured before triple patterns that seems to be the outer operand.

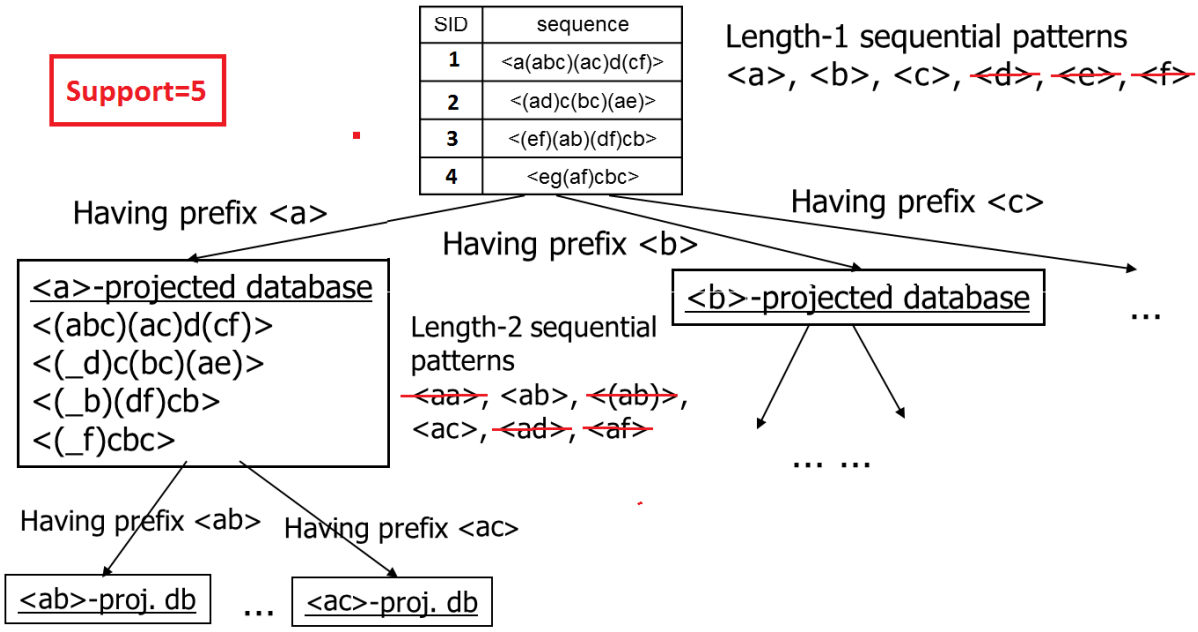


Figure 3.1 Extraction of frequent episodes with pattern growth algorithms, by projecting only subsequences with frequent pre xes.

(B) Pattern growth algorithms : Even when constraints are employed by apriori-based algorithms, the number of generated episodes is still high especially when the datasets are large. On the other hand, pattern growth algorithms while generally more complex to develop, test and maintain, can be faster with large volumes of data. In order to do so, these algorithms e.g., FreeSpan, Pre xSpan, SLPMiner, apply the Frequent Pattern growth (FP growth) paradigm. In this case, frequent episodes are compressed into a database represented as a frequent pattern tree, which is subsequently divided into a set of projected databases during the generation of episodes. The main idea, is to save a subsequence into the frequent pattern tree only if its pre x is frequent enough. With this approach pattern growth algorithms, compared to apriori based, are able to extract progressively frequent episodes and in general with only one scan of the input dataset.

For the example of Figure 3.3, consider again that $support = 5$. First, regarding serial class of events, subsequences with frequent pre xes of length $= 1$ are projected i.e., subsequences with pre xes $< a >$, $< b >$ and $< c >$, as $occurrences_{<a>} = 7$, $occurrences_{<b>} = 5$ and $occurrences_{<c>} = 7$. Then, all episodes of length $= 2$ are generated and then subsequences that have these episodes as pre x, are projected. For instance, only subsequences with pre xes $< a; b >$ and $< a; c >$ are then projected, as $occurrences_{<b>} = 5$ and $occurrences_{<c>} = 8$ in the $< a >$ projected database For this projected database, the occurrences of episodes $< a; b >$ and $< a; c >$ are calculated as $occurrences_{<ab>} = occurrences_{<b>} = 5$

and occurrences$_{\langle ac \rangle}$ = occurrences$_{\langle c \rangle}$ = 8 respectively.

Regarding our scientific problem, like *apriori-based*, we could use *pattern growth* algorithms to extract BGPs of user queries by considering an execution log as a single transaction. But again with this approach, we do not use any more the timestamps and all events of a single log are unordered i.e., considered as parallel.

(C) Temporal sequences : Sequence mining is not applied only for data stored in distinct and independent database instances. The need of events that are statistically dependent emerges in some domains i.e., for events that are episodic in nature. In such domains, data can be viewed as series of events occurring at specific times and therefore the problem becomes a search for collections of events that occur frequently together. There exist various algorithms such as MINEPI, WINEPI or PROWL, that are actually apriori-like[2] and for which the FP growth paradigm also holds. The limitation of such approaches is that the size of generated episodes may be still important regardless the FP growth paradigm, as it depends on the user-defined threshold.

Table 3.2 on page 46, corresponds to an example of timestamped HTTP log, used directly to apply algorithms of temporal sequences. Similarly to web logs, our log is formatted as a sequence of timestamped subqueries/triple patterns that are accessed with the HTTP protocol on data providers. Therefore, we argue that the problem of BGP reversing is related to sequential pattern mining over temporal sequences.

In the next section, we present two state of art sequential mining algorithms over temporal sequences, namely WINEPI and MINEPI .

3.2.2 State of art algorithms: WINEPI and MINEPI

WINEPI [26] decomposes a temporal sequence into overlapping sliding windows which size is defined by the user, and thereafter calculates frequencies of episodes over these windows. MINEPI [27] instead, looks for all minimal occurrences of episodes into a specific time interval. A minimal occurrence is an interval such that no sub-interval contains the episode. The minimum threshold of minimal occurrences of an episode is called support. The minimum frequency (for WINEPI), the minimum support (for MINEPI) and the maximum window size (for both), are thresholds defined by the user.

In Figure 3.2 we see an example of temporal sequence[3]. This timestamped log of events is used to illustrate, step by step, how WINEPI and MINEPI are employed to extract frequent episodes of events and deduce the association rules between them.

[2] Note that all approaches of sequential pattern mining are *apriori-like* , as they aim to generate association rules between events of frequent episodes.

[3] This example is taken from [21].

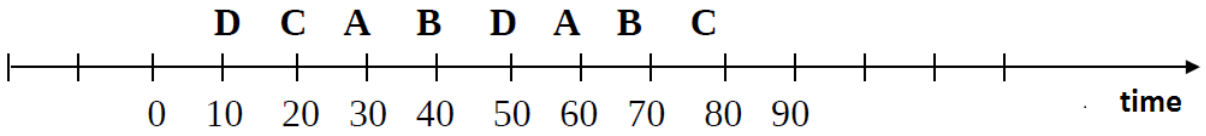


Figure 3.2 Abstract example of a temporal sequence, used as input to WINEPI and MINEPI.

Figure 3.3, presents how WINEPI parses the temporal sequence, during the interval $[20; 120]$ and over 11 sliding windows. Figure 3.5 on page 52, presents all sets of deduced episodes, per sliding window. For instance, we observe that the episode $(A; B; D)$, considering parallel class of events, is identified in 5/11 windows i.e. frequency$_{(A;B;D)} = 0,45$.

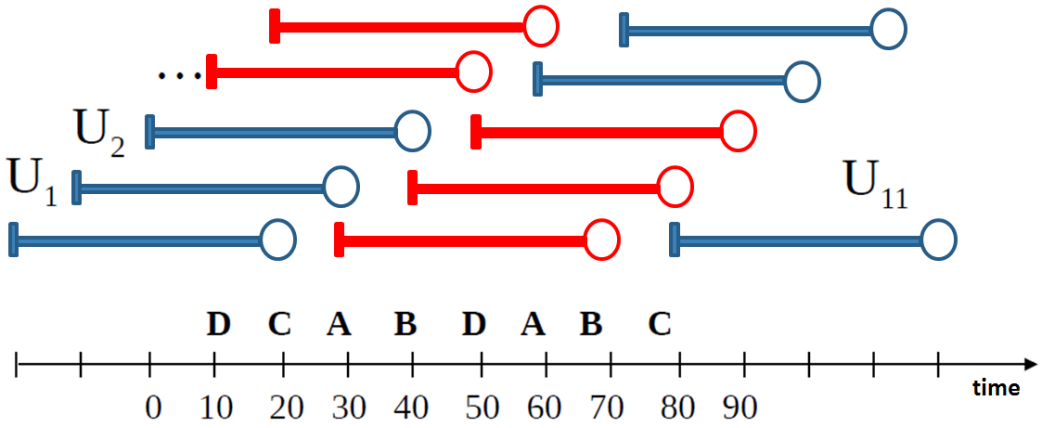


Figure 3.3 Sliding windows of length = 40 for WINEPI over the temporal sequence in interval $[0; 120[$. Episodes containing $A; B; D$ are identified in windows U_4, U_5, U_6, U_7 and U_8 (in red color).

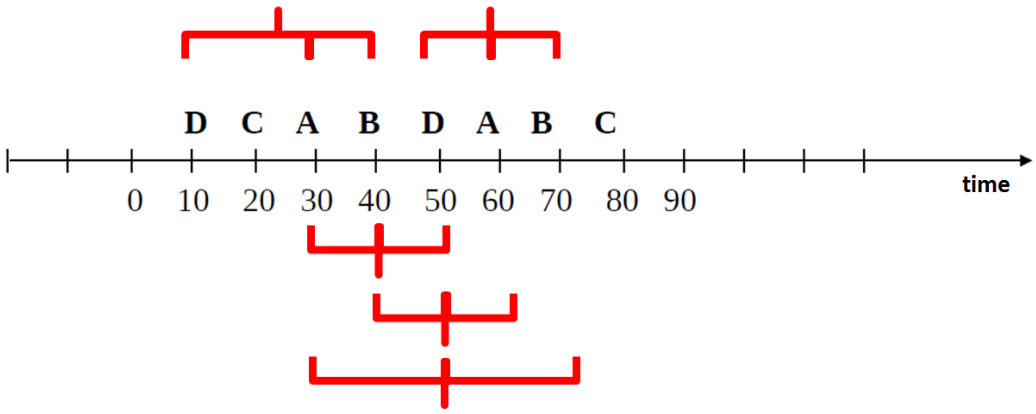


Figure 3.4 Minimal occurrences for MINEPI over the temporal sequence in interval $[0; 120[$. Episodes containing $A; B; D$ are identified in intervals $[10; 40]$, $[30; 70]$, $[30; 50]$, $[50; 70]$ and $[40; 60]$ (in red color).

Figure 3.4 presents how MINEPI identifies all frequent episodes, over the temporal sequence in interval $[20; 120]$. Figure 3.6 presents all sets of deduced episodes over this

Window U_i	Contents of U_i	Parallel episodes occurring in U_i
$U_{1,[-20,20[}$	$[\_, \_, \_, D]$	$\{D\}$
$U_{2,[-10,30[}$	$[\_, \_, D, C]$	$\{C, D\}$, $\{CD\}$
$U_{3,[0,40[}$	$[\_, D, C, A]$	$\{A, C, D\}$, $\{AC, AD, CD\}$, $\{ACD\}$
$U_{4,[10,50[}$	$[D, C, A, B]$	$\{A, B, C, D\}$, $\{AB, AC, AD, BC, BD, CD\}$, $\{ABC, \underline{ABD}, ACD, BCD\}$, $\{ABCD\}$
$U_{5,[20,60[}$	$[C, A, B, D]$	$\{A, B, C, D\}$, $\{AB, AC, AD, BC, BD, CD\}$, $\{ABC, \underline{ABD}, ACD, BCD\}$, $\{ABCD\}$
$U_{6,[30,70[}$	$[A, B, D, A]$	$\{A, B, D\}$, $\{AB, AD, BD\}$, $\{\underline{ABD}\}$
$U_{7,[40,80[}$	$[B, D, A, B]$	$\{A, B, D\}$, $\{AB, AD, BD\}$, $\{\underline{ABD}\}$
$U_{8,[50,90[}$	$[D, A, B, C]$	$\{A, B, C, D\}$, $\{AB, AC, AD, BC, BD, CD\}$, $\{ABC, \underline{ABD}, ACD, BCD\}$, $\{ABCD\}$
$U_{9,[60,100[}$	$[A, B, C, \_]$	$\{A, B, C\}$, $\{AB, AC, BC\}$, $\{ABC\}$
$U_{10,[70,110[}$	$[B, C, \_, \_]$	$\{B, C\}$, $\{BC\}$
$U_{11,[80,120[}$	$[C, \_, \_, \_]$	$\{C\}$

Figure 3.5 Frequent episodes of the temporal sequence in interval $[0, 120[$ for WINEPI with sliding windows of length = 40.

Minimal occurrences	Episode : # occurs
D 10-10 50-50	D : 2
C 20-20 80-80	C : 2
A 30-30 60-60	A : 2
B 40-40 70-70	B : 2
D C 10-20 50-80	D C : 2
D A 10-30 50-60	D A : 2
D B 10-40 50-70	D B : 2
C D 20-50	C D : 1
C A 20-30	C A : 1
C B 20-40	C B : 1
A D 30-50	A D : 1
A C 60-80	A C : 1
A B 30-40 60-70	A B : 2
B D 40-50	B D : 1
B C 70-80	B C : 1
B A 40-60	B A : 1
D C A 10-30	D C A : 1
D C B 10-40	D C B : 1
D A C 50-80	D A C : 1
D A B 10-40 50-70	D A B : 2

Minimal (serial) occurrences

D B C 50-80	D B C : 1
C D A 20-60	C D A : 1
C A D 20-50	C A D : 1
C A B 20-40	C A B : 1
C B D 20-50	C B D : 1
C B A 20-60	C B A : 1
A D B 30-70	A D B : 1
A B D 30-50	A B D : 1
A B C 60-80	A B C : 1
B D C 40-80	B D C : 1
B D A 40-60	B D A : 1
B A C 40-80	B A C : 1
D C A B 10-40	D C A B : 1
D A B C 50-80	D A B C : 1
C A B D 20-50	C A B D : 1
C B D A 20-60	C B D A : 1
B D A C 40-80	B D A C : 1

Figure 3.6 Frequent episodes of the temporal sequence in interval $[0, 120[$ for MINEPI with support = 1.

interval. For instance, we observe that episode ABD, considering serial class of events, is identified 5 times and in different orders i.e., $occurrences_{BAB} = 2$, $occurrences_{ADB} = 1$, $occurrences_{ABD} = 1$ and $occurrences_{BDA} = 1$.

The main difference of these two approaches, is that WINEPI can be interpreted as the probability of encountering an episode over sliding windows of randomly chosen size, while MINEPI counts exact minimal occurrences of episodes over a log of fixed size. Compared to classic apriori-based algorithms, candidate episodes for MINEPI and WINEPI are generated progressively by extending already identified frequent subsequences. As WINEPI operates over sliding windows it is more efficient in the first phases of the episode generation, while MINEPI outperforms in the latter iterations. The limitation of WINEPI is that while sliding windows iterate over a dynamic log, the cost of maintaining frequent episodes and rules can be high if previously deduced episodes are not any longer observed. On the other hand, MINEPI's localisation of minimal occurrences can be high at the first iterations when required data structures are larger than the original sequence. Time complexity of WINEPI [26] is $O((m=w)kj\ j + m)$ for parallel and $O(mkj\ j + m)$ for serial class of events, where w are shifts, k generated episodes, j, j prefixes for each episode and m the size of the log. The complexity of finding whether a serial or parallel episode has an occurrence in a sequence for MINEPI, is NP complete [25].

In this section, we illustrated WINEPI and MINEPI, two state of art sequential mining algorithms. The question that emerges is whether these mining algorithms can effectively discover joins over a query log. Next, we apply the MINEPI algorithm over query logs and identify its limitations.

3.3 MINEPI over query logs

In this section, we apply the MINEPI [4] state-of-art algorithm using as input query logs that are collected from data providers, but similar observations may hold for WINEPI as well. The challenge, is to explore if MINEPI is sufficient to solve our scientific problem as presented on page 39 in Chapter 2, or, we need to furthermore process the query log in order to have representative results in both recall and precision of joins.

First, in Section 3.3.1, we present the experimental testbed of MINEPI. Thereafter, in Section 3.3.2, we apply MINEPI default version over query logs.

3.3.1 Experimental testbed

Experiments in this chapter are evaluated using execution traces of queries of the Cross Domain (CD) collection, which is taken from FedBench [46]. From this benchmark, we used the setup of DBpedia[5], NY Times, LinkedMDB, Jamendo, Geonames and SW Dog

[4] We execute MINEPI using parallel class of events, as subqueries in a federated log are partially ordered.

[5] DBpedia is distributed in 12 data subsets (http://fedbench.fluidops.net/resource/Datasets), in our setup, DBpedia Ontology dataset is duplicated in all SPARQL endpoints, so we install 11 SPARQL endpoints for DBpedia instead of 12.

Food datasets. Each of these datasets is installed into a SPARQL endpoint using Virtuoso OpenLink[6] 6.1.7.

We executed federated queries with FedX 3.0. We implemented a tool to shu e several logs of queries executed in isolation, according to di erent parameters[7]. Thus, given $E(FQ_1); :::; E(FQ_n)$ we were able to produce di erent signi cant representations of $E(FQ_1 \ k \ ::: \ k \ FQ_n)$. Produced traces with this tool vary in (i) the order of federated queries, (ii) the number of subqueries of the same federated query, appearing continuously in the shu ed log (blocks of 1 to 16 subqueries), and (iii) the delay between each subquery (from 1 to 16 units of time). As we aimed to deduce the joins of triple patterns in the original queries, we extracted only episodes of size = 2 with MINEPI.

3.3.2 Experiments with MINEPI

Query/Collection	Alphabet size
CD_1	3
CD_2	3
CD_3	695
CD_4	17
CD_5	12
CD_6	1229
CD_7	371
CD concurrent	2316

Table 3.4 Alphabet sizes of events of CD traces, produced by a federated query engine and executed over a federation of SPARQL endpoints, for MINEPI default version with triple pattern granularity.

Query/Collection	Frequent episodes (length = 2, support = 1)	Frequent episodes (length = 2, support = 2)
CD_1	3	0
CD_2	3	0
CD_3	75	36
CD_4	136	81
CD_5	66	15
CD_6	754606	92570
CD_7	68635	56895
CD concurrent	2033005	404330

Table 3.5 Frequent episodes of CD traces, produced by a federated query engine and executed over a federation of SPARQL endpoints, for MINEPI default version with triple pattern granularity, length = 2 and di erent support thresholds.

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

[7] The program to shu e several execution logs in isolation, used as input either to MINEPI, LIFT or FETA, is available at: https://github.com/coumbaya/traceMixer

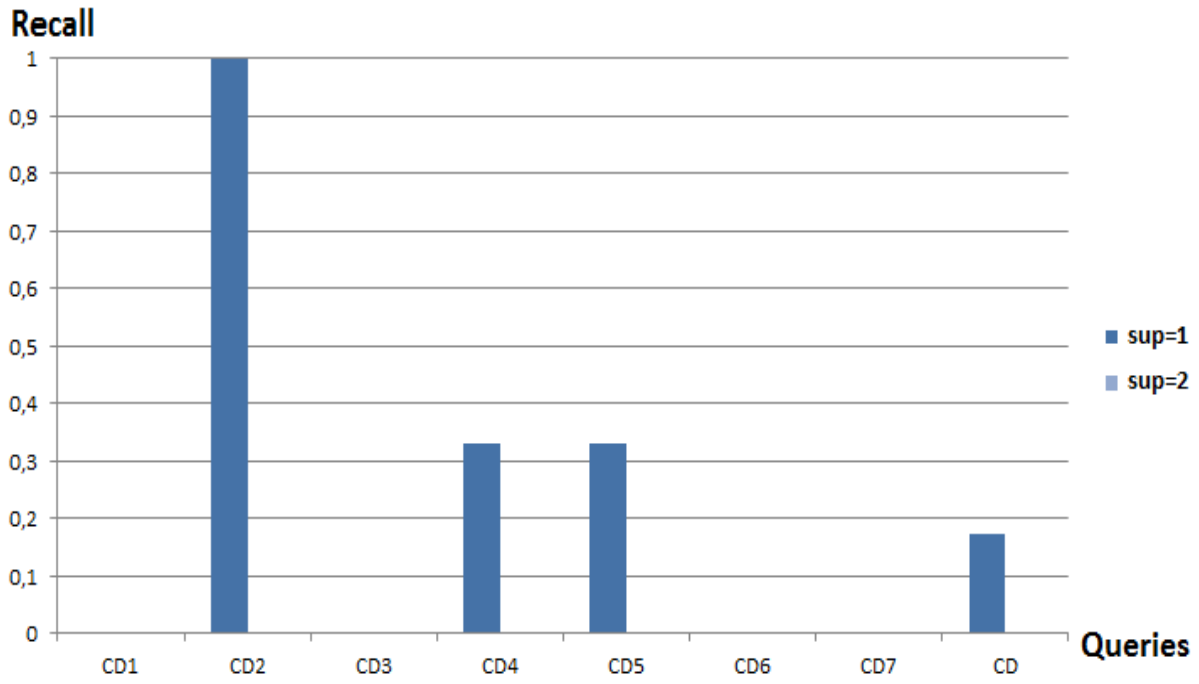


Figure 3.7 Recall of joins of traces of CD queries, produced by a federated query engine and executed over a federation of SPARQL endpoints, for MINEPI default version with triple pattern granularity and di erent support thresholds.

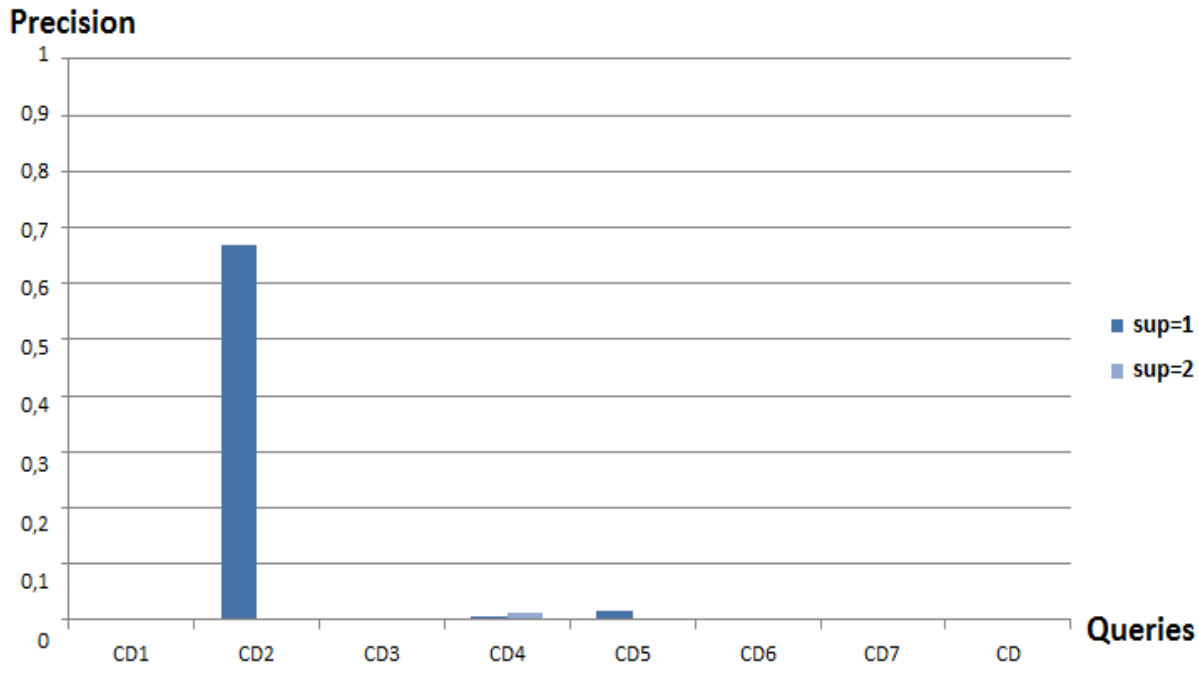


Figure 3.8 Precision of joins of traces of CD queries, produced by a federated query engine and executed over a federation of SPARQL endpoints, for MINEPI default version with triple pattern granularity and di erent support thresholds.

In this section we executed the MINEPI default version, by considering as events the triple patterns. Table 3.4 presents alphabet sizes of events of MINEPI for triple pattern granularity. Table 3.5 presents frequent episodes of length=2 for di erent support threshold values. Figures 3.7 and 3.8, illustrate MINEPI's performance in recall and precision of joins, respectively.

From this experiments, rst, we observe that the alphabet of events may consist of hundreds of triple patterns, even when executing only one query. Even worst, MINEPI performs poorly in recall but also in precision of joins, for both traces of queries executed in isolation or in concurrence. Finally, we observe that for support > 1 along with the number of episodes recall also decreases dramatically or event zeroed.

We conclude with this experiment, that MINEPI is not adequate to reveal the actual joins in original queries for both traces produced in isolation or in concurrence, even when using the most favourable execution condition i.e., for support = 1. Next, we aim to interpret the results obtained in this section and to identify the limitations of sequential pattern mining when applied over query logs, using an abstract example.

3.4 Limitations of query log analysis

In this section we aim to explain why sequential pattern mining algorithms, in their current form, perform poorly in recall and precision of joins when analyzing query logs. From their limitations, we identify which is the necessary processing e ort that needs to be employed, so that these algorithm become more e cient in deducing the actual joins of original queries. In order to explain our proposed perspectives, we use the following example.

Consider again the abstract log of Table 3.1 on page 46, corresponding to execution plans of queries Q_A = SELECT ?x ?y WHERE f ?x p1 o1 : ?x p2 ?y : ?y p3 ?zg, Q_B = SELECT ?y WHERE f ?x p2 ?y g, Q_C = SELECT ?y WHERE f ?y p3 ?z g and Q_D = SELECT ?x WHERE f ?x p1 o1 g. We present next, one by one, the limitations of such algorithms over a log of (sub) queries.

1. The pertinence of the alphabet : The alphabet of events in a query log is proportional to the cardinality of triples residing in the Linked Data[8]. However, the main issue for triple patterns[9] is not their quantity but their pertinence. Depending on optimization techniques employed by query engines, constant values of triple patterns actually may correspond to mappings that replace a join variable of a triple pattern in the original user query. That is, when two triple patterns of a user query are joined through a nested-loop, as we presented in Chapter 2 on page 25, the former triple pattern pushes its mappings into the latter. In other words, the triple patterns we observe in the log may actually be the result of the decomposition of original triple patterns in user queries. Without knowing the exact triple patterns,

[8] We do not take into account triple patterns that do not correspond to actual Linked Data resources i.e., IRI/literals of triple patterns that are contained in queries posed by data consumers. In this case, the alphabet of events is in nite.

[9] We could consider di erent levels of granularity, regarding the accessed resource of our log: predicate, triple pattern or subquery. We choose triple patterns, as we aim to extract BGPs of user queries.

Heuristic	Alphabet
none	f f ?x p1 o1 g; f s1 p2 ?y g; f o3 p3 ?z g; f s2 p2 ?y g; f o4 p3 ?z g; f ?x p2 ?y g; f ?y p3 ?z g; f ?x p1 o1 g g
NestedLoopDetection	f f ?x p1 o1 g; f ?x p2 ?y g; f ?y p3 ?z g g

Table 3.6 Alphabet of events of Q_A - Q_D traces, produced by a federated query engine and executed over a federation of SPARQL endpoints, for MINEPI with triple pattern granularity and with or without the NestedLoopDetection heuristic.

we are not able to identify the joins of these queries. Therefore, we need a reverse function the reveals hidden join variables from triple patterns, by detecting nested-loops between them. We denote such function as NestedLoopDetection, which we de ne next.

De nition 6 (NestedLoopDetection) Suppose a query log of one or more data providers corresponding to execution traces of queries they process, as de ned in De nition 4 on page 39. We de ne NestedLoopDetection as a nested function that detects pushed mappings from a triple pattern in the log into followings, respecting a time-interval threshold between them. Once a nested-loop is detected, pushed mappings in the latter triple patterns are replaced with the original variable of the former to which they are joined.

Therefore without NestedLoopDetection, performance in recall may be considerably low as original patterns in user queries are decomposed in tens or hundreds of triple patterns during nested-loops. In our example, using NestedLoopDetection we realize that entries 2, 4 of Table 3.1 on page 46 are the result of a nested-loop between f ?x p1 o1g and f ?x p2 ?yg, where subject mappings f s1; s2g of the former are pushed in the subject of the latter. Likewise, entries 3, 5 are the result of a nested-loop between f ?x p2 ?yg and f ?y p3 ?zg. Table 3.6 presents the alphabet for triple pattern granularity, with or without NestedLoopDetection

2. The size of the alphabet : By default, sequential pattern mining is applied over logs spread over days or weeks of usage, as it aims to discover episodes that occur multiple times e.g., with support > 1. The diversity of these triple patterns will not depend only on the nested-loops employed by query engines, but also on queries posed by users. So, regarding the previous challenge, the size of the alphabet may be considerably large even if we apply the NestedLoopDetection function. Table 3.1 presents a log of only a few seconds. The size of the alphabet of events for larger logs e.g., one hour, can be unpredictably large.

3. The choice of threshold : In general, Data Mining algorithms are using a threshold either de ned asfrequency for WINEPI or support for MINEPI, to extract frequent episodes. The choice of such value is completely arbitrary. Using a small threshold value, we may get a lot of false positives regarding discovered episodes and their correspondence to joins. On the other hand, a larger threshold would exclude some non frequent sets of triple patterns that correspond to joins. But as in general sequential mining algorithms are applied over logs of a signi cant duration, this threshold is greater that one. For our example, withsupport = 2 we deduce the episode associating f ?x p2 ?yg and f ?y p3 ?zg, but not the complete BGP of Q_A i.e., f f ?x p1 o1g; f ?x p2 ?yg; f ?y p3 ?zg g as it occurs only once.

4. The di erence between apparition and join ordering : The order in which events are captured in the log does not necessary correspond to joins. Frequent episodes in MINEPI or WINEPI, group events that occur together. Nevertheless, joins are not always made over consecutively appearing triple patterns, even for the execution of a single query. For instance, joins in query Q_E, composed by triple patterns tp_1 = f ?x p1 o1g, tp_2 = f ?w p2 ?zg and tp_3 = f ?x p3 ?zg, are between f tp_1, tp_3g and f tp_2, tp_3g, even if a sequential mining algorithm will also identify f tp_1, tp_2g. This challenge a ects mostly performance in precision but in presence of concurrence it may a ect also performance in recall, as we see next.

5. The concurrent execution of queries : In the context of concurrent execution of queries, either decomposed in multiple subqueries or posed directly over data providers, both precision and recall may be a ected. We present below in which cases these situations emerge.

   First, occurrences of events in a query log is related to the selectivity of operations. Suppose an additional query Q_F = SELECT ?x WHERE f ?x p3 ?z : ?y p4 ?z g. A data consumer may decide to execute the join with a nested-loop. So f ?x p3 ?zg will appear once in the log, while triple patterns with f ?y p4 IRIs g will appear many times according to the selectivity of f ?x p3 ?zg. Searching for frequent episodes will raise up episodes with triple patterns containing false positives of joins, for instance p2 and p4 but actual joins were between f p1; p2g, f p2; p3g and f p3; p4g. Second, due to the di erence between apparition of events and the join ordering decided by the data consumer to evaluate them, sequential mining algorithms may combine triple patterns contained in di erent queries instead those contained in the same. For instance, the episode composed by tp_1 = f ?x p1 o1g and tp_3 = f ?x p3 ?zg, may be created from tp_1 of the rst pattern of Q_E and the rst of Q_F. In this case, the actual join between tp_1 and tp_3 of Q_E will never be detected.

In this section, we illustrated the limitations of query log analysis and realized that in order to have a pragmatic view of actual joins we need to apply a reverse function in order to reveal the actual events i.e., triple patterns that we aim to track. This is made through the NestedLoopDetection heuristic, either using a phase of pre-processing data transformation or by applying post-processing constraints, that we explain next.

3.5 MINEPI with pre or post-processing

In this section, we aim to enchance the MINEPI [10] state-of-art algorithm, by processing the input logs enough in order to have a more pragmatic view of actual joins of original queries. The challenge is to explore if MINEPI enhanced with a processing effort is sufficient to solve our scientific problem, as presented on page 39 in Chapter 2, or, its performance is still moderate in terms of either recall or precision of joins.

In the experiments of the next sections, we adopt the experimental tesbed of Section 3.3.1. In order to reveal actual joins of triple patterns, we aim to apply the NestedLoop Detection using two different strategies. First, in Section 3.5.1, we apply MINEPI with a pre-processing NestedLoopDetection, which is a applied as a data transformation phase over the whole query log. Finally, in Section 3.5.2, we present MINEPI with a post-processing NestedLoopDetection, which is applied as a pruning phase only to the minimal occurrences of frequent episodes identified by MINEPI.

3.5.1 MINEPI with data transformation

As in every Data Mining process, a pre-processing phase can be applied before data analysis in order to transform the raw input into a homogeneous schema. In particular, we apply the NestedLoopDetection heuristic over the whole query log. Our aim is to eventually reveal from which variable's mappings, the constants of each triple pattern were pushed during nested-loops.

Query/Collection	Frequent episodes (length = 2, support = 1)	Frequent episodes (length = 2, support = 2)
CD_1	2	0
CD_2	2	0
CD_3	10	6
CD_4	6	2
CD_5	6	3
CD_6	3	2
CD_7	3	2
CD concurrent	230	87

Table 3.7 Frequent episodes of CD traces, produced by a federated query engine and executed over a federation of SPARQL endpoints, for MINEPI with triple pattern granularity, NesteLoopDetection as data transformation and length = 2.

Table 3.8 presents the sizes of the alphabet of events, when applying NestedLoop Detection as data transformation before applying MINEPI, for execution logs of CD queries. Table 3.7 presents frequent episodes of length=2 for different support threshold values. Figures 3.9 and 3.10, illustrate MINEPI's performance when enhanced with a pre-processing phase of NestedLoopDetection, regarding both recall and precision of joins respectively.

[10] We execute MINEPI using parallel class of events, as subqueries in a federated log are partially ordered.

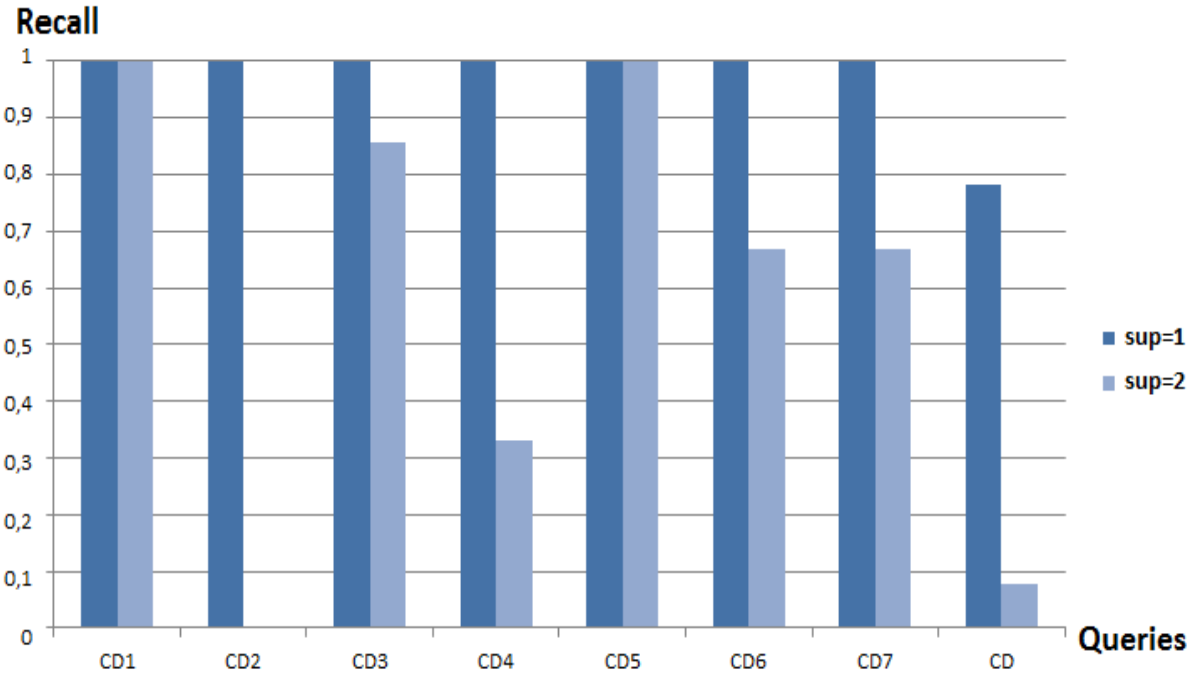


Figure 3.9 Recall of joins of traces of CD queries, produced by FedX query engine and executed over a federation of SPARQL endpoints, for MINEPI with NestedLoopDetection as data transformation and different support thresholds.

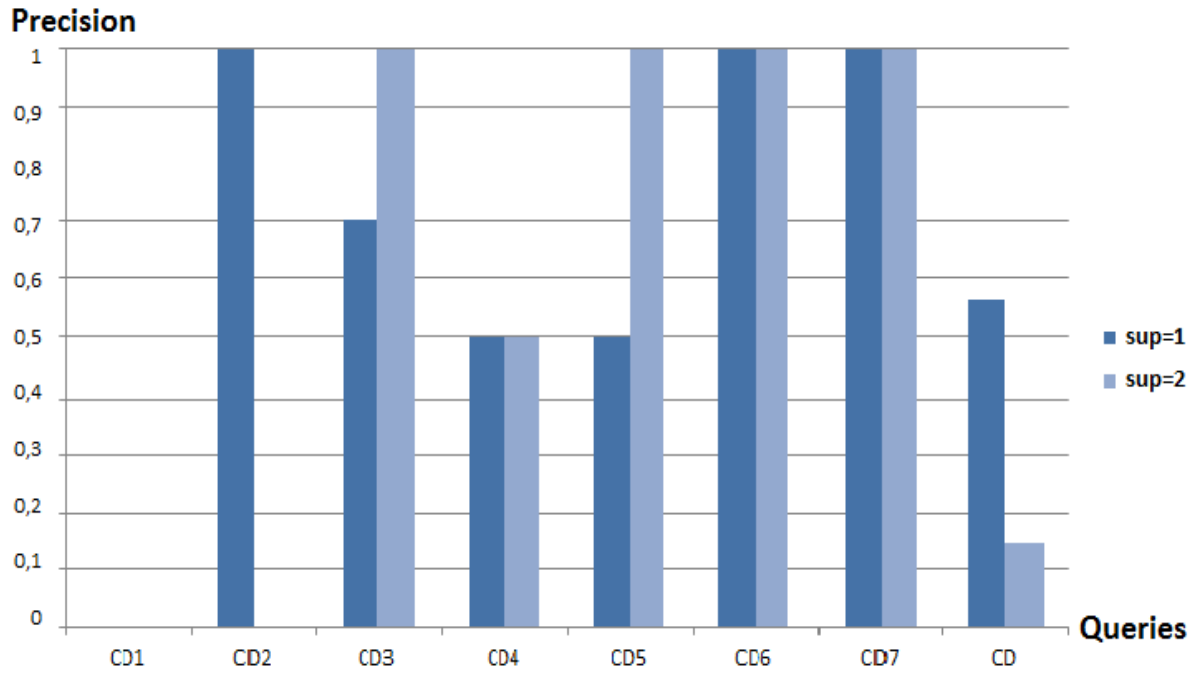


Figure 3.10 Precision of joins of traces of CD queries, produced by FedX query engine and executed over a federation of SPARQL endpoints, for MINEPI with NestedLoopDetection as data transformation and different support thresholds.

Query/Collection	Alphabet size
CD_1	3
CD_2	3
CD_3	5
CD_4	4
CD_5	5
CD_6	4
CD_7	4
CD concurrent	28

Table 3.8 Alphabet sizes of events of CD execution traces, produced by a federated query engine and executed over a federation of SPARQL endpoints, for MINEPI triple pattern granularity and NestedLoopDetection as data transformation.

First, we observe that the alphabet is reduced significantly, some times from thousands into just tens of triple patterns. Even better, we are able to have a pragmatic view of real joins of the log (cf. Limitation 1). In addition, similarly to the default MINEPI version, both the number of episodes and recall significantly decrease for $support > 1$ (cf. Limitations 2, 3). Finally, we observe that precision compared to the default version of MINEPI is low, as frequent episodes may concern false joins from triple patterns of different queries (cf. Limitation 5).

Next, we aim to enhance MINEPI with NestedLoopDetection but this time in form of a post-processing constraint, thus minimizing the intervention to those events that are identified as minimal occurrences by MINEPI.

3.5.2 MINEPI with pruning constraints

As presented in Section 3.2, all apriori-like algorithms have the problem of producing an exponential number of generated episodes. So in order to minimize the volume of episodes, we can apply constraints after the episode generation of MINEPI. In particular, we apply the NestedLoopDetection heuristic as a constraint, only to those entries identified by the minimal occurrences of MINEPI and not to the whole input log.

Table 3.9 presents frequent episodes of length=2 for different support threshold values and predicate granularity. Figures 3.11 and 3.12, illustrate MINEPI's performance when enhanced with a post-processing phase of NestedLoopDetection, in recall and precision of joins respectively.

As expected, similarly to the default version of MINEPI and MINEPI with preprocessing transformation, both the number of deduced episodes and recall significantly decrease with support > 1 (cf. Limitations 2, 3). Furthermore, even if we have a more pragmatic view of joins comparing to the default version of MINEPI, recall is not as good as MINEPI's with pre-processing data transformation. This is explained from the fact that MINEPI with post processing only identifies the minimal occurrences of episodes, which in presence of concurrence combine triple patterns from concurrently executed queries and not those from the same (cf. Limitations 4,5).

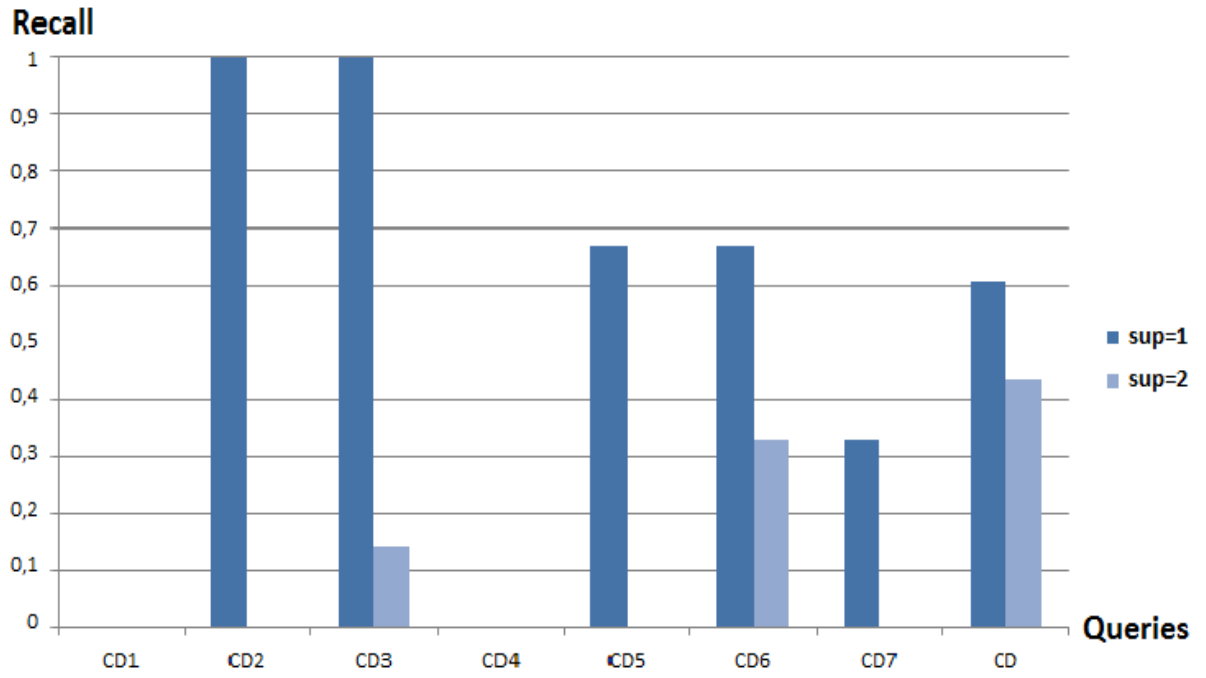


Figure 3.11 Recall of joins of traces of CD queries, produced by FedX query engine and executed over a federation of SPARQL endpoints, for MINEPI with NestedLoopDetection as pruning constraint and different support thresholds.

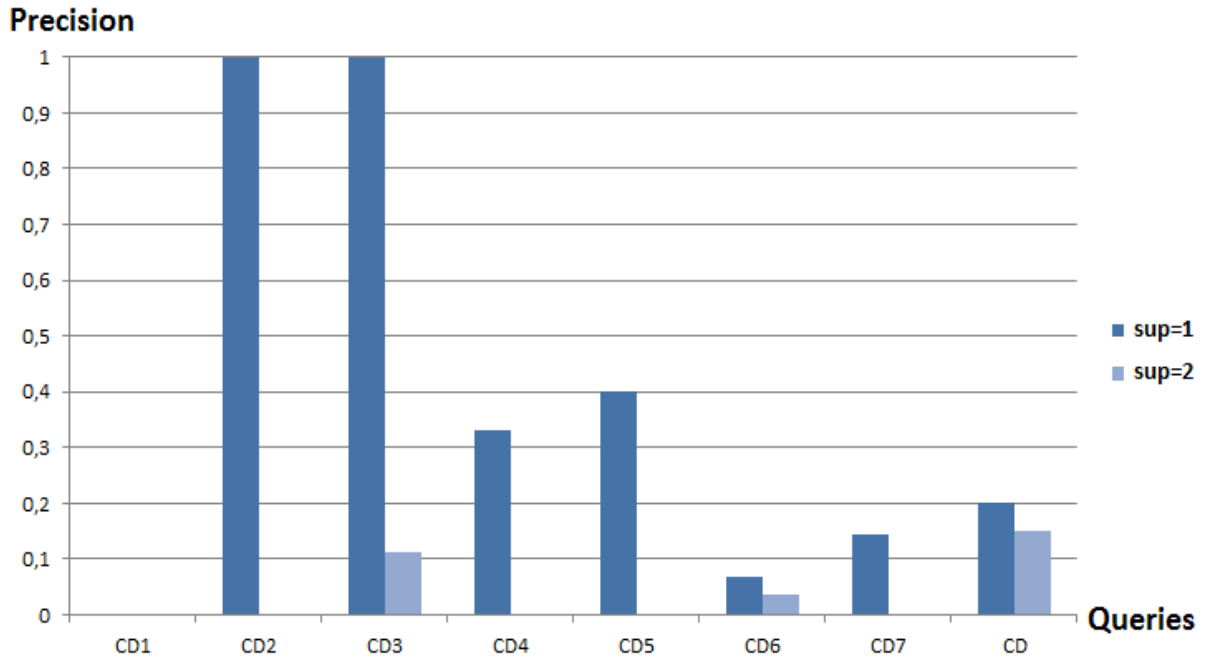


Figure 3.12 Precision of joins of traces of CD queries, produced by FedX query engine and executed over a federation of SPARQL endpoints, for MINEPI with NestedLoopDetection as pruning constraint and different support thresholds.

Query/Collection	Frequent episodes (length = 2, support = 1)	Frequent episodes (length = 2, support = 2)
CD_1	1	0
CD_2	2	0
CD_3	7	2
CD_4	4	1
CD_5	3	1
CD_6	3	2
CD_7	3	2
CD concurrent	17	9

Table 3.9 Frequent episodes of CD traces, produced by a federated query engine and executed over a federation of SPARQL endpoints, for MINEPI with triple pattern granularity, NesteLoopDetection as pruning constraint and length = 2.

In summary, it is necessary to enhance MINEPI with reverse heuristics, either as a pre-processing or post-processing phase, in order to have a more pragmatic view of joins in original queries. But even so, recall depends on the support threshold and is also affected by concurrency. Anyhow, precision can perform poorly as deduced episodes of triple patterns do not correspond always to joins.

Therefore, we need to process logs of subqueries by linking directly triple patterns based only on their mappings without relying on occurrences of their sets. In the next chapters, we present our proposed BGP reversing approaches that aims to solve our scientific problem, presented on page 39 in Chapter 2.

4

LIFT : LInked data Fragment Tracking

Contents

In this chapter we presentLIFT, our proposed approach that aims to answer the question: "Can TPF servers track and approximate BGPs they process from their logs?" This question is addressed both over single and federations of TPF servers, as TPF clients decompose SPARQL queries even when only one server is concerned. The challenge to infer queries that are evaluated with this approach, over single or federations of servers, is to link maybe hundreds of single triple pattern subqueries per query execution. Such an endeavour must be resistant in presence of concurrent execution of other queries.

This chapter rst illustrates the scienti c problem we aim to solve, as described on page 39 in Chapter 2, Section 2.5 over the context of TPF query processing, in Section 4.1. Thereafter, the BGP reversing approach ofLIFT is presented in Section 4.2. Finally, experiments are reported in Section 4.3.

4.1 Illustration example

In Figure 4.1, two clients, c_1 and c_2, execute concurrently queries Q_1 and Q_2 over the TPF server of DBpedia. Q_1 asks for movies starring Brad Pitt and Q_2 for movies starring Natalie Portman. Q_1 and Q_2 are taken from the TPF web application[1].

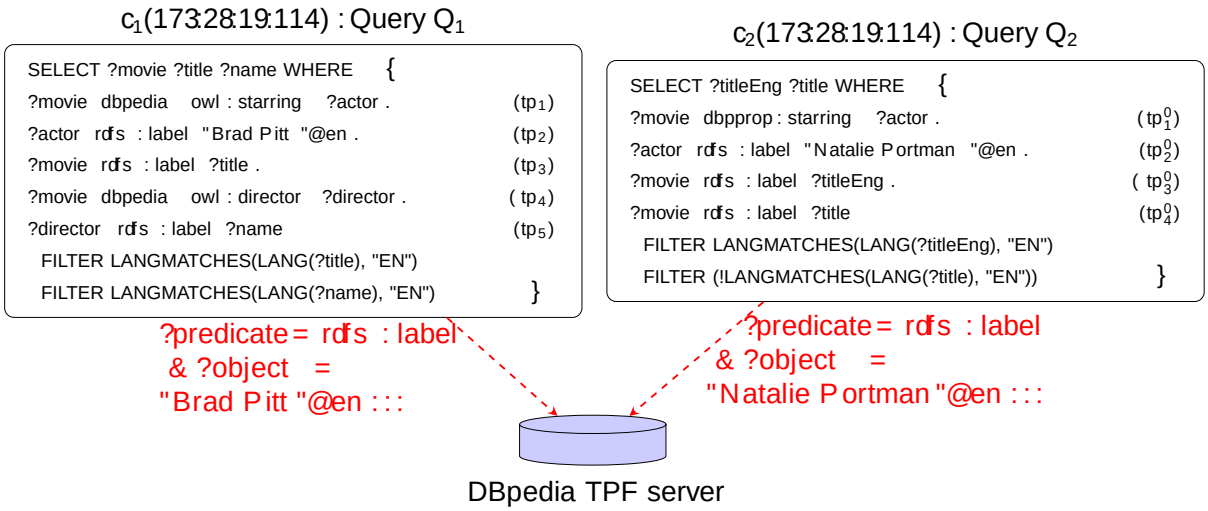


Figure 4.1 Concurrent execution of queries Q_1 and Q_2, produced by TPF client with 173.28.19.114 IP Address and executed on the DBpedia TPF server.

TPF clients decompose SPARQL queries into a sequence of triple pattern queries, as partially presented in Table 4.1 for query Q_1. Lines in grey color correspond to answers of the single triple pattern queries in enumerated lines. As the TPF server only see triple pattern queries, the original queries e.g. Q_1 and Q_2 remain unknown to the data provider.

In this chapter, we address the following research question: "Can TPF servers track and approximate BGPs they process, from their logs?" This question is addressed both over single or federations of TPF servers, as TPF clients decompose SPARQL queries even when only one TPF server is concerned. In particular, we aim to solve the scientific problem of BGPs reversing (cf. Definition 5 on 40), of query evaluation over TPF servers. We also consider the definition, notation and property of query log (cf. Definition 4 on 39), execution trace and resistance to concurrency (cf. Property 1 on 41) respectively.

In our example, the DBpedia TPF server log corresponds to $E(Q_1 \,k\, Q_2)$. We aim to extract two BGPs from this, one corresponding to Q_1, $BGP[1] = \{tp_1 : tp_2 : tp_3 : tp_4 : tp_5\}$ and another corresponding to Q_2, $BGP[2] = \{tp'_1 : tp'_2 : tp'_3 : tp'_4\}$.

In Figure 4.1, if c_1 and c_2 have different IP addresses it is possible to separate $E(Q_1 \,k\, Q_2)$ into $E(Q_1)$ and $E(Q_2)$, 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 $f(E(Q_1 \,k\, Q_2)) \neq f(E(Q_1)) \cup f(E(Q_2))$.

[1] http://client.linkeddatafragments.org/ .

	LD provider	IP	Time	Triple pattern suqbuery/TPF
[1]	DBpedia	173...	11:24:19	?predicate=rdfs:label & object="Brad Pitt"@en
				{ < ... controls ... > , < { dbpedia:Brad_Pitt rdfs:label "Brad Pitt"@en} >, < ... metadata ... > }
[2]	DBpedia	173...	11:24:24	?predicate=dbpedia-owl:starring & object= dbpedia:Brad_Pitt
				{ < ... controls ... > , < { dbpedia:A_River_Runs_Through_It_(lm) dbpedia-owl:starring dbpedia:Brad_Pitt}, ... , {dbpedia:Troy_(lm) dbpedia-owl:starring dbpedia:Brad_Pitt} >, < ... metadata ... > }
[3]	DBpedia	173...	11:24:28	?subject=dbpedia:A_River_Runs_Through_It_(lm) & predicate=rdfs:label
				{ < ... controls ... > , < {dbpedia:A_River_Runs_Through_It_(lm) rdfs:label "A River Runs Through It (lm)"@en}, ... , {dbpedia:A_River_Runs_Through_It_(lm) rdfs:label "Et au milieu coule une rivière"@fr} >, < ... metadata ... > }
[4]	DBpedia	173...	11:24:31	?subject=dbpedia:A_River_Runs_Through_It_(lm) & predicate=dbpedia-owl:director
				{ < ... controls ... > , < {dbpedia:A_River_Runs_Through_It_(lm) dbpedia-owl:director dbpedia:Robert_Redford } >, < ... metadata ... > }
[5]	DBpedia	173...	11:24:34	?subject=dbpedia:Robert_Redford & predicate=rdfs:label
				{ < ... controls ... > , < {dbpedia:Robert_Redford rdfs:label "Robert Redford"@en}, ... , {dbpedia:Robert_Redford rdfs:label "Robert Redford"@fr} >, < ... metadata ... > }

Table 4.1 Partial log of Q_1 traces, produced by TPF client with 173.28.19.114 IP Address and executed on DBpedia TPF server. Answers are extracted from data providers in form of Triple Pattern Fragment.

Next, we present our proposed BGP reversing solution Linked data Fragment Tracking (LIFT), which we evaluate with traces of queries from the TPF web application interface executed (i) in isolation and (ii) in concurrence, over single or federations of TPF servers. In addition, we report that LIFT extracts useful BGPs with traces of the real log of USEWOD 2016 [28].

4.2 LIFT: a reversing approach

LIFT is a system of algorithms based on heuristics, to implement the reverse function Γ^{-1}. The idea is to detect nested-loop joins. In Table 4.1, the mappings returned in Line 2 are reused in the next triple pattern query in Line 3. We track such bindings in order to link different triple pattern queries.

In this chapter, we make the following hypothesis:

1. We consider only bound predicates[2],
2. We consider that TPF servers do not use a web cache (this information can be easily obtained by data providers), and
3. We consider that clients do not use a cache (concerning both selectivity of triple patterns and their mappings).

Figure 4.2 presents a simplified log of Q_3 = SELECT * WHERE { ?x p2 toto . ?x p1 ?y }, Q_4 = SELECT * WHERE { ?x p3 titi . ?x p1 ?y . ?x p4 tata } and $E(Q_3 \,\|\, Q_4)$.

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. Unknown variables are named ?s or ?o. μ_o represents the mappings of variables resulting from the evaluation of tp on data. We call them output-mappings.

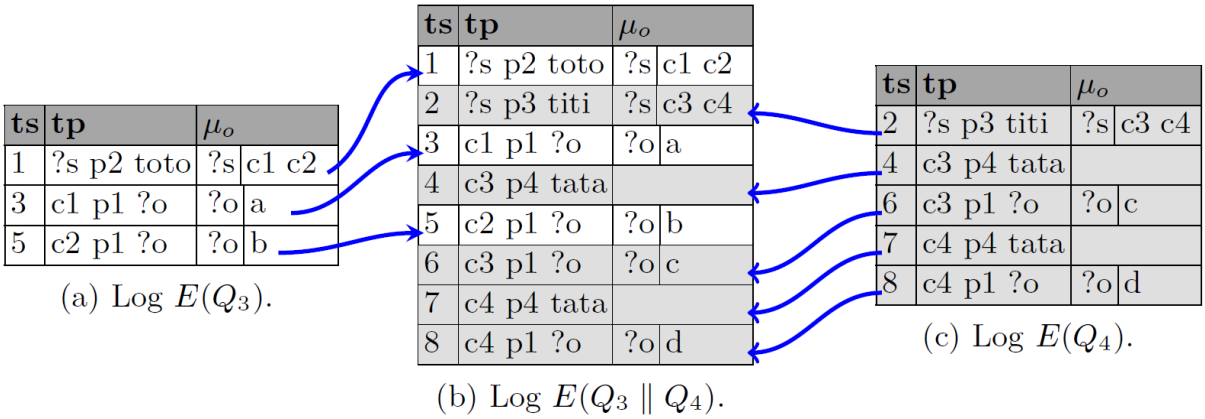


Figure 4.2 Examples of simplified TPF logs, for Q_3 and Q_4 traces.

Algorithm 1 shows the three phases of LIFT:

1. First, LIFT merges triple patterns into candidate triple patterns. It allows to gather triple patterns that can be part of the same outer or inner operand of a join. We denote the set of candidate triple patterns as CTP.

[2] As reported in [8], predicates of triple patterns are frequently bound. Nevertheless, FETA like LIFT can be extended to deal with predicates just like they deal with subjects and objects.

2. Next, LIFT looks for an inclusion relationship among mappings of candidate tps. If it does not exist, LIFT splits candidate triple patterns to build it. This produces a set of graphs, which we denote as DTP, where nodes are deduced triple patterns and edges represent inclusion relationships between these triple patterns. This detects nested-loops.

3. Finally, LIFT extracts BGPs from the DTP Graph set. Ideally, LIFT(E(Q_3 k Q_4); gap) should compute the 2 BPGs of Q_3 and Q_4: f ?s p2 toto : ?s p1 ?og and f ?s p3 titi : ?s p1 ?o : ?s p4 tatag.

The basic intuition of LIFT is to detect if mappings are bound in next requests. This can be challenging, as mappings can be: (i) bound several times (e.g., in star queries), (ii) bound partially as a side-effect of LIMIT and FILTER clauses, or (iii) bound into a different concurrent query. As a real log can be huge, LIFT analyzes the log using a constraint as a sliding window which is defined by a gap i.e., a time interval. When LIFT reads an entrye in the log with a timestamp ts, it considers only entries reachable within the gap i.e., ts gap.

Algorithm 1: Global algorithm of LIFT

```
1 Function LIFT (log; gap) is
    input  : a log; a gap in time units (seconds)
    output : a set of BGPs
    data   : CTP a set of candidate tps, DTP a set of graphs of deduced tps
2   CTP ctpExtraction (log; gap)
3   DTP nestedLoopDetection (CTP; gap)
4   return BGP bgpExtraction (DTP)
```

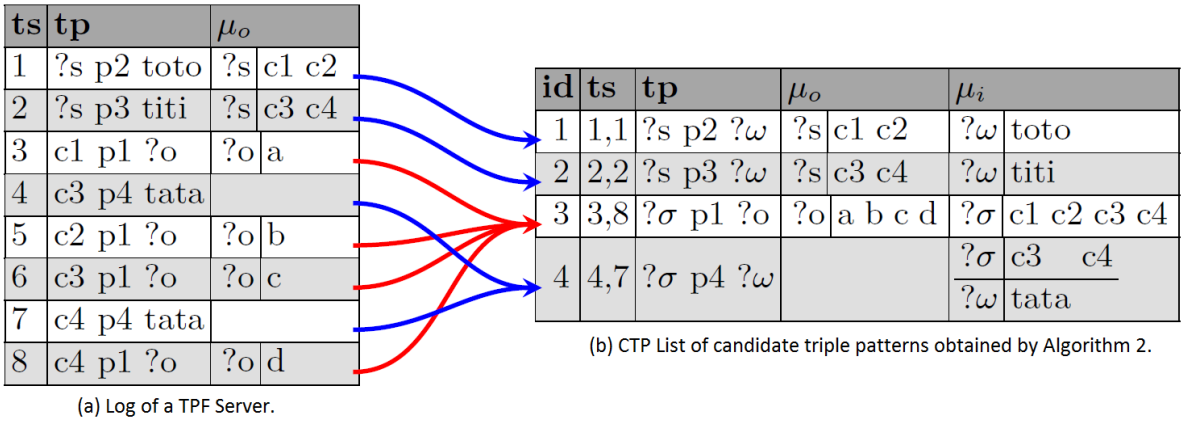
Section 4.2.1 details the CTP extraction. Section 4.2.2 describes the nested-loop detection. Finally, Section 4.2.3 presents the final phase of extraction of BGPs.

4.2.1 Extraction of candidate triple patterns

ctpExtraction aims to aggregate together log entries that seem to participate in the same outer or inner operand of a join. Aggregated entries are represented by candidate triple patterns. All candidate triple patterns form the CTP set.

A c2 CTP is a tuple[3] hip; ts; tp; $_o$; $_i$i where ip is an IP address, ts is a pair of timestamps (ts:min; ts:max) representing a range; when creating a candidate triple pattern, both timestamps are identical and correspond to the timestamp of the current entry in the log. tp is a triple pattern query, $_o$ (output-mappings) is the list of solution mappings for variables of tp. $_i$ (input-mappings) is a set of mappings built during the ctpExtraction. Basically, we replace any constant of tp by a variable, we use for subject and ! for object. Replaced constants are regrouped in .

[3] Note, that in the case of a federated log, the candidate tp tuple is enhanced with an additional field, which we denote as f tsr g, with the set of TPF servers that evaluate it.



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 ?ω	c3 c4 tata

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

Figure 4.3 TPF log and CTP List, produced by Algorithm 2 with $E(Q_3 \text{ k } Q_4)$ and for gap = 8.

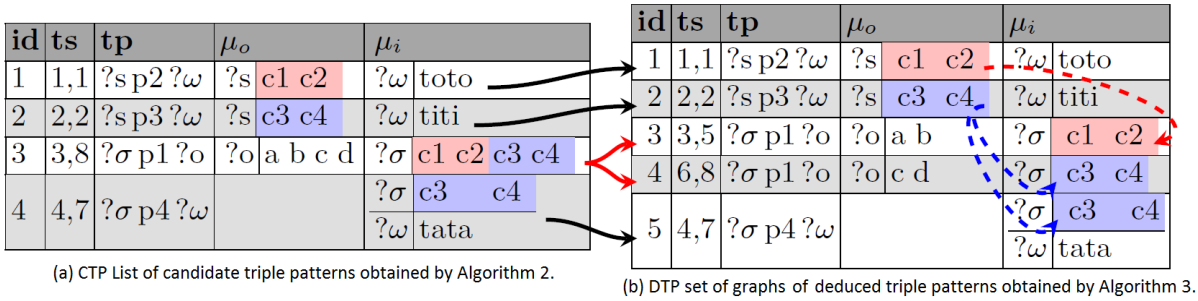
Algorithm 2: Extraction of Candidate Triple Patterns

```
1  Function ctpExtraction (log; gap) is
       input  : a TPF log; gap an interval in time units (seconds)
       output : CTP a list of candidate tps
2      CTP  [ ]
3      foreach e 2 log do
4          c  read(e) as (ip, (ts,ts), tp, o) switch ctp do
5              case ?s p o: ctp  ?s p ?o_in ; c_i  ?! |o
6              case s p ?o: ctp  ?s_in p ?o ; c_i  ? |s
7              case s p o: ctp  ?s_in p ?o_in ; c_i  ? |s, ?! |o
8              case ?s p ?o: ctp  ?s p ?o ; c_i  ;
9          if 9 c_k 2 CTP | ingap(c, c_k, gap) ^ (c_k:ip = cip) ^ (ctp = c_k:tp) then
10             (c_k. o [ c o); (c_k. i [ c i); (c_k:ts:max = cts:max);
11         else CTP.add(c)
12     return CTP
```

Algorithm 2 outlines the extraction of a CTP List from a TPF log for a particular gap. Figure 4.3 illustrates the effect of this algorithm on log $E(Q_3 \text{ k } Q_4)$ for gap=8.

The log is processed in sequential order. Lines 5 to 8 initialize input-mappings by replacing constants by variables or ! . Next, lines 9 to 10 merge the current candidate triple pattern with an existing and compatible, if there exist one. An existing candidate tp is compatible if it has the same tp, it is produced by the same ip address and ts in the gap. The ingap(c, c_k; gap) function returns true if cts:min c_k:ts:max gap. If the current candidate tp is compatible with an existing one, output/input-mappings and timestamps are merged. Otherwise, we create a new entry in line 11. When updating timestamps, the lower timestamp remains always the same and only the upper timestamp can grow up. A variable of tp can not belong to o and i simultaneously.

This algorithm can aggregate triple patterns that do not belong to the same nested-loop as it is the case in our example, 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



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 set of graphs of deduced triple patterns obtained by Algorithm 3.

Figure 4.4 CTP List and DTP Graph set, produced by Algorithm 3 for gap = 8.

the case, next algorithm splits candidate tps to separate these nested-loops.

4.2.2 Nested-loop join detection

Algorithm 3 describes how to link variables of different candidate tps, produced by Algorithm 2, and builds a set of graphs of deduced triples patterns, which we denote as DTP, by linking different candidate tps if there a relation of inclusion between them. Figure 4.4 presents the DTP Graph set produced by Algorithm 3, using the CTP List of Algorithm 2. Dashed links represent linked variables deduced by Algorithm 3.

If the $_i$ of a candidate tp is a subset of the $_o$ of a previous candidate tp, then we consider that the 2 corresponding variables can be linked. This happens in the example described in Figure 4.4, with CTP[2] and CTP[4]. We consider that ?σ of CTP[4] is linked to ?s of CTP[2]. We formalize this behaviour at lines 6 to 7 of Algorithm 3.

Algorithm 3: Detection of nested-loop joins

```
1  Function nestedLoopDetection (gap, CTP) is
       input  : gap an interval in time units (seconds), CTP a set of candidate tps
       output : DTP a set of graphs of deduced tps
2      foreach c2 ∈ CTP do
3          if split(c2) = ∅ then CTP.insertAfter(cid, split(c2));
4          else DTP.addnode(c2); foreach v_o ∈ vars(c_o) do
5              foreach (c_k, v_i) ∈ {(c_k, v_i) | c_k ∈ CTP ∧ (c_k.id > cid) ∧ ingap(c_k, c2, gap) ∧
               ∃ v_i ∈ vars(c_k : _i) | (c_k : _i(v_i) \ c_o(v_o) ≠ ∅)} do
6                  if c_k : _i(v_i) ≠ c_o(v_o) then
7                      DTP.addnode(c_k); DTP.addEdge(c2, c_k, (v_o,v_i));
8                  else DTP.addnode(s=split(c_k, v_i, c2, v_o)); DTP.addEdge(c2, 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]. Indeed, Q_3 and Q_4 have a common triple pattern (?x, p1, ?y) and Algorithm 2 aggregates them. We solve this problem by splitting a candidate tp. The idea is to produce a deduced tp from a candidate tp, if it exists an intersection between the $_o$ of another candidate triple pattern and the $_i$ of this one. In the example

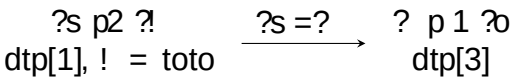
described in Figure 4.4, CTP[3] is split two times: one when analyzing CTP[1] (DTP[3] is produced) and another when analyzing CTP[2] (DTP[4] is produced) because both $_0$ intersect the $_i$ of CTP[3]. Splitting does not a ect only the input-mappings, it also impacts timestamps and output-mappings. After 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. The function split is straightforward, as it basically remerges 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 candidate tp with its produced deduced tps. In our example, for CTP[3] we have 2 split relations; split (CTP[3]; CTP[3^0]) and split (CTP[3]; CTP[4^0]).

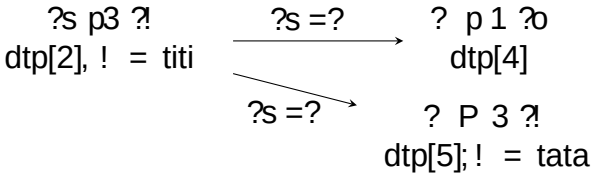
Splitting has an e ect on CTP traversal that we see in Line 3. Output-mappings of produced deduced tps must be analyzed, so when the nested-loop detection analyzes a split candidate tp it inserts in the CTP List the deduced tps that are produced with this split. split (c) returns the set of deduced tps produced by splitting this candidate tp.

4.2.3 BGP extraction

Figure 4.5 represents the connected components of the DTP Graph set shown in Figure 4.4. From this representation, it is easy to compute the nal BGPs with a variable renaming and restitution of an IRI/literal in place of ! when there is only one input mapping e.g., toto; titi and tata.



(a) {?s1 p2 toto . ?s1 p1 ?o3}



(b) {?s2 p3 titi . ?s2 p1 ?o4 . ?s2 p4 tata}

Figure 4.5 Connected components of the DTP Graph set, produced by Algorithm 3 for gap = 8.

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 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 4.3, we measure experimentally the precision and recall of LIFT in di erent situations.

4.2.4 Time complexity of LIFT

The computational complexity of LIFT is O(N M + M^2). N is the number of entries in the TPF log and M is the size of the CTP List. The cost of ctpExtraction is O(N M), because it extracts the candidate tp from each entry of the TPF log and merges it with an existing similar candidate tp of CTP, or, creates a new one. The cost of nestedLoopDetection is O(M M) that gives O(M^2), as each candidate tp of CTP is compared with each other. The cost of bgpExtraction is M.

The overload produced by LIFT is high, but we underline that the size of the log corresponds to a slicing window of time e.g., a separate log for each hour of the day, and that the log analysis can be made as a batch processing.

4.3 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. In Section 4.3.1 we present the experimental testbed of LIFT. In Section 4.3.2, we evaluate precision and recall of LIFT, with traces of queries in the TPF web application executed in isolation. In Section 4.3.3, we evaluate precision and recall of LIFT, with traces of queries in the TPF web application executed concurrently under the worst case scenario, that is when they come from the same IP address. In Section 4.3.4, we analyze LIFT with the TPF log of USEWOD 2016 [28].

Query	Selectors	Query	Selectors	Query	Selectors
Q_1	114	Q_{11}	85	Q_{21}	1223
Q_2	1133	Q_{12}	29	Q_{22}	103
Q_3	27	Q_{13}	100	Q_{23}	1588
Q_4	113	Q_{14}	274	Q_{24}	217
Q_5	296	Q_{15}	54	Q_{25}	881
Q_6	114	Q_{16}	106	Q_{26}	76
Q_7	103	Q_{17}	6	Q_{27}	193
Q_8	207	Q_{18}	20	Q_{28}	excluded
Q_9	7	Q_{19}	44	Q_{29}	4
Q_{10}	119	Q_{20}	3615	Q_{30}	18981
Total	2233	Total	4330	Total	23266

Table 4.2 Number of requests of single triple patterns for queries in the TPF web application, produced by a TPF client and executed in isolation on single TPF servers (DBpedia, Ughent, VIAF or LOV).

4.3.1 Experimental tesbed of LIFT

We extracted 30 queries from the TPF web application concerning DBpedia 2015-04, UGhent, LOV and VIAF datasets. We captured http requests and answers of queries using

the webInspector 1.2tool[4]. We implemented a tool to shu e several TPF logs of queries executed in isolation, according to di erent parameters[5]. Thus, given $E(Q_1); :::; E(Q_n)$ we were able to produce di erent signi cant representations of $E(Q_1 \text{ k } ::: \text{ k } Q_n)$. Produced traces with this tool vary in: (i) the order of queries, (ii) the number of subqueries of the same query, appearing continuously in the shu ed log (blocks of 1 to 16 subqueries), and (iii) the delay between each subquery (from 1 to 16 units of time).

Furthermore, we analyzed the log of the DBpedia TPF server available in the USEWOD 2016 dataset [28]. This log contains http requests from October 2014 to November 2015. We analyzed the rst quarter of the log representing 4,720,874 single triple pattern queries (until 27th February 2015). We cleaned 1% of the log with entries that do not correspond to TPF requests. We considered that all queries were sent by the same TPF client. To obtain corresponding answers, we re-executed the log directly over the DBpedia TPF server[6]. Source code ofLIFT is available at https://github.com/coumbaya/lift .

Query	Runtime	Query	Runtime	Query	Runtime
Q_1	< 1	Q_{11}	< 1	Q_{21}	2
Q_2	2	Q_{12}	< 1	Q_{22}	< 1
Q_3	< 1	Q_{13}	< 1	Q_{23}	< 1
Q_4	< 1	Q_{14}	< 1	Q_{24}	< 1
Q_5	< 1	Q_{15}	< 1	Q_{25}	< 1
Q_6	< 1	Q_{16}	< 1	Q_{26}	< 1
Q_7	< 1	Q_{17}	11	Q_{27}	< 1
Q_8	< 1	Q_{18}	2	Q_{28}	excluded
Q_9	< 1	Q_{19}	< 1	Q_{29}	< 1
Q_{10}	< 1	Q_{20}	10	Q_{30}	220
Average	0.2	Average	2.3	Average	22

Table 4.3 Runtimes (seconds) ofLIFT with traces of queries in the TPF web application, produced by a TPF client and executedn isolation on single TPF servers (DBpedia, Ughent, VIAF or LOV).

Table 4.2 presents the number of requests produced for each query executed in isolation. Table 4.3 presents the runtimes ofLIFT for each execution trace produced in isolation[7]. As we observe, the TPF client produces hundreds of requests for most queries. The execution of Q_{30} produces the largest amount of requests of single triple patterns i.e., 18980, which is the most time consuming forLIFT to analyse. Q_{30} is composed by tp_1 = f ?s a ?typeg and tp_2 = f ?type rdf : label ?labelg. The TPF client by default rewrites tp_1 into f ?s rdf : type ?typeg, which matches 94,190,063 tp_2 matches 20,755,041 triples. The TPF client evaluates the query incrementally by fetching mappings from one and pushing them into the other triple pattern, page by page, thus producing this large amount of requests.

[4] https://sourceforge.net/p/webinspector/wiki/Home/

[5] The program to shu e several execution logs in isolation, used as input either to MINEPI, LIFT or FETA, is available at: https://github.com/coumbaya/traceMixer

[6] http://fragments.dbpedia.org/

[7] We run our experiments in Linux 64 bit machine, with 32 CPUs and 800 Mhz CPU speed.

4.3.2 LIFT deductions of queries in isolation

For each query Q_i, we ran LIFT ($E(Q_i)$, 1). Figure 4.6 presents precision and recall of LIFT deductions in terms of joins, against original queries of the TPF web application[8]. These results show to which extent $LIFT(E(Q_i))$ $BGP(Q_i)$ (cf. De nition 5 on page 40). In average, LIFT obtained 97% of recall and 75% of precision of joins. LIFT deduces perfectly 15/30 BGPs: Q_1 Q_6, Q_9, Q_{11}, Q_{15} Q_{18}, Q_{22}, and Q_{29} Q_{30}.

Concerning Q_9 and Q_{29}, LIFT does not detect UNION queries. Q_9 is a query in the form {(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 joins, thus LIFT detects two separate triple patterns. We consider this behaviour correct.

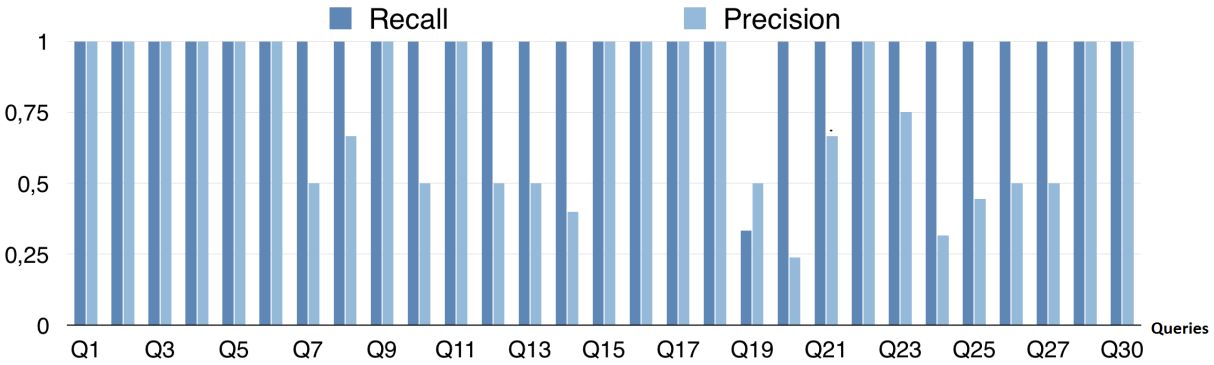


Figure 4.6 Precision and recall of joins for LIFT with traces of queries in the TPF web application, produced by a TPF client and executed in isolation on single TPF servers (DBpedia, Ughent, VIAF or LOV).

Figure 4.7 describes Q_7 and its deduced BGPs. BGP[1] is correct, while BGP[2] is not. When processing Q_7, like for all queries, the TPF client asks for the cardinality of each triple pattern and decides to begin with the rst 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 rst request (for the cardinality) 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 ltered in a post-processing. Such situation appears in 6/29 queries: Q_7, $Q_{12\ 14}$, Q_{21} and Q_{25}.

Concerning Q_8, Q_{10}, Q_{14}, Q_{20}, $Q_{23\ 27}$, LIFT nds two possible variables for a component of a triple pattern (a subject or an object). That is due to the fact that, during the NestedLoopDetection, a join is detected even if there is a partial inclusion between output and input-mappings of two triple patterns. This is more challenging to lter. We illustrate this with Q_8, in Figure 4.7. Deduced BGP of 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 that are very syntactically 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).

[8] Queries, TPF logs and LIFT results are available at: https://github.com/coumbaya/lift/blob/master/experiments.md

To summarize, in some cases LIFT deduces additional triple patterns and thus false joins with well deduced triple patterns, because an intersection between mappings of semantically similar triple patterns that are not originally joined[9]. But as right triple patterns are in general well deduced, recall is good.

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}

Figure 4.7 Deduced BGPs for LIFT with traces of Q_7 and Q_8 queries in the TPF web application, executed in isolation on the DBpedia TPF server.

4.3.3 Does LIFT resist to concurrency?

We grouped all queries of the TPF web application, into 6 generated collections of randomly chosen queries both on single or over federations of TPF servers, as presented in Tables 4.4 and 4.5 respectively. For each query set, we evaluated $LIFT(E(Q_1)) \bowtie \ldots \bowtie (LIFT(E(Q_n)) \neq LIFT(E(Q_1 \bowtie \ldots \bowtie Q_n))$ in terms of recall and precision of joins for different gap values. gap varies from 1% to 100% of the log duration. Each query set was shuffled 4 times and we calculated the average of LIFT results by gap[10].

Figures 4.8 and 4.10 show the average of precision whereas Figures 4.9 and 4.11 show the average of recall, when analyzing single TPF query logs. Figures 4.12 and 4.13 show the average of precision and recall respectively, when analyzing federated logs of TPF servers.

Concerning gap, according to its value increase we observe that globally precision and recall improve, as shown in Figures 4.8 - 4.10 and Figures 4.9 - 4.11. When gap is small (less than 50%) precision decreases significantly. A small gap leads LIFT to split values of an inner loop across different blocks i.e., the tpExtraction algorithm can not aggregate in one candidate tp all triple patterns of the inner operand of a join. This is explained from the pipelined nested-loop operator that is implemented by TPF clients. Actually, TPF clients evaluate consecutive joins of multiple triple patterns of a query in blocks of pushed mappings, without waiting first all output-mappings of a triple pattern to be pushed to the following. For more details see page 27 in Chapter 2.

[9] We consider that two semantically similar triple patterns match same triples.

[10] Note that as we vary the gap between two subqueries from 1 to 16 seconds, the duration of each shuffled log we produce diverges from some seconds to one hour and a half.

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	$UG_1 = \{Q_2; Q_{23}; Q_{25}; Q_{29}; Q_{30}\}$
LOV	$LV_1 = \{Q_{17}; Q_{18}; Q_{26}; Q_{29}; Q_{30}\}$
VIAF	$VF_1 = \{Q_{29}; Q_{30}\}$

Table 4.4 Query sets executed concurrently on single TPF servers (DBpedia, Ughent, VIAF or LOV).

Dataset	Query sets
Federated log	$LF_1 = \{Q_1; Q_2; Q_8; Q_{14}; Q_{22}\}$ $LF_4 = \{Q_4; Q_{23}; Q_{12}; Q_{24}\}$ $LF_2 = \{Q_3; Q_{11}; Q_{15}; Q_{20}; Q_{25}\}$ $LF_5 = \{Q_7; Q_{16}; Q_{17}; Q_{21}; Q_5\}$ $LF_3 = \{Q_6; Q_{13}; Q_{18}; Q_{19}; Q_{27}\}$ $LF_6 = \{Q_9; Q_{10}; Q_{26}; Q_{29}; Q_{30}\}$

Table 4.5 Query sets executed concurrently over a federation of TPF servers (DBpedia, Ughent, VIAF and LOV).

Concerning recall, LIFT is moderately impacted by concurrency. Indeed, 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 concurrently 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 over the set of deduced BGPs, could filter these false joins.

4.3.4 Analysis of the TPF log of USEWOD 2016

We ran LIFT with log slices, each of one hour, from the USEWOD 2016 traces [28] using a maximum gap (one hour). We obtain 595 BGPs of size >1 and 169,491 BGPs of size=1.

Table 4.14 describes the most frequently deduced BGPs. Unsurprisingly, most of them correspond to the queries available on the TPF web application. Observing that both queries of the TPF web application and deduced BGPs with the TPF log of USEWOD 2016 are similar, provides with a proof of concept for LIFT. BGP[1] corresponds to Q_1, while BGP[2] is like BGP[1] except that *dbpedia-owl:starring* is replaced by *dbprop:starring*. BGP[1] and BGP[2] do not co-exist in time, thanks to LIFT we observed that the "Brad Pitt" query was modified on 27/10/2014. This observation also provides with a proof of concept for LIFT. BGP[3] corresponds to the query used as the motivation example of [44], BGP[4] corresponds to Q_3, BGP[5] to Q_6, etc. In this top 14 list, only 1/3 of BGPs were unknown: BGP[6], BGP[7], BGP[8], BGP[12] and BGP[13]. In addition, we observe that almost all deduced BGPs by LIFT, start with a triple pattern containing a constant in its subject or object. The latter observation is explained from the fact, that triple patterns with constants are generally the most selective ones and a TPF client starts the query evaluation with them. As a TPF server receives the selective patterns first, they appear first in the log and thus in LIFT deductions.

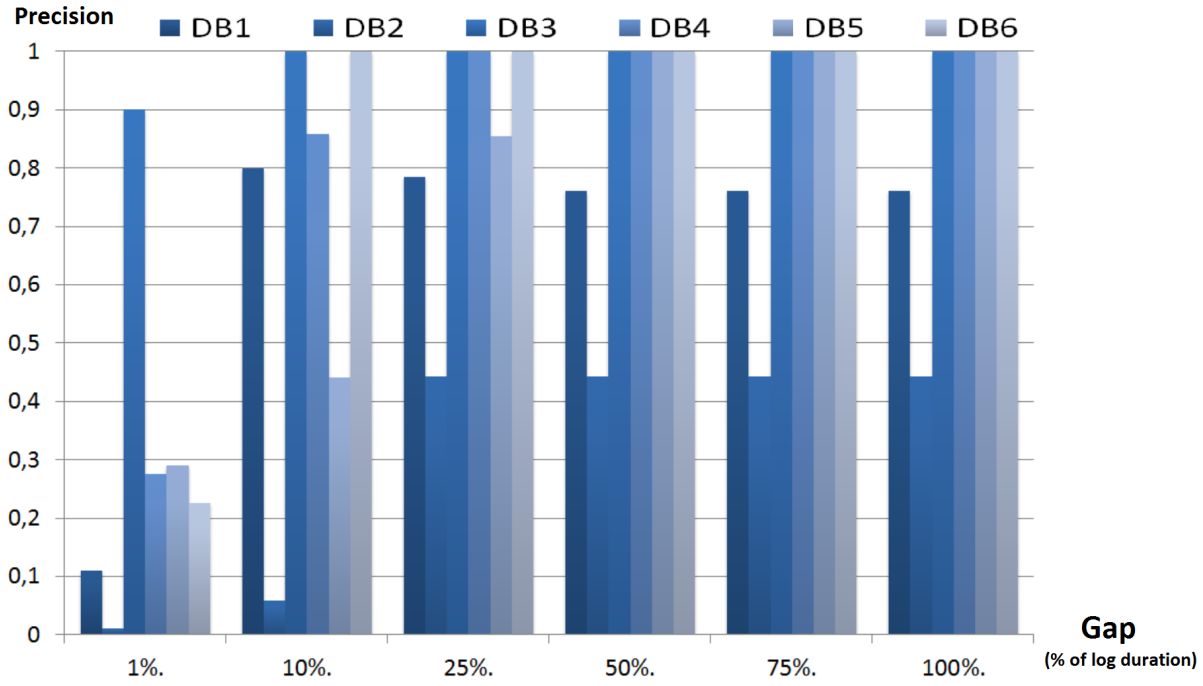


Figure 4.8 Precision of joins for LIFT per gap with traces of queries in the TPF web application, produced by a TPF client and executed in concurrence on the DBpedia TPF server.

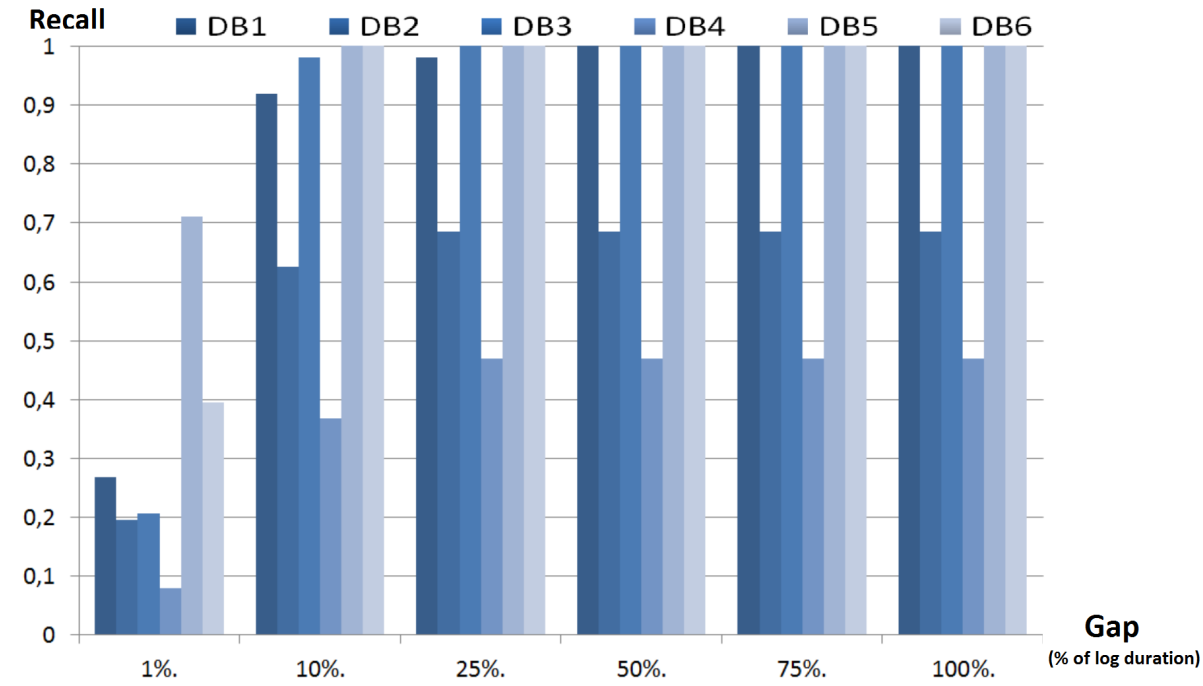


Figure 4.9 Recall of joins for LIFT per gap with traces of queries in the TPF web application, produced by a TPF client and executed in concurrence on the DBpedia TPF server.

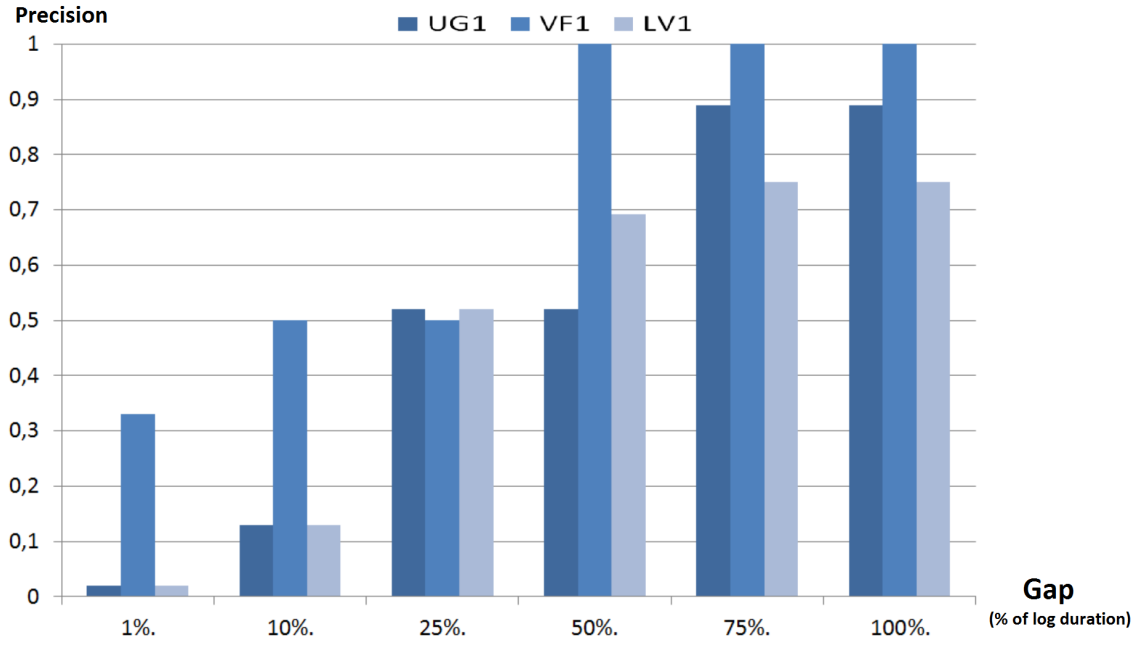


Figure 4.10 Precision of joins forLIFT per gap with traces of queries in the TPF web application, produced by a TPF client and executedn concurrence on single TPF servers (DBpedia, Ughent, VIAF or LOV).

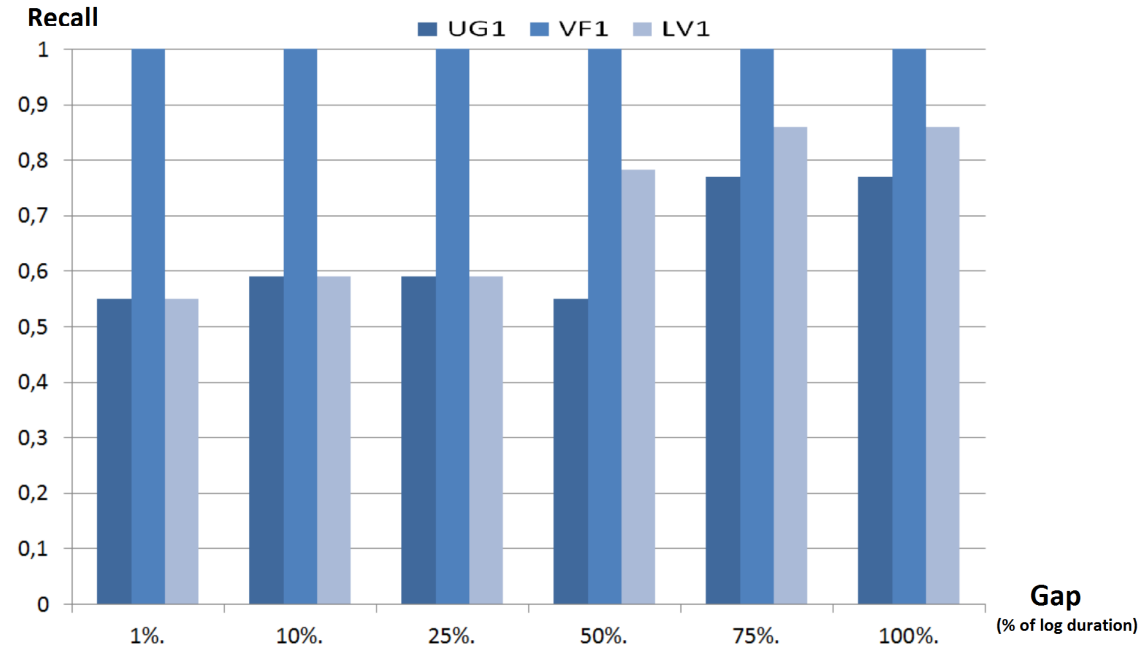


Figure 4.11 Recall of joins forLIFT per gap with traces of queries in the TPF web application, produced by a TPF client and executedn concurrence on single TPF servers (DBpedia, Ughent, VIAF or LOV).

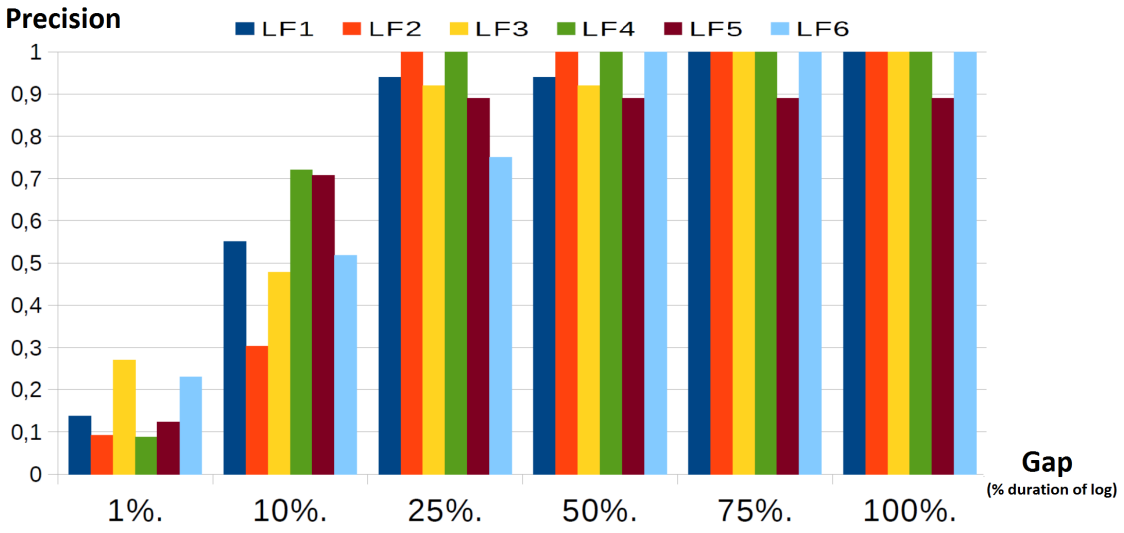


Figure 4.12 Precision of joins forLIFT per gap with traces of queries in the TPF web application, produced by a TPF client and executedn concurrence over a federation of TPF servers (DBpedia, Ughent, VIAF and LOV).

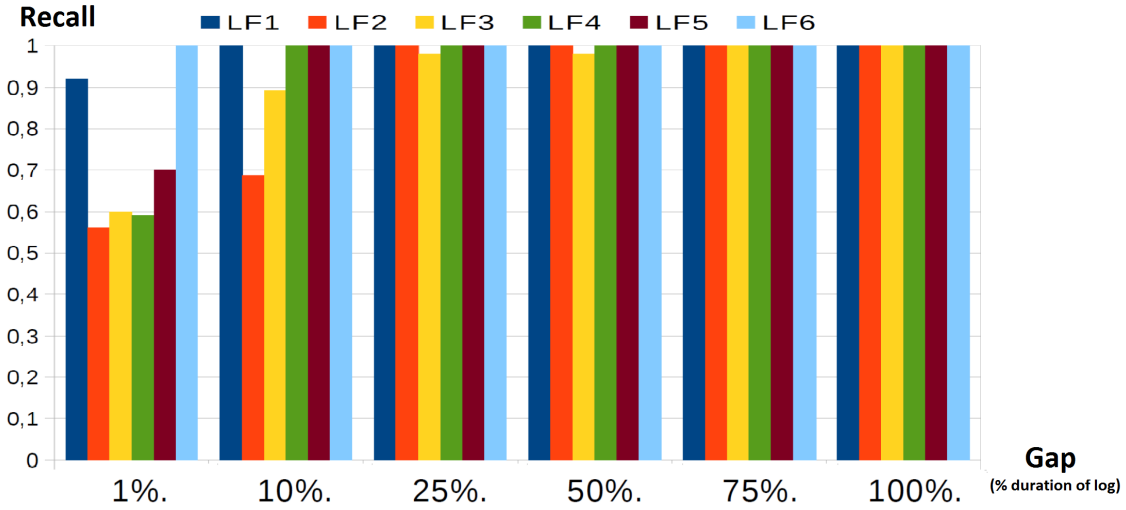


Figure 4.13 Recall of joins forLIFT per gap with traces of queries in the TPF web application, produced by a TPF client and executedn concurrence over a federation of TPF servers (DBpedia, Ughent, VIAF and LOV).

BGP[1]- deduced 126 times	BGP[2] - deduced 47 times
{?s1 rdfs:label "Brad Pitt"@en . ?s2 dbpo:starring ?s1 . ?s2 rdfs:label ?o3 . ?s2 dbpo:director ?o4 . ?o4 rdfs:label ?o5}	{?s1 rdfs:label "Brad Pitt"@en . ?s2 dbpprop:starring ?s1 . ?s2 rdfs:label ?o3 . ?s2 dbpo:director ?o4 . ?o4 rdfs:label ?o5}
BGP[3] - deduced 43 times	BGP[4] - deduced 34 times
{?s1 rdfs:label "York"@en . ?s2 dbpo:birthPlace ?s1 . ?s2 rdf:type dbpo:Artist}	{?s1 dbpedia owl:in uencedBy dbpedia:Pablo_Picasso . ?s1 rdf:type dbpedia owl:Artist . ?s1 dbpedia owl:birthDate ?o3}
BGP[5] - deduced 34 times	BGP[6] - deduced 20 times
{?s1 dbpprop:cityServed dbpedia:Italy . ?s1 rdf:type dbpo:Airport}	{dbpedia owl:Agent rdfs:subClassOf ?o1 . ?o1 rdfs:subClassOf ?o2}
BGP[7] - deduced 17 times	BGP[8] - deduced 16 times
{dbpedia owl:Activity rdfs:subClassOf ?o1 . ?o1 rdfs:subClassOf ?o2}	{?s1 rdfs:label "Trinity College, Dublin"@en . ?s2 dbpedia owl:almaMater ?s1 . ?s2 rdf:type dbpedia owl:Writer}
BGP[9] - deduced 15 times	BGP[10] - deduced 13 times
{?s1 rdf:type dbpedia owl:Book . ?s1 dbpedia owl:author ?o2}	{?s1 rdf:type yago:PeopleExecuted ByCruci xion . ?s1 rdf:type yago:Carpenters}
BGP[11] - deduced 11 times	BGP[12] - deduced 11 times
{?s1 dbpedia owl:ingredient ?o1 . ?s1 dbpedia owl:kingdom dbpedia:Plant}	{?s1 dbpedia owl:birthPlace dbpedia:Urbel_del_Castillo . ?s1 dbpedia owl:team ?o2}
BGP[13] - deduced 10 times	BGP[14] - deduced 10 times
{?s1 rdf:type foaf:Person . ?s1 foaf:isPrimaryTopicOf ?o2}	{?s1 dbpedia owl:type dbpedia:Dessert . ?s1 dbpedia owl:ingredient ?o2 . ?o2 dbpedia owl:kingdom dbpedia:Plant}

Figure 4.14 Frequent BGPs extracted with LIFT from the TPF log of USEWOD 2016.

To summarize, we presented LIFT, a BGP reversing approach that aims to infer BGPs of queries executed over TPF servers. We provided with experiments, illustrating LIFT's good recall and precision that depends not only to the similarity of concurrently executed queries but also execution parameters of LIFT.

5

FETA : Federated quEry TrAcking

Contents

In this chapter we presentFETA, our proposed approach that aims to answer the question: "If several SPARQL endpoints share their logs, can they track and approximate BGPs they process?"Compared to LIFT we address this problem only over federations of SPARQL endpoints, because single SPARQL endpoints are already aware of the whole queries that are addressed only to them as they are not decomposed in subqueries. Like LIFT, the challenge with this approach is to link maybe hundreds of subqueries per query execution and to be resistant in presence of concurrent execution of other federated queries. The di erence although withLIFT, is that FETA in addition must be able to (a) detect di erent physical joins operators i.e., exclusive group, nested-loop and symmetric hash joins, and (b) adapt to di erent optimization techniques, produced by query engines during query execution to push mappings through nested-loops from one triple pattern into another.

This chapter first illustrates the scientific problem we aim to solve, as described on page 39 in Chapter 2, Section 2.5 over the context of federated query processing, in Section 5.1. Thereafter, the BGP reversing approach of FETA is presented in Section 5.2. Finally, experiments are reported in Section 5.3.

5.1 Illustration example

In Figure 5.1, two data consumers, c_1 and c_2, execute concurrently federated queries CD_3 and CD_4 of FedBench [46] over the federation of SPARQL endpoints composed by LMDB, DBpedia InstanceTypes, DBpedia InfoBox and NYTimes. They use Anapsid [1, 2] or FedX [48] federated query engines.

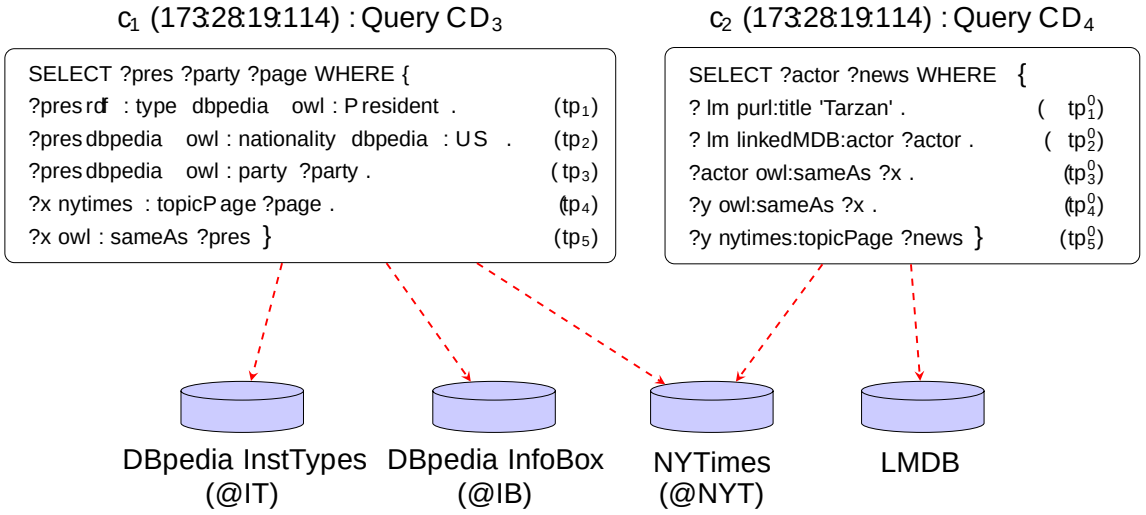


Figure 5.1 Concurrent execution of FedBench queries CD_3 and CD_4, produced by a federated query engine with 173.28.19.114 IP Address and executed over a federation of SPARQL endpoints.

Federated query engines, decompose SPARQL queries into a sequence of subqueries as partially presented in Tables 5.1 and 5.2 for query CD_3 using Anapsid and FedX, respectively. Lines in grey color correspond to answers of the queries in enumerated lines. As this federation of SPARQL endpoints receive only subqueries corresponding to physical execution plans, the original queries e.g., CD_3 and CD_4 remain unknown to the corresponding data providers.

In this chapter, we address the following research question: "If several SPARQL endpoints share their logs, can they track and approximate BGPs they process?" In particular, we aim to solve the scientific problem of BGPs reversing (cf. Definition 5 on page 40), for federated query processing over SPARQL endpoints. Compared to DFT, we address this problem only over federations of SPARQL endpoints. We do so, because queries addressed to a single source are not decomposed by query engines and thus are already know by the SPARQL endpoint they evaluates them. We also consider the definition, notation and property of query log (cf. Definition 4 on page 39), execution trace and resistance to concurrency (cf. Property 1 on page 39), respectively.

In our example, CD_3 can be decomposed into $\{tp_1^{@IT} : (tp_2 : tp_3)^{@IB} : (tp_4 : tp_5)^{@NYT}\}$, and NYTimes data provider just observes tp_4 and tp_5; it does not know that these triple patterns are joined with tp_1 from DBpedia InstanceTypes and $(tp_2; tp_3)$ from DBpedia InfoBox. Consequently, NYTimes provider does not know the real usage of data it provides.

	LD provider	IP	Time	Subquery/Answer
[1]	@IT	173..	11:24:19	SELECT ?pres WHERE { ?pres rdf:type dbpedia-owl:President }
				{ {"pres", "http://dbpedia.org/Ernesto_Samper" },... {"pres", "http://dbpedia.org/Shimon_Peres" },... {"pres", "http://dbpedia.org/Barack_Obama" },... }
[2]	@IB	173..	11:24:21	SELECT ?party ?pres WHERE { ?pres dbpedia-owl:nationality dbpedia:United_States . ?pres dbpedia-owl:party ?party }
				{ {{"party","http:.../Democratic_Party_%28United_States%29" }, {"pres", "http://dbpedia.org/Barack_Obama " }},... {{"party","http:.../Democratic_Party_%28United_States%29" }, {"pres", "http://dbpedia.org/Johnny_Anders " }},... {{"party","http:.../Republican_Party_%28US%29" }, {"pres", "http://dbpedia.org/Judith_Flanagan_Kennedy" }},... }
[3]	@NYT	173..	11:24:25	SELECT ?pres ?x ?page WHERE { ?x nytimes:topicPage ?page . ?x owl:sameAs ?pres . FILTER ((?pres= <http://dbpedia.org/Barack_Obama>),... || (?pres=<http://dbpedia.org/Johnny_Anders>),... || (?pres=<http://dbpedia.org/Judith_Flanagan_Kennedy>),...) }} LIMIT 10000 OFFSET 0
				{ {"pres","http://dbpedia.org/Barack_Obama" }, {"x","http://data.nytimes.com/47452218948077706853" }, {"page", "http://topics.nytimes.com/.../barack_obama/index.html"} }

Table 5.1 Partial federated log of CD_3 traces, produced by Anapsid ($E_{Anapsid}(CD_3)$) with 173.28.19.114 IP Address and executed over a federation of SPARQL endpoints. Answers are extracted from data providers in json format.

In our example, if c_1 and c_2 have different IP addresses then it is straightforward to split $E(CD_3 \bowtie CD_4)$ into $E(CD_3)$ and $E(CD_4)$, and apply the reversing function separately. However, in the worst case c_1 and c_2 share the same IP address[1]. In this case, we expect that $f(E(CD_3 \bowtie CD_4)) \neq f(E(CD_3)) \cup f(E(CD_4))$.

[1] Like TPF evaluation this can be produced when, for instance, a proxy server is used as a mediator between clients and federations of SPARQL endpoints.

	LD provider	IP	Time	Subquery/Answer
[1]	@IB	173..	17:04:08	SELECT ?pres ?party WHERE { ?pres owl:nationality <http://dbpedia.org/dbpedia.org/United_States>. ?pres owl:party ?party }
				{ {{"pres", " http:dbpedia.org/Barack_Obama "}, {"party", "http:.../Democratic_Party_%28United_States%29" }},... {{"pres", " http:dbpedia.org/Johnny_Anders " }, {"party", "http:.../Independent_%28politics%29" }}, ... {{"pres", " http:dbpedia.org/Judith_Flanagan_Kennedy " }, {"party", "http:.../Republican_Party_%28US%29" }},... }
[2]	@IT	173..	17:04:11	SELECT ?o_0 ?o_1 ?o_2... WHERE { {<http:dbpedia.org/Barack_Obama> rdf:type ?o_0. FILTER(?o_0 = <http://dbpedia.org/ontology/President>) } UNION {<http:dbpedia.org/Johnny_Anders> rdf:type ?o_1. FILTER(?o_1 = <http://dbpedia.org/ontology/President>) } UNION {<http:dbpedia.org/Judith_Flanagan_Kennedy> rdf:type ?o_2. FILTER(?o_2 = <http://dbpedia.org/ontology/President>) },... }
				{ { "o_0", "http://dbpedia.org/ontology/President" } { "o_1", "" } { "o_2", "" },... }
[3]	@NYT	173..	17:04:13	SELECT ?x WHERE { ?x owl:sameAs<http:dbpedia.org/Barack_Obama> . }
				{ { "x", "http://data.nytimes.com/47452218948077706853" } }
[4]	@NYT	173..	17:04:15	SELECT ?page WHERE { <http://data.nytimes.com/47452218948077706853> nytimes:topicPage ?page }
				{ { "page","http://topics.nytimes.com/.../barack_obama/index.html"} }

Table 5.2 Partial federated log of CD_3 traces, produced by FedX ($E_{FedX}(CD_3)$) with 173.28.19.114 IP Address and executed over a federation of SPARQL endpoints. Answers are extracted from data providers in json format.

Next, we present our proposed BGP reversing solution, Federated query TrAcking (FETA), which we evaluate with traces of FedBench [46] queries executed (i) in isolation and (ii) in concurrence, over federations of SPARQL endpoints.

5.2 FETA: a reversing approach

FETA is a system of algorithms based on heuristics, to implement the reverse function f. The idea is to detect exclusive groups, nested-loops and symmetric hash joins. In Table 5.2, the mappings returned from DBpedia InfoBox in line 2, are reused in DBpedia InstanceTypes in the next subquery, in line 3. We track such bindings in order to link different subqueries.

In this chapter, we make the following hypothesis:

1. We consider only bound predicates[2],
2. We consider that SPARQL endpoints do not use a web cache (this information can be easily obtained by data providers), and
3. We do not consider query engines use a cache (concerning both the location of SPARQL endpoints that evaluate triple patterns and also their answers).

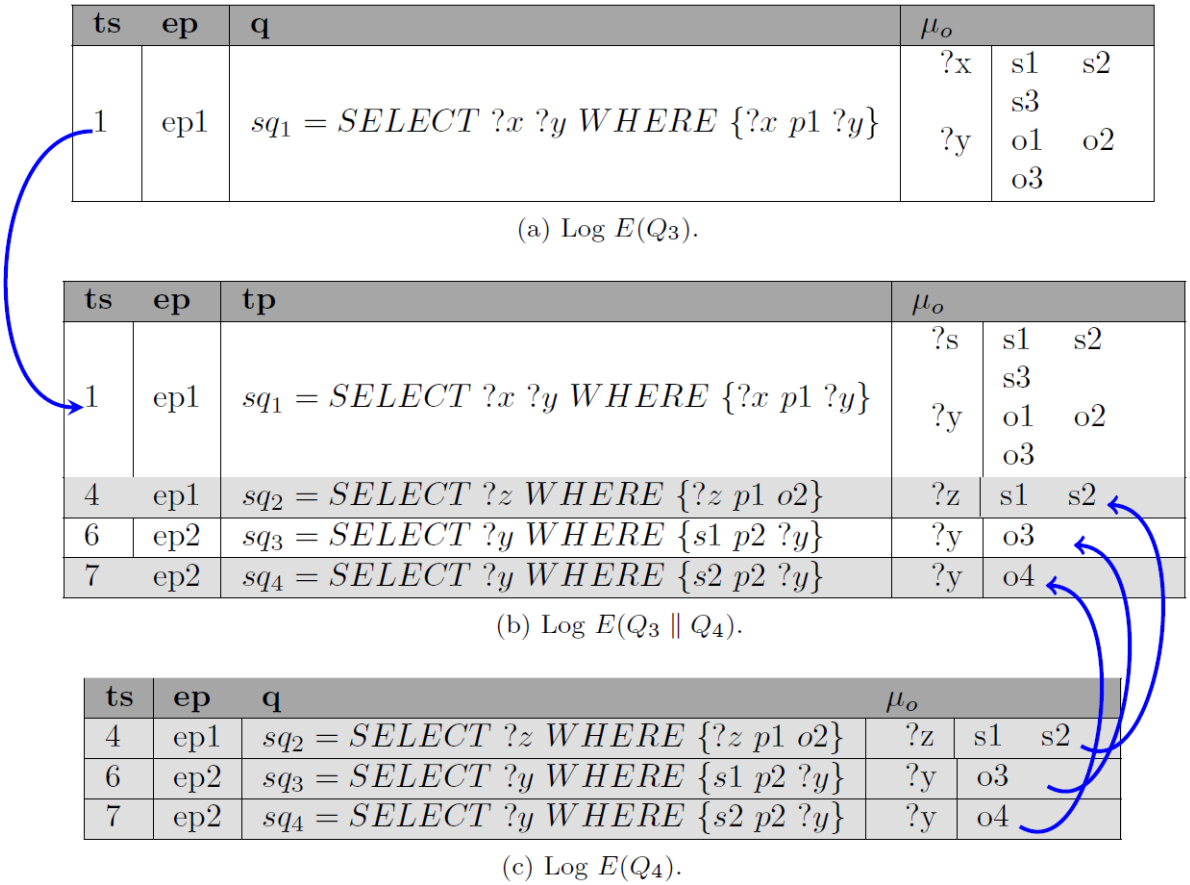
Figure 5.2 presents a simplified federated log of two SPARQL endpoints ep_1 and ep_2, corresponding to Q_3 = SELECT ?z ?y WHERE {?z p1 o2 . ?z p2 ?y}), Q_4 = SELECT ?x ?y WHERE {?x p1 ?y}) and $E(Q_3 \bowtie Q_4)$.

For the sake of simplicity, timestamps are transformed into integers. The IP address of the query engine is the same for Q_3 and Q_4, so we removed the ip column. Query engines use the same variable names for subqueries as those used in the original user queries, in contrast to TPF clients that rename them either as subject or object. ω represents the mappings of variables resulting from the evaluation of the triple pattern on data. Like LIFT, we call them output-mappings.

Algorithm 4 shows the five phases of FETA:

1. First, FETA cleans the input log from ASK queries, aggregates SELECT queries into merged SELECT queries if they differ only in their OFFSET values or are sent to different SPARQL endpoints, and then groups them into the same graph if they are syntactically joinable. We denote the set of graphs of subqueries as MSQ.
2. Second, FETA reduces MSQ into graphs of triple patterns, by merging them into a set of candidate triple patterns, which we denote as CTP. It allows to gather triple patterns of queries, that can be part of the same inner operand of a join. Compared to LIFT, we need also to save the information regarding which candidate tps are joined as exclusive groups or that are syntactically joinable. The former will be excluded during detection of nested-loops. The latter will be used to detect symmetric hash joins.
3. Next, FETA looks for an inclusion relationship among output and input-mappings of CTPs. If it does not exist, FETA splits candidate triple patterns to build it. This produces a set of graphs, which we denote as DTP, where nodes are reduced triple

[2] As reported in [8], predicates of triple patterns are frequently bound. Nevertheless, FETA like LIFT can be extended to deal with predicates just like they deal with subjects and objects.



ts	ep	q	μ_o		
1	ep1	$sq_1 = SELECT\ ?x\ ?y\ WHERE\ \{?x\ p1\ ?y\}$	?x ?y	s1 s3 o1 o3	s2 o2

(a) Log $E(Q_3)$.

ts	ep	tp	μ_o		
1	ep1	$sq_1 = SELECT\ ?x\ ?y\ WHERE\ \{?x\ p1\ ?y\}$	?s ?y	s1 s3 o1 o3	s2 o2
4	ep1	$sq_2 = SELECT\ ?z\ WHERE\ \{?z\ p1\ o2\}$	?z	s1	s2
6	ep2	$sq_3 = SELECT\ ?y\ WHERE\ \{s1\ p2\ ?y\}$	?y	o3	
7	ep2	$sq_4 = SELECT\ ?y\ WHERE\ \{s2\ p2\ ?y\}$	?y	o4	

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

ts	ep	q	μ_o		
4	ep1	$sq_2 = SELECT\ ?z\ WHERE\ \{?z\ p1\ o2\}$	?z	s1	s2
6	ep2	$sq_3 = SELECT\ ?y\ WHERE\ \{s1\ p2\ ?y\}$	?y	o3	
7	ep2	$sq_4 = SELECT\ ?y\ WHERE\ \{s2\ p2\ ?y\}$	?y	o4	

(c) Log $E(Q_4)$.

Figure 5.2 Examples of simplified logs of SPARQL endpoints, for Q_3 and Q_4 traces.

patterns and edges represent inclusion relationships between them. This detects nested-loops. Note that compared to LIFT, FETA adapts on different optimization techniques that are employed by query engines in order to detect pushed values from the outer into the inner operand of a nested-loop.

4. Thereafter, FETA looks for intersection relationship among output-mappings of DTPs that are connected only with unlabeled edges i.e., triple patterns that are syntactically joinable. This maintains the set of graphs DTP, by confirming that every syntactical connection between triple patterns correspond actually to an intersection relationship between their output-mappings. If not, their edge is removed. This detects symmetric hash joins, a heuristic not applied with LIFT.

5. Finally, FETA extracts BGPs from DTP graph set. Ideally, $FETA(E(Q_3 \parallel Q_4); gap)$ should compute the 2 BPGs of Q_3 and Q_4: $\{?z\ p1\ o2 : ?z\ p2\ ?y\}$ and $\{?x\ p1\ ?y\}$.

The basic intuition of FETA is to detect if mappings are bind in next requests but also if there exist and intersection between output-mappings of different requests. This can be challenging, as mappings can be: (i) bind several times (e.g., in star queries), (ii) bind partially as a side-effect of LIMIT clauses[3], or (iii) bind into a different concurrent query.

[3]FILTER clauses are in general detectable because they are pushed to relevant SPARQL endpoints to

As a real log can be huge, FETA analyzes the log using a constraint as a sliding window which is defined by a gap i.e., a time interval. When FETA reads an entry e in the log with a timestamp ts, it considers only entries reachable within the gap i.e., $ts \leq gap$.

Algorithm 4: Global algorithm of FETA

```
1 Function FETA (log, gap) is
    input  : a federated log, gap an interval in time units (seconds)
    output : a set of BGPs
    data   : MSQ a set of graphs of merged subqueries, CTP a set of candidate tps,
             DTP an edge-labelled set of graphs of deduced tps
2   MSQ  graphConstruction (log, gap)
3   CTP  graphReduction (MSQ)
4   DTP  nestedLoopDetection (CTP, gap)
5   DTP  symmetricHashDetection (DTP)
6   return BGP  bgpExtraction (DTP)
```

Section 5.2.1 presents the construction of syntactically joinable subqueries. Section 5.2.2 explains the reduction of this graph into a graph of candidate triple patterns. Section 5.2.3 describes the nested-loop detection. Section 5.2.4 presents the symmetric hash detection. Finally, Section 5.2.5 returns the BGP graphs that FETA deduces.

5.2.1 Graph construction

The graphConstruction heuristic first aggregates same or similar queries and then constructs graphs of syntactically joinable subqueries, from the input log. We consider that two queries are similar, if they differ only on their OFFSET values. Aggregated queries are represented by a merged SELECT query. All graphs of syntactically merged queries, form the MSQ Graph set. Queries that have same projected variables or constants are connected to the same graph, respecting a user-defined gap value.

A $m \in MSQ$, is a tuple $\langle ip, ts, q, \Omega_o, \{ep\}\rangle$ where ip is an IP address, ts is a pair of timestamps $(ts.min, ts.max)$ representing a range; when creating an m, both timestamps are identical and correspond to the timestamp of the current entry in the log. q is a SPARQL SELECT query, Ω_o (output-mappings) is the list of solution mappings for projected variables of q. $\{ep\}$ is the set of SPARQL endpoints that evaluate the merged query. This module executes two main functions: (a) $logPreparation(log, gap)$ and (b) $groupQueryGraphs(MSQ)$, as we explain next.

(a) logPreparation, prepares and cleans the input log. ASK queries are suppressed. Identical or subqueries differing only in their OFFSET values are aggregated in one single query, respecting a gap value, as we see in Algorithm 6, lines 5 and 6. If it is the first time we observe this query, then it is saved as a new graph, line 7. In this phase, each graph is composed by a single merged query. Timestamp of such aggregated query becomes an interval. Identical queries are sent twice to the same SPARQL endpoint to be sure obtaining an answer and to different to have complete answers. Similar queries with different OFFSETs are sent to avoid reaching the limit response of SPARQL endpoints.

minimize local processing at the data consumer [47], in contrast with TPF clients that bear the processing load of all SPARQL features.

Algorithm 5: Construction of a set of graphs of syntactically joinable subqueries

```
1 Function graphConstruction(log, gap) is
    input  : a federated log, gap an interval in time units (seconds)
    output : MSQ a set of graphs of subqueries
2   MSQ ← logPreparation(log, gap)
3   MSQ ← groupQueryGraphs(MSQ, gap)
4   return MSQ
```

Algorithm 6: Cleaning of the input log from ASK and redundant queries

```
1 Function logPreparation(log, gap) is
    input  : a federated log, gap an interval in time units (seconds)
    output : MSQ a set of graphs of merged subqueries
    data   : m a merged subquery of multiple query entries in the log
2   foreach e ∈ log do
3     m ← read(e) as (ip, (ts,ts), q, ∅, ∅)
4     if !isAsk(m.q) then
5       if ∃ m_k ∈ MSQ | ingap(m, m_k, gap) ∧ (m_k.ip = m.ip) ∧
        (m.q = m_k.moduloOFFSET(q)) then
6         (m_k.∅ ← m.∅); (m_k.ts.max = m.ts.max); (m_k.fe ← m.fe);
7       else MSQ.add(m)
8   return MSQ
```

(b) groupQueryGraphs, presented in Algorithm 7, incrementally connects single subquery graphs in MSQ. Different (merged) queries are connected depending on the gap value either on their common projected variables, or, if their triple patterns have common IRI/literal on their subjects or objects, line 4. 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. Joins detected until here are not labeled.

Algorithm 7: Grouping of syntactically joinable subqueries into the same graph

```
1 Function groupQueryGraphs(MSQ, gap) is
    input  : MSQ a set of single (merged) subquery graphs, gap an interval in time units
             (seconds)
    output : MSQ a set of connected graphs of merged subqueries
2   foreach m_i ∈ MSQ do
3     foreach m_j ∈ MSQ do
4       if ingap(m_i, m_j, gap) ∧ (m_i.ip = m_j.ip) ∧
        (sameProjectedVars(m_i.q, m_j.q) ∨ sameConstants(m_i.q, m_j.q)) then
5         MSQ.addEdge(m_i, m_j)
6         break;
7   return MSQ
```

In our example, with a gap equal to 5, two graphs are constructed: $MSQ = \{\{sq_1, sq_3, sq_4\},$

f (sq1; sq3); (sq1; sq4); (sq3; sq4g i; h fsq2g i g[4], as we see in Figure 5.3 on page 92.

ts	ep	q	μ_o	
1	ep1	$sq_1 = SELECT\ ?x\ ?y\ WHERE\ \{?x\ p1\ ?y\}$	?s ?y	s1 s2 s3 o1 o2 o3
4	ep1	$sq_2 = SELECT\ ?z\ WHERE\ \{?z\ p1\ o2\}$	?z	s1 s2
6	ep2	$sq_3 = SELECT\ ?y\ WHERE\ \{s1\ p2\ ?y\}$	?y	o3
7	ep2	$sq_4 = SELECT\ ?y\ WHERE\ \{s2\ p2\ ?y\}$	?y	o4

Figure 5.3 Deduced graphsm1, m2 2 MSQ, in blue and red colors respectively, produced by Algorithm 7 for gap=5.

5.2.2 Graph reduction

graphReduction aims to transform graphs of queries into set of triple patterns. Triple patterns that belong to di erent queries in MSQ, are aggregated if they seem to participate in the same outer or inner operand of a join. Aggregated triple patterns are represented by a candidate triple pattern All candidate triple patterns form the CTP set.

Algorithm 8: Reduction of a set of subquery graphs into a set of CTPs

```
1  Function graphReduction(MSQ) is
       input  : MSQ a set of graphs of merged subqueries
       output : CTP a set of candidate tps
       data   : CTP_m a temporary set of candidate tps for each merged subquery
2      foreach m 2 MSQ do
3          CTP_m read(m:q) asf (ip, (ts; ts), tp,  o,  i, f epg, fh ig) g
4          foreach c2 CTP_m do
5              switch ctp do
6                  case ?s p o: ctp ?s p ?o_in ; c_i ?! |o
7                  case s p ?o: ctp ?s_in p ?o ; c_i ? |s
8                  case s p o: ctp ?s_in p ?o_in ; c_i ? |s, ?! |o
9                  case ?s p ?o: ctp ?s p ?o ; c_i ;
10             if 9 c_k 2 CTP | ( c_k:ip = cip) ^ ( ctp = c_k:tp) then
11                 ( c_k. o [ c o); ( c_k. i [ c i); ( c_k.f epg [ cf epg); ( c_k:fh ig [ cfh ig);
                   ( c_k:ts:max = cts:max);
12             else CTP.add(c)
13     return CTP
```

The CTP de ned for FETA is enhanced with two additional arguments, compared to the one de ned for LIFT: (a) a set of SPARQL endpoints, which evaluate the candidate triple pattern and (b) a set of pair-tuples, each representing to which other triple pattern

[4]To simplify, all annotations to sqi are omitted.

the current one is joined and with what type of join i.e., syntactical (which would be identi ed as symmetric hash or be removed), exclusive group or nested-loop. In this phase, we identify syntactically or exclusive group joins, that will be excluded during following heuristics. We re de ne next, the notion of candidate triple pattern.

A c2 CTP is a tuple hip; ts; tp; $_o$; $_i$; f epg; fh igi . ip is an IP address. ts is a pair of timestamps (ts:min; ts:max) representing a range; when creating a candidate triple pattern both timestamps are identical and correspond to the timestamp of the current entry in the log. tp is a triple pattern, $_o$ (output-mappings) is the list of solution mappings for variables of tp. $_i$ (input-mappings) is a set of mappings built during the graphReduction. Basically, we replace any constant of tp by a variable, we use for subject and ! for object. Replaced constants are regrouped in $_i$. f epg is the set of SPARQL endpoints that evaluate the current triple pattern. fh ig (labeled edges) is a set of key-valuetuples, each representing to which other candidate triple pattern i.e. key the current triple pattern is joined and with what type of join i.e., value: "unlabeled" (or eventually "symmetricHash"), "exclusiveGroup" or "nested-loop".

ts	ep	q	μ_o	
1	ep1	$sq_1 = SELECT\ ?x\ ?y\ WHERE\ \{?x\ p1\ ?y\}$	?x ?y	s1 s2 s3 o1 o2 o3
6	ep2	$sq_3 = SELECT\ ?y\ WHERE\ \{s1\ p2\ ?y\}$	?y	o3
7	ep2	$sq_4 = SELECT\ ?y\ WHERE\ \{s2\ p2\ ?y\}$	?y	o4

(a) Graph of subqueries $m_1 \in MSQ$.

id	ts	ep	tp	μ_o		μ_i	
1	ep1	1,1	$?x$ p1 $?y$	?x ?y	s1 s2 s3 o1 o2 o3		
2	ep1	4,4	?z p1 $?\omega$	?z	s1 s2	$?\omega$	o2
3	ep2	6,7	$?\sigma$ p2 $?y$	?y	o3 o4	$?\sigma$	s1 s2

(b) CTP List of candidate triple patterns obtained by Algorithm 8.

ts	ep	q	μ_o	
4	ep1	$sq_2 = SELECT\ ?z\ WHERE\ \{?z\ p1\ o2\}$	?z	s1 s2

(c) Graph of subqueries $m_2 \in MSQ$.

Figure 5.4 Federated log and CTP List, produced by Algorithm 8 for gap=5.

The heuristic graphReduction of Algorithm 8 di ers from the heuristic employed by LIFT of Algorithm 2, in three points: (i) it does not need a gap value, as triple patterns that seems to be part of the same outer or inner operand of a join have already been grouped in the same graph regarding the gap, (ii) it iterates over temporary candidate

triple patterns, CTP_m, as they are red from each query in MSQ[5] and not log entries, and nally (iii) it maintains exclusive groups or syntactical joins of CTP_m, into the nal CTP list. Actually, the graphReduction module will signi cantly reduce the size of each graph in MSQ, because nested-loops can be executed with hundreds of subqueries. Figure 5.4 illustrates the CTP List after graphReduction, for our motivating example.

5.2.3 Nested-loop join detection

Like LIFT, this heuristic identi es nested-loops joins between pairs of candidate patterns, as described in Algorithm 9. In particular, nestedLoopDetection builds a set of graphs of deduced triples patterns, which we denote as DTP, by linking di erent candidate triple patterns if there is a relation of inclusion between them[6]. The di erence with LIFT is that edges between triple patterns created with this heuristic, are labeled as "nested loop", to be excluded during the symmetricHashDetection.

Algorithm 9: Detection of nested-loop joins

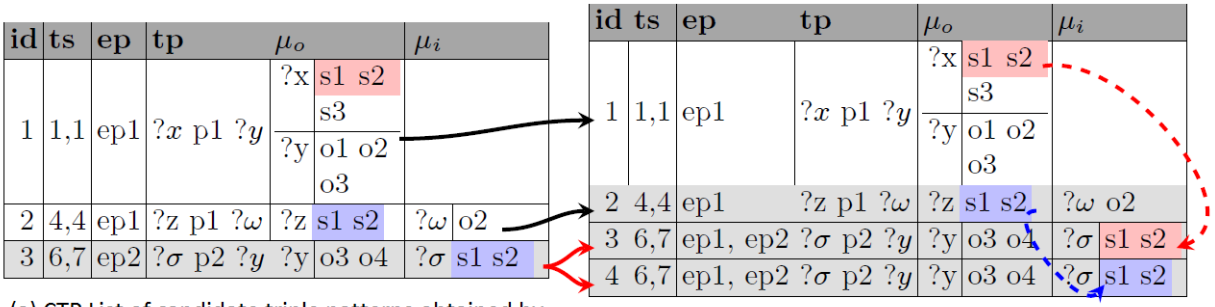
```
1  Function nestedLoopDetection (gap, CTP) is
   |  input  : gap an interval in time units (seconds), CTP a list of candidate tps
   |  output : DTP an edge-labelled set of graphs of deduced tps
2  |  foreach c2 CTP do
3  |  |  if split(c) 6= ; then CTP.insertAfter(cid, split(c));
4  |  |  else DTP.addnode(c);
5  |  |  foreach v_o 2 vars(c_o) do
6  |  |  |  foreach (c_k, v_i) 2 {(c_k, v_i) | c_k 2 CTP ^ (c_k.id > cid) ^ (ingap(c_k, c, gap)) ^
   |  |  |  9 v_i 2 vars(c_k : _i) | (c_k : _i(v_i) \ c_o(v_o) 6= ;)} do
7  |  |  |  |  if !(unlabeldedEdge(c, c_k) _ exclusiveGroupEdge(c, c_k)) ^ (c_k : _i(v_i) c_o(v_o))
   |  |  |  |  then
8  |  |  |  |  |  DTP.addnode(c_k);
9  |  |  |  |  |  DTP.addEdge(c, c_k, (v_o,v_i), "nested loop");
10 |  |  |  |  else DTP.addnode(s=split(c_k, v_i, c, v_o));
11 |  |  |  |  |  DTP.addEdge(c, s, (v_o,v_i), "nested loop");
12 |  |  |
13 |  return DTP;
```

5.2.4 Symmetric hash join detection

symmetricHashDetection identi es possible joins between output-mappings of pairs of deduced tps. In particular, this module deals with pairs of triple patterns that have not been connected with "exclusiveGroup" or "nested loop" labels, but are syntactically connected by transitivity as the queries that contained them were already syntactically joined.

[5] Note that a query may correspond to more than one joined candidate triple patterns, a join named exclusive group

[6] Note that in [35], we de ned the notion of inverse mapping in order to detect the inclusion of input-mapping values of the inner from output-mappings of the outer triple pattern of a nested loop.



id	ts	ep	tp	μ_o		μ_i	
1	1,1	ep1	?x p1 ?y	?x	s1 s2 s3		
				?y	o1 o2 o3		
2	4,4	ep1	?z p1 ?ω	?z	s1 s2	?ω	o2
3	6,7	ep2	?σ p2 ?y	?y	o3 o4	?σ	s1 s2

(a) CTP List of candidate triple patterns obtained by Algorithm 8.

id	ts	ep	tp	μ_o		μ_i	
1	1,1	ep1	?x p1 ?y	?x	s1 s2 s3		
				?y	o1 o2 o3		
2	4,4	ep1	?z p1 ?ω	?z	s1 s2	?ω	o2
3	6,7	ep1, ep2	?σ p2 ?y	?y	o3 o4	?σ	s1 s2
4	6,7	ep1, ep2	?σ p2 ?y	?y	o3 o4	?σ	s1 s2

(b) DTP set of graphs of deduced triple patterns obtained by Algorithm 9.

Figure 5.5 CTP List and DTP Graph set, produced by Algorithm 3 for gap=5.

Algorithm 10: Detection of symmetric hash joins

```
1 Function symmetricHashDetection (DTP) is
    input  : DTP an edge-labelled set of graphs of deduced tps, either as "unlabeled",
             "exclusiveGroup" or "nested-loop".
    output : DTP an edge-labelled set of graphs of deduced tps, either as
             "exclusiveGroup", "nested-loop" or "symmetricHash"
2   foreach dtp_i ; dtp_j 2 DTP do
3     if unlabeldedEdge(dtp_i ; dtp_j ) then
4       if CheckConceptSimilarity (dtp_i : _o ; dtp_j : _o) then
5         if (dtp_i : _o \ dtp_j : _o) 6= ; then
6           DTP.replaceEdge(dtp_i , dtp_j , "symmetricHash ");
7         else
8           DTP.removeEdge(dtp_i , dtp_j );
9   return DTP
```

Algorithm 10 shows how the symmetricHashDetection proceeds over pairs of deduced triple patterns. First, Line 3 identifies edges of syntactically joined triple patterns. Then, Line 4, checks if output-mappings of these triple patterns are on same or similar concepts. We consider that two concepts are similar if they are connected by the "sameAs" ontology. This heuristic is presented in Algorithm 11. The idea is that through a query executed on SPARQL endpoints, it will be possible to know if two sets of IRIs of syntactically joined triple patterns are actually on same or similar concepts[7]. Then, in Line 6, if the intersection of the output-mappings of deduced triple patterns is not an emptyset, they are connected through an edge labeled as "symmetricHash". Otherwise, the edge representing a syntactical join is removed, in Line 8.

This heuristic produces false positives because it infers all possible joins that can be made locally at the query engine. If a star-shape join of triple patterns exists, all possible combinations of joins will be deduced instead of the subset of joins chosen by the query engine. The consequence for FETA, compared to LIFT, is that it is more vulnerable to

[7] Another way to do this is to have locally at the data consumer, all ontologies of the federation. The advantage is to avoid surcharging SPARQL endpoints, but the risk is to have old versions of ontologies.

Algorithm 11: Check for same/similar concepts of output-mappings of DTPs

```
1  Function checkConceptSimilarity (dtp_i : _o; dtp_j : _o) is
       input  : a pair of output-mappings of two deduced tps
       output : b a boolean value, to verify if two deduced tps have same or similar concepts
       data   : q_iri a SPARQL query to retrieved concepts and parent concepts of an IRI
2      foreach iri_l 2 dtp_i : _o do
3          q_iri_l  SELECT distinct ?class ?parent WHERE {
4                       iri_l a ?class .
5                       ?class rdfs:subClassOf ?parent }
6          foreach iri_k 2 dtp_j : _o do
7              q_iri_k  SELECT distinct ?class ?parent WHERE {
8                           iri_k a ?class .
9                           ?class rdfs:subClassOf ?parent }
10             b  areSameOrSimilar (execute(q_iri_k); execute(q_iri_l))
11             return b
```

detect false joins because of the SymmetricHashDetection heuristic. Like LIFT, FETA privileges recall to the detriment of precision. In our example, we detect one symmetric hash join between DTP[1] and DTP[4] as there exist an intersection for their output-mappings on variable ?y i.e., ?y 7! o3, as we see in Figure 5.6.

id	ts	ep	tp	μ_o		μ_i	
1	1,1	ep1	?x p1 ?y	?x	s1 s2 s3		
				?y	o1 o2 o3		
2	4,4	ep1	?z p1 ?ω	?z	s1 s2	?ω	o2
3	6,7	ep1, ep2	?σ p2 ?y	?y	o3 o4	?σ	s1 s2
4	6,7	ep1, ep2	?σ p2 ?y	?y	o3 o4	?σ	s1 s2

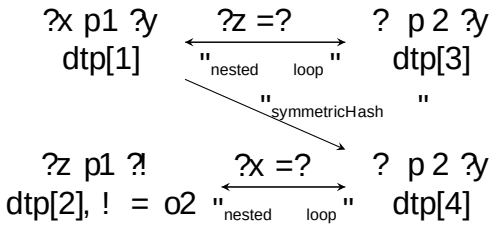
Figure 5.6 DTP Graph set with detection of a symmetric hash joint between DTP[1] and DTP[4], produced by Algorithm 10 for gap=5.

5.2.5 BGP extraction

Figure 5.7 represents the connected components of DTP shown in Figure 5.5. From this representation, it is easy to compute the final BGPs with a variable renaming and restitution of an IRI/literal in place of ! when there is only one input-mapping, for our example it is "o2".

5.2.6 Time complexity of FETA

The computational complexity of the global algorithm of FETA is in the worst case $O(N^2 + N \quad M + M^2)$, while in the best case $O(N + M^2)$. N is the number of queries in the log



(a) {?z p1 o2 . ?z p2 ?y . ?x p1 ?y . ?x p2 ?y}

Figure 5.7 Connected components of the DTP Graph set, produced by Algorithm 10 for gap=5.

and M is the number of candidate triple patterns of CTP.

The cost of the graphConstruction heuristic, is the addition of costs of logPreparation and commonJoinCondition. logPreparation in Algorithm 6 costs O(N). The worst case complexity of commonJoinCondition in Algorithm 7 is $O(N^2)$, when all queries in the log are compared together. On the other hand, ideally each subquery is syntactically joined with just the previous one i.e., when they belong to the same execution plan. In the latter case, commonJoinCondition costs O(N). Therefore, the worst time complexity of graphConstruction is O(N) + $O(N^2)$, that gives $O(N^2)$, and the best O(N) + O(N), that gives O(N). graphReduction in Algorithm 8 costs O(N M), as it extracts and merges similar triple patterns of already syntactically connected queries in MSQ. nestedLoopDetection in Algorithm 3 costs $O(M^2)$ as it compares every candidate tp with any other. symmetricHashDetection in Algorithm 10 also costs $O(M^2)$. First, it checks all pairs of triple patterns of DTP that are syntactically joinable, for same or similar concepts and then for a possible intersection, thus O(2 M^2) that gives $O(M^2)$.[8] Finally the cost of extracting BGPs is linear to the size of DTP, that is O(M). To summarize, the worst case complexity of FETA is $O(N^2 + N M + M^2 + M^2 + M)$, that gives $O(N^2 + N M + M^2)$, while the best complexity is $O(N + N M + M^2 + M)$ or equivalently O(N M + M^2). If we bypass the graphConstruction phase, the complexity of FETA is always the same i.e. O(N M + M^2), like LIFT.

The overload produced by FETA, like LIFT, is high but we underline that the size of the log corresponds to a slicing window of time e.g., a separate log for each hour of the day, and that the log analysis can be made as a batch processing.

[8] Note that even if only nested-loops or symmetric hash joins were used to evaluate a query, the cost of nestedLoopDetection and symmetricHashDetection would always be the same as all pairs of triple patterns must be compared together.

5.3 Evaluation

The goals of the experiments is to evaluate precision and recall of FETA's results. In Section 5.3.1 we present the experimental testbed of FETA. In Section 5.3.2 we evaluate precision and recall of FETA, with traces of federated queries executed in isolation. In Section 5.3.3 we evaluate precision and recall of FETA, with traces of federated queries executed concurrently under a worst case scenario, that is when they come from the same IP address. In contrast with LIFT, neither a public set nor a log with traces of real federated queries executed over the Linked Data does exist, to the best of our knowledge[9].

5.3.1 Experimental tesbed of FETA

Experiments are evaluated by reusing the queries and the setup of FedBench [46]. We use the collections of Cross Domain (CD) and Life Science (LS), each one has 7 federated queries. CD queries concern datasets of DBpedia[10], NY Times, LinkedMDB, Jamendo, Geonames and SW Dog Food. LS queries use datasets of DBpedia, KEGG, Drugbank and CheBi. We setup 19 SPARQL endpoints using Virtuoso OpenLink[11] 6.1.7.

Query	Anapsid	FedX	Query	Anapsid	FedX
CD1	14	164	LS1	2	32
CD2	4	38	LS2	34	154
CD3	142	196	LS3	872	4736
CD4	4	138	LS4	10	36
CD5	4	82	LS5	792	946
CD6	16	596	LS6	3252	20908
CD7	52	638	LS7	240	1000
Total	236	1852	Total	5202	27812

Table 5.3 Number of requests of SELECT subqueries for CD and LS queries, produced with Anapsid or FedX and executed in isolation over a federation of SPARQL endpoints.

We executed federated queries with Anapsid 2.7 and FedX 3.0. We configured Anapsid to use Star Shape Grouping Multi-Endpoints (SSGM) heuristic[12]. We captured http requests and answers from SPARQL endpoints with justniffer 0.5.12[13]. We implemented a tool to shuffle several logs of queries executed in isolation, according to different parameters[14]. Thus, given $E(Q_1); :::; E(Q_n)$ we were able to produce different significant representations of $E(Q_1 \ k \ ::: \ k \ Q_n)$, like LIFT. Produced traces with this tool vary in (i) the order of queries, (ii) the number of subqueries of the same query, appearing

[9] On the other hand, there exist a public set of queries executed over single SPARQL endpoints [41]

[10] DBpedia is distributed in 12 data subsets (http://fedbench.fluidops.net/resource/Datasets), in our setup, DBpedia Ontology dataset is duplicated in all SPARQL endpoints, so we install 11 SPARQL endpoints for DBpedia instead of 12.

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

[12] The difference of SSGM and SSGS, as presented in Chapter 2, is that the latter minimizes the scope of addressed SPARQL endpoints to the first that can evaluate a triple pattern.

[13] http://justniffer.sourceforge.net/

[14] The program to shuffle several execution logs in isolation, used as input either to MINEPI, LIFT or FETA, is available at: https://github.com/coumbaya/traceMixer

continuously in the shu ed log (blocks of 1 to 16 subqueries), and (iii) the delay between each subquery (from 1 to 16 units of time). Source code of FETA is available at https://github.com/coumbaya/feta .

Table 5.3 presents the number of requests produced by FedX and Anapsid, for the execution of FedBench queries in isolation. Table 5.4 presents the runtimes of FETA for each execution trace produced in isolation[15]. As we observe, the number of subqueries produced by query engines can be up to 20908 i.e., for query LS6 executed with FedX, which is the most time consuming for FETA to analyse. In addition we observe that there is a signi cant di erence between FedX and Anapsid. For instance, when LS6 is executed with Anapsid it produces 3252 subqueries i.e., 6 times less than FedX. This is explained by two reasons. First, Anapsid uses bushy tree execution plans which is proven to create less requests than the left-linear tree execution plans of FedX, where consecutive joins between multiple triple patterns are produced sequentially. Second, the user-de ned block size of bound joins for FedX is generally smaller than the constant block size of FILTER options for Anapsid, thus the latter produces more requests during nested-loops. For more details see page 35 in Chapter 2.

Query	Anapsid	FedX	Query	Anapsid	FedX
CD1	< 1	< 1	LS1	< 1	< 1
CD2	< 1	< 1	LS2	< 1	< 1
CD3	< 1	< 1	LS3	4	10
CD4	< 1	< 1	LS4	14	< 1
CD5	< 1	< 1	LS5	1	2
CD6	33	< 1	LS6	10	271
CD7	< 1	< 1	LS7	98	2
Average	4.7	< 1	Average	43	40.7

Table 5.4 Runtimes (seconds) of FETA with traces of CD and LS queries, produced with Anapsid or FedX and executed in isolation over a federation of SPARQL endpoints.

5.3.2 FETA deductions of queries in isolation

Like LIFT, for each query of the Cross Domain and Life Science collections, we ran FETA (E(Q),1). Figures 5.9 to 5.10 present precision and recall of FETA's deductions in terms of triple patterns and Figures 5.11 - 5.12 in terms of joins, by query and query engine. In average, we obtained 94,64% of precision and 94,64% of recall of deduced triple patterns. We obtain 79,40% of precision and 87,80% of recall for detected joins. FETA succeeds in deducing 11 out of 14 exact BGPs from Anapsid traces, and 7 out of 14 from FedX traces. Globally FETA nds 18/28 exact BGPs i.e., 64%. If we include Union queries where all triple patterns are deduced, FETA nds (18+3)/28 queries i.e., 75% of FedBench queries.

From Anapsid traces, deduced BGPs correspond to CD and LS queries except for Union queries i.e., CD_1, LS_1 and LS_2. For CD_1, presented in Figure 5.8a, FETA gives one BGP instead of two, because of the joinable common variables and the common IRI used in their BGPs. This query has two BGPs but a join is possible between them. As the Union of each query is made locally at the query engine, FETA deduces a symmetric

[15] We run our experiments in Linux 64 bit machine, with 32 CPUs and 800 Mhz CPU speed.

```
SELECT ?predicate ?object WHERE {
{ dbpedia:Barack_Obama ?predicate ?object }   (tp1)
UNION
{ ?subject owl:sameAs dbpedia:Barack_Obama    (tp2)
  ?subject ?predicate ?object }}              (tp3)
```

(a) CD1

```
SELECT ?drug ?melt WHERE{
{ ?drug drugbank:meltingPoint ?melt           (tp1) }
UNION
{ ?drug dbpedia-owl-drug:meltingPoint ?melt}} (tp2)
```

(b) LS1

Figure 5.8 Two UNION queries of FedBench.

hash join: $FETA(E_{Anapsid}(CD_1)) = \{(tp_2 : tp_3)^{@NYT} : tp_1^{@DBpedia}\}$. The deduction is similar for LS_2. For LS_1, presented in Figure 5.8b, FETA deduces only the first BGP because Anapsid does not send a subquery for the second BGP of the Union (tp2). From its source selection process, Anapsid knows that there is no SPARQL endpoint that can evaluate tp2 and only tp1 is send to Drugbank.

From FedX traces, deduced BGPs correspond exactly to the original BGPs of 7 queries: CD_2, CD_3, CD_5, CD_6, CD_7, LS_4 and LS_7. For LS_1, FETA finds one BGP instead of two but unlike Anapsid, all triple patterns are well deduced. All other problems of deduction come from the nested-loop detection of FETA. For CD_1 and LS_2, FETA fails to find some triple patterns. We illustrate what happens on CD_1. Instead of finding the object of tp_2 that is an IRI, it finds the variable ?object. The reason is that this IRI is contained in the mapping of tp_3 that is used in a nested-loop with tp_1.

Concerning CD_4, LS_3, LS_5, and LS_6, FETA finds two possible variables for a component of a triple pattern (a subject or an object). That is because during the NestedLoop Detection, a join is detected even if there is a partial inclusion between output and input-mappings. We illustrate this case with CD_4 (see Figure 5.1 on page 85). FETA finds that two variables may correspond to the subject of an inner operand of a join: ?y and ?actor. That is because the set of mappings of ?y corresponds to a subset of the mappings of ?actor. As FETA can not decide which variable is the good one it produces two triple patterns, the good one with ?y (tp5) and another with ?actor ($tp5'$)[16]. In this case: $FETA(E_{FedX}(CD_4)) = \{(tp_1 : tp_2)^{@LMDB} : (tp_3 : tp_4)^{@Geonames;etc:} : tp_5^{@NYT} : tp_5'^{@NYT}\}$.

To summarize, in some cases FETA like LIFT deduces additional triple patterns and thus false joins with well deduced triple patterns, because an intersection between mappings of semantically similar triple patterns that are not originally joined[17]. Furthermore, FETA compared to LIFT detects additional false positives of joins, because of the symmetricHashDetection heuristic. But as right triple patterns are in general well deduced, recall is good.

[16] Triple patterns tp3 and tp4 are joined as an exclusive group to: Geonames, NYT, Jamendo, SWDF, LMDB, DBpediaNYT, DBpediaLGD, representing an exclusive group.

[17] We consider that two semantically similar triple patterns match same triples.

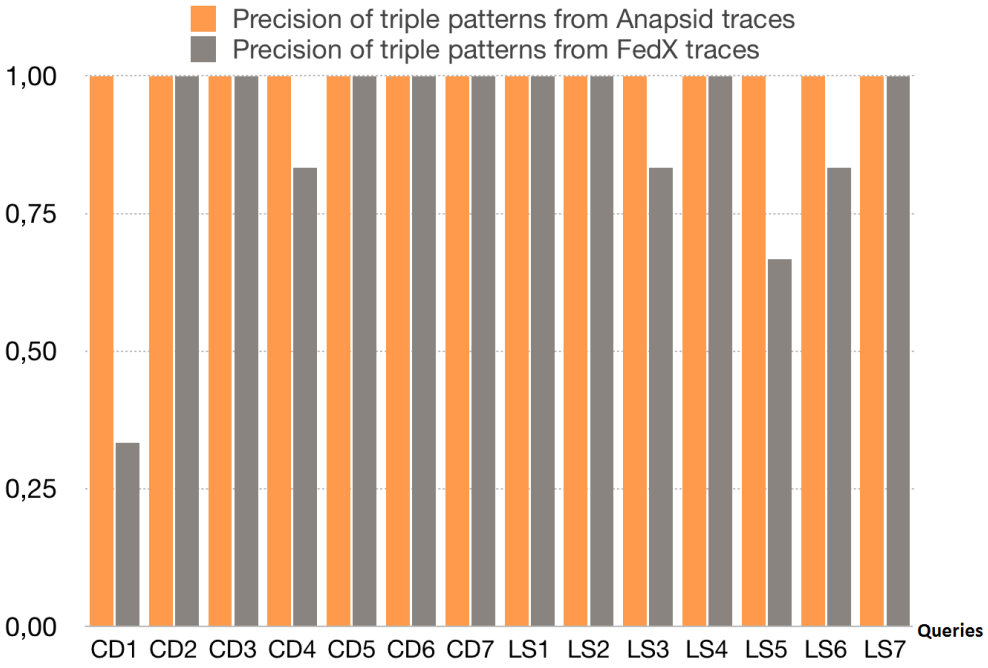


Figure 5.9 Precision of triple patterns for FETA with traces of CD and LS queries, produced with Anapsid or FedX and executed in isolation over a federation of SPARQL endpoints.

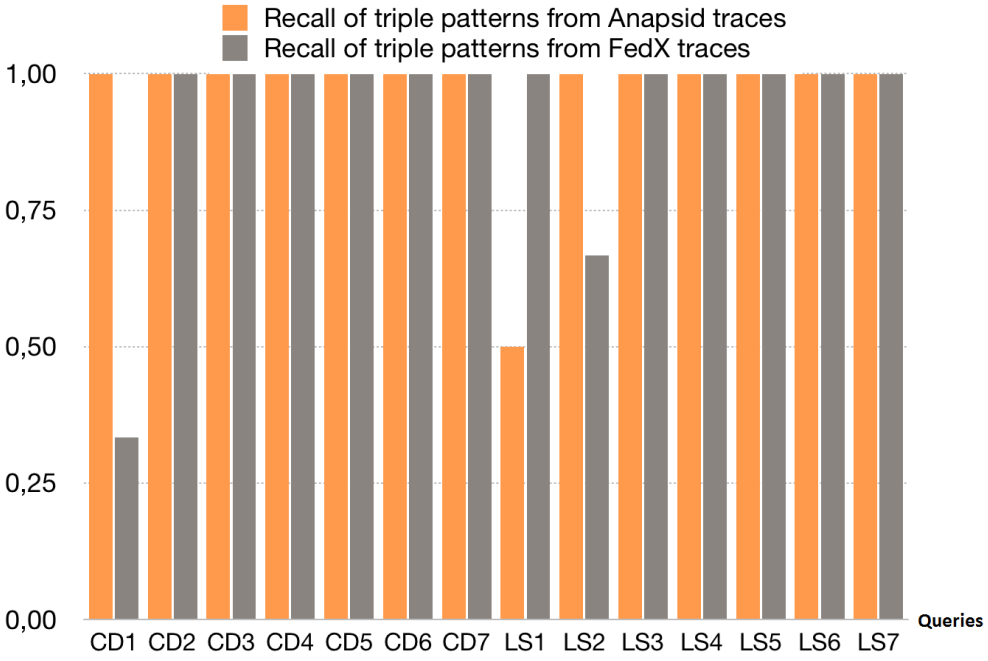


Figure 5.10 Recall of triple patterns for FETA with traces of CD and LS queries, produced with Anapsid or FedX and executed in isolation over a federation of SPARQL endpoints.

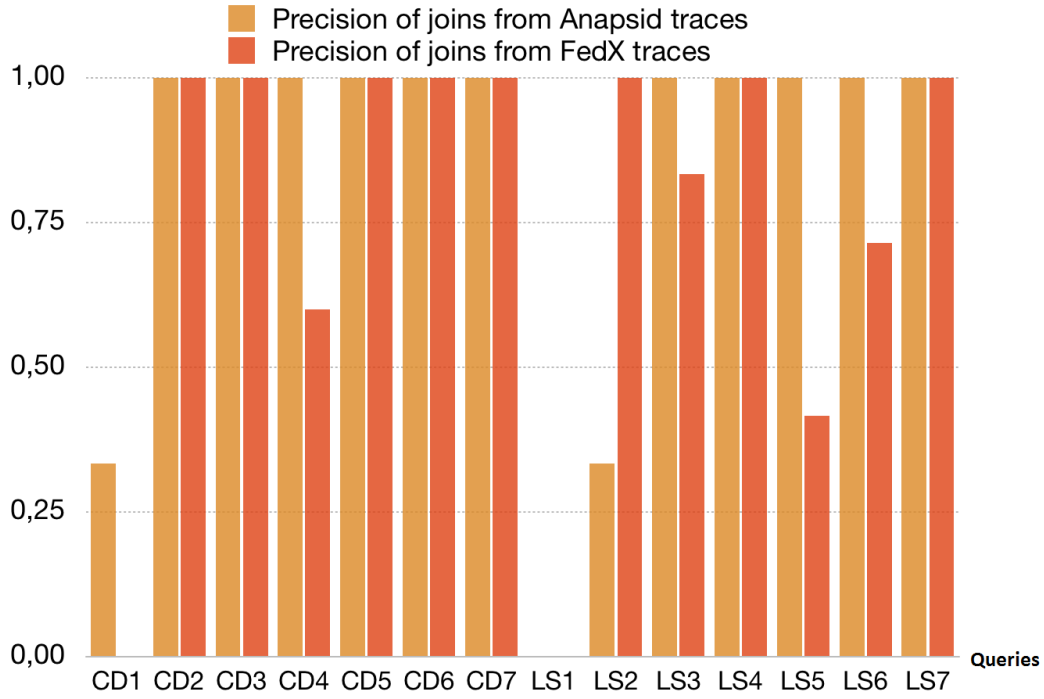


Figure 5.11 Precision of joins for FETA with traces of CD and LS queries, produced with Anapsid or FedX in isolation and executed over a federation of SPARQL endpoints.

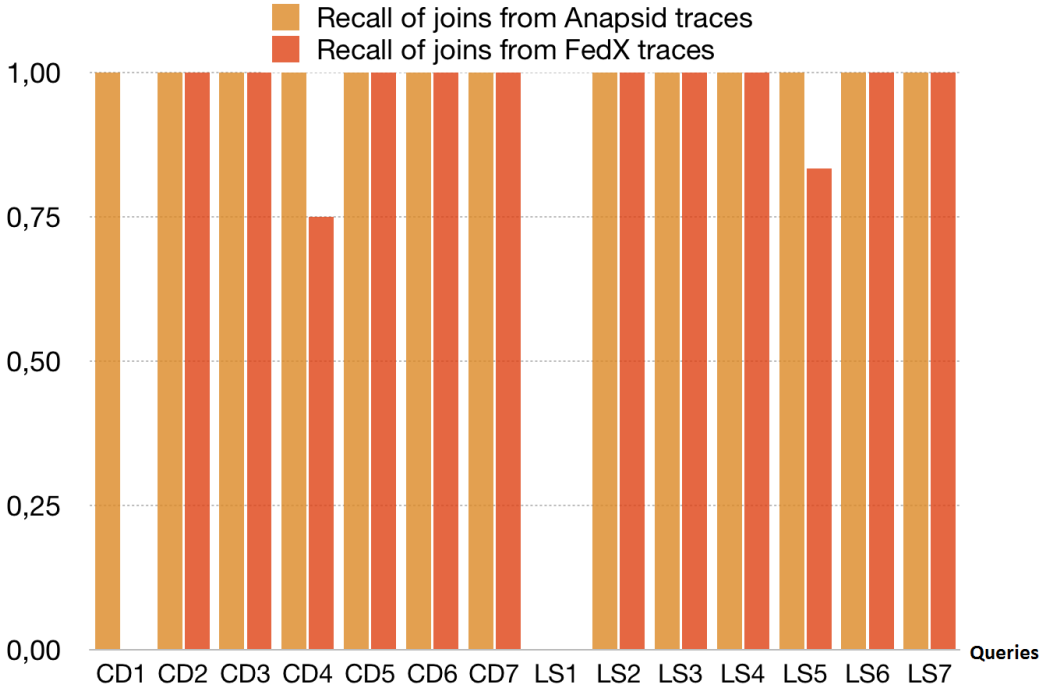


Figure 5.12 Recall of joins for FETA with traces of CD and LS queries, produced with Anapsid or FedX and executed in isolation over a federation of SPARQL endpoints.

5.3.3 Does FETA resist to concurrency?

We executed concurrently all queries of the same collection i.e., Cross Domain and Life Science, over the federations of SPARQL endpoints presented in the beginning of this section. For each query set, we evaluated how FETA $(E(FQ_1)) [\ldots [(FETA(E(FQ_n))$ $FETA(E(FQ_1 \ k \ldots k \ Q_n))$ in terms of recall and precision of joins for different gap values. gap varies from 1% to 100% of the log duration. Each query set was shuffled 4 times and we calculated the average of FETA's results by gap[18].

Figures 5.13 and 5.14 show the average of precision and recall of concurrently executed queries of the CD and LS collections, over a federation of SPARQL endpoints. Figures 5.15 and 5.16 show the recall of 4 mixes for a set of non similar queries executed with Anapsid and FedX, respectively, over federated logs of SPARQL endpoints. This set of chosen federated queries having distinguishable triple patterns is: CD_3, CD_4, CD_5, CD_6, LS_2 and LS_3.

Concerning gap, according to its value increase we observe that globally precision and recall improve, as shown in Figures 5.13 and 5.14 respectively. Compared to LIFT, we observe that FETA has still good results in precision and recall, even when the gap is small (less than 50%). This is explained, as nested-loop operators of Anapsid and FedX are not fully pipelined like the one implemented by TPF clients, but first they retrieve all mappings of the outer before pushing them into the inner dataset (in blocks to avoid reaching the limit response of SPARQL endpoints). Thus, FETA even with a small gap associates triple patterns that belongs to the same inner operand of a join and do not split them in many blocks of joins like LIFT. For more details see page 35 in Chapter 2.

Concerning recall, FETA like LIFT is moderately impacted by concurrency, as shown in Figure 5.13. Indeed, FETA favours recall by producing all possible joins in the nested-loop detection. In general, FETA results on recall for FedX and Anapsid traces are similar. On the other hand, recall for LS is better than recall for CD traces. This happens because for traces of LS queries, FETA generate lots of symmetric hash joins including the good ones. Finally, concerning non-similar queries, recall of joins for Anapsid and FedX traces is even better.

Concerning precision, FETA is more impacted by concurrency and even more than LIFT, as shown in Figure 5.14. When executed queries have triple patterns that are semantically or syntactically similar, then FETA generates many false joins that impact precision. This is explained from the fact that queries of Cross Domain and Life Science are very similar, thus FETA detects inclusion relations between mappings of triple patterns of different queries during nestedLoopDetection. On the other hand, concerning the collection of non-similar queries, presented above, we get 100% of recall with a gap of 50% from traces of both query engines, as shown Figures in 5.15 and 5.16 respectively.

[18] Note that as we vary the gap between two subqueries from 1 to 16 seconds, the duration of each shuffled log we produce diverges from some seconds to one hour and a half.

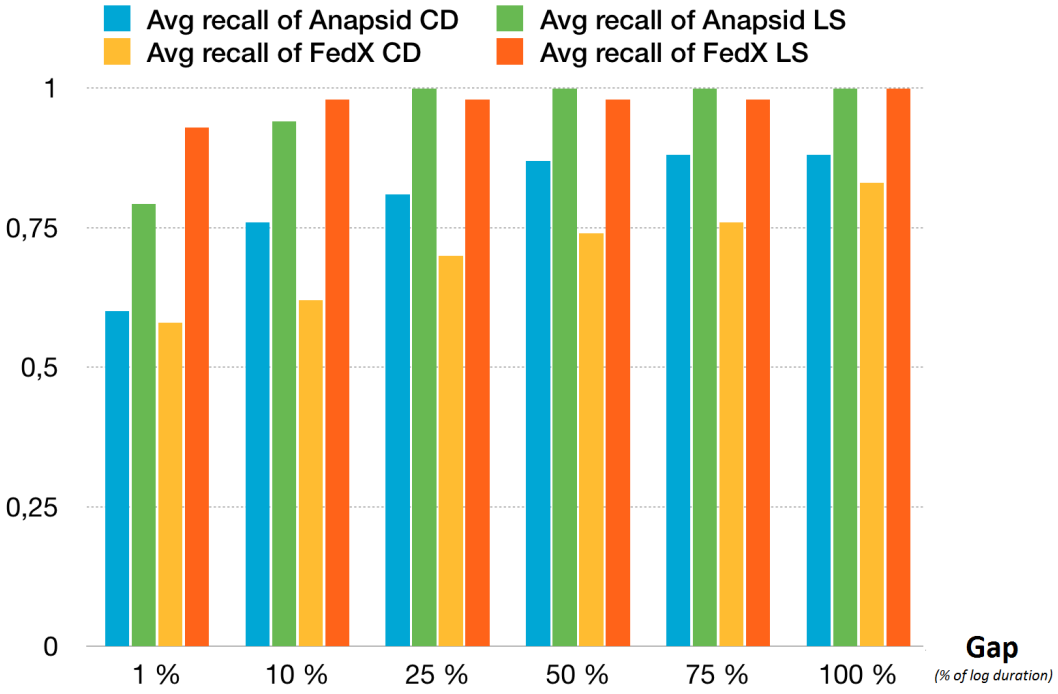


Figure 5.13 Recall (average) of joins per gap for FETA with traces of CD and LS queries, produced with Anapsid or FedX and executed in concurrence over a federation of SPARQL endpoints.

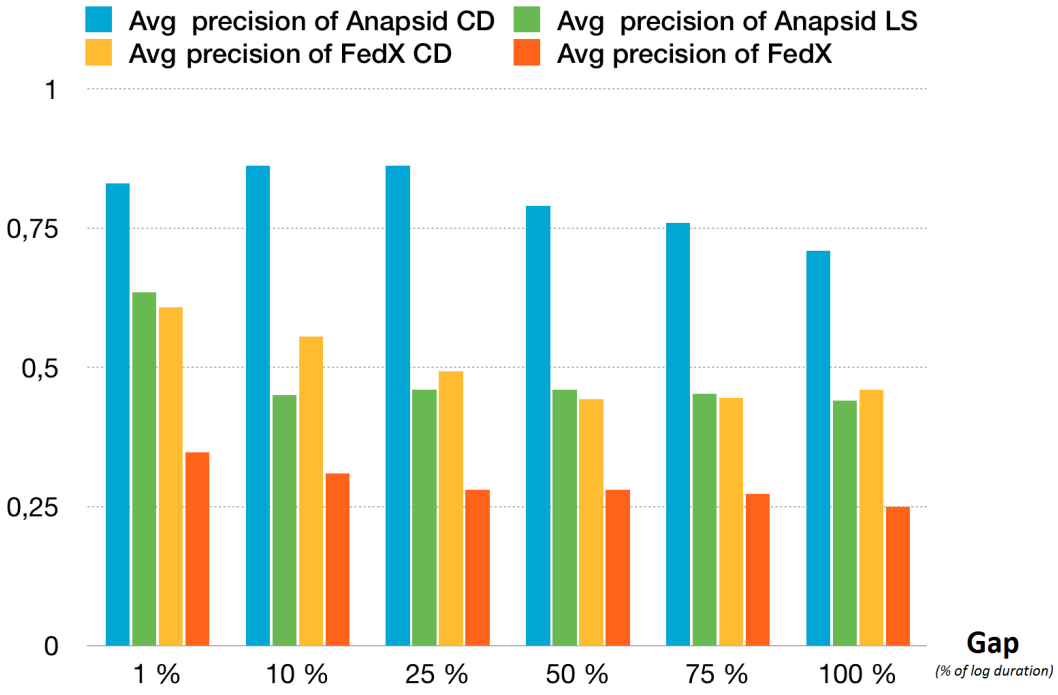


Figure 5.14 Precision (average) of joins per gap for FETA with traces of CD and LS queries, produced with Anapsid or FedX and executed in concurrence over a federation of SPARQL endpoints.

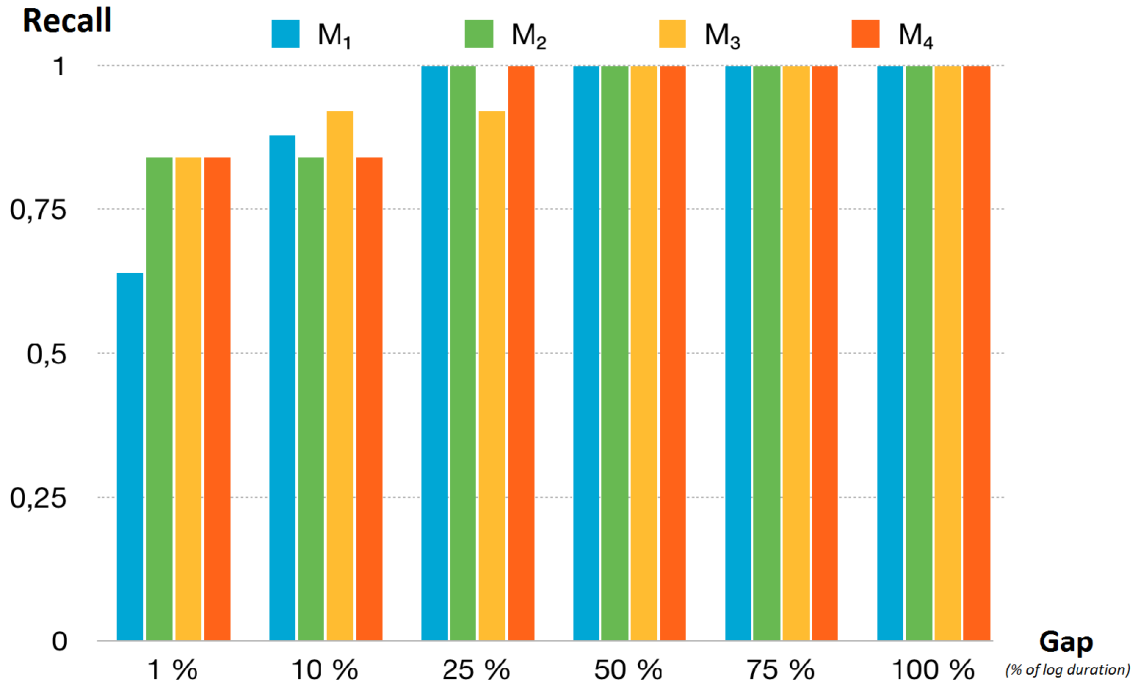


Figure 5.15 Recall of joins per gap and per mix for FETA with traces of CD and LS selective queries, produced with Anapsid and executed in concurrence over a federation of SPARQL endpoints.

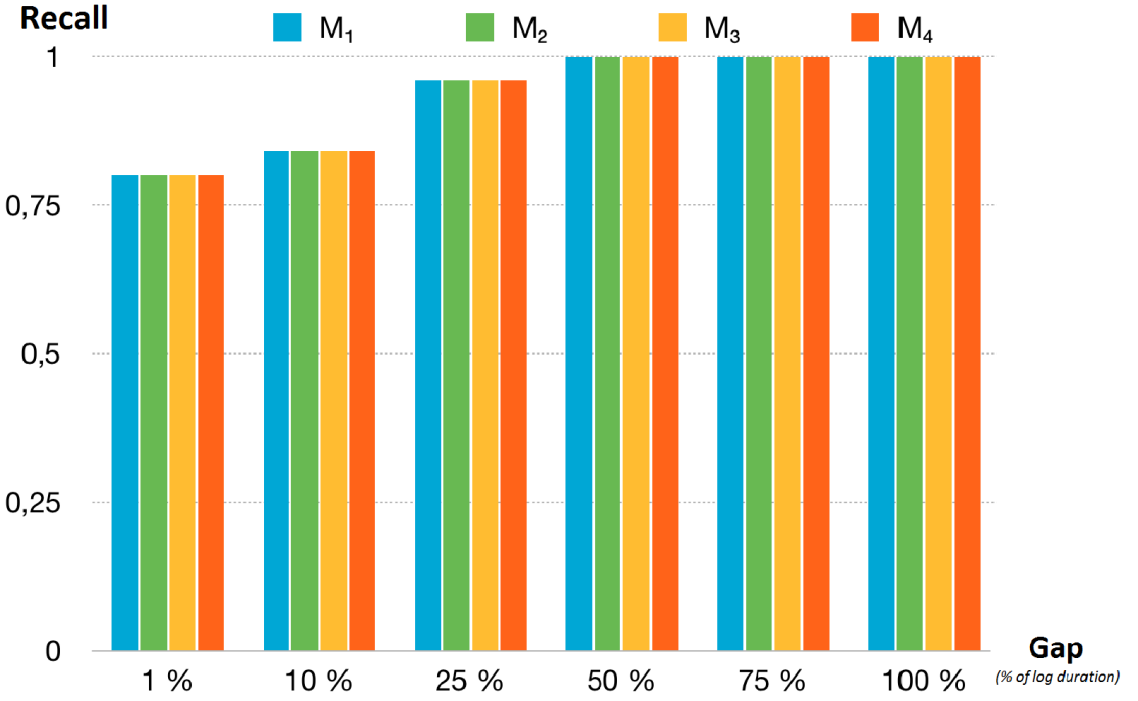


Figure 5.16 Recall of joins per gap and per mix for FETA with traces of CD and LS selective queries, produced with FedX and executed in concurrence over a federation of SPARQL endpoints.

To summarize, we presented FETA, a BGP reversing approach that aims to infer BGPs of queries executed over federations of SPARQL endpoints. We provided with experiments that illustrate FETA's good recall and precision. Obtained results depend not only on the similarity of concurrently executed queries but also execution parameters of FETA. Compared to LIFT, FETA is less efficient in recall and precision for both queries executed in isolation or in concurrence. This is explained from the fact that (i) the symmetricHashDetection employed by FETA generates a lot of false positives, and (ii) FETA's queries are more similar to each other than those used in LIFT. lat

6

Conclusion and perspectives

Contents

In this chapter, section 6.1 presents the conclusions. Section 6.2 describes the perspectives.

6.1 Conclusion

In this thesis, we aimed to infer the general form of SPARQL queries executed over the Linked Data, that is to infer the set of joined triple patterns of these queries. In particular, we focused on the following research question:

How to infer Basic Graph Patterns (BGPs) of SPARQL queries executed by data consumers from logs of servers hosted by data providers?

Answering this question allows data providers to know how their data are used. The knowledge of how data are used is a valuable asset that may be exploited individually by each data provider or as a group, for a diversity of purposes: ensure usage control, optimize the cost of provided services (i.e., access to their Linked Data), justify return on investment, improve their users' experience or even create business models to discover usage trends over the Semantic Web.

Concerning this research question, we proposed four contributions:

First, we formally defined the scientific problem of BGP reversing and the property of resistance to concurrency of multiple queries executed at the same time. We addressed this problem on query processing over both (a) single or federations of TPF servers, and (b) federations of SPARQL endpoints.

Second, we analyzed how sequential Data Mining algorithms can be used to tackle this problem. Frequent episodes detected by MINEPI on raw logs of queries, do not correspond to BGPs of SPARQL queries. This can be improved in terms of joins between triple patterns, with specific pre and post-processing. Even so, this approach is not able to resist to concurrency, regarding both precision and recall of joins.

Third, we proposed LIFT. LIFT takes as input the logs of single triple pattern queries from single or federations of Triple Pattern Fragment (TPF) servers, and extracts a set of BGPs to which these logs correspond to. LIFT groups triple patterns that seems to be part of the same outer or inner operand of a join and subsequently detects nested-loops between these triple patterns. Experimental results reported that LIFT is able to extract BGPs with good recall and precision. However, deducing BGPs with LIFT is challenging in presence of concurrence.

Fourth, we proposed FETA. FETA takes as input the logs of queries from federations of SPARQL endpoints, and extracts a set of BGPs to which these logs correspond to. Compared to LIFT, FETA does not need to process logs of single SPARQL endpoints, as they are already aware of the single source queries they process. FETA groups triple patterns that seems to be part of the same outer or inner operand of a join and subsequently detects nested-loops and symmetric hash joins between these triple patterns. FETA deals with different optimizations techniques employed by query engines to push mappings from a triple pattern into another during nested-loops. Like LIFT, experimental results reported that FETA is able to extract BGPs

with good recall and precision. However, deducing BGPs with HETA, like LIFT, is challenging in presence of concurrence.

In this thesis we introduced the BGP reversing problem and proposed an approach to solve it in two different contexts, using either logs of TPF servers or logs of federations of SPARQL endpoints. Our solutions aimed to extract BGPs of user queries, by processing logs off-line and privileging recall in the detriment of precision. In the next section we present some perspectives related to these choices.

6.2 Perspectives

This thesis opens the following perspectives:

First, we can extract BGPs in real-time. Currently, we do off-line analysis.

Second, we can handle false-positives due to concurrency, with post-processing based on occurrences of BGPs extractions. As concurrently executed queries are not mixed systematically, we can prune detected false joins between their triple patterns.

Third, we can propose new strategies to link subqueries in the log. Different strategies offer different trade-offs between precision and recall.

In order to explain our proposed perspectives, we use the following example. Consider queries Q_A = SELECT ?x ?y WHERE { ?x p1 o1 . ?x p2 ?y . ?y p3 ?z } and Q_F = SELECT ?x WHERE { ?x p3 ?z . ?y p4 ?z } on page 56 in Chapter 3. Table 6.1 corresponds to the log with shuffled execution traces of Q_A and Q_F[1]. Queries Q_A and Q_F are evaluated by joining triple patterns through nested-loops.

LD provider	IP	Time	Requested tp	Answer
p_A	ip_1	0	?x p1 o1	{?x → {s1, s2}}
p_A	ip_1	10	s1 p2 ?y	{?y → {o3}}
p_A	ip_1	20	?x p3 ?z	{?x → {s3, s4}; ?z → {o3, o4}}
p_A	ip_1	40	s2 p2 ?y	{?y → {o4}}
p_A	ip_1	50	o3 p3 ?z	{?z → {o5}}
p_A	ip_1	60	?y p4 o3	{?y → {s5}}
p_A	ip_1	80	?y p4 o4	{?y → {s6}}
p_A	ip_1	90	o4 p3 ?z	{?z → {o6}}

Table 6.1 Query log corresponding to execution of Q_A and Q_F, produced by data consumer with ip_1 IP Address and executed on p_A data provider. Traces in red color correspond to query Q_A while traces in green correspond to query Q_F.

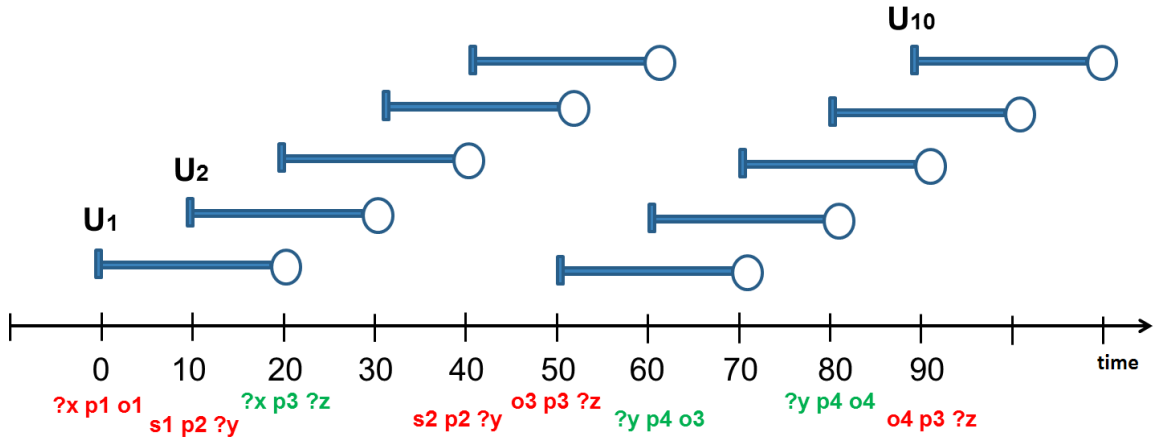


Figure 6.1 Sliding windows of length = 20 seconds with an incremental approach to extract BGPs of executed queries in the log [0; 110[. Traces in red color correspond to query Q_A while traces in green correspond to query Q_F.

6.2.1 Real-time extraction of BGPs

As a first perspective we aim to infer BGPs of user queries, this time incrementally over server logs that are dynamically created [26]. The idea, is to associate progressively joinable triple patterns that appear in consecutive windows of user-defined size. The challenge, like for WINEPI (cf. Chapter 3 on page 50), is to manage intermediate deduced BGPs. Once joins between triple patterns are inferred, it is not obvious to finalize deduced BGPs and free the in-memory allocated to host their mappings. That is, deduced BGPs (and their mappings) may be associated to triple patterns of following windows because: (i) mappings can be bind several times (e.g., star queries), (ii) apparition of triple patterns is not related always to joins e.g., during the execution of a specific query, a block of traces of other concurrently executed queries could interfere. Experiments will reveal to which deduction parameters, the inferred BGPs by this approach are stabilised.

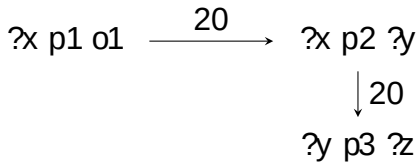
Suppose that we extract BGPs from the log of Table 6.1, using an incremental BGP reversing approach over 10 sliding windows, each with a size of 20 seconds, as we see in Figure 6.1. For instance, we observe that mappings of ?x p1 o1 are injected in the second and fifth window, into " s1 p2 ?y" and "s2 p2 ?y". As described above, the challenge is to considered that a deduced BGPs is final. If consider that BGP_1 = { ?x p1 o1 : ?x p2 ?y } is final after the third window and remove its mappings from the memory, we will not detect the join in Q_A between ?x p2 ?y and ?y p3 ?z, in the sixth and tenth window respectively.

6.2.2 Handling false-positives due to concurrency, with post-processing

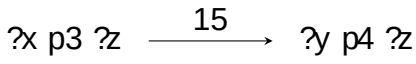
Handling concurrency is a very difficult problem. When similar queries run concurrently in the same time, it is nearly impossible to extract correctly BPGs. However, it is unlikely that similar queries always run concurrently. Consequently, if we run LIFT or FETA on

[1] Note that for simplicity we supposed that timestamps are integers.

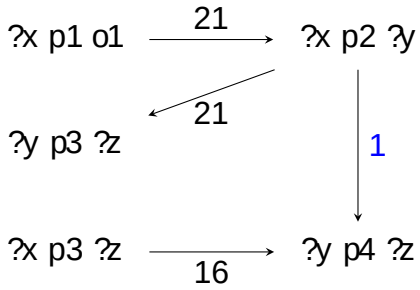
long periods, false positives due to concurrency will get less occurrences than correct deduced BPGs. Running algorithms that exploit occurrences of BGPs is now possible thanks to our BGP reversing approaches because, in some way, they transform a non-transaction log (timestamped triple pattern queries for LIFT or FETA) into a transactional log (timestamped list of linked triple pattern queries). With a transactional log of deduced BGPs it is possible to find their occurrences and subsequently the occurrences of their joins. Figure 6.2 represents the extracted BGPs from log of Table 6.1 using LIFT or FETA for logs of multiple hours, which edges are annotated with the occurrences of their joins. The first deduced BGP corresponds to Q_A, the second to Q_F and the third to the mixed BGP corresponding to Q_A and Q_F. In this figure, edges between triple patterns are labeled with the occurrence of their joins (in form of nested-loops).



(a) Deduced BGP for traces of Q_A executed in isolation.



(b) Deduced BGP for traces of Q_F executed in isolation.



(c) Deduced BGP for traces of Q_A and Q_F executed in concurrence.

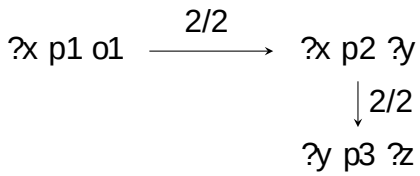
Figure 6.2 Set of deduced BGPs with LIFT when applied on logs of multiple hours, where each edge is annotated with the occurrences of the join of two triple patterns. The less frequent join is presented in blue.

One approach to handle false positives, is by associating deduced triple patterns on the number of their occurrences. Computing frequent association of BPGs can be done with Apriori-based algorithms[4]. Hence, given 10 hours of a log, we aim to see if the precision of frequent BGPs detected over 10 hours is better than precision of BGPs detected on 1 hour. For the example of Figure 6.2 using a threshold equal to 5, we can prune the false join between triple patterns ?x p2 ?y and ?y p4 ?z, from Q_A and Q_F in the third BGP.

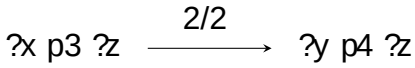
Another approach to handle false positives, is by grouping deduced triple patterns on their confidence represented by their occurrences. Extracting subgraphs of BGPs

in presence of uncertainty can be done with the k-core approach [49]. That is, given a deduced BGP which is uncertain because its triple patterns are joined with different levels of confidence, we can extract a set of subgraphs with the same confidence. For the example of Figure 6.2 we can extract three subgraphs: {?x p1 o1 . ?x p2 ?y . ?y p3 ?z}, {?x p3 ?z . ?y p4 ?z} and {?x p2 ?y . ?y p4 ?z} with confidences 2/3, 1/6 and 1/2 respectively.

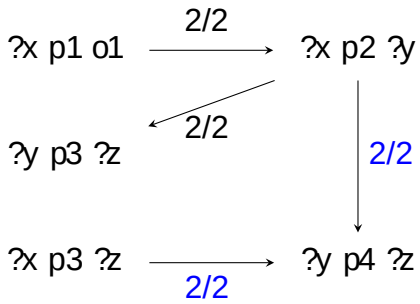
6.2.3 Other strategies to link subqueries



(a) Deduced BGP for traces of Q_A executed in isolation.



(b) Deduced BGP for traces of Q_F executed in isolation.



(c) Deduced BGP for traces of Q_A and Q_F executed in concurrence.

Figure 6.3 Set of deduced BGPs with LIFT when applied on a log, where each edge is annotated with the coverage of the mappings of two triple patterns. The two alternative options of coverage of the injected mappings into {?y p4 ?z}, are presented in blue.

There is a diversity of strategies linking triple patterns, each with a different trade-off between precision and recall. In LIFT and FETA, we made the arbitrary choice to link all triple patterns which mappings intersect. With this choice, our approaches were able to favour recall in the detriment of precision. Alternatively, we could employ strategies that take into account the quality of the matching between mappings of triple patterns. This can be done, using the *set-covering* strategy [20] where triple patterns are grouped based on the *coverage* of their mappings and each triple pattern can participate only in one set. The set-covering approach favours precision of joins as in general the number of deduced joins is minimized, when at the same time it eventually detriments recall. However, in

presence of concurrence, performance in both recall and precision of joins may be a ected. We illustrate this on extracted BGPs from log of Table 6.1.

Figure 6.3 presents extracted BGPs, where each edge is annotated with the coverage of mappings of two triple patterns. When LIFT or FETA is applied to extract BGPs from the log of Table 6.1, recall in terms of joins is 3/3=1 while precision is 3/4=0.75, as we see in the third BGP of Figure 6.3. If the set-covering approach was employed to extract BGPs from the log of Table 6.1, we would have two possible combinations as ?y p4 ?z can be assigned to two di erent sets: (a) f ?x p1 o1 : ?x p2 ?y : ?y p3 ?z g and f ?x p3 ?z : ?y p4 ?z g, or, (b) f ?x p1 o1 : ?x p2 ?y : ?x p3 ?z : ?y p4 ?z g and f ?y p3 ?z g. In terms of both recall and precision of joins, the set-covering approach detects 3/3=1 joins in the former set of deduced BGPs while 2/3=0.66 in the latter set of deduced BGPs. Compared to LIFT or FETA, performance in precision is better while recall remains the same for the former set of deduced BGPs. However, both recall and precision are inferior for the latter set of deduced BGPs, compared to LIFT or FETA.

Bibliography

[1] M. Acosta and M. Vidal. The ANAPSID Evolution: An adaptive SPARQL query engine. 2014. 33, 37, 85

[2] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I, pages 18 34, 2011. 17, 33, 34, 37, 85

[3] R. Agrawal, T. Imielinski, and A. N. Swami. Mining Association Rules between Sets of Items in Large Databases. In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993, pages 207 216, 1993. 19

[4] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, pages 487 499, 1994. 47, 111

[5] R. Angles and C. Gutierrez. The Expressive Power of SPARQL. In The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings, pages 114 129, 2008. 24

[6] C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II, pages 277 293, 2013. 18

[7] C. B. Aranda, A. Polleres, and J. Umbrich. Strategies for Executing Federated Queries in SPARQL1.1. In The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II , pages 390 405, 2014. 24, 26

[8] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An Empirical Study of Real-World SPARQL Queries. CoRR, abs/1103.5043, 2011. 68, 88

[9] C. Basca and A. Bernstein. Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In Proceedings of the ISWC 2010 Posters & Demonstrations Track: Collected Abstracts, Shanghai, China, November 9, 2010, 2010. 17, 34

[10] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. Int. J. Semantic Web Inf. Syst., 5(3):1 22, 2009. 11, 15, 16

[11] Dijkstra, Edsger W. A note on two problems in connexion with graphs. Numerische mathematik, 1(1):269 271, 1959.

[12] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011, 2011. 17, 34

[13] G. Graefe. Query Evaluation Techniques for Large Databases. ACM Comput. Surv., 25(2):73 170, 1993. 26, 31

[14] P. Haase, T. Mathäÿ, and M. Ziller. An evaluation of approaches to federated query processing over linked data. In Proceedings the 6th International Conference on Semantic Systems, I-SEMANTICS 2010, Graz, Austria, September 1-3, 2010, 2010. 33

[15] J. Han, M. Kamber, and J. Pei. Data Mining: Concepts and Techniques. Elsevier, 2011. 19

[16] O. Hartig. Querying a Web of Linked Data - Foundations and Query Execution, volume 24 of Studies on the Semantic Web. IOS Press, 2016. 16

[17] O. Hartig, C. Bizer, and J. C. Freytag. Executing SPARQL Queries over the Web of Linked Data. In The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings, pages 293 309, 2009. 18

[18] T. Heath and C. Bizer. Linked Data: Evolving the Web into a Global Data Space. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011. 15

[19] Heiko Betz and Francis Gropengieÿer and Katja Hose and Kai-Uwe Sattler. Learning from the History of Distributed Query Processing - A Heretic View on Linked Data Management. In Proceedings of the Third International Workshop on Consuming Linked Data, COLD 2012, Boston, MA, USA, November 12, 2012, 2012. 33

[20] D. S. Johnson. Approximation Algorithms for Combinatorial Problems. J. Comput. Syst. Sci, 9(3):256 278, 1974. 112

[21] M. Klemettinen and P. Moenand. Course on Data Mining (581550-4): Episodes and episode rules. 50

[22] D. Kossmann. The State of the art in distributed query processing. ACM Comput. Surv., 32(4):422 469, 2000. 33

[23] G. Ladwig and T. Tran. Linked Data Query Processing Strategies. In The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I, pages 453 469, 2010. 18

[24] M. Lanthaler and C. Guetl. Hydra: A Vocabulary for Hypermedia-Driven Web APIs. In Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013, 2013. 29

[25] H. Mannila and H. Toivonen. Discovering Generalized Episodes Using Minimal Occurrences. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA, pages 146 151, 1996. 53

[26] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering Frequent Episodes in Sequences. In Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada, August 20-21, 1995, pages 210 215, 1995. 50, 53, 110

[27] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of Frequent Episodes in Event Sequences. Data Min. Knowl. Discov., 1(3):259 289, 1997. 50

[28] L.-R. Markus, A. Saud, B. Bettina, and H. Laura. USEWOD Research Dataset., 2016. http://dx.doi.org/10.5258/SOTON/385344. 67, 73, 74, 77

[29] D. P. Miranker, R. Depena, H. Jung, J. Sequeda, and C. Reyna. Diamond Debugger Demo: Rete-Based Processing of Linked Data. In Proceedings of the RuleML2012@ECAI Challenge, at the 6th International Symposium on Rules, Montpellier, France, August 27th-29th, 2012, 2012. 18

[30] P. Mishra and M. H. Eich. Join Processing in Relational Databases. ACM Comput. Surv., 24(1):63 113, 1992. 26

[31] G. Montoya, M. Vidal, and M. Acosta. A Heuristic-Based Approach for Planning Federated SPARQL Queries. In Proceedings of the Third International Workshop on Consuming Linked Data, COLD 2012, Boston, MA, USA, November 12, 2012, 2012. 37

[32] C. Mooney and J. F. Roddick. Sequential pattern mining - approaches and algorithms. ACM Comput. Surv., 45(2):19, 2013. 19, 45, 47

[33] Moore, Edward F. The shortest path through a maze. Bell Telephone System., 1959.

[34] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo and Josiane Xavier Parreira and Helena F. Deus and Manfred Hauswirth. DAW: Duplicate-AWare Federated Query Processing over the Web of Data. In The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I, pages 574 590, 2013. 34

[35] G. Nassopoulos, P. Serrano-Alvarado, P. Molli, and E. Desmontils. FETA: Federated QuEry TrAcking for Linked Data. In International Conference on Database and Expert Systems Applications-DEXA, pages 303 312, 2016. 30, 94

[36] A. N. Ngomo and M. Saleem. Federated Query Processing: Challenges and Opportunities. In Proceedings of the 3rd International Workshop on Dataset PROFIling and

fEderated Search for Linked Data (PROFILES '16) co-located with the 13th ESWC 2016 Conference, Anissaras, Greece, May 30, 2016, 2016. 33, 34

[37] M. T. Özsu and P. Valduriez. Principles of Distributed Database Systems, Third Edition. Springer, 2011. 11, 33, 34

[38] S. K. Pal, V. Talwar, and P. Mitra. Web mining in soft computing framework: relevance, state of the art and future directions. IEEE Trans. Neural Networks, 13(5):1163 1177, 2002. 45

[39] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings, pages 30 43, 2006. 24

[40] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, Proceedings, pages 524 538, 2008. 33, 34

[41] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo. LSQ: The Linked SPARQL Queries Dataset. In The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II , pages 261 269, 2015. 98

[42] M. Saleem and A. N. Ngomo. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings, pages 176 191, 2014. 34

[43] M. Saleem, S. S. Padmanabhuni, A. N. Ngomo, A. Iqbal, J. S. Almeida, S. Decker, and H. F. Deus. TopFed: TCGA Tailored Federated Query Processing and Linking to LOD. J. Biomedical Semantics, 5:47, 2014. 34

[44] 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. 77

[45] F. Schmedding. Incremental SPARQL Evaluation for Query Answering on Linked Data. In Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011, 2011. 18

[46] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, Proceedings, Part I, pages 585 600, 2011. 53, 85, 87, 98

[47] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings, pages 4 33, 2010. 24, 33, 90