HAL open science

Dealing with Plethoric Answers of SPARQL Queries

Louise Parkin, Brice Chardin, Stéphane Jean, Allel Hadjali, Mickaël Baron

▶ **To cite this version:**

Louise Parkin, Brice Chardin, Stéphane Jean, Allel Hadjali, Mickaël Baron. Dealing with Plethoric Answers of SPARQL Queries. Proceedings of the 32nd International Conference on Database and Expert Systems Applications, Sep 2021, Linz, Austria. pp.292-304, 10.1007/978-3-030-86472-9_27 . hal-03356184

HAL Id: hal-03356184
https://hal.science/hal-03356184

Submitted on 27 Sep 2021

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

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

Dealing with Plethoric Answers of SPARQL Queries

Louise Parkin[1], Brice Chardin[1], Stéphane Jean[2], Allel Hadjali[1], Mickaël Baron[1]

[1] ISAE-ENSMA, LIAS
[2] Université de Poitiers, LIAS

Abstract. When querying Knowledge Bases (KBs), users are faced with large sets of data, often without knowing their underlying structures. It follows that users may make mistakes when formulating their queries, therefore receiving an unhelpful response. In this paper, we address the plethoric answers problem, the situation where a query produces significantly more results than the user was expecting. We deal with this problem by identifying the parts of the failing query, called *Minimal Failure Inducing Subqueries* (MFIS), that cause plethoric answers. As long as the query contains an MFIS, it will fail to reach a sufficiently low amount of answers. Thanks to these MFIS, interactive and automatic approaches can be set up to help the user reformulate their query. The dual notion of MFIS, *maXimal Succeeding Subqueries* (XSS), is also useful. They are queries with the most parts of the original query that return non plethoric answers. Our goal is to compute MFIS and XSS efficiently, so that they may be used to solve the plethoric answers problem. We propose two algorithms that leverage query and data properties to compute MFIS and XSS. We show experimentally that our algorithms clearly outperform a baseline method on generated queries as well as real user-submitted queries.

1 Introduction

A Knowledge Base (KB) is a collection of entities and facts about them. With the development of the Semantic Web, numerous KBs have been created in academic and industrial areas. A well known example of a KB is *DBpedia* [1]. These KBs store information as RDF triples (subject, predicate and object) and are queried with the *SPARQL* language [2] using triple patterns which are triples containing variables. KBs typically store billions of facts and are often structured using an ontological schema and rules, such as those provided by *RDFS* [3].

A new user querying a KB is often unfamiliar with the KB's structure and the data within it. Thus, *mistakes* or *misconceptions* can manifest in queries, and cause unexpected or unsatisfactory answers. Mistakes refer to the user incorrectly writing their query, for example creating an unwanted Cartesian product by omitting a triple pattern, or misspelling a term. Misconceptions refer to the difference between a user's view of a KB, and its reality [4]. For example if in a hospital KB, the property *treats* can only link a *Doctor* to a *Patient*, and a user

writes a query based on the patients that a *Nurse treats*, they will be frustrated to receive no answers. Alternatively, a user may believe a property *birthPlace* only gives a person's town of birth whereas in the KB *birthPlace* is used for the country, county, town, and address of birth. A query involving *birthPlace* may overwhelm the user by producing four times as many answers as expected. The issue of unexpected answers is a challenge to database usability [5]. There are five unexpected answers problems, each associated with a why-question: no answers (*why-empty*), too few answers (*why-so-few*), too many answers (*why-so-many*), missing expected answers (*why-not*), and unwanted answers (*why-so*). We focus on the the too many answers problem, also called the plethoric answers problem, where users struggle to extract useful information from an overwhelming result. A query's results are said to be plethoric when there are more than a threshold K.

Most state of the art methods to deal with plethoric answers rely on ordering results and selecting an adequately sized subset of answers to be returned to the user. These methods are called *top-K* methods. Solutions vary by the way results are ordered, and the extent of user involvement. They guarantee the number of answers will be at most K. Yet, if a query is based on a misconception on the user's part no ordering strategy will solve the underlying problem. In this paper, we claim that the first step to solve the plethoric answers problem should be to understand why a query produces plethoric answers. Our failure causes can be directly provided to users in an effort to educate them in formulating their queries. They can also be used as a basis for automatic or interactive query rewriting, in order to avoid suggesting queries that are known to fail, and thus accelerate the process.

Drawing on work on the empty-answers problem in KBs [6], we propose the basis of a cooperative method to deal with the plethoric answers problem. We provide two notions that can help with query rewriting: the smallest subqueries that cause plethoric answers (MFIS) and the largest subqueries that do not produce plethoric answers (XSS). We propose an algorithm to compute MFIS and XSS, leveraging query properties to avoid executing some subqueries. Improvements based on a data property, i.e. predicate cardinalities, are also discussed. The performance of our algorithms is assessed through experimental evaluation, using queries generated for the *WatDiv* synthetic dataset [7], and user-submitted DBpedia queries from the *Linked SPARQL Queries Dataset* logs [8].

This paper is organized as follows. Section 2 gives a motivating example that will illustrate our proposal throughout the paper. Section 3 details related work. We provide preliminary notions in Section 4. Section 5 presents our approaches to calculate MFIS and XSS. Section 6 describes the experimental evaluation of our algorithms. We conclude and introduce future work in Section 7.

2 Motivating Example

We consider a simplified hospital KB (figure 1a), and a user wanting information on doctors, nurses and patients. Figure 1b shows an example of a user query defined as a conjunction of triple patterns $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4 \wedge t_5$ (or $t_1t_2t_3t_4t_5$

subject	predicate	object
d_1	experience	25
d_1	supervises	n_3
d_1	supervises	n_2
d_1	treats	p_1
d_1	treats	p_2
d_2	experience	14
d_2	supervises	n_1
d_2	treats	p_3
n_1	type	SurgicalNurse
n_1	providesCare	p_3
n_2	type	ERNurse
n_2	providesCare	p_2
n_2	providesCare	p_3
n_3	type	ERNurse
n_3	providesCare	p_1
n_3	providesCare	p_2

(a) Knowledge base D

```
Q : SELECT * WHERE {
    ?d treats ?p .            #t1
    ?d experience ?e .        #t2
    ?d supervises ?n .        #t3
    ?n providesCare ?pt .     #t4
    ?n type ERNurse }         #t5
```

(b) Query $Q = t_1 t_2 t_3 t_4 t_5$

?d	?p	?e	?n	?pt
d1	p_1	25	n3	p_1
d1	p_2	25	n3	p_1
d1	p_1	25	n3	p_2
d1	p_2	25	n3	p_2
d1	p_1	25	n2	p_2
d1	p_2	25	n2	p_2
d1	p_1	25	n2	p_3
d1	p_2	25	n2	p_3

(c) Results of Q on D

Fig. 1. A Knowledge Base, a SPARQL query and its results

in short). When executing Q on our dummy KB the query produces 8 results (figure 1c). On a real KB containing hundreds of doctors and patients, this query would produce thousands of results, which would not be manageable for the user. To illustrate our method, we set a small threshold of plethoric answers $K = 3$, so Q is considered failing as its number of results exceeds this threshold.

A top-K method could be used to reduce the number of results, such as ordering patients alphabetically, but this would only return information about a few patients. Our approach will focus on explaining plethoric answers based on failure causes, so the query can be modified to return fewer answers. In our example, there are three failure causes: $t_1 t_3$, $t_1 t_5$ and t_4. As every subquery of Q containing one of these subqueries fails, each of the failure causes must be resolved in order to reach the desired number of answers. Each failure cause can be interpreted to explain plethoric answers, which will in turn suggest possible corrections.

- $t_1 t_3$ and $t_1 t_5$ indicate that asking for both patients and nurses produces plethoric answers. They suggest splitting the query into two parts, one concerning patients, and the other concerning nurses.
- t_4, indicates that nurses are assigned to a plethoric number of patients. It suggests removing t_4 from the query, or adding some additional conditions.

We also provide the succeeding queries which are not subqueries of any other succeeding query: $t_2 t_3 t_5$ and $t_1 t_2$. They partially meet the user's requirement and can be used as alternative queries, knowing that they have at most K results.

The interpretation of failure causes, and their presentation to a user along with alternative queries in an interactive query refining system is planned for future work, and is not further studied in this paper. We focus here on the efficient computation of the failure causes, for use in a query rewriting system.

3 Related Work

Existing approaches dealing with the plethoric answers problem can be divided into two categories: those focusing on data and those focusing on queries.

Data-oriented methods suppose that the query submitted by the user is correct, and present results in an organised fashion, so that certain information is easily visible. Top-K methods are the most widely used type of data-oriented methods. They order results based on user preferences and return only the top K answers. Ilyas et al. present top-K query processing techniques for relational database systems [9]. Other data-oriented strategies have been proposed for cases where user preferences are unknown. Regret-minimization strategies combine features from top-K and skyline methods [10]. They return a set of K answers which maximizes the minimal satisfaction of any user with any preference function. Finally, grouping methods aggregate results into categories, and show the user the common features of each category [11, 12]. If the initial query correctly matches the user's requirement, data-oriented methods can be useful to sort through large result sets. However, if the original query contains an underlying issue, these methods are not appropriate, as they do not attempt to fix the query.

Query-oriented methods modify the user's query so that it returns fewer answers. In the field of fuzzy queries, intensification strategies are used to make patterns present in the user's query more restrictive [13, 14]. Alternatively, new patterns are added to the query [15]. They are chosen based on a measure of correlation between predicates so that they are semantically close to the original query and reduce the number of answers. In the field of knowledge graphs, recent work on the why-not and why-so problems – where an expected answer is missing or an unexpected answer appears in the response – can be extended to the plethoric answers problem [16]. Exact algorithms and heuristics are proposed to refine a user's query. A final approach, which is most similar to ours, considers subqueries of the original query, to find the parts with few enough answers [17]. However, this algorithm does not consider failure causes, and uses no inference rules to avoid exploring parts of the subqueries search space. Query-based solutions are more appropriate to address an underlying issue in the original query. However, as none of the existing approaches study the cause of plethoric answers, the query intensification is done blindly. So patterns causing multiple results may be missed or take several attempts to find.

While failure causes have not previously been considered for the plethoric answers problem, they have been used for other unexpected answers problems. For the why-not problem, a divide-and-conquer approach is used, first studying a query's triple patterns and then its SPARQL operators [18]. A failure cause shows users which triple pattern or operator causes an answer to be absent. For the empty answers problem, Godfrey [19] suggested providing users with failure causes (called MFS) and alternate subqueries (called XSS). MFS have subsequently been used in interactive and automatic query relaxation to accelerate the process, by pruning the search space of queries which necessarily fail [20, 21]. We propose extending the definitions of MFS and XSS to deal with the plethoric answers problem in the context of RDF KBs.

4 Preliminaries and Problem Statement

We describe the formalism and semantics of RDF and SPARQL necessary for this paper. We use the notations and definitions provided by Pérez et al. [22].

4.1 Basic Notions

Data Model We consider three pairwise disjoint infinite sets: I the set of IRIs, B the set of blank nodes, and L the set of literals. We denote by T the union $I \cup B \cup L$. An *RDF triple* is a triple (subject, predicate, object) $\in (I \cup B) \times I \times T$.

RDF Queries Consider V a set of variables disjoint from T. A triple t (subject, predicate, object) $\in (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a *triple pattern*. We denote by $s(t)$, $p(t)$, $o(t)$, and $var(t)$ the subject, predicate, object and variables of t. *RDF queries* are defined as conjunctions of triple patterns $Q = t_1 \cdots t_n$. The variables of a query are $var(Q) = \bigcup var(t_i)$. We define an order on queries using triple pattern inclusion. Given $Q = t_1 \cdots t_n$, $Q' = t_i \cdots t_j$ is a *subquery* of Q, denoted by $Q' \subseteq Q$, iff $\{i, \cdots, j\} \subseteq \{1, \cdots, n\}$. Then Q is a *superquery* of Q'.

Query Evaluation A mapping μ from V to T is a partial function $\mu : V \rightarrow T$. For a triple pattern t, we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* if $\forall x \in dom(\mu_1) \cap dom(\mu_2),\ \mu_1(x) = \mu_2(x)$. The join of two sets of mappings Ω_1 and Ω_2 is: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1,\ \mu_2\ \in \Omega_2 \ \textit{are compatible mappings}\}$. The *evaluation* of a triple pattern t over a KB D, is $[[t]]_D = \{\mu \mid dom(\mu) = var(t) \wedge \mu(t) \in D\}$. The evaluation of a query $Q = t_1 \cdots t_n$ over D is $[[Q]]_D = [[t_1]]_D \bowtie \cdots \bowtie [[t_n]]_D$.

4.2 Notions of MFIS and XSS

In the plethoric answers problem, for a threshold K, a failing subquery of a query Q is a query that returns more than K answers and a succeeding subquery of a query Q is a query that returns at most K answers. We introduce a Boolean property of query failure: $\text{FAIL}_K(Q, D) = |[[Q]]_D| > K$. As the evaluation of the empty query returns one answer mapping no variables, it succeeds (if $K > 0$).

The notion of Minimal Failing Subqueries (MFS) was introduced for the empty answers problem [19]. In that problem, the failure property is monotonic, i.e. if a query fails, its superqueries fail. With this monotony, MFS are the smallest parts of a query that cause failure. In the plethoric answers problem, there is no such monotony. A failing query can have a succeeding superquery: in our example t_2t_5 fails but $t_2t_3t_5$ succeeds. We therefore define two new notions.

Definition 1. *A Failure Inducing Subquery (FIS) of a query Q is one of its failing subqueries whose superqueries all fail.*

Definition 2. *A Minimal Failure Inducing Subquery (MFIS) of a query Q is one of its FIS having no subqueries that are FIS.*

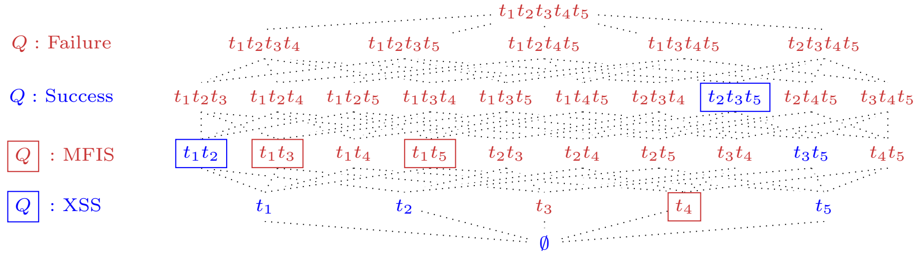


Fig. 2. Lattice of subqueries of $Q = t_1t_2t_3t_4t_5$

If the failure condition is monotonic, MFIS and MFS are equivalent notions. The notion of maXimal Succeeding Subqueries (XSS) was also defined for the empty answers problem. They are the succeeding queries that are most similar to the original query and can be used as alternative queries. The notion of XSS applies to the plethoric answers problem, as it does not require monotony.

Definition 3. *A maXimal Succeeding Subquery (XSS) of a query Q is a succeeding subquery whose superqueries are all FISs.*

Problem Statement We are concerned with efficiently computing the MFIS and XSS for an RDF query Q and a given threshold K in a KB D.

5 Computing MFIS and XSS

We discuss here MFIS and XSS computation. First, a baseline algorithm is presented, then improved versions are introduced by leveraging various properties.

5.1 Baseline

A baseline approach to calculate all MFIS and XSS is to execute every subquery of the original query. Figure 2 shows the lattice of subqueries of Q from our running example. In this first algorithm, which we call Base, the lattice is explored in a *Breadth-First* order, so we start by executing the query with the most triple patterns. For a query with n triple patterns, Base requires $2^n - 1$ query executions (the empty query is not executed), which is time-consuming for queries with many triple patterns. To make the search for MFIS and XSS more efficient, we want to reduce the number of queries to be executed.

5.2 General properties

A first improvement is to avoid executing queries irrelevant to the search for MFIS and XSS, with a property deduced from the definitions of MFIS and XSS.

Property 1. If a subquery Q' succeeds, and $Q'' \subset Q'$, then Q'' is neither an MFIS nor an XSS.

```
Var/Full(Q, D, K)
    inputs : A failing query Q = t_1 ∧ ... ∧ t_n, a KB D, a threshold K
    outputs: MFIS and XSS of Q
1   mfis ← ∅, xss ← ∅, fis ← ∅, queryStatus ← ∅, list ← {Q}
2   while list ≠ ∅ do
3       Q' ← first query of list in BFS ordering
4       list ← list − {Q'}
5       parents_fis ← true
6       foreach t ∈ triplePatterns(Q) − triplePatterns(Q') do
7           parents_fis ← parents_fis ∧ ((Q' ∧ t) ∈ fis)
8       if parents_fis then
9           if Q' ∉ queryStatus then
10              queryStatus [Q'] ← FAIL_K(Q', D)
11          if queryStatus [Q'] then // if Q' fails
12              fis ← fis ∪ {Q'}
13              mfis ← mfis − superQueries(Q')
14              mfis ← mfis ∪ {Q'}
15              foreach t ∈ triplePatterns(Q') do
16                  if (Q' − t) ∉ list then
17                      list ← list ∪ (Q' − t)
18                      if var(Q' − t) = var(Q') then
19                          queryStatus [Q' − t] ← true
20                      else if card_max(p(t), D) = 1 ∧ s(t) ∈ var(Q' − t) then
21                          queryStatus [Q' − t] ← true
22          else // Q' is successful, and therefore an XSS
23              xss ← xss ∪ {Q'}
24  return mfis, xss
```

Algorithm 1: Enumerate the MFIS and XSS of a query Q

When using this property to avoid query executions, query success or failure is not known over the whole lattice but partial knowledge is sufficient here. Next, we consider deduction rules that predict query failures without executing them. As we run a breadth-first-search, a query is studied after all its superqueries, so we can leverage properties deducing the failure of a query from the failure of its superqueries.

Property 2. Given a query Q and triple pattern t, if $var(Q \wedge t) = var(Q)$ then $Q \wedge t$ fails $\Rightarrow Q$ fails.

Property 2 states that, if removing a triple pattern from a query does not remove any variables, then the number of answers cannot decrease. In our example, adding t_5 (?n type ERNurse) to a query containing variable n adds a constraint on n, and so cannot increase the number of answers.

Adding properties 1 and 2 to Base creates an improved algorithm, Var, shown in algorithm 1 (lines 20 and 21 do not apply, they are used in the next version). The main data structures are a list (list) of subqueries to evaluate and a map (queryStatus) storing the result of their evaluations: failure or success (lines 9-10). From the list, we consider queries from the lattice in a breadth-first order

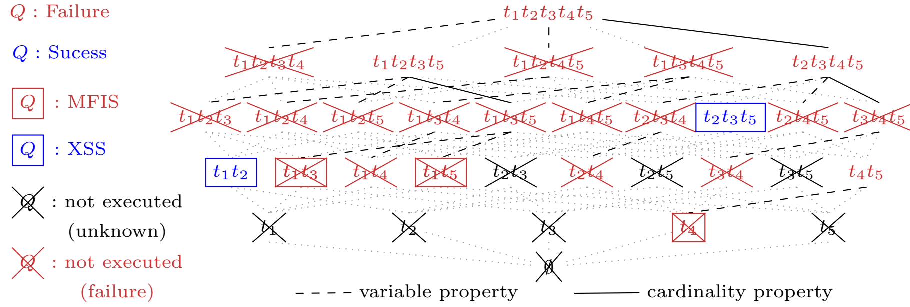


Fig. 3. Lattice of subqueries of Q with FULL algorithm

(lines 1-4). According to property 1, every direct superquery of Q' has to be an FIS for further consideration (lines 5-8). On query failure, the sets fis and mfis are updated (lines 12-14): the subquery Q' being considered replaces its direct superqueries in the mfis set as those can no longer be minimal (lines 13-14). We then consider every direct subquery of Q' (lines 15-21) for future evaluation (line 17), checking if property 2 is applicable (line 18) to predict its failure without executing it. If, instead, Q' succeeded, it is added to the xss set (line 23).

5.3 Cardinality-based Property

The last improvement leverages a property involving both the query and the data. We consider triple patterns that add a piece of information to each answer but do not overall change the number of answers. In the running example if t_2 (?d experience ?e) is removed from a query, as each person has at most one *experience*, each answer will lose a piece of information, but no two answers will become identical. So the number of answers will not decrease. We focus on predicates with maximum cardinality 1, of which any subject has at most one occurrence.

Definition 4. *The global maximum cardinality of a predicate p in a dataset D is [23]:* $\mathrm{card}_{\max}(p, D) = \max_{s \mid \exists\ p,o:(s,p,o) \in D} |\{(s,p,o) \mid (s,p,o) \in D\}|$

Property 3. Given a query Q, and a triple pattern t with a fixed predicate $p(t)$, if $\mathrm{card}_{\max}(p(t), D) = 1$ and $s(t) \in var(Q)$ then $Q \wedge t$ fails $\Rightarrow Q$ fails.

The complete algorithm, FULL, is created by adding property 3 to VAR (lines 20-21).

Figure 3 shows the lattice of subqueries of the query from the motivating example, and the subqueries avoided by FULL. For example, t_2 has maximum cardinality 1, the subject of t_2 (d) is one of the variables of $t_1t_3t_4t_5$, and $t_1t_2t_3t_4t_5$ fails. We deduce that $t_1t_3t_4t_5$ fails using property 3. In our example, FULL only executes 6 queries (rather than 31 for BASE and 9 for VAR).

Other cardinality definitions, like Class Cardinality [23], consider a smaller set of subjects than global cardinalities so may provide more precise values.

But these are query specific so a single cardinality value does not hold over the whole lattice. Cardinalities would need to be calculated at each query evaluation. Extending our approach with other cardinalities is a perspective for future work.

The deduction rule based on variables (property 2) is applicable to any query in any dataset as it relies only on information contained within the original query. However, the cardinality-based condition (property 3) requires additional information: if cardinalities are not enforced by the schema, they must be calculated for all predicates. For frequently modified KBs, cardinalities would need to be updated each time the data is changed. This requires KB administrators to provide updated cardinality values or users to regularly query the KB to obtain cardinalities, which is costly. So we provide two versions of our algorithm: the Full version with all optimization properties, and the variable-only version, Var.

6 Experimental Evaluation

Hardware Our experiments were run on a Ubuntu Server LTS system with an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz and 32GB RAM. The results presented are the average of five consecutive runs of the algorithms. To prevent a cold start effect, a preliminary run is performed but not included in the results.

Algorithms The Base, Var and Full algorithms are implemented[3] in Oracle Java 1.8 64bits and run using the Jena TDB triplestore. Cardinalities are precomputed for the Full algorithm. We set the threshold for plethoric answers K=100, as it is the default limit used by the DBpedia SPARQL endpoint. As Cartesian products are costly to execute, queries containing Cartesian products are split into connected parts so that in each part every triple pattern shares a variable with at least one other triple pattern, and so that separate parts share no variables. Each part is then executed separately, the number of answers of the original query being the product of the number of results of each part.

Synthetic Dataset and Queries We used a dataset of 11M triples, generated with the WatDiv benchmark. We have considered 21 queries with 4 to 12 triple patterns. There are 7 star queries (all triple patterns have the same subject), 7 chain queries (subject of triple pattern j+1 is the object of triple pattern j) and 7 composite queries (any other configuration). All chain queries and some star and composite queries are based on the WatDiv test cases (IL-1-10, F1, F2, F4, C2, C3). We added new composite and star queries to have varied characteristics.

Real Dataset and Queries We used the 3.9 version of the English DBpedia dataset, which contains 812M triples. Queries come from the LSQ project [8] which recorded user-submitted queries to DBpedia (version 3.5.1) between April 30 and June 20, 2010. Some minor adaptations were made as some original URIs were not compatible with version 3.9 of DBpedia. We have used 9 star or composite queries, containing 4 to 10 triple patterns.

[3] Our implementation is available at https://forge.lias-lab.fr/projects/tma4kb with a tutorial to reproduce experiments.

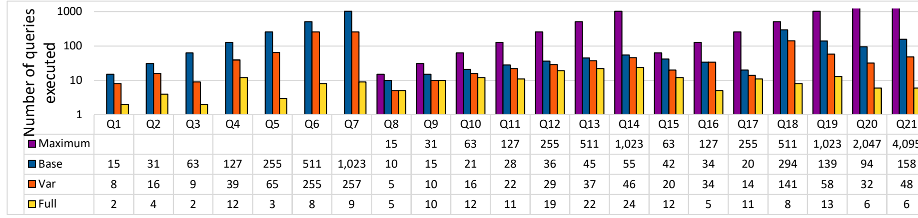


Fig. 4. # Executed queries Watdiv 11M triples

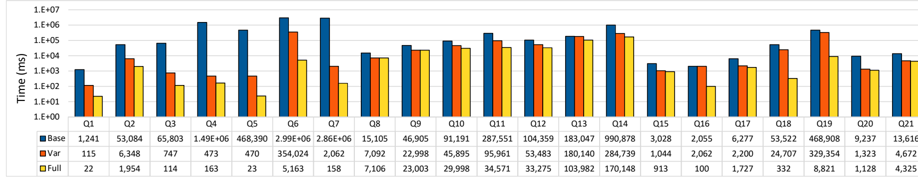
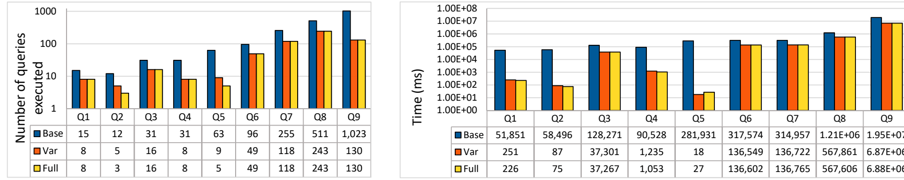


Fig. 5. Execution time Watdiv 11M triples



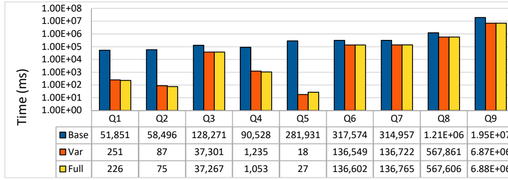


Fig. 6. # Executed queries DBpedia

Fig. 7. Execution time DBpedia

6.1 Results

Synthetic Dataset and Queries First we study the performance of the three algorithms on a WatDiv generated dataset of 11M triples. The number of executed queries and the execution time for each algorithm are given in figures 4 and 5.

For all the queries tested, we verify that Var and Full execute at most as many queries as Base. This is a guarantee of the properties presented in section 5. The improvement is less notable for chain (Q8 to Q14) and composite (Q15 to Q21) queries. Indeed, these queries have subqueries containing Cartesian products. Since we execute Cartesian products by separating them into connected parts, queries that have a succeeding superquery can be executed as part of a Cartesian product. The number of queries executed by the Base algorithm if Cartesian products were executed directly, $2^n - 1$, is given in the *Maximum* bars on figure 4. Overall Var and Full execute respectively 46% and 75% fewer queries than Base. The execution times follow the same general trend. Full is faster than Var, itself faster than Base. The improvement in execution time is smaller than the improvement in query executions. Indeed, all query executions are not equal, and the executions avoided can have short execution times. Overall Var saves 65% of the Base execution time, and Full saves 83%.

Real Dataset and Queries We show the number of executed queries and the runtime of each algorithm in figures 6 and 7. As in the WatDiv experiments, Var

and Full execute at most as many queries as Base. However, we notice that Full rarely executes fewer queries than Var. This can be explained by cardinalities in DBpedia. Few of the predicates used have maximum cardinality 1, so the cardinality property cannot be applied to them. Only 0.67% of predicates in DBpedia have maximum cardinality 1, but 66% of predicates have maximum cardinality 2. Upon further investigation, many predicates that should in theory have maximum cardinality 1, such as BirthDate, in fact have maximum cardinality 2. This can be due to errors in the data or uncertain information [24, 25]. Using a curated dataset would likely improve the benefit of the cardinality-based pruning. Consequently, Var and Full have very similar execution times, each saving around 78% of the baseline time.

Some queries, like Q9, have long execution times (over an hour). The execution time depends heavily on the time the triplestore takes to answer a query on our server. It could be reduced by using a distributed solution or optimizing how the query evaluation is performed by the triplestore. This has not yet been investigated and is a prospect of improvement. Our experiments show that for most queries, Var and Full run in a few seconds, despite using a centralized server.

7 Conclusion

In this paper, we have addressed the plethoric answers problem in the context of RDF queries. We have identified that none of the approaches proposed in the literature try to identify why the user query produced plethoric answers. Yet, several approaches developed for other unsatisfactory answers problems have shown that the first step in a query adjustment process designed to meet the user expectation should be understanding why the query failed.

Our goal was to fill this gap. We have first shown that the notions defined for other unsatisfactory answers problem are too restrictive for our context and defined a more general notion, named MFIS. Starting from a baseline method to calculate all MFIS and XSS of a failing query, we have proposed improvements based on query and data properties, to reduce the number of queries that need to be executed, and therefore reduce the run-time of our algorithms. Experiments using both synthetic and real data show that our optimized algorithms offer a significant improvement. Var saves 71% of the baseline time and can be used for any query, and Full saves 82% of the baseline time but requires additional information: predicate cardinalities.

The next step will be to use the MFIS and XSS to aid in rewriting queries with plethoric answers. Query modification can be performed entirely by the user (i.e. we provide the MFIS and XSS and the user interprets them to adapt their query), entirely automatically, or with an interactive approach, where the user is guided through changes applied to their query.

References

1. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - A Large-

scale, Multilingual Knowledge Base Extracted from Wikipedia. Semantic Web **6**(2) (2015) 167–195
2. Harris, S., Seaborne, A.: Sparql 1.1 query language. W3C Recommendation (2013)
3. Brickley, D., Guha, R.: Rdf schema 1.1. W3C Recommendation (2014)
4. Webber, B.L., Mays, E.: Varieties of user misconceptions: Detection and correction. In: IJCAI'83. Volume 2. (1983) 650–652
5. Jagadish, H.V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A., Yu, C.: Making database systems usable. In: SIGMOD'07. (2007) 13–24
6. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Cooperative techniques for SPARQL query relaxation in RDF databases. In: ESWC'15, Springer (2015) 237–252
7. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of rdf data management systems. In: ISWC'14, Springer (2014) 197–212
8. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: The Linked SPARQL Queries Dataset. In: ISWC'15. (2015) 261–269
9. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. **40**(4) (2008)
10. Xie, M., Wong, R.C.W., Peng, P., Tsotras, V.J.: Being happy with the least: Achieving α-happiness with minimum number of tuples. In: ICDE'20, IEEE (2020) 1009–1020
11. Chaudhuri, S., Das, G., Hristidis, V., Weikum, G.: Probabilistic ranking of database query results. In: VLDB'04. Volume 30. (2004) 888–899
12. Ozawa, J., Yamada, K.: Discovery of global knowledge in a database for cooperative answering. In: IEEE'95. Volume 2. (1995) 849–854
13. Bosc, P., HadjAli, A., Pivert, O.: About overabundant answers to flexible queries. In: IPMU'06. Volume 6. (2006) 2221–2228
14. Moises, S.A., Pereira, S.d.L.: Dealing with empty and overabundant answers to flexible queries. Journal of Data Analysis and Information Processing (2014) 12–18
15. Bosc, P., Hadjali, A., Pivert, O., Smits, G.: Une approche fondée sur la corrélation entre prédicats pour le traitement des réponses pléthoriques. In: EGC'10. 273–284
16. Song, Q., Namaki, M.H., Wu, Y.: Answering why-questions for subgraph queries in multi-attributed graphs. In: ICDE'19. (2019) 40–51
17. Vasilyeva, E., Thiele, M., Bornhövd, C., Lehner, W.: Answering "why empty?" and "why so many?" queries in graph databases. JCSS **82**(1) (2016) 3–22
18. Wang, M., Liu, J., Wei, B., Yao, S., Zeng, H., Shi, L.: Answering why-not questions on sparql queries. Knowledge and Information Systems **58** (2019) 169–208
19. Godfrey, P.: Minimization in Cooperative Response to Failing Database Queries. International Journal of Cooperative Information Systems **6**(2) (1997) 95–149
20. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Rdf query relaxation strategies based on failure causes. In: European semantic web conference, Springer (2016) 439–454
21. Jannach, D.: Techniques for fast query relaxation in content-based recommender systems. In: Annual Conference on Artificial Intelligence, Springer (2006) 49–63
22. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. ACM Trans. Database Syst. **34**(3) (2009)
23. Dellal, I.: Management and Exploitation of Large and Uncertain Knowledge Bases. PhD thesis, ISAE-ENSMA - Poitiers (2019)
24. Giacometti, A., Markhoff, B., Soulet, A.: Mining Significant Maximum Cardinalities in Knowledge Bases. In: ISWC'19. Volume 11778. (2019)
25. Muñoz, E., Nickles, M.: Mining cardinalities from knowledge bases. In: DEXA'17. Volume 10438. (2017) 447–462