HAL open science

Explaining Unexpected Answers of SPARQL Queries

Louise Parkin, Brice Chardin, Stéphane Jean, Allel Hadjali

▶ To cite this version:

Louise Parkin, Brice Chardin, Stéphane Jean, Allel Hadjali. Explaining Unexpected Answers of SPARQL Queries. Web Information Systems Engineering – WISE 2022, Nov 2022, Biarritz, France. pp.136-151, 10.1007/978-3-031-20891-1_11 . hal-03847824

HAL Id: hal-03847824
https://hal.science/hal-03847824

Submitted on 10 Nov 2022

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.

Explaining Unexpected Answers of SPARQL Queries

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

[1] ISAE-ENSMA, LIAS, Chasseneuil-du-Poitou, France
louise.parkin@ensma.fr brice.chardin@ensma.fr allel.hadjali@ensma.fr
[2] Université de Poitiers, LIAS, Poitiers, France
stephane.jean@univ-poitiers.fr

Abstract. *"Why am I not getting the right answer?"* is a question many Knowledge Base users may ask themselves. In particular, novice users can easily make mistakes and find differences between the answer they expected and the answer they got. This problem is known as the unsatisfactory answer problem. A subproblem, where no answers are returned, has been widely studied and identifying failure causes can help users modify their queries to fit their requirements. But users may be unhappy with their results for multiple other reasons: they may be overwhelmed by too many answers, expect a particular answer that is not included, or even encounter a combination of these problems. In this paper, we classify the various types of unsatisfactory answers, and propose algorithms to compute generalized failure causes. We evaluate the performance of our algorithms and show that they perform comparably to existing problem-specific methods, while being more extensive.

Keywords: Knowledge bases, SPARQL, Unexpected answers, MFIS, XSS

1 Introduction

A Knowledge Base (KB) is a solution for storing data as RDF triples (subject, predicate, object). KBs are widely used in industry and academia. Well-known examples of KBs are DBpedia [16] and Google's Knowledge Vault [6]. Users unfamiliar with the technology are likely to need to extract information from a KB. Specific interfaces can allow for text based searching, but the most common method for retrieving information from a KB is the SPARQL query language [11].

This version of the contribution has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at http://dx.doi.org/10.1007/978-3-031-20891-1_11. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www/springernature.com/gp/open-research/policies/accepted-manuscript-terms.

subject	predicate	object
p_1	age	25
p_1	suffersFrom	brokenArm
p_1	suffersFrom	flu
p_1	ward	ICU
p_1	status	Dead
p_2	age	47
p_2	suffersFrom	stroke
p_2	ward	ER
n_1	worksIn	ICU
n_1	treats	p_3
n_2	worksIn	ER
n_2	treats	p_2
n_2	treats	p_3
n_3	worksIn	ICU
n_3	treats	p_1
n_3	treats	p_2

(a) Knowledge base D

```
SELECT * WHERE {
?p ward ICU .          #t1
?p status Dead .       #t2
?n treats ?p .         #t3
?n ward ICU .          #t4
?p suffersFrom ?i } #t5
No answers
```

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

```
SELECT * WHERE {
?p ward ICU .      #t1
?p status Dead . #t2
?n treats ?p .     #t3
?n ward ICU .   } #t4
No answers
```

(c) $Q' = t_1 t_2 t_3 t_4$

```
SELECT * WHERE {
?p ward ICU .          #t1
?p status Dead .       #t2
?n ward ICU .          #t4
?p suffersFrom ?i } #t5
Too many answers
```

(d) Query $Q'' = t_1 t_2 t_3 t_5$

```
SELECT * WHERE {
?p ward ICU .          #t1
?p status Dead .       #t2
?p suffersFrom ?i } #t5
Satisfactory answers
```

(e) $Q''' = t_1 t_2 t_5$

Fig. 1. A Knowledge Base, SPARQL queries and their results

Novice users can struggle to write queries, inaccurately describing their requirements. Therefore, *mistakes* or *misconceptions* may appear, causing unexpected or unsatisfactory answers. Mistakes refer to the user incorrectly writing their query, for example misspelling a term. Misconceptions refer to a user's incorrect understanding of a KB [27]. There are five unexpected answers problems, each linked to a why-question: no answers (*why empty*), too few answers (*why so few*), too many answers (*why so many*), missing answers (*why not*), and unwanted answers (*why so*). So far each problem has been studied separately, so combined problems are difficult to handle. These can occur if fixing one type of unexpected answer creates another (i.e. turning an insufficiently restricted query into a too restrictive one), or in situations with precise cardinality requirements.

Consider the example of a hospital KB, and a user who wants information on the patients who died in the intensive care unit (ICU) and suppose the user expects around 100 answers based on their knowledge of the hospital. A section of the KB and succession of query attempts are shown in figure 1. The user writes a SPARQL query (b), but receives no answers. From their contextual knowledge, the user determines there must be a mistake somewhere. However they have no way of knowing what is causing the problem, as there are several possibilities. There could be an inappropriate term within the query (ICU rather than Intensive Care Unit), or some incompatible properties (dead patients are not being treated by anyone). Once the user manages to produce some results (d), they find over 10,000 answers, which is far too many to be able to deal with. Again they will wonder where the mistake is, as they have fixed the initial problem, but created another. Some possible explanations for the overabundant answers are having two triple patterns whose combination causes a multiplication of the number of answers (illness and the people treating a patient) or having an insufficiently constrained variable. All in all, a user faced with unsatisfactory answers must undergo a time-consuming and frustrating trial and error process to fix their query without a guarantee to receive the expected answers in the end.

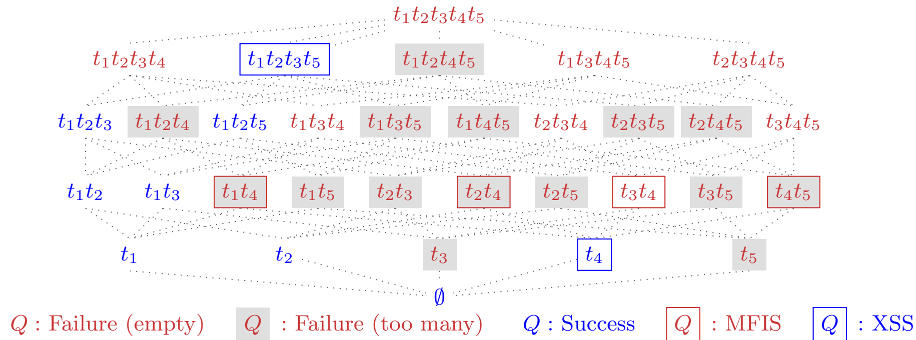

Fig. 2. Lattice of subqueries of Q

To help a user facing an unexpected answer, there are two approaches: explain what is causing these answers, or suggest changes to the query. Existing work focusing on a specific type of unexpected answers has mainly explored the query modification solution [18, 23]. The answer explanation strategy has been successfully used to deal with empty answers [8] and was shown to increase the performance of subsequent query modification steps [2, 14, 7]. Figure 2 shows how identifying failure causes would help to deal with the previous query by studying its constituent parts. The lattice shows the successful queries, and those that fail either because they have empty answers or too many answers (for a threshold of 200 in this example). We are interested in providing the MFIS (Minimal Failure Inducing Subqueries) and XSS (maXimal Succeeding Subqueries). The MFIS are the smallest parts of the query that induce unexpected answers if they are part of a query and the XSS are largest parts of the query that produce acceptable answers. Here, the MFIS show that the inclusion of triple pattern t_4 : `?n ward ICU` with any other triple pattern causes unexpected answers (either because there will be empty answers if t_3 is also included, or because there will be too many answers with another triple pattern). Indeed, the user has used the property *ward* which applies to a patient, but with subject n which in the context of the query is meant to indicate a member of the hospital staff. This query could be fixed by removing triple pattern t_4, therefore replacing the query with the XSS $t_1t_2t_3t_5$, or by modifying the predicate in t_4 from *ward* to *worksIn*. The interpretation of MFIS and XSS to fix a query is beyond the scope of this paper.

In this paper, we apply the failure cause definition used for the why empty and why so many problems, called Minimal Failure Inducing Subquery (MFIS) to the other elementary problems in order to propose a generalized system to deal with unsatisfactory answers. We will show that our method can cope with combinations of unexpected answers and we will perform experimental evaluation to determine its usability on real data from DBpedia along with real queries from the *Linked SPARQL Queries Dataset* [22].

We start by exploring related work for each unexpected answer problem in section 2. In section 3, we give the elementary notions of RDF, and introduce a

classification for unexpected answers and related properties. Section 4 presents the algorithms for computing failure causes. We perform experimental evaluation of our algorithms in section 5 and conclude with future prospects in section 6.

2 Related Work

Unexpected answers have been studied using both a data based and a query based approach. The data based approach provides information on data provenance: operations that led to missing information [12] or the data source producing problematic answers [28]. These methods can help database providers enhance data quality, but rarely help end users with no control over the KB content. On the other hand, the query based approach uses the hypothesis that the user incorrectly specified their query so it does not match their requirement.

To fix the unexpected answer problem, most query-based methods modify the user query by removing, changing or adding triple patterns. Various modification strategies have been used for specific problems. For the why empty and why so many problems in graph queries, maximum common connected subgraphs are computed by removing parts of the initial query to create the largest succeeding query graph [25]. For the same problems in KBs, maXimal Succeeding Subqueries are computed by triple pattern suppression [9]. For the why not problem in relational databases, a similarity metric based on edit distance is used to rank modified queries [24]. Exact algorithms and heuristics are used for the why not and why so problems in knowledge graphs, to change a query step by step to include new answers [23]. To deal with the why so many problem with fuzzy queries, intensification techniques can be used [18]. New queries are found via an exhaustive search [25, 23], or using semantic information to chose more relevant queries [24]. New queries may still produce unexpected answers, so finding queries that return the desired answers remains a trial and error process.

To address this issue, some query based techniques identify the reasons for unexpected answers. Failure causes were introduced for the why empty problem: false presuppositions are returned to the user if their query produces no answers [15, 19]. Minimal Failing Subqueries (MFS) have then been used to describe the smallest parts of a query that lead to empty answers [9, 7, 5]. A related notion, Minimal Failure Inducing Subquery, is used for the why so many problem [20]. Failure causes have also been used in the why not problem in relational databases [1], and KBs [26] to identify the triple pattern or SPARQL operator responsible for an absent answer. To our knowledge, no query based failure causes have been studied for the why so few and why so problems, or combined problems.

Failure causes can be used to enhance query modification by focusing changes on the parts responsible for unexpected answers. MFS have been used in automated query modification systems dealing with empty answers [2, 14] and in interactive query rewriting frameworks, where users can select parts to be relaxed [14, 17]. In RDF, a hybrid method balances the MFS computation cost and the gain of not executing modified queries [7]. Thus, the efficiency of query modification methods can be improved if unexpected answers have been explained first.

3 Problem Formalization

We start by describing the formalism and semantics of RDF and SPARQL necessary for the paper, using notations and definitions from Pérez et al. [21]. We then give definitions for unexpected answers and introduce related properties. For space considerations, some proofs are not provided here.

3.1 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. An *RDF triple* is a triple (subject, predicate, object) $\in (I \cup B) \times I \times (I \cup B \cup L)$. An *RDF database* stores a set of RDF triples. We also consider V a set of variables disjoint from $I \cup B \cup L$.

RDF Queries 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. In this paper, we will consider *RDF queries* defined as conjunctions of triple patterns $Q = SELECT\ *\ WHERE\ t_1 \cdots t_n$, which we write as $Q = t_1 \cdots t_n$. We write that t is a triple pattern appearing in a query Q by $t \in Q$. 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 $\forall t \in Q', t \in Q$. Then Q is a *superquery* of Q'. For a query $Q = t_1 \cdots t_n$ and a triple pattern $t \notin Q$, we denote the addition of t to Q by $Q \wedge t = t_1 \cdots t_n t$. This notation is extended to queries, so $Q \wedge Q'$ refers to the conjunction of Q and Q'. Conversely for $t' \in Q$, removing t from Q is denoted by $Q - t$.

Query Evaluation A mapping μ from V to T is a partial function $\mu : V \to T$. 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. The restriction of a mapping μ to a subset of variables $var \subseteq V$ is denoted by $\mu_{|var}$ and defined as a partial function $\mu_{|var} : var \to T$, where $\forall x \in var, \mu_{|var}(x) = \mu(x)$. 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 $[[Q]]_D = [[t_1]]_D \bowtie \cdots \bowtie [[t_n]]_D$.

Boolean Function A boolean function of n variables is a function on B^n into B, where B is the set [0,1], n is a positive integer, and B^n denotes the n-fold cartesian product of the set B with itself [3].

3.2 Unexpected Answers

There are many reasons why a user may be unsatisfied by the answer to their query. To formalize this, we define a *failure condition*, using the formalism of

boolean functions, which takes as entry a query Q, and returns a boolean value describing whether its answer is satisfactory or not when executed on a given KB. The boolean function has n variables, $(x_1, ..., x_n)$ where n is the number of triple patterns of the initial query, and $\forall i$, $x_i = 1 \iff Q$ contains t_i.

For a query Q and a KB D, we denote this property: $FAIL(Q, D)$. There are five elementary types of query failure [13]. They are based on the query result $[[Q]]_D$, and can be split into cardinality based, and content based properties.

Cardinality based failure conditions depend on the number of answers of a query. As such, it is not necessary to know the content of the answers to determine if the query succeeds or fails. The failure conditions will be based on a threshold value K to be determined by the user or the system. They are:

- *Why so many*: $FAIL_{>K}(Q, D) = (|[[Q]]_D| > K)$ for $K \geq 0$.
- *Why so few*: $FAIL_{<K}(Q, D) = (|[[Q]]_D| < K)$ for $K > 0$.
- *Why empty*: $FAIL_{\emptyset}(Q, D) = (|[[Q]]_D| = 0)$. This is a special case of *Why so few* with K=1.

Example 1. $FAIL_{<2}(Q, D)$ is a failure property denoting that queries with under 2 answers fail. In the example in figure 1, $t_1t_3t_4$ and t_2t_5 fail with respectively 0 and 1 answer. t_4t_5 and $t_1t_2t_5$ succeed with respectively 3 and 2 answers.

Content based failure conditions are based on the content of the answers. In particular, a user is interested in obtaining or avoiding a particular mapping. The failure conditions will be based on a provided mapping μ_w. They are:

- *Why not* : $FAIL_{\not\subseteq\mu_w}(Q, D) = (\forall\mu \in [[Q]]_D : \mu_w \textit{ and } \mu \textit{ are not compatible})$
- *Why so* : $FAIL_{\subseteq\mu_w}(Q, D) = (\exists\mu \in [[Q]]_D : \mu_w \textit{ and } \mu \textit{ are compatible})$

For the content based failure condition to have meaning, the variables of the mapping must all be included in the query. As such, $FAIL_{\not\subseteq\mu_w}(Q, D)$ and $FAIL_{\subseteq\mu_w}(Q, D)$ are undefined if $dom(\mu_w) \not\subseteq var(Q)$. So when studying subqueries obtained by removing triple patterns, we will eliminate from the search space any query which does not respect this condition.

Example 2. For μ_w, where $dom(\mu_w) = \{p\}$ and $\mu_w(p) = p_2$, $FAIL_{\not\subseteq\mu_w}(Q, D)$ is a failure property denoting that queries where no answer has p_2 as the value for variable p fail. In figure 1, t_1t_3 and t_2t_5 fail, and t_4t_5 succeeds. t_4 is not a valid query for this problem, as variable p from the mapping is missing.

For μ_w, where $dom(\mu_w) = \{n\}$ and $\mu_w(n) = p_2$, $FAIL_{\subseteq\mu_w}(Q, D)$ is a failure property denoting that queries where an answer has p_2 as the value for variable n fail. In figure 1, $t_1t_2t_3t_4t_5$ succeeds, and t_2t_4 fails. t_1 is not a valid query for this problem, as variable n is missing.

These five elementary unexpected answers can be combined using the basic logical operators conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$).

Negation For each problem, the negation can be expressed as another problem:

- $\neg(FAIL_{>K}(Q, D)) = FAIL_{<K+1}(Q, D)$

- $\neg(FAIL_{\subseteq\mu_w}(Q,D)) = FAIL_{\not\subseteq\mu_w}(Q,D)$
- $\neg(FAIL_{\emptyset}(Q,D)) = FAIL_{>0}(Q,D)$

Conjunction and Disjunction We define conjunction and disjunction based on the failure conditions:

- $(FAIL_1 \wedge FAIL_2)(Q,D) = FAIL_1(Q,D) \wedge FAIL_2(Q,D)$
- $(FAIL_1 \vee FAIL_2)(Q,D) = FAIL_1(Q,D) \vee FAIL_2(Q,D)$

3.3 Relations Between Problems

The why not problem can be transformed into a why empty problem. We consider a query Q and a why not question μ_w mapping some variables v to $\mu_w(v)$. We build a new query Q', based on Q by replacing the variables v with their mappings according to μ_w. This technique was used to deal with the why not problem in KBs [26]. The following property transforms a failure of Q into a failure of Q':

Property 1. Consider a query Q, a why not question μ_w, and Q' built by replacing each $v \in dom(\mu_w) \cap var(Q)$ by $\mu_w(v)$. $FAIL_{\not\subseteq\mu_w}(Q,D) = FAIL_{\emptyset}(Q',D)$

Proof. If $FAIL_{\not\subseteq\mu_w}(Q,D) = true$, suppose $\exists\mu' \in [[Q']]_D$. From the definition of Q', $dom(\mu') \cap dom(\mu_w) = \emptyset$, so μ' and μ_w are compatible. So $\mu = \mu' \cup \mu_w$ is a mapping, $\mu \in [[Q]]_D$, and μ and μ_w are compatible which contradicts $\nexists\mu \in [[Q]]_D, \mu$ and μ_w are compatible. So $[[Q']]_D = \emptyset$ and $FAIL_{\emptyset}(Q',D) = true$.

If $FAIL_{\emptyset}(Q',D) = true$, suppose $\exists\mu \in [[Q]]_D$ such that μ_w and μ are compatible. From the definition of Q', $\mu_{|var(Q')} \in [[Q']]_D$ which contradicts $[[Q']]_D = \emptyset$. So $\nexists\mu \in [[Q]]_D, \mu$ and μ_w are compatible, and $FAIL_{\not\subseteq\mu_w}(Q,D) = true$.

Similarly, a why so problem can be transformed into a why so many problem (with a threshold K=1).

Property 2. Consider a query Q, a why not question μ_w, and Q' built by replacing each $v \in dom(\mu_w) \cap var(Q)$ by $\mu_w(v)$. $FAIL_{\subseteq\mu_w}(Q,D) = FAIL_{>1}(Q',D)$

3.4 Monotony of the Failure Condition

A particular type of boolean functions are Monotone Boolean Functions (MBF) [3]. A boolean function f is *positive* if for each of its variables x, $f_{x=0} \leq f_{x=1}$. In that case, we call the associated failure condition *positive*, meaning that if a query fails, all its superqueries fails, and therefore if a query succeeds, all its subqueries succeed. More formally, $FAIL(Q',D) \wedge Q'' \subseteq Q' \implies FAIL(Q'',D)$. Considering the partial order of queries based on triple pattern inclusion, the failure condition is upward closed. A boolean function f is *negative* if for each of its variables x, $f_{x=0} \geq f_{x=1}$. In that case, we call the associated failure condition *negative*, meaning that if a query succeeds, all its superqueries succeed, and if a query fails, all its subqueries fail. A *monotonic* failure condition is either a positive or a negative failure condition.

problem	cardinality or content	failure condition	monotonic
why empty	cardinality	$[[Q]]_D = \emptyset$	positive
why so many	cardinality	$\|[[Q]]_D\| > K$	no monotony
why so few	cardinality	$\|[[Q]]_D\| < K$	no monotony
why not	content	$\forall \mu \in [[Q]]_D \mu_w \not\subseteq \mu$	positive
why so	content	$\exists \mu \in [[Q]]_D \mu_w \subseteq \mu$	negative

Fig. 3. The five elementary failure conditions

Property 3. $FAIL_\emptyset$ and $FAIL_{\not\subseteq \mu_w}$ are positive properties.

As $\neg(FAIL_{\subseteq \mu_w}(Q, D)) = FAIL_{\not\subseteq \mu_w}(Q, D)$, it follows that the why so problem has a negative failure condition. We show that the why so few and why so many problems do not have monotonic failure conditions with a counter-example.

Example 3. We use KB D and query Q from figure 1, failure causes $F_1(Q, D) = FAIL_{>1}(Q, D)$, $F_2(Q, D) = FAIL_{<2}(Q, D)$ and subqueries $Q_1 = t_3$, $Q_2 = t_2t_3$, $Q_3 = t_1t_2t_3t_5$. $|[[Q_1]]_D| = 5$, $|[[Q_2]]_D| = 1$, $|[[Q_1]]_D| = 2$. As $Q_1 \subseteq Q_2 \subseteq Q_3$ and $F_1(Q_1, D) = true$, $F_1(Q_2, D) = false$, $F_1(Q_3, D) = true$, F_1 is not monotnic. As $F_2(Q_1, D) = false$, $F_2(Q_2, D) = true$, $F_2(Q_3, D) = false$, F_2 is not monotonic.

Table 3 summarises the characteristics of all five elementary failure conditions. Some combinations of failure properties can also be monotonic.

Property 4. A conjunction or disjunction of positive (resp. negative) properties is positive (resp. negative).

4 Failure Causes

Having defined what a user can consider a query failure, we want to provide tools describing the parts of the query responsible for query failure. These notions should be computed quickly to improve user experience. To that end we suggest algorithms to determine failure causes that execute as few queries as possible.

4.1 Definitions

Our aim is to identify the parts of a query responsible for its failure. Building on the Minimal Failing Subquery (MFS) and maXimal Succeeding Subqueries (XSS) introduced in the why empty problem, the following definitions have been proposed to deal with the why so many answers problem [20].

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

$$\mathit{fis}(Q, D, FAIL) = \{Q^* \mid Q^* \subseteq Q \wedge FAIL(Q^*, D) \wedge \forall Q' \subseteq Q,\ Q^* \subset Q' \Rightarrow FAIL(Q', D)\}$$

Definition 2. *A Minimal Failure Inducing Subquery (MFIS) of a query is one of its failure inducing subqueries such that none of its subqueries are FIS. The set of MFIS of a query Q is:*

$$mfis(Q, D, FAIL) = \{Q^* \in fis(Q, D, FAIL) \mid \nexists Q' \subset Q^*,\ Q' \in fis(Q, D, FAIL)\}$$

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

$$xss(Q, D, FAIL) = \{Q^* \mid Q^* \subseteq Q \land \neg FAIL(Q^*, D) \land \\ \forall Q', Q^* \subset Q' \Rightarrow Q' \in fis(Q, D, FAIL)\}$$

These definitions do not rely on the monotony of the failure condition. In this paper we apply them in a general setting, for any unexpected answer problem. In the case of a negative failure condition (for the why so problem), if the initial query fails then all its subqueries fail. This means that there are no XSS, and that the MFIS are all the smallest subqueries that contain the variables of the missing mapping. The notions of MFIS and XSS are therefore not very useful to solve problems with a negative failure condition. They can however be used if that failure condition is a part of a disjunction in a bigger failure condition. We will now consider methods for identifying the MFIS and XSS of a query.

4.2 Computation

Finding the queries that succeed and fail can be accomplished by executing them on the KB and analyzing the answer or by applying deduction rules based on the answers of related queries. To measure the cost of finding MFIS, we will start by using the metric of number of queries executed. The number of query executions is measured relative to n, the number of triple patterns in the initial query, as the search space, i.e. the number of subqueries is equal to $2^n - 1$. For any problem, a baseline method, called BASE is to execute every subquery, and check if it succeeds or fails using the failure property.

In the general case, finding all the MFIS and XSS can require an exponential number of operations, as a query can have up to $\binom{n}{\lfloor n/2 \rfloor}$ MFIS and XSS [9]. However for some problems, we can apply properties to execute fewer queries.

Positive Failure Properties If the failure property is positive, the Lattice Based Algorithm (LBA) proposed by Fokou [8] based on the a_mel_fast algorithm by Godfrey [9] can be used to compute the MFIS. It uses the following property:

Property 5. Consider a positive failure property $FAIL$, queries Q and Q_i and triple pattern t_i with $Q = Q_i \land t_i$. If $FAIL(Q, D)$ and $\neg FAIL(Q_i, D)$, every MFIS of Q contains t_i.

The LBA algorithm finds a first MFIS Q^* by removing triple patterns from the initial query Q one by one. For a triple pattern t_i, and Q_i where $Q = Q_i \land t_i$, if Q_i succeeds, t_i is added to Q^*. LBA continues, replacing Q with $Q_i \land Q^*$. A first MFIS Q^* is found once all triple patterns have been removed. The process is repeated over the largest subqueries of Q that do not contain the found MFIS.

Pruning Properties To reduce the number of query executions, some query properties can be leveraged. A general property can be applied to all problems [20]:

Property 6. If a subquery Q' succeeds and $Q'' \subset Q'$, Q'' is not an MFIS or XSS.

Using this property, subqueries of XSS do not need to be studied, so we start by checking that the query does not have a succeeding query, and only study its failure if that is the case. The why so many study also introduces properties that use the failure of a query to determine the failure of another query [20]. We have adapted these to be useful to any cardinality based problem.

Property 7. Given a query Q and triple pattern t, if $var(Q \wedge t) = var(Q)$ then $|[[Q \wedge t]]_D| \leq |[[Q]]_D|$.

Definition 4. *The global minimum and maximum cardinality of a predicate p in a dataset D are [4]:*

$$\text{card}_{\min}(p, D) = \min_{s \mid \exists\ p,o:(s,p,o) \in D} |\{(s,p,o) \mid (s,p,o) \in D\}|$$

$$\text{card}_{\max}(p, D) = \max_{s \mid \exists\ p,o:(s,p,o) \in D} |\{(s,p,o) \mid (s,p,o) \in D\}|$$

Property 8. Given a query Q, and a triple pattern t with a fixed predicate $p(t)$, a variable object $o(t) \notin var(Q)$ and $s(t) \in var(Q)$, if $\text{card}_{\max}(p(t), D) = 1$ then $|[[Q \wedge t]]_D| \leq |[[Q]]_D|$ and if $\text{card}_{\min}(p(t), D) = 1$ then $|[[Q \wedge t]]_D| \geq |[[Q]]_D|$.

These properties are useful to determine an upper or lower bound of the number of answers to a query if the number of answers of another query is known. For cardinality based problems they can be used to determine success or failure of queries without executing them. In a top-down algorithm, these properties can be used to determine the status of a query Q, knowing $Q \wedge t$, but for an algorithm with another execution order, they can be applied in the other direction.

Dealing with Combined Problems When considering a combined problem, there are two possibilities for finding the MFIS. The first option is to execute an algorithm for each elementary part of the failure cause, then combine the results. The second option is to execute a single algorithm as previously, applying the combined failure definition when considering a query execution. In the first case, we should consider whether the knowledge of the MFIS and XSS for two failure properties is sufficient to determine the MFIS and XSS for their combination. The MFIS and XSS of a query for a conjunction of failure properties can be computed from the MFIS and XSS of the query for each property.

Property 9. Consider a query Q, a failure condition $FAIL = F_1 \wedge F_2$, Q' an MFIS of Q for $FAIL$ and Q'' an XSS of Q for $FAIL$. Q' is a superquery of an MFIS of Q for F_1 and a superquery of an MFIS of Q for F_2. Q'' is an XSS of Q for F_1 or for F_2.

Proof. Consider Q' an MFIS of Q for $FAIL$. Since $Q' \in mfis(Q, D, FAIL)$, then $Q' \in fis(Q, D, FAIL)$. So $Q' \in fis(Q, D, F_1)$ and $Q' \in fis(Q, D, F_2)$.

From the MFIS definition, $\exists Q_1 \in mfis(Q, D, F_1)$ and $\exists Q_2 \in mfis(Q, D, F_2)$ such that $Q_1 \subseteq Q'$ and $Q_2 \subseteq Q'$.

Consider Q' an XSS of Q for $FAIL$. Suppose Q' is not an XSS of Q for F_1. Either $\exists Q'', Q' \subseteq Q'' \wedge F_1(Q'', D) = false$, which means $FAIL(Q'', D) = false$ and contradicts the assertion that Q' is an XSS of Q for $FAIL$ or $F_1(Q', D) = true$. Since $FAIL(Q', D) = false$, then $F_2(Q', D) = false$. Suppose $\exists Q''', Q' \subseteq Q''' \wedge F_2(Q''', D) = false$, meaning $FAIL(Q''', D) = false$ and contradicting the assertion that Q' is an XSS of Q for $FAIL$. So Q' is an XSS of Q for F_2.

To show that this is not the case for a disjunction of failure properties, consider the introduction example. The failure cause was $FAIL = FAIL_{\emptyset} \vee FAIL_{>200}$. For the initial query Q, we have $FAIL_{>200}(Q, D) = false$. So Q is the XSS and there are no MFIS for $FAIL_{>200}$. But $FAIL_{>200}$ is significant for finding the MFIS and XSS for $FAIL$. So for a disjunction of failure properties, we need to know the entire status of the lattice. The previous algorithms must therefore be adapted to return the status of the whole lattice rather than the MFIS and XSS.

5 Experimentation

We propose two sets of experiments. The first will compare variations of our algorithms to illustrate the various improvements from the use of the properties introduced in section 4. The second will compare our generic algorithm, Shiny, with specific solutions proposed for the why empty [7], why so many [20], and why not [26] problems. We will observe the performance cost of a general method compared with specific solutions, and compare the proposed failure causes.

5.1 Experimental Setup

Hardware Our experiments were run on a Ubuntu Server 16.04 LTS system with Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz 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 algorithms are implemented in Oracle Java 1.8 64bits and run on top of Jena TDB. To compare the generic solution to specific methods from the state of the art, we have used the QARS system [7] (why empty), the TMA4KB system [20] (why so many), and the ANNA system [26] (why not). We used the reference implementations for the first two systems[3][4], and we re-implemented the ANNA algorithm as a reference implementation was not available.

[3] https://forge.lias-lab.fr/projects/qars
[4] https://forge.lias-lab.fr/projects/tma4kb

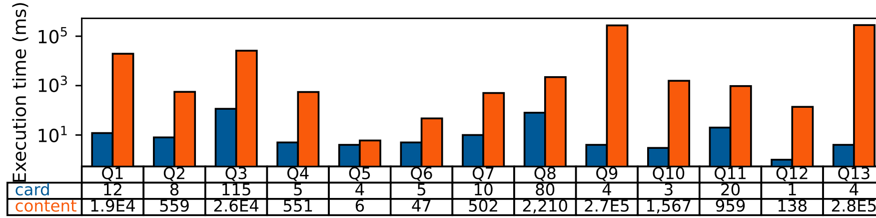

Fig. 4. Execution time for WhyNot queries with cardinality or content method

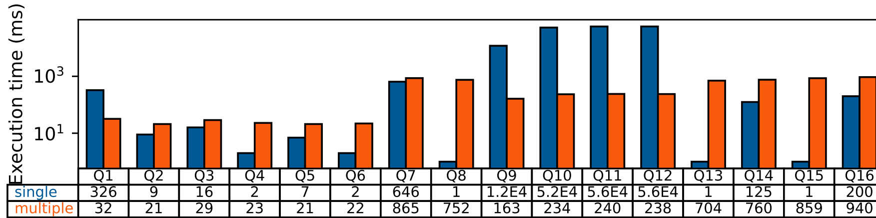

Fig. 5. Execution time for combined failure causes with single or multiple algorithms

Dataset and Queries We downloaded the DBpedia dataset (the English 3.9 version) which contains 812M triple patterns and used conjunctive queries from the LSQ project [22] for our first experiments. Minor adaptations were made as some original queries used URIs incompatible with DBpedia v.3.9. Values for content-based questions were taken from the domain of one of the triple patterns of query in order to have plausible missing answers.

To compare with state of the art methods, we have used the datasets and queries from the initial papers: 7 queries on the LUBM dataset [10] for why empty, 9 queries on DBpedia for why so many, 5 queries with 6 missing mappings on DBpedia for why not.

5.2 Results

Content-Based Problems Figure 4 shows the execution times for finding the MFIS and XSS for 13 why not questions. The card method uses the transformation to an equivalent why empty problem, whereas the content method searches for the expected mapping in the query answers. Dealing with the why not problem by transforming it into a cardinality problem is a significant improvement. Execution times are reduced by up to five orders of magnitude. On average, the card method runs in 7% of the content method's time. For the content method the execution time is related to the size of the result set as determining that an answer is missing requires checking every other answer. In the card method, we only need to know if a query produced an answer, without examining all the answers. By transforming the content problem to a cardinality problem, results unrelated to the mapping of interest are not generated and most of the processing is performed by the underlying triplestore.

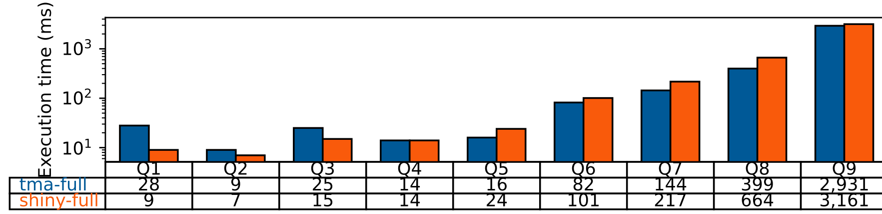

Fig. 6. Execution time for why so many problem with TMA4KB and Shiny algorithms

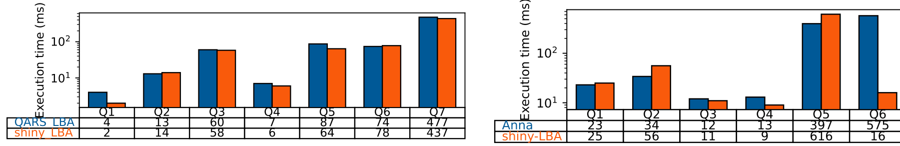

Fig. 7. Execution time for why empty problem with QARS and Shiny algorithms

Fig. 8. Execution time for why not problem with ANNA and Shiny algorithms

Initial query	`SELECT * WHERE {?f dbo:director ?dir . ?f dbo:starring ?dir . ?dir dbp:name ?name . ?f dbp:name ?fname . ?dir dbp:gender Female . ?dir dbp:awards dbr:Academy_Award_for_Best_Picture}`
Mapping	`?fname` → `Argo`
XSS	`SELECT * WHERE {?f dbo:director ?dir . ?f dbo:starring ?dir . ?dir dbp:name ?name . ?f dbp:name ?fname}`
ANNA modification	`SELECT * WHERE {?f dbo:director ?dir . ?f dbo:starring ?dir . ?dir dbp:name ?name . ?f dbp:name ?fname . ?dir ?v0 dbr:Academy_Award_for_Best_Picture}`

Fig. 9. Results for Q6 in the why not experiment

Combined Problems Figure 5 shows the execution times for finding the MFIS and XSS for 16 combinations of two failure causes. All odd numbered executions are disjunctions, and even numbered executions are conjunctions. The single method uses one algorithm execution, using the Base algorithm and leveraging property 6, and determines failure based on the conjunction of failure conditions. The multiple method executes one algorithm per failure condition then combines the results, either combining the MFIS and XSS for a conjuction or the complete lattices for a disjunction. In most cases, the single execution performs better, in particular for conjunctions. When executing the single algorithm, a success with a conjunction is detected faster, since only one failure condition needs to be false for a query to succeed. For queries 9 to 12, the failure condition involves a why not failure condition. In the multiple execution method, this is dealt with by transforming the why not failure condition into a cardinality failure condition, which is a significant performance improvement as shown previously. Here the multiple execution method performs up to three orders of magnitude better.

Comparison with State of the Art Algorithms Figure 6 shows the execution time to identify the MFIS and XSS in the why so many problem, using TMA4KB

[20], and our generic algorithm. Figure 7 shows the time to identify the MFS and XSS using QARS [7], and to find the MFIS and XSS with our algorithm. For the empty answer problem, MFIS and MFS are the same. For the why not problem, figure 8 shows the execution time to produce modified queries using ANNA [26], and to identify the MFIS and XSS with our algorithm. An example of the results returned is given in table 9. The modified answer returned by ANNA is similar to an XSS, but may differ slightly if one of the triple patterns has been relaxed. For all the experiments, the execution times are close, our system is on average 1% slower than ANNA, 13% faster than QARS, and 8% slower than TMA4KB. Overall, the added genericity does not have a major impact on performance.

6 Conclusion

In this paper, we have addressed the problem on unexpected answers in the context of RDF. While the specific problem of empty answers has received much attention in existing work, no solution for a problem with multiple unsatisfactory aspects has yet been proposed. Through the study of specific problems, we have proposed a framework for unexpected answers and identified existing definitions of MFIS and XSS which can be applied to the general problem.

Using existing algorithms for specific problems, we studied adaptations to deal with any unsatisfactory answers. We have shown the benefits of the adaptations experimentally, and compared the performance of our generic method with three specialized algorithms. Our methods perform on average 1% faster than the specialized algorithms execution time, and are faster in 55% of cases.

Our next goal is to use the MFIS in a query modification process. So far, our algorithms only allow modification through triple pattern removal, but the MFIS could help identify triple patterns for relaxation. In the comparison with ANNA, we saw that the XSS provided by our method are similar to the modified queries provided to deal with the why not problem, but that modifying the content of triple patterns can provide a finer tuning of the user's queries.

References

1. Bidoit, N., Herschel, M., Tzompanaki, K.: Query-based why-not provenance with nedexplain. In: Extending database technology (EDBT) (2014). https://doi.org/10.5441/002/edbt.2014.14
2. Bosc, P., Hadjali, A., Pivert, O.: Incremental controlled relaxation of failing flexible queries. Journal of Intelligent Information Systems **33**(3), 261 (2009). https://doi.org/10.1007/s10844-008-0071-6
3. Crama, Y., Hammer, P.L.: Boolean functions: Theory, algorithms, and applications. Cambridge University Press (2011)
4. Dellal, I.: Management and Exploitation of Large and Uncertain Knowledge Bases. Ph.D. thesis, ISAE-ENSMA - Poitiers (2019)
5. Dellal, I., Jean, S., Hadjali, A., Chardin, B., Baron, M.: On addressing the empty answer problem in uncertain knowledge bases. In: DEXA'17. pp. 120–129 (2017). https://doi.org/10.1007/978-3-319-64468-4_9

6. Dong, X., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., Strohmann, T., Sun, S., Zhang, W.: Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In: SIGKDD'14. pp. 601–610 (2014). https://doi.org/10.1145/2623330.2623623
7. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Rdf query relaxation strategies based on failure causes. In: European semantic web conference. pp. 439–454. Springer (2016). https://doi.org/10.1007/978-3-319-34129-3_27
8. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Cooperative techniques for SPARQL query relaxation in RDF databases. In: ESWC'15. pp. 237–252. Springer (2015). https://doi.org/10.1007/978-3-319-18818-8_15
9. Godfrey, P.: Minimization in Cooperative Response to Failing Database Queries. International Journal of Cooperative Information Systems **6**(2), 95–149 (1997). https://doi.org/10.1142/S0218843097000070
10. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Journal of Web Semantics **3**(2-3), 158–182 (2005). https://doi.org/10.1016/j.websem.2005.06.005
11. Harris, S., Seaborne, A.: Sparql 1.1 query language. W3C Recommendation (2013), https://www.w3.org/TR/sparql11-query/.
12. Huang, J., Chen, T., Doan, A., Naughton, J.F.: On the provenance of non-answers to queries over extracted data. Proceedings of the VLDB Endowment **1**(1) (2008). https://doi.org/10.14778/1453856.1453936
13. Jagadish, H.V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A., Yu, C.: Making database systems usable. In: SIGMOD'07. p. 13–24 (2007). https://doi.org/10.1145/1247480.1247483
14. Jannach, D.: Techniques for fast query relaxation in content-based recommender systems. In: Annual Conference on Artificial Intelligence. pp. 49–63. Springer (2006). https://doi.org/10.1007/978-3-540-69912-5_5
15. Kaplan, S.J.: Cooperative responses from a portable natural language query system. Artificial Intelligence **19**(2), 165–187 (1982). https://doi.org/10.1016/0004-3702(82)90035-2
16. 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), 167–195 (2015). https://doi.org/10.3233/SW-140134
17. McSherry, D.: Incremental relaxation of unsuccessful queries. In: European Conference on Case-Based Reasoning. pp. 331–345. Springer (2004). https://doi.org/10.1007/978-3-540-28631-8_25
18. Moises, S.A., Pereira, S.d.L.: Dealing with empty and overabundant answers to flexible queries. Journal of Data Analysis and Information Processing pp. 12–18 (2014). https://doi.org/10.4236/jdaip.2014.21003
19. Motro, A.: Seave: A mechanism for verifying user presuppositions in query systems. ACM Transactions on Information Systems (TOIS) **4**(4), 312–330 (1986). https://doi.org/10.1145/9760.9762
20. Parkin, L., Chardin, B., Jean, S., Hadjali, A., Baron, M.: Dealing with plethoric answers of sparql queries. In: DEXA'21. pp. 292–304. Springer (2021). https://doi.org/10.1007/978-3-030-86472-9_27
21. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3) (2009). https://doi.org/10.1145/1567274.1567278
22. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: The Linked SPARQL Queries Dataset. In: ISWC'15. pp. 261–269 (2015). https://doi.org/10.1007/978-3-319-25010-6_15

23. Song, Q., Namaki, M.H., Wu, Y.: Answering why-questions for subgraph queries in multi-attributed graphs. In: ICDE'19. pp. 40–51 (2019). https://doi.org/10.1109/ICDE.2019.00013
24. Tran, Q.T., Chan, C.Y.: How to conquer why-not questions. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (2010). https://doi.org/10.1145/1807167.1807172
25. Vasilyeva, E., Thiele, M., Bornhövd, C., Lehner, W.: Answering "why empty?" and "why so many?" queries in graph databases. JCSS **82**(1), 3–22 (2016). https://doi.org/10.1016/j.jcss.2015.06.007
26. Wang, M., Liu, J., Wei, B., Yao, S., Zeng, H., Shi, L.: Answering why-not questions on SPARQL queries. Knowledge and Information Systems **58**, 169–208 (2019). https://doi.org/10.1007/s10115-018-1155-4
27. Webber, B.L., Mays, E.: Varieties of user misconceptions: Detection and correction. In: IJCAI'83. vol. 2, pp. 650–652 (1983)
28. Woodruff, A., Stonebraker, M.: Supporting fine-grained data lineage in a database visualization environment. In: ICDE. pp. 91–102. IEEE (1997). https://doi.org/10.1109/ICDE.1997.581742