Cooperative Techniques for Dealing with Unsatisfactory Answers in RDF Knowledge Bases

Louise Parkin
supervised by Brice Chardin, Stéphane Jean, Allel Hadjali
LIAS - ISAE/ENSMA
Chasseneuil-du-Poitou, France
louise.parkin@ensma.fr

ABSTRACT

When querying Knowledge Bases, 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. These unhelpful responses have been categorized into five types that consider either the number of results returned or the content of the results. These problems have been studied individually, and a category of proposed solutions involves modifying the original query in order to produce answers better suited to the user requirement. Similarities between the five problems suggest that techniques developed for one problem could be used to improve the treatment of the others. The goal of this PhD thesis is to propose a unified framework to deal with unexpected or unsatisfactory answers. In this paper, we analyse the state of the art on the unexpected answer problems and discuss the similarities between them, present our first contribution towards generalising the problems and present our future work.

PVLDB Reference Format:
Louise Parkin. Cooperative Techniques for Dealing with Unsatisfactory Answers in RDF Knowledge Bases. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

A *Knowledge Base (KBs)* 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 fields. Well known examples of KBs include *DBpedia* [20], and *LinkedGeoData* [2]. These KBs store information using RDF triples (subject, predicate, object) and are queried using the *SPARQL* language [14]. KBs typically contain billions of facts and are often structured using an ontological schema and rules, such as those provided by *RDFS* [9] or *OWL* [3].

A new end user querying a KB is often unfamiliar with the KB's structure and the data within it. As such, *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 represent the difference between a user's view of a KB, and its reality [33]. For instance if in a hospital database, 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 that the property *birthPlace* uniquely describes 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 one of the challenges to database system usability [17].

There are five basic types of unexpected or unsatisfactory answer problems, each associated with a why-question.

1. The query returns no answers (*why-empty*).
2. The query returns too few answers (*why-so-few*).
3. The query returns too many answers (*why-so-many*).
4. An expected answer is missing from the result (*why-not*).
5. An unwanted answer is included in the result (*why-so*).

In the first three cases, the issue is with the number of answers, and in the last two cases it is with the content of the answers. In all cases, the user does not understand the result of their query, and is usually forced to modify their query through frustrating trial and error, before achieving the desired output.

In every unexpected answer problem, the answer produced by executing a query does not meet a user requirement. We use the formalism of Perez et al. [27] for the result set of a query Q executed on a triplestore D: $[[Q]]_D$. For each type of problem, we can define a boolean function on the query answer, such that a query either succeeds (the answers meet the user requirement) or fails (the answers do not meet the user requirement). It is important to note that while query failure has been used in literature to refer to a query producing no answers, our definition of *query failure* relies on the boolean property specific for each type of unexpected answer problem. For instance, for the *why-so-few* problem, with a threshold of K answers, the boolean failure property is $|[[Q]]_D| < K$, which means a query fails if it produces fewer than K answers, and succeeds otherwise. In the *why-so* problem, with an unwanted answer mapping μ, the boolean property is $\mu \notin [[Q]]_D$, which means that a query fails if μ is included in its answers and succeeds otherwise.

There are links between the five problems. The *why-not* and *why-so* have strictly opposite failure conditions. This is also the case for the *why-so-few* and *why-so-many* problems. The *why-empty* problem is also an extreme case of the *why-so-few* problem, with a threshold for insufficient answers set to 0. Furthermore, some problems require a combination of failure conditions. This is the case for cardinality problems where the number of answers must

Proceedings of the VLDB 2021 PhD Workshop, August 16th, 2021. Copenhagen, Denmark. Copyright (C) 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2971/paper09.pdf

be between two limits, requiring a combination of *why-so-few* and *why-so-many*. Content-based problems may also need combining when a user wants to include some answers and exclude others.

Several studies have considered these problems separately, but a unified framework to deal with unsatisfactory answers has yet to be proposed. Our goal is to build upon the ideas proposed for individual problems, and produce a unified explanation and rewriting strategies for any query producing unexpected answers.

2 STATE OF THE ART

Our work falls in the scope of cooperative query answering, where the focus is on making a database response understandable and useful to users, even if they are unfamiliar with its content and technical requirements. Motro categorized cooperative query answering methods into two groups [24]. The first seeks to aid users when they formulate their requirements and queries, and includes methods such as example queries [25, 35]. The second category analyses user queries to detect anomalies. The latter methods can be used when a user is not satisfied with their query's answer. Existing work has two main focuses: *answer explanation*, and *query modification*.

2.1 Answer explanation

Answer explanations can either be based on the data manipulated, or on the query formulated by a user.

Data based explanations exist for the *why-not* and *why-so* problems. In the case of *why-so*, several techniques can be used to identify the provenance of a piece of data [34]. In the *why-not* problem, a data-based explanation can identify operations on a database that lead to some missing information [16]. Since data based methods do not consider the user query, they cannot help to fix it if it contains a mistake or misconception. In that case, query based explanations are particularly useful, as the root cause of unsatisfactory answers is the user's lack of understanding of the database's structure, content, or query process.

Failure causes were introduced to deal with the *why-empty* problem [19, 23]. Rather than return a potentially misleading answer, users are provided with the *false presuppositions* included in their query. This led to the definition of Minimal Failing Subqueries (MFS) in relational databases [13]. The MFS are the smallest parts of a query that cause it to fail. They have then been used in the context of certain and uncertain KBs [10, 11]. Query-based failure causes have also been considered for the *why-not* problem, first in relational databases [5] then in KBs [32]. In the RDF context, a divide-and-conquer approach is used, first studying the query's triple patterns, then its SPARQL operators. A failure cause shows users which triple pattern or operator causes an answer to be absent. To our knowledge, no query based failure causes have been studied for the *why-so-many*, *why-so-few* and *why-so* problems.

2.2 Query modification

Query modification involves rewriting a query submitted by the user in order to remove the mistakes and misconceptions it may contain. This step is successful if the results of the modified query are better suited to the user requirements. There are three ways to redefine a query: relaxation (the new query's answers contain the answers of the previous query), refinement (the new query's answers are contained in the answers of the previous query) or modification (the new query's answers contain some of the answers of the previous query and add others). The query rewriting process can have varying user involvement, from simply providing users with explanations and leaving rewriting up to them, to a purely automatic process where the user is absent, through interactive processes where users guide the process by providing preferences.

For the *why-not* problem in relational databases, the ConQueR system [30] modifies queries to obtain the missing answer as part of the results of the new query. A similarity metric based on editing distance, and imprecision metric are jointly used to rank modified queries. First, the system attempts to find a modified query where only the selection predicates are changed, and if none exist then modifies other parts of the query. In the field of knowledge graphs, recent work on the *why-not* and *why-so* problems has proposed exact algorithms and heuristics to refine a user's query [29]. They incrementally modify a query in order to obtain some of the answers originally missed. They also define metrics to measure the rewriting cost, and the answer closeness or precision.

For the plethoric answers problem (*why-so-many*), in the field of fuzzy queries, intensification strategies are used to strengthen patterns present in the user query to make them more restrictive [6, 22]. Alternatively, new patterns are added to the query [8]. 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 graph queries, a method based on removing elements of the initial query finds the largest parts of the query that succeed. They are called maximum common connected subgraphs (MCCS) and are provided to the user as alternative non failing queries [31]. A similar notion is considered in the empty answers problem (*why-empty*), called maXimal Succeeding Subqueries. They are the largest succeeding queries obtained by removing parts of the original query [13].

The MFS defined for the empty answers problem have been used as part of several automated query modification systems dealing with relational databases [7, 18]. In the context of RDF, the additional cost of computing MFS has been weighed against the time saved by avoiding executing queries that are known to fail once the MFS are known, and a hybrid method computing the MFS at key points in the relaxation has been presented [11]. This work uses an existing measure of similarity between queries based on information content measure [15]. Another use of MFS in the empty answer problem is as part of an interactive query rewriting framework [18, 21]. At each step in the query relaxation users choose the parts to be relaxed. These works show that the efficiency of query modification methods can be improved if an answer explanation step has been previously performed.

3 FAILURE CAUSE DEFINITION

In existing work, attempts to explain query failure and to modify queries have been suggested for individual problems. Starting by identifying failure causes can improve the performance of query modification strategies. However, this failure cause identification step has not been studied for all the unexpected answer problems. As such, the existing query modification methods rely on trial and error. Our first goal is to establish a method of identifying failure

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

Figure 1: Query Q

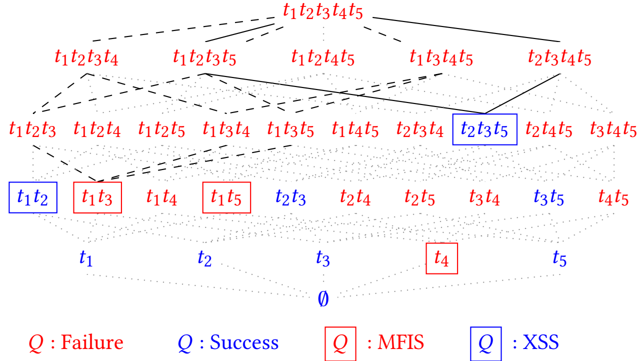

Figure 2: Lattice of subqueries of Q

causes that will support any unexpected answer problem. Our first contribution is the extension of the definition of failure causes used for the empty answers problem to the plethoric answers problem.

Several existing approaches rely on decomposing an initial failing query into parts to identify failure causes. In both the *why-empty* problem [13] and the *why-not* problem [32] conjunctive queries are decomposed by removing triple patterns incrementally. We have used this decomposition of SPARQL queries as the base of our approach. From an original query Q given in figure 1, written as a conjunction of its triple patterns t_1 *AND* t_2 *AND* t_3 *AND* t_4 *AND* t_5 or $t_1t_2t_3t_4t_5$ for short, we build its lattice of subqueries as shown in figure 2. From this decomposition of the initial query, the next step is defining failure causes. To that end, a boolean condition indicating query failure is defined for every query in the lattice of subqueries. Its exact definition depends on the unexpected answer problem we are dealing with.

A widely used notion in the empty answers problem, is the minimal failing subquery (MFS) [4, 7, 10, 11, 13, 18, 19, 21, 23]. MFS are subqueries that fail, that have no failing subqueries. In the empty answers problem, when dealing with conjunctive queries, Godfrey showed that if a query fails (in this case *fails* means *produces no answers*) then all its superqueries fail. In this case, we say that the failure condition is monotonic. With a monotonic failure condition, any query containing an MFS fails, so MFS are a good way to describe failure causes.

When studying the plethoric answers problem, we noted that its failure condition (a query produces more than a threshold K answers) is not monotonic. This is also the case for the insufficient answers problem, whose failure condition is that a query produces fewer than a threshold K answers. For a problem whose failure condition is not monotonic, a failing query can have a succeeding superquery. As such, an MFS can be a part of a succeeding query so the notion of MFS is no longer adequate to describe a failure cause. We have introduced a new concept, that of minimal failure inducing subquery (MFIS) to describe failure causes in cases where the failure cause is not monotonic. A failure inducing subquery (FIS) of a query Q is one of its subqueries Q' such that all subqueries of Q that are superqueries of Q' fail. An MFIS is then defined as an FIS that does not have any other FIS as a subquery. In the case of a monotonic failure condition, the definitions of MFIS and MFS are strictly equivalent. Since this definition of failure cause no longer relies on a monotonic failure condition it can be used in any unsatisfactory answer problem.

The dual notion of MFS which was also introduced to deal with the empty answers problem, is called maximum succeeding subquery (XSS). An XSS of a query Q is one of its succeeding subqueries that is not a subquery of any other succeeding query. The concept of XSS does not depend on the monotony of the failure condition, so it can also be used in every answer problem. In terms of query failure, the set of MFIS and the set of XSS provide the same information. They do not necessarily determine the success or failure of every subquery of the initial query, but do identify all failure causes (parts of a query that if present mean that the query will fail). There can be exponentially many MFS and XSS [13], and we have shown that is also the case for MFIS. So no algorithm can compute MFIS and XSS in polynomial time in the worst case.

For the plethoric answers problem, we have shown that leveraging query and data properties can improve performance in certain cases [26]. These properties allow us to determine that a query fails without executing it, based on the failure of another query. They require conditions based on the variables contained in the queries, or the cardinalities of predicates. We created improved algorithms based on these properties and implemented them in Java with three triplestores (JenaTDB, Jena Fuseki, and Virtuoso). These algorithms have been experimentally evaluated first using synthetic data and queries from the WatDiv benchmark [1], then real data from DBpedia and query logs from the Linked SPARQL Queries Dataset project [28]. Our two algorithms ran respectively 36% and 44% faster than a baseline algorithm which executes all subqueries. Our implementation is available at https://forge.lias-lab.fr/projects/tma4kb with a tutorial to reproduce our experiments. Previous work on the empty answers problem in the RDF context produced similar results. It showed that query properties can be used to compute MFS and XSS effectively [10, 11]. These experimental results are promising in regards to the applicability of an approach based on failure causes to deal with unexpected answer problems.

4 FUTURE WORK

Our work so far has involved taking a method used for the why-empty problem and apply it to the plethoric answers problem. The existing notions were too restrictive for this new problem, so we introduced some new definitions and properties. We end up with a method applicable to both problems, which we have shown is usable in practise.

Our goal is to build a framework identifying failure causes in unsatisfactory answer problems, through the following steps:

(1) to determine the search space, i.e. the set of queries under consideration,

(2) to establish a partial order relationship between the queries in the search space,
(3) to define a failure condition, a boolean property expressed over the set of queries,
(4) to provide a failure cause definition, i.e. the MFIS,
(5) to develop particular inference rules describing the links between queries and query failure.

Currently, the search space is the lattice of subqueries, which contains the original query and its subqueries obtained by removing patterns. In the long run, this search space can be extended by considering queries with additional triple patterns or modified triple patterns from the original query. The partial order is also related to triple pattern inclusion. This supposes that the queries considered are conjunctive queries, which means that they do not contain SPARQL operators such as UNION, OPTIONAL, FILTER. This hypothesis is used in some existing solutions[11, 13], but it is a significant restriction, as a study of SPARQL queries has shown that a large proportion of SPARQL queries contain these operators [12]. We have therefore started the extension of subquery definition to support these operators.

For each problem, the failure condition is linked to the result set of a query $[[Q]]_D$ and considers either its cardinality or its content. We have shown that the MFIS can be used as failure causes for the empty answers and plethoric answers problems, and we are working on applying them to the other three problems. The last step of determining inference rules will allow us to determine the success or failure of a query based on the success or failure of one of the queries it is linked to. This can make the computation of MFIS and XSS much more efficient.

After completing the answer explanation step, we will consider query modification based on these explanations, drawing from existing query rewriting algorithms. We plan to offer users a varying degree of participation in the query modification. Expert users can simply be provided with failure causes and fix their queries themselves, whereas more novice users may prefer an automatic approach where the modified query is computed with minimal input from them. Evaluation of the query modification will be performed with various metrics introduced in existing work, such as similarity, imprecision [32], answer closeness and editing cost [29].

REFERENCES

[1] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management systems. In *ISWC'14*. Springer, 197–212.
[2] Sören Auer, Jens Lehmann, and Sebastian Hellmann. 2009. LinkedGeoData: Adding a Spatial Dimension to the Web of Data. In *ISWC'09*. Springer.
[3] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. 2004. OWL Web Ontology Language Reference. *W3C Recommendation* (2004). https://www.w3.org/TR/owl-ref/.
[4] Chourouk Belheouane, Stéphane Jean, Hamid Azzoune, and Allel Hadjali. 2019. Cooperative treatment of failing queries over uncertain databases: a matrix-computation-based approach. *Journal of Intelligent Information Systems* 52, 1 (2019), 211–238.
[5] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Query-based why-not provenance with nedexplain. In *Extending database technology (EDBT)*.
[6] Patrick Bosc, Allel HadjAli, and Olivier Pivert. 2006. About overabundant answers to flexible queries. In *IPMU'06*, Vol. 6. 2221–2228.
[7] Patrick Bosc, Allel Hadjali, and Olivier Pivert. 2009. Incremental controlled relaxation of failing flexible queries. *Journal of Intelligent Information Systems* 33, 3 (2009), 261.
[8] Patrick Bosc, Allel Hadjali, Olivier Pivert, and Grégory Smits. 2010. Une approche fondée sur la corrélation entre prédicats pour le traitement des réponses pléthoriques.. In *EGC'10*. 273–284.
[9] Dan Brickley and R.V. Guha. 2014. RDF Schema 1.1. *W3C Recommendation* (2014). https://www.w3.org/TR/rdf-schema/.
[10] Ibrahim Dellal, Stéphane Jean, Allel Hadjali, Brice Chardin, and Mickaël Baron. 2017. On Addressing the Empty Answer Problem in Uncertain Knowledge Bases. In *DEXA'17*. 120–129.
[11] Géraud Fokou, Stéphane Jean, Allel Hadjali, and Mickaël Baron. 2016. RDF query relaxation strategies based on failure causes. In *European semantic web conference*. Springer, 439–454.
[12] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. 2011. An Empirical Study of Real-World SPARQL Queries. In *Proceedings of the USEWOD workshop co-located with WWW'11*.
[13] Parke Godfrey. 1997. Minimization in Cooperative Response to Failing Database Queries. *International Journal of Cooperative Information Systems* 6, 2 (1997), 95–149.
[14] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language. *W3C Recommendation* (2013). https://www.w3.org/TR/sparql11-query/.
[15] Hai Huang, Chengfei Liu, and Xiaofang Zhou. 2012. Approximating query answering on RDF databases. *World Wide Web* 15, 1 (2012), 89–114.
[16] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F Naughton. 2008. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment* 1, 1 (2008), 736–747.
[17] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. 2007. Making Database Systems Usable. In *SIGMOD'07*. 13–24.
[18] Dietmar Jannach. 2006. Techniques for fast query relaxation in content-based recommender systems. In *Annual Conference on Artificial Intelligence*. Springer, 49–63.
[19] S Jerrold Kaplan. 1982. Cooperative responses from a portable natural language query system. *Artificial Intelligence* 19, 2 (1982), 165–187.
[20] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.
[21] David McSherry. 2004. Incremental relaxation of unsuccessful queries. In *European Conference on Case-Based Reasoning*. Springer, 331–345.
[22] Samyr Abrahão Moises and S do L Pereira. 2014. Dealing with empty and overabundant answers to flexible queries. *Journal of Data Analysis and Information Processing* (2014), 12–18.
[23] Amihai Motro. 1986. Seave: A mechanism for verifying user presuppositions in query systems. *ACM Transactions on Information Systems (TOIS)* 4, 4 (1986), 312–330.
[24] Amihai Motro. 1996. Cooperative database systems. *International Journal of Intelligent Systems* 11, 10 (1996), 717–731.
[25] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2014. Exemplar queries: Give me an example of what you need. *Proceedings of the VLDB Endowment* 7, 5 (2014), 365–376.
[26] Louise Parkin, Ibrahim Dellal, Brice Chardin, Stéphane Jean, and Allel Hadjali. 2020. A Cooperative Approach to Address the Overabundant Answers Problem in RDF Knowledge Bases. In *36ème Conférence sur la Gestion de Données – Principes, Technologies et Applications*.
[27] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3, Article Article 16 (2009), 45 pages.
[28] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: The Linked SPARQL Queries Dataset. In *ISWC'15*. 261–269.
[29] Q. Song, M. H. Namaki, and Y. Wu. 2019. Answering Why-Questions for Subgraph Queries in Multi-attributed Graphs. In *ICDE'19*. 40–51.
[30] Quoc Trung Tran and Chee-Yong Chan. 2010. How to conquer why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 15–26.
[31] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. 2016. Answering "why empty?" and "why so many?" queries in graph databases. *J. Comput. System Sci.* 82, 1 (2016), 3–22.
[32] Meng Wang, Jun Liu, Bifan Wei, Siyu Yao, Hongwei Zeng, and Lei Shi. 2019. Answering why-not questions on SPARQL queries. *Knowledge and Information Systems* 58 (2019), 169–208.
[33] Bonnie Lynn Webber and Eric Mays. 1983. Varieties of user misconceptions: Detection and correction. In *IJCAI'83*, Vol. 2. 650–652.
[34] Allison Woodruff and Michael Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings 13th International Conference on Data Engineering*. IEEE, 91–102.
[35] Moshe M. Zloof. 1977. Query-by-example: A data base language. *IBM systems Journal* 16, 4 (1977), 324–343.