A Cooperative Treatment of the Plethoric Answers Problem in RDF

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

▶ To cite this version:

Louise Parkin, Brice Chardin, Stéphane Jean, Allel Hadjali, Mickael Baron. A Cooperative Treatment of the Plethoric Answers Problem in RDF. Knowledge and Information Systems (KAIS), 2022, 64 (9), pp.2481-2514. 10.1007/s10115-022-01710-8 . hal-03768027

HAL Id: hal-03768027
https://hal.science/hal-03768027v1

Submitted on 2 Sep 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.

The Version of Record of this article is published in Knowledge and Information Systems, and is available online at https://doi.org/10.1007/s10115-022-01710-8.

A Cooperative Treatment of the Plethoric Answers Problem in RDF

Louise Parkin · Brice Chardin · Stéphane Jean · Allel Hadjali · Mickael Baron

Received: 22 Mar 2021 / Revised: 15 Apr 2022 / Accepted: 28 May 2022

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 the user query produces significantly more results than the user was expecting. The common approach to solving this problem, i.e. the top-K approach, reduces the query's result size by applying various criteria to select only some answers. This selection is performed without considering the causes producing plethoric answers, and can therefore miss an underlying issue within the query.

We deal with this problem by proposing an approach that identifies 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. Thus, thanks to these MFIS, interactive and automatic approaches can be set up to help the user in reformulating their query. The dual notion of MFIS, called *Maximal Succeeding Subqueries (XSS)*, is also useful. They provide queries with a maximal number of 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

L. Parkin
LIAS, ISAE-ENSMA
E-mail: louise.parkin@ensma.fr

B. Chardin
LIAS, ISAE-ENSMA
E-mail: brice.chardin@ensma.fr

S. Jean
LIAS, Université de Poitiers
E-mail: stephane.jean@ensma.fr

A. Hadjali
LIAS, ISAE-ENSMA
E-mail: allel.hadjali@ensma.fr

M. Baron
LIAS, ISAE-ENSMA
E-mail: mickael.baron@ensma.fr

show that computing this information is an NP-hard problem. Thus a baseline exhaustive search method cannot be used for most queries. We propose two algorithms that leverage properties of queries and data to compute MFIS and XSS efficiently for queries of reasonable size. We show experimentally that our two algorithms clearly outperform a baseline method on generated queries as well as real user-submitted queries.

Keywords Knowledge bases · RDF data · SPARQL queries · Plethoric answers · MFIS · XSS

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. Well known examples of KBs include *DBpedia* (Lehmann et al., 2015), and *LinkedGeoData* (Auer et al., 2009). These KBs store information using RDF triples (subject, predicate, object) and are queried with the *SPARQL* language (Harris and Seaborne, 2013) 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* (Brickley and Guha, 2014) or *OWL* (Bechhofer et al., 2004).

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 (Webber and Mays, 1983). 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* will 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 (Jagadish et al., 2007). There are five types of unexpected 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 or desired answer is missing from the result (*why-not*).
5. An unexpected or unwanted answer is included in the result (*why-so*).

We focus here on the the third problem, also referred to as the plethoric answers problem, where users struggle to extract useful information from an overwhelming result. A query's result is said to be plethoric when it contains more than K answers. This threshold, K, can be user-defined and expressed within the query or set by default by the SPARQL endpoint. A study of DBpedia queries collected over two months in 2010 (Saleem et al., 2015) shows that over ten thousand queries

with non empty answers return more than 100 answers, which is the default limit for the DBpedia SPARQL endpoint.

State-of-the-art methods to tackle the plethoric answers problem rely mainly on ordering or categorizing 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 that the number of answers will be less than or equal to K. Yet, if a query is based on a misconception from the user, no ordering or classification 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 the query produces a plethoric answer. Our failure causes can be directly provided to users, in an effort to educate them in formulating their queries, or more generally whenever a user wants to understand why their query produced plethoric answers. They can also be used as a basis for automatic or interactive query rewriting, in order to avoid suggesting queries which are known to fail. By pinpointing the parts of the query that need refining, this method can accelerate the process of query rewriting. The identification of failure causes has been studied in the context of the empty-answers problem, using the notion of Minimal Failing Subqueries (MFS). This notion is not directly relevant to the plethoric answers problem, as the presence of an MFS does not necessarily cause a query to fail. Therefore a new definition of failure causes is necessary to deal with plethoric answers.

Drawing on previous work on the empty-answers problem in KBs (Fokou et al., 2015), we have introduced the basis of a cooperative method to deal with the plethoric answers problem in a previous paper (Parkin et al., 2021). In this paper we provide an extended presentation of our method with the following contributions.

1. We provide two cooperative notions that can be used to help with query rewriting: the smallest subqueries that cause plethoric answers (Minimal Failure Inducing Subqueries, or MFIS) and the largest subqueries which do not produce plethoric answers (maXimal Succeeding Subqueries, or XSS). We show that the enumeration of MFIS and XSS is an NP-hard problem.
2. We propose an algorithm to compute MFIS and XSS, using query properties to avoid executing some subqueries. This algorithm is further improved using predicate cardinalities, a data property, when such information is available.
3. We evaluate our algorithms extensively, using three triplestore implementations, with queries generated for the *WatDiv* synthetic dataset (Aluç et al., 2014), and user-submitted DBpedia queries from the *Linked SPARQL Queries Dataset* logs (Saleem et al., 2015).

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 formalize our problem in section 4. Section 5 presents our approaches to calculate MFIS and XSS. Section 6 shows how SPARQL operators can be accounted for. Section 7 describes the experimental evaluation of our algorithms. We conclude and introduce future work in section 8.

2 Motivating Example

For our example, we consider a hospital KB, and a user wanting information on the relationships between doctors, nurses and patients. We show in figure 1b an

subject	predicate	object
d_1	experience	14
d_1	supervises	n_3
d_1	supervises	n_2
d_1	treats	p_1
d_1	treats	p_2
d_2	experience	25
d_2	supervises	n_2
d_2	treats	p_3
d_3	supervises	n_1
n_1	type	SurgicalNurse
n_1	providesCare	p_1
n_1	operativeRole	Instrument
n_2	type	ERNurse
n_2	providesCare	p_2
n_2	providesCare	p_3
n_3	type	ERNurse
n_3	providesCare	p_2
n_3	providesCare	p_3

(a) Database D

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

(b) Query $Q = t_1t_2t_3t_4t_5$

?d	?p	?e	?n	?pt
d1	p_1	14	n_3	p_3
d1	p_2	14	n_3	p_3
d1	p_1	14	n_3	p_2
d1	p_2	14	n_3	p_2
d1	p_1	14	n_2	p_2
d1	p_2	14	n_2	p_2
d1	p_1	14	n_2	p_3
d1	p_2	14	n_2	p_3
d2	p_3	25	n_2	p_2
d2	p_3	25	n_2	p_3

(c) Evaluation of Q on D

Fig. 1 A KB, a SPARQL query and its results

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 $t_1t_2t_3t_4t_5$ in short), and in figure 1a a simplified KB D. When executing Q on our dummy KB (containing only three doctors and four patients), the query produces 8 results. They are shown in figure 1c. Using a real KB containing a hundred doctors, with tens of patients each, this query would produce thousands of results, which would be unmanageable 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. We can apply a top-K method to reduce the number of results. Several ordering strategies could be used, such as ordering doctors alphabetically. In the example, this method would only return information regarding a single doctor, and information concerning a small number of doctors in the real-world situation. Our approach will focus on explaining plethoric answers so that the query can be modified to return fewer answers.

We want to identify failure causes, which are the subqueries that are never included in a subquery which succeeds. Figure 2 shows the number of results of each subquery of the initial query from our example. In this particular example, our solution aims at providing the following concise feedback:

- asking for both patients (variable `p`) and nurses (variable `n`) in the same query produces plethoric answers,
- listing patients (variable *pt*) nurses care for (predicate `providesCare`) gives plethoric results.

Each failure cause must be resolved in order to reach the desired number of answers. In practice, this feedback is provided to the user as subqueries, that is to say as subsets of predicates that, when occurring together in the context of the original query, necessarily produce plethoric answers. These subqueries are called Minimal Failure Inducing Subqueries (MFIS). In this example, the three MFIS

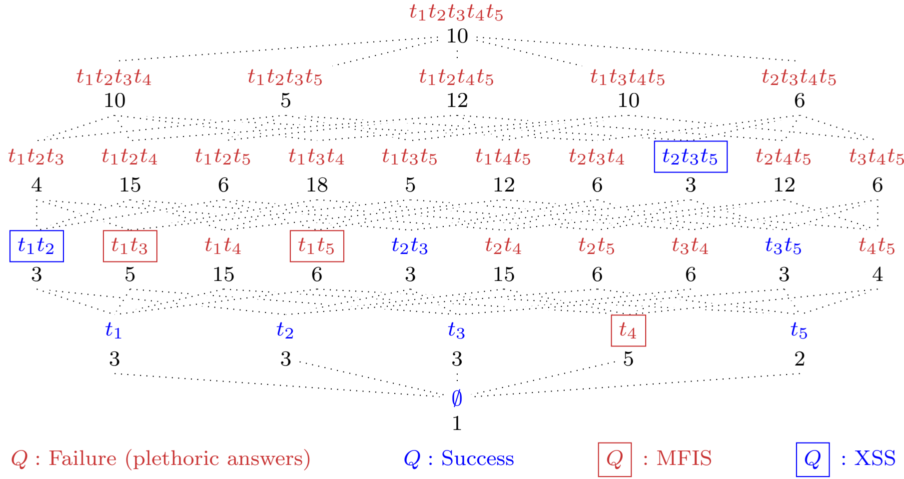


Fig. 2 Number of results of each subquery of Q

are t_1t_3 and t_1t_5, that together support the first aforementioned feedback, and t_4, that supports the second feedback.

To assist the user in their query reformulation process, our failure causes identification method also lists the succeeding queries $t_2t_3t_5$ and t_1t_2, which are not themselves subqueries of any other succeeding query. These queries partially meet the user's original requirement as far as predicates are concerned. They can be used as alternative queries, with confidence that they will have no more than K results.

- $t_2t_3t_5$ lists doctors with their years of experience and the nurses they supervise.
- t_1t_2 returns doctors with their years of experience and the patients they treat.

To conclude on this example, another way of looking at the plethoric answer problem is to suppose the existence of an ideal query which an expert user would have formulated, if they had the same requirements as the novice user. In this case, an ideal query could be replacing variable `p` by `pt` in the first triple pattern (or vice versa). This means that the user did not intend to consider two distinct patients in their query. This ideal query returns three answers in our example, which fits the requirement. This modification adds an extra constraint to triple pattern $t4$, fixing that failure cause, and to triple pattern t_1 fixing the other two failure causes. Subquery $t_2t_3t_5$, listed as an alternative query, is incidentally the common part between the original and reformulated queries.

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 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 main categories: those focusing on data and those focusing on queries.

Data-oriented methods suppose that the query submitted by the user is correct, and seek to 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 (Ilyas et al., 2008). In the absence of user preferences, alternative data-oriented strategies have been proposed. Regret-minimization strategies combine features from top-K and skyline methods (Xie et al., 2020). 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 the common features of each category are provided to the user (Chakrabarti et al., 2004; Ozawa and Yamada, 1995). Data-oriented methods can be useful to sort through large result sets, providing the initial query correctly matches the user's requirement. 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. There are several query modification techniques. In the field of fuzzy queries, intensification strategies are used to strengthen patterns present in the user's query to make them more restrictive (Bosc et al., 2006; Moises and Pereira, 2014). Alternatively, new patterns are added to the query (Bosc et al., 2010). 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 (Song et al., 2019). 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 (Vasilyeva et al., 2016). 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 answer problems. Wang et al. have proposed failure causes for the why-not problem (Wang et al., 2019). They use a divide-and-conquer approach, first studying a query's triple patterns and then its SPARQL operators. A failure cause shows users which triple pattern or operator causes an answer to be absent. Godfrey (Godfrey, 1997) proposed using failure causes (called MFS) and alternate subqueries (called XSS) for the empty answer problem. He suggested providing this information directly to the user, who would be left to interpret it. Another use of MFS in the empty answer problem is as part of an interactive query rewriting framework (Jannach, 2006; McSherry, 2004). At each step in the query relaxation users choose the parts to be relaxed. MFS have also been used in automatic query relaxation to accelerate the process, by pruning the search space of queries which necessarily fail (Bosc et al., 2009; Fokou et al., 2016; Jannach, 2006). We propose extending the definitions

of MFS and XSS, to deal with the plethoric answers problem in the context of RDF KBs.

4 Formalization

We describe the formalism and semantics of RDF and SPARQL necessary for this paper, using the notations and definitions provided by Pérez et al. (Pérez et al., 2009).

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) $\in (I \cup B) \times I \times T$. An *RDF database* (or *triplestore*) stores a set of RDF triples. We additionally consider V a set of variables disjoint from T.

Conjunctive Queries A triple t (subject, predicate, object) $\in (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is called a *triple pattern.* We denote by $s(t)$, $p(t)$, $o(t)$, and $var(t)$ the subject, predicate, object and variables of t. In the first part of this paper, we will study *RDF queries* defined as conjunctions of triple patterns $Q = SELECT * WHERE\ t_1\ AND \cdots AND\ t_n$, which we write as $Q = t_1 \wedge \cdots \wedge t_n$, or, with conjunction symbols omitted, $t_1 \cdots t_n$ for short. The variables of a query are $var(Q) = \bigcup var(t_i)$.

To study parts of an initial query, we define a partial order on queries based on the subset partial ordering of their graph patterns. Given $Q = t_1 \cdots t_n$, $Q' = t_i \cdots t_j$ is a *subquery* of Q, denoted by $Q' \subseteq Q$, iff $\{t_i, \cdots, t_j\} \subseteq \{t_1, \cdots, t_n\}$. Then Q is a *superquery* of Q'. *Direct subqueries* of a query Q are $Q' \mid Q' \subset Q \wedge \nexists Q'', Q' \subset Q'' \subset Q$.

Query Evaluation A mapping μ from V to T is a partial function $\mu : V \rightarrow T$. Abusing notation, 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(μ), is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* when $\forall x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2),\ \mu_1(x) = \mu_2(x)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. For two sets of mappings Ω_1 and Ω_2, the join of Ω_1 and Ω_2 is: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1,\ \mu_2 \in \Omega_2\ \textit{are compatible mappings}\}$. For an RDF database D and a triple pattern t, the *evaluation* of t over D is defined as $[[t]]_D = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \wedge \mu(t) \in D\}$. For a conjunctive query $Q = t_1 \cdots t_n$, the evaluation of Q over D is defined as $[[Q]]_D = [[t_1]]_D \bowtie \cdots \bowtie [[t_n]]_D$.

SPARQL operators In section 6, we will extend the proposed method to more complex queries. We introduce here the formalization of SPARQL queries including operators, the associated complete definition of subqueries will be given in section 6. SPARQL graph patterns are defined recursively starting with triple patterns

1. A triple pattern is a *graph pattern.*

2. If P_1 and P_2 are graph patterns, then expressions (P_1 OPTIONAL P_2), (P_1 AND P_2), and (P_1 UNION P_2) are graph patterns, where AND, UNION and OPTIONAL are SPARQL operators.
3. If P is a graph pattern and R is a *built-in condition*, then (P FILTER R) is a graph pattern, where FILTER is a SPARQL operator. A built-in condition contains elements of $V \cup I \cup L$, constants, logical connectives, inequality and equality symbols as well as unary predicates such as *bound*. For a graph pattern P FILTER R, we assume $\text{var}(R) \subseteq \text{var}(P)$.

Four types of SPARQL queries are defined in the W3C standard: SELECT, ASK, DESCRIBE, and CONSTRUCT. In this paper we consider only SELECT queries, in the form of *SELECT S WHERE P*, where P is a graph pattern and $S \subseteq \text{var}(P)$.

4.2 Notions of MFIS and XSS

In the plethoric answers problem, for a given threshold K, a failing subquery of a query Q is a query that returns more than K answers. Conversely, a succeeding subquery of a query Q is a query that returns no more than K answers. We introduce a Boolean property of query failure:

$$\text{FAIL}_K(Q) = |[[Q]]_D| > K$$

As the evaluation of the empty query $\emptyset$ returns a single answer mapping no variables (Harris and Seaborne, 2013), it succeeds (if $K > 0$).

To deal with the empty answers problem, Godfrey introduced *Minimal Failing Subqueries (MFS)* (Godfrey, 1997). They are the smallest subqueries that fail, i.e. return no answers. In the empty answer problem, the failure condition is monotonic with respect to the partial order on queries defined in section 4.1, meaning that if a query fails, all its superqueries fail, and if a query succeeds, all its subqueries succeed. With this monotonic property, the MFS are the smallest parts of a query that will cause a failure. This is useful information for query rewriting, so that these parts can be changed.

In the plethoric answers problem, there is no such monotony. A failing query can have a successful superquery. In our running example consider t_2t_5, which fails, but has a succeeding superquery $t_2t_3t_5$. Therefore, the notion of MFS cannot be used directly as a failure cause in the context of plethoric answers, so we introduce the new notion of failure inducing subquery.

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

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

To present information concisely, we consider the minimal failure causes.

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

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

Note that if the failure condition is monotonic, MFIS and MFS are strictly equivalent notions. MFIS are therefore a more general notion that can be used to describe failure causes for other types of unexpected answers problems.

MaXimal Succeeding Subqueries (XSS) were also used by Godfrey (Godfrey, 1997) to deal with the empty answers problem. They represent the queries that succeed that are most similar to the original query, and can be used as alternative queries. The notion of XSS does not rely on the monotny of the failure property, so is applicable to the plethoric answers problem.

Definition 3 A *maXimal Succeeding Subquery (XSS)* of a query Q is one of its succeeding subqueries whose strict superqueries are all FISs. The set of XSS of a query Q is:

$$xss^K(Q) = \left\{ Q^* \;\middle|\; Q^* \subseteq Q \wedge \neg\text{FAIL}_K(Q^*) \wedge \forall Q' \subseteq Q \left(Q^* \subset Q' \Rightarrow Q' \in fis^K(Q) \right) \right\}$$

Despite using concepts introduced by Godfrey (Godfrey, 1997), our work differs in two key aspects. First, we consider a problem in RDF KBs where Godfrey uses relational databases. Second, we are studying the too many answers problem, while he considers the empty answers problem. As our problem does not have a monotonic failure condition, the algorithms employed by Godfrey cannot be used here.

Problem Statement We are concerned with efficiently computing $mfis^K(Q)$ and $xss^K(Q)$ for an RDF query Q and a given threshold K in a KB D.

4.3 Complexity of the MFIS and XSS enumeration problem

The enumeration of MFIS and XSS is hard, first of all because of the potential number of elements to enumerate.

Property 1 The maximum number of MFIS (or XSS) of a query with n triple patterns is $\binom{n}{\lfloor n/2 \rfloor}$, and is reached when all subqueries with a size of $n/2$ are MFIS.

Proof We prove this using Sperner's theorem (Sperner, 1928). An antichain is a family of sets in which no set is a strict subset of another. From the MFIS and XSS definitions, it follows that the set of MFIS (resp. the set of XSS) is an antichain, as no MFIS (resp. XSS) can be a subquery of another MFIS (resp. XSS).

The theorem states that the maximum size of an antichain is bounded by $\binom{n}{\lfloor n/2 \rfloor}$, where n is the maximum number of elements in a set. This maximum size is reached when the antichain consists of all elements of size $\lfloor n/2 \rfloor$. □

Using the Stirling approximation for a factorial, the maximum number of MFIS is approximated by $\binom{n}{\lfloor n/2 \rfloor} \approx \sqrt{\frac{2}{\pi n}} * 2^n$.

We now show that the enumeration of n MFIS of a query with n triple patterns is NP-hard. To that end, we construct a polynomial reduction from the problem of enumerating n MFS, which has been shown to be NP-hard (Godfrey, 1997).

Property 2 The enumeration of n MFIS of a query of size n is NP-hard.

Proof We borrow the abstraction provided by Godfrey, which uses a set $S = \{e_1, ...e_n\}$, and a Turing machine T with the following properties

- T is defined over 2^S, the parts of S.
- T halts and returns yes or no for any input in its domain.
- T runs in polynomial time for any input in its domain.

Furthermore, if for an input A, T returns yes on A implies that for any $B \subseteq S$, where $A \subseteq B$, T returns yes on B, T is called upward closed.

The set of Minimal Elements in a Lattice is defined by Godfrey as $MEL = \{A \subseteq S \mid T \text{ returns yes on } A \wedge \forall B \subset A\,(T \text{ returns no on } B)\}$ for an upward closed Turing machine T. This is an abstraction of the MFS of a query. For non upward closed Turing machines, we propose Generalised Minimal Elements in a Lattice:

$$GMEL = \{A \subseteq S \mid \forall B\,(A \subseteq B \Rightarrow T \text{ returns yes on } B) \wedge \\ \forall C \subset A\,(\exists D\,(C \subseteq D \Rightarrow T \text{ returns no on } D))\}$$

This is an abstraction of the MFIS of a query. Godfrey showed that the enumeration of *MEL* is an NP-hard problem. We show that the enumeration of *GMEL* is also NP-hard using a reduction from *MEL* to *GMEL*.

Consider an *MEL* enumeration problem involving a set S, and an upward closed Turing machine T. The associated *GMEL* enumeration problem also uses S and T, so the transformation is polynomial. We show that, for an upward closed Turing machine, $A \in MEL \iff A \in GMEL$.

If $A \in MEL$, then T returns yes on A. Consider $B, A \subseteq B$, T returns yes on B because of the upward closure of T. Consider $C \subset A$, from the definition of an *MEL*, T returns no on C, so $\exists D, C \subseteq D$ (e.g. $C = D$) where T returns no on D. Therefore $A \in GMEL$ and then $A \in MEL \Rightarrow A \in GMEL$.

If $A \in GMEL$, then T returns yes on A. Consider $B \subset A$, if T returns yes on B then from the upward closure of T, $\forall C, B \subseteq C$ T returns yes on C, which contradicts $A \in GMEL$. So T returns no on B and $A \in MEL$. Therefore, $A \in GMEL \Rightarrow A \in MEL$

In conclusion the enumeration of *GMEL* is NP-hard. □

5 Computing MFIS and XSS

The complexity of the plethoric answers problem means that no polynomial algorithm can compute the MFIS and XSS exactly. While approximation algorithms could be more efficient, they would not provide all failure causes, and therefore would only partially solve the plethoric answers problem. In order to correctly modify queries, we need to know every MFIS. When calculating the MFIS, our algorithm will also determine the XSS, without needing to execute any additional queries. As the XSS are alternative queries, the user could be provided with only the most useful XSS. For now, our algorithms will return all MFIS and XSS to the user if there are several. In future work, we will consider ordering XSS to choose the most relevant ones. In this section, we discuss our strategies for MFIS and XSS computation. First, a baseline algorithm is presented in section 5.1, then improved versions are introduced by leveraging various properties. In section 5.2, we present some general properties which can be used for any queries. Then, in section 5.3 we introduce a new piece of information regarding the data, predicate cardinality, which further improves the algorithm but is only usable if cardinalities are available. Finally, we discuss the complexity of our algorithms in section 5.4.

5.1 Baseline

A baseline approach to calculate all MFIS and XSS is to execute every subquery of the original query, as is shown in figure 2 from the motivating example. In this first algorithm, which we call Base, the lattice of subqueries is explored in a *Breadth-First* order, meaning that we start by executing the query with the largest number of triple patterns. The order of query evaluation is: $t_1t_2t_3t_4t_5$, $t_1t_2t_3t_4$, $t_1t_2t_3t_5$, $t_1t_2t_4t_5$, $t_1t_3t_4t_5$, $t_2t_3t_4t_5$, $t_1t_2t_3$, ... When studying a query, we execute it, store its status (success or failure), then check the status of its superqueries. Failing queries whose superqueries fail replace their superqueries in the set of MFIS, and queries that succeed whose superqueries fail are added to the set of XSS. For an original query with n triple patterns, Base requires $2^n - 1$ query executions (the empty query $\emptyset$ is not executed), which is time-consuming for queries with many triple patterns. In our example 31 queries are executed.

To make the search for MFIS and XSS more efficient, we want to reduce the number of queries executions. Various properties which may be leveraged are presented in the next two sections.

5.2 General properties

A first basic improvement is pruning the search space to avoid considering areas that are irrelevant to the search for MFIS and XSS. We use the following property which is immediately deduced from the definitions of FIS and XSS.

Property 3 (query success) If a subquery Q' succeeds, and $Q'' \subset Q'$, then Q'' is neither an MFIS nor an XSS.

Proof Consider Q, Q' a succeeding subquery of Q, and $Q'' \subset Q'$. Suppose Q'' is an MFIS. Therefore Q'' is also an FIS. From the definition of an FIS, all superqueries of Q'' fail. In particular, Q' fails. This contradicts the hypothesis, so Q'' is not an MFIS. Suppose Q'' is an XSS. From the definition of an XSS, all strict superqueries of Q'' are FIS. In particular Q' is an FIS so Q' fails. Likewise, Q'' is not an XSS. □

Thanks to this property, only subqueries of failing queries will be studied in all our improved algorithms. Next, we propose a deduction rule which will allow us to declare that a query fails without needing to execute it.

5.2.1 Variable-based property

Since we are running a breadth-first-search, a query is always studied after all its superqueries, so we can use properties deducing failure of a query from the failure of its superqueries. We consider removing from a query a triple pattern that does not reduce the number of variables. The intuition is that if the removal of a triple pattern from a query does not remove any variables, then the number of answers cannot decrease. Consider triple pattern t_5: *?n service Emergency* from our running example. Adding this triple pattern to a query (which already contains the variable n) adds a constraint on n, so can only reduce the number of answers.

Property 4 (variable-based) Given a query Q and triple pattern t, if $\text{var}(Q \wedge t) = \text{var}(Q)$ then $Q \wedge t$ fails $\Rightarrow Q$ fails.

```
Var/Full(Q, D, K)
    inputs : A failing query Q = t_1 ∧ ... ∧ t_n
             An RDF database D
             A threshold K
    outputs: MFIS and XSS of Q
1   mfis ← ∅, xss ← ∅, fis ← ∅, queryStatus ← ∅
2   list ← {lattice(Q)}// lattice of subqueries of Q, by decreasing number of triple
     patterns
3   while list ≠ ∅ do
4       Q′ ← first query of list in BFS ordering
5       list ← list − {Q′}
6       parents_fis ← true
7       superqueries ← superQueries(Q)
8       foreach Q″ ∈ superqueries do
9           parents_fis ← parents_fis ∧ ((Q″) ∈ fis)
10      if parents_fis then
11          if Q′ ∉ queryStatus then
12              queryStatus [Q′] ← FAIL_K(Q′)
13          if queryStatus [Q′] then // if Q′ fails
14              fis ← fis ∪ {Q′}
15              mfis ← mfis − superqueries
16              mfis ← mfis ∪ {Q′}
17              foreach Q″ ∈ subQueries(Q′) do
18                  if var(Q″) = var(Q′) then
19                      queryStatus [Q″] ← true
20                  else if cardinality_max(t, Q′) = 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

This property is used if multiple triple patterns connect the same variables, if a triple pattern contains only one variable, or if removing a triple pattern creates a Cartesian product. Its proof is provided in appendix A. Using properties 3 and 4, we devise a new algorithm: VAR.

5.2.2 The VAR *algorithm*

The VAR algorithm is given in algorithm 1 (lines 20 and 21 do not apply, they will be used in the next algorithm). In this algorithm, the two 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 11-12). From the list, we consider queries from the lattice of subqueries in a breadth-first order (lines 2-5). The first improvement over the baseline method is that, according to property 3, every direct superquery of Q' has to be an FIS for further consideration (lines 6-10). On query failure, the sets fis and mfis are updated (lines 14-16). The subquery Q' being considered replaces its direct superqueries in the mfis set as they can no longer be minimal (lines 15-16). We then consider every direct subquery of Q' (lines 17-21),

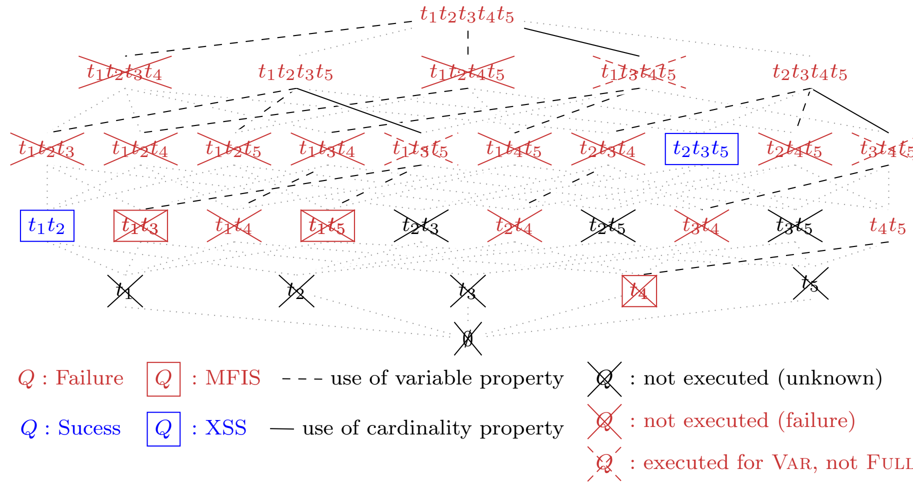


Fig. 3 Lattice of subqueries of Q for the VAR and FULL algorithms

checking if property 4 is applicable (line 18) to predict its status without having to evaluate it. If, instead, the query Q' succeeded, it is added to the xss set (line 23).

We show in figure 3 the lattice of subqueries of query Q from the motivating example and show which subqueries are executed by VAR. Queries avoided because they have a succeeding superquery have an unknown status. Here queries t_2t_3, t_2t_5, t_3t_5, t_2, t_3 and t_5 are not executed because $t_2t_3t_5$ succeeds. Queries avoided by using the variable-based property are known to fail. For example, $t_1t_2t_3t_5$ fails, and has the same variables as $t_1t_2t_3$, so $t_1t_2t_3$ must fail according to property 4. Overall, VAR executes 9 queries, which is a clear improvement over BASE (31 queries).

5.3 Cardinality-based Properties

In the last section, we have used query-based properties to avoid executing certain subqueries. To avoid extra query executions, we introduce a data property called cardinality. We consider triple patterns that add a piece of information to each answer but do not overall change the number of answers. Consider triple pattern t_2: *?d experience ?e* in the running example. If this pattern is removed from a query, as each person has at most one *experience* value, each answer will lose a piece of information, but no two answers will become identical. So the number of answers can only increase. To formalise this notion, we draw upon the concept of predicate cardinality (Dellal, 2019). We start by introducing the definitions of three types of cardinality (global, class, and Characteristic Set), then present properties that can be used to avoid query executions and finally give the complete algorithm in section 5.3.3.

5.3.1 Definitions

Cardinality is a measure of the number of occurrences of a predicate per subject in a dataset. The definitions of minimum and maximum cardinality of a predicate

p for a set of subjects S are (Dellal, 2019):

$$\text{cardinality}_{\min}(p, S) = \min_{s \in S} \text{count}(s, p) = \min_{s \in S} |\{(s, p, o) \mid (s, p, o) \in D\}|$$
$$\text{cardinality}_{\max}(p, S) = \max_{s \in S} \text{count}(s, p) = \max_{s \in S} |\{(s, p, o) \mid (s, p, o) \in D\}|$$

In the following, we focus on maximum cardinalities, and present three methods for calculating them.

Global Cardinality The simplest form of cardinality is globally computed for all the subjects of a dataset D: $\text{subject}(D) = \{s \mid \exists\ p, o : (s, p, o) \in D\}$. The global cardinality of a predicate p is then (Dellal, 2019):

$$\text{global_cardinality}_{\max}(p) = \max_{s \in \text{subject}(D)} \text{count}(s, p)$$

For instance, from the motivating example of figure 1, we can compute the following global cardinalities:

- $\text{global_cardinality}_{\max}(\textit{experience}) = 1$,
- $\text{global_cardinality}_{\max}(\textit{providesCare}) = 3$,
- $\text{global_cardinality}_{\max}(\textit{supervises}) = 2$.

A drawback of global cardinalities is that they are rarely precise. By grouping subjects into classes and calculating cardinalities on each class, we can refine cardinality values.

Class Cardinality RDFS classes can be used to allocate a type to subjects in a dataset. The instances of an RDFS class C in a dataset D are defined as:

$$\text{instances}(C) = \{s \mid \exists(s, \textit{rdfs:type}, C) \in D\}$$

Given a class C and a predicate p, the class cardinality of p in C is (Dellal, 2019):

$$\text{class_cardinality}_{\max}(p, C) = \max_{s \in \text{instances}(C)} \text{count}(s, p)$$

Unlike global cardinalities, class cardinalities are context dependent. In order to use them, we need to define the classes involved. For our application, we are interested in determining predicate cardinalities in the context of a query. We will describe the classes used to calculate cardinalities based on information contained in the query. We start with the notion of predicate domain defined in RDFS. For a triple pattern t with a variable subject s(t), a mapping of s(t) belongs to the *rdfs:domain* of p(t). In KBs, *saturation* insures compatibility between the *rdfs:type* and *rdfs:domain* properties, by inferring potentially missing triples. The standard states that if a property has multiple values for *rdfs:domain*, its domain is their intersection, and is therefore defined as:

$$\text{domain}(p) = \bigcap_{C \mid \exists(p, \textit{rdfs:domain}, C) \in D} C$$

Within the context of a query Q, the domain of the predicate $p(t)$ of a triple pattern t can be further refined from every triple pattern of Q sharing the same subject. We then define the class cardinality of the predicate $p(t)$ as the minimum

subject	predicate	object
providesCare	rdfs:domain	Nurse
operativeRole	rdfs:domain	SurgicalNurse
SurgicalNurse	rdfs:subClassOf	Nurse

(a) Data model of D in RDFS

```
SELECT * FROM {
  ?p supervises ?s .      # t1
  ?s providesCare ?c .    # t2
  ?s operativeRole ?r }   # t3
```

(b) Example query Q_1

```
SELECT * FROM {
  ?p supervises ?s .      # t1
  ?s providesCare ?c }    # t2
```

(c) Example query Q_2

Fig. 4 Database extension for class cardinalities

cardinality among the classes involved in each triple pattern of Q that share the same subject, since $s(t)$ must be an instance of every one of those classes.

$$\text{domain}(t, Q) = \bigcap_{(s,p,o)\in Q \wedge s(t)=s} \text{domain}(p)$$

$$\text{class_cardinality}_{\max}(t, Q) = \min_{C \subseteq \text{domain}(t,Q)} \text{class_cardinality}_{\max}(p, C)$$

Example 1 We extend the motivating example with the RDFS triples listed in table 4a and consider two queries Q_1 and Q_2 transcribed in figures 4b and 4c:

- $\text{class_cardinality}_{\max}(\mathit{providesCare}, \mathit{SurgicalNurse}) = 2$,
- $\text{class_cardinality}_{\max}(\mathit{providesCare}, \mathit{Nurse}) = 3$,
- $\text{class_cardinality}_{\max}(t_2, Q_1) = 2$,
- $\text{class_cardinality}_{\max}(t_2, Q_2) = 3$.

All datasets do not use RDFS formalism to define classes. In that case, another way of grouping subjects has been proposed.

Characteristic Set Cardinalities A Characteristic Set (CS), as defined by Neumann and Moerkotte (Neumann and Moerkotte, 2011), is the set of predicates of a subject. For an entity s, occurring in a dataset D the Characteristic Set of s is $S_C(s) = \{p \mid \exists o, (s, p, o) \in D\}$. As each subject has exactly one CS, subjects can be grouped by CS. The set of all CS in a database D is given by $S_C(D) = \{S_C(s) \mid \exists p, o : (s, p, o) \in D\}$. For a CS P, and a predicate p, we define the CS cardinality of p in P as:

$$\text{CS_cardinality}_{\max}(p, P) = \max_{s, S_C(s)=P} \text{count}(s, p)$$

Like for class cardinalities, this definition needs adapting to the context of a query, by identifying the relevant CS. Within a query Q, the CS applicable to a subject $s(t)$ of a triple pattern t are all the CS which are supersets of the set of predicates associated with $s(t)$ in Q.

$$S_C(t, Q) = \bigcup_{P \in S_C(D), \forall t' \in Q (s(t')=s(t) \Rightarrow p(t') \in P)} P$$

The maximum cardinality of $p(t)$ in Q is retained as an upper bound of the cardinality of $p(t)$ in each of these CS.

$$\text{CS_cardinality}_{\max}(t, Q) = \max_{P \in S_C(t,Q)} \text{CS_cardinality}_{\max}(p(t), P)$$

Example 2 In our running example, there are seven subjects and four associated CS: $S_1 = \{experience, supervises, treats\}$ shared by d_1 and d_2, $S_2 = \{supervises\}$ for d_3, $S_3 = \{type, providesCare, operativeRole\}$ for n_1, and $S_4 = \{type, providesCare\}$ shared by n_2, n_3 and n_4. Using the motivating example, and the queries from figure 4, we can compute the following CS cardinalities:

- $\text{CS_cardinality}_{\max}(providesCare, S_3) = 1$,
- $\text{CS_cardinality}_{\max}(providesCare, S_4) = 3$,
- $\text{CS_cardinality}_{\max}(t_2, Q_1) = 1$,
- $\text{CS_cardinality}_{\max}(t_2, Q_2) = 3$.

Class and CS cardinalities depend on the query the predicate is contained in. So when using them on a lattice of queries they cannot be calculated once on the original query (as is the case for global cardinalities) but must be updated when each subquery is considered.

5.3.2 Properties

Cardinalities can be used to deduce the success or failure of a query from that of other queries. The following property and its corollary can be used with global, class, or CS cardinality. So we use $\text{cardinality}_{\max}(t, Q)$ to refer indifferently to $\text{global_cardinality}_{\max}(p(t))$, $\text{class_cardinality}_{\max}(t, Q)$, or $\text{CS_cardinality}_{\max}(t, Q)$. The proof of this property is provided in appendix A.

Property 5 (cardinality-based) Given a query Q, and a triple pattern t with a fixed predicate $p(t)$, if $s(t) \in \text{var}(Q)$ and $\frac{[[Q \wedge t]]_D}{\text{cardinality}_{\max}(t,Q)} > K$ then Q fails.

A corollary, for the special case where the maximum cardinality is exactly 1 is:

Property 6 (1-cardinality-based) Given a query Q, and a triple pattern t with a fixed predicate $p(t)$, if $s(t) \in \text{var}(Q)$ and $\text{cardinality}_{\max}(t, Q) = 1$ then $Q \wedge t$ fails $\Rightarrow Q$ fails.

This second property means that removing a triple pattern whose predicate has a cardinality of 1 cannot reduce the number of answers. So if the first query fails, its subquery will also fail. When using property 5, it is not sufficient to know whether $Q \wedge t$ fails, it is necessary to know its exact number of answers.

Properties 3 and 4 rely only on information contained within the original query, so are applicable to any query in any dataset. However, the cardinality based properties (5 and 6) require additional information: cardinalities of all the predicates must be calculated. For KBs likely to be modified often, it would be necessary to update cardinalities each time the data is changed. This requires either database administrators to provide cardinality values and keep them updated or users to regularly run a query on the KB to obtain cardinalities from the data, which is a costly operation. As such, we provide a variable-only version of our algorithm, Var, as well as the Full version that includes the *1-cardinality-based* property. In section 7, we will show that determining the exact number of answers of a query requires extra computation, which is why property 5 is not used in the Full version. The Full algorithm is detailed in the next section.

5.3.3 FULL *algorithm*

The complete algorithm, FULL, is also given in Algorithm 1. The only difference with VAR is the additional use of property 6 (line 20) to predict the status of a query without executing it. This cardinality-based property, can be based on global, class or CS cardinality.

Figure 3 also shows the subqueries executed by FULL with global cardinalities. The same queries as for VAR are avoided due to properties 3 and 4. Additionally, some queries are known to fail due to the cardinality property. For example, t_2 has maximum cardinality 1 (any subject has at most one *experience* value), 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 6 (line 21). Likewise, FULL avoids executing $t_3t_4t_5$ and $t_1t_3t_5$. Overall, for our running example, FULL executes only 6 queries.

5.4 Algorithm analysis

We complete this section with two algorithm properties. We start by showing that all the algorithms correctly return the set of MFIS and XSS of a query, then we analyse the complexity of the algorithms.

Correctness The correctness of our algorithms is proved by induction.

Property 7 Algorithms BASE, VAR and FULL are sound and complete, which means that they return exactly all MFIS and XSS of an RDF query Q, for a threshold K.

Proof For this proof, we consider an initial query Q with n triple patterns. Each algorithm builds the lattice of subqueries of Q, and studies the subqueries in a Breadth-First order. We will therefore break down our analysis into each layer of the lattice. The k-th layer of the lattice is composed of all the subqueries with k triple patterns. We will show that for any $k \leq n$, once the k-th level of the lattice has been completed, the three following invariants hold

- fis contains the FIS with k or more triple patterns.
- mfis contains the MFIS with $k+1$ or more triple patterns and the fis with exactly k triple patterns.
- xss contains the XSS with k or more triple patterns.

We wil show that this is true for k=n, and that if it is true for k, it remains true for $k-1$. The algorithm terminates once k=0, at which point xss contains the XSS with 0 or more triple pattern and mfis contains the MFIS with 1 or more triple pattern, as there are no FIS with 0 triple patterns since the empty query succeeds.

For k=n, there is only one query of size k, the initial query. It is executed by every algorithm. If it fails, it is an FIS, and is added to fis and mfis, and there are no XSS of size $\geq k$ so xss is empty. If it succeeds, it is an XSS, it is added to xss, and there are no FIS of size $\geq k$. The three invariants are true for $k = n$.

We suppose the invariants are true for a layer k, and prove they remain true on layer $k-1$. A query on layer $k-1$ is an FIS if it fails and all its direct superqueries fail. BASE checks failure first then superqueries, the other algorithms check superqueries first and only if they are all FIS, do they check query failure. If and only if both conditions are true, all the algorithms add the query to fis and

mfis. Likewise, a query on layer $k-1$ is an XSS if it succeeds and all its direct superqueries fail. Base checks success first then superqueries, the other algorithms check superqueries first and only if they are all FIS, do they check query success. If and only if both conditions are true, all the algorithms add the query to xss. The use of properties to determine query failure, rather than execution in the triplestore does not affect the validity of the method, as we have shown previously that they correctly provide the succes or failure status of the query. Since fis and xss contained respectively the FIS and XSS with k or more triple patterns at layer k, and we have added the FIS with $k-1$ triple patterns to fis and the XSS with $k-1$ triple patterns to xss, the two invariants are true for layer $k-1$. The direct superqueries of each FIS with $k-1$ triple patterns are removed from mfis. These are the FIS with k triple patterns which are not MFIS (because one of their subqueries is an FIS). Since at layer k, mfis contained the MFIS with $k+1$ or more triple patterns and the FIS with k triple patterns, and we have removed the FIS with k triple patterns that are not MFIS, and added the FIS with $k-1$ triple patterns, the invariant is true on layer $k-1$. □

Complexity We express the complexity of our algorithms in terms of the number of query executions that they require, using the size n of the original query. As Base executes every query of the lattice, it requires $2^n - 1$ executions. For all the other algorithms, the worst case is if the properties cannot be used (i.e. the whole lattice fails for property 3, all triple pattern removals remove variables for property 4, and no predicate cardinalities are 1 for property 6). Their worst case complexity is therefore the same as Base ($2^n - 1$). However, the improvement of our algorithms can be measured in specific cases when the properties can be used.

For property 3 the avoided query executions are the subqueries of XSS. Therefore the number of query executions avoided depends on the number of XSS of the query, and how many triple patterns they contain. The more triple patterns contained in the XSS, the more query executions will be avoided. As the XSS are not known before the algorithm is executed, we cannot determine in advance how many query executions will be avoided by property 3. Likewise, the cardinality property which involves the number of answers to a superquery does not allow us to determine in advance the number of queries executions avoided.

However, for properties 4 and 6 with global cardinality, we can use the structure of the query to determine before executing the algorithm how many query executions will be required. Star queries, where all triple patterns have the same subject variable, are the most widely used queries (Gallego et al., 2011). We give the formula for star queries because their simple structure means that triple patterns can be studied one at a time, without considering how they are linked to the rest of the query. It shows that each triple pattern which allows us to use a property halves the number of queries to be executed. The star shaped nature of the initial query is useful here because property 4 or 6 will either apply to a triple pattern t over the whole lattice, or not apply at all. For the same reason, we use global cardinalities here, because when using class or CS cardinalities, property 6 can apply to a triple pattern only for some queries in the lattice. For queries with a more complex structure, it is also possible to determine in advance the number of queries which will be avoided, but as this depends on the structure of the query, a formal description is not given.

Property 8 (star-query complexity) For a star query with n triple patterns, where k is the number of triple patterns where a variable or global cardinality property can be used (with $k \geq 1$). The number of query executions avoided by use of properties 4 and 6 is:

$$\sum_{i=1}^{k} 2^{n-i} - 1$$

Proof We prove the formula recursively.

For k=1, we consider one triple pattern, t, to which one of properties 4 or 6 can be applied. Both properties imply that for any query Q, if $Q \wedge t$ fails then Q fails. If $Q \wedge t$ succeeds, then Q is not executed according to property 3. Therefore, whether $Q \wedge t$ succeeds or not, the execution of any query Q not containing t can be avoided. The number of such queries is the number of queries in a lattice of $n-1$ elements minus the empty query, i.e. $2^{n-1} - 1$.

Suppose the property is true for a value k, we will prove it remains true for $k+1$. We call $t_1, ..., t_k$ the first k triple patterns, and t_{k+1} the new triple pattern considered. According to our hypothesis, the number of query executions avoided by $t_1, ..., t_k$ is $\sum_{i=1}^{k} 2^i - 1$. The queries avoided by t_{k+1}, not avoided by $t_1, ..., t_k$ are queries containing all the $t_1, ..., t_k$, but not t_{k+1}. This amounts to choosing between 0 and $n-k-1$ triple patterns to add to $t_1, ..., t_k$ to create a query. The number of such queries is therefore : $\sum_{i=0}^{n-k-1} \binom{n-k-1}{i} = 2^{n-k-1} = 2^{n-(k+1)}$. In total, the number of query executions avoided is : $\sum_{i=1}^{k+1} 2^i - 1$. □

We use Q from the motivating example to illustrate why there is no simple formula for non star queries. Q contains three triple patterns that a property can be applied to: t_2 for the cardinality property, t_3 and t_5 for the variable property. However, due to the shape of the query, we cannot apply property 6 to a query $Q_a \wedge t_2$ unless $s(t_2)$, i.e., d is part of query Q_a. So we cannot use this property on query $t_2t_4t_5$ to avoid executing query t_4t_5. Likewise to use property 4 on a query $Q_b \wedge t_5$, Q_b must contain the variable n. We cannot use this property on query $t_1t_2t_5$ to avoid executing query t_1t_2. In total, 23 query executions can be avoided using properties 4 and 6, rather than the 27 expected, if our formula held for non star queries.

6 Operator Integration

Up to now, our algorithms have been designed with conjunctive queries in mind. This excludes a large part of SPRAQL queries that contain other operators (Gallego et al., 2011). We detail in this section how our methods can be applied to queries containing operators acting on a graph pattern level: FILTER, UNION and OPTIONAL. Operators like DISTINCT or GROUP BY which appear outside graph patterns, on a SPARQL query level are perspectives for future work. The previous definition of subqueries based on triple pattern inclusion cannot be directly applied when operators are present, so we start by providing a more general definition for the subqueries of a SPARQL query. We then detail the particularities of each operator and how they affect the calculation of MFIS and XSS.

Definition 4 Subqueries of a SPARQL query are defined recursively as follows.

1. The empty query $\emptyset$ is a subquery of every query.
2. The only non empty subquery of a triple pattern t is t.
3. If $Q = Q_1$ *AND* Q_2, the subqueries of Q are Q'_1 *AND* Q'_2 where $Q'_1 \subseteq Q_1$ and $Q'_2 \subseteq Q_2$.
4. If $Q = Q_1$ *FILTER*(R), the subqueries of Q are queries Q'_1 and Q'_1 *FILTER*(R) where $Q'_1 \subseteq Q_1$ providing the resulting query is valid SPARQL (i.e. that the variables in the filters are included in the triple patterns).
5. If $Q = Q_1$ *UNION* Q_2, the non empty subqueries of Q are queries Q'_1, Q'_2 and Q'_1 *UNION* Q'_2 where $Q'_1 \subseteq Q_1$, $Q'_2 \subseteq Q_2$, $Q'_1 \neq \emptyset$ and $Q'_2 \neq \emptyset$.
6. If $Q = Q_1$ *OPT* Q_2, the non empty subqueries of Q are queries Q'_1 and Q'_1 *OPT* Q'_2 where $Q'_1 \subseteq Q_1$, $Q'_2 \subseteq Q_2$, $Q'_1 \neq \emptyset$ and $Q'_2 \neq \emptyset$.

This definition encompasses the previous definition of subqueries for conjunctive SPARQL queries so the previous algorithms can be used with this new definition.

6.1 FILTER

The FILTER operator is defined by a built in condition, which is a logical statement linking variables, constants, URIs with equality or inequality symbols. The corresponding semantics are, for a graph pattern P, a built-in condition R, and a dataset D: $[[P\ FILTER\ R]]_D = \{\mu \in [[P]]_D, \mu\ \textit{satisfies}\ R\}$.

To apply our algorithms to queries containing filters, we can consider the filter condition in a similar manner as triple patterns. Subqueries are created by removing either a triple pattern or the filter condition. In order to insure predictable behaviour when executing a query, the variables included in a built-in condition must appear in the graph patterns of the query. Therefore, when removing triple patterns, we must verify that this condition is still met. If not, the query in question is considered invalid, and is not included in the lattice.

Example 3 Using query Q from the motivating example, we add *FILTER* $?e \geq 20$, which we call f_1, creating query $Q' = t_1t_2t_3t_4t_5f_1$. When creating the lattice, we consider for instance $Q'' = t_1t_3t_4t_5f_1$. Since the variable e appears in f_1, but not in $t_1t_3t_4t_5$, Q'' is not a valid SPARQL query, and is not included in the lattice.

The effect of a FILTER condition is to select only answers respecting certain conditions. As removing a FILTER condition from a query cannot remove any variables (or the condition that all variables appearing in a built-in condition must appear in a triple pattern would not be respected), property 4 tells us that removing a filtering condition from a query cannot reduce its number of answers. Therefore, finding the MFIS of a query containing a FILTER condition requires at most the same number of query executions as the query with no FILTER condition.

6.2 OPTIONAL

The formal definition of the OPTIONAL operator for two graph patterns P_1 and P_2, and a dataset D is: $[[P_1\ OPT\ P_2]]_D = [[P_1]]_D \bowtie [[P_2]]_D \cup [[P_1]]_D \setminus [[P_2]]_D$ where $\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 | \forall \mu' \in \Omega_2,\ \mu$ and μ' are not compatible$\}$, and $\cup$ is the multiset sum operator. While there is no direct relation between the number of answers of A, B and A *OPT* B, we can deduce the failure of A *OPT* B from the failure of A.

```
SELECT * WHERE {
  { { ?d treats ?p .                # t1
      ?d type Surgeon }             # t2
    UNION
    { ?d type Cardiologist } } .    # t3
      ?d supervises ?n }            # t4
```

(a) Q_3 in factored form

```
SELECT * WHERE {
  { ?d treats ?p .           # t1
    ?d type Surgeon .        # t2
    ?d supervises ?n }       # t4
  UNION
  { ?d type Cardiologist .   # t3
    ?d supervises ?n } }     # t4'
```

(b) Q_3 in developed form

Fig. 5 $Q_3 = (t_1 t_2 \cup t_3) t_4 = t_1 t_2 t_4 \cup t_3 t'_4$

Property 9 $|[[A\ OPT\ B]]_D| \geq |[[A]]_D|$

Proof We will show that two distinct answers of query A can be linked to two distinct answers of $A\ OPT\ B$.

Consider $\mu_1 \neq \mu_2 \in [[A]]_D$. If $\exists \mu_3 \in [[B]]_D$ such that μ_3 and μ_1 are compatible, then $\mu_1 \cup \mu_3 \in [[A]]_D \bowtie [[B]]_D$ so $\mu_1 \cup \mu_3 \in [[A\ OPT\ B]]_D$. Otherwise, $\mu_1 \in [[A]]_D \backslash [[B]]_D$ so $\mu_1 \in [[A\ OPT\ B]]_D$. We will call m_1 the one of these mappings included in $[[A\ OPT\ B]]_D$. Likewise, either $\exists \mu_4 \in [[B]]_D$ such that $\mu_2 \cup \mu_4 \in [[A\ OPT\ B]]_D$ or $\mu_2 \in [[A\ OPT\ B]]_D$. We will call m_2 this mapping. Since $\mu_1 \neq \mu_2$, $\exists v \in \text{var}(A), \mu_1(v) \neq \mu_2(v)$. From the definition of m_1 and m_2, $m_1(v) = \mu_1(v)$ and $m_2(v) = \mu_2(v)$. We then have $v \in \text{var}(A\ OPT\ B)$ and $m_1(v) \neq m_2(v)$. So we have two distinct answers $m_1 \neq m_2 \in [[A\ OPT\ B]]_D$. □

From this we deduce two properties that provide information on the subqueries of $A\ OPT\ B$ using the MFIS and XSS of A. Their proofs are given in appendix A.

Property 10 (optional MFIS) The MFIS of a query A are MFIS of query $A\ OPT\ B$.

Property 11 (optional XSS) All subqueries of $A\ OPT\ B$ which are not subqueries of a query $A^*\ OPT\ B'$ where A^* is an XSS of A fail.

Using these properties, if the MFIS and XSS of A are known, several queries of the lattice of subqueries of $A\ OPT\ B$ do not need to be executed.

Example 4 We transform query Q from the motivating example, adding an optional part $t_6 t_7$: *?d degreeFrom ?u . ?d specialty ?s* creating query $Q' = t_1 t_2 t_3 t_4 t_5\ OPT\ t_6 t_7$. The MFIS and XSS of $Q = t_1 t_2 t_3 t_4 t_5$ are determined with the previous algorithm. The MFIS are $t_1 t_3$, $t_1 t_5$ and t_4. These are also MFIS of Q'. The XSS of Q are $t_1 t_2$ and $t_2 t_3 t_5$. Therefore the queries that are left to execute to find all MFIS and XSS of Q' are restricted to subqueries of $t_1 t_2\ OPT\ t_6 t_7$, and $t_2 t_3 t_5\ OPT\ t_6 t_7$.

6.3 UNION

The definition of the UNION operator for graph patterns P_1 and P_2, and dataset D is: $[[P_1\ UNION\ P_2]]_D = [[P_1]]_D \cup [[P_2]]_D$. The UNION operator in SPARQL is distributive over the AND, FILTER and OPTIONAL operators. This means that queries containing UNION operators can be written in different ways. In figure 5

we present a query in *factored form* (5a), and in *developed form* (5b). To avoid the inconsistency of two forms of the same query leading to different subqueries, we require that queries always be presented in the same form. As queries in developed form can also create problems for MFIS interpretation, we will deal with queries written in factored form. We illustrate this with an example.

Example 5 Consider $Q_3 = (t_1t_2 \cup t_3)t_4$, depicted in figure 5. When writing the query in developed form, we have created triple patterns $t'_4 = t_4$ to distinguish between the two occurrences of the triple pattern. When creating the subqueries, t_4 and t'_4 will be treated independently. As such, it is possible for t_4 to be an MFIS, and at the same time, for $t_3t'_4$ to succeed, because $t_3t'_4$ is not considered a superquery of t_4. When we return the MFIS and XSS, this can be confusing, as t'_4 and t_4 represent the same triple pattern: *?d supervises ?n* which should not be both a failure cause, and part of a succeeding query.

When executing subqueries in order to find the MFIS and XSS, we can keep using the properties given in section 5 for subqueries with only AND operators. Additionnaly, the definition of the UNION operator gives us: $|[[P_1 \cup P_2]]_D| = |[[P_1]]_D| + |[[P_2]]_D|$. Like for the OPTIONAL operator, this gives us two more properties that link the MFIS and XSS of a query written in factored form $A \cup B$ to the MFIS and XSS of queries A and B. Their proofs are given in appendix A.

Property 12 (union MFIS) The MFIS of a query A are MFIS of query $A \cup B$.

Property 13 (union XSS) All subqueries of $A \cup B$ which are not subqueries of a query $A^* \cup B^*$ where A^* is an XSS of A and B^* is an XSS of B fail.

If the MFIS and XSS of A and B are known, the additional queries we need to study are restricted to the subqueries of a union of an XSS of A and an XSS of B.

The complete formalization of subqueries allows us to apply the concepts of MFIS and XSS to any SPARQL query containing operators AND, FILTER, UNION and OPTIONAL. To avoid the potential ambiguity caused by SPARQL distributivity properties, we require queries to be presented in a factored form. This does not restrict the applicability of the approach, as a simple algorithm can be used to transform the form of a query. The algorithms previously presented are still applicable to the conjunctive parts of a query containing other operators, and we have detailed here how additional specific properties can be used to improve the computation of MFIS and XSS when SPARQL operators are included.

7 Experimental Evaluation

In this section we experimentally evaluate the Var and Full algorithms, and discuss some implementation considerations. Both algorithms are compared with the Base algorithm to show the improvements in execution time brought on by leveraging the properties presented in section 5. We also verify that the execution times are acceptable for queries of a reasonable size. Our implementation is available at `https://forge.lias-lab.fr/projects/tma4kb` with a tutorial to reproduce our experiments.

7.1 Experimental Setup

Hardware Our experiments were run on a Ubuntu Server LTS system with 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 algorithms are implemented in Oracle Java 1.8 64bits and run using the Jena TDB triplestore. When cardinalities are required, they are precomputed before running the algorithms. We set the threshold for plethoric answers K=100, as it is the default limit used by the DBpedia SPARQL endpoint.

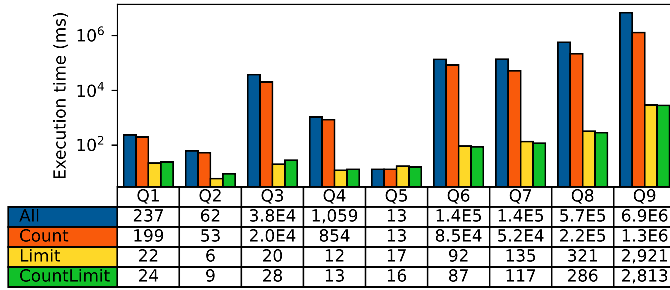
As Cartesian products are costly to execute, to determine the number of answers of a query containing a Cartesian product we split it into connected parts and execute each part separately. The number of results is the product of the number of results of each part. As each part of the Cartesian product is itself a subquery which may have been executed later, we store individual results for future use, which potentially reduces the total number of executed queries. This is especially noticeable with Base, where fewer than $2^n - 1$ queries are executed for queries containing Cartesian products in their lattice of subqueries.

Synthetic Dataset and Queries We have used a dataset of 11M triples, generated with the WatDiv benchmark. We have considered a total of 21 queries with 7 star-shaped queries, 7 chain queries (subject of triple pattern j+1 is the object of triple pattern j) and 7 composite queries (any other configuration). These queries contain 4 to 12 triple patterns. We have based our queries on the queries given in the WatDiv test cases. In particular, all our chain queries are derived from chain query IL-1-10, by removing triple patterns to reach from 4 to 10 triple patterns. Composite queries F1, F2, F4 and C2 and star query C3 from the test case were also used (queries Q15, Q16, Q18, Q19 and Q3). We created additional composite and star queries in order to have varied characteristics (number of triple patterns in the original query and in the MFIS and XSS, and number of MFIS and XSS).

Real Dataset and Queries We have downloaded the DBpedia dataset (the English 3.9 version) which contains 812M triple patterns and used queries from the LSQ project (Saleem et al., 2015). The queries we used come from a log of queries submitted by users to DBpedia (version 3.5.1) between April 30, and June 20, 2010. This log records over 500,000 queries of which we have selected 9 queries that produce large result sets. We selected conjunctive queries with the largest result sets of each size (4 to 10 triple patterns). They are star-shaped or composite. Some minor adaptations were made as some queries were not compatible with version 3.9 of DBpedia.

7.2 Comparison of execution methods

In section 5, we have used query properties to reduce the number of executed queries. Another way to reduce the time to find MFIS and XSS is to reduce the execution time of each query. As such, we have studied several methods to submit queries to a triplestore. We begin our experiments by comparing execution



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
All	237	62	3.8E4	1,059	13	1.4E5	1.4E5	5.7E5	6.9E6
Count	199	53	2.0E4	854	13	8.5E4	5.2E4	2.2E5	1.3E6
Limit	22	6	20	12	17	92	135	321	2,921
CountLimit	24	9	28	13	16	87	117	286	2,813

Fig. 6 Execution time Full algorithm DBpedia

methods in order to subsequently use the fastest execution method to compare the algorithms. Four execution methods are compared.

- ALL: submit a query as is, receive all the answers from the triplestore, count them, and determine if there are more than the threshold K.
- COUNT: add the SPARQL operator COUNT to count answers within the triplestore rather than in the algorithm.
- LIMIT: add the SPARQL operator LIMIT so that the triplestore returns a maximum of K+1 answers.
- COUNT-LIMIT: add both SPARQL operators COUNT and LIMIT.

In the last two cases, we do not know the exact number of answers returned by the query, only whether or not there are more than K.

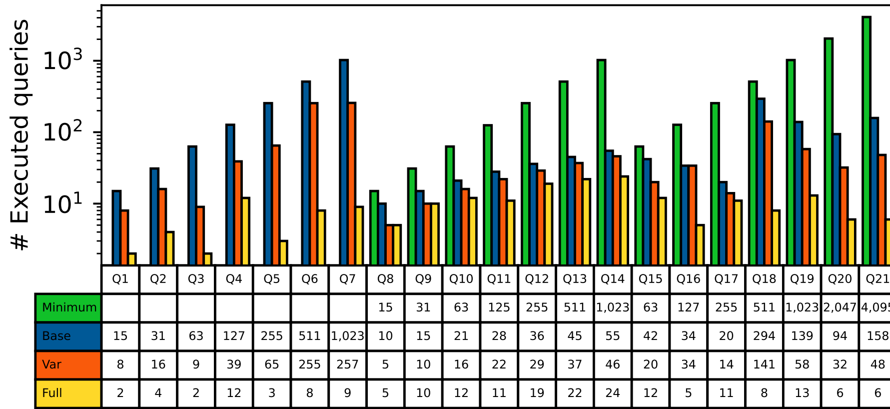
We have used the Full algorithm (global cardinalities) and the DBpedia dataset and queries for this experiment. Figure 6 presents the execution times of each method. Unsurprisingly, the methods that return the exact number of answers (ALL and COUNT) are significantly slower, especially as the query size increases: for queries Q6 to Q9, there are three orders of magnitude between methods ALL and LIMIT. The fastest method is the COUNT-LIMIT method. This has also been verified with the WatDiv dataset. Therefore, the COUNT-LIMIT execution method is used in the following experiments, unless otherwise specified.

7.3 Algorithm comparison

We compare the performance of three algorithms: Base, Var and Full (using the *1-cardinality-based* property with global cardinalities), first on the generated dataset, then the real dataset.

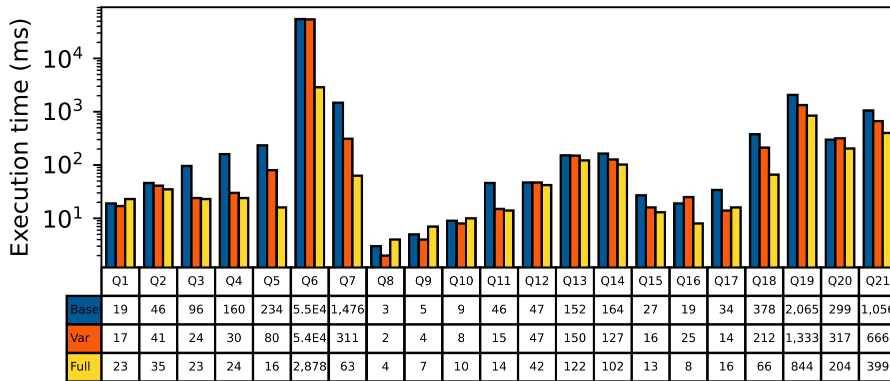
Synthetic Dataset and Queries For the WatDiv dataset, figures 7 and 8 give the number of executed queries and the execution time, for each algorithm and query.

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, as we execute Cartesian products by separating them into their connected parts, queries that have a succeeding superquery can be executed as part of a Cartesian product. The number of queries executed if Cartesian products were



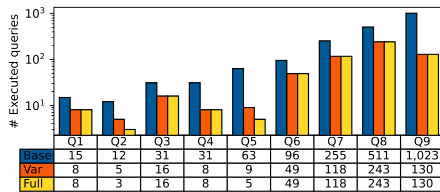
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21
Minimum								15	31	63	125	255	511	1,023	63	127	255	511	1,023	2,047	4,095
Base	15	31	63	127	255	511	1,023	10	15	21	28	36	45	55	42	34	20	294	139	94	158
Var	8	16	9	39	65	255	257	5	10	16	22	29	37	46	20	34	14	141	58	32	48
Full	2	4	2	12	3	8	9	5	10	12	11	19	22	24	12	5	11	8	13	6	6

Fig. 7 # Executed queries Watdiv 11M triples



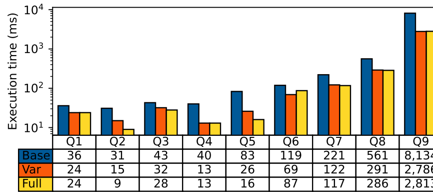
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21
Base	19	46	96	160	234	5.5E4	1,476	3	5	9	46	47	152	164	27	19	34	378	2,065	299	1,056
Var	17	41	24	30	80	5.4E4	311	2	4	8	15	47	150	127	16	25	14	212	1,333	317	666
Full	23	35	23	24	16	2,878	63	4	7	10	14	42	122	102	13	8	16	66	844	204	399

Fig. 8 Execution time Watdiv 11M triples



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Base	15	12	31	31	63	96	255	511	1,023
Var	8	5	16	8	9	49	118	243	130
Full	8	3	16	8	5	49	118	243	130

Fig. 9 # Executed queries DBpedia



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Base	36	31	43	40	83	119	221	561	8,134
Var	24	15	32	13	26	69	122	291	2,786
Full	24	9	28	13	16	87	117	286	2,813

Fig. 10 Execution time Jena DBpedia

executed directly, $2^n - 1$, is given on Figure 7 in the *Maximum* bars and shows that the baseline time would be higher if we executed Cartesian products. 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 31% of the BASE algorithm execution time, whereas FULL saves 44%.

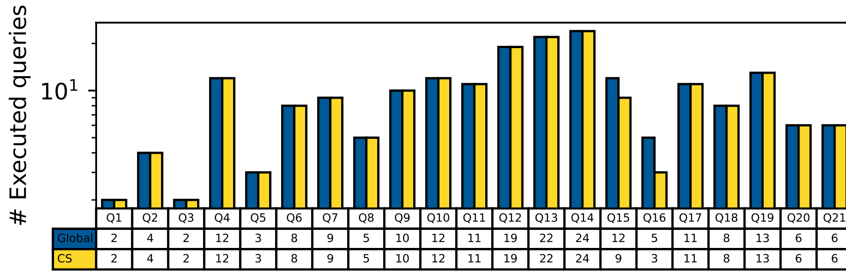
Real Dataset and Queries We reproduce the experiment on a real dataset – DBpedia – with user-submitted queries extracted from a query log. We show the number of executed queries and the run-time of each algorithm in figures 9 and 10.

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 predicate cardinalities in DBpedia, as few of the predicates used have maximum cardinality 1. The cardinality property cannot be applied to these predicates. We have noted that 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 (Giacometti et al., 2019; Muñoz and Nickles, 2017). Using a curated dataset would likely improve the benefit of the cardinality-based pruning. Consequently, VAR and FULL have very similar execution times. VAR saves around 49% of the baseline time, and FULL saves around 52%.

7.4 Cardinality variations

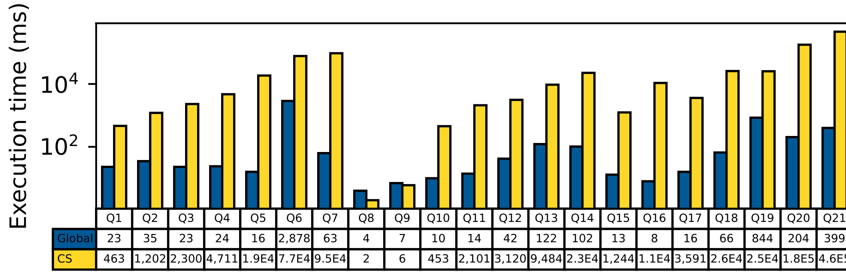
Next, we compare the use of global cardinalities with class or CS cardinalities for the FULL algorithm. As WatDiv does not provide a schema with classes, class cardinalities are only tested with the DBpedia dataset. The number of executed queries and execution times are given in figures 11 and 12 for WatDiv and figures 13 and 14 for DBpedia. Class and CS cardinalities rarely reduce the number of executed queries. For class cardinalities that is only the case for Q5, whose execution time is nearly halved. For the other queries, the execution times using class or global cardinalities are very similar. However class cardinalities can only be used if the dataset uses an RDFS schema, as is the case with DBpedia, but not for WatDiv. CS cardinalities cause a much longer execution time. This is because cardinality values are checked over all CS when examining each query. Due to this additional cost, and despite the theoretical benefits of CS cardinality, global and class cardinalities are the most appropriate for our algorithm.

We also compare using the *1-cardinality-based* property 6 (that only uses maximum cardinalities of 1), and the *cardinality-based* property 5 (that can be used for any cardinality value). This experiment uses the DBpedia dataset and queries and global cardinalities. We show the number of executed queries in figure 13 (*Global* for property 6, and *AnyCard* for property 5), and the algorithm execution times in figure 15. Since property 5 requires knowing the exact number of answers of a query, rather than if it succeeds or fails, we cannot use the most efficient query execution method (COUNT-LIMIT). We provide the execution times with the COUNT method for properties 5 (*any-cardinality_count*) and 6 (*1-cardinality_count*), and using the COUNT-LIMIT method for property 6 (*1-cardinality_countlimit*). Leveraging property 5 presents a significant improvement in the number of queries executed, and in execution time with a reduction of up to an order of magnitude. However, this improvement is smaller than the one achieved by using the COUNT-LIMIT method. So the algorithm using property 5 does not offer sufficient improvement to compensate it requiring a less effective query execution method.



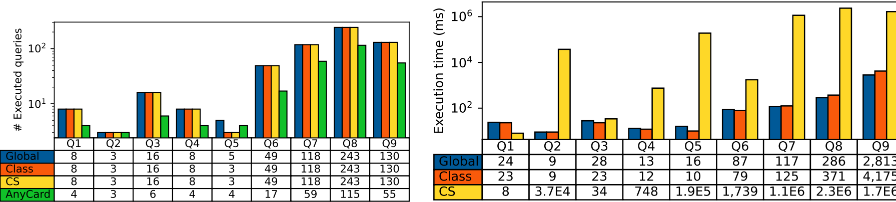
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21
Global	2	4	2	12	3	8	9	5	10	12	11	19	22	24	12	5	11	8	13	6	6
CS	2	4	2	12	3	8	9	5	10	12	11	19	22	24	9	3	11	8	13	6	6

Fig. 11 # Executed queries Watdiv 11M triples Full



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21
Global	23	35	23	24	16	2,878	63	4	7	10	14	42	122	102	13	8	16	66	844	204	399
CS	463	1,202	2,300	4,711	1.9E4	7.7E4	9.5E4	2	6	453	2,101	3,120	9,484	2.3E4	1,244	1.1E4	3,591	2.6E4	2.5E4	1.8E5	4.6E5

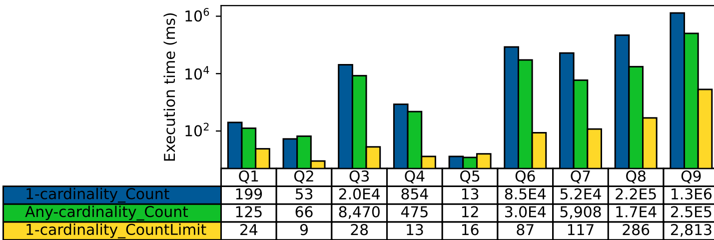
Fig. 12 Execution time Watdiv 11M triples Full



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Global	8	3	16	8	5	49	118	243	130
Class	8	3	16	8	3	49	118	243	130
CS	8	3	16	8	3	49	118	243	130
AnyCard	4	3	6	4	4	17	59	115	55

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Global	24	9	28	13	16	87	117	286	2,813
Class	23	9	23	12	10	79	125	371	4,175
CS	8	3.7E4	34	748	1.9E5	1,739	1.1E6	2.3E6	1.7E6

Fig. 13 # Executed queries DBpedia Full **Fig. 14** Execution time DBpedia Full

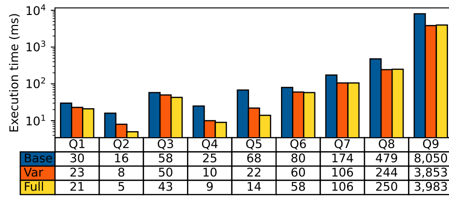


	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
1-cardinality_Count	199	53	2.0E4	854	13	8.5E4	5.2E4	2.2E5	1.3E6
Any-cardinality_Count	125	66	8,470	475	12	3.0E4	5,908	1.7E4	2.5E5
1-cardinality_CountLimit	24	9	28	13	16	87	117	286	2,813

Fig. 15 Execution time DBpedia Full

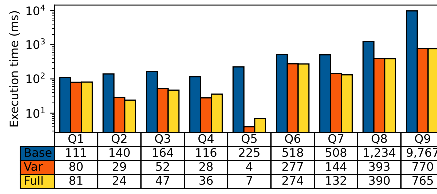
7.5 Impact of the threshold

We have run experiments changing the threshold where answers are considered plethoric. We compare the three algorithms with new thresholds K=10 and K=1000 using the DBpedia queries. The execution times are given in figures 16 and 17. As the threshold increases, the cost of executing each query grows, so Var and Full



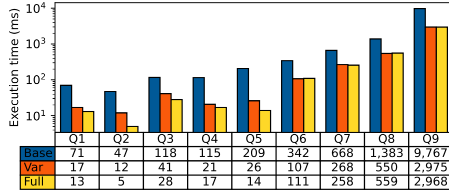
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Base	30	16	58	25	68	80	174	479	8,050
Var	23	8	50	10	22	60	106	244	3,853
Full	21	5	43	9	14	58	106	250	3,983

Fig. 16 Execution times for K=10



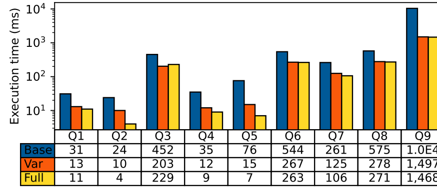
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Base	111	140	164	116	225	518	508	1,234	9,767
Var	80	29	52	28	4	277	144	393	770
Full	81	24	47	36	7	274	132	390	765

Fig. 17 Execution times for K=1000



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Base	71	47	118	115	209	342	668	1,383	9,767
Var	17	12	41	21	26	107	268	550	2,975
Full	13	5	28	17	14	111	258	559	2,968

Fig. 18 Execution time Fuseki



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Base	31	24	452	35	76	544	261	575	1.0E4
Var	13	10	203	12	15	267	125	278	1,497
Full	11	4	229	9	7	263	106	271	1,468

Fig. 19 Execution time Virtuoso

perform better compared with Base. Queries Q2, Q5, and Q9 have a low execution time for K=1000, because the initial query succeeds (it has fewer than 1000 answers) so only one query is executed. Discounting these queries for K=1000, Var saves on average 42% of the Base time for K=10, and 59% for K=1000 (against 49% for K=100). Likewise, Full saves on average 47% of the Base time for K=10, and 59% for K=1000 (against 52% for K=100).

7.6 Impact of triplestores

In our last experiment, we study the influence that the triplestore has on the execution times of our algorithms. In addition to Jena TDB, which is a native triplestore, we have implemented two SPARQL endpoint triplestores : Fuseki and Virtuoso. We have reproduced the algorithm comparison experiment using DBpedia. The execution times with Fuseki and Virtuoso are given in figures 18 and 19. They are to be compared with figure 10 which shows the results for Jena TDB.

Jena TDB is generally faster than Fuseki, so our method is better suited to a native triplestore, rather than its access through a SPARQL endpoint. There is no clear better solution between Fuseki and Virtuoso, but Virtuoso performs poorly for queries with large result sets (Q3 and Q6). Overall, even with large queries, the execution times of our algorithms are reasonable in all three triplestores, so our methods are usable in practice across various triplestores and dataset sizes.

7.7 Details of execution time

For all of our experiments, we have compared the query execution time to the total algorithm execution time in order to check our hypothesis that the query execution time dominates the run time. Figure 20 shows the ratio of query execution time to total execution time, against the total execution time for the previous experiments.

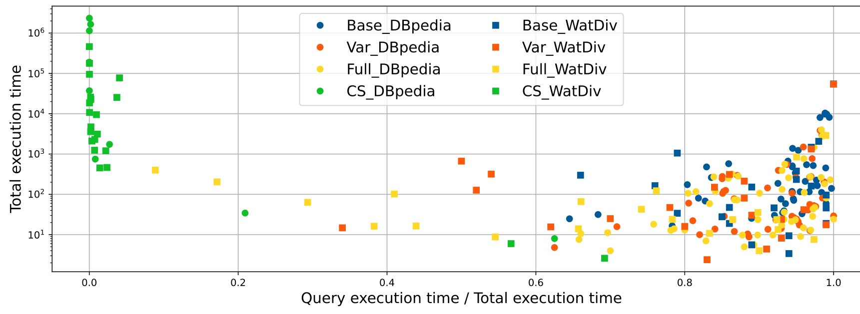


Fig. 20 Total execution time against the ratio between query and total execution times

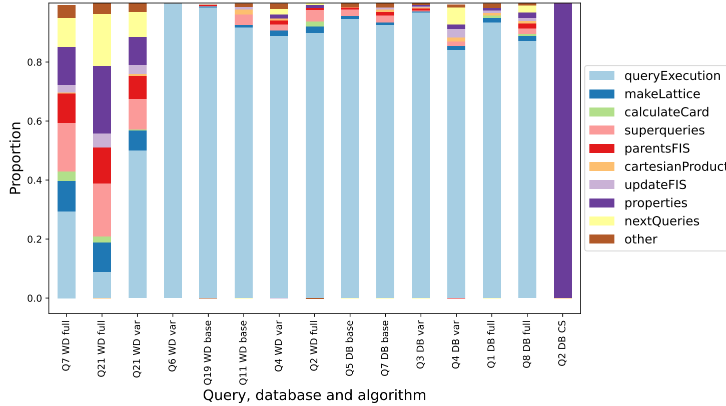


Fig. 21 Breakdown of execution times

We have only presented the experiments using the CountLimit method. The Limit method has similar results, and the All and Count methods have long execution times and a ratio of query execution time to total execution time close to 1.

For the three algorithms, the query execution time is usually between 80% and 100%. When the total execution time is larger, the query execution time represents a larger portion of the total execution time. For the DBpedia executions, the average ratio of query execution time to total execution time is 91% (92% for Base, 91% for Var and 90% for Full). For the WatDiv executions, it is 79% (90% for Base, 79% for Var and 68% for Full). This validates the hypothesis made in section 5 that the query execution time dominates the total run time.

As we claimed in the cardinality variation experiments, the CS based experiments have a very low ratio of query execution time to total execution time, and the increased run time compared with global cardinalities is due to longer computation of the cardinalities, rather than an increase in query execution time.

Finally, in figure 21 we show the detailed breakdown of the algorithm execution time for some typical and extreme queries from the previous chart. The first three bars are queries with particularly low query execution time to total execution time ratios, the next two have a high ratio and high execution time, the last bar is from a CS algorithm and the others are typical queries. The main parts of the algorithm that make up the execution time are the use of properties (for the Var and Full

algorithms lines 18 and 20), the calculation of direct superqueries (line 7) and direct subqueries (line 17), and the initial calculation of the lattice (line 2). When the initial query has a large number of triple patterns (such as Q7 with 9, or Q21 with 12) the number of subqueries and superqueries calculated in the algorithms rises exponentially. Since these queries also have short query execution times, the algorithm computation time dominates the total run time.

8 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. As such, we have defined a more general notion, named MFIS. If queries are relaxed by removing triple patterns, as long as an MFIS is included in the user query, it will fail. So an MFIS is a reason for the query's failure.

Having defined the notion of MFIS, our next objective was to develop efficient approaches to compute them. We have first shown that enumerating the MFIS and their dual notion, the XSS, is an NP-hard problem. As the complexity is related to the number of triple patterns in the query, our aim was to develop algorithms that provide acceptable performance for queries of a reasonable size. We presented a baseline method to calculate all MFIS and XSS of a failing query, then proposed improvements based on query and data properties to reduce the number of queries executions, and therefore reduce the run-time of our algorithms. A complexity analysis shows that the number of query executions is potentially halved for each triple pattern of the original query verifying one of the properties.

Experiments on a synthetic dataset as well as a real dataset with user submitted queries show that our optimized algorithms offer a significant improvement. Our Var algorithm saves around 40% of the baseline time and can be used for any query. Our Full algorithm saves around 48% of the baseline time but requires additional information – predicate cardinalities. We have studied various types of cardinalities and shown that while more precise cardinalities are theoretically useful to reduce the number of executed queries, in practice, the time saved by these methods is less than the extra computation cost they induce.

The next step is using the MFIS and XSS to aid in rewriting queries with plethoric answers. Query modification can be performed entirely by the user (i.e. when we provide the user with MFIS and XSS and they interpret them to adapt their query), entirely automatically, or by means of an interactive approach, where the user is guided through various changes applied to their query. The challenge is identifying how the MFIS and XSS can accelerate this process. Our long term goal is to build a framework to deal with any unexpected answer problem. This means identifying the similarities between the problems, and the specificities of each. The definition of MFIS introduced in this paper is a first step toward this goal as it generalises the notion of failure causes to non monotonic problems.

References

Aluç G, Hartig O, Özsu MT, Daudjee K (2014) Diversified stress testing of rdf data management systems. In: ISWC'14, Springer, pp 197–212

Auer S, Lehmann J, Hellmann S (2009) LinkedGeoData: Adding a Spatial Dimension to the Web of Data. In: ISWC'09, Springer

Bechhofer S, van Harmelen F, Hendler J, Horrocks I, McGuinness DL, Patel-Schneider PF, Stein LA (2004) Owl web ontology language reference. W3C Recommendation URL `https://www.w3.org/TR/owl-ref/.`

Bosc P, HadjAli A, Pivert O (2006) About overabundant answers to flexible queries. In: IPMU'06, vol 6, pp 2221–2228

Bosc P, Hadjali A, Pivert O (2009) Incremental controlled relaxation of failing flexible queries. Journal of Intelligent Information Systems 33(3):261

Bosc P, Hadjali A, Pivert O, Smits G (2010) Une approche fondée sur la corrélation entre prédicats pour le traitement des réponses pléthoriques. In: EGC'10, pp 273–284

Brickley D, Guha R (2014) Rdf schema 1.1. W3C Recommendation URL `https://www.w3.org/TR/rdf-schema/.`

Chakrabarti K, Chaudhuri S, Hwang Sw (2004) Automatic categorization of query results. In: SIGMOD'04, pp 755–766

Dellal I (2019) Management and Exploitation of Large and Uncertain Knowledge Bases. PhD thesis, ISAE-ENSMA - Poitiers

Fokou G, Jean S, Hadjali A, Baron M (2015) Cooperative techniques for SPARQL query relaxation in RDF databases. In: ESWC'15, Springer, pp 237–252

Fokou G, Jean S, Hadjali A, Baron M (2016) Rdf query relaxation strategies based on failure causes. In: European semantic web conference, Springer, pp 439–454

Gallego MA, Fernández JD, Martínez-Prieto MA, de la Fuente P (2011) An Empirical Study of Real-World SPARQL Queries. In: Proceedings of the USEWOD workshop co-located with WWW'11

Giacometti A, Markhoff B, Soulet A (2019) Mining Significant Maximum Cardinalities in Knowledge Bases. In: ISWC'19, vol 11778

Godfrey P (1997) Minimization in Cooperative Response to Failing Database Queries. International Journal of Cooperative Information Systems 6(2):95–149

Harris S, Seaborne A (2013) SPARQL 1.1 query language. W3C Recommendation URL `https://www.w3.org/TR/sparql11-query/.`

Ilyas IF, Beskales G, Soliman MA (2008) A survey of top-k query processing techniques in relational database systems. ACM Comput Surv 40(4)

Jagadish HV, Chapman A, Elkiss A, Jayapandian M, Li Y, Nandi A, Yu C (2007) Making database systems usable. In: SIGMOD'07, p 13–24

Jannach D (2006) Techniques for fast query relaxation in content-based recommender systems. In: Annual Conference on Artificial Intelligence, Springer, pp 49–63

Lehmann J, Isele R, Jakob M, Jentzsch A, Kontokostas D, Mendes PN, Hellmann S, Morsey M, van Kleef P, Auer S, Bizer C (2015) DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. Semantic Web 6(2):167–195

McSherry D (2004) Incremental relaxation of unsuccessful queries. In: European Conference on Case-Based Reasoning, Springer, pp 331–345

Moises SA, Pereira SdL (2014) Dealing with empty and overabundant answers to flexible queries. Journal of Data Analysis and Information Processing pp 12–18

Muñoz E, Nickles M (2017) Mining cardinalities from knowledge bases. In: DEXA'17, vol 10438, pp 447–462

Neumann T, Moerkotte G (2011) Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In: IEEE'11, pp 984–994

Ozawa J, Yamada K (1995) Discovery of global knowledge in a database for cooperative answering. In: IEEE'95, vol 2, pp 849–854

Parkin L, Chardin B, Jean S, Hadjali A, Baron M (2021) Dealing with plethoric answers of sparql queries. In: DEXA'21, Springer, pp 292–304

Pérez J, Arenas M, Gutierrez C (2009) Semantics and complexity of SPARQL. ACM Trans Database Syst 34(3)

Saleem M, Ali MI, Hogan A, Mehmood Q, Ngomo AN (2015) LSQ: The Linked SPARQL Queries Dataset. In: ISWC'15, pp 261–269

Song Q, Namaki MH, Wu Y (2019) Answering why-questions for subgraph queries in multi-attributed graphs. In: ICDE'19, pp 40–51

Sperner E (1928) Ein satz über untermengen einer endlichen menge. Mathematische Zeitschrift 27(1):544–548

Vasilyeva E, Thiele M, Bornhövd C, Lehner W (2016) Answering "why empty?" and "why so many?" queries in graph databases. Journal of Computer and System Sciences 82(1):3–22

Wang M, Liu J, Wei B, Yao S, Zeng H, Shi L (2019) Answering why-not questions on SPARQL queries. Knowledge and Information Systems 58:169–208

Webber BL, Mays E (1983) Varieties of user misconceptions: Detection and correction. In: IJCAI'83, vol 2, pp 650–652

Xie M, Wong RCW, Peng P, Tsotras VJ (2020) Being happy with the least: Achieving α-happiness with minimum number of tuples. In: ICDE'20, IEEE, pp 1009–1020

A Proof of properties 4, 5, 10, 11, 12 and 13

To prove properties 4 and 5 the idea is to find a bound, relative to the threshold K, to the number of answers to a query based on the number of answers to another query. To that end we define the following function. It is the set of mappings in the result of the evaluation of $Q \wedge t$ with their domains restricted to $\text{var}(Q)$:

$$f_{Q,t} : \begin{cases} [[Q \wedge t]]_D \to [[Q]]_D \\ \qquad \mu \mapsto \mu_{|\text{var}(Q)} \end{cases}$$

We additionally define a constant value A, the maximum number of inverse images of an element of $[[Q]]_D$ by $f_{Q,t}$, i.e., $\forall \mu, |\{\mu' \mid f_{Q,t}(\mu') = \mu\}| \leq A$. The constant A can be used to draw relationships between the number of answers of two queries:

Lemma 1 *If* $\frac{|[[Q \wedge t]]_D|}{A} > K$ *then* $|[[Q]]_D| > K$ *and* Q *fails.*

Proof From the definition of A, $|[[Q]]_D| \cdot A \geq |[[Q \wedge t]]_D|$ so $|[[Q]]_D| \geq \frac{|[[Q \wedge t]]_D|}{A}$. □

Proof of property 4

We show that if $\text{var}(Q \wedge t) = \text{var}(Q)$ then $A = 1$. Suppose $A > 1$, and consider $\mu_1 \neq \mu_2 \in [[Q \wedge t]]_D$ where $f_{Q,t}(\mu_1) = f_{Q,t}(\mu_2)$, i.e. $\forall x \in \text{var}(Q),\ \mu_1(x) = \mu_2(x)$. Since $\text{dom}(\mu_1) = \text{dom}(\mu_2) = \text{var}(Q \wedge t)$ and from our hypothesis, $\text{var}(Q \wedge t) = \text{var}(Q)$, then $\forall x \in \text{dom}(\mu_1) = \text{dom}(\mu_2),\ \mu_1(x) = \mu_2(x)$, i.e. $\mu_1 = \mu_2$. So $A = 1$ and from lemma 1 if $Q \wedge t$ fails Q fails. □

To proove the cardinality-based properties, we rely on the following lemma.

Lemma 2 (cardinalities) *For a triple pattern t, and a query Q, using either global, class or CS cardinality* $\text{cardinality}_{\max}(t, Q) \geq \max_{s \in \{\mu(s(t)), \mu \in [[Q]]_D\}} \text{count}(s, p(t)).$

Proof Call $S = \{\mu(s(t)), \mu \in [[Q]]_D\}$

In the case of global cardinality, $\text{global_cardinality}_{\max}(t, Q) = \max_{s \in \text{subject}(D)} \text{count}(s, p)$ and $S \subseteq \text{subject}(D)$. Therefore $\text{global_cardinality}_{\max}(t, Q) \geq \max_{s \in S} \text{count}(s, p(t))$

For class cardinality, consider $s \in S$, and a class C such that $\exists t_1 \in Q, s(t_1) = s(t) \wedge C \subseteq \text{domain}(p(t_1))$. According to the definition of s, $\exists o \mid (s, p(t_1), o) \in D$, so from the definition of *domain*, s belongs to all classes that are domains of $p(t_1)$, in particular, $s \in \text{instances}(C)$. So $\text{count}(s, p) \leq \max_{s \in \text{instances}(C)} \text{count}(s, p) = \text{class_cardinality}_{\max}(p, C)$. As this is true for all such classes C, $\text{count}(s, p) \leq \text{class_cardinality}_{\max}(t, Q)$, for all $s \in S$. Therefore, $\text{class_cardinality}_{\max}(t, Q) \geq max_{s \in S} \text{count}(s, p(t))$

For CS cardinality, consider $s \in S$. Call P the CS of s: $S_C(s) = P$. We have $\text{count}(s, p) \leq \max_{s', S_C(s') = P} \text{count}(s', p) = \text{CS_cardinality}_{\max}(p, P)$. The CS P contains the predicate of every triple $t' \in Q$ where $s(t) = s(t')$, so $\forall t' \in Q\, (s(t') = s(t) \Rightarrow p(t') \in P)$.
Therefore, $\exists P\, (\forall t' \in Q\, (s(t') = s(t) \Rightarrow p(t') \in P) \wedge \text{count}(s, p) \leq \text{CS_cardinality}_{\max}(p, P))$, so $\text{count}(s, p) \leq \text{CS_cardinality}_{\max}(t, Q)$ for all $s \in S$. Therefore, $\text{CS_cardinality}_{\max}(t, Q) \geq \max_{s \in S} \text{count}(s, p(t))$ □

Proof of property 5

We want to prove that $A \leq \text{cardinality}_{\max}(t, Q)$. Then, $\frac{|[[Q \wedge t]]_D|}{A} \geq \frac{|[[Q \wedge t]]_D|}{\text{cardinality}_{\max}(t, Q)}$. So if $\frac{|[[Q \wedge t]]_D|}{\text{cardinality}_{\max}(t, Q)} > K$, then using lemma 1 $\frac{|[[Q \wedge t]]_D|}{A} > K$, so Q fails.

Consider $\mu \in [[Q]]_D$, let $a(\mu)$ be the number of inverse images of μ by $f_{Q,t}$. Let us show that $a(\mu) \leq \text{cardinality}_{\max}(t, Q)$. Let μ_1 be an inverse image of μ by $f_{Q,t}$. μ_1 is entirely defined by the values of $\mu_1(v)$ for $v \in \text{var}(Q \wedge t)$. As $p(t)$ is not a variable and $s(t) \in \text{var}(Q)$, if $o(t)$ is not a variable, we can apply the previous property with $\text{var}(Q \wedge t) = \text{var}(Q)$. So let us consider that $o(t)$ is a variable. Therefore $\text{var}(Q \wedge t) = \text{var}(Q) \cup o(t)$. Additionally $\forall v \in \text{var}(Q)$, $\mu_1(v) = \mu(v)$, so μ_1 is entirely described by $\mu_1(o(t))$. Then we have $a(\mu) = |\{(\mu(s(t)), p(t), o(t)) \in D\}| = \text{count}(\mu(s(t)), p(t))$. From lemma 2, $\text{cardinality}_{\max}(t, Q) \geq \max_{s \in \{\mu(s(t)), \mu \in [[Q]]_D\}} \text{count}(s, p(t))$ so $a(\mu) \leq \text{cardinality}_{\max}(t, Q)$ and by definition of A, $A \leq \text{cardinality}_{\max}(t, Q)$ □

Proof of properties 10 and 12 We consider C an MFIS of A, and prove that C is an MFIS of $A\ OPT\ B$ *(resp. of* $A \cup B$*)*. Based on the definition of an MFIS, we know that C fails. Consider C' a superquery of C. C' is of the form A' or $A'\ OPT\ B'$ *(resp.* $A' \cup B'$*)* where $C \subseteq A' \subseteq A$ and $B' \subseteq B$. As C is an MFIS of A, and we have $C \subseteq A' \subseteq A$, A' fails. Since $|[[C']]_D| \geq |[[A']]_D|$ according to property 9 *(resp according to* $|[[P_1 \cup P_2]]_D| = |[[P_1]]_D| + |[[P_2]]_D|$*)* and $|[[A']]_D| > K$ then $|[[C']]_D| > K$ i.e. C' fails. So C is an FIS of $A\ OPT\ B$ *(resp. of* $A \cup B$*)*. Suppose that C is not an MFIS of $A\ OPT\ B$ *(resp. of* $A \cup B$*)*, i.e. there exists D an FIS of $A\ OPT\ B$ *(resp. of* $A \cup B$*)*, such that $D \subset C \subseteq A$. So D is also an FIS of A which contradicts the fact that C is an MFIS of A. Therefore C is an MFIS of $A\ OPT\ B$ *(resp. of* $A \cup B$*)*. □

Proof of property 11 Consider a succeeding query of $A\ OPT\ B$, which can be of the form A' or $A'\ OPT\ B'$, with $A' \subseteq A$ and $B' \subseteq B$. From property 9 if $A\ OPT\ B$ succeeds A' succeeds. From the XSS definition, A' is either an XSS of A or the subquery of an XSS of A. □

Proof of property 13 Consider a succeeding query of $A \cup B$, which can be of the form A', B' or $A' \cup B'$, where $A' \subseteq A$ and $B' \subseteq B$. As $|[[P_1 \cup P_2]]_D| = |[[P_1]]_D| + |[[P_2]]_D|$, if $A' \cup B'$ succeeds then A' and B' both succeed. From the definition of XSS, this means that A' either is an XSS of A or is the subquery of an XSS of A and likewise for B'. □