Towards practical federation of BGPs in the presence of blank nodes

Jonas Halvorsen
Norwegian Defence Research Establishment, Norway
Institute for Informatics, University of Oslo, Norway
jonas.halvorsen@ffi.no

Audun Stolpe
Norwegian Defence Research Establishment, Norway
audun.stolpe@ffi.no

ABSTRACT

Distributed query processing, when blank nodes might occur in sources and only the signature of sources are known, may in the worst case require that all possible query partitions be evaluated in order to ensure soundness and completeness of answers. The work presented in this paper attempts to push the boundary as to what query sizes and structures lies within practical feasibility. The presented approach utilizes semantic information obtained by probing sources for occurrences of blank nodes, interleaved in a recursive algorithm for calculating restricted growth strings, in order to detect and abort unfruitful branches in the partition generating process. The approach is evaluated against a well-known SPARQL benchmark, modified for the distributed case, and tentative conclusions regarding the effectiveness are drawn.

CCS CONCEPTS

• **Information systems → Information retrieval query processing**; • **Theory of computation → Database query processing and optimization (theory)**;

ACM Reference Format:
Jonas Halvorsen and Audun Stolpe. 2018. Towards practical federation of BGPs in the presence of blank nodes. In *WIMS '18: 8th International Conference on Web Intelligence, Mining and Semantics, June 25–27, 2018, Novi Sad, Serbia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3227609.3227644

1 INTRODUCTION

Distributed SPARQL processing aka. SPARQL federation is all but implicit in the Semantic Web vision. The Semantic Web is designed to promote and enable data integration from all over the internet, depending on opportunity and relevance for a given task, and distributed SPARQL processing is one way of automating it.

The term "distributed SPARQL processing" may tentatively be understood as the problem of answering a query by decomposing and distributing it to different contributing sources, with the goal of combining the partial answers into a global answer that is semantically equivalent to what would be obtained were the data held in a single source.

The most general problem of federation is the case where nothing about the structure and content of the data in the contributing sources is assumed to be known. In particular, one cannot rule out the presence of blank nodes in the data. The literature has not devoted much attention to the impact of blank nodes on the federation process. In most cases blank nodes are ruled out by stipulation. Indeed, the only study we know of that attempts to confront the problem is the prequel [17] to this paper, where it is shown that blank nodes change the federation problem quite profoundly, and have major ramifications for sound and complete query answering strategies.

In a nutshell, the situation is this: in conformity to the SPARQL specification, blank node identifiers are only in scope in the execution context that produced them. That is, they cannot be re-identified between different runs of different or even identical queries. So if the same blank node is bound to a variable in two different runs of two possibly different queries, then irrespective of the form of those queries, there is no way to tell that the solution bindings for that variable were generated by the very same node. In distributed terms, this creates a tension between picking up on cross-source joins, on the one hand, and on local joins involving blank nodes on the other. For example, if one partitions a query into, say, singleton triple patterns, then local joins involving blank nodes will be missed. If, on the other hand, one sends too large chunks of a query to a single source then remote joins will be missed. The only way to solve this in the general case where no knowledge about the sources is assumed is to "brute-force" the possibilities, splitting a query in all possible ways and trying each of these partitions. For all but trivially small SPARQL queries, this turns out not to be feasible. To put it into perspective, a query consisting of 10 patterns will, in the absence of heuristic countermeasures, require that 115 795 distinct partitions be evaluated. This is of course, due to the sheer combinatorial complexity of calculating partitions. This combinatorial problem and the ways of dealing with it is the mathematical backdrop against which zero-knowledge SPARQL federation must be studied.

It is the purpose of the present paper to make a first stab at this. We shall try to get the combinatorial explosion under control using heuristics to prune the space of partitions a federator needs to compute. There's nothing to be done about the worst case of course, so we'll settle for progress on realistic and useful examples of SPARQL queries. The discussion applies only to conjunctive SPARQL queries, aka. basic graph patterns (BGPs), which may be considered a natural theoretical baseline. Given all that has been said so far, it should come as no surprise that we explicitly allow blank nodes to occur in the data. We refer to this problem of federating BGPs over data that may contain blank nodes as *general* BGP federation.

ACM ISBN 978-1-4503-5489-9/18/06…$15.00
https://doi.org/10.1145/3227609.3227644

Given the results from [17], it isn't possible to make any progress with this unless one leverages some form of knowledge about the data sources. Yet, there are tangible advantages to the zero-knowledge approach to federation that it'd be desirable to retain. For one, the more unassuming a theory is the more generally applicable it is. Assuming *no* knowledge makes the theory applicable to all other special cases, but as it looks, impracticable. The only way out of this bind is to *acquire* knowledge as the federation process unfolds. The idea explored here is that this can be done by evaluating SPARQL probes at the fringe of a search for restricted growth strings (RGSs).

There is a one-to-one correspondence between restricted growth strings over of length n and the set of partitions of a set of the same cardinality. A number of well studied algorithms for listing RG strings exist already, see [12, 13]. The present paper aims to exploit one such algorithm which, speaking in terms of partitions rather than strings, has the property that combinations of subsets are built up in a recursive exploration of different ways to obtain a partition. Given a negative answer to a probe, the recursion can be terminated, in effect ignoring the entire set of partitions in the leaves under that point . The probes in question take the form of SPARQL ASK queries designed to gather information about the position of blank nodes in the data, cf. Sec. 6.

To this main theme we add two subordinate ones concerned with preprocessing the input before it goes in to the RGS computation. One is based on the concept of an exclusive group from [15], that is, on query patterns that are answerable by at most a single contributing source. Exclusive groups can be treated as atoms in the combinatorics of calculating partitions in the sense that it is never, from a semantical point of view, necessary to split them up. We exploit this notion by treating exclusive groups as single elements, thereby simplifying the intial combinatorial problem. The second preprocessing procedure considers the effect of ordering the elements of the seed set. Ordering can have a measurable effect insofar as it can force probes to fail early. It cannot reduce the number of computed partitions, but it reduces the number of recursive calls that has to be made to compute them.

An empirical evaluation is included towards the end of the paper. It suggests that the effect of these countermeasures goes some way towards bringing general BGP federation within the limits of practicability for medium-sized SPARQL queries. The experiments are performed against a standardized SPARQL benchmark, adjusted for the distributed case of study, with query patterns that capture fairly typical use cases.

2 A DESCRIPTION OF THE PROBLEM

2.1 Nomenclature

RDF graphs. Let U, B and L denote pairwise disjoint infinite sets of IRIs, blank nodes, and literals respectively. In conformity with the nomenclature of [3], $\mathbb{IL}$ abbreviates $I \cup L$ and T abbreviates $I \cup B \cup L$. T is the set of *RDF terms*, elements of which will be denoted individually by u_i, v_j. RDF triples are defined as usual as elements of $(I \cup B) \times I \times (I \cup B \cup L)$, and are symbolized a_i. where the 'a' is meant to stand for '*a*ssertion'. An RDF graph is a finite set of RDF triples. RDF graphs are denoted by G and H, and sets of RDF graphs by $\mathscr{G}$. RDF graphs may also be referred to as RDF *sources*.

SPARQL queries. Turning now to SPARQL queries, the analysis in the present paper will be limited to the select-project-join fragment, thus essentially to basic graph patterns (henceforth BGPs). Blurring the line between syntax and semantics, BGPs will be treated as sets of triple patterns. More specifically, where V is an infinite set of variables x_i, elements $t_i, t_j \in (IL \cup V) \times (I \cup V) \times (IL \cup V)$ are *triple patterns*[1] and $\{t_i, t_j\}$ is a BGP.

Query evaluation. The evaluation of a BGP P over an RDF graph G will be denoted by $[\![P]\!]^c_G$, and elements of $[\![P]\!]^c_G$, called *solutions*, by μ_c. Sets of solutions will be denoted by Ω. There is a minor deviation here from standard practice, namely the parameter c which names the execution context in which a query is run. It is essentially a way of standardizing apart names of blank nodes *in answer sets*—a necessary device for keeping distributed query processing sound. Formal details can be found in [17]. Explicit reference to contexts may be omitted when it is not important.

For a *set* of graphs $\mathscr{G}$, the evaluation $[\![P]\!]_{\mathscr{G}}$ of P is understood to be the *distributed evaluation* of P over $\mathscr{G}$, that is, it refers to the outcome of the federation process according to the semantics given in [17]. In contrast, let $m(\mathscr{G})$ denote the *merge* of the graphs in $\mathscr{G}$, that is, $m(\mathscr{G})$ is the single graph that results from taking the union of all elements of $\mathscr{G}$ after standardizing apart blank nodes from different graphs. Then $[\![P]\!]_{m(\mathscr{G})}$ is just the evaluation of P over the single source $m(\mathscr{G})$. The two are emphatically not the same. Indeed, the semantics of federated zero-knowledge query processing is precisely the conditions under which $[\![P]\!]_{\mathscr{G}} = [\![P]\!]_{m(\mathscr{G})}$. The left-to-right inclusion says that the federation process is sound, and the converse inclusion says that it is complete.

2.2 Motivating example

It seems natural to expect, by default at least, that a distributed query processor should aim to return all answers as warranted by the union of the sources. To illustrate what we mean with this, consider the two RDF graphs in Figs. 1 and 2 respectively. The abovementioned graphs encode information regarding members of the European Parliament (MEP), as found in the LinkedEP dataset produced by the Talk of Europe project [18], a dataset covering plenary debates held as well as biographical information regarding members of parliament. More specifically, source A encodes information regarding the MEP Eva Joly and her political functions, while source B encodes information regarding MEP Carl Schlyter. From the data, we see that they represent different national parties but belong to the same EU political party (Europarty). However, the information in source A alone is *not* enough to conclude that Eva Joly is associated with a Europarty, as EFA is not typed as such. This missing piece of information is, however, present in source B. Thus, when the sources are merged, as shown in Fig. 3, the political institutions are all appropriately typed. Hence, posing the query in Fig. 4, asking for the name of the MEPs in the EU parliament that are politically affiliated with a Europarty (not all MEPs are), as well as the party name, produces the answers in Fig. 6.

[1]Literal subjects are not allowed to occur in RDF triples, yet the SPARQL 1.1 specification allows them in triple patterns.

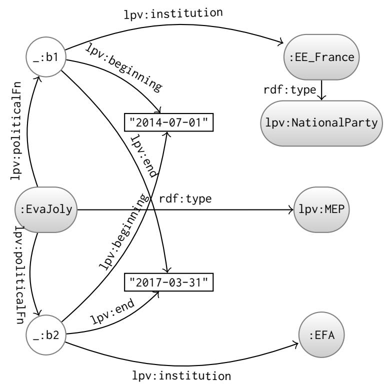


Figure 1: RDF source A

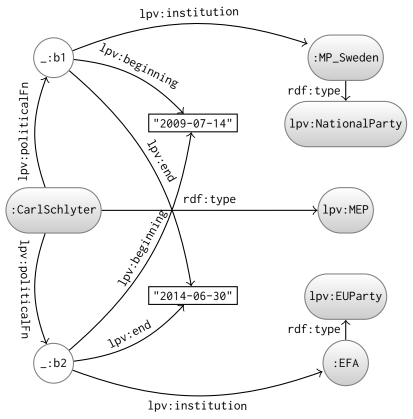


Figure 2: RDF source B

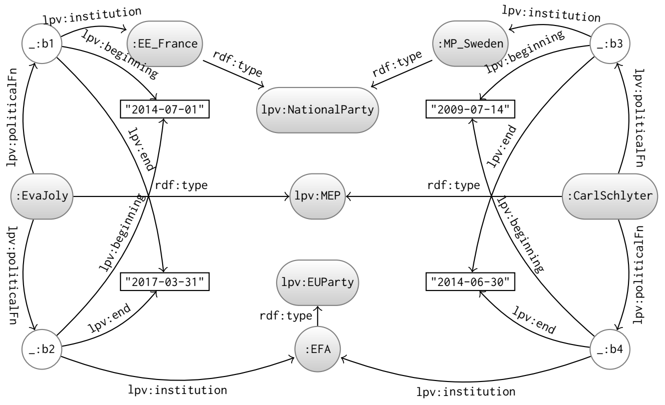


Figure 3: The union of sources A and B modulo renaming of blank nodes.

Now, if we only evaluate the query in Fig. 4 against each source separately, for so to take the union of the results, we get an incomplete set of answers as shown in Fig. 5. In other words, it is clear that the sum of the whole is more than the sum of its separate parts. That is, the total amount of information contained by

```
SELECT ?person ?party WHERE {
  ?person a lpv:MEP.
  ?person lpv:politicalFn ?x.
  ?x lpv:institution ?party.
  ?party rdf:type lpv:EUParty.}
```

Figure 4: Get MEP and EU party

the two sources combined, resides not only in what each of them can contribute separately, but in also in the combination or join of elements across sources. In other words, the query cannot simply be executed as a whole against each source—that is too coarse. It must rather be split up into parts tailored to capture the cross-site joins.

Unfortunately, there is a complicating factor that blocks any straightforward realization of this idea, namely the presence of blank nodes in join positions. More specifically, sources A and B utilize blank nodes to represent complex attributes in the form of statements about statements, as recommended by the Semantic Web Best Practices and Deployment Working Group. In this case, that "X had a political affiliation to institution Y between dates A and B" is codified using blank nodes. In the distributed case, such a join, if it is not handled with special care, will quickly become a drain through which information will leak. As described in detail in [17], this is due to the fact that anaphoric reference is lost whenever the same blank node is processed in two separate execution contexts. According to the SPARQL 1.1 specification, every distinct query constitutes a distinct and sealed scope for blank node identifiers, which means that a blank node from one execution context cannot be referenced in another. In other words, blank nodes are similar to existential variables in the sense that they are anaphors within the same quantificational context only. Now, a blank node that receives different names in different query execution contexts obviously cannot be used for cross-site joins, so there it is.

It is worth emphasizing that none of the more straightforward and better known query-decomposition strategies from the literature, such as the *even decomposition*, so called in [16] as implemented in DARQ [10], and the *standard decomposition* as implemented in FedX [15] solve this problem.

Exemplifying, the *even decomposition* will evaluate each triple pattern (from the *global* query, let's call it) against every source that *may* contain an answer for it (meaning that the RDF property from the triple pattern in question occurs in that source). For instance, the even decomposition will evaluate both of the triple patterns `?person lpv:politicalFn ?x` and `?x lpv:institution ?party` from the query in Fig. 4 separately against each of A and B. Collecting the solutions in separate tables, we have the answer sets in Figs. 7 and 8, where the identifiers for blank nodes have been given distinct subscripts *c* and *d* to signify that they are not to be treated as the same names. Now, as these tables do not join, the even distribution produces no answer to the example query, not even the ones that derive from the same source. This time it comes down to the fact that query is split too finely.

Taking stock, these examples can be taken to show the following: If answering a query involves joins on blank nodes, then the granularity of the decomposition of that query matters a great deal. If the query is split too finely, then answers from a single source

?person	?party
:CarlSchlyter	:EFA

Figure 5: Union of answers over A and B

?person	?party
:EvaJoly	:EFA
:CarlSchlyter	:EFA

Figure 6: Answer over the merge of A and B

?person	?x
:EvaJoly	$\_:b1_c$
:EvaJoly	$\_:b2_c$

Figure 7: `?person lpv:politicalFn ?x` over A

?x	?party
$\_:b1_d$	:EE_France
$\_:b2_d$	:EFA

Figure 8: `?x lpv:institution ?party` over A.

may be lost due to the loss of join information linking the partial answers. If on the other hand the query is split too coarsely, then cross-site joins may be lost. From a semantical point of view, distributed query answering is essentially about balancing these two opposing forces.

3 THE EXISTENCE OF DECOMPOSITIONS

The partition immediately below gives a decomposition of the query in Fig. 4 that produces a complete answer.

P_1 := {?person lpv:politicalFunction ?x.,
    ?x lpv:institution ?party.}
P_2 := {?person a lpv:MEP.}
P_3 := {?party a lpv:EUParty.}

The crucial thing about this decomposition is first, that it groups together those triple patterns that match a join on a blank node, thus ensuring that the join arguments are kept within one and the same execution context so as not to lose anaphoric reference (P_1). Secondly, it is also of crucial importance that all other triple patterns are shipped as singletons, or else cross-site joins would be lost (P_2 and P_3).

As it happens, there is only one solution μ to the query in Fig. 4 over the sources in Figs. 1 and 2. The partition P_1-P_3 corresponds to this solution—the reader may verify this for himself—in the sense that the join of the respective unions of evaluating each subquery over the sources that can answer yields the set of bindings $\{\mu\}$.

In the general case that a query has more than one solution—'having a solution' should here be understood as having an answer in the *merge* of the contributing sources—correspond to the same partition. Indeed, it is not even entirely obvious that there is a partition for every solution. The demonstration that there is, relies on the concepts of a b-component and a b-connected set:

Definition 3.1 (b-connectedness). Let G, $\{a\}$ be RDF graphs, then

(1) $\{a\}$ is b-connected
(2) $G \cup \{a\}$ is b-connected if G is b-connected and G and a share a blank node.

A b-component, on the other hand, is a subquery that matches a maximally b-connected subgraph modulo some solution μ:

Definition 3.2 (b-component). Let $\mu_c \in [\![P]\!]^c_{m(\mathscr{G})}$ and suppose $P_i \subseteq P$. Then P_i is a b-component of P relative to μ_c iff $\mu_c(P_i)$ is a maximal b-connected subset of $\mu_c(P)$.

Note that b-connected sets are RDF graphs, whereas b-components are SPARQL query patterns. Note also that subquery P_i is a b-component *relative to* a particular solution μ. We shall say that μ *induces* the the b-component P_i. Now, let μ_c be a solution to P in a graph G and let $f(\mu_c, P)$ denote the set of b-components of P modulo μ_c. Then f is a function and $f(\mu_c, P)$ partitions P. Indeed $f(\mu_c, P)$ selects the partition that corresponds to μ_c.

THEOREM 3.3. *Let $\mathscr{G} := \{G_i\}_{i \in I}$ be a set of sources of RDF graphs and let $\mu_c \in [\![P]\!]^c_{m(\mathscr{G})}$. Put $f(\mu_c, P) := P_1, \ldots, P_k$. Then there is a set $\{m, \ldots, n\} \subseteq I$ such that there is a $\mu' \in [\![P_1]\!]^{c_m}_{G_m} \bowtie \ldots \bowtie [\![P_k]\!]^{c_n}_{G_n}$, for any distinct set of execution contexts $c_m, \ldots, c_n$, and $\mu_c(P) \dashv\vDash \mu'(P)$.*

Theorem 3.3 shows that every solution to a query P in the merge of a set of sources $\mathscr{G}$ is RDF equivalent (symbolized by $\dashv\vDash$) to a solution contained in the join of the separate evaluation of the cells of *some* partition of P. These two solutions are alphabetic variants of each other obtained by substituting names of blank nodes for names of blank nodes. In other words, there is always a solution to be had by federation if there is a solution in the merge of the sources.

Yet, Theorem 3.3 is just an existence theorem. It does not provide a recipe for actually finding the partition corresponding to a solution. Indeed, the theorem presupposes that we already have the solution and works backwards from there to identify the corresponding partition. Moreover, although every solution induces a partition, the converse is not true. For any given set of sources and a query, there will usually be plenty of partitions with cells that have no answer in any of the sources. The simplest example is a pair of non-overlapping RDF graphs, a query that requests information from both, and the partition that contains only the whole query. These partitions, and all the cells in it, are redundant from the point of view of the evaluation process. They do not threaten soundness, they just don't return solutions.

Taking stock, one might say that zero-knowledge federation (of general BGPs) is the task of searching the space of all possible partitions of a query until all the elements that produce solutions have been found. Since the redundant partitions do not jeopardize soundness, the brute force approach that computes *all* partitions, is one way of doing it. But clearly, it is not a feasible one in practice, since a set of $n + 1$ elements has B_{n+1} (the n+1-th Bell number) partitions where the B_{n+1} is given by a recurrence relation involving binomial coefficients:

$$B_{n+1} = \sum_{k=1}^{n} \binom{n}{k} B_k$$

So, for example a query with 10 triple patterns has 115975 partitions.

4 COMPUTING PARTITIONS

As noted earlier, a considerable amount of literature has been produced regarding efficiently generating all set partitions. Here, the work done on restricted growth strings (RGS) [11, 13] seems to currently yield the best results.

The restricted growth strings of length n are the strings represented as a sequence of non-negative integers $s_0, \ldots, s_{n-1}$ s.t.

$$s_0 = 0$$
$$s_i \leq 1 + max\{s_0, \ldots, s_{i-1}\}$$

The relevance of these strings wrt. set partitions, as described in [11] and [12], is that there is a bijection between the partitions of sets of the form $\{1, \ldots, n\}$, and the restricted growth strings of length n. Specifically, if we order the blocks of a partition according to the least element in each block, label the blocks incrementally starting from 0, and for $1 \leq i < n$ assign s_i to the label of the block that i occurs in, we get the associated RGS. E.g., the corresponding RGS string representation of the partition $\{\{1, 4\}, \{2\}, \{3\}\}$ is 0120.

Now, in order to utilize this for our purpose, we must be able to associate a BGP P of size n with a string representation of the form $\{1, \ldots, n\}$. We can get this by simply assigning any linear ordering of the triple patterns that constitute P for so to associate this sequence position-wise with RGS strings. That is, assuming $t_0, \ldots, t_n$ is a sequence of triple patterns, and $s_0, \ldots, s_n$ is an RGS string of length n, then grouping t_i into sets based on the value of s_i yields a partition of P. E.g., the triple pattern sequence t_0, t_1, t_2, t_3 and the RGS string 0120 defines the partition $\{\{t_0, t_3\}, \{t_1\}, \{t_2\}\}$.

Although there are many efficient RGS algorithms available, we are after one that has the feature that it recursively expands blocks. This yields a controlled behavior of fringe expansion that allows pruning at intermediate nodes in the tree. Generating the sequences of RGS recursively, as per Algorithm 4.22 in [11], yields the trace tree (left to right) shown in Fig. 9.

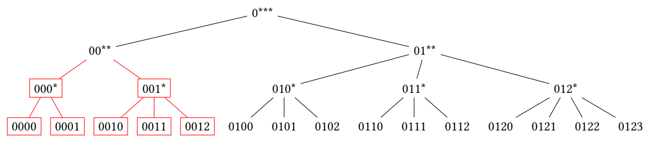


Figure 9: Pruning RGS computation tree

As hinted to earlier, there is an important pruning opportunity shown in this computation tree. For if we can deem that an intermediate node in the computation tree yields a block that a) will not contribute to the final answers, and b) holds for any superset block as well, then we can prune away the branches from that node. Exemplified, say the intermediate allocation $\langle 00**\rangle$ is such that the position-wise translation wrt. the global query P, denoted $P[00**]$, yields $P[00**] = \{\{t_0, t_1\}\}$, where $\{t_0, t_1\}$ is a BGP satisfying the abovementioned condition. By pruning at the identified node in tree, we can then prevent the partitions encoded in the leafs of this branch from being generated, as shown by the red coloring in the figure. The following section describes a probe that identifies such an opportunity.

5 PROBING THE FRINGE

As we have characterized it, general BGP federation is at bottom the combinatorial problem of computing those partitions of a query that correspond to solution mappings.

From Theorem 3.3 we know that these partitions have the form $f(\mu, P)$ where P is the global query pattern and μ is a solution. It follows by contraposition that if a partition is not of this form then it is redundant for query answering.

Further, recall that all elements of $f(\mu, P)$ are b-components wrt. μ. Hence, by the contrapositive again, if a partition contains an element that fails to be a b-component wrt. every solution—call such an element a *redundant block*—then that partition does not correspond to a solution under any of them. We may therefore infer that if a family of subsets of a query pattern contains an element that is not a b-component wrt. to *any* solution, then that suffices to conclude that that family of sets as such does not correspond to a solution.

The fringe probe, as it will henceforth be called, tests each of the elements in the respective families of sets at the fringe of the RGS computation tree. It tries to find a member that is demonstrably not a b-component under any solution, in order to dismiss that family as a candidate partition. For each triple pattern in the set in question the fringe probe asks each RDF source whether that source can provide a blank node for a variable in that triple pattern. If not, then it follows from what was said above that the set to which the triple pattern belongs is not a b-component under any solution. Hence, no combination of subsets in which it occurs corresponds to a solution. There is one important proviso though; the subset in question must be of cardinality higher than 1, since all singletons are b-components trivially.

It should be emphasized that whether a BGP P_i is a b-component wrt. a solution cannot be determined by syntactic means from the form of the graph pattern alone since a BGP need not itself be connected in order to match a b-connected graph. The concept of a b-component is an irreducibly semantic notion, so probes are necessary.

Moving towards a formal regimentation of these ideas, we begin by noting that b-connected sets satisfy the condition that all elements have a blank node in subject or object position. For easy reference let the set of all such sets be defined as follows:

Definition 5.1.

$$\mathbb{B}_{tr} =_{df} \{(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L) | s \in B \vee o \in B\}$$

The fringe probe, interchangeably referred to as the Δ-probe, can now be defined abstractly as follows:

Definition 5.2. If $Vars(P_i) \subseteq dom(\Omega)$, then

$$\Delta(P_i, \Omega) =_{def} \{\mu \colon \mu \in \Omega \wedge \mu(P_i) \subseteq \mathbb{B}_{tr}\}$$

The test case of interest is $\Delta(P_i, [\![P_i]\!]_{\mathscr{G}_i}) = \emptyset$. This says that no valuation of P_i over any RDF source binds a variable in a triple pattern $t \in P_i$ to a blank node. Such a probe is a *negative probe* and signals, or so we have surmised, that any partition in which P_i occurs can be dismissed as a candidate partition. Proposition 5.3 confirms this:

Proposition 5.3. *If $P_i \subseteq P$, $|P_i| > 1$, for any $\mu \in [\![P]\!]^c_{m(\mathscr{G})}$ then*

$$P_i \notin f(\mu, P) \text{ if } \Delta(P_i, [\![P_i]\!]^d_{G_n}) = \emptyset \text{ for every } G_n \in \mathscr{G}$$

Negativity is monotone wrt. set inclusion. That is, if $P_i \subseteq P_j$ and P_i is the BGP of a negative probe, then so is P_j. This is recorded in Proposition 5.4.[2] Fig. 10 illustrates this simple but useful fact. As shown, if the set $\{t_1, t_2\}$ is dismissed, then all the partitions marked in red can be disregarded too, as they all contain supersets of it.

Proposition 5.4. *If $P_i \subseteq P_j$, and $|P_i| > 1$, then*

$$\Delta(P_j, [\![P_j]\!]^c_G) = \emptyset \text{ if } \Delta(P_i, [\![P_i]\!]^d_G) = \emptyset$$

This monotonicity property fits hand in glove with the RGS algorithm selected for this paper, insofar as the algorithm has the following properties: the algorithm computes families of subsets of an input set. As the computation tree expands, the family of sets and each of its elements increase in size (though not necessarily strictly). This process continues until the input set is finally exhausted in the leaves of the computation tree (cf. Fig. 9). Proposition 5.4 therefore implies that a negative probe automatically dismisses all families of sets below and including the current one, in particular the partitions in the leaf nodes. In other words, a negative probe warrants the termination of the RGS recursion, which in turn reduces the size of the combinatorial space that will have to be explored.

It is actually possible to push the envelope a bit: Proposition 5.4 tells us rather more than we have exploited so far, namely that a negative probe dismisses all families that contain a superset of the negative probe *irrespective of their location in the computation tree.* These families don't have to be positioned below the point at which the probe is performed in the tree, but might equally well occur in sibling branches. This is a possibility that the RGS algorithm allows. A negative probe can be used to trim all these branches. Implementation-wise, this would involve caching and reapplying probe results. Standard tabling techniques akin to memoization will do. More on this in Sec. 6.

To round off this section, we mention that the Δ-probe has a rather obvious translation to a SPARQL ASK query: the graph pattern P_i of the probe translates into the graph pattern of the SPARQL ASK query, and each triple pattern $t \in P_i$ adds a filter expression checking that triple pattern against the data for occurrences of blank nodes, cf. Fig. 11.

6 COMPUTING QUERY DECOMPOSITIONS

The previous section defined a probe that identifies pruning opportunities. Furthermore, the RGS computation process outlined has the property that combinations of subsets are built up recursively in an iterative exploration of different ways to obtain a partition. It follows that if a particular combination of subsets somewhere in the RGS computation tree is discounted by a probe, then so are subtrees. This is due to the property of the RGS algorithm that any block in the computation tree is included in some block in the level below, hence in effect, allows us to ignore an entire class of partitions. In addition to this, the result of probes are also applied for pruning in sibling branches using a tabling technique. That is, a subtree in the RGS computation tree is liable to be discounted by probes in sibling branches if it contains a block that is a superset of probe pattern.

The probes themselves take the form of SPARQL ASK queries designed to retrieve information obtained about the position of blank nodes in the data. This information is used to determine when a cell in a partition is too coarse or too fine–if it is found to be either then none of the partitions that stem from that node in the RGS computation tree need to be expanded, so the combinatoric space shrinks.

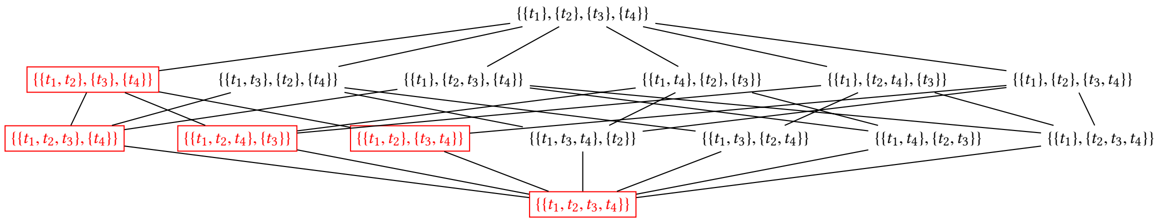
The RGS-generating algorithm developed for the present purpose is shown in Algo. 1. It is based on a well-known recursive RGS-generating algorithm found in [11] (Algo. 4.22), which runs in constant amortized time. We extend this algorithm by introducing a cut-off point that terminates the recursion, conditioned by a function that identifies if a sequence update yields a redundant block. The main mechanism of the algorithm is that it recursively updates positions in the input sequence, in a controlled way that ultimately produces RGS strings of the correct length. The algorithm works position-wise from left-to-right. That is, it explores all allowed updates of position l before recursively moving on to $l + 1$. The variable m acts as an upper limit to what position l can be set to, essentially tracking the max number of blocks allowed, based on the largest value seen so far in the sequence. This is used to ensure that the sequence, up to position l, satisfies the RGS condition.

Now, in order to keep the number of blocks that need to be checked to a minimum, updating a value in sequence position is performed at the onset of the function (lines 2-3) rather than in the argument passed to the recursive call. By doing it this way, the redundancy-checking function has enough information available to deduce what new blocks were generated, hence only focus on checking these wrt. redundancies (line 4). Lines 6-7 returns a valid RGS sequence, by identifying that it has reached a leaf node of the correct depth (i.e. sequence length same as the triple pattern sequence). Lines 9-17 implements the main RGS generating process in the way outlined above, and described in detail in [11], by updating the sequence assignments recursively, altering the position pointer and the max value of allowed allocations in a controlled matter. The call REST-GROWTH(1,0,0,A) where A is a sequence of length $|P|$ and $A[i] = nil$ initializes the partition generation process, and the output is a set of RGSs.

The actual process of identifying redundant blocks is captured by Algos. 2 and 3. More specifically, Algo. 2 yields a redundancy identifying function, in the correct form of a two-place function as required in Algo. 1, if partially instantiated with a sequential ordering of disjoint BGPs that constitutes a partitioning of the global query P into atomic elements (e.g. triple pattern singletons). That is, identifying redundancies using the method outlined earlier, depends on being able to translate an RGS to a set of BGPs (i.e. a distribution), which is what the BGP sequence provides by position-wise mapping.

[2] Note that even though $P_i \subseteq P_j$, it is still not the case that $[\![P_i]\!]^c_G \subseteq [\![P_j]\!]^d_G$ since the execution contexts are not the same—the latter sets have no blank nodes in common.

Figure 10: Partition lattice


$\{\{t_1\},\{t_2\},\{t_3\},\{t_4\}\}$
$\{\{t_1,t_2\},\{t_3\},\{t_4\}\}$
$\{\{t_1,t_3\},\{t_2\},\{t_4\}\}$
$\{\{t_1\},\{t_2,t_3\},\{t_4\}\}$
$\{\{t_1,t_4\},\{t_2\},\{t_3\}\}$
$\{\{t_1\},\{t_2,t_4\},\{t_3\}\}$
$\{\{t_1\},\{t_2\},\{t_3,t_4\}\}$
$\{\{t_1,t_2,t_3\},\{t_4\}\}$
$\{\{t_1,t_2,t_4\},\{t_3\}\}$
$\{\{t_1,t_2\},\{t_3,t_4\}\}$
$\{\{t_1,t_3,t_4\},\{t_2\}\}$
$\{\{t_1,t_3\},\{t_2,t_4\}\}$
$\{\{t_1,t_4\},\{t_2,t_3\}\}$
$\{\{t_1\},\{t_2,t_3,t_4\}\}$
$\{\{t_1,t_2,t_3,t_4\}\}$

```
{?s p:p1 ?y.,
 ?s p:p2 ?z.}
```

```
ASK WHERE
{?x1 p:p1 ?y1.
?x2 p:p2 ?y2.
filter(isBlank(?x1) || isBlank(?y1))
filter(isBlank(?x2) || isBlank(?y2))}
```

Figure 11: ASK probe

Algorithm 2 Check if new alloc introduces a redundant block

```
Require: Input:
    $\vec{P}$: sequence of disjoint BGPs exhausting the global query $P$.
Require: Functions:
    RG2BGP$(rgs, ps)$: returns BGPs, given RGS and BGP sequence
1: function SOME-REDUNDANT$(newseq, oldseq, \vec{P})$
2:     $\mathbb{D} \leftarrow$ RG2BGP$(newseq, \vec{P}) \setminus$ RG2BGP$(oldseq, \vec{P})$
3:     return $\{D \in \mathbb{D} \colon |D| \geq 2 \land \neg$VIABLE$(D)\} \neq \emptyset$    ▷ true/false
4: end function
```

Algorithm 1 Generating partitions

```
Require: Function:
    SOME-REDUNDANT$(x, y) \leftarrow$ SOME-REDUNDANT$(x, y, \vec{P})$
Require: Input:
    seq: current RGS
    l: position in seq to be updated
    x: value to be assigned to $seq_l$
    m: largest value allocated so far
1: function REST-GROWTH$(l, x, m, seq)$
2:     $nseq \leftarrow seq$
3:     $nseq[l] \leftarrow x$
4:     if SOME-REDUNDANT$(nseq[1,l], seq[1,l])$ then
5:         return $\emptyset$
6:     else if $l = len(seq)$ then          ▷ Length of seq is constant
7:         return $\{nseq\}$                   ▷ Valid distribution
8:     else
9:         $iExp \leftarrow \emptyset$          ▷ Solutions generated in lower level
10:        $mExp \leftarrow \emptyset$        ▷ Solutions generated in sibling branch
11:        for $i = 0$ to $m$ do
12:            $iExp \leftarrow iExp \ \cup$ REST-GROWTH$(l + 1, i, m, nseq)$
13:        end for
14:        if $m < (n - 1)$ then
15:            $mExp \leftarrow$ REST-GROWTH$(l + 1, m + 1, m + 1, nseq)$
16:        end if
17:        return $iExp \cup mExp$
18:    end if
19: end function
```

In Algo. 2, lines 2-3 are responsible for a) identifying new blocks that were not in the former allocation, and b) checking these (of size > 1) for redundancies. Here, the importance of initializing the input sequence that kicks off the RGS-algorithm to *nil* rather than 0 comes into play, as the former will not count as block allocations when extracting BGPs (the function RG2BGP). If any new block in the sequence allocation is deemed redundant, then the sequence captures a family of likewise redundant patterns that should not be further explored.

Algo. 3 defines the part that handles both the Δ-probing as well as a tabling-approach that facilitates pruning in sibling branches. Prerequisites to calling this function is a) a set of endpoints, used for probing, and b) a global set of BGPs that have been deemed redundant (aka. nogoods, initially empty). The first step (lines 2-3) captures the secondary pruning opportunity by checking if the block in question is a superset of another already deemed redundant. If so, then the subject block is redundant as well. Lines 4-6 are responsible for the Δ-probing, as well as updating the collection of redundant patterns. That is, we check that there exists at least one answer that, when taking the image wrt. the block in question, yields a triple without any blank nodes. This involves a SPARQL ASK probe against each relevant source. We note that a source overlap check is essentially baked into this stage, since if two blocks do not have any sources in common, then the Δ-probe vacuously returns an empty set, i.e. the union of blocks is redundant. If the check fails, then the block is added to the set of nogoods, hence building up knowledge regarding block redundancies on-the-fly, before the function returns $false$. Otherwise it returns $true$ to indicate that the block is non-redundant and must be included in partitions. Note that the set of nogoods does not strictly contain all blocks that have been deemed redundant (only the probed ones). Rather, it is the closure of this set wrt. $\subseteq$ (Proposition 5.4) that does this, hence the need for lines 2-3.

Finally, we note that the output of the RGS-algorithm is a set of RGSs and not a set of distributions. Applying the translation function RG2BGP, noted in Algo. 3 to the elements in this set yields the set of distributions sought.

Algorithm 3 Block redundancy check

```
Require: Global variables:
    $S$: set of sources
    $*nogoods*$: set of nogood BGPs
Require: Functions:
    $\Delta$-QUERY($P$): create a $\Delta$-query from BGP $P$
    PROBE($P, s$): SPARQL ASK for pattern $P$ against source $s$
Require: Input:
    $P_i$: a BGP
1:  function VIABLE($P_i$)
2:      if $\exists_{P_j \in *nogoods*} P_j \subseteq P_i$ then
3:          return false
4:      else if $\{s \in S : \text{PROBE}(\Delta\text{-QUERY}(P_i), s)\} = \emptyset$ then
5:          $*nogoods* \leftarrow *nogoods* \cup \{P_i\}$
6:          return false
7:      else
8:          return true
9:      end if
10: end function
```

7 PRE-PROCESSING

The final group of heuristics we shall look into, is that of problem pre-processing. First, we will explore a pre-processing stage that simplifies the problem that is fed to the partition generating algorithm, based on identifying sets of triple patterns that are exclusive to a single source. Secondly, we explore the effect that specific sequential orderings of triple patterns have on the efficiency of the RGS computation.

7.1 Exclusive groups

The theory of federation developed in [16] and [17] and further explored in this paper, is based on the idea of using *signatures*—aka. *RDF vocabularies*—to route BGPs to the sources that may be able to answer them. The signature of a BGP *or* RDF graph S is the set $S^2 \setminus (V \cup B)$ where S^2 denotes the projection of S onto its second coordinates.[3] Signatures are assumed to be known ahead of the federation process hence available to be used to constrain the generation of query partitions whenever possible.

A natural way to use them is to take advantage of *exclusive groups* in the global query. An exclusive group is a subquery which is such that its signature is included in the signature of exactly one source. The importance of these subqueries for heuristic purposes was, to the present authors' knowledge at least, first noted by [15], who observed that these patterns may be used to simplify the query decomposition process. The idea is that, since exclusive groups can only be answered by a single source, it is not necessary to subdivide them. Such a BGP can be shipped as is to be evaluated as a whole in a single execution context.

With respect to the process of generating partitions, this is the same as to say that exclusive groups can be treated as atomic units: the partition generator will compute all ways of combining exclusive groups with singletons to form a partition of a set, but no more than that. In other words, exclusive groups reduce the number of objects that is to be combined.

In the case where there are non-trivial exclusive groups—i.e. exclusive groups of size larger than 1—the size of the set of partitions that keeps those groups undivided will be smaller than the set of *all* partitions by the number of exponents that corresponds to the sum of elements in the exclusive groups in question.

In general, if P is a set of n elements and P_i a subset of k elements, then the number of partitions of P which has P_i as an element is $B_{n-(k-1)}$. When P_i is a singleton then obviously $B_{n-(k-1)} = B_n$ and so there is no reduction.

Generalizing to a finite sequence $P_1, \ldots, P_m$ of subsets of P, we have that the set of partitions of P which has every P_i for $1 \leq i \leq m$ as an element is

$$B_{(n - \sum_{k=1}^{n}(|P_k|-1))}$$

This number may still happen to be equal to B_n, however empirical evidence suggests that non-trivial exclusive groups are frequent (cf. 8). When they exist, the combinatoric reduction is significant. For instance for a query with 10 triple patterns, if there are two exclusive groups each with three triple patterns, the number of partitions that is computed drops from 115975 to 52.

7.2 Ordering of triple patterns

The final aspect that we will explore is the ordering of the triple pattern sequence that the RGS-algorithm ultimately utilizes. That is, the effectiveness wrt. branches pruned and number of checks performed, is very much affected by ordering of elements in the RGS input sequence. In essence, we want to prune branches as far up in the tree as possible.

Since the aim is to prune the tree as high up as possible, this indicates that "failing early"-approach is warranted. Hence, we seek a linearization of the query as a sequence, where adjacent triple patterns in the sequence are unlikely to produce b-connected graphs.

We define a simple heuristic based on the assumption that stars are more likely to contain blank nodes, hence placing two triple patterns belonging to the same star next to each other in the sequence is less likely to produce effective pruning.

The heuristic is based on the concept of *characteristic sets* as described in [6] to identify stars based on outgoing edges. We utilize characteristic sets to group triple patterns, for so to sequentially pick one of each set to produce the sequence. The characteristic set for a node s in BGP P is given by:

$$S_{c,t}(s, P) =_{def} \{(s, p, o) : \exists_{p,o}.(s, p, o) \in P\}$$

and the collection of characteristic sets in P is given by

$$S_{c,t}(P) =_{def} \{S_{c,t}(s, P) : \exists_{s,p,o}.(s, p, o) \in P\}$$

That is, we pick, in order, one triple pattern per characteristic set, so as to increase the likelihood that adjacent patterns in the sequence do not share blank nodes. Thus, the pruning starts out with a pair of triple patterns that do not occur in the same star, hence less likely to share a blank node. Now, the inclusion of exclusive groups makes it a bit more complicated. Triple patterns belonging to the same exclusive group are more likely to share a blank node than those that do not. However, unlike characteristic sets, the triple patterns that constitute exclusive groups do not necessarily share

[3]Using signatures for routing queries is a common strategem in the literature (cf. [5] and [15]) for which we claim no originality.

join variables. Hence, we seek a triple pattern sequence that factors in these two features at the same time, in a way that further reduces the likelihood that two adjacent triple patterns share a blank node. Algo. 4 produces the ordering that we seek, hereby referred to as $\mathbb{E}/C_s$-ordering.

Algorithm 4 Generate $\mathbb{E}/C_s$-ordered sequence

```
Require: Functions:
    GET-EXCLUSIVES($P$): returns a set of exclusive groups
    COUNT($S$): returns the size of $S$.
    SORT$_\leq$($f, S$): sort elements of $S$ in inc. order according to $f[S]$.
    INTERLEAVE(S): Given a sequence of collections (possibly different sizes), sequentially pick the head of each, repeat for the tail, until all are exhausted. If collection is empty, skip to next.
Require: Input:
    $P$: BGP, global query
function E-C-SEQ($P$)
    $\mathbb{E} \leftarrow$ GET-EXCLUSIVES($P$)
    $current \leftarrow \emptyset$
    $new \leftarrow S_{c,t}(P)$
    while $current != new$ do                     ▷ Merge CS wrt. excl. groups
        $current \leftarrow new$
        if $S_i, S_j \in current, S_i \neq S_j$, AND
         $E \in \mathbb{E}$ s.t. $E \cap S_i \neq \emptyset \neq E \cap S_j$ then
            $new \leftarrow (current \setminus \{S_i, S_j\}) \cup \{S_i \cup S_j\}$
        end if
    end while
    $S_P^e = \emptyset$
    for $S_i \in new$ do                          ▷ Substitutes in exclusives
        $S_i^e \leftarrow \{S' \subseteq S_i : S' \in \mathbb{E}\}$          ▷ exclusive groups in $S_i$
        $S_i^n \leftarrow \{\{tp\} : tp \in S_i \setminus (\bigcup S_i^e)\}$     ▷ unit clauses of rest
        $S_P^e = S_P^e \cup \{S_i^e \cup S_i^n\}$
    end for
    return INTERLEAVE(SORT$_\leq$(COUNT, $S_P^e$))
end function
```

Lines 5-11 define a fixpoint that produces collections of triple patterns that satisfy our criteria. Lines 12-17 are responsible for transforming triple pattern into BGPs and substituting in exclusive groups. That is, all elements are turned into singleton sets of triple patterns, except for exclusive groups that may be non-singleton sets. Line 18 is responsible for sequentially picking from the collections, so as to form a sequence. What the algorithm finally returns is a sequence of BGPs which basically acts as a position-wise map from RGS strings to atomic BGPs.

8 EVALUATION

8.1 Experimental setup

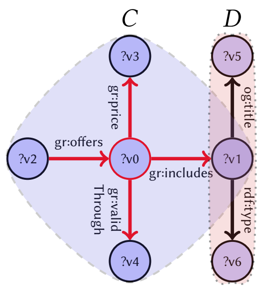
The queries that are of interest for study are those that are of a moderate size and complexity, with structures that are commonly seen in practice. From the data side, the experiment requires that a) blank nodes occur in the data, and b) that the data is partitioned over more than one source and in a variety of configurations involving both horizontal and vertical slicing.

Alas, to the authors' knowledge, no benchmark suite involving blank nodes and (partially) overlapping data sets is currently available, hence there was a need to create a set of tests specifically for this paper. However, instead of starting completely from scratch, we decided to reuse the well-known Waterloo SPARQL Diversity Test Suite (WatDiv) [2] as a basis, using both query patterns and the dataset that the benchmark provides to generate distributed test cases.

8.1.1 Query selection. We selected a handful of query templates from the WatDiv Basic Testing[4] use-case. More specifically, we chose queries that were either snowflake-shaped or complex-shaped (WatDiv categories), which ensured that star-shapes occurred as sub-patterns of the overall query. More concretely, the queries selected were WatDiv basic test queries $F1, F3, F5, C1$ and $C2$.

The importance that the queries contain star-shaped patterns is due to the nature of how blank nodes usually are manifested. As noted in [7], blank nodes are not only prevalent in the wild, but they tend to occur as the subject of more than just a single triple. More specifically, the edges that connect to blank nodes are more likely to fan-out rather than in, taking on a star-like structure with blank nodes in the center.

8.1.2 Data slicing templates. Based on the aforementioned observed frequency of blank nodes in star-shaped patterns, we define some abstract templates that aim to capture typical data distributions. That is, a template describes a data distribution, modulo a query pattern. More specifically, we define slicing templates $T1-T3$, visualized in Fig. 12 for the case of WatDiv query $F5$. In the figure, C and D denote the subgraphs that are combined in various configurations. It is worth noting that C has a special form: it contains only the dominant star[5] and any incoming edges to the center node (colored in red), the node subject for blank node substitution. In the cases where a blank node is substituted into the center node of the pattern C, we denote this with a superscript $\mathbb{B}$. Template figures for the remaining queries can be found in the appendix.



$$T1 = \{C^{\mathbb{B}} \cup D, C \cup D\}$$
$$T2 = \{C^{\mathbb{B}}, D\}$$
$$T3 = \{C^{\mathbb{B}}, C^{\mathbb{B}} \cup D\}$$

Figure 12: Templates for F5

The description and motivations behind the slicing templates are as follows:

T1: system wide vocabulary The query pattern is duplicated twice, i.e. total vocabulary overlap of slices, with a blank node inserted into the center of the dominant star in one of the

[4]http://dsg.uwaterloo.ca/watdiv/basic-testing.shtml
[5]The largest star-pattern in the query.

slices. The other slice remains unmodified wrt. how it occurs in the original dataset. Intuitively, this template is meant to capture the case where a vocabulary is reused as a complete data specification for a domain. That is, the template captures a distribution akin to that where we have different implementations of the same specification. An example of such a setting is the case where sensors and IoT-devices serialize data using a shared vocabulary (such as the Semantic Sensor Network vocabulary) but instantiates data differently where some use blank nodes.

T2: linked topical islands Pattern C, containing the dominant star, is used to create the first dataset slice. Futhermore, a blank node is inserted in the center. The second slice is generated from the remaining triple patterns of the query, i.e. D. Hence the original pattern is partitioned along the border of the dominant star. The template is meant to capture the case typically found in the LOD-world, where each source typically constitutes a linked topical island using its own vocabulary, but where there are outgoing links to elements in other data islands. A concrete example is the links between DBPedia, LinkedGeoData and Freebase.

T3: standard vocabularies As T2, but the second slice gets the complete original query pattern and not just the remaining difference. Futhermore, a blank node is inserted in the center of C in both slices. This template is meant to capture the case where a vocabulary has high reuse, but is seldom used only on its own. That is, often used with other vocabularies when representing data in individual sources. Concrete examples of such high reuse and widely deployed vocabularies are FOAF and the RDF Data Cube vocabulary.

It is worth noting that each template only generates two slices, i.e. two sources. We do this as it is not the increase in number of sources itself that has an effect on the number of distributions generated for a global query. Rather, it is whether or not individual triple patterns are exclusive to a single source that is the key player.

Now, for each query and for each abstract template, we derive a set of SPARQL CONSTRUCT queries that instantiates the distribution of the data in the query as intended by the template. More specifically, fragments of the selected query patterns were used to generate SPARQL CONSTRUCT queries that both sliced and/or replicated data, as well as inserting blank nodes in predetermined positions. These were then used to generate data slices from the original WatDiv dataset (10M Triples dataset from [2]), for so to be exposed as separate SPARQL endpoints.

8.2 Metrics

The main metric of interest is, of course, the relative reduction of the number of partitions that need to be evaluated, indicating the effectiveness of the pruning heuristics. Furthermore, we wish to determine the relative efficiency of the RGS-algorithm in terms of effectively utilizing the heuristics, as well as the costs associated with on-the-fly information gathering. That is, how many distinct probes are evaluated , how many of these probes identify redundancies (indicating primary pruning), and how many redundancies are identified based on subset lookup in the table of previously identified redundancies (denoted $\subseteq$-hits, indicating secondary pruning). We also measure the effect of the outlined ordering approach and exclusive groups. The effectiveness of the outlined ordering algorithm is judged by comparing the measured metrics with averaged sample runs of 100 random orderings. The effectiveness of treating exclusive groups as atomic entities is indicated by the relative reduction in the Bell number.

8.3 Results

The results of the experimental run are shown in Table 1. Table heading $|\mathbb{E}(P)|$ denotes the number of exclusive groups found in pattern P, while $|P \setminus \bigcup \mathbb{E}(P)|$ denotes the number of non-exclusive triple patterns. Furthermore, $\text{Bell}_{\mathbb{E}}$ indicates the Bell number of the problem where exclusive groups are treated as atomic entities. The column header Δ-checks / hits indicate the number of distinct block probes that needed to be performed and how many of them successfully identified redundant blocks for primary pruning, while $\subseteq$-hits indicates how many blocks were identified as redundant by the secondary check in sibling branches. Finally, the Partitions column indicates how many distributions the algorithm returns.

Immediately, we notice that treating exclusive groups as atomic entities can greatly reduce the problem, as indicated by comparing the Bell numbers. Hence the pre-processing step is clearly justified.

Moving on to analyzing the effectiveness of the query decomposition algorithm itself, we can clearly see that it is effective in reducing partitions by comparing the $\text{Bell}_{\mathbb{E}}$-number with the number of output distributions produced.

Diving into details, we focus on the internal measurements produced when running the pruning algorithm, using the previously outlined ordering. These results are found under the $\mathbb{E}/C_s$-ordering table heading. We first look at the number of Δ-checks performed; a high number of Δ-checks, relative to the size of the powerset of the input problem, will tell us that many blocks were probed, indicating a generally low pruning effect, while a low number indicating the opposite. For the collected data, we get that in the worst case it requires that 1/2 of the subsets needed to be probed, while in the best case, only about 1/20 need to be probed. We interpret this as indicating a high to medium pruning effectiveness.

Moving on, we compare the relative values of Δ-hits versus $\subseteq$-hits. More specifically, a low Δ-hit count coupled with a low $\subseteq$-hit count would indicate effective primary pruning in the higher levels of the computation tree, hence the ideal sought. A high count in Δ and low in $\subseteq$ indicates a dominating primary pruning, but in relative low level nodes, while the opposite order would indicate a dominating secondary pruning.

We first explain the odd ones out in terms of $\{F3, F5, C1, C2\} : T2$. Here, the pre-processing stage reduces the problem to that of calculating the combination of two exclusive groups. That is, in all cases, $n = 2$, there are no non-exclusive triple patterns, and the exclusive groups do not have overlapping sources. Hence, the probing stage essentially reduces to a source overlap check discounting the union of the two exclusive groups, without incurring an ASK probe.

For $F1 : T1$ and $F1 : T3$, we see that the $\subseteq$-hits dominate that of Δ-hits. It does so as well in $C2 : T1$, but due to the relatively high cardinality of the powerset for the case, the ratios are 1/20 and 2/20 thus not significantly different. For the remaining cases, the Δ- and $\subseteq$-hit counts are about the same, and both relatively low compared

Table 1 WatDiv evaluation results

							$\mathbb{E}/C_s$-ordered		Avg, 100 randomized orders		
Query	n	Bell	Templ.	$\lvert\mathbb{E}(P)\rvert$	$\lvert P \setminus \bigcup \mathbb{E}(P)\rvert$	$\text{Bell}_{\mathbb{E}}$	Δ-checks / |hits|	⊆-hits	Δ-checks / hits	⊆-hits	Partitions
F1	6	203	T1	0	6	203	31/5	19	33/7	41	52
			T2	2	2	15	7/3	1	8/4	2	5
			T3	1	5	203	31/5	19	33/7	38	52
F3	6	203	T1	0	6	203	20/9	10	23/12	30	15
			T2	2	0	2	1/1	-	1/1	-	1
			T3	1	4	52	16/5	5	16/5	10	15
F5	6	203	T1	0	6	203	21/10	11	23/12	29	15
			T2	2	0	2	1/1	-	1/1	-	1
			T3	1	4	52	15/4	5	16/5	11	15
C1	8	4140	T1	0	8	4140	33/22	20	38/27	81	15
			T2	2	0	2	1/1	-	1/1	-	1
			T3	1	4	52	15/4	5	16/5	12	15
C2	10	115975	T1	0	10	115975	54/50	95	54/50	95	5
			T2	2	0	2	1/1	-	1/1	-	1
			T3	1	3	15	7/3	1	8/4	2	5

to the powerset count, hence indicating that the combined pruning approach is effective in these cases.

Finally, we compare the statistics for the $\mathbb{E}/C_s$-ordering, as described so far, with that of the statistics generated by averaging 100 randomized orders of the original problem sequence. We see that in all cases, the $\mathbb{E}/C_s$-ordering never performs worse than the average, both in terms of primary and secondary pruning efficiency, and in most cases is significantly more efficient in terms of skewing the $\Delta/\subseteq$ ratio towards Δ rather than $\subseteq$, hence towards the preferred, less computationally expensive, primary pruning. This indicates that the ordering, in general, is more likely to result in a more efficient run of the RGS-algorithm.

In summary, we interpret the results to indicate that the overall algorithm and optimizations are feasible in practice.

9 RELATED WORK

Distributed query processing involves analyzing a query to identify a set of relevant RDF graphs, with or without the aid of statistics, to which subqueries can be assigned. Examples of this approach include [1, 4, 9, 10] and [15].

However, little emphasis has been dedicated to cater for the occurrence of blank nodes when it comes to the decomposition of the queries and ensuring soundness and completeness. In fact, apart from the authors previous work in [17], the research found in literature are all based on the assumption that blank nodes do not occur in the sources, hence the simple query-decomposition strategy partitioning into unit clauses, as implemented in DARQ [10] is sufficient. FedX [15], improves this by creating exclusive groups, but there is still only one partitioning of the original query at play, and does not address the blank node issue.

Following this path of assumptions, most current work regarding distributed query answering (without blank nodes), such as [14] and [8], focus on the process of source selection past that of simple triple-pattern wise source selection in order to reduce the number of sources queried. This addresses an orthogonal aspect of distributed query answering than what this paper considers, and more importantly, does not address the problem of blank nodes. Furthermore, both rely on the assumption that one can identify authoritative sources, a restriction that does not universally fit for all cases.

Another line of inquiry that tries to address the issue of distributed query processing by focusing on reducing client-server load, is that of *Triple Pattern Fragments (TPFs)*, as outlined in [19] and [20]. However, this assumes blank-node free sources, and assumes that existing data is converted into fragments beforehand, hence not a general solution.

Finally, it should be noted that the relationship between the present theory and the SPARQL 1.1 federation extension, is that the two are essentially orthogonal to each other: the latter, by providing the SERVICE keyword for source selection, provides a way of evaluating different parts of a query against different endpoints, but it doesn't tell you which parts. The present paper, in contrast, is concerned precisely with the question of which parts, that is, finding the decompositions.

10 CONCLUSION

Summing up the main tenets of the present paper, we have argued that zero-knowledge SPARQL processing is mathematically speaking just the problem of computing partitions of a query. From the perspective of computer science, however, this will not do as such, since the numbers of partitions of a set grows exponentially with the size of the set. A generation algorithm that is amenable to heuristic control is therefore needed. Existing algorithms for computing restricted growth strings seem to fit the bill as they allow semantic information obtained from probing the source for occurrences of blank nodes to be interleaved with the partition generation process in such a way that unfruitful branches can be detected and aborted.

We have implemented this idea in running code and subjected it to empirical testing. The results suggest that this may be a line of research worth pursuing. In many cases queries that with no heuristics applied are way out of reach of efficient computation fall well within it after the heuristic countermeasures are deployed.

More work needs to be done, though, and we note the following: the efficiency of the trimming of the RGS computation tree is quite dependent on the ordering of the subsets within the encompassing family (concretely a vector) of sets. This is an interesting topic for further study, perhaps with dynamic reordering.

Another area that holds some promise, is to explore other ways of probing the data sources, and other ways of preprocessing the combinatorial problem.

From the algorithmic side, it would be interesting to compare different restricted growth algorithms in terms of their effectiveness for the proposed kind of interleaved pruning. For lack of space, we leave this one and other possibilities for future work.

REFERENCES

[1] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. 2011. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints.. In *Proceedings of the 10th International Semantic Web Conference (ISWC 2011)*.

[2] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management systems. In *Proceedings of the 13th International Semantic Web Conference*. Springer, 197–212.

[3] Marcelo Arenas, Claudio Gutierrez, and Jorge Perez. 2009. Foundations of RDF Databases. In *Reasoning Web: Semantic Technologies for Information Systems*. LNCS, Vol. 5689. Springer, 158–204.

[4] Cosmin Basca and Abraham Bernstein. 2010. Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In *The 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2010)*. 64–79.

[5] Olaf Görlitz and Steffen Staab. 2011. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data (COLD 2011)*.

[6] Andrey Gubichev and Thomas Neumann. 2014. Exploiting the query structure for efficient join ordering in SPARQL queries.. In *EDBT*, Vol. 14. 439–450.

[7] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. 2014. Everything you always wanted to know about blank nodes. *Web Semantics: Science, Services and Agents on the World Wide Web* 27 (2014), 42–69.

[8] Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. 2017. Decomposing federated queries in presence of replicated fragments. *Web Semantics: Science, Services and Agents on the World Wide Web* 42 (2017), 1–18.

[9] Gabriela Montoya, Maria-Esther Vidal, and Maribel Acosta. 2012. A heuristic-based approach for planning federated sparql queries. In *Proceedings of the Third International Conference on Consuming Linked Data*. CEUR-WS.org, 63–74.

[10] Bastian Quilitz and Ulf Leser. 2008. Querying distributed RDF data sources with SPARQL. In *European Semantic Web Conference*. Springer, 524–538.

[11] Frank Ruskey. 2003. Combinatorial generation. *Preliminary working draft of book. University of Victoria, Victoria, BC, Canada* (2003).

[12] Frank Ruskey and Carla Savage. 1994. Gray codes for set partitions and restricted growth tails. *Australasian Journal of Combinatorics* 10 (1994), 85–96.

[13] Ahmad Sabri and Vincent Vajnovszki. 2014. Two Reflected Gray Code-Based Orders on Some Restricted Growth Sequences. *Comput. J.* 58, 5 (2014), 1099–1111.

[14] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. 2014. Hibiscus: Hypergraph-based source selection for sparql endpoint federation. In *European Semantic Web Conference*. Springer, 176–191.

[15] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. FedX: optimization techniques for federated query processing on linked data. In *Proceedings of the 10th international conference on The semantic web - Volume Part I (ISWC'11)*. Springer-Verlag, Berlin, Heidelberg, 601–616.

[16] Audun Stolpe. 2015. A logical characterisation of SPARQL federation. *Semantic Web* 6, 6 (2015), 565–584.

[17] Audun Stolpe and Jonas Halvorsen. 2017. Distributed query processing in the presence of blank nodes. *Semantic Web* 8, 6 (2017), 1001–1021.

[18] Astrid van Aggelen, Laura Hollink, Max Kemman, Martijn Kleppe, and Henri Beunders. 2017. The debates of the European parliament as linked open data. *Semantic Web* 8, 2 (2017), 271–281.

[19] Ruben Verborgh, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. 2014. Web-scale querying through linked data fragments. In *7th Workshop on Linked Data on the Web*.

[20] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Web Semantics: Science, Services and Agents on the World Wide Web* 37 (2016), 184–206.

APPENDIX A - QUERIES

Figure 13: Query F1

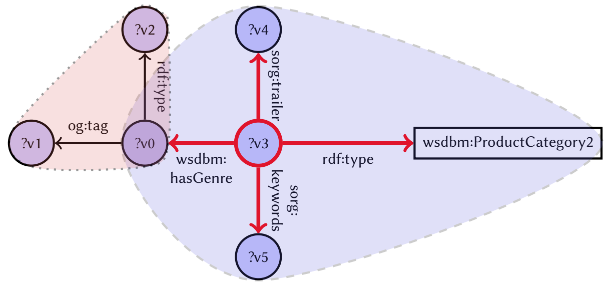


Figure 14: Query F3

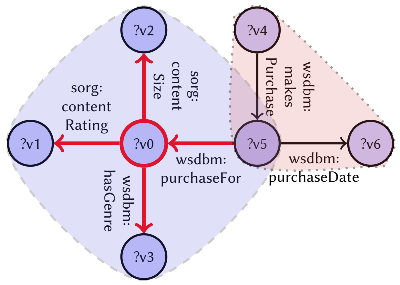


Figure 15: Query C1

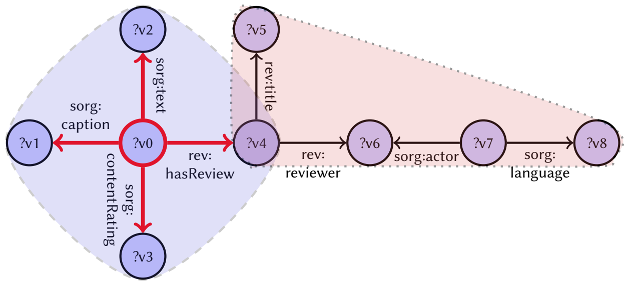
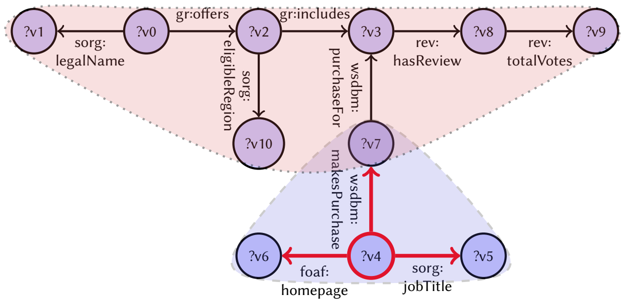


Figure 16: Query C2


?v1
?v0
gr:offers
?v2
gr:includes
?v3
?v8
?v9
sorg:
legalName
rev:
hasReview
rev:
totalVotes
sorg:
eligibleRegion
wsdbm:
purchaseFor
?v10
?v7
wsdbm:
makesPurchase
?v6
?v4
?v5
foaf:
homepage
sorg:
jobTitle