Semantic Web 13 (2022) 829–893
DOI 10.3233/SW-212871
IOS Press

Semantics and canonicalisation of SPARQL 1.1

Jaime Salas [*] and Aidan Hogan
DCC, Universidad de Chile; IMFD, Chile
E-mails: jaime.os.salas@gmail.com, ahogan@dcc.uchile.cl

Editor: Guilin Qi, Southeast University, China
Solicited reviews: Meng Wang, Southeast University, China; Guohui Xiao, Free University of Bozen-Bolzano, Italy; three anonymous reviewers

Abstract. We define a procedure for canonicalising SPARQL 1.1 queries. Specifically, given two input queries that return the same solutions modulo variable names over any RDF graph (which we call *congruent queries*), the canonicalisation procedure aims to rewrite both input queries to a syntactically canonical query that likewise returns the same results modulo variable renaming. The use-cases for such canonicalisation include caching, optimisation, redundancy elimination, question answering, and more besides. To begin, we formally define the semantics of the SPARQL 1.1 language, including features often overlooked in the literature. We then propose a canonicalisation procedure based on mapping a SPARQL query to an RDF graph, applying algebraic rewritings, removing redundancy, and then using canonical labelling techniques to produce a canonical form. Unfortunately a full canonicalisation procedure for SPARQL 1.1 queries would be undecidable. We rather propose a procedure that we prove to be sound and complete for a decidable fragment of monotone queries under both set and bag semantics, and that is sound but incomplete in the case of the full SPARQL 1.1 query language. Although the worst case of the procedure is super-exponential, our experiments show that it is efficient for real-world queries, and that such difficult cases are rare.

Keywords: SPARQL, RDF, query, semantics, caching, canonicalisation, congruence, equivalence

1. Introduction

The Semantic Web provides a variety of standards and techniques for enhancing the machine-readability of Web content in order to increase the levels of automation possible for day-to-day tasks. RDF [54] is the standard framework for the graph-based representation of data on the Semantic Web. In turn, SPARQL [24] is the standard querying language for RDF, composed of basic graph patterns extended with expressive features that include path expressions, relational algebra, aggregation, federation, among others.

The adoption of RDF as a data model and SPARQL as a query language has grown significantly in recent years [4,26]. Prominent datasets such as DBpedia [35] and Wikidata [61] contain in the order of hundreds of millions or even billions of RDF triples, and their associated SPARQL endpoints receive millions of queries per day [37,52]. Hundreds of other SPARQL endpoints are likewise available on the Web [4]. However, a survey carried out by Buil-Aranda et al. [4] found that a large number of SPARQL endpoints experience performance issues such as latency and unavailability. The same study identified the complexity of SPARQL queries as one of the main causes

[*]Corresponding author. E-mail: jaime.os.salas@gmail.com.

of these problems, which is perhaps an expected result given the expressivity of the SPARQL query language where, for example, the decision problem consisting of determining if a solution is given by a query over a graph is PSPACE-hard for the SPARQL language [44].

One way to address performance issues is through caching of sub-queries [41,62]. The caching of queries is done by evaluating a query, then storing its result set, which can then be used to answer future instances of the same query without using any additional resources. The caching of sub-queries identifies common query patterns whose results can be returned for queries that contain said query patterns. However, this is complicated by the fact that a given query can be expressed in different, semantically equivalent ways. As a result, if we are unable to verify if a given query is equivalent to one that has already been cached, we are not using the cached results optimally: we may miss relevant results.

Ideally, for the purposes of caching, we could use a procedure to *canonicalise* SPARQL queries. To formalise this idea better, we call two queries *equivalent* if (and only if) they return the same solutions over any RDF dataset. Note however that this notion of equivalence requires the variables of the solutions of both queries to coincide. In practice, variable names will often differ across queries, where we would still like to be able to cache and retrieve the results for queries whose results are the same modulo variable names. Hence we call two queries *congruent* if they return the same solutions, modulo variable names, over any RDF dataset; in other words, two queries are congruent if (and only if) there exists a one-to-one mapping from the variables in one query to the variables of the other query that makes the former equivalent to the latter.

In this paper, we propose a procedure by which congruent SPARQL queries can be "canonicalised". We call such a procedure *sound* if the output query is congruent to the input query, and *complete* if the same output query is given for any two congruent input queries.

Example 1.1. Consider the following two SPARQL queries asking for names of aunts:

```
SELECT DISTINCT ?z WHERE {
  { ?w :mother ?x . } UNION { ?w :father ?x. }
  ?x :sister ?y . ?y :name ?z .
}
```

```
SELECT DISTINCT ?z WHERE {
  { ?a :name ?z . ?c :mother ?p . ?p :sister ?a . }
  UNION
  { ?a :name ?z . ?c :father ?p . ?p :sister ?a . }
}
```

Both queries are equivalent: they always return the same results for any RDF dataset. Now rather consider a third SPARQL query:

```
SELECT DISTINCT ?n WHERE {
  ?a :name ?n . ?b :name ?n .
  { ?v1 :mother ?v2 . ?v2 :sister ?a . } UNION
  { ?v3 :father ?v4 . ?v4 :sister ?a . }
}
```

Note that the pattern `?b :name ?n .` in this query is redundant. This query is not equivalent to the former two because the variable that is returned is different, and thus the solutions (which contain the projected variable), will not be the same. However all three queries are congruent; for example, if we rewrite `?n` to `?z` in the third query, all three queries become equivalent.

Canonicalisation aims to rewrite all three (original) queries to a unique, congruent, output query.

The potential use-cases we foresee for a canonicalisation procedure include the following:

Query caching: As aforementioned, a canonicalisation procedure can improve caching for SPARQL endpoints. By capturing knowledge about query congruence, canonicalisation can increase the cache hit rate. Similar techniques could also be used to identify and cache frequently appearing (congruent) sub-queries [41].

Views: In a conceptually similar use case to caching, our canonical procedure can be used to describe views [9]. In particular, the canonicalisation procedure can be used to create a key that uniquely identifies each of the views available.

Log analysis: SPARQL endpoint logs can be analysed in order to understand the importance of different SPARQL features [7,52], to build suitable benchmarks [52], to understand how users build queries incrementally [7,64], etc. Our canonicalisation procedure could be used to pre-process and group congruent queries in logs.

Query optimisation: Canonicalisation may help with query optimisation by reducing the variants to be considered for query planning, detecting duplicate sub-queries that can be evaluated once, removing redundant patterns (as may occur under query rewriting strategies for reasoning [33]), etc.

Learning over queries: Canonicalisation can reduce superficial variance in queries used to train machine learning models. For example, recent question answering systems learn translations from natural language questions to queries [10], where canonicalisation can be used to homogenise the syntax of queries used for training.

Other possible but more speculative use-cases involve signing or hashing SPARQL queries, discovering near-duplicate or parameterised queries (by considering constants as variables), etc. Furthermore, with some adaptations, the methods proposed here could be generalised to other query languages, such as to canonicalise SQL queries, Cypher queries [3], etc.

A key challenge for canonicalising SPARQL queries is the prohibitively high computational complexity that it entails. More specifically, the *query equivalence* problem takes two queries and returns true if and only if they return the same solutions for any dataset, or false otherwise. In the case of SPARQL, this problem is intractable (NP-complete) even when simply permitting joins (with equality conditions). Even worse, the problem becomes undecidable when features such as projection and optional matches are added [45]. Since a canonicalisation procedure can be directly used to decide equivalence, this means that canonicalisation is at least as hard as the equivalence problem in computational complexity terms, meaning it will likewise be intractable for even simple fragments and undecidable when considering the full SPARQL language. There are thus fundamental limitations in what can be achieved for canonicalising SPARQL queries.

With these limitations in mind, we propose a canonicalisation procedure that is always sound, but only complete for a monotone fragment of SPARQL under set or bag semantics. This monotone fragment permits unions and joins over basic graph patterns, some examples of which were illustrated in Example 1.1. We further provide sound, but incomplete, canonicalisation of the full SPARQL 1.1 query language, whereby the canonicalised query will be congruent to the input query, but not all pairs of congruent input queries will result in the same output query. In the case of incomplete canonicalisation, we are still able to find novel congruences, in particular through canonical labelling of variables, which further allows for ordering operands in a consistent manner. Reviewing the aforementioned use-cases, we believe that this "best-effort" form of canonicalisation is still useful, as in the case of caching, where missing an equivalence will require re-executing the query (which would have to be done in any case), or in the case of learning over queries, where incomplete canonicalisation can still increase the homogeneity of the training examples used.

As a high-level summary, our procedure combines four main techniques for canonicalisation.

1. The first technique is to convert SPARQL queries to an *algebraic graph*, which abstracts away syntactic variances, such as the ordering of operands for operators that are commutative, and the grouping of operands for operators that are associative.
2. The second technique is to apply *algebraic rewritings* on the graph to achieve normal forms over combinations of operators. For example, we rewrite monotone queries – that allow any combination of join, union, basic graphs patterns, etc. – into unions of basic graph patterns; this would rewrite the first and third queries shown in Example 1.1 into a form similar to the second query.
3. The third technique is to apply *redundancy elimination* within the algebraic graph, which typically involves the removal of elements of the query that do not affect the results; this technique would remove the redundant `?b :name ?n .` pattern from the third query of Example 1.1.
4. The fourth and final technique is to apply a *canonical labelling* of the algebraic graph, which will provide consistent labels to variables, and which in turn allows for the (unordered) algebraic graph to be serialised back into the (ordered) concrete syntax of SPARQL in a canonical way.

We remark that the techniques do not necessarily follow the presented order; in particular, the second and third techniques can be interleaved in order to provide further canonicalisation of queries.

This paper extends upon our previous work [50] where we initially outlined a sound and complete procedure for canonicalising monotone SPARQL queries. The novel contributions of this extended paper include:

– We close a gap involving translation of monotone queries under bag semantics that cannot return duplicates into set semantics.
– We provide a detailed semantics for SPARQL 1.1 queries; formalising and understanding this is a key prerequisite for canonicalisation.
– We extend our algebraic graph representation in order to be able to represent SPARQL 1.1 queries, offering partial canonicalisation support.
– We implement algebraic rewriting rules for specific SPARQL 1.1 operators, such as those relating to filters; we further propose techniques to canonicalise property path expressions.
– We provide more detailed experiments, which now include results over a Wikidata query log, a comparison with existing systems from the literature that perform pairwise equivalence checks, and more detailed stress testing.

We also provide extended proofs of results that were previously unpublished [51], as well as providing extended discussion and examples throughout.

The outline of the paper is then as follows. Section 2 provides preliminaries for RDF, while Section 3 provides a detailed semantics for SPARQL. Section 4 provides a problem statement, formalising the notion of canonicalisation. Section 5 discusses related works in the areas of systems that support query containment, equivalence, and congruence. Sections 6 and 7 discuss our SPARQL canonicalisation framework for monotone queries, and SPARQL 1.1, respectively. Section 8 presents evaluation results. Section 9 concludes.

2. RDF data model

We begin by introducing the core concepts of the RDF data model over which the SPARQL query language will later be defined. The following is a relatively standard treatment of RDF, as can be found in various papers from the literature [22,28]. We implicitly refer to RDF 1.1 unless otherwise stated.

2.1. Terms and triples

RDF assumes three pairwise disjoint sets of terms: *IRIs* ($\mathbf{I}$), *literals* ($\mathbf{L}$) and *blank nodes* ($\mathbf{B}$). Data in RDF are structured as *triples*, which are 3-tuples of the form $(s, p, o) \in \mathbf{IB} \times \mathbf{I} \times \mathbf{IBL}$ denoting the *subject* s, the *predicate* p, and the *object* o of the triple.[1] There are three types of literals: a *plain literal* s is a simple string, a *language-tagged literal* (s, l) is a pair of a simple string and a language tag, and (s, d) is a pair of a simple string and an IRI (denoting a datatype).

In this paper we use Turtle/SPARQL-like syntax, where `:a`, `xsd:string`, etc., denote IRIs; `_:b`, `_:x1`, etc., denote blank nodes; `"a"`, `"xy z"`, etc., denote plain literals; `"hello"@en`, `"hola"@es`, etc., denote language-tagged literals; and `"true"^^xsd:boolean`, `"1"^^xsd:int`, etc., denote datatype literals.

2.2. Graph

An RDF graph G is a set of RDF triples. It is called a graph because each triple $(s, p, o) \in G$ can be viewed as a directed labelled edge of the form $s \xrightarrow{p} o$, and a set of such triples forms a directed edge-labelled graph.

[1] In this paper we concatenate set names to denote their union; e.g., **IBL** is used as an abbreviation for the union $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$.

2.3. *Simple entailment and equivalence*

Blank nodes in RDF have special meaning; in particular, they are considered to be existential variables. The notion of *simple entailment* [22,25] captures the existential semantics of blank nodes (among other fundamental aspects of RDF). This same notion also plays a role in how the SPARQL query language is defined.

Formally, let $\alpha : \mathbf{B} \to \mathbf{IBL}$ denote a mapping that maps blank nodes to RDF terms; we call such a mapping a *blank node mapping*. Given an RDF graph G, let $\text{bnodes}(G)$ denote all of the blank nodes appearing in G. Let $\alpha(G)$ denote the image of G under α; i.e., the graph G but with each occurrence of each blank node $b \in \text{bnodes}(G)$ replaced with $\alpha(b)$. Given two RDF graphs G and H, we say that G *simple-entails* H, denoted $G \models H$, if and only if there exists a blank node mapping α such that $\alpha(H) \subseteq G$ [22,25]. Furthermore, if $G \models H$ and $H \models G$, then we say that they are *simple equivalent*, denoted $G \equiv H$.

Deciding simple entailment $G \models H$ is known to be NP-complete [22]. Deciding the simple equivalence $G \equiv H$ is likewise known to be NP-complete.

We remark that the RDF standard defines further entailment regimes that cover the semantics of datatypes and the special RDF and RDFS vocabularies [25]; we will not consider such entailment regimes here.

2.4. *Isomorphism*

Given that blank nodes are defined as existential variables [25], two RDF graphs differing only in blank node labels are thus considered *isomorphic* [19,28].

Formally, if a blank node mapping of the form $\alpha : \mathbf{B} \to \mathbf{B}$ is one-to-one, we call it a *blank node bijection*. Two RDF graphs G and H are defined as *isomorphic*, denoted $G \simeq H$, if and only if there exists a blank node bijection α such that $\alpha(G) = H$; i.e., the two RDF graphs differ only in their blank node labels. We remark that if $G \simeq H$, then $G \equiv H$; however, the inverse does not always hold as we discuss in the following.

Deciding the isomorphism $G \simeq H$ is known to be GI-complete [28] (as hard as graph isomorphism).

2.5. *Leanness and core*

Existential blank nodes may give rise to redundant triples. In particular, an RDF graph G is called *lean* if and only if there does not exist a proper subgraph $G' \subsetneq G$ of it such that $G' \models G$; otherwise G is called *non-lean*. Non-lean graphs can be seen, under the RDF semantics, as containing redundant triples. For example, given an RDF graph $G = \{(\texttt{:x}, \texttt{:y}, \texttt{:z}), (\texttt{:x}, \texttt{:y}, \texttt{\_:b})\}$, the second triple is seen as redundant: it states that `:x` has *some* value on `:y`, but we know this already from the first triple, so the second triple says nothing new.

The *core* of an RDF graph G is then an RDF graph G' such that $G' \equiv G$ and G' is lean; intuitively it is a version of G without redundancy. For example the core of the aforementioned graph would be $G' = \{(\texttt{:x}, \texttt{:y}, \texttt{:z})\}$; note that $G' \equiv G$ and G' is lean, but $G' \not\simeq G$. The core of a graph is unique modulo isomorphism [22]; hence we refer to *the* core of a graph.

Deciding whether or not an RDF G is lean is known to be coNP-complete [22]. Deciding if G' is the core of G is known to be DP-complete [22].

2.6. *Merge*

Blank nodes are considered to be scoped to a local RDF graph. Hence when combining RDF graphs, applying a *merge* (rather than union) avoids blank nodes with the same name in two (or more) graphs clashing. Given two RDF graphs G and G', and a blank node bijection α such that $\text{bnodes}(\alpha(G)) \cap \text{bnodes}(G') = \emptyset$, we call $\alpha(G) \cup G'$ an *RDF merge*, denoted $G \uplus G'$. The merge of two graphs is unique modulo isomorphism.

3. SPARQL 1.1 semantics

We now define SPARQL 1.1 in detail [24]. We will begin by defining a SPARQL dataset over which queries are evaluated. We then introduce an abstract syntax for SPARQL queries. Thereafter we discuss the evaluation of queries under different semantics.

Table 1

Studies that define the semantics of features in SPARQL (1.1), including **Mon**otone (basic graph patterns, joins, `UNION`, un-nested `SELECT DISTINCT`), **Filt**ers, **Opt**ionals, **Neg**ation (`OPTIONAL & !BOUND`, `MINUS`, `FILTER (NOT) EXISTS`), **N**amed **Gra**phs (`GRAPH`, `FROM (NAMED)`), **Path**s, **Fed**eration (`SERVICE`), **Ass**ig**n**ment (`BIND`, `VALUES`), **Agg**regation (`GROUP BY` and aggregate functions), **Sub Q**ueries (nested `SELECT`), **Sol**ution **M**odifiers (`LIMIT`, `OFFSET`, `ORDER BY`), Query **Form**s (`CONSTRUCT`, `ASK`, `DESCRIBE`), **Exp**ressions and Functions (e.g., `+`, `BOUND`, `COUNT`, `IF`), **Bag** Semantics; we denote by "*" partial definitions or discussion

Paper	Year	Mon	Filt	Opt	Neg	NGra	Path	Fed	Assn	Agg	SubQ	SolM	Form	Exp	Bag
Perez et al. [43,44]	2006	✓	✓	✓	*									*	
Polleres [46]	2007	✓	✓	✓	*	✓					*		*	*	
Alkhateeb et al. [2]	2009	✓	✓	✓	*		✓							*	
Arenas and Pérez [5]	2012	✓	✓	✓	*	*	✓	✓	*					*	✓
Polleres and Wallner [47]	2013	✓	✓	✓	✓	✓	✓		✓		✓	✓	*	*	✓
Kaminski et al. [32]	2017	✓	✓	✓	✓		✓		✓	✓	✓	*		*	✓
Salas and Hogan	2021	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	*	✓

Table 2

SPARQL property path syntax

The following are path expressions	
p	a predicate (IRI)
$!(p_1 \mid \ldots \mid p_k \mid \hat{}p_{k+1} \mid \ldots \mid \hat{}p_n)$	any (inv.) predicate not listed
and if e, e_1, e_2 are path expressions the following are also path expressions:	
$\hat{}e$	an inverse path
e_1 / e_2	a path of e_1 followed by e_2
$e_1 \mid e_2$	a path of e_1 or e_2
$e*$	a path of zero or more e
$e+$	a path of one or more e
$e?$	a path of zero or one e
(e)	brackets used for grouping

These definitions extend similar preliminaries found in the literature. However, our definitions of the semantics of SPARQL 1.1 extend beyond the core of the language and rather aim to be exhaustive, where a clear treatment of the full language is a prerequisite for formalising the canonicalisation of queries using the language. Table 1 provides a summary of prior works that have defined the semantics of SPARQL features. We exclude works that came after one of the works shown and use a subset of the features of that work (even if they may contribute novel results about those features). Some SPARQL 1.0 features, such as `UNION`, `FILTER` and `OPTIONAL`, have been featured in all studies. In terms of SPARQL 1.1, the most extensive formal definitions have been provided by Polleres and Wallner [47], and by Kaminski et al. [32]. However, both works omit query features: Polleres and Wallner [47] omit federation and aggregation, whereas Kaminski et al. [32] omit named graphs, federation, and non-`SELECT` query forms. Compared to these previous works, we aim to capture the full SPARQL 1.1 query language, with one simplification: we define functions and expressions abstractly, rather than defining all of the many built-ins that SPARQL 1.1 provides (e.g., `+`, `BOUND`, `COUNT`, `IF`, etc.)

3.1. Query syntax

Before we introduce an abstract syntax for SPARQL queries, we provide some preliminaries:

– A *triple pattern* (s, p, o) is a member of the set $\mathbf{VIBL} \times \mathbf{VI} \times \mathbf{VIBL}$ (i.e., an RDF triple allowing variables in any position and literals as subject).
– A *basic graph pattern* B is a set of triple patterns. We denote by vars $B := \bigcup_{(s,p,o)\in B} \mathbf{V} \cap \{s, p, o\}$ the set of variables used in B.
– A *path pattern* (s, e, o) is a member of the set $\mathbf{VIBL} \times \mathbf{P} \times \mathbf{VIBL}$, where $\mathbf{P}$ is the set of all path expressions defined by Table 2.

- A *navigational graph pattern* N is a set of paths patterns and triple patterns (with variable predicates). We denote by $\text{vars}\, N := \bigcup_{(s,e,o)\in N} \mathbf{V} \cap \{s, e, o\}$ the set of variables used in N.
- A term in **VIBL** is a *built-in expression*. Let $\bot$ denote an unbound value and ε an error. We call ϕ a *built-in function* if it takes a tuple of values from $\mathbf{IBL} \cup \{\bot, \varepsilon\}$ as input and returns a single value in $\mathbf{IBL} \cup \{\bot, \varepsilon\}$ as output. An expression $\phi(R_1, \dots, R_n)$, where each $R_1, \dots, R_n$ is a *built-in expression*, is itself a *built-in expression*.
- An *aggregation function* ψ is a function that takes a bag of tuples from **IBL** as input and returns a value in $\mathbf{IBL} \cup \{\bot, \varepsilon\}$ as output. An expression $\psi(R_1, \dots, R_n)$, where each $R_1, \dots, R_n$ is a *built-in expression*, is an *aggregation expression*.
- If R is a *built-in expression*, and Δ is a boolean value indicating ascending or descending order, then (R, Δ) is an *order comparator*.

We then define the abstract syntax of a SPARQL query as shown in Table 3. Note that we abbreviate `OPTIONAL` as OPT, `FILTER EXISTS` as FE, and `FILTER NOT EXISTS` as FNE. Otherwise mapping from SPARQL's concrete syntax to this abstract syntax is straightforward, with the following exceptions:

- For brevity, we consider the following SPARQL 1.1 operators to be represented as functions:
  - boolean operators: `!` for negation, `&&` for conjunction, `||` for disjunction;
  - equality and inequality operators: `=`, `<`, `>`, `<=`, `>=`, `!=`;
  - numeric operators: unary `+` and `-` for positive/negative numbers; binary `+` and `-` for addition/subtraction, `*` for multiplication and `/` for division;

  for example, replacing `?a+?b`, we assume addition to be defined as a function SUM(`?a`, `?b`).
- We combine `FROM` and `FROM NAMED` into one feature, FROM, so they can be evaluated together.
- A query such as `DESCRIBE <x> <y>` in the concrete syntax can be expressed in the abstract syntax with an empty pattern $\text{DESCRIBE}_{\{x,y\}}(\{\})$.
- Aggregates without grouping can be expressed with $\text{GROUP}_{\{\}}(Q)(D)$. We assume that every query in the abstract syntax with a group-by pattern uses AGG – possibly $\text{AGG}_{\{\}}(Q)$ – to generate a graph pattern (and "flatten" groups).
- Some aggregation functions in SPARQL take additional arguments, including a `DISTINCT` modifier, or a delimiter in the case of `CONCAT`. For simplicity, we assume that these are distinct functions, e.g., COUNT$(\cdot)$ versus COUNTDISTINCT$(\cdot)$.
- SPARQL allows `SELECT *` to indicate that values for all variables should be returned. Otherwise SPARQL requires that at least one variable be specified. A `SELECT *` clause can be written in the abstract syntax as $\text{SELECT}_V(Q)$ where Q is a graph pattern on V. Also the abstract syntax allows empty projections $\text{SELECT}_{\{\}}(Q)$, which greatly simplifies certain definitions and proofs; this can be represented in the concrete syntax as `SELECT ?empty`, where `?empty` is a fresh variable not appearing in the query.
- In the concrete syntax, `SELECT` allows for built-in expressions and aggregation expressions to be specified. We only allow variables to be used. However, such expressions can be bound to variables using BIND or AGG.[2]
- In the concrete syntax, `ORDER BY` allows for using aggregation expressions in the order comparators. Our abstract syntax does not allow this as it complicates the definitions of such comparators. Ordering on aggregation expressions can rather be achieved using sub-queries.
- We use $[Q_1 \text{SERVICE}_x^{\Delta} Q_2]$ to denote `SERVICE`, where $\Delta = \texttt{true}$ indicates the `SILENT` keyword is invoked, and $\Delta = \texttt{false}$ indicates that it is not.
- We do not consider `SERVICE` with variables as it has no normative semantics in the standard [48].

Aside from the latter point, these exceptions are syntactic conveniences that help simplify later definitions.

[2]In our proposed abstract syntax and the concrete syntax, the ordering of variables in the `SELECT` is not meaningful, though in practice engines may often present variables in the results following the same order in which they are listed by the `SELECT` clause.

Table 3

Abstract SPARQL syntax

– B is a basic graph pattern.	$\therefore$ B is a graph pattern on vars B.
– N is a navigational graph pattern.	$\therefore$ N is a graph pattern on vars N.
– Q_1 is a graph pattern on V_1. – Q_2 is a graph pattern on V_2.	$\therefore$ $[Q_1 \text{AND} Q_2]$ is a graph pattern on $V_1 \cup V_2$; $\therefore$ $[Q_1 \text{UNION} Q_2]$ is a graph pattern on $V_1 \cup V_2$; $\therefore$ $[Q_1 \text{OPT} Q_2]$ is a graph pattern on $V_1 \cup V_2$; $\therefore$ $[Q_1 \text{MINUS} Q_2]$ is a graph pattern on V_1.
– Q is a graph pattern on V. – Q_1 is a graph pattern on V_1. – Q_2 is a graph pattern on V_2. – v is a variable not in V. – R is a built-in expression.	$\therefore$ $\text{FILTER}_R(Q)$ is a graph pattern on V; $\therefore$ $[Q_1 \text{FE} Q_2]$ is a graph pattern on V_1; $\therefore$ $[Q_1 \text{FNE} Q_2]$ is a graph pattern on V_1; $\therefore$ $\text{BIND}_{R,v}(Q)$ is a graph pattern on $V \cup \{v\}$.
– Q is a graph pattern on V. – $\mathfrak{M}$ is a bag of solution mappings on $V_{\mathfrak{M}} = \bigcup_{\mu \in \mathfrak{M}} \text{dom}(\mu)$.	$\therefore$ $\text{VALUES}_{\mathfrak{M}}(Q)$ is a graph pattern on $V \cup V_{\mathfrak{M}}$.
– Q is a graph pattern on V. – x is an IRI. – v is a variable.	$\therefore$ $\text{GRAPH}_x(Q)$ is a graph pattern on V. $\therefore$ $\text{GRAPH}_v(Q)$ is a graph pattern on $V \cup \{v\}$.
– Q_1 is a graph pattern on V_1. – Q_2 is a graph pattern on V_2. – x is an IRI. – Δ is a boolean value.	$\therefore$ $[Q_1 \text{SERVICE}_x^{\Delta} Q_2]$ is a graph pattern on $V_1 \cup V_2$.
– Q is a graph pattern on V. – Q' is a group-by pattern on (V', V). – V'' is a set of variables. – A is an aggregation expression – Λ is a (possibly empty) set of pairs $\{(A_1, v_1), \ldots, (A_n, v_n)\}$, where $A_1, \ldots, A_n$ are aggregation expressions, $v_1, \ldots, v_n$ are variables not appearing in $V \cup V'$ such that $v_i \neq v_j$ for $1 \leqslant i < j \leqslant n$, and where vars $\Lambda = \{v_1, \ldots, v_n\}$	$\therefore$ Q is a group-by pattern on $(\emptyset, V)$. $\therefore$ Q' is a graph pattern on V'. $\therefore$ $\text{GROUP}_{V''}(Q)$ is a group-by pattern on (V'', V). $\therefore$ $\text{HAVING}_A(Q')$ is a group-by pattern on (V', V). $\therefore$ $\text{AGG}_{\Lambda}(Q')$ is a graph pattern on $(V' \cup \text{vars}\,\Lambda, V)$.
– Q is a graph pattern or sequence pattern on V. – Ω is a non-empty sequence of order comparators. – k is a non-zero natural number.	$\therefore$ Q is a graph pattern and sequence pattern on V $\therefore$ $\text{ORDER}_{\Omega}(Q)$ is a sequence pattern on V. $\therefore$ $\text{DISTINCT}(Q)$ is a sequence pattern on V. $\therefore$ $\text{REDUCED}(Q)$ is a sequence pattern on V. $\therefore$ $\text{OFFSET}_k(Q)$ is a sequence pattern on V. $\therefore$ $\text{LIMIT}_k(Q)$ is a sequence pattern on V.
– Q is a sequence pattern on V that does not contain the same blank node b in two different graph patterns. – V' is a set of variables. – B is a basic graph pattern. – X is a set of IRIs and/or variables.	$\therefore$ $\text{SELECT}_{V'}(Q)$ is a query and a graph pattern on V'. $\therefore$ $\text{ASK}(Q)$ is a query. $\therefore$ $\text{CONSTRUCT}_B(Q)$ is a query. $\therefore$ $\text{DESCRIBE}_X(Q)$ is a query.
– Q is a query but not a from query – X and X' are sets of IRIs	$\therefore$ $\text{FROM}_{X,X'}(Q)$ is a from query and a query.

3.2. Datasets

SPARQL allows for indexing and querying more than one RDF graph, which is enabled through the notion of a SPARQL dataset. Formally, a *SPARQL dataset* $D := (G, \{(x_1, G_1), \ldots, (x_n, G_n)\})$ is a pair of an RDF graph G called the *default graph*, and a set of *named graphs* of the form (x_i, G_i), where x_i is an IRI (called a *graph name*) and G_i is an RDF graph; additionally, graph names must be unique, i.e., $x_j \neq x_k$ for $1 \leqslant j < k \leqslant n$. We denote by G_D the default graph G of D and by $D^* = \{(x_1, G_1), \ldots, (x_n, G_n)\}$ the set of all named graphs in D. We further denote by $G_{D[x_i]}$ the graph G_i such that $(x_i, G_i) \in D^*$ or the empty graph if x_i does not appear as a graph name in D^*.

3.3. Services

While the SPARQL standard defines a wide range of features that compliant services must implement, a number of decisions are left to a particular service. First and foremost, a service chooses what dataset to index. Along these lines, we define a SPARQL service as a tuple $\mathcal{S} = (D, \mathcal{R}, \mathcal{A}, \leqslant, \text{describe})$, where:

– D is a dataset;
– $\mathcal{R}$ is a set of supported built-in expressions;
– $\mathcal{A}$ is a set of supported aggregation expressions;
– $\leqslant$ is a total ordering of RDF terms and $\bot$;
– describe is a function used to describe RDF terms.

We will denote by $D_{\mathcal{S}}$ the dataset of a particular service. The latter two elements will be described in more detail as they are used. The SPARQL standard does define some minimal requirements on the set of built-in expressions, the set of aggregation expressions, the ordering of terms, etc., that a standards-compliant service should respect. We refer to the SPARQL standard for details on these requirements [24].

3.4. Query evaluation

The semantics of a SPARQL query Q can be defined in terms of its evaluation over a SPARQL dataset D, denoted $Q(D)$ which returns *solution mappings* that represent "matches" for Q in D.

3.4.1. Solution mappings

A *solution mapping* μ is a partial mapping from variables in $\mathbf{V}$ to terms $\mathbf{IBL}$. We denote the set of all solution mappings by $\mathbf{M}$. Let $\text{dom}(\mu)$ denote the domain of μ, i.e., the set of variables for which μ is defined. Given $\{v_1, \dots, v_n\} \subseteq \mathbf{V}$ and $\{x_1 \dots, x_n\} \subseteq \mathbf{IBL} \cup \{\bot\}$, we denote by $\{v_1/x_1, \dots, v_n/x_n\}$ the mapping μ such that $\text{dom}(\mu) = \{v_i \mid x_i \notin \{\bot, \varepsilon\}\}$ for $1 \leqslant i \leqslant n$ and $\mu(v_i) = x_i$ for $v_i \in \text{dom}(\mu)$. We denote by $\mu_\emptyset$ the empty solution mapping (such that $\text{dom}(\mu_\emptyset) = \emptyset$).

We say that two solution mappings μ_1 and μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, if and only if $\mu_1(v) = \mu_2(v)$ for every $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$. We say that two solution mappings μ_1 and μ_2 are *overlapping*, denoted $\mu_1 * \mu_2$, if and only if $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) \neq \emptyset$.

Given two compatible solution mappings $\mu_1 \sim \mu_2$, we denote by $\mu_1 \cup \mu_2$ their combination such that $\text{dom}(\mu_1 \cup \mu_2) = \text{dom}(\mu_1) \cup \text{dom}(\mu_2)$, and if $v \in \text{dom}(\mu_1)$ then $(\mu_1 \cup \mu_2)(v) = \mu_1(v)$, otherwise if $v \in \text{dom}(\mu_2)$ then $(\mu_1 \cup \mu_2)(v) = \mu_2(v)$. Since the solution mappings μ_1 and μ_2 are compatible, for all $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it holds that $(\mu_1 \cup \mu_2)(v) = \mu_1(v) = \mu_2(v)$, and thus $\mu_1 \cup \mu_2 = \mu_2 \cup \mu_1$.

Given a solution mapping μ and a triple pattern t, we denote by $\mu(t)$ the image of t under μ, i.e., the result of replacing every occurrence of a variable v in t by $\mu(v)$ (generating an unbound $\bot$ if $v \notin \text{dom}(\mu)$). Given a basic graph pattern B, we denote by $\mu(B)$ the image of B under μ, i.e., $\mu(B) := \{\mu(t) \mid t \in B\}$. Likewise, given a navigational graph pattern N, we analogously denote by $\mu(N)$ the image of N under μ.

Blank nodes in SPARQL queries can likewise play a similar role to variables though they cannot form part of the solution mappings. Given a blank node mapping α, we denote by $\alpha(B)$ the image of B under α and by $\text{bnodes}(B)$ the set of blank nodes used in B; we define $\alpha(N)$ and $\text{bnodes}(N)$ analogously.

Finally, we denote by $R(\mu)$ the result of evaluating the image of the built-in expression R under μ, i.e., the results of replacing all variables v in R (including in nested expressions) with $\mu(v)$ and evaluating the resulting expression. We denote by $\mu \models R$ that μ satisfies R, i.e., that $R(\mu)$ returns a value interpreted as true.

3.4.2. Set vs. bag vs. sequence semantics

SPARQL queries can be evaluated under different semantics, which may return a set of solution mappings M, a bag of solution mappings $\mathfrak{M}$, or a sequence of solution mappings M. Sets are unordered and do not permit duplicates. Bags are unordered and permit duplicates. Sequences are ordered and permit duplicates.

Given a solution mapping μ and a bag of solution mappings $\mathfrak{M}$, we denote by $\mathfrak{M}(\mu)$ the multiplicity of μ in M, i.e., the number of times that μ appears in $\mathfrak{M}$; we say that $\mu \in \mathfrak{M}$ if and only if $\mathfrak{M}(\mu) > 0$ (otherwise, if $\mathfrak{M}(\mu) = 0$, we say that $\mu \notin \mathfrak{M}$). We denote by $|\mathfrak{M}| = \sum_{\mu \in \mathfrak{M}} \mathfrak{M}(\mu)$ the (bag) cardinality of $\mathfrak{M}$. Given two bags $\mathfrak{M}$ and $\mathfrak{M}'$, we say that $\mathfrak{M} \subseteq \mathfrak{M}'$ if and only if $\mathfrak{M}(\mu) \leqslant \mathfrak{M}'(\mu)$ for all $\mu \in \mathbf{M}$. Note that $\mathfrak{M} \subseteq \mathfrak{M}'$ and $\mathfrak{M}' \subseteq \mathfrak{M}$ if and only if $\mathfrak{M} = \mathfrak{M}'$.

Given a sequence M of length n (we denote that $|\mathsf{M}| = n$), we use $\mathsf{M}[i]$ (for $1 \leqslant i \leqslant n$) to denote the i^{th} solution mapping of M, and we say that $\mu \in \mathsf{M}$ if and only if there exists $1 \leqslant i \leqslant n$ such that $\mathsf{M}[i] = \mu$. We denote by $\mathsf{M}[i \dots j]$ (for $1 \leqslant i \leqslant j$) the sub-sequence $(\mathsf{M}[i], \mathsf{M}[i+1], \dots, \mathsf{M}[j-1], \mathsf{M}[j])$ of elements i to j of M, inclusive, in order; if $i = j$, then $\mathsf{M}[i \dots j]$ is defined to be $(\mathsf{M}[i])$; if $i > n$, then $\mathsf{M}[i \dots j]$ is defined to be the empty sequence $()$; otherwise if $j > n$, then $\mathsf{M}[i \dots j]$ is defined to be $\mathsf{M}[i \dots n]$. Given two sequences M_1 and M_2, we denote by $\mathsf{M}_1\mathsf{M}_2$ their concatenation. For a sequence M of length $n > 0$, we define the deletion of index $i \leqslant n$, denoted $\mathrm{del}(\mathsf{M}, i)$, as the concatenation $\mathsf{M}[1 \dots i-1]\mathsf{M}[i+1 \dots n]$ if $1 < i \leqslant n$, or $\mathsf{M}[2 \dots n]$ if $i = 1$, or $\mathsf{M}[1 \dots n-1]$ otherwise. We call j a repeated index of M if there exists $1 \leqslant i < j$ such that $M[i] = M[j]$. We define $\mathrm{dist}(\mathsf{M})$ to be the fixpoint of recursively deleting repeated indexes from M. We say that M' is contained in M, denoted $\mathsf{M}' \subseteq \mathsf{M}$, if and only if we can derive M' by recursively removing zero-or-more indexes from M. Note that $\mathsf{M}' \subseteq \mathsf{M}$ and $\mathsf{M} \subseteq \mathsf{M}'$ if and only if $\mathsf{M} = \mathsf{M}'$.

Next we provide some convenient notation to convert between sets, bags and sequences. Given a sequence M of length n, we denote by $\mathrm{bag}(\mathsf{M})$ the bag that preserves the multiplicity of elements in M (such that $\mathrm{bag}(\mathsf{M})(\mu) := |\{i \mid 1 \leqslant i \leqslant n \text{ and } \mathsf{M}[i] = \mu\}|$). Given a set M, we denote by $\mathrm{bag}(M)$ the bag such that $\mathrm{bag}(M)(\mu) = 1$ if and only if $\mu \in M$; otherwise $\mathrm{bag}(M)(\mu) = 0$. Given a bag $\mathfrak{M}$, we denote by $\mathrm{set}(\mathfrak{M}) := \{\mu \mid \mu \in \mathfrak{M}\}$ the set of elements in $\mathfrak{M}$; we further denote by $\mathrm{seq}(\mathfrak{M})$ a random permutation of the bag $\mathfrak{M}$ (more formally, any sequence $\mathrm{seq}(\mathfrak{M})$ satisfying $\mathrm{bag}(\mathrm{seq}(\mathfrak{M})) = \mathfrak{M}$). Given a sequence M, we use $\mathrm{set}(\mathsf{M})$ as a shortcut for $\mathrm{set}(\mathrm{bag}(\mathsf{M}))$, and given a set M, we use $\mathrm{seq}(M)$ as a shortcut for $\mathrm{seq}(\mathrm{bag}(M))$. Finally, given a set M, a bag $\mathfrak{M}$ and a sequence M, we define that $\mathrm{set}(M) = M$, $\mathrm{bag}(\mathfrak{M}) = \mathfrak{M}$ and $\mathrm{seq}(\mathsf{M}) = \mathsf{M}$.

We continue by defining the semantics of SPARQL queries under set semantics. Later we cover bag semantics, and subsequently discuss aggregation features. Finally we present sequence semantics.

3.5. *Query patterns: Set semantics*

Query patterns evaluated under set semantics return sets of solution mappings without order or duplicates. We first define a set algebra of operators and then define the set evaluation of SPARQL graph patterns.

3.5.1. *Set algebra*

The SPARQL query language can be defined in terms of a relational-style algebra consisting of unary and binary operators [18]. Here we describe the operators of this algebra as they act on sets of solution mappings. Unary operators transform from one set of solution mappings (possibly with additional arguments) to another set of solution mappings. Binary operators transform two sets of solution mappings to another set of solution mappings. In Table 4, we define the operators of this set algebra. This algebra is not minimal: some operators (per, e.g., the definition of left-outer join) can be expressed using the other operators.

3.5.2. *Navigational graph patterns*

Given an RDF graph G, we define the set of terms appearing as a subject or object in G as follows: $\mathrm{so}(G) := \{x \mid \exists p, y : (x, p, y) \in G \text{ or } (y, p, x) \in G\}$. We can then define the evaluation of path expressions as shown in

Table 4

Set algebra, where M, M_1, and M_2 are sets of solution mappings; V is a set of variables and v is a variable; and R is a built-in expression

$M_1 \bowtie M_2 := \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\}$	Natural join
$M_1 \cup M_2 := \{\mu \mid \mu \in M_1 \text{ or } \mu \in M_2\}$	Union
$M_1 \rhd M_2 := \{\mu_1 \in M_1 \mid \nexists \mu_2 \in M_2 : \mu_1 \sim \mu_2\}$	Anti-join
$M_1 - M_2 := \{\mu_1 \in M_1 \mid \nexists \mu_2 \in M_2 : \mu_1 \sim \mu_2 \text{ and } \mu_1 \nsim \mu_2\}$	Minus
$M_1 ⟕ M_2 := (M_1 \bowtie M_2) \cup (M_1 \rhd M_2)$	Left-outer join
$\pi_V(M) := \{\mu' \mid \exists \mu \in M : \mu \sim \mu' \text{ and } \mathrm{dom}(\mu') = V \cap \mathrm{dom}(\mu)\}$	Projection
$\sigma_R(M) := \{\mu \in M \mid \mu \models R\}$	Selection
$\beta_{R,v}(M) := \{\mu \cup \{v/R(\mu)\} \mid \mu \in M\}$	Bind

Table 5

Path expressions where G is an RDF graph, $p, p_1 \dots p_n$ are IRIs, and e, e_1, e_2 are path expressions

$p(G) := \{(s, o) \mid (s, p, o) \in G\}$	Predicate
$!\,(p_1 \mid \dots \mid p_n)\,(G) := \{(s, o) \mid \exists q : (s, q, o) \in G \text{ and } q \notin \{p_1, \dots, p_n\}\}$	Negated property set
$!\,(\hat{}p_1 \mid \dots \mid \hat{}p_n)\,(G) := \{(s, o) \mid \exists q : (o, q, s) \in G \text{ and } q \notin \{p_1, \dots, p_n\}\}$	Negated inverse property set
$!\,(p_1 \mid \dots \mid p_k \mid \hat{}p_{k+1} \mid \dots \mid \hat{}p_n)\,(G) := \,!\,(p_1 \mid \dots \mid p_k)\,(G) \cup \,!\,(\hat{}p_{k+1} \mid \dots \mid \hat{}p_n)\,(G)$	Negated (inverse) property set
$\hat{}e(G) := \{(s, o) \mid (o, s) \in e(G)\}$	Inverse
$e_1/e_2(G) := \{(x, z) \mid \exists y : (x, y) \in e_1(G) \text{ and } (y, z) \in e_2(G)\}$	Concatenation
$e_1 \mid e_2(G) := e_1(G) \cup e_2(G)$	Disjunction
$e{+}(G) := \{(y_1, y_{n+1}) \mid \text{ for } 1 \leqslant i \leqslant n : \exists (y_i, y_{i+1}) \in e(G)\}$	One-or-more
$e{\star}(G) := e{+}(G) \cup \{(x, x) \mid x \in \text{so}(G)\}$	Zero-or-more
$e?(G) := e(G) \cup \{(x, x) \mid x \in \text{so}(G)\}$	Zero-or-one

Table 5 [34], which returns a set of pairs of nodes connected by a path in the RDF graph G that satisfies the given path expression.

Given a navigational graph pattern N, we denote by $\text{paths}(N) := \{e \in \mathbf{P} \mid \exists s, o : (s, e, o) \in N\}$ the set of path expressions used in N (including simple IRIs, but not variables). We define the *path graph* of G under N, which we denote by G^N, as the set of triples that materialise paths of N in G; more formally $G^N := \{(s, e, o) \mid e \in \text{paths}(N) \text{ and } (s, o) \in e(G)\}$.

3.5.3. *Service federation*

The SERVICE feature allows for sending graph patterns to remote SPARQL services. In order to define this feature, we denote by ω a *federation mapping* from IRIs to services such that, given an IRI $x \in \mathbf{I}$, then $\omega(x)$ returns the service $\mathcal{S}$ hosted at x or returns ε in the case that no service exists or can be retrieved. We denote by $\mathcal{S}.Q(D_{\mathcal{S}})$ the evaluation of a query Q on a remote service $\mathcal{S}$. When a service of the query evaluation is not indicated (e.g., $Q(D)$), we assume that it is evaluated on the local service. Finally, we define that $\varepsilon.Q(D_\varepsilon)$ invokes a query-level error ε – i.e., the evaluation of the entire query fails – while $\varepsilon.Q^*(D_\varepsilon)$ returns a set with the empty solution mapping $\{\mu_\emptyset\}$.[3]

3.5.4. *Set evaluation*

The set evaluation of a SPARQL graph pattern transforms a SPARQL dataset D into a set of solution mappings. The base evaluation is given in terms of $B(D)$ and $N(D)$, i.e., the evaluation of a basic graph pattern B and a navigational graph pattern N over a dataset D, which generate sets of solution mappings. These solution mappings can then be transformed and combined using the aforementioned set algebra by invoking the corresponding pattern. The set evaluation of graph patterns is then defined in Table 6. We remark that for the definition of FE (filter exists) and FNE (filter not exists), there is some ambiguity about what $\mu(Q_2)$ precisely means when Q_2 involves variables mentioned outside of a basic graph pattern or a path expression; this is a known issue for the SPARQL 1.1 standard [27,32,42,55], which we will discuss in more detail in Section 3.10.

3.6. *Query patterns: Bag semantics*

Query patterns evaluated under bag semantics return bags of solution mappings. Like under set semantics, we first define a bag algebra of operators and then define the bag evaluation of SPARQL graph patterns.

3.6.1. *Bag algebra*

The bag algebra is analogous to the set algebra, but further operates over the multiplicity of solution mappings. We define this algebra in Table 7.

[3]We remark that $\{\mu_\emptyset\}$ is the join identity; i.e., $\{\mu_\emptyset\} \bowtie M = M$. On the other hand $\{\}$ is the join zero; i.e., $\{\} \bowtie M = \{\}$.

Table 6

Set evaluation of graph patterns where D is a dataset; B is a basic graph pattern; N is a navigational graph pattern; Q, Q_1 and Q_2 are graph patterns; V is a set of variables; R is a built-in expression; v is a variable; M is a set of solution mappings; and x is an IRI

$B(D) := \{\mu \mid \exists\alpha : \mu(\alpha(B)) \subseteq G_D \text{ and } \mathrm{dom}(\mu) = \mathrm{vars}\, B \text{ and } \mathrm{dom}(\alpha) = \mathrm{bnodes}(B)\}$
$N(D) := \{\mu \mid \exists\alpha : \mu(\alpha(N)) \subseteq G_D^N \cup G_D \text{ and } \mathrm{dom}(\mu) = \mathrm{vars}\, N \text{ and } \mathrm{dom}(\alpha) = \mathrm{bnodes}(N)\}$
$[Q_1 \, \mathrm{AND} \, Q_2](D) := Q_1(D) \bowtie Q_2(D)$
$[Q_1 \, \mathrm{UNION} \, Q_2](D) := Q_1(D) \cup Q_2(D)$
$[Q_1 \, \mathrm{FE} \, Q_2](D) := \{\mu \in Q_1(D) \mid (\mu(Q_2))(D) \neq \emptyset\}$ (see † below)
$[Q_1 \, \mathrm{FNE} \, Q_2](D) := \{\mu \in Q_1(D) \mid (\mu(Q_2))(D) = \emptyset\}$ (see † below)
$[Q_1 \, \mathrm{MINUS} \, Q_2](D) := Q_1(D) - Q_2(D)$
$[Q_1 \, \mathrm{OPT} \, Q_2](D) := Q_1(D) ⟕ Q_2(D)$
$\mathrm{SELECT}_V(Q)(D) := \pi_V(Q(D))$
$\mathrm{FILTER}_R(Q)(D) := \sigma_R(Q(D))$
$\mathrm{BIND}_{R,v}(Q)(D) := \beta_{R,v}(Q(D))$
$\mathrm{VALUES}_M(Q)(D) := Q(D) \bowtie M$
$\mathrm{GRAPH}_x(Q)(D) := Q((G_{D[x]}, D^*))$
$\mathrm{GRAPH}_v(Q)(D) := \bigcup_{(x_i, G_i) \in D^*} \beta_{x_i, v}(Q((G_i, D^*)))$
$[Q_1 \, \mathrm{SERVICE}_x^{\mathtt{false}} \, Q_2](D) := Q_1(D) \bowtie \omega(x).Q_2(D_{\omega(x)})$
$[Q_1 \, \mathrm{SERVICE}_x^{\mathtt{true}} \, Q_2](D) := Q_1(D) \bowtie \omega(x).Q_2^*(D_{\omega(x)})$

† $\mu(Q_2)$ is not well-defined in all cases. Please see Section 3.10 for discussion.

Table 7

Bag algebra where $\mathfrak{M}$, $\mathfrak{M}_1$, and $\mathfrak{M}_2$ are bags of solution mappings; μ, μ', μ_1 and μ_2 are solution mappings; V is a set of variables and v is a variable; and R is a built-in expression; for legibility, we use iverson bracket notation where $[\phi] = 1$ if and only if ϕ holds; otherwise, if ϕ is false or undefined, then $[\phi] = 0$

$\mathfrak{M}_1 \bowtie \mathfrak{M}_2(\mu) := \sum_{(\mu_1, \mu_2) \in \mathfrak{M}_1 \times \mathfrak{M}_2} \mathfrak{M}_1(\mu_1) \cdot \mathfrak{M}_2(\mu_2) \cdot [\mu_1 \sim \mu_2 \text{ and } \mu_1 \cup \mu_2 = \mu]$	Natural join
$\mathfrak{M}_1 \cup \mathfrak{M}_2(\mu) := \mathfrak{M}_1(\mu) + \mathfrak{M}_2(\mu)$	Union
$\mathfrak{M}_1 \rhd \mathfrak{M}_2(\mu) := \mathfrak{M}_1(\mu) \cdot [\nexists \mu' \in \mathfrak{M}_2 : \mu \sim \mu']$	Anti-join
$\mathfrak{M}_1 - \mathfrak{M}_2(\mu) := \mathfrak{M}_1(\mu) \cdot [\nexists \mu' \in \mathfrak{M}_2 : \mu \sim \mu' \text{ and } \mu * \mu']$	Minus
$\mathfrak{M}_1 ⟕ \mathfrak{M}_2(\mu) := ((\mathfrak{M}_1 \bowtie \mathfrak{M}_2) \cup (\mathfrak{M}_1 \rhd \mathfrak{M}_2))(\mu)$	Left-outer join
$\pi_V(\mathfrak{M})(\mu) := \sum_{\mu' \in \mathfrak{M}} \mathfrak{M}(\mu') \cdot [\mathrm{dom}(\mu') = V \cap \mathrm{dom}(\mu) \text{ and } \mu \sim \mu']$	Projection
$\sigma_R(\mathfrak{M})(\mu) := \mathfrak{M}(\mu) \cdot [\mu \models R]$	Selection
$\beta_{R,v}(\mathfrak{M})(\mu) := \sum_{\mu' \in \mathfrak{M}} \mathfrak{M}(\mu') \cdot [\mu' \cup \{v/R(\mu')\} = \mu]$	Bind

3.6.2. Bag evaluation

The bag evaluation of a graph pattern is based on the bag evaluation of basic graph patterns and navigational graph patterns, as defined in Table 8, where the multiplicity of each individual solution is based on how many blank node mappings satisfy the solution. With the exceptions of FE and FNE – which are also defined in Table 8 – the bag evaluation of other graph patterns then follows from Table 6 by simply replacing the set algebra (from Table 4) with the bag algebra (from Table 7). Note that $\mathrm{VALUES}_{\mathfrak{M}}(Q)$ can now also accept a bag of solutions, and that there are again issues with the definition of FE and FNE that will be discussed in Section 3.10. The set evaluation of paths (from Table 5) is again used with the exception that some path expressions in navigational patterns are rewritten (potentially recursively) to analogous query operators under bag semantics. Table 9 lists these rewritings.

3.7. Group-by patterns: Aggregation

Let $(\mu, \mathfrak{M})$ denote a *solution group*, where μ is a solution mapping called the *key of the solution group*, and $\mathfrak{M}$ is a bag of solution mappings called the *bag of the solution group*. Group-by patterns then return a set of solution groups $\mathcal{M} := \{(\mu_1, \mathfrak{M}_1), \ldots, (\mu_n, \mathfrak{M}_n)\}$ as their output. We now define their semantics along with the AGG graph pattern, which allows for converting a set of solution groups to a set of solution mappings.

Table 8

Bag evaluation of graph patterns where D is a dataset; B is a basic graph pattern; N is a navigational graph pattern; Q_1, Q_2 are graph patterns

$B(D)(\mu) := \lvert\{\alpha \mid \mu(\alpha(B)) \subseteq G_D \text{ and } \mathrm{dom}(\alpha) = \mathrm{bnodes}(B)\}\rvert \cdot [\mathrm{dom}(\mu) = \mathrm{vars}\, B]$
$N(D)(\mu) := \lvert\{\alpha \mid \mu(\alpha(N)) \subseteq G_D^N \text{ and } \mathrm{dom}(\alpha) = \mathrm{bnodes}(N)\}\rvert \cdot [\mathrm{dom}(\mu) = \mathrm{vars}\, N]$
$[Q_1 \text{FE} Q_2](D)(\mu) := Q_1(D)(\mu) \cdot [(\mu(Q_2))(D) \neq \emptyset]$ (see † below)
$[Q_1 \text{FNE} Q_2](D)(\mu) := Q_1(D)(\mu) \cdot [(\mu(Q_2))(D) = \emptyset]$ (see † below)

† $\mu(Q_2)$ is not well-defined in all cases. Please see Section 3.10 for discussion.

Table 9

Bag evaluation of navigational patterns where D is a dataset, N is a navigational pattern, and x is a fresh blank node

$N(D) :=$	$\{(o, e, s)\} \cup (N \setminus \{(s, \hat{}e, o)\})(D)$	if there exists $(s, \hat{}e, o) \in N$
	$\{(s, e_1, x), (x, e_2, o)\} \cup (N \setminus \{(s, e_1/e_2, o)\})(D)$	if there exists $(s, e_1/e_2, o) \in N$
	$[[\{(s, e_1, o)\} \text{UNION} \{(s, e_2, o)\}] \text{AND} N \setminus \{(s, e_1 \mid e_2, o)\}](D)$	if there exists $(s, e_1 \mid e_2, o) \in N$
	$N(D)$ under set semantics	otherwise

Table 10

Aggregation algebra under bag semantics, where $\mathfrak{M}$ and $\mathfrak{M}'$ are bags of solution mappings; V is a set of variables and v is a variable; A is an aggregation expression; and $\mathcal{M}$ is a set of solution groups; we recall that $\mathbf{M}$ is the set of all solution mappings

$\gamma_V(\mathfrak{M}) := \{(\mu, \mathfrak{M}') \mid \mu \in \pi_V(\mathfrak{M}) \text{ and } \forall \mu' \in \mathbf{M} : \mathfrak{M}'(\mu') = \mathfrak{M}(\mu') \cdot [\pi_V(\{\mu'\}) = \{\mu\}]\}$	Group (bag)
$\sigma'_A(\mathcal{M}) := \{(\mu, \mathfrak{M}) \in \mathcal{M} \mid \mathfrak{M} \models A\}$	Selection (aggregation)
$\beta'_{A,v}(\mathcal{M}) := \{(\mu \cup \{v/A(\mathfrak{M})\}, \beta_{A(\mathfrak{M}),v}(\mathfrak{M})) \mid (\mu, \mathfrak{M}) \in \mathcal{M}\}$	Bind (aggregation)
$\zeta(\mathcal{M}) := \{\mu \mid (\mu, \mathfrak{M}) \in \mathcal{M}\}$	Flatten

Table 11

Evaluation of group-by patterns where D is a dataset, Q is a graph pattern or group-by pattern, V is a set of variables, $v_1, \ldots, v_n$ are variables, and $A, A_1, \ldots, A_n$ are aggregation expressions

$\text{GROUP}_V(Q)(D) := \gamma_V(Q(D))$
$\text{HAVING}_A(Q)(D) := \sigma'_A(Q(D))$
$\text{AGG}_{\{(A_1,v_1),\ldots,(A_n,v_n)\}}(Q)(D) := \zeta(\beta'_{A_1,v_1}(\ldots(\beta'_{A_n,v_n}(Q(D)))\ldots))$ note: $\text{AGG}_{\{\}}(Q)(D) := \zeta(Q(D))$

3.7.1. Aggregation algebra

We define an aggregation algebra in Table 10 under bag semantics with four operators that support the generation of a set of solution groups (aka., *group by*), the selection of solution groups (aka., *having*), the binding of new variables in the key of the solution group, as well as the flattening of a set of solution groups to a set of solution mappings by projecting their keys. Note that analogously to the notation for built-in expressions, given an aggregation expression A, we denote by $A(\mathfrak{M})$ the result of evaluating A over $\mathfrak{M}$, and we denote by $\mathfrak{M} \models A$ the condition that $\mathfrak{M}$ satisfies A, i.e., that $A(\mathfrak{M})$ returns a value interpreted as true. The aggregation algebra can also be defined under set semantics: letting M denote a set of solution mappings, we can evaluate $\gamma_V(\mathrm{bag}(M))$ before other operators.

3.7.2. Aggregation evaluation

We can use the previously defined aggregation algebra to define the semantics of group-by patterns in terms of their evaluation, per Table 11.

3.8. Sequence patterns and semantics

Sequence patterns return sequences of solutions as their output, which allow duplicates and also maintain an ordering. These sequence patterns in general refer to solution modifiers that allow for ordering solutions, slicing the set of solutions, and removing duplicate solutions. We will again first define an algebra before defining the evaluation of sequence patterns.

Table 12

Sequence algebra, where M and M' are sequences of solutions, and Ω is a non-empty sequence of order comparators

$\mathrm{order}_\Omega(\mathsf{M}) := \mathsf{M}'$ such that $\mathrm{bag}(\mathsf{M}') = \mathrm{bag}(\mathsf{M})$ and $\mathsf{M}'[i] \leqslant_\Omega \mathsf{M}'[j]$ for all $1 \leqslant i < j \leqslant \lvert\mathsf{M}'\rvert$	Order by
$\mathrm{distinct}(\mathsf{M}) := \mathrm{dist}(\mathsf{M})$	Distinct
$\mathrm{reduced}(\mathsf{M}) := \mathsf{M}'$ such that $\mathsf{M}' \subseteq \mathsf{M}$ and $\mathrm{set}(\mathsf{M}) = \mathrm{set}(\mathsf{M}')$	Reduced

Table 13

Evaluation of sequence patterns where Ω is a non-empty sequence of order comparators, and k is a (non-zero) natural number

$\mathrm{ORDER}_\Omega(Q)(D) := \mathrm{order}_\Omega(\mathrm{seq}(Q(D)))$
$\mathrm{DISTINCT}(Q)(D) := \mathrm{distinct}(\mathrm{seq}(Q(D)))$
$\mathrm{REDUCED}(Q)(D) := \mathrm{reduced}(\mathrm{seq}(Q(D)))$
$\mathrm{OFFSET}_k(Q)(D) := \mathrm{seq}(Q(D))[(k+1)\ldots\infty]$
$\mathrm{LIMIT}_k(Q)(D) := \mathrm{seq}(Q(D))[1\ldots k]$

3.8.1. *Sequence algebra*

Sequences deal with some ordering over solutions. We assume a total order $\leqslant$ over $\mathbf{IBL} \cup \{\bot\}$ to be defined by the service (see Section 3.3), i.e., over the set of all RDF terms and unbounds. Given a non-empty sequence of order comparators $\Omega := ((R_1, \Delta_1), \ldots, (R_n, \Delta_n))$, we define the total ordering $\leqslant_\Omega$ of solutions mappings as follows:

- $\mu_1 =_\Omega \mu_2$ if and only if $R_i(\mu_1) = R_i(\mu_2)$ for all $1 \leqslant i \leqslant n$;
- otherwise let j denote the least value $1 \leqslant j \leqslant n$ such that $R_j(\mu_1) \neq R_j(\mu_2)$; then:
  * $R_j(\mu_1) < R_j(\mu_2)$ and Δ_k implies $\mu_1 <_\Omega \mu_2$;
  * $R_j(\mu_1) > R_j(\mu_2)$ and Δ_k implies $\mu_1 >_\Omega \mu_2$;
  * $R_j(\mu_1) < R_j(\mu_2)$ and not Δ_k implies $\mu_1 >_\Omega \mu_2$;
  * $R_j(\mu_1) > R_j(\mu_2)$ and not Δ_k implies $\mu_1 <_\Omega \mu_2$.

In Table 12, we present an algebra composed of three operators for ordering sequences of solutions based on order comparators, and removing duplicates.

3.8.2. *Sequence evaluation*

Using the sequence algebra, we can then define the evaluation of sequence patterns as shown in Table 13.

3.9. *Safe and possible variables*

We now characterise different types of variables that may appear in a graph pattern in terms of being always bound, never bound, or sometimes bound in the solutions to the graph pattern. This characterisation will become important for rewriting algebraic expressions [53]. Specifically, letting Q denote a graph pattern, recall that we denote by vars Q all of the variables mentioned (possibly nested) in Q; furthermore:

- we denote by $\mathrm{svars}(Q)$ the *safe variables* of Q, defined to be the set of variables $v \in \mathbf{V}$ such that, for all datasets D, if $\mu \in Q(D)$, then $v \in \mathrm{dom}(\mu)$;
- we denote by $\mathrm{pvars}(Q)$ the *possible variables* of Q, defined to be the set of variables $v \in \mathrm{vars}\ Q$ such that there exists a dataset D and a solution $\mu \in Q(D)$ where $v \in \mathrm{dom}(\mu)$.

Put more simply, $\mathrm{svars}(Q)$ denotes variables that are never unbound in any solution, while $\mathrm{pvars}(Q)$ denotes variables that may be unbound but are bound in at least one solution over some dataset.[4]

[4]It may be tempting to think that $\mathrm{svars}(Q) \subseteq \mathrm{pvars}(Q)$, but if Q never returns results (e.g., $Q = [Q' \mathrm{MINUS} Q']$), then $\mathrm{svars}(Q) = \mathbf{V}$ and $\mathrm{pvars}(Q) = \emptyset$ per the previous definitions.

Example 3.1. Consider the query (pattern) Q:

```
SELECT * WHERE {
  { { ?s :sister ?x } UNION { ?s :brother ?y } }
  MINUS { ?s :twin ?z }
}
```

Now:

- $\text{vars}\ Q = \{\texttt{?s}, \texttt{?x}, \texttt{?y}, \texttt{?z}\}$;
- $\text{svars}(Q) = \{\texttt{?s}\}$;
- $\text{pvars}(Q) = \{\texttt{?s}, \texttt{?x}, \texttt{?y}\}$.

Unfortunately, given a graph pattern Q, deciding if $v \in \text{svars}(Q)$ or $v \in \text{pvars}(Q)$ is undecidable for the full SPARQL 1.1 language as it can answer the question of the satisfiability of Q, i.e., whether or not Q has any solution over any dataset; specifically, $v \notin \text{vars}\ Q$ is safe in Q if and only if Q is unsatisfiable, while $v \notin \text{vars}\ Q$ is possible in $\text{BIND}_{1,v}(Q)$ if and only if Q is satisfiable. For this reason we resort to syntactic approximations of the notions of safe and possible variables. In fact, when we say that Q_1 is a graph pattern on V_1, we can consider V_1 to be a syntactic over-approximation of the possible variables of Q (called "in-scope" variables by the standard [24]), which ensures no clashes of variables in solutions (e.g., defining $\text{BIND}_{1,v}(Q)$ when Q can bind v). Later (in Table 21) we will define a syntactic over-approximation of safe variables to decide when it is okay to apply rewriting rules that are invalid in the presence of unbound variables.

3.10. *Issues with (NOT)* EXISTS

The observant reader may have noticed that in Table 6 and Table 8, in the definitions of $[Q_1 \text{FE} Q_2]$ (FILTER EXISTS) and $[Q_1 \text{FNE} Q_2]$ (FILTER NOT EXISTS), we have used the expression $\mu(Q_2)$. While we have defined this for a basic graph pattern B and a navigational graph pattern N – replace any variable $v \in \text{dom}(\mu)$ appearing in B or N, respectively, with $\mu(v)$ – per the definition of the syntax, Q_2 can be any graph pattern. This leaves us with the question of how $\mu(\text{SELECT}_V(Q'))$ or $\mu([Q'_1 \text{MINUS} Q'_2])$, for example, is defined. Even in the case where Q_2 is a basic graph pattern, it is unclear how we should handle variables that are replaced with blank nodes, or predicate variables that are replaced with literals. The standard does not define such cases unambiguously [27,32,42,55]. The precise semantics of $[Q_1 \text{FE} Q_2]$ and $[Q_1 \text{FNE} Q_2]$ thus cannot be defined until the meaning of $\mu(v)$ – which the standard calls *substitution* – is clarified. We provide an example illustrating one issue that arises.

Example 3.2. We will start with a case that is not semantically ambiguous. Take a query:

```
SELECT DISTINCT * WHERE {
  { ?x :sister ?y  }
  FILTER NOT EXISTS { ?y :sister ?z }
}
```

To evaluate it on a dataset D, we take each solution $\mu \in \{(\texttt{?x}, \texttt{:sister}, \texttt{?y})\}(D)$ from the left of the FILTER NOT EXISTS and keep the solution μ in the final results of the query if and only if the evaluation of the pattern $\{(\mu(\texttt{?y}), \texttt{:sister}, \texttt{?z})\}(D)$ is empty.

We next take an example of a query that is syntactically valid, but semantically ill-defined.

```
SELECT DISTINCT * WHERE {
  { ?x :sister ?y  }
  FILTER NOT EXISTS {  SELECT ?y WHERE { ?y :sister ?z } }
}
```

Given a solution μ from the left, if we follow the standard literally and replace "*every occurrence of a variable v in* [the right pattern] *by $\mu(v)$ for each v in* $\text{dom}(\mu)$", the result is a pattern $\text{SELECT}_{\{\mu(?y)\}}(\mu(Q_2))$. For example, if $\mu(?y) = \texttt{:a}$, then the right pattern, in concrete syntax, would become:

```
SELECT :a WHERE { :a :sister ?z }
```

which is syntactically invalid.

A number of similar issues arise from ambiguities surrounding substitution, and while work is underway to clarify this issue, at the time of writing, various competing proposals are being discussed [32,42,55]. We thus postpone rewriting rules for negation until a standard semantics for substitution is agreed upon.

3.11. Queries

A query accepts a set, bag or sequence of solution modifiers, depending on the semantics selected (and features supported). In the case of a SELECT query, the output will likewise be a set, bag or sequence of solution modifiers, potentially projecting away some variables. An ASK query rather outputs a boolean value. Finally, CONSTRUCT and DESCRIBE queries output an RDF graph. We will define the evaluation of these queries in terms of solution sequences, though the definitions generalise naturally to bags and sets (through bag$(\cdot)$ and set$(\cdot)$). First we give preliminary notation. Given a sequence of solution mappings M, we denote by $\pi_V(\mathsf{M})$ a projection that preserves the input order of solution mappings (and such that $\text{bag}(\pi_V(\mathsf{M})) = \pi_V(\text{bag}(\mathsf{M}))$). Given a dataset D and a set of RDF terms X, we assume a function describe(X, D) that returns an RDF graph "describing" each term $x \in X$ to be defined by the service (see Section 3.3); as a simple example, describe(X, D) may be defined as the set of triples in G_D that mention any $x \in X$. The evaluation of queries is then shown in Table 14.

3.12. Dataset modifier

Queries are evaluated on a SPARQL dataset D, where dataset modifiers allow for changing the dataset considered for query evaluation. First, let X and X' denote (possibly empty or overlapping) sets of IRIs and let D denote a SPARQL dataset. We then denote by $D(X, X') := (\biguplus_{x \in X} G_{D[x]}, \{(x', G_{D[x']}) \mid x' \in X'\})$ a new dataset formed from D by merging all named graphs of D named by X to form the default graph of $D(X, X')$, and by selecting all named graphs of D named by X' as the named graphs of $D(X, X')$. We define the semantics of dataset modifiers in Table 15.

3.13. Non-determinism

A number of features can lead to non-determinism in the evaluation of graph patterns as previously defined. When such features are used, there may be more than one possible valid result for the graph pattern on a dataset. These features are as follows:

– Built-in expressions and aggregation expressions may rely on non-deterministic functions, such as `rand()` to generate a random number, `SAMPLE` to randomly sample solutions from a group, etc.

Table 14

Evaluation of queries where D is a dataset, Q is a sequence pattern or graph pattern, V is a set of variables, B is a basic graph pattern, and X is a set of IRIs and/or variables

$\text{SELECT}_V(Q)(D) := \pi_V(Q(D))$
$\text{ASK}(Q)(D) :=$ true if $\text{set}(Q(D)) \neq \emptyset$; false otherwise
$\text{CONSTRUCT}_B(Q)(D) := \biguplus_{\mu \in Q(D)} \{\mu(s, p, o) \in \mathbf{IB} \times \mathbf{I} \times \mathbf{IBL} \mid (s, p, o) \in B\}$
$\text{DESCRIBE}_X(Q)(D) := \text{describe}(X', D)$ where $X' := \{x' \in \mathbf{IBL} \mid x' \in X$, or $\exists x \in X \cap \mathbf{V}, \mu \in Q(D) : \mu(x) = x'\}$

Table 15

Evaluation of dataset modifiers where X and X' are sets of IRIs

$\text{FROM}_{X,X'}(Q)(D) := Q(D(X, X'))$

– $\text{REDUCED}(Q)(D)$ permits a range of multiplicities for solutions (between those for $Q(D)$ under bag semantics and $\text{DISTINCT}(Q)(D)$).
– The use of sequence patterns without an explicit $\text{ORDER}_\Omega(Q)$ gives a non-deterministic ordering (e.g., with $\text{OFFSET}_k(Q)$ and/or $\text{LIMIT}_k(Q)$).

In non-deterministic cases, we can say that $Q(D)$ returns a family of (potentially infinite) valid sets/bags/sequences of solutions, denoting the space of possible results for evaluating the graph pattern. In practice any such set/bag/sequence of solutions can be returned. If $Q(D)$ returns a singleton family for all datasets D, we consider Q to be deterministic (even if using a non-deterministic feature), where it returns the set/bag/sequence of solutions rather than the singleton; for example, we assume that $\text{REDUCED}(Q)$ is deterministic if Q cannot generate duplicates.

3.14. Relationships between the semantics

The SPARQL standard is defined in terms of sequence semantics, i.e., it is assumed that the solutions returned have an order. However, unless the query explicitly uses the sequence algebra (and in particular `ORDER BY`), then the semantics is analogous to bag semantics in the sense that the ordering of solutions in the results is arbitrary. Likewise when a query does not use the sequence algebra or the aggregation algebra, but invokes `SELECT DISTINCT` (in the outermost query), `ASK`, `CONSTRUCT` or `DESCRIBE`, then the semantics is analogous to set semantics. Note however that when the aggregate or sequence algebra is included, set semantics is not the same as bag semantics with `DISTINCT`. Under set semantics, intermediate results are treated as sets of solutions. Under bag semantics, intermediate results are treated as bags of solutions, where final results are deduplicated. If we apply a count aggregation, for example, then set semantics will disregard duplicate solutions, while bag semantics with distinct will consider duplicate solutions (the distinct is applied to the final count, with no effect).

3.15. Query containment and equivalence

Query containment states that the results of one graph pattern are contained in the other. To begin, take two deterministic graph patterns Q_1 and Q_2. We say that Q_1 *is contained in* Q_2 under set, bag or sequence semantics, denoted $Q_1 \sqsubseteq Q_2$, if and only if for every dataset D, it holds that $Q_1(D) \subseteq Q_2(D)$.

If Q_1 and Q_2 are non-deterministic, then under set semantics we assume that $Q_1(D)$ and $Q_2(D)$ will return a family of sets of solutions. If for every dataset D, and for all $M_1 \in Q_1(D)$, there exists an $M_2 \in Q_1(D)$ such that $M_1 \subseteq M_2$, then we say that Q_1 is contained in Q_2 under set semantics, again denoted $Q_1 \sqsubseteq Q_2$. On the other hand, if Q_1 is deterministic, and Q_2 is non-deterministic, then $Q_1 \sqsubseteq Q_2$ if and only if for every dataset D and for all $M \in Q_2(D)$, it holds that $Q_1(D) \subseteq M$. Conversely if Q_2 is deterministic, and Q_1 is non-deterministic, then $Q_1 \sqsubseteq Q_2$ if and only if for every dataset D and for all $M_1 \in Q_1(D)$, it holds that $M_1 \subseteq Q_2(D)$. Containment can be defined analogously for bags or sequences.

Query equivalence is a relation between graph patterns that states that the results of one graph pattern are equal to the other. Specifically, given two graph patterns Q_1 and Q_2 (be they deterministic or non-deterministic), we say that they are equivalent under set, bag or sequence semantics, denoted $Q_1 \equiv Q_2$, if and only if for every dataset D, it holds that $Q_1(D) = Q_2(D)$. We remark that if Q_1 and Q_2 are deterministic, then $Q_1 \equiv Q_2 \Leftrightarrow Q_1 \sqsubseteq Q_2 \wedge Q_2 \sqsubseteq Q_1$ under the corresponding semantics. If they are non-deterministic, then $Q_1 \equiv Q_2 \Rightarrow Q_1 \sqsubseteq Q_2 \wedge Q_2 \sqsubseteq Q_1$, but $Q_1 \equiv Q_2 \not\Leftarrow Q_1 \sqsubseteq Q_2 \wedge Q_2 \sqsubseteq Q_1$.[5]

Example 3.3. In Fig. 1 we provide examples of query containment and equivalence. The leftmost query finds the maternal grandparents of `:Bob` while the latter three queries find both maternal and paternal grandparents. Hence the first query is contained in the latter three queries, which are themselves equivalent.

[5]For example, if $Q_1(D) = \{\{\mu_1, \mu_2\}, \{\mu_1\}\}$ and $Q_2(D) = \{\{\mu_1, \mu_2\}, \{\mu_2\}\}$, this is consistent with Q_1 and Q_2 being contained in each other, but not with their being equivalent.

```
SELECT DISTINCT ?gp
WHERE {
  :Bob :mother ?p .
  { ?p :mother ?gp }
  UNION
  { ?p :father ?gp }
}
```

$\sqsubseteq$

```
SELECT DISTINCT ?gp
WHERE {
  { :Bob :mother ?p }
  UNION
  { :Bob :father ?p }
  { ?p :mother ?gp }
  UNION
  { ?p :father ?gp }
}
```

$\equiv$

```
SELECT DISTINCT ?gp
WHERE {
  { :Bob :mother [ :mother ?gp ] .}
  UNION
  { :Bob :mother [ :father ?gp ] .}
  UNION
  { :Bob :father [ :mother ?gp ] .}
  UNION
  { :Bob :father [ :father ?gp ] .}
}
```

$\equiv$

```
SELECT DISTINCT ?gp
WHERE {
  :Bob (:mother|:father)/
     (:mother|:father) ?gp .
}
```

Fig. 1. Examples of query containment and equivalence.

```
SELECT DISTINCT ?gp
WHERE {
  { :Bob :mother ?p }
  UNION
  { :Bob :father ?p }
  { ?p :mother ?gp }
  UNION
  { ?p :father ?gp }
}
```

$\cong$

```
SELECT DISTINCT ?z
WHERE {
   :Bob :father|:mother ?y .
   ?y :father|:mother ?z .
}
```

Fig. 2. Example of query congruence.

Regarding equivalence of non-deterministic graph patterns, we highlight that any change to the possible space of results leads to a non-equivalent graph pattern. For example, for a graph pattern Q, it holds that $\text{DISTINCT}(Q) \sqsubseteq \text{REDUCED}(Q) \sqsubseteq Q$, and if Q cannot return duplicates (e.g., Q is a basic graph pattern without blank nodes), then $\text{DISTINCT}(Q) \equiv \text{REDUCED}(Q) \equiv Q$. However, if Q may give duplicates, then $\text{DISTINCT}(Q) \not\equiv \text{REDUCED}(Q) \not\equiv Q \not\equiv \text{DISTINCT}(Q)$ under bag or sequence semantics. Likewise, for example, replacing a function like `RAND()` in Q with a constant like `0.5` changes the semantics of Q, generating a non-equivalent graph pattern.

While the previous discussion refers to graph patterns (which may include use of (sub)`SELECT`), we remark that containment and equivalence can be defined for `ASK`, `CONSTRUCT` and `DESCRIBE` in a natural way. For two deterministic `ASK` queries Q_1 and Q_2, we say that Q_1 is contained in Q_2, denoted $Q_1 \sqsubseteq Q_2$, if and only if for any dataset D, it holds that $Q_1(D)$ implies $Q_2(D)$; i.e., for any dataset which Q_1 returns true, Q_2 also returns true. For two deterministic `CONSTRUCT` queries or `DESCRIBE` queries Q_1 and Q_2, we say that Q_1 is contained in Q_2, denoted $Q_1 \sqsubseteq Q_2$, if and only if for any dataset D, it holds that $Q_2(D) \models Q_1(D)$ under simple entailment. Two queries Q_1 and Q_2 are then equivalent, denoted $Q_1 \equiv Q_2$ if and only if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$. Containment and equivalence of non-deterministic queries are then defined as before.

3.16. Query isomorphism and congruence

Many use-cases for canonicalisation prefer not to distinguish queries that are equivalent up to variable names. We call a one-to-one variable mapping $\rho : \mathbf{V} \to \mathbf{V}$ a *variable renaming*. We say that Q_1 and Q_2 are *isomorphic*, denoted $Q_1 \simeq Q_2$ if and only if there exists a variable renaming ρ such that $\rho(Q_1) = Q_2$. Note that in the case of `SELECT` queries, isomorphism does not imply equivalence as variable naming matters to the solutions produced. For this reason we introduce the notion of *query congruence*. Formally we say that two graph patterns Q_1 and Q_2 are *congruent*, denoted $Q_1 \cong Q_2$, if and only if there exists a variable renaming ρ such that $\rho(Q_1) \equiv Q_2$. It is not difficult to see that isomorphism implies congruence.

Example 3.4. We provide an example of non-equivalent but congruent queries in Fig. 2. If we rewrite the variable `?gp` to `?x` in the first query, we see that the two queries become equivalent.

Like equivalence and isomorphism, congruence is reflexive ($Q \cong Q$), symmetric ($Q_1 \cong Q_2 \Leftrightarrow Q_2 \cong Q_1$) and transitive ($Q_1 \cong Q_2 \wedge Q_2 \cong Q_3 \Rightarrow Q_1 \cong Q_3$); in other words, congruence is an equivalence relation. We remark that congruence is the same as equivalence for `ASK`, `CONSTRUCT` and `DESCRIBE` queries since the particular choice of variable names does not affect the output of such queries in any way.

3.17. *Query classes*

Based on a query of the form $\text{SELECT}_V(Q)$, we define eleven syntactic query classes corresponding to classes that have been well-studied in the literature.

- *basic graph patterns* (BGPs): Q is a BGP and $V = \text{vars}\, Q$.
- *unions of basic graph patterns* (UBGPs): Q is a graph pattern using BGPs and UNION and $V = \text{vars}\, Q$.
- *conjunctive queries* (CQs): Q is a BGP.
- *unions of conjunctive queries* (UCQs): Q is a graph pattern using BGPs and UNION.
- *monotone queries* (MQs): Q is a graph pattern using BGPs, UNION and AND.[6]
- *non-monotone queries* (NMQs): Q is a graph pattern using BGPs, UNION, AND and MINUS.
- *navigational graph patterns* (NGPs): Q is an NGP and $V = \text{vars}\, Q$.
- *unions of navigational graph patterns* (UNGPs): Q is a graph pattern using NGPs and UNION and $V = \text{vars}\, Q$.
- *conjunctive path queries* (CPQs): Q is an NGP.
- *unions of conjunctive path queries* (UCPQs): Q is a graph pattern using NGPs and UNION.
- *monotone path queries* (MPQs): Q is a graph pattern using NGPs, UNION and AND.
- *non-monotone path queries* (NMPQs): Q is a graph pattern using NGPs, UNION, AND and MINUS.

These query classes are evaluated on an RDF graph (the default graph) rather than an RDF dataset, though results extend naturally to the RDF dataset case. Likewise, since we do not consider the sequence algebra, we have the (meaningful) choice of set or bag semantics under which to consider the tasks; furthermore, since the aggregation algebra is not considered, set and distinct-bag semantics coincide.

Unlike UCQs, which are strictly unions of joins (expressed as basic graph patterns), MQs further permit joins over unions. As such, UCQs are analogous to a disjunctive normal form. Though any monotone query (under set semantics) can be rewritten to an equivalent UCQ, certain queries can be much more concisely expressed as MQs versus UCQs, or put another way, there exist MQs that are exponentially longer when rewritten as UCQs. For example, the first three queries of Fig. 1 are MQs, but only the third is a UCQ; if we use a similar pattern as the third query to go search back n generations, then we would require 2^n BGPs with n triple patterns each; if we rather use the most concise MQ, based on the second query, we would need $2n$ BGPs with one triple pattern each.

CPQs and UCPQs are closely related to the query fragments of *conjunctions of 2-way regular paths queries* (C2RPQs) and *unions of conjunctions of 2-way regular paths queries* (UC2RPQs), but additionally allow negated property sets and variables in the predicate position [34]. NMQs are semantically related to the fragment with BGPs, projection, UNION, AND, OPTIONAL and $\text{FILTER}_{\texttt{!bound}}$ as studied by Pérez et al. [44].

3.18. *Complexity*

We here consider four decision problems:

Query Evaluation Given a solution μ, a query Q and a graph G, is $\mu \in Q(G)$?
Query Containment Given two queries Q_1 and Q_2, does $Q_1 \sqsubseteq Q_2$ hold?
Query Equivalence Given two queries Q_1 and Q_2, does $Q_1 \equiv Q_2$ hold?
Query Congruence Given two queries Q_1 and Q_2, does $Q_1 \cong Q_2$ hold?

In Table 16, we summarise known complexity results for these four tasks considering both bag and set semantics along with a reference for the result. The results refer to combined complexity, where the size of the queries and data (in the case of EVALUATED) are included. The "Full" class refers to any SELECT query using any of the deterministic SPARQL features,[7] while BGP′, UBGP′, NGP′ and UNGP′ refer to BGPs, UBGPs, NGPs and UNGPs

[6]Here we use "monotone queries" to refer to a syntactic class of queries, per the work of Sagiv and Yannakakis [49], rather than a semantic class of queries [6]. All monotone queries are (semantically) monotonic [6], but there may be monotonic SPARQL queries that are not (syntactically) monotone.

[7]We assume that built-in and aggregation expressions can be evaluated using at most polynomial space, as is the case for SPARQL.

Table 16

Complexity of SPARQL tasks on core fragments (considering combined complexity for EVALUATION)

	EVALUATION	CONTAINMENT	EQUIVALENCE	CONGRUENCE
		Set semantics		
BGP′	PTIME [44]	PTIME [1]*	PTIME [1]*	GI-complete [1,12]*
UBGP′	PTIME [44]*	PTIME [1,49]*	PTIME [1,49]*	GI-hard, NP
CQ	NP-complete [44]	NP-complete [11]*	NP-complete [11]*	NP-complete
UCQ	NP-complete [44]	NP-complete [11]*	NP-complete [11]*	NP-complete
MQ	NP-complete	Π_2^P-complete [49]*	Π_2^P-complete [49]*	Π_2^P-complete
NMQ	PSPACE-complete [44]	Undecidable [60]*	Undecidable [60]*	Undecidable
NGP′	PTIME [34]	PSPACE-complete [34]	PSPACE	GI-hard, EXPSPACE
UNGP′	PTIME [34]	PSPACE-complete [34]	PSPACE	GI-hard, EXPSPACE
CPQ	NP-complete [34]	EXPSPACE-complete [34]	NP-hard, EXPSPACE	NP-hard, EXPSPACE
UCPQ	NP-complete [34]	EXPSPACE-complete [34]	NP-hard, EXPSPACE	NP-hard, EXPSPACE
MPQ	NP-hard [34]*	EXPSPACE-hard [34]*	Π_2^P-hard	Π_2^P-hard
NMPQ	PSPACE-hard	Undecidable	Undecidable	Undecidable
Full	PSPACE-hard	Undecidable	Undecidable	Undecidable
		Bag semantics		
BGP′	PTIME	PTIME [1]*	PTIME [1]*	GI-complete
CQ	NP-complete	NP-hard, *Decidability open*	GI-complete [12]*	GI-complete
UCQ	NP-complete	Undecidable [30]*	GI-complete [17]*	GI-complete
MQ	NP-complete	Undecidable	GI-hard	GI-hard
NMQ	PSPACE-complete	Undecidable	Undecidable [45]*	Undecidable
Full	PSPACE-hard	Undecidable	Undecidable	Undecidable

without blank nodes, respectively.[8] We do not present results for query classes allowing paths under bag semantics as we are not aware of work in this direction; lower bounds can of course be inferred from the analogous fragment without paths under bag semantics.

An asterisk implies that the result is not explicitly stated, but trivially follows from a result or technique used. These cases include analogous results for relational settings, upper-or-lower bounds from tasks with obvious reductions to or from the stated problem, etc. We may omit references in case a result directly follows from other results in the table. A less obvious case is that of CONGRUENCE, which has not been studied in detail. However, with the exception of queries without projection (nor blank nodes), the techniques used to prove equivalence apply analogously for CONGRUENCE, which is similar to resolving the problem of non-projected variables whose names may differ across the input queries without affecting the given relation. In the case of BGPs (without projection nor blank nodes), it is sufficient to find an isomorphism between the input queries; in fact, without projection, since the input graph is a *set* of triples,[9] BGPs cannot produce duplicates, and thus results for set and bag semantics coincide.

Some of the more notable results include:

- The decidability of CONTAINMENT of CQs under bag semantics is a long open problem [12].
- EQUIVALENCE (and CONGRUENCE) of CQs and UCQs are potentially easier under bag semantics (GI-complete) than under set semantics (NP-complete) as the problem under bag semantics relates to isomorphism, rather than homomorphic equivalence under set semantics.
- Although UCQ and MQ classes are semantically equivalent (each UCQ has an equivalent MQ and vice versa), under set semantics the problems of CONTAINMENT and EQUIVALENCE (and CONGRUENCE) are potentially harder for MQs than UCQs; this is because MQs are more concise.

[8]Blank nodes act like projection, where their complexity then follows that of *CQs and *CPQs.

[9]This is also known as bag–set semantics, where the data form a set of tuples, but the query is evaluated under bag semantics [12].

- While CONTAINMENT for NMQs is undecidable under set semantics (due to the undecidability of FOL satisfiability), the same problem for UCQs under bag semantics is already undecidable (it can be used to solve Hilbert's tenth problem).

These results – in particular those of CONGRUENCE – form an important background for this work.

4. Problem

With these definitions in hand, we now state the problem we wish to address: *given a query Q, we wish to compute a canonical form of the query* $\text{can}(Q)$ *such that* $\text{can}(Q) \cong Q$ *(sound), and for all queries such that* $Q' \cong Q$*, it holds that* $\text{can}(Q) = \text{can}(Q')$ *(complete)*. In other words, we aim to compute a syntactically canonical form for the class of queries congruent to Q where the canonical query is also in that class.

With this canonicalisation procedure, we can decide the congruence $Q \cong Q'$ by deciding the equality $\text{can}(Q) = \text{can}(Q')$. We can thus conclude from Table 16 that canonicalisation is not feasible for queries in NMQ as it could be used to solve an undecidable problem. Rather we aim to compute a sound and complete canonicalisation procedure for MQs (which can decide a Π_2^P−complete problem, per Table 16) under both bag and set semantics, and a sound procedure for the full language under any semantics. This means that for two queries Q and Q' that fall outside the MQ class, with a sound but incomplete canonicalisation procedure, $\text{can}(Q) = \text{can}(Q')$ implies $Q \cong Q'$, but $\text{can}(Q) \neq \text{can}(Q')$ does not necessarily imply $Q \ncong Q'$.

Indeed, even in the case of MQs, deciding $\text{can}(Q) = \text{can}(Q')$ is likely to be a rather inefficient way to decide $Q \cong Q'$. Our intended use-case is rather to partition a set of queries $\mathcal{Q} = \{Q_1, \dots, Q_n\}$ into the quotient set $\mathcal{Q}/_{\cong}$, i.e., to find all sets of congruent queries in $\mathcal{Q}$. This is useful, for example, in the context of caching applications where $\mathcal{Q}$ represents a log or stream of queries, where given Q_j, we wish to know if there exists a query Q_i $(i < j)$ that is congruent in order to reuse its results. Rather than applying pairwise congruence checks, we can canonicalise queries and use their canonical forms as keys for partitioning. While these pairwise checks do not affect the computational complexity, in practice most queries are small and relatively inexpensive to canonicalise, where the $O(|\mathcal{Q}|^2)$ cost of pairwise checks can dominate, particularly for a large set of queries $\mathcal{Q}$. We will later analyse this experimentally. As per the introduction, canonicalisation is also potentially of interest for analysing logs, optimising queries, and/or learning over queries.

5. Related works

In this section, we discuss implementations of systems relating to containment, equivalence and canonicalisation of SPARQL queries.

A number of systems have been proposed to decide the containment of SPARQL queries. Among these, Letelier et al. [36] propose a normal form for *quasi-well-designed pattern trees* – a fragment of SPARQL allowing restricted use of `OPTIONAL` over BGPs – and implement a system called SPARQL Algebra for deciding containment and equivalence in this fragment based on the aforementioned normal form. The problem of determining equivalence of SPARQL queries can also be addressed by reductions to related problems. Chekol et al. [14] have used a μ-calculus solver and an XPath-equivalence checker to implement SPARQL containment/equivalence checks. These works implement pairwise checks.

Some systems have proposed isomorphism-based indexing of sub-queries. In the context of a caching system, Papailiou et al. [41] apply a canonical labelling algorithm (specifically Bliss [31]) on BGPs in order to later find isomorphic BGPs with answers available; their approach further includes methods for generalising BGPs such that it is more likely that they will be reused later. More recently, Stadler et al. [57] propose a system called JSAG for solving the containment of SPARQL queries. The system computes normal forms for queries, before representing them as a graph and applying subgraph isomorphism algorithms to detect containments. Such approaches do not discuss completeness, and would appear to miss containments for CQs under set semantics (and distinct-bag semantics), which require checking for homomorphisms rather than (sub-graph) isomorphisms.

We remark that in the context of relational database systems, there are likewise few implementations of query containment, equivalence, etc., as also observed by Chu et al. [15,16], who propose two systems for deciding the equivalence of SQL queries. Their first system, called Cosette [16], translates SQL into logical formulae, where a constraint solver is used to try to find counterexamples for equivalence; if not found, a proof assistant is used to prove equivalence. Chu et al. [15] later proposed the UDP system, which expresses SQL queries – as well as primary and foreign key constraints – in terms of unbounded semiring expressions, thereafter using a proof assistant to test the equivalence of those expressions; this approach is sound and complete for testing the equivalence of UCQs under both set and bag semantics. Zhou et al. [65] recently propose the EQUITAS system, which converts SQL queries into FOL-based formulae, reducing the equivalence problem to a satisfiability-modulo-theories (SMT) problem, which allows for capturing complex selection criteria (inequalities, boolean expressions, cases, etc.). Aside from targeting SQL, a key difference with our approach is that such systems apply pairwise checks.

In summary, while problems relating to containment and equivalence have been well-studied in the theoretical literature, relatively few practical implementations have emerged, perhaps because of the high computational costs, and indeed the undecidability results for the full SPARQL/SQL language. Of those that have emerged, they either offer sound and complete checks in a pairwise manner for query fragments, such as UCQs (e.g., [15]), or they offer sound but incomplete canonicalisation focused on isomorphic equivalence (e.g., [41]). To the best of our knowledge, the approach that we propose here, which we call QCan, is the only one that allows for canonicalising queries with respect to congruence, and that is sound and complete for monotone queries under both set and bag semantics. Our experiments will show that despite high theoretical computational complexity, QCan can be deployed in practice to detect congruent equivalence classes in large-scale, real-world query logs or streams, which are dominated by relatively small and simple queries.

6. Canonicalisation of monotone queries

In this section, we will first describe the different steps of our proposed canonicalisation process for monotone queries (MQs), i.e., queries with basic graph patterns, joins, unions, outer projection and distinct (see Section 3.17). In fact, we consider a slightly larger set of queries that we call *extended monotone queries* (EMQs), which are monotone queries that additionally support property paths using the (non-recursive) features "/" (followed by), "^" (inverse) and "|" (disjunction); property paths using such queries can be rewritten to monotone queries. We will cover the (sound but incomplete) canonicalisation of other features of SPARQL 1.1 later in Section 7.

As mentioned in the introduction, the canonicalisation process consists of: algebraic rewriting of parts of the query into normal forms, the representation of the query as a graph, the minimisation of the monotonic parts of the query by leaning and containment checks, the canonical labelling of the graph, and finally the mapping back to query syntax. We describe these steps in turn and then conclude the section by proving that canonicalisation is sound and complete for EMQs.

6.1. *UCQ normalisation*

In this section we describe the rules used to rewrite EMQs into a normal form based on unions of conjunctive queries (UCQs). We first describe the steps we apply for rewriting property paths into monotone features (where possible), thus converting EMQs into MQs. We then describe the rewriting of MQs into UCQ normal form. We subsequently describe some postprocessing of variables to ensure that those with the same name are correlated and that variables that are always unbound are removed. Finally we extend the normal form to take into account set vs. bag semantics.

6.1.1. *Property path elimination*

Per Table 9, property paths that can be rewritten to joins and unions are considered to be equivalent to their rewritten form under both bag and set semantics. We make these equivalences explicit by rewriting such property paths to joins and unions; i.e.:

$$(o, \hat{}e, s) \Rightarrow (s, e, o)$$

$$(s, e_1/e_2, o) \Rightarrow (s, e_1, x), (x, e_2, o)$$

$$(s, e_1|e_2, o) \Rightarrow \big[(s, e_1, o)\text{UNION}(s, e_2, o)\big]$$

where x denotes a fresh blank node. The exact rewriting is provided in Table 9. These rewritings may be applied recursively, as needed. If the input query is an EMQ, then the output of the recursive rewriting will be an MQ, i.e., a query without property paths.

Example 6.1. Consider the following query based on Example 1.1 looking for names of aunts.

```
SELECT DISTINCT ?z WHERE {
  ?x ^(:mother|:father) ?w .
  ?x :sister/:name ?z .
}
```

This query will be rewritten to:

```
SELECT DISTINCT ?z WHERE {
  ?w (:mother|:father) ?x .
  ?x :sister _:y . _:y :name ?z .
}
```

And then recursively to:

```
SELECT DISTINCT ?z WHERE {
  { ?w :mother ?x . } UNION { ?w :father ?x . }
  ?x :sister _:y . _:y :name ?z .
}
```

In this case we succeed in removing all property paths; however, property paths with $*$ or $+$ cannot be rewritten to other query features in this way.

6.1.2. *Union normalisation*

Pérez et al. [44] establish that, under set semantics, joins and unions in SPARQL are commutative and associative, and that joins distribute over unions. We summarise these results in Table 17. Noting that under set semantics, the multiplicity of joins and unions is given by the multiplication and addition of natural numbers, respectively; that both multiplication and addition are commutative and associative; and that multiplication distributes over addition; the same results also apply under bag semantics.

Another (folklore) result of interest is that BGPs can be rewritten to equivalent joins of their triple patterns. However, care must be taken when considering blank nodes in BGPs; otherwise the same blank node in two different triple patterns might be matched to two different terms, breaking the equivalence. Along these lines, let $\eta : \mathbf{B} \to \mathbf{V}$ denote a one-to-one mapping of blank nodes to variables; we assume that η will rewrite blank nodes to fresh variables not appearing elsewhere in a query. Given a basic graph pattern $B = \{t_1, \ldots, t_n\}$ and a mapping η such

Table 17
Equivalences given by Pérez et al. [44] for set semantics

Join is commutative	$[Q_1 \text{AND} Q_2] \equiv [Q_2 \text{AND} Q_1]$
Union is commutative	$[Q_1 \text{UNION} Q_2] \equiv [Q_2 \text{UNION} Q_1]$
Join is associative	$[Q_1 \text{AND}[Q_2 \text{AND} Q_3]] \equiv [[Q_1 \text{AND} Q_2] \text{AND} Q_3]$
Union is associative	$[Q_1 \text{UNION}[Q_2 \text{UNION} Q_3]] \equiv [[Q_1 \text{UNION} Q_2] \text{UNION} Q_3]$
Join distributes over union	$[Q_1 \text{AND}[Q_2 \text{UNION} Q_3]] \equiv [[Q_1 \text{AND} Q_2] \text{UNION}[Q_1 \text{AND} Q_3]]$

that $\eta(B) = \{t'_1, \ldots, t'_n\}$ – where $t'_i = \eta(t_i)$ for $1 \leqslant i \leqslant n$ – the following holds:

$$B \equiv \text{SELECT}_{\text{vars } B}\big(\big[\{t'_1\}\text{AND}\big[\ldots\text{AND}\{t'_n\}\big]\big]\big)$$

i.e., a BGP B is equivalent to the result of rewriting its blank nodes to fresh variables, joining the individual triple patterns, and projecting only the variables originally in B. This equivalence again holds under bag semantics since the multiplicity of a solution $\mu \in B(D)$ under bag semantics is defined in SPARQL as the number of blank node mappings satisfying the solution μ.

These known results give rise to a UCQ normal form for MQs [44]. More specifically, given a pattern $[Q_1 \text{AND}[Q_2 \text{UNION} Q_3]]$, we can rewrite this to $[[Q_1 \text{AND} Q_2]\text{UNION}[Q_1 \text{AND} Q_3]]$; in other words, we translate joins of unions to (equivalent) patterns involving unions of joins. Also, as Schmidt et al. [53] observe, $[Q \text{AND} Q] \equiv Q$ and $[Q \text{UNION} Q] \equiv Q$ under set semantics. Hence we can abstract the commutativity and associativity of both joins and unions by introducing two new syntactic operators:

$$\text{AND}(Q_1, \ldots, Q_n) := \big[Q_1 \text{AND}[\ldots \text{AND} Q_n]\big]$$
$$\text{UNION}(Q_1, \ldots, Q_n) := \big[Q_1 \text{UNION}[\ldots \text{UNION} Q_n]\big]$$

Given the aforementioned equivalences, the arguments of $\text{AND}(\cdot)$ and $\text{UNION}(\cdot)$ can be considered as a set of operands under set semantics and a bag of operands under bag semantics (wherein duplicate operands may affect the multiplicities of results).

The UCQ normal form for MQs is then of the form $\text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))$, where each Q_i $(1 \leqslant i \leqslant n)$ is of the form $\text{AND}(\{t_{i,1}\}, \ldots, \{t_{i,m}\})$, where each $t_{i,j}$ $(1 \leqslant k \leqslant m)$ is a triple pattern. Given that duplicate triple patterns in a join do not affect the multiplicity of results, we can further remove these duplicates such that in our normal form, $t_{i,j} \neq t_{i,k}$ for $1 \leqslant j < k \leqslant m$. For this reason, in the case of UCQs, we can analogously consider each $Q_1, \ldots, Q_n$ to be a set of triple patterns without blank nodes.

Example 6.2. We show a case where the multiplicity of union operands changes the multiplicity of results under bag semantics. Consider the following MQ Q:

```
SELECT ?s ?o WHERE {
 { ?s :p ?o } UNION { ?s :p ?o }
 { ?s :p ?o } UNION { ?s :p ?o }
}
```

Assume a dataset D with a default graph $\{(:\text{s}, :\text{p}, :\text{o})\}$. Let $\mu = \{?\text{s}/:\text{s}, ?\text{o}/:\text{o}\}$. Note that $Q(D)(\mu) = 4$ since each union generates μ twice, where the multiplicity of the join is then the product of both. We can describe the multiplicity of μ as $(1 + 1)(1 + 1) = 4$.

If we rewrite this query to a UCQ, in the first step, pushing the first join inside the union, we generate:

```
SELECT ?s ?o WHERE {
 { ?s :p ?o  { ?s :p ?o } UNION { ?s :p ?o } }
   UNION
 { ?s :p ?o  { ?s :p ?o } UNION { ?s :p ?o } }
}
```

We may now describe the multiplicity of μ as $1(1 + 1) + 1(1 + 1) = 4$. In the next step, we have:

```
SELECT ?s ?o WHERE {
 { { ?s :p ?o . ?s :p ?o } UNION { ?s :p ?o . ?s :p ?o } }
   UNION
 { { ?s :p ?o . ?s :p ?o } UNION { ?s :p ?o . ?s :p ?o  } }
}
```

The multiplicity of this query is described as $(1 \cdot 1 + 1 \cdot 1) + (1 \cdot 1 + 1 \cdot 1) = 4$. Since BGPs are sets of triple patterns, we should remove the duplicates. Subsequently unnesting the unions, the query then becomes:

```
SELECT ?s ?o WHERE {
 { ?s :p ?o . }
   UNION { ?s :p ?o . }
   UNION { ?s :p ?o . }
   UNION { ?s :p ?o . }
}
```

The multiplicity is then $1 + 1 + 1 + 1 = 4$. In this case, the duplicate union operands are needed to preserve the original multiplicities of the query.

As was previously mentioned, the UCQ normal form may be exponentially larger than the original MQ; for example, a relatively concise EMQ of the form $\{(x, (p_1|q_1)/\ldots/(p_n|q_n), y)\}$ would be rewritten to an MQ with a join of n unions with two triple patterns each, and then to a UCQ with a union of 2^n BGPs (for each combination of p_i and q_i), with each BGP containing n triple patterns.

Example 6.3. Let us take the output query of Example 6.1 and apply the UCQ normal form.

```
SELECT DISTINCT ?z WHERE {
  { ?w :mother ?x . } UNION { ?w :father ?x . }
  ?x :sister _:y . _:y :name ?z .
}
```

Blank nodes are rewritten to variables, and then join is distributed over union, giving the following query:

```
SELECT DISTINCT ?z WHERE {
  { ?w :mother ?x . ?x :sister ?y . ?y :name ?z . }
  UNION
  { ?w :father ?x . ?x :sister ?y . ?y :name ?z . }
}
```

If we were to consider the names of aunts or uncles (`:sister|:brother`) then we would end up with four unions of BGPs with four triple patterns each. If we were to consider the names of children (`:son|:daughter`) of aunts or uncles, we would end up with eight unions of BGPs with five triple patterns each. In this way, the UCQ rewriting may result in a query that is exponentially larger than the input.

6.1.3. Unsatisfiability normalisation

We recall that a graph pattern Q is considered unsatisfiable if and only if there does not exist a dataset D such that $Q(D)$ is non-empty; i.e., the graph pattern never generates solutions. There is one trivial case of unsatisfiability for UCQs that must be taken into account: when subjects are literals. Specifically, SPARQL allows literal subjects even though they are disallowed in RDF graphs; this was to enable forwards-compatibility with a possible future generalisation of RDF to allow literal subjects, which has not happened as of RDF 1.1. As such, BGPs with any literal subject are unsatisfiable.

Lemma 6.1. *Let Q denote a BGP. Q is unsatisfiable if and only if it contains a literal subject.*

Please see Appendix A.1.1 for the proof.

Moving to UCQs, it is not difficult to see that a union is satisfiable if and only if one of its operands is satisfiable, or, equivalently, that it is unsatisfiable if and only if all of its operands are unsatisfiable.

Lemma 6.2. *Let $Q = \text{UNION}(Q_1, \ldots, Q_n)$. Q is unsatisfiable if and only if all of $Q_1, \ldots, Q_n$ are unsatisfiable. Further assume that Q_k is unsatisfiable and let $Q' = \text{UNION}(Q_1, \ldots, Q_{k-1}, Q_{k+1}, \ldots, Q_n)$ denote Q removing the operand Q_k. It holds that $Q \equiv Q'$.*

Please see Appendix A.1.2 for the proof.

To deal with CQs of the form $\text{SELECT}_V(Q)$, where Q is a BGP containing a triple pattern with a literal subject, we simply replace this CQ with an arbitrary but canonical unsatisfiable query $Q_\emptyset$; for example, $Q_\emptyset := \text{SELECT}_{\{\texttt{?u}\}}(\{(\texttt{"uns"}, \texttt{?u}, \texttt{?u})\})$. In the case of UCQs, we remove operand BGPs that are unsatisfiable; if all operands are unsatisfiable, we replace the entire UCQ with the canonical query $Q_\emptyset$. If $Q_\emptyset$ is produced, canonicalisation can stop.

Example 6.4. Take the CQ:

```
SELECT DISTINCT ?x WHERE { "x" :x ?x }
```

We replace this with the canonical query:

```
SELECT ?u WHERE { "uns" ?u ?u }
```

Next take the UCQ:

```
SELECT ?x WHERE { { "x" :x ?x } UNION { ?x :x "x" } }
```

We will rewrite this to

```
SELECT ?x WHERE { ?x :x "x" }
```

by removing the unsatisfiable operand.

6.1.4. *Variable normalisation*

The same variable may sometimes occur in multiple query scopes such that replacing an occurrence of the variable in one scope with a fresh variable does not change the query results. We say that such variable occurrences are not correlated. There is one case where this issue may arise in UCQs. We call a variable v a *union variable* if it occurs in a union $Q = \text{UNION}(Q_1, \ldots, Q_n)$ $(n > 1)$. An occurrence of v does not correlate with other occurrences of v in different operands of the same union unless v is correlated outside of Q in a query.[10] In the particular case of UCQs, occurrences of non-projected union variables in different operands of the union do not correlate.

Lemma 6.3. *Let $Q = \text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))$ denote a UCQ. Let $\lambda_1, \ldots, \lambda_n$ denote variable-to-variable mappings such that for $1 \leqslant i \leqslant n$, $\lambda_i : \mathbf{V} \to \mathbf{V}$ where $\text{dom}(\lambda_i) = \text{vars}\, Q_i \setminus V$, and, for all $v \in \text{dom}(\lambda_i)$, it holds that $\lambda_i(v) \notin \text{vars}\, Q$ and there does not exist λ_j $(1 \leqslant i < j \leqslant n)$ and $v' \in \text{dom}(\lambda_j)$ such that $\lambda_i(v) = \lambda_j(v')$. In other words, each variable-to-variable mapping rewrites each non-projected variable of each union operand to a fresh variable. Then the following equivalence holds:*

$$\begin{aligned}&\text{SELECT}_V\big(\text{UNION}(Q_1, \ldots, Q_n)\big)\\ &\quad\equiv \text{SELECT}_V\big(\text{UNION}\big(\lambda_1(Q_1), \ldots, \lambda_n(Q_n)\big)\big)\end{aligned}$$

Please see Appendix A.1.3 for the proof.

These non-correlated variables give rise to non-trivial equivalences based on the "false" correspondences between variables with the same name that have no effect on each other. We address such cases by differentiating union variables that appear in multiple operands of the union but not outside the union.

Example 6.5. We take the output of Example 6.3:

```
SELECT DISTINCT ?z WHERE {
  { ?w :mother ?x . ?x :sister ?y . ?y :name ?z . }
  UNION
  { ?w :father ?x . ?x :sister ?y . ?y :name ?z . }
}
```

The variable `?z` correlates across both operands because both occurrences correlate with the same external appearance of `?z` in the `SELECT` clause. Conversely, the variables `?w`, `?x` and `?y` do not correlate across both operands of the union as they do not correlate with external occurrences of the same variable. Hence we differentiate `?w`, `?x` and `?y` in both operands:

[10] If considering the direct results of a query *pattern*, then the naming of variables matters as they are bound in the solutions.

```
SELECT DISTINCT ?z WHERE {
  { ?w1 :mother ?x1 . ?x1 :sister ?y1 . ?y1 :name ?z . }
  UNION
  { ?w2 :father ?x2 . ?x2 :sister ?y2 . ?y2 :name ?z . }
}
```

The resulting query is equivalent to the original query, but avoids "false correspondences" of variables.

Next we apply a simple rule to remove variables that are always unbound in projections. Left unattended, such variables could otherwise constitute a trivial counterexample for the completeness of canonicalisation. We recall from Section 3.9 the notation $\text{pvars}(Q)$ to denote the possible variables of a graph pattern, i.e., the variables that are bound to some RDF term in some solution of Q over some dataset D.

Lemma 6.4. *Let Q be a graph pattern, let V' be a set of variables, and let V'' be a set of variables such that* $\text{pvars}(Q) \cap V'' = \emptyset$. *It holds that:*

$$\text{SELECT}_{V'}(Q) \equiv \text{SELECT}_{V' \cup V''}(Q)$$

Please see Appendix A.1.4 for the proof.

We deal with such cases by removing variables that are always unbound from the projection.[11]

Example 6.6. Take a query:

```
SELECT DISTINCT ?w ?z WHERE {
  ?w :mother ?m .
}
```

We can remove the variable `?z` without changing the semantics of the query as it will always be unbound, no matter what dataset is considered. In practice engines may return solution tables with blank columns for variables like `?z`, but our definitions do not allow such columns (such columns can easily be added in practice if required).

6.1.5. Set vs. bag normalisation

The presence or absence of `DISTINCT` (or `REDUCED`) in certain queries does not affect the solutions that are generated because no duplicates can occur. In the case of UCQs, this can occur under two specific conditions. The first such case involves CQs.

Lemma 6.5. *Let Q denote a satisfiable BGP. It holds that:*

$$\text{DISTINCT}\big(\text{SELECT}_V(Q)\big) \equiv \text{SELECT}_V(Q).$$

if and only if $\text{vars}\, Q \subseteq V$ *and* $\text{bnodes}(Q) = \emptyset$.

Please see Appendix A.1.5 for the proof.

The second case involves unions.

Lemma 6.6. *Let $Q_1, \ldots, Q_n$ denote satisfiable BGPs and let $Q = Q_1 \cup \cdots \cup Q_n$ denote the set union of their triple patterns. It holds that:*

$$\text{DISTINCT}\big(\text{SELECT}_V\big(\text{UNION}(Q_1, \ldots, Q_n)\big)\big)$$
$$\equiv \text{SELECT}_V\big(\text{UNION}(Q_1, \ldots, Q_n)\big)$$

if and only if $\text{vars}\, Q \subseteq V$, $\text{bnodes}(Q) = \emptyset$ *and* $\text{vars}\, Q_i \neq \text{vars}\, Q_j$ *for all* $1 \leqslant i < j \leqslant n$.

[11] In concrete SPARQL syntax, a `SELECT` query must specify either `*` or at least one variable. Allowing an empty projection in the abstract syntax avoids having to deal explicitly with the empty projection case, simplifying matter. When mapping from abstract syntax to concrete syntax we can simply add a fresh canonical variable to the `SELECT` clause in order to represent empty projections.

Please see Appendix A.1.6 for the proof.

The same equivalences trivially hold for REDUCED, which becomes deterministic when no duplicate solutions are returned. We deal with all such equivalences by simply adding DISTINCT in such cases (or replacing REDUCED with DISTINCT).[12]

Example 6.7. Take a query such as:

```
SELECT ?w ?x ?y ?z WHERE {
  ?w :mother ?x . ?x :sister ?y . ?y :name ?z .
}
```

Since the query is a BGP with all variables projected and no blank nodes, no duplicates can be produced, and thus we can add a DISTINCT keyword to ensure that canonicalisation will detect equivalent or congruent queries irrespective of the inclusion or exclusion of DISTINCT or REDUCED in such queries. If the query were to not project a single variable, such as ?z, then duplicates become possible and adding DISTINCT would change the semantics of the query.

An example of a case involving union is as follows:

```
SELECT ?w ?x ?y ?z ?n WHERE {
  { ?w :parent ?x  . ?x :name ?n . }
  UNION { ?w :father ?y . ?y :name ?n . }
  UNION { ?w :mother ?z . ?z :name ?n . }
}
```

First we note that the individual basic graph patterns forming the operands of the UNION do not contain blank nodes and have all of their variables projected; hence they cannot lead to duplicates by themselves. Regarding the union, the set of variables is different in each operand, and hence no duplicates can be given: the first operand will (always and only) produce unbounds for ?y, ?z in its solutions; the second will produce unbounds for ?x, ?z; and the third will produce unbounds for ?x, ?y. Hence no operand can possibly duplicate a solution from another operand. Since the query cannot produce duplicates, we can add DISTINCT without changing its semantics. If we were instead to project {?w, ?n}, then adding DISTINCT would change the semantics of the query as the three operands may produce the same solution, and individual BGPs may duplicate solutions.

6.1.6. *Summary*

Given an EMQ Q, we denote by $\mathrm{U}(Q)$ the process described herein involving the application of:

1. property path elimination (§6.1.1);
2. union normalisation (§6.1.2);
3. unsatisfiability normalisation (§6.1.3);
4. variable normalisation (§6.1.4);
5. set vs. bag normalisation (§6.1.5).

6.2. *Graph representation*

Given an EMQ as input, the previous steps either terminate with a canonical unsatisfiable query, or provide us with a satisfiable query in UCQ normal form, with blank nodes replaced by fresh variables, non-correlated variables differentiated, variables that are always unbound removed from the projection (while ensuring that the projection is non-empty), and the DISTINCT keyword invoked in cases where duplicate solutions can never be returned. Before continuing, we first review an example that illustrates the remaining syntactic variations and redundancies in congruent UCQs that are left to be canonicalised.

[12]The choice to add rather than remove DISTINCT is for convenience: it allows us to later keep track of queries that can be normalised under set semantics. However, if performance were a focus, removing DISTINCT might lead to slightly more efficient queries where the planner will not invoke unnecessary deduplication.

Example 6.8. Consider the following UCQs:

```
SELECT DISTINCT ?n WHERE {
 { ?w :mother ?x . ?x :sister ?y , ?z . ?y :name ?n }
  UNION
 { ?a :father ?b . ?b :sister ?c . ?c :name ?n . ?d ?e ?n }
}
```

```
SELECT DISTINCT ?z WHERE {
 { ?a :name ?z . ?b :sister ?a . ?c :father ?b . }
  UNION
 { ?d :name ?z . ?e :sister ?d . ?f :mother ?e . }
  UNION
 { ?g :name ?z . ?h :sister ?g . :Jo :mother ?h . }
}
```

These queries are congruent, but differ in:

1. the ordering of triple patterns within BGPs;
2. the ordering of BGPs within the UCQ;
3. the naming of variables;
4. a redundant triple pattern in each BGP of the first query (those containing `?z` and `?d`, `?e`);
5. the redundant third BGP in the second query.

We are left to canonicalise such variations.

Our overall approach to address such variations is to encode queries as RDF graphs that we call *representational graphs* (r-graphs). This representation will allow for identifying and removing redundancies, and for canonically labelling variables such that elements of the query can be ordered deterministically.

We first establish some notation. Let $\lambda()$ denote a function that returns a fresh blank node, and $\lambda(x)$ denote a function that returns a fresh blank node unique to x. Let $\iota(\cdot)$ denote an id function such that:

– if $x \in \mathbf{IL}$, then $\iota(x) = x$;
– if $x \in \mathbf{VB}$, then $\iota(x) = \lambda(x)$;
– if x is a natural number then $\iota(x) = \texttt{"}x\texttt{"\^\^xsd:integer}$;
– if x is a boolean value then $\iota(x) = \texttt{"}x\texttt{"\^\^xsd:boolean}$;
– otherwise $\iota(x) = \lambda()$.

We assume that natural numbers and boolean values produce datatype literals in canonical form (for example, we assume that $\iota(2) =$ `"2"^^xsd:integer` rather than, say, `"+02"^^xsd:integer`).

Table 18 then provides formal definitions for transforming a UCQ Q in abstract syntax into its r-graph $\mathrm{R}(Q)$; we assume that a BGP is expressed as a join of its triple patterns. Note that for brevity, when we write $\iota(\cdot)$, we assume that the same blank node is used for the current expression as was assigned in the parent expression. The result is then deterministic modulo isomorphism. In order to capture the intuition, we provide a more visual depiction of

Table 18

Definitions for representational graphs $\mathrm{R}(Q)$ of a UCQ Q, where "a" abbreviates `rdf:type`, B is a basic graph pattern, $Q_1, \dots, Q_n$, Q' are graph patterns, V is a set of variables, and (s, p, o) is a triple pattern

$\cdot$	$\mathrm{R}(\cdot)$
$\mathrm{AND}(Q_1, \dots, Q_n)$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q_1)), \dots, (\iota(\cdot), \texttt{:arg}, \iota(Q_n)), (\iota(\cdot), \texttt{a}, \texttt{:And})\} \cup \mathrm{R}(Q_1) \cup \dots \cup \mathrm{R}(Q_n)$
$\mathrm{UNION}(Q_1, \dots, Q_n)$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q_1)), \dots, (\iota(\cdot), \texttt{:arg}, \iota(Q_n)), (\iota(\cdot), \texttt{a}, \texttt{:Union})\} \cup \mathrm{R}(Q_1) \cup \dots \cup \mathrm{R}(Q_n)$
$\mathrm{SELECT}_V(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{a}, \texttt{:Select})\} \cup \bigcup_{v \in V} \{(\iota(\cdot), \texttt{:var}, \iota(v))\} \cup \mathrm{R}(Q')$
$\mathrm{DISTINCT}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{a}, \texttt{:Distinct})\} \cup \mathrm{R}(Q')$
(s, p, o)	$\{(\iota(\cdot), \texttt{:s}, \iota(s)), (\iota(\cdot), \texttt{:p}, \iota(p)), (\iota(\cdot), \texttt{:o}, \iota(o)), (\iota(\cdot), \texttt{a}, \texttt{:TP})\}$

this transformation of UCQs in Table 19, where dashed nodes of the form x are replaced with $\iota(x)$, and the graph extended with $\mathrm{R}(x)$. We further provide the following example.

Example 6.9. We present an example of a UCQ and its r-graph in Fig. 3. For clarity (in particular, to avoid non-planarity), we embed the types of nodes into the nodes themselves; e.g., the lowermost node expands to `_:u1` `rdf:type` `:Union`. Given an input query Q_1' that varies from Q_1 in the naming of variables, applying the same process, the r-graph for Q_1 and Q_1' would be isomorphic, varying only in blank node labels.

Part of the benefit of this graph representation is that it abstracts away the ordering of the operands of query operators where such order does not affect the semantics of the operator. This representation further allows us to leverage existing tools to eliminate redundancy and later canonically label variables.

6.3. *Minimisation*

The minimisation step removes two types of redundancies: redundant triple patterns in BGPs, and redundant BGPs in unions. It is important to note that such redundancies only apply in the case of set semantics [49]; under bag semantics, these "redundancies" affect the multiplicity of results, and thus cannot be removed without changing the query's semantics.

6.3.1. *BGP minimisation*

The first type of redundancy we consider stems from redundant triple patterns. Consider a BGP Q (without blank nodes, for simplicity). We denote by $\rho : \mathbf{V} \to \mathbf{VIBL}$ a partial mapping from variables to variables and RDF terms, whose domain is denoted by $\mathrm{dom}(\rho)$. Now, for a given mapping ρ such that $\mathrm{dom}(\rho) = \mathrm{vars}\, Q$, it holds that $\rho(Q) \equiv \text{DISTINCT}(\text{SELECT}_{\mathrm{vars}\,\rho(Q)}(Q))$ if and only if $\rho(Q) \subseteq Q$. This is due to a classical result by Chandra and Merlin [11], where ρ is a homomorphism of Q onto itself: Q and $\rho(Q)$ are *homomorphically equivalent*.

One may note a correspondence to RDF entailment (see Section 2.5), which is also based on homomorphisms, where the core of an RDF graph represents a redundancy-free (lean) version of the graph. We can exploit this correspondence to remove redundancies in BGPs by computing their core. However, care must be taken to ensure that we do not remove variables from the BGP that are projected; we achieve this by temporarily replacing them with IRIs so that they cannot be eliminated during the core computation.

Example 6.10. Consider the following query, Q:

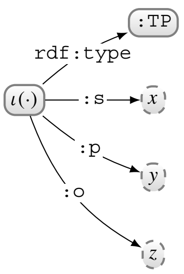
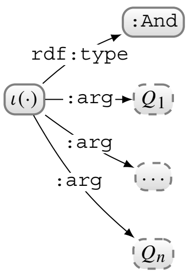
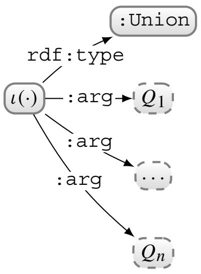
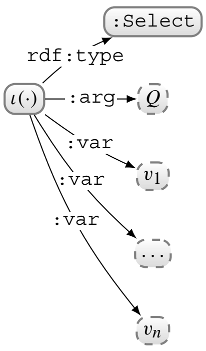
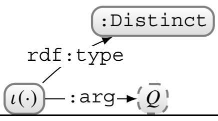
```
SELECT DISTINCT ?z WHERE {
  { :Jo :mother ?x . }
  UNION { ?w :father ?x. ?x :sister ?y . }
  UNION { ?c :mother ?d . ?d :sister ?y . }
  ?d ?p ?e . ?e :name ?f . ?x :sister ?y . ?y :name ?z .
}
```

Though perhaps not immediately obvious, this query is congruent with the three queries of Example 1.1. After applying UCQ normal forms and creating the base r-graph for Q, we end up with an r-graph analogous to the following query with a union of three BGPs:

```
SELECT DISTINCT ?z WHERE {
  { :Jo :mother ?x1 . ?d1 ?p1 ?e1 . ?e1 :name ?f1 .
         ?x1 :sister ?y1 .  ?y1 :name ?z .  }
  UNION { ?w2 :father ?x2 . ?x2 :sister ?y2 .
         ?d2 ?p2 ?e2 . ?e2 :name ?f2 .
         ?x2 :sister ?y2 . ?y2 :name ?z .  }
  UNION { ?c3 :mother ?d3 . ?d3 :sister ?y3 .
         ?d3 ?p3 ?e3 . ?e3 :name ?f3 .
         ?x3 :sister ?y3 . ?y3 :name ?z .  }
}
```

We then replace the blank node for the projected variable `?z` with a fresh IRI, and compute the core of the sub-graph for each BGP (the graph induced by the BGP node with type `:And` and any node reachable from that node

Table 19
Mapping UCQs to r-graphs

SPARQL	↦ **Representational graph**
x for $x \in \mathbf{IL}$	↦ x
x for $x \in \mathbf{B}$	↦ `_:b`x
x for $x \in \mathbf{V}$	↦ `_:v`x
(x, y, z)	↦ 
AND$(Q_1, \ldots, Q_n)$	↦ 
UNION$(Q_1, \ldots, Q_n)$	↦ 
SELECT$_{\{v_1,\ldots,v_n\}}(Q)$	↦  :Select rdf:type $\iota(\cdot)$:arg Q :var v_1 :var … :var v_n
DISTINCT(Q)	↦ 

```
SELECT DISTINCT ?z WHERE {
  { ?w1 :mother ?x1 . ?x1 :sister ?y1 . ?y1 :name ?z . }
  UNION
  { ?w2 :father ?x2 . ?x2 :sister ?y2 . ?y2 :name ?z . }
}
```

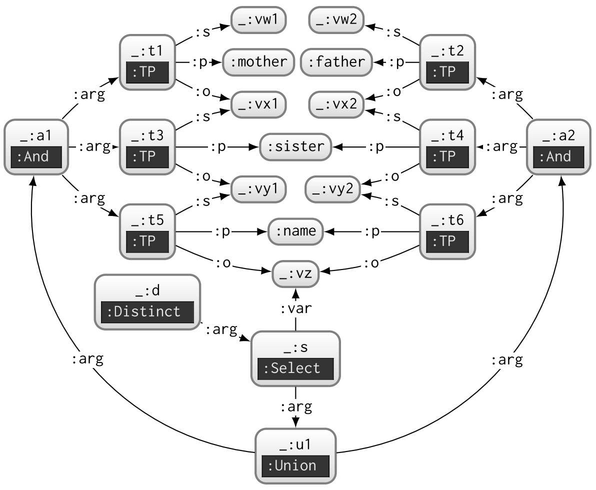


Fig. 3. UCQ (above) and its r-graph (below).

```
  { ?c3 :mother ?d3 . ?d3 :sister ?y3 .
    ?d3 ?p3 ?e3 . ?e3 :name ?f3 .
    ?x3 :sister ?y3 . ?y3 :name ?z . }
```

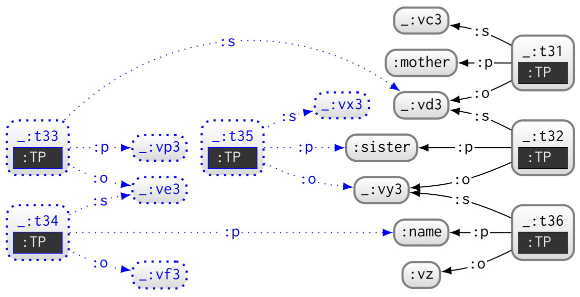


Fig. 4. BGP (above) and its r-graph (below) with the sub-graph removed during the core computation shown dashed (and in blue).

in the directed r-graph). Figure 4 depicts the sub-r-graph representing the third BGP (omitting the `:And` -typed node for clarity: it will not affect the core). Dashed nodes and edges are removed from the core per the blank node mapping:

$$\{\texttt{\_:vx3}/\texttt{\_:vd3}, \texttt{\_:t35}/\texttt{\_:t32}, \ \texttt{\_:t33}/\texttt{\_:t32}, \texttt{\_:vp3}/\texttt{:sister},$$
$$\texttt{\_:ve3}/\texttt{\_:vy3}, \texttt{\_:t34}/\texttt{\_:t36}, \texttt{\_:vf3}/\texttt{:vz}, \ldots\}$$

with the other nodes mapped to themselves. Observe that the projected variable `:vz` is now an IRI, and hence it cannot be removed from the graph.

If we consider applying this core computation over all three conjunctive queries, we would end up with an r-graph corresponding to the following query:

```
SELECT DISTINCT ?z WHERE {
  { :Jo :mother ?x1 . ?x1 :sister ?y1 . ?y1 :name ?z . }
  UNION
  { ?w2 :father ?x2 . ?x2 :sister ?y2 . ?y2 :name ?z . }
  UNION
  { ?c3 :mother ?d3 . ?d3 :sister ?y3 . ?y3 :name ?z . }
}
```

We see that the projected variable is preserved in all BGPs. However, we can still see (inter-BGP) redundancy with respect to the first and third BGPs (the first is contained in the third), which we address now.

6.3.2. *Union minimisation*

After removing redundancy from the individual BGPs, we may still be left with a union containing redundant BGPs as highlighted by the output of Example 6.10, where the first BGP is contained in the third BGP: when we take the union, the names of `:Jo`'s aunts returned by the first BGP will already be contained in the third, and since we apply distinct/set semantics, the duplicates (if any) will be removed. Hence we must now apply a higher-level normalisation of unions of BGPs in order to remove such redundancy. Specifically, we must take into consideration the following equivalence [49]; let $Q := \text{UNION}(Q_1, \ldots, Q_n)$ and $Q' := \text{UNION}(Q_1, \ldots, Q_{k-1}, Q_{k+1}, \ldots Q_n)$; then:

$$\text{DISTINCT}\big(\text{SELECT}_V(Q)\big) \equiv \text{DISTINCT}\big(\text{SELECT}_V(Q')\big)$$

if and only if $\text{SELECT}_V(Q_k) \sqsubseteq \text{SELECT}_V(Q_j)$ for $i \leqslant j \leqslant n, i \leqslant k \leqslant n, j \neq k$. [13] To resolve such equivalences, we remove from Q:

1. all Q_k $(1 \leqslant k \leqslant n)$ such that there exists Q_j $(1 \leqslant j < k \leqslant n)$ such that $\text{SELECT}_V(Q_j) \equiv \text{SELECT}_V(Q_k)$; *and*
2. all Q_k $(1 \leqslant k \leqslant n)$ where there exists Q_j $(1 \leqslant j \leqslant n)$ such that $\text{SELECT}_V(Q_j) \sqsubset \text{SELECT}_V(Q_k)$ (and $\text{SELECT}_V(Q_j) \not\equiv \text{SELECT}_V(Q_k)$);

i.e., we first remove all but one BGP from each group of equivalent BGPs and then remove all BGPs that are properly contained in another (considering in both cases the projected variables V).

To implement condition (1), let us first assume that all BGPs contain all projected variables. Note that in the previous step we have removed all redundancy from the CQs and hence it is sufficient to check for isomorphism between them; we can thus take the current r-graph G_j for each Q_j and apply iso-canonicalisation of G_j, removing any other Q_k $(k > j)$ whose G_k is isomorphic. Thereafter, to implement step (2), we can check the simple entailment $G_k \models G_j$ $(j \neq k)$, where if such an entailment holds, we can remove G_k (and thus Q_k); more specifically, we can implement this entailment check using a boolean SPARQL ASK query encoding G_j and evaluated over G_k (which will return true if the entailment holds). Note that in both processes, we should preserve projected variables in V, meaning they should only be mapped to each other; to ensure this, we can simply maintain the unique IRIs created for them earlier.

Per this process, the first BGP in the output of Example 6.10 is removed as it is contained in the third BGP, with the projected variable corresponding in both. We now take another example.

Example 6.11. Consider the following UCQ, where each BGP contains the lone projected variable:

[13] One may observe that this relates to the aforementioned rule for unsatisfiability in the case of UCQs; however, while the unsatisfiability rule applies in the case of both bag and set semantics, this rule only applies in the case of set semantics.

```
SELECT DISTINCT ?n WHERE {
  { ?m1 :cousin ?n . } UNION { ?n :cousin ?m2 . }
  UNION { ?n :cousin ?x3 . } UNION { ?x4 ?y4 ?n . }
  UNION { ?w5 ?x5 ?n . ?n ?y5 ?z5 . }
  UNION { ?x6 :name ?n . }
}
```

If we consider the first two BGPs, they do not contribute the same results to ?n; however, had we left the blank node _:vn to represent ?n, their r-graphs would be isomorphic whereas temporarily grounding :vn ensures they are no longer isomorphic. On the other hand, the r-graphs of the second and third BGP will remain isomorphic and thus one will be removed (for the purposes of the example, let's arbitrarily say the third is removed). There are no further isomorphic CQs and thus we proceed to containment checks.

The fourth BGP maps to (i.e., contains) the first BGP, and thus the first BGP will be removed. This containment check is implemented by creating the following ASK query from the r-graph for the fourth BGP:

```
ASK WHERE {
  _:and4 a :And ; :arg _:tp41 .
  _:tp41 a :TP ; :s _:x4 ; :p _:y4 ; :o :vn .
}
```

and applying it to the r-graph of the first BGP:

```
_:and1 a :And ; :arg _:tp11 .
_:tp11 a :TP ; :s _:m1 ; :p :cousin ; :o :vn .
```

This returns true and hence the first BGP is removed. Likewise the fourth BGP maps to the fifth BGP and also the sixth BGP and hence the fifth and sixth BGPs will also be removed. This leaves us with an r-graph representing the following UCQ query:

```
SELECT DISTINCT ?n WHERE {
  { ?n :cousin ?m2 . } UNION { ?x4 ?y4 ?n . }
}
```

This UCQ query is redundancy-free.

Now we drop the assumption that all CQs contain the same projected variables in V, meaning that we can generate unbounds. To resolve such cases, we can partition the BGP operands $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ of the union into sets of queries $\{\mathcal{Q}_1, \ldots, \mathcal{Q}_m\}$ based on the projected variables they contain. More formally, given two graph patterns Q_1 and Q_2, let $Q_1 \sim_V Q_2$ denote the equivalence relation such that $\mathrm{pvars}(Q_1) \cap V = \mathrm{pvars}(Q_2) \cap V$. Then $\{\mathcal{Q}_1, \ldots, \mathcal{Q}_m\}$ is the quotient set of $\mathcal{Q}$ by $\sim_V$. We can then apply the same procedure as described previously, checking equivalence and containment within each such set $\mathcal{Q}_1, \ldots, \mathcal{Q}_m$.

Example 6.12. Take the following UCQ, where the BGPs now contain different projected variables:

```
SELECT DISTINCT ?v ?w WHERE {
  { ?v :cousin ?w . } UNION { ?w :cousin ?v . }
  UNION { ?v :cousin ?x3 . } UNION { ?v :cousin ?y4 . }
  UNION { :a :b :c . } UNION { ?x6 ?y6 ?z6 . }
}
```

Let $\{Q_1, \ldots, Q_6\}$ denote the six BGPs, respectively. Further let $V = \{\texttt{?v}, \texttt{?w}\}$ denote the set of projected variables. Partitioning the BGPs by the projected variables, we end up with three sets of BGPS: $\{\{Q_1, Q_2\}, \{Q_3, Q_4\}, \{Q_5, Q_6\}\}$ given by $\{\texttt{?v}, \texttt{?w}\}$, $\{\texttt{?v}\}$ and $\{\}$, respectively. Within each group we apply the previous conditions. Thus, for example, we do not remove Q_1 even though it would be naively contained in, for example, Q_3 (where ?x3 in Q_3 would map to the IRI :vw in Q_1). Rather, Q_1, Q_2, Q_3 (or Q_4), and Q_6 would be maintained, resulting in the query:

```
SELECT DISTINCT ?v ?w WHERE {
  { ?v :cousin ?w . } UNION { ?w :cousin ?v . }
  UNION { ?v :cousin ?x3 . }  UNION { ?x6 ?y6 ?z6 . }
}
```

The first two BGPs can return multiple solutions, where none can have an unbound; the third BGP will return the same solutions for `?v` as the first CQ but `?w` will be unbound each time; the fourth CQ will return a single tuple with an unbound for `?v` and `?w` if and only if the RDF graph is not empty.

The result of this process will be an r-graph for a redundancy-free UCQ. On this r-graph, we apply some minor post-processing: (i) we replace the temporary IRIs for projected variables with their original blank nodes to allow for canonical labelling in a subsequent phase; and (2) we remove unary AND or UNION operators from the r-graph, reconnecting child and parent.

6.3.3. Summary

Given a UCQ Q being evaluated under set semantics (with distinct), we denote by $\text{M}(Q)$ the result of minimising the UCQ, involving the two procedures:

1. BGP minimisation (§6.3.1);
2. union minimisation (§6.3.2).

Given a UCQ Q being evaluated under bag semantics (without distinct), we define that $\text{M}(Q) = Q$. If bag semantics is selected, the UCQ can only contain a syntactic form of redundancy: exact duplicate triple patterns in the same BGP, which are implicitly removed since we model BGPs as sets of triple patterns. Any other form of redundancy mentioned previously – be it within or across BGPs – will affect the multiplicity of results [12]. Hence if bag semantics is selected, we do not apply any redundancy elimination other than removing duplicate triple patterns in BGPs.

6.4. Canonical labelling

The second-last step of the canonicalisation process consists of applying a canonical labelling to the blank nodes of the RDF graph output from the previous process [28]. Specifically, given an RDF graph G, we apply a canonical labelling function $\text{L}(\cdot)$ such that $\text{L}(G) \simeq G$ and for all RDF graphs G', it holds that $G \simeq G'$ if and only if $\text{L}(G) = \text{L}(G')$; in other words, $\text{L}(\cdot)$ bijectively relabels the blank nodes of G in a manner that is deterministic modulo isomorphism, meaning that any isomorphic graph will be assigned the same labels. This is used to assign a deterministic labelling of query variables represented in the r-graph as blank nodes; other blank nodes presenting query operators will also be labelled as part of the process but their canonical labels are not used.

6.5. Inverse mapping

The final step of the canonicalisation process is to map from the canonically labelled r-graph to query syntax. More specifically, we define an inverse r-mapping, denoted $\text{R}^{-}(G)$, to be a partial mapping from RDF graphs to query expressions such that $\text{R}^{-}(\text{R}(Q)) = Q$; i.e. converting Q to its r-graph and then applying the inverse r-mapping yields the query Q again.[14] We can achieve this by applying the inverse of Table 18, where canonical blank nodes in RDF term or variable positions (specifically, the objects of triples in the r-graph with predicate `:s`, `:p`, `:o`, or `:el`) are mapped to canonical variables or blank nodes using a fixed, total, one-to-one mapping $\xi : \mathbf{B} \to \mathbf{VB}$ [50].[15]

To arrive at a canonical concrete syntax, we order the operands of commutative operators using a syntactic ordering on the canonicalised elements, and then serialise these operands in their lexicographical order. This then concludes the canonicalisation of EMQs.

[14] Here we assume the use of $\text{UNION}(\cdot)$, etc., to abstract away the ordering of operands of commutative operators.

[15] Inverting Table 19, we can define ξ as $\xi(\texttt{\_:vx}) = \texttt{?x}$, $\xi(\texttt{\_:bx}) = \texttt{\_:x}$, and so forth for all blank nodes and variables. For the case of EMQs, all blank nodes are mapped to variables since blank nodes in BGPs were replaced earlier by variables.

6.6. *Soundness and completeness*

Given an EMQ as input, we prove soundness – i.e., that the output query is congruent to the input query – and completeness – i.e., that the output for two input queries is the same if and only if the input queries are congruent – for the proposed canonicalisation scheme.

6.6.1. *Soundness*

We begin the proof of soundness by showing that the UCQ normalisation preserves congruence.

Lemma 6.7. *For an EMQ Q, it holds that:*

$$\mathrm{U}(Q) \cong Q.$$

Please see Appendix A.1.7 for the proof.

Next we prove that the canonical labelling of blank nodes in the r-graph does not affect the properties of the inverse r-mapping.

Lemma 6.8. *Given a UCQ Q, it holds that:*

$$\mathrm{R}^{-}\big(\mathrm{L}\big(\mathrm{R}(Q)\big)\big) \cong Q.$$

Please see Appendix A.1.8 for the proof.

Finally we prove that the minimisation of UCQs through their r-graphs preserves congruence.

Lemma 6.9. *Given a UCQ Q, it holds that:*

$$\mathrm{R}^{-}\big(\mathrm{M}\big(\mathrm{R}(Q)\big)\big) \cong Q.$$

Please see Appendix A.1.9 for the proof.

The following theorem then establishes soundness; i.e., that the proposed canonicalisation procedure preserves congruence of EMQs.

Theorem 6.1. *For an EMQ Q, it holds that:*

$$\mathrm{R}^{-}\big(\mathrm{L}\big(\mathrm{M}\big(\mathrm{R}\big(\mathrm{U}(Q)\big)\big)\big)\big) \cong Q.$$

Please see Appendix A.1.10 for the proof.

6.6.2. *Completeness*

We now establish completeness: that for any two EMQs, they are congruent if and only if their canonicalised queries are equal. We will prove this by proving lemmas for various cases.

We begin by stating the following remark, which will help us to abbreviate some proofs.

Remark 6.1. The following hold:

1. if $Q_1 \simeq Q_2$, then $\mathrm{U}(Q_1) \simeq \mathrm{U}(Q_2)$.
2. if $\mathrm{U}(Q_1) \simeq \mathrm{U}(Q_2)$, then $\mathrm{R}(\mathrm{U}(Q_1)) \simeq \mathrm{R}(\mathrm{U}(Q_2))$;
3. if $\mathrm{R}(\mathrm{U}(Q_1)) \simeq \mathrm{R}(\mathrm{U}(Q_2))$, then $\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_1))) \simeq \mathrm{M}(\mathrm{R}(\mathrm{U}(Q_2)))$;
4. if $\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_1))) \simeq \mathrm{M}(\mathrm{R}(\mathrm{U}(Q_2)))$, then $\mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_1)))) = \mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_2))))$;
5. if $\mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_1)))) = \mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_2))))$, then $\mathrm{R}^{-}(\mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_1))))) = \mathrm{R}^{-}(\mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_2)))))$.

Thus, if any premise 1–5 is satisfied, it holds that $\mathrm{R}^{-}(\mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_1))))) = \mathrm{R}^{-}(\mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(Q_2)))))$.

In order to prove the result for various cases, our goal is thus to prove isomorphism of the input queries, the queries in UCQ normal form, the r-graphs of the queries, or the minimised r-graphs.

Our first lemma deals with unsatisfiable UCQs, which is a corner-case specific to SPARQL.

Lemma 6.10. *Let Q_1 and Q_2 denote UCQs. If Q_1 and Q_2 are unsatisfiable (which implies $Q_1 \cong Q_2$), then:*

$$\textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_1)\big)\big)\big)\big) = \textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_2)\big)\big)\big)\big).$$

Please see Appendix A.1.11 for the proof.

In practice, if a UCQ Q is unsatisfiable, then the canonicalisation process can stop after $\textsc{u}(Q)$ yields $Q_\emptyset$. We state the result in this way to align the process for both satisfiable and unsatisfiable cases. We can now focus on cases where both queries are satisfiable.

We will start with satisfiable CQs evaluated under set semantics (with distinct).

Lemma 6.11. *Let Q_1 and Q_2 denote satisfiable BGPs and V_1 and V_2 sets of variables. Further let $Q_1' =$ $\textsc{distinct}(\textsc{select}_{V_1}(Q_1))$ and likewise let $Q_2' = \textsc{distinct}(\textsc{select}_{V_2}(Q_2))$. If $Q_1' \cong Q_2'$ then*

$$\textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_1')\big)\big)\big)\big) = \textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_2')\big)\big)\big)\big).$$

Please see Appendix A.1.12 for the proof.

We move to CQs evaluated under bag semantics (without distinct; the result also considers cases where the CQ cannot return duplicates).

Lemma 6.12. *Let Q_1 and Q_2 denote satisfiable BGPs and V_1 and V_2 sets of variables. Further let $Q_1' =$ $\textsc{select}_{V_1}(Q_1)$ and $Q_2' = \textsc{select}_{V_2}(Q_2)$. If $Q_1' \cong Q_2'$ then*

$$\textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_1')\big)\big)\big)\big) = \textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_2')\big)\big)\big)\big).$$

Please see Appendix A.1.13 for the proof.

We now move to UCQs evaluated under set semantics (with distinct).

Lemma 6.13. *Let Q_1 and Q_2 denote satisfiable UCQs with distinct. If $Q_1 \cong Q_2$ then*

$$\textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_1)\big)\big)\big)\big) = \textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_2)\big)\big)\big)\big).$$

Please see Appendix A.1.14 for the proof.

We next consider UCQs under bag semantics (without distinct; again, this also holds in the case that the UCQs cannot return duplicates).

Lemma 6.14. *Let Q_1 and Q_2 denote satisfiable UCQs without distinct. If $Q_1 \cong Q_2$ then*

$$\textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_1)\big)\big)\big)\big) = \textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q_2)\big)\big)\big)\big).$$

Please see Appendix A.1.15 for the proof.

Finally we consider what happens when one (U)CQ has distinct, and the other does not but is congruent to the first query.

Lemma 6.15. *Let Q denote a satisfiable UCQ without distinct. Let $Q' = \textsc{distinct}(Q)$. If $Q \cong Q'$, then:*

$$\textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q)\big)\big)\big)\big) = \textsc{r}^-\big(\textsc{l}\big(\textsc{m}\big(\textsc{r}\big(\textsc{u}(Q')\big)\big)\big)\big).$$

Please see Appendix A.1.16 for the proof.

Having stated all of the core results, we are left to make the final claim of completeness.

Theorem 6.2. *Given two EMQs Q_1 and Q_2, if $Q_1 \cong Q_2$ then*

$$\mathrm{R}^-\big(\mathrm{L}\big(\mathrm{M}\big(\mathrm{R}\big(\mathrm{U}(Q_1)\big)\big)\big)\big) = \mathrm{R}^-\big(\mathrm{L}\big(\mathrm{M}\big(\mathrm{R}\big(\mathrm{U}(Q_2)\big)\big)\big)\big).$$

Please see Appendix A.1.17 for the proof.

Finally we can leverage soundness and completeness for the following stronger claim.

Theorem 6.3. *Given two EMQs Q_1 and Q_2, it holds that $Q_1 \cong Q_2$ if and only if*

$$\mathrm{R}^-\big(\mathrm{L}\big(\mathrm{M}\big(\mathrm{R}\big(\mathrm{U}(Q_1)\big)\big)\big)\big) = \mathrm{R}^-\big(\mathrm{L}\big(\mathrm{M}\big(\mathrm{R}\big(\mathrm{U}(Q_2)\big)\big)\big)\big).$$

Please see Appendix A.1.18 for the proof.

6.6.3. Complexity

With respect to the complexity of the problem of computing the canonical form of (E)MQs in SPARQL, a solution to this problem can be trivially used to decide the equivalence of MQs, which is Π_2^P−complete.

With respect to the complexity of the algorithm $\mathrm{R}^-(\mathrm{L}(\mathrm{M}(\mathrm{R}(\mathrm{U}(\cdot)))))$, for simplicity we will assume as input an MQ Q such that all projected variables are contained in the query,[16] which will allow us to consider the complexity at the level of triple patterns. We will denote by n the number of triple patterns in Q.

Letting $n = km$, then the largest query that can be produced by $\mathrm{U}(Q)$ is when we have as input:

$$\begin{aligned} Q = \mathrm{AND}\big(&\mathrm{UNION}\big(\{t_{1,1}\}, \ldots, \{t_{1,k}\}\big), \\ &\ldots, \\ &\mathrm{UNION}\big(\{t_{m,1}\}, \ldots, \{t_{m,k}\}\big)\big) \end{aligned}$$

which will produce a query with a union of k^m BGPs, each of size m:

$$\begin{aligned} \mathrm{U}(Q) = \mathrm{UNION}\big(&\{t_{1,1}, \ldots, t_{1,k}\} \\ &\times \ldots \\ &\times \{t_{m,1}, \ldots, t_{m,k}\}\big) \end{aligned}$$

Thus $\mathrm{U}(Q)$ may produce a UCQ with mk^m triple patterns in total. Given $n = km$, when $n > 2$, then k^m is maximised in the general case when $k = \lceil e \rceil = 3$ (e is Euler's number) and $m = n/k = n/3$. We thus have at most $O(mk^m) = O((n/3)3^{n/3}) = O(n3^{n/3})$ triple patterns for $\mathrm{U}(Q)$ in the worst case, with at most $O(3^{n/3})$ BGPs, and the largest BGP having at most $O(n)$ triple patterns. We remark that the complexity of the other steps for $\mathrm{U}(Q)$ is trivially upper-bounded by $O(n3^{n/3})$.

With respect to $\mathrm{R}(\cdot)$, the number of triples in the r-graph is $O(j)$ on j the number of triple patterns in the input query, giving us $O(n3^{n/3})$ for the $\mathrm{R}(\cdot)$ step in $\mathrm{R}(\mathrm{U}(\cdot))$, i.e., applying $\mathrm{R}(\cdot)$ on the result of $\mathrm{U}(\cdot)$.

With respect to $\mathrm{M}(\cdot)$, first we consider BGP minimisation, which requires computing the core of each BGP's r-graph G. Letting j denote the number of unique subject and objects in G being minimised, which is also an upper bound for the number of blank nodes, we will assume a brute-force $O(j^j)$ algorithm that searches over every mapping of blank nodes to terms in G, looking for the one that maps to the fewest unique terms (this mapping indicates the core [28]). Note that the number of triples in the r-graph for each BGP is bounded by $O(n)$, and so is the number of unique subjects and objects. Furthermore, the number of BGPs is bounded by $O(3^{n/3})$. Thus the cost of minimising all BGPs is $O(3^{n/3}n^{cn})$ for some constant $c > 1$. We must also check containment between each pair of BGP r-graphs (G', G'') in order to apply UCQ minimisation. Again, assuming the number of subjects and

[16]Other cases are not difficult to manage, but require considering the length of a property path, the number of projected variables not appearing the query, etc., in the input, which we consider to be inessential to the complexity, and to our discussion here.

objects in $G' \cup G''$ to be bounded by j, we can assume a brute-force $O(j^j)$ algorithm that considers all mappings. Given $O(3^{n/3})$ BPGs, we have $O((3^{n/3})^2) = O(3^{2n/3})$ pairs of BGPs to check, giving us a cost of $O(3^{2n/3}n^{cn})$. Adding both BGP and UCQ minimisation costs, we have $O(n^{cn}(3^{n/3} + 3^{2n/3})) = O(n^{cn}3^{2n/3})$ for the $\mathtt{M}(\cdot)$ step in $\mathtt{M}(\mathtt{R}(\mathtt{U}(\cdot)))$. We can then reduce $O(n^{cn}3^{2n/3})$ to $O(2^{cn \log n})$ by converting both bases to 2 and removing the constant factors.[17]

With respect to $\mathtt{L}(\cdot)$, letting j denote the number of triples in the input, we will assume a brute-force $O((cj)!)$ algorithm, for some constant $c > 0$, that searches over all ways of canonically labelling blank nodes from the set $\{$`_:x1`$, \ldots,$ `_:x`$b\}$, where b is the number of blank nodes (in $O(j)$). We remark that the total size of the r-graph is still bounded by $O(n3^{n/3})$, as the minimisation step does not add to the size of the r-graph. Since the number of blank nodes is bounded by $O(n3^{n/3})$, the cost of the $\mathtt{L}(\cdot)$ step in $\mathtt{L}(\mathtt{M}(\mathtt{R}(\mathtt{U}(\cdot))))$ is $O((cn3^{n/3})!)$ for some constant $c > 0$.

Finally, given a graph with j triples, then $\mathtt{R}^-(\cdot)$ is possible in time $O(j \log j)$, where some sorting is needed to ensure a canonical form. Given an input r-graph of size $O(n3^{n/3})$, we have a cost of $O(n3^{n/3} \log n3^{n/3}) = O(n3^{n/3}(\log n + (n/3)\log 3)) = O(n^2 3^{n/3})$ for the $\mathtt{R}^-(\cdot)$ step in $\mathtt{R}^-(\mathtt{L}(\mathtt{M}(\mathtt{R}(\mathtt{U}(\cdot)))))$.

Putting it all together, the complexity of canonicalising an MQ Q with n triple patterns using the procedure $\mathtt{R}^-(\mathtt{L}(\mathtt{M}(\mathtt{R}(\mathtt{U}(Q)))))$ is as follows:

$$O\big(n3^{n/3} + n3^{n/3} + 2^{cn \log n} + \big(cn3^{n/3}\big)! + n^2 3^{n/3}\big)$$

which we can reduce to $O((cn3^{n/3})!)$, with the factorial canonical labelling of the complete exponentially-sized UCQ r-graph yielding the dominant term.

Overall, this complexity assumes worst cases that we expect to be rare in practice, and our analysis assumes brute-force methods for finding homomorphisms, computing cores, labelling blank nodes, etc., whereas we use more optimised methods. For example, the exponentially-sized UCQ r-graphs form a tree-like structure connecting each BGP, where it would be possible to canonically label this structure in a more efficient manner than suggested by this worst-case analysis. Thus, though the method has a high computational cost, this does not necessarily imply that it will be impractical for real-world queries. Still, we can conclude that the difficult cases for canonicalisation are represented by input queries with joins of unions, and that minimisation and canonical labelling will likely have high overhead. We will discuss this further in the context of experiments presented in Section 8.

7. Canonicalisation of SPARQL 1.1 queries

While the previous section describes a sound and complete procedure for canonicalising EMQs, many SPARQL 1.1 queries in practice use features that fall outside of this fragment. Unfortunately we know from Table 16 that deciding equivalence for the full SPARQL 1.1 language is undecidable, and thus that an algorithm for sound and complete canonicalisation (that is guaranteed to halt) does not exist. Since completeness is not a strong requirement for certain use-cases (e.g., for caching, it would imply a "cache miss" that would otherwise happen without canonicalisation), we rather aim for a sound canonicalisation procedure that supports all features of SPARQL 1.1. Such a procedure supports all queries found in practice, preserving congruence, but may produce different canonicalised output for congruent queries.

7.1. Algebraic rewritings

We now describe the additional rewritings we apply in the case of SPARQL 1.1 queries that are not EMQs, in particular for filters, for distinguishing local variables, and for property paths (RPQs). We further describe how canonicalisation of monotone sub-queries is applied based on the previous techniques.

[17]With $n^{cn} = (2^{\log n})^{cn} = 2^{cn \log n}$, and $3^{2n/3} = (2^{\log 3})^{2n/3} = 2^{2n/3 \log 3}$, then $n^{cn}3^{2n/3} = 2^{cn \log n + 2n/3 \log 3} \in O(2^{cn \log n})$.

Table 20
Equivalences given by Schmidt et al. [53] for filters under set semantics

Pushing filters inside/outside union	$[\text{FILTER}_R(Q_1)\text{UNION}\text{FILTER}_R(Q_2)] \equiv \text{FILTER}_R([Q_1 \text{UNION} Q_2])$
Filter conjunction	$\text{FILTER}_{R_1}(\text{FILTER}_{R_2}(Q_1)) \equiv \text{FILTER}_{R_1 \wedge R_2}(Q)$
Filter disjunction	$[\text{FILTER}_{R_1}(Q)\text{UNION}\text{FILTER}_{R_2}(Q)] \equiv \text{FILTER}_{R_1 \vee R_2}(Q)$
Pushing filters inside/outside join	$[\text{FILTER}_R(Q_1)\text{AND}Q_2] \equiv \text{FILTER}_R([Q_1 \text{AND} Q_2])$ if vars $R \subseteq \text{svars}(Q_1)$
Pushing filters inside/outside optional	$[\text{FILTER}_R(Q_1)\text{OPT}Q_2] \equiv \text{FILTER}_R([Q_1 \text{OPT} Q_2])$ if vars $R \subseteq \text{svars}(Q_1)$

Table 21
Syntactic approximation of safe variables where B is a basic graph pattern; N is a navigational graph pattern; Q', Q_1 and Q_2 are graph patterns; x is an IRI, v is a variable; V is a set of variables; R s a built-in expression; $\mathfrak{M}$ is a bag of solution mappings; Λ is a set of aggregation expression–variable pairs; and Q'' is a group-by pattern

$Q = B$	$\therefore \text{svars}(Q) = \text{vars } B$
$Q = N$	$\therefore \text{svars}(Q) = \text{vars } N$
$Q \in \{[Q_1 \text{AND} Q_2], [Q_1 \text{SERVICE}_x^{\texttt{false}} Q_2]\}$	$\therefore \text{svars}(Q) = \text{svars}(Q_1) \cup \text{svars}(Q_2)$
$Q = [Q_1 \text{UNION} Q_2]$	$\therefore \text{svars}(Q) = \text{svars}(Q_1) \cap \text{svars}(Q_2)$
$Q \in \{[Q_1 \text{FE} Q_2], [Q_1 \text{FNE} Q_2], [Q_1 \text{MINUS} Q_2], [Q_1 \text{OPT} Q_2], [Q_1 \text{SERVICE}_x^{\texttt{true}} Q_2]\}$	$\therefore \text{svars}(Q) = \text{svars}(Q_1)$
$Q \in \{\text{SELECT}_V(Q'), \text{GROUP}_V(Q')\}$	$\therefore \text{svars}(Q) = \text{svars}(Q') \cap V$
$Q \in \{\text{FILTER}_R(Q'), \text{BIND}_{R,v}(Q'), \text{GRAPH}_x(Q')\}$	$\therefore \text{svars}(Q) = \text{svars}(Q')$
$Q = \text{GRAPH}_v(Q')$	$\therefore \text{svars}(Q) = \text{svars}(Q') \cup \{v\}$
$Q = \text{VALUES}_{\mathfrak{M}}(Q')$	$\therefore \text{svars}(Q) = \text{svars}(Q') \cup \text{svars}(\mathfrak{M})$
$Q \in \{\text{HAVING}_A(Q''), \text{AGG}_\Lambda(Q'')\}$	$\therefore \text{svars}(Q) = \text{svars}(Q'')$

7.1.1. *Filter normalisation*

Schmidt et al. [53] propose a number of rules for filters, which form the basis for optimising queries by applying filters as early as possible to ensure that the number of intermediate results are reduced. We implement the rules shown in Table 20. It is a folklore result that such rewritings further hold in the case of bag semantics, where they are used by a wide range of SPARQL engines for optimisation purposes: intuitively, filters set to zero the multiplicity of solutions that do not pass in the case of both bag or set semantics, and preserve the multiplicity of other solutions.

With respect to the latter two rules, we remark that this holds only if the variables of the filter expression R are contained within the safe variables of Q_1, i.e., the variables that must be bound in any solution of Q_1 over any dataset. While we defined this notion semantically in Section 3.9, in order to apply such rules in practice, Schmidt et al. [53] define safe variables syntactically. We extend their syntactic definitions to cover more features of SPARQL 1.1, as shown in Table 21. These syntactic definitions do not cover all cases, but rather under-approximate the set of safe variables; as was mentioned in Section 3.9, deciding if a variable is (semantically) safe or not is undecidable. By conservatively under-approximating safe variables, we will apply rewritings in a subset of the cases in which they may actually apply, choosing soundness over completeness in the face of undecidability.

In our case, rather than decomposing filters with disjunction or conjunction, we join them together, creating larger filter expressions that can be normalised.

Example 7.1. Consider the following query:

```
SELECT DISTINCT ?x ?y ?z WHERE {
  { ?x :sibling ?y FILTER(?x != ?y) }
  OPTIONAL { ?x :twin ?z FILTER(?x != ?z) }
  FILTER(isIRI(?x)) FILTER(strlen((str(?x))) > 4)
}
```

We will rewrite this as follows:

```
SELECT DISTINCT ?x ?y ?z WHERE {
  { ?x :sibling ?y
    FILTER(isIRI(?x) && ?x != ?y && strlen((str(?x))) > 4 )
  }
  OPTIONAL { ?x :twin ?z FILTER(?x != ?z) }
}
```

Note that the `FILTER` inside the optional cannot be moved: if there is a solution μ such that $\mu(\texttt{?x}) = \mu(\texttt{?z})$, having the filter inside the `OPTIONAL` may result in `?z` being unbound, while having it outside would always filter the solution entirely.

7.1.2. Local variable normalisation

Like in the case of union variables, we identify another case where the correspondences between variables in different scopes is coincidental; i.e., where variables with the same name do not correlate. Specifically, we call a variable v *local* to a graph pattern Q on V (see Table 3) if $v \in \text{vars } Q$ and $v \notin V$. Much like in the case of union variables, we can simply rename local variables to distinguish them from variables with the same name in other scopes.

Example 7.2. Consider the following query looking for the names of aunts of people without a father or without a mother.

```
SELECT DISTINCT ?z WHERE {
  { ?w :mother ?m . ?m :sister ?y . ?y :name ?z .
    MINUS { ?w :father ?f  }
  }
  UNION
  { ?w :father ?f . ?f :sister ?y . ?y :name ?z .
    MINUS { ?w :mother ?m  }
  }
}
```

In this case, we distinguish the union variables. However, the variables `?f` and `?m` are local to the first and second `MINUS` clauses, respectively, and thus we can also differentiate them as follows:

```
SELECT DISTINCT ?z WHERE {
  { ?w1 :mother ?m1 . ?m1 :sister ?y1 . ?y1 :name ?z .
    MINUS { ?w1 :father ?f1  }
  }
  UNION
  { ?w2 :father ?f2 . ?f2 :sister ?y2 . ?y2 :name ?z .
    MINUS { ?w2 :mother ?m2  }
  }
}
```

The resulting query is equivalent to the original query, but avoids "false correspondences" of variables.

7.1.3. UCQ normalisation

We continue to apply many of the rules of UCQ normalisation described in Section 6.1. Most of these rules were stated in a general way, and thus apply when other (non-EMQ) features are used in the subqueries of the union and join operators (or outside such operators). There are, however, certain caveats to consider in the more general case:

– In the case of variable normalisation, deciding the set of possible variables becomes undecidable for the full language. It suffices for soundness (but not completeness) to use an overapproximation; given a graph pattern Q on V, we can thus simply take V as the set of possible variables.
– In the case of unsatisfiability normalisation, there are now many other possible causes of unsatisfiable graph patterns; unfortunately, deciding if a pattern is unsatisfiable or not for the full language is undecidable [60]. We currently only remove BGPs with literal subjects, as before for EMQs.
– In the case of set vs. bag normalisation, deciding whether or not a query can return duplicate solutions is undecidable (noting that $\text{UNION}(Q, Q)$ cannot return duplicates if and only if Q is unsatisfiable). Currently we only apply this normalisation in EMQs, though this could be extended in future to consider other cases (such as C2RPQs).

Table 22

Equivalences given by Pérez et al. [44] for set semantics and well-designed patterns

Join can be pushed into optional	$[Q_1 \text{AND}[Q_2 \text{OPT} Q_3]] \equiv [[Q_1 \text{AND} Q_2] \text{OPT} Q_3]$
Join can be pushed into optional	$[[Q_1 \text{OPT} Q_2] \text{AND} Q_3] \equiv [[Q_1 \text{AND} Q_3] \text{OPT} Q_2]$
Filter expression can be pushed into optional	$\text{FILTER}_R([Q_1 \text{OPT} Q_2]) \equiv [\text{FILTER}_R(Q_1) \text{OPT} Q_2]$

7.1.4. Well-designed pattern normalisation

As mentioned in Section 3, SPARQL allows the querying of optional data, that is, values are returned if they are matched by a graph pattern, or are unbound otherwise. *Well-designed patterns* denote a class of graph patterns where for each sub-pattern of the form $Q = [Q_1 \text{OPT} Q_2]$ it follows that all variables that appear both outside of Q and in Q_2 must also appear in Q_1. We can check for *well-designedness* in linear time over the size of the query pattern and the number of optional sub-patterns it contains. A classical result for SPARQL is that well-designed patterns can avoid leaps in computational complexity for the evaluation of queries when adding (unrestricted) OPTIONAL. Furthermore, well-designed patterns permit additional normal forms involving OPTIONAL to be applied [44], per Table 22. We exploit these rules to capture additional equivalences involving such patterns.

Example 7.3. Let us consider the query Q_1:

```
SELECT DISTINCT ?x ?y ?n WHERE {
  { ?x :father ?y }
    OPTIONAL { ?y :firstname ?n }
  ?x :firstname ?n .
}
```

This query is not well-designed due to the variable ?n appearing on the right but not the left of an OPTIONAL, while also appearing outside the OPTIONAL.

On the other hand, the following query Q_2 contains a well-designed pattern inside its WHERE:

```
SELECT DISTINCT ?x ?y ?n1 ?n2 WHERE {
  { ?x :father ?y }
    OPTIONAL { ?y :firstname ?n2 }
  ?x :firstname ?n1 .
}
```

Thus we can rewrite it to:

```
SELECT DISTINCT ?x ?y ?n1 ?n2 WHERE {
  { ?x :father ?y . ?x :firstname ?n1 }
    OPTIONAL { ?y :firstname ?n2 }
}
```

per the second rule of Table 22.

Note that if we were to try to apply the same rule on the non-well-designed pattern in Q_1, we would get:

```
SELECT DISTINCT ?x ?y ?n WHERE {
  { ?x :father ?y . ?x :firstname ?n }
    OPTIONAL { ?y :firstname ?n }
}
```

This query is not equivalent to Q_1: it returns the first names (?n) of children (?x) with some father (?y); in other words, the OPTIONAL is redundant. On the other hand, Q_1 returns the first names (?n) of children (?x) with some father (?y) that does not have a first name or has a first name the same as the child (?n); this is because in the original query, the variable ?n is potentially bound to the father's first name (if it exists) before the join on the child's first name is applied.

We note that for queries with well-designed patterns, these rules are not sufficient for the purposes of completeness; we will show an example of incompleteness later in Section 7.5.4. Per the results of Pichler and Skritek [45], equivalence considering projection and well-designed patterns is already undecidable.

7.1.5. Summary

Given a SPARQL 1.1 query Q, we denote by $\mathrm{A}(Q)$ the process involving the application of:

1. filter normalisation (§7.1.1);
2. local variable normalisation (§7.1.2);
3. UCQ normalisation (§7.1.3);
4. well-designed pattern normalisation (§7.1.4).

7.2. Graph representation

We extend the graph representation defined for EMQs in Section 7.2 so as to cover all SPARQL 1.1 query features. This extension is provided in Table 23 (we again include the EMQ features for reference).

The reader may have noted that we omitted three details from Table 23: how to represent built-in and aggregate expressions, and property paths. We now discuss these representations in turn.

7.2.1. Expressions

We recall that a term in **VIBL** is a *built-in expression*, and that if ϕ takes a tuple of values from $\mathbf{IBL} \cup \{\bot, \varepsilon\}$ as input and returns a single value in $\mathbf{IBL} \cup \{\bot, \varepsilon\}$ as output, then an expression $\phi(R_1, \dots, R_n)$, where each $R_1, \dots, R_n$ is a built-in expression, is itself a built-in expression. If a built-in expression R is simply a term $R \in \mathbf{VIBL}$, then we use $\iota(R)$ to represent the expression, where $\mathrm{R}(R) = \emptyset$. Otherwise, if $R = \phi(R_1, \dots, R_n)$ and either ϕ has at most one argument, or ϕ is an commutative function – i.e., the order of arguments is not meaningful – then:

$$\mathrm{R}(R) = \bigl\{\bigl(\iota(R), \texttt{a}, \texttt{:BIExp}\bigr), \bigl(\iota(R), \texttt{:func}, \iota(\phi)\bigr)\bigr\} \\ \cup \bigcup_{i=1}^{n}\bigl(\bigl\{\bigl(\iota(R), \texttt{:arg}, \iota(R_i)\bigr)\bigr\} \cup \mathrm{R}(R_i)\bigr)$$

where $\iota(\phi)$ is an IRI that is assumed to uniquely identify the function, and $\iota(R)$, $\iota(R_i)$ are fresh blank nodes. If ϕ has more than one argument and is not commutative – i.e., if the order of argument is meaningful – then for each R_i, we additionally add a triple $(\iota(R_i), \texttt{:ord}, \iota(i))$ to the above transformation.

As previously remarked, we consider operators (e.g., `&&`, `+`, `=`, etc.) to be represented by functions. We assume that commutative (and associative) operators – e.g., `?a+?b+?c`, are represented as commutative n-ary functions – e.g., `SUM(?a, ?b, ?c)`. This allows for the representational graphs to abstract away details regarding the ordering of operands of commutative functions. We further remark that $[Q_1 \mathrm{FE} Q_2]$ and $[Q_1 \mathrm{FNE} Q_2]$ are considered to be filters in the concrete syntax; given that they do not use a built-in expression, we have rather defined their representation in Table 18.

With respect to aggregation expressions, we recall that if ψ is a function that takes a bag of tuples from **IBL** and returns a value in $\mathbf{IBL} \cup \{\bot, \varepsilon\}$, then an expression $A = \psi(R_1, \dots, R_n)$, where each $R_1, \dots, R_n$ is a built-in expression, is an aggregation expression. An expression A of this form can be represented as:

$$\mathrm{R}(A) = \bigl\{\bigl(\iota(A), \texttt{a}, \texttt{:AggExp}\bigr), \bigl(\iota(A), \texttt{:func}, \iota(\psi)\bigr)\bigr\} \\ \cup \bigcup_{i=1}^{n}\bigl(\bigl\{\bigl(\iota(A), \texttt{:arg}, \iota(R_i)\bigr)\bigr\} \cup \mathrm{R}(R_i)\bigr)$$

where $\iota(\psi)$ is an IRI that uniquely identifies the function ψ, and $\iota(A)$, $\iota(R_i)$ are fresh blank nodes.[18]

7.2.2. Property paths

Because property paths without inverses and negated property sets are regular expressions, and any regular expression can be transformed into a finite automaton, and a finite automaton has a graph-like structure, we opt to

[18] Though not necessary for SPARQL, `:ord` triples can be added if the order of operands matters, as before for built-in expressions.

Table 23

Definitions for representational graphs $\mathrm{R}(Q)$ of graph patterns Q, where “a” abbreviates `rdf:type`, B is a basic graph pattern, N is a navigational graph pattern, $Q_1, \dots, Q_n$, Q' are graph patterns, c is an RDF term, e is a non-simple property path (not an IRI), k is a non-zero natural number, v is a variable, w is a variable or IRI, x is an IRI, y is a variable or property path, μ is a solution mapping, Δ is a boolean value, V is a set of variables, X is a set of variables and/or IRIs, Y and Y' are sets of IRIs, R is a built-in expression, A is an aggregate expression, $\mathfrak{M}$ is a bag of solution mappings, Λ is a set of aggregate expression–variable pairs, and Ω is a non-empty sequence of order comparators

$\cdot$	$\mathrm{R}(\cdot)$
B	$\{(\iota(\cdot), \texttt{a}, \texttt{:And})\} \cup \bigcup_{(s,p,o)\in B}\{(\iota(\cdot), \texttt{:arg}, \iota((s, p, o)))\} \cup \mathrm{R}((s, p, o))$
N	$\{(\iota(\cdot), \texttt{a}, \texttt{:And})\} \cup \bigcup_{(s,y,o)\in N}\{(\iota(\cdot), \texttt{:arg}, \iota((s, y, o)))\} \cup \mathrm{R}((s, y, o))$
$\mathrm{AND}(Q_1, \dots, Q_n)$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q_1)), \dots, (\iota(\cdot), \texttt{:arg}, \iota(Q_n)), (\iota(\cdot), \texttt{a}, \texttt{:And})\} \cup \mathrm{R}(Q_1) \cup \cdots \cup \mathrm{R}(Q_n)$
$\mathrm{UNION}(Q_1, \dots, Q_n)$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q_1)), \dots, (\iota(\cdot), \texttt{:arg}, \iota(Q_n)), (\iota(\cdot), \texttt{a}, \texttt{:Union})\} \cup \mathrm{R}(Q_1) \cup \cdots \cup \mathrm{R}(Q_n)$
$[Q_1 \mathrm{MINUS} Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{a}, \texttt{:Minus})\} \cup \mathrm{R}(Q_1) \cup \mathrm{R}(Q_2)$
$[Q_1 \mathrm{OPT} Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{a}, \texttt{:Optional})\} \cup \mathrm{R}(Q_1) \cup \mathrm{R}(Q_2)$
$\mathrm{FILTER}_R(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:exp}, \iota(R)), (\iota(\cdot), \texttt{a}, \texttt{:Filter})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(R)$
$[Q_1 \mathrm{FE} Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{a}, \texttt{:Exists})\} \cup \mathrm{R}(Q_1) \cup \mathrm{R}(Q_2)$
$[Q_1 \mathrm{FNE} Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{a}, \texttt{:NotExists})\} \cup \mathrm{R}(Q_1) \cup \mathrm{R}(Q_2)$
$\mathrm{BIND}_{R,v}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:exp}, \iota(R)), (\iota(\cdot), \texttt{:var}, \iota(v)), (\iota(\cdot), \texttt{a}, \texttt{:Bind})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(R)$
$\mathrm{VALUES}_{\mathfrak{M}}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:vals}, \iota(\mathfrak{M})), (\iota(\cdot), \texttt{a}, \texttt{:Values})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(\mathfrak{M})$
$\mathrm{GRAPH}_w(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:graph}, \iota(w))\} \cup \mathrm{R}(Q')$
$[Q_1 \mathrm{SERVICE}^{\Delta}_{x} Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{:srv}, x), (\iota(\cdot), \texttt{:sil}, \iota(\Delta)), (\iota(\cdot), \texttt{a}, \texttt{:Service})\} \cup \mathrm{R}(Q_1) \cup \mathrm{R}(Q_2)$
$\mathrm{GROUP}_V(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{a}, \texttt{:GroupBy})\} \cup \bigcup_{v\in V}\{(\iota(\cdot), \texttt{:var}, \iota(v))\} \cup \mathrm{R}(Q')$
$\mathrm{HAVING}_A(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:exp}, \iota(A)), (\iota(\cdot), \texttt{a}, \texttt{:Having})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(A)$
$\mathrm{AGG}_\Lambda(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:exp}, \iota(\Lambda)), (\iota(\cdot), \texttt{a}, \texttt{:Aggregate})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(\Lambda)$
$\mathrm{ORDER}_\Omega(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:exp}, \iota(\Omega)), (\iota(\cdot), \texttt{a}, \texttt{:OrderBy})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(\Omega)$
$\mathrm{DISTINCT}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{a}, \texttt{:Distinct})\} \cup \mathrm{R}(Q')$
$\mathrm{REDUCED}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{a}, \texttt{:Reduced})\} \cup \mathrm{R}(Q')$
$\mathrm{OFFSET}_k(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:off}, \iota(k)), (\iota(\cdot), \texttt{a}, \texttt{:Offset})\} \cup \mathrm{R}(Q')$
$\mathrm{LIMIT}_k(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:limit}, \iota(k)), (\iota(\cdot), \texttt{a}, \texttt{:Limit})\} \cup \mathrm{R}(Q')$
$\mathrm{SELECT}_V(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{a}, \texttt{:Select})\} \cup \bigcup_{v\in V}\{(\iota(\cdot), \texttt{:var}, \iota(v))\} \cup \mathrm{R}(Q')$
$\mathrm{ASK}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{a}, \texttt{:Ask})\} \cup \mathrm{R}(Q')$
$\mathrm{CONSTRUCT}_B(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:cons}, \iota(B)), (\iota(\cdot), \texttt{a}, \texttt{:Construct})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(B)$
$\mathrm{DESCRIBE}_X(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:desc}, \iota(X)), (\iota(\cdot), \texttt{a}, \texttt{:Describe})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(X)$
$\mathrm{FROM}_{Y,Y'}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:from}, \iota(Y)), (\iota(\cdot), \texttt{:fromn}, \iota(Y')), (\iota(\cdot), \texttt{a}, \texttt{:From})\} \cup \mathrm{R}(Q') \cup \mathrm{R}(Y) \cup \mathrm{R}(Y')$
(s, p, o)	$\{(\iota(\cdot), \texttt{:s}, \iota(s)), (\iota(\cdot), \texttt{:p}, \iota(p)), (\iota(\cdot), \texttt{:o}, \iota(o)), (\iota(\cdot), \texttt{a}, \texttt{:TP})\}$
(s, e, o)	$\{(\iota(\cdot), \texttt{:s}, \iota(s)), (\iota(\cdot), \texttt{:p}, \iota(e)), (\iota(\cdot), \texttt{:o}, \iota(o)), (\iota(\cdot), \texttt{a}, \texttt{:NP})\} \cup \mathrm{R}(e)$
e	minimal DFA with $\iota(e)$ as start node; see Section 7.2.2
X	$\{(\iota(\cdot), \texttt{a}, \texttt{:IriVarSet})\} \cup \bigcup_{x\in X}\{(\iota(\cdot), \texttt{:el}, x)\}$
Y	$\{(\iota(\cdot), \texttt{a}, \texttt{:IriSet})\} \cup \bigcup_{y\in Y}\{(\iota(\cdot), \texttt{:el}, y)\}$
$\mathfrak{M}$	$\{(\iota(\cdot), \texttt{a}, \texttt{:SolBag})\} \cup (\bigcup_{\mu\in\mathfrak{M}}\{(\iota(\cdot), \texttt{:sol}, \iota((\mu, \mathfrak{M}(\mu))))\} \cup \mathrm{R}((\mu, \mathfrak{M}(\mu))))$
(μ, k)	$\{(\iota(\cdot), \texttt{a}, \texttt{:Binding}), (\iota(\cdot), \texttt{:num}, \iota(k))\} \cup (\bigcup_{v\in\mathrm{dom}(\mu)}\{(\iota(\cdot), \texttt{:el}, \iota((v, \mu(v))))\} \cup \mathrm{R}((v, \mu(v)))$
(v, c)	$\{(\iota(\cdot), \texttt{:var}, \iota(v)), (\iota(\cdot), \texttt{:val}, \iota(c))\}$
R	see Section 7.2.1
A	see Section 7.2.1
Λ	$\{(\iota(\cdot), \texttt{a}, \texttt{:ABindSet})\} \cup (\bigcup_{(A,v)\in\Lambda}\{(\iota(\cdot), \texttt{:arg}, \iota((A, v)))\} \cup \mathrm{R}((A, v)))$
(A, v)	$\{(\iota(\cdot), \texttt{:exp}, \iota(A)), (\iota(\cdot), \texttt{:var}, \iota(v))\} \cup \mathrm{R}(A)$
Ω	$\{(\iota(\cdot), \texttt{a}, \texttt{:OBExpSeq})\} \cup (\bigcup_{k=1}^{\lvert\Omega\rvert}\{(\iota(\cdot), \texttt{:arg}, \iota((\Omega[k], k)))\} \cup \mathrm{R}((\Omega[k], k)))$
$((R, \Delta), k)$	$\{(\iota(\cdot), \texttt{:exp}, \iota(R)), (\iota(\cdot), \texttt{:asc}, \iota(\Delta)), (\iota(\cdot), \texttt{:ord}, \iota(k))\} \cup \mathrm{R}(R)$

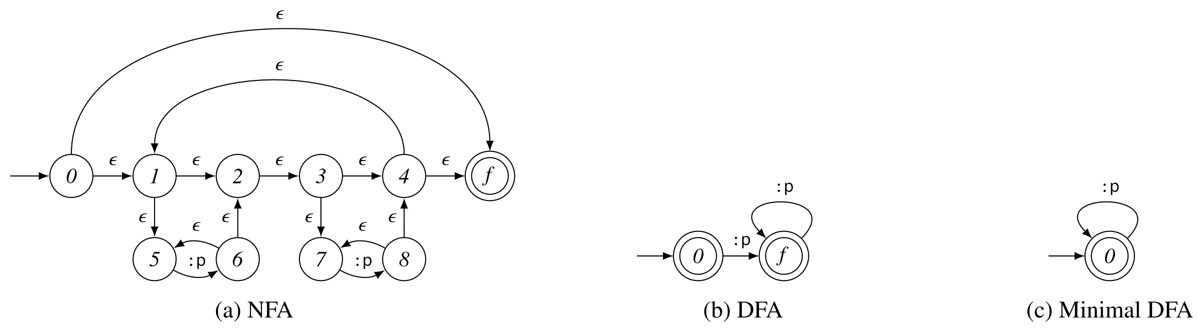


Fig. 5. NFA, DFA and minimal DFA produced for the RPQ (:p*/:p*)*.

represent property paths based on finite automata. This approach allows us to apply known normal forms for finite automata, providing, in turn, a normal form for RPQs, i.e., property paths without inverses or negated property sets; it further provides partial canonicalisation in the case of full property paths. Another benefit of this approach is that the automaton can be converted straightforwardly into RDF and used for the graph representation.

Given an RPQ, we apply the following process:

1. we construct a non-deterministic finite automaton (NFA) based on Thompson's construction for transforming regular expressions to NFAs [59];
2. we convert this NFA into a deterministic finite automaton (DFA) by a standard subset expansion (the resulting DFA may be exponential on the number of states in the NFA);
3. we perform a minimisation of the DFA using Hopcroft's algorithm [29], which produces a canonical DFA such that all regular expressions that express the same language will produce the same DFA (modulo isomorphism on states).

We now provide an example to illustrate the process.

Example 7.4. Consider an RPQ (:p*/:p*)*. In Fig. 5 we provide an example of the corresponding NFA produced by Thompson's algorithm, the DFA produced by subset expansion, and the minimal DFA produced by Hopcroft's algorithm.

The minimal DFA produced by Hopcroft's algorithm for the RPQ e can then be encoded as an r-graph $\mathrm{R}(e)$ where each state is assigned a fresh blank node, and transitions are labelled with their predicate IRI.

In order to generalise this process to full property paths, we must also support inverses and negated property sets. For inverses, we initially attempt to eliminate as many inverses as possible; for example, ^(^:p)* would be rewritten to simply :p* Thereafter, any remaining inverses or negated property sets are represented with a canonical IRI; for example, ^:p* becomes :p-inv*. Thus the property path again becomes a regular expression, and is converted into a DFA using the aforementioned process. It is worth noting that the resulting DFA may not be canonical, and may thus miss equivalences between property paths (e.g., :p*/^p*, ^:p*/p* and (:p|^p)* should all be equivalent, but are not when represented as :p*/:p-inv*, :p-inv*/:p* and (:p|:p-inv)*, respectively). The r-graph can then be computed as before, where for a negated property set $e = \texttt{!(}p_1 | \dots | p_n\texttt{)}$, we also add:

$$\mathrm{R}(e) = \big\{\big(\iota(e), \texttt{a}, \texttt{:notOneOf}\big),\\ \big(\iota(e), \texttt{:arg}, \iota(p_1)\big), \dots, \big(\iota(e), \texttt{:arg}, \iota(p_n)\big)\big\}.$$

We will discuss the inverse mapping from the minimal DFA back to a path expression in Section 7.4.

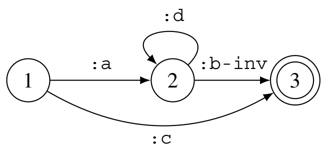
7.3. Minimisation and canonicalisation

Minimisation is applied only to BGPs and UBGPs that are contained within the larger query in the r-graph, considering any variable appearing outside of the BGP or union as a projected variable. We apply canonicalisation on the entire r-graph as before, computing a deterministic labelling for blank nodes.

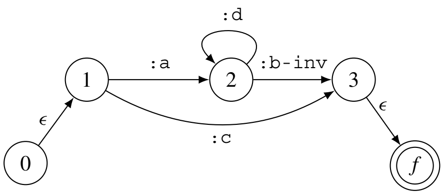
7.4. Inverse mapping

The inverse mapping is defined analogously such that $\mathrm{R}^{-}(\mathrm{R}(Q)) = Q$, with one exception: property paths. This is because we do not represent property paths syntactically in the r-graph, but rather convert them to a minimal DFA, from which we must construct the property path once again. In order to construct the property path from the minimal DFA, we convert it into a regular expression by using the state elimination method [8], from which we can retrieve a normalised RPQ. Finally the inverse and negated property set IRIs are substituted accordingly (if present).

Example 7.5. Consider the following DFA:

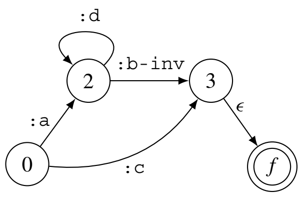


The first step consists in transforming this DFA into an equivalent NFA by introducing a new initial state q_0 and a single accepting state q_f. We add an ϵ transition from q_0 to q_1, and from all accepting states to q_f.

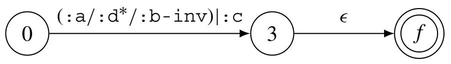


To eliminate a state q_a we have to replace all transitions that pass through q_a. Assuming there exists a path between q_i and q_j that passes through q_a, we define the new transition e_{ij} as $e_{ij} = (e_{ia}/e_a^*/e_{aj})|e'_{ij}$. In this expression, we have e_{ia}, the transition from q_i to q_a, followed by zero or more instances of e_a, which represents any self-loops in q_a, or ϵ if no such self-loops exist. This is then followed by e_{aj}, the transition from q_a to q_j. Finally, we have to consider the fact that there may have already been a path from q_i to q_j, denoted here by e'_{ij}; hence we append $\ldots|e'_{ij}$ to the expression to include the existing transition.

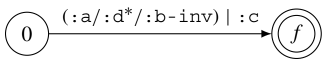
We now eliminate q_1 following this process. Since q_1 has no self-loops and there are no existing paths between q_0 and q_2, these terms are excluded from the expression. Then $e_{02} = (e_{01}/e_1^*/e_{12})|e_{02} = \epsilon/\texttt{:a} = \texttt{:a}$, and $e_{03} = (e_{01}/e_1^*/e_{13})|e_{03} = \epsilon/\texttt{:c} = \texttt{:c}$.



Next we eliminate q_2. In this case, q_2 does have a self-loop, and there exists a transition between q_0 and q_3 so the expression is as follows: $e_{03} = (e_{02}/e_2^*/e_{23})|e_{03} = (\texttt{:a}/\texttt{:d}^*/\texttt{:b-inv})\ |\ \texttt{:c}$.



We now eliminate q_3. In this case:



Finally, we substitute the IRI `:b-inv` for the inverse expression `^:b`, yielding `(:a/:d*/^:b)|:c`.

7.5. *Soundness, completeness and incompleteness*

Given a SPARQL 1.1 query Q, the canonicalisation procedure is then defined by $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{A}(Q)))))$. We now discuss some formal properties of this procedure.

7.5.1. *Soundness and completeness for EMQs*

First we look at the case of EMQs, and ask if the extended canonicalisation procedure is sound and complete for this fragment.

Lemma 7.1. *Given two EMQs Q_1 and Q_2, it holds that $Q_1 \cong Q_2$ if and only if*

$$\text{R}^-\big(\text{L}\big(\text{M}\big(\text{R}\big(\text{A}(Q_1)\big)\big)\big)\big) = \text{R}^-\big(\text{L}\big(\text{M}\big(\text{R}\big(\text{A}(Q_2)\big)\big)\big)\big).$$

Please see Appendix A.2.1 for the proof.

7.5.2. *Soundness for SPARQL 1.1*

Next we show that the process is sound for queries in the full SPARQL 1.1 query language.

Lemma 7.2. *For a SPARQL 1.1 query Q, it holds that:*

$$\text{R}^-\big(\text{L}\big(\text{M}\big(\text{R}\big(\text{A}(Q)\big)\big)\big)\big) \cong Q.$$

Please see Appendix A.2.2 for the proof.

7.5.3. *Complexity*

In terms of the complexity of (partially) canonicalising queries with these additional features, if we assume that the size of the input query Q is in $O(n)$, where n is the number of unique triple patterns and path patterns in Q, then the complexity remains bounded by that of canonicalising monotone queries: $O(2^{cn\log n})$ for some constant $c > 0$. This assumes that features of the query that do not involve triple or path patterns (e.g., `BIND`, `VALUES`, `FILTER`, etc.) are of bounded size and bounded in number, while path expressions are of bounded length. This may be an oversimplification.

We may rather consider the "token length" of the query, which is the number of terminals – RDF terms, variables, keywords, etc. – appearing in the syntax of the query. In this case, we must additionally consider the costs of computing a normal form for RPQs. Given an RPQ of length l, which includes the number of non-parenthetical symbols (including IRIs, $\ast$, $+$, $|$, $/$) Thompson's construction creates an NFA of size $O(l)$. Subset expansion may then create a DFA with $O(2^l)$ states that remains exponential after minimisation. This will result in an exponentially-sized representation of the RPQ. Minimisation is only applied on (U)BGPs, and thus does not apply on this representation. However, canonical labelling will occur on this representation, where assuming again a brute-force method in the order of $O(b!)$ for b the number of blank nodes, we now have a complexity of $O(2^l!)$ for canonicalising the RPQ representation graph of a property path of length l, and given that we may have n such property paths, where n is the number of triple patterns and path expressions, this generates a cost of $O((cn2^l)!)$ for canonically labelling a navigational graph pattern, which we can add to the cost for monotone queries: $O((cn2^l)! + (cn3^{n/3})!)$, where l is the length of the longest property path, n is the number of triple patterns and path patterns, and $c > 0$ is some constant to account for the additional "syntactic" blank nodes that appear in the r-graph.

7.5.4. *Incompleteness for SPARQL 1.1*

We provide some examples of incompleteness to illustrate the limitations of the canonicalisation process for the full SPARQL 1.1 language.

We start with filters, which, when combined with a particular graph pattern, may always be true, may never be true, may contain redundant elements, etc.; however, detecting such cases can be highly complex.

Example 7.6. Consider the following example:

```
SELECT ?o
WHERE {
  :Ed ?p ?o .
  FILTER(!isIRI(?p))
}
```

The `FILTER` here will always return false as the predicate in an RDF graph must always be an IRI. Thus the query is unsatisfiable and thus ideally would be rewritten to $Q_\emptyset$; however, we do not consider the semantics of filter functions (other than boolean combinations).

Note that reasoning about filters is oftentimes far from trivial. Consider the following example:

```
SELECT ?o
WHERE {
  :Ed ?p ?o .
  FILTER(!contains(":",str(?p)))
}
```

This query is unsatisfiable because predicates must be IRIs, and IRIs must always contain a colon (to separate the scheme from the hierarchical path) [20].

Next we consider issues with property paths.

Example 7.7. Consider the following example:

```
SELECT ?anc
WHERE {
  :Ed :parent*/:parent* ?anc .
}
```

Clearly this is equivalent to:

```
SELECT ?anc
WHERE {
  :Ed :parent* ?x . ?x :parent* ?anc .
}
```

But also to:

```
SELECT ?anc
WHERE {
  :Ed :parent* ?anc .
}
```

Currently we rewrite concatenation, inverse and disjunction in paths (not appearing within a recursive expression) to UCQ features. This means that we currently capture equivalence between the first and second query, but not the first and third.

Other examples are due to inverses, or negated property sets; consider for example:

```
SELECT ?anc
WHERE {
  :Ed :parent|!(:parent) ?anc .
}
```

This is equivalent to:

```
SELECT ?anc
WHERE {
  :Ed ?p ?anc .
}
```

However, we do not consider the semantic relation between the expressions `!(:parent)` and `:parent`.

Incompleteness can also occur due to negation, which is complicated by the ambiguities surrounding `NOT EXISTS` as discussed in Section 3.10. We have postponed algebraic rewritings involving negation until this issue is officially resolved.

Incompleteness can also occur while normalising well-designed query patterns with `OPTIONAL`.

Example 7.8. Consider the query Q_1:

```
SELECT ?anc
WHERE {
  { :Ed :parent ?anc .
  OPTIONAL { :Ed :email ?email } } .
  { :Bob :parent ?anc .
  OPTIONAL { :Bob :address ?address } }
}
```

Since AND is commutative, this is equivalent to Q_2:

```
SELECT ?anc
WHERE {
  { :Bob :parent ?anc .
  OPTIONAL { :Bob :address ?address } } .
  { :Ed :parent ?anc .
  OPTIONAL { :Ed :email ?email } }
}
```

If we rewrite each well-designed pattern by pushing the `OPTIONAL` operators outside, we obtain the following equivalent query for Q_1:

```
SELECT ?anc
WHERE {
  { :Ed :parent ?anc .
    :Bob :parent ?anc .
    OPTIONAL { :Bob :address ?address } }
  OPTIONAL { :Ed :email ?email }
}
```

and, analogously, for Q_2:

```
SELECT ?anc
WHERE {
  { :Ed :parent ?anc .
    :Bob :parent ?anc .
    OPTIONAL { :Ed :email ?email } }
  OPTIONAL { :Bob :address ?address }
}
```

However, in the general case it does not hold that $[[Q_1 \text{OPT} Q_2] \text{OPT} Q_3] \equiv [[Q_1 \text{OPT} Q_3] \text{OPT} Q_2]$, and thus we do not capture these equivalences.

We could list an arbitrary number of ways in which arbitrary features can give rise to unsatisfiability or redundancy, or where queries using seemingly different features end up being equivalent. We could likewise provide an arbitrary number of rewritings and methods to deal with particular cases. However, any such method for canonicalising SPARQL 1.1 queries will be incomplete. Furthermore, many such "corner cases" would be rare in practice, where dealing with them might have limited impact. We then see two interesting directions for future work to address these limitations:

1. Use query logs or other empirical methods to determine more common cases that this query canonicalisation framework may miss and implement targeted methods to deal with such cases.
2. Extend the query fragment for which sound and complete canonicalisation is possible; an interesting goal, for example, would be to explore EMQs with full property paths (such queries are similar to C2RPQs [34], for which containment and related problems are decidable).

8. Experiments

In these experiments, we wish to address two principal questions, as follows:

Q1: How is the performance of the canonicalisation procedure in terms of runtime? Which aspects of the procedure take the most time? What kinds of queries are most expensive?

Q2: How many more additional congruent queries can the procedure find in real-world logs versus baseline methods? Which aspects of the procedure are most important for increasing the number of congruent queries detected?

With respect to the first question, we might expect poor performance given that the worst-case of the algorithm is super-exponential. However, this is a worst-case analysis with respect to the size of the query, where queries in practice are often relatively small and simple. Hence our hypothesis is that although there exist queries for which canonicalisation is not computationally feasible in theory, it should run efficiently (in fractions of a second) for the majority of real-world queries in practice (as found in public query logs).

With respect to the second question, most of our expected use-cases benefit from being able to find a wider range of congruent, real-world queries, as found in public query logs; for example, in the case of caching, finding more congruent queries will translate into higher cache hit rates. Thus it is of interest to see how many additional congruent queries our canonicalisation procedure can find from public query logs when compared with baseline (syntactic) methods, and in particular, which parts of the procedure have the most impact in terms of finding more congruent queries. In general, we hypothesise that canonical labelling will be an important component as variable naming would likely be a common variation in real-world queries; on the other hand, we hypothesise that minimisation will be less impactful in terms of finding more congruent queries as we expect few real-world queries to contain the types of redundancy dealt with in Section 6.3.

We thus design a number of experiments over real-world and synthetic queries in order to address these questions and evaluate our expectations.

8.1. Implementation: QCan

We have implemented the canonicalisation procedure in a system that we call *QCan* written in Java. The system uses Jena ARQ [38] for query parsing and processing. Algebraic rewritings are implemented on top of the query algebra that Jena ARQ provides. Constructing the r-graph is likewise implemented on top of Jena. Minimisation is conducted by using the blabel system [28] to compute the core of RDF graphs representing BGPs, thereafter implementing containment checks across BGPs using `ASK` queries evaluated in Jena. Finally canonical labelling is implemented using blabel [28]. All other aspects of the procedure are implemented directly in QCan, including the steps involving the representation and manipulation of property paths as automata. The source code of QCan, along with a suite of congruence test-cases, are available online at: http://github.com/RittoShadow/QCan. The machine used for these experiments has 12 Intel® Xeon® 1.9GHz processors, and runs on Devuan GNU/Linux 2.1, with a maximum heap size of 10GB.

8.2. Real-world query logs

In order to test our algorithm in a real-world setting, we used two query log datasets: the LSQ dataset [52],[19] which itself contains queries from DBpedia, Linked Geo Data, RKB Explorer/Endpoint and Semantic Web Dog Food logs; and Wikidata logs [37] (Interval 7; organic queries).[20] With respect to the datasets:

– DBpedia [35] (DBP) is an RDF dataset extracted principally from Wikipedia, with the main source of content being info-boxes.
– LinkedGeoData [56] (GEO) is an RDF dataset that contains spatial data extracted from the *OpenStreetMap* [23] project.
– The RKB Endpoint (REN) and Explorer (REX) datasets [21] primarily pertain to cultural and heritage data from the British Museum.
– Semantic Web Dog Food (SWDF) [39] was a dataset that is now part of the *Scholarlydata* dataset [40], containing information about scholarly publications and conferences.
– Wikidata [61] (WD) is a collaboratively-edited knowledge graph that complements the projects of Wikimedia with central, structured content.

All of these datasets have public SPARQL endpoints that receive queries from users over the Web. These queries have been published as the Wikidata and LSQ logs for research purposes. Table 24 contains the distribution of features in each of the query sets in our real-world setting. In total we consider 2.8 million queries for our experiments. Despite the fact that BGPs are equivalent to joins of triple patterns, we only count as "AND" those joins for features other than triple patterns. We further gather together under "MODS" the solution modifiers `ORDER BY`, `LIMIT`, `OFFSET` and projection. We observe that BGPs are present in almost all queries, which is an expected result. Features new to SPARQL 1.1 are used more rarely in the LSQ logs, but this is likely because these logs predate the release of the standard; such features are quite widely used in the more recent Wikidata logs. The large use of SERVICE on Wikidata relates to the use of the custom label service offered by the endpoint as a syntactic convenience to help choose language preferences. With respect to the use of property paths, many are quite simple, usually used to find objects of a certain class or its transitive subclasses. Further details of these queries can be found in analyses by Saleem et al. [52], Malyshev et al. [37], Bonifati et al. [7], etc.

Table 24
Distribution of queries using SPARQL features in the query logs; we mark features new to SPARQL 1.1 with an asterisk

Feature	DBP	GEO	REN	REX	SWDF	WD	All
BGP	424,328	842,794	169,425	335,450	112,398	861,383	2,745,778
DISTINCT	192,168	5,477	147,575	210,058	57,195	538,878	1,151,351
AND	58,952	78,575	19,355	34,180	54,702	751,243	997,007
MODS	4,576	412,343	40,957	80,655	60,199	321,239	919,969
OPT	31,211	19,914	9,741	23,587	27,824	343,838	456,115
FILTER	200,001	125,849	8,974	14,938	4,716	92,807	447,285
SERVICE*	0	0	0	0	0	406,990	406,990
UNION	89,766	49,299	1,653	5,550	32,459	57,029	235,756
PATHS*	0	0	96	192	37	268,530	268,855
BIND*	0	0	0	19,641	15,307	71,102	106,050
AGG*	0	0	0	16,444	10,268	42,777	69,489
VALUES*	0	174	0	0	83	55,037	55,294
MINUS*	0	0	0	192	25	5,700	5,917
TOTAL	424,362	842,794	169,617	335,833	112,470	868,993	2,754,069

[19] https://aksw.github.io/LSQ/
[20] https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en

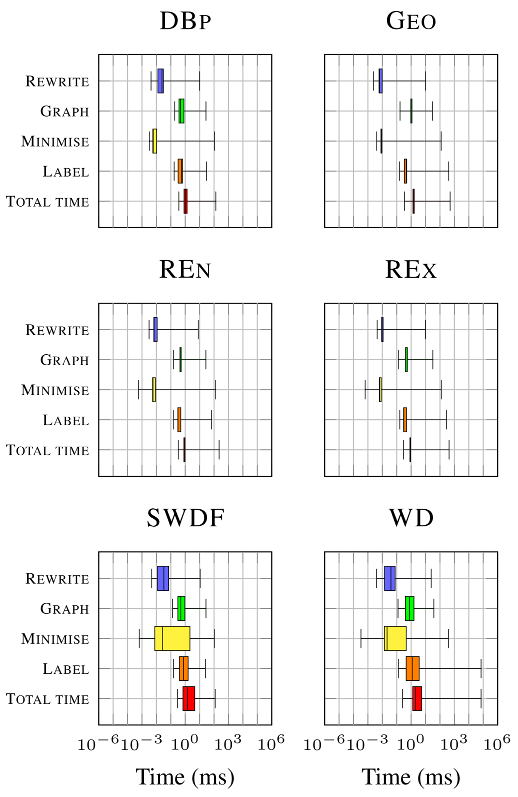


Fig. 6. Runtimes for each step of the canonicalisation algorithm.

8.2.1. Canonicalisation runtimes

We now present the runtimes for canonicalising the queries of the aforementioned logs. All queries in all six logs were successfully canonicalised. The results in Fig. 6 indicate that the fastest queries to canonicalise take around 0.00025 seconds, median times vary from 0.00084 seconds (REn) up to 0.00206 seconds (WD), while max times vary from 0.12 seconds (SWDF) up to 71 seconds (WD). The slowest part of the process, on average, tended to be the canonical labelling, though in the case of REn and REx, the graph construction was slightly slower on average.

Figure 7 shows the runtimes for the canonicalisation of the queries in the aforementioned logs, limited to those that contain features introduced in SPARQL 1.1. Notably the ranges of runtimes of DBp, Geo, REn, and REx are much more stable, and present much less variance than the same runtimes for the full datasets. On the other hand, SWDF and WD present similar minimum and maximum runtimes than the full datasets, but the runtimes in the interquartile range are far more stable than those in the full datasets. This suggests that the addition of queries with features introduced in SPARQL 1.1 do not add a significant overhead to the performance of the algorithm.

Figure 8 shows that the WD set of queries produces the largest r-graphs, with the largest graph containing 3,456 nodes (WD). This is consistent with the results in Fig. 6 since the total runtimes for WD queries can be much higher than those of the other query sets.

The results in Fig. 9 show the runtimes for all query sets grouped by the features they use (at least once, possibly in conjunction with other features). We include SPARQL 1.0 features along with property paths for those logs using such features. These results indicate that queries containing `UNION` tend to take a longer time to process; i.e., it is the most computationally challenging feature to deal with in these queries. This is an expected result since the rewriting step may cause an exponential blow-up on the number of basic graph patterns in a monotone fragment of a query. Since most real world queries are not in a UCQ form, but rather contain combinations of joins and unions, it is likely that any query that contains unions will produce larger r-graphs than those without any unions. We also see that `OPTIONAL` is an expensive feature, particularly for the Geo dataset. However, looking into the Geo results

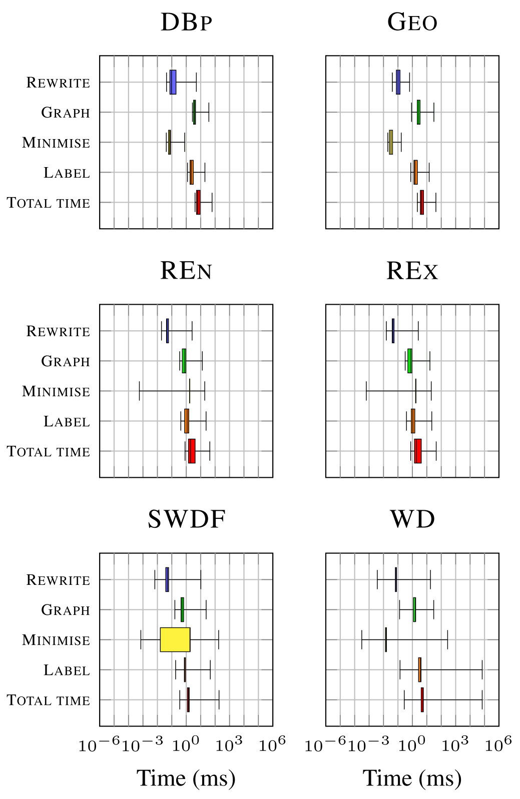


Fig. 7. Runtimes for each step of the canonicalisation algorithm (considering queries with some SPARQL 1.1 feature).

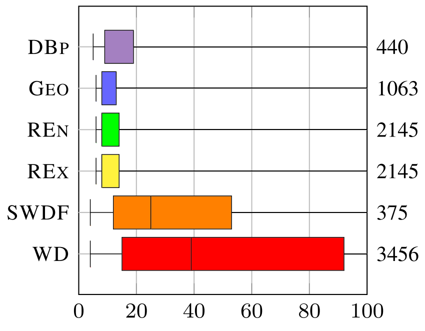


Fig. 8. R-graph sizes (number of nodes) for each set of queries; the value on the right indicates the maximum size.

in more detail, most of the queries with `OPTIONAL` appear to come from a single agent – an application using the SPARQL endpoint as a backend – and also feature numerous disjunctive filters (using `bound(·)` over multiple `OPTIONAL` clauses) that are rewritten into `UNION`, which is then the underlying reason that the queries are slower. This result – relating to the requests of a single prolific agent – can thus be considered something of an outlier.

8.2.2. *Duplicates found*

We now look at the number of duplicates found, where we compare the following methods:

Raw The raw query string is used, without any kind of normalisation.

Parse The raw query string is parsed using Jena ARQ into its query algebra, and then serialised back into concrete SPARQL syntax.

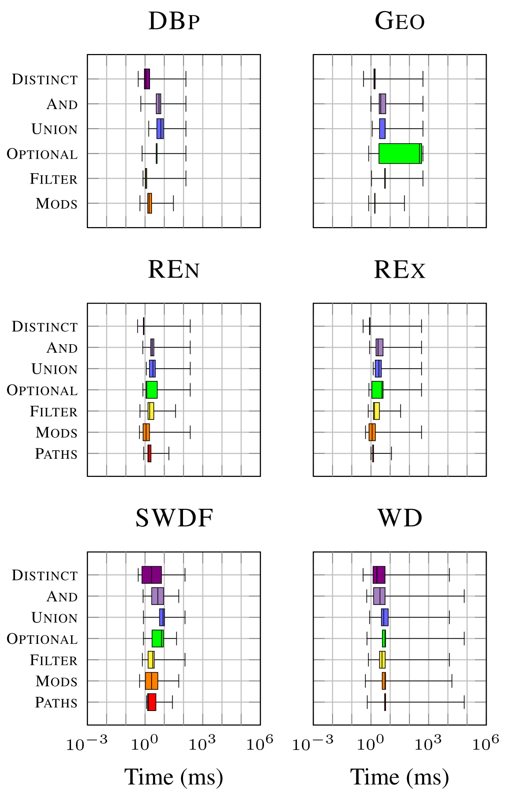


Fig. 9. Runtimes for queries grouped by selected features.

Table 25

Total number of duplicates found by each method

Query set	RAW	PARSE	LABEL	REWRITE	FULL
DBP	250,940	251,283	251,315	251,315	251,315
GEO	723,116	736,331	739,695	739,700	739,702
REN	142,032	143,523	144,007	144,007	144,008
REX	299,892	301,419	301,910	301,910	301,911
SWDF	53,061	53,263	53,388	53,388	53,388
WD	683,132	686,453	687,654	687,751	687,760

LABEL The raw query string is parsed using Jena ARQ into its query algebra, the r-graph is constructed and canonically labelled, and then serialised back into concrete SPARQL syntax.

REWRITE The raw query string is parsed using Jena ARQ into its query algebra, the query is rewritten per the algebraic rules, the r-graph is constructed and canonically labelled, and then serialised back into concrete SPARQL syntax.

FULL The raw query string is parsed using Jena ARQ into its query algebra, the query is rewritten per the algebraic rules, the r-graph is constructed, minimised, canonically labelled, and then serialised back into concrete SPARQL syntax.

Tables 25 and 26 denote the total number of duplicate (congruent) queries found, and the most duplicates found for a single query. In general, there is a high number of exact duplicate query strings, possibly from the same query being sent many times to refresh the results. Thereafter, the number of duplicates found either remains the same or increases in each successive algorithm. In particular, excluding duplicate query strings (RAW), the highest increase occurs with PARSE and thereafter LABEL, with WD being the query set for which the most additional duplicates

Table 26

Most duplicates of a single query found by each method

Query set	RAW	PARSE	LABEL	REWRITE	FULL
DBP	5,464	5,514	5,514	5,514	5,514
GEO	22,582	31,379	40,744	40,744	40,744
REN	3,814	3,814	3,814	3,814	3,814
REX	14,690	14,910	14,910	14,910	14,910
SWDF	2,388	2,633	4,938	4,938	4,938
WD	232,339	232,339	232,339	232,339	232,339

Table 27

UCQ features supported by SPARQL Algebra (SA), Alternating Free two-way μ-calculus (AFMU), Tree Solver (TS) and Jena-SPARQL-API Graph-isomorphism (JSAG); note that ABGP denotes Acyclic Basic Graph Patterns

Method	ABGP	BGP	Projection	Union
SA	✓	✓		
AFMU	✓		✓	✓
TS	✓		✓	✓
JSAG	✓	✓	✓	✓
QCan	✓	✓	✓	✓

are found with these methods, where the duplicates detected increase by a few thousand in each step. In the other query sets this increase is less pronounced. In addition, there is almost no difference beyond LABEL, meaning that algebraic rewritings and minimisation find very few additional congruent queries in these logs.

8.3. Comparison with existing systems

In this section, we compare the runtime performance of our algorithm with existing systems that can perform pairwise SPARQL containment checks. We provide an overview of these tools in Table 27. In our experiments, we consider SA [36] and JSAG [58] as they support cyclic queries. The queries used are part of the test suite designed by Chekol et al. [13] as part of their experiments. These correspond to two sets of queries: one of CQs without projection, and one of UCQs with projection. As discussed in Section 5, the SA and JSAG systems are not analogous to ours. We focus on query equivalence and congruence, and not on containment; conversely, SA and JSAG support containment. On the other hand, we compute a canonical form of a query, whereas SA and JSAG focus on pairwise checks (though JSAG offers indexing approaches based on constants in the query and isomorphic DAGs). Our approach is sound and complete for UCQs under both bag and set semantics; conversely, SA only considers set semantics, while JSAG focuses on detecting sub-graph isomorphisms between algebraic expressions under bag semantics. In the case of CQs without projection, checking containment is tractable (see Table 16), and quite trivial, requiring checking that one BGP is set-contained in the other.

Figure 10 shows the runtimes for our comparison of both containment checkers and our method (QCan). Note that there are no values for SA with UCQs because the UCQ set uses projection and SA does not support queries with projection. The results indicate that most queries for SA and JSAG take between one and ten milliseconds, whereas most queries under our method take between ten and one hundred milliseconds. In terms of the slowest queries, our method is faster than JSAG but slower than SA.

In general, the conclusion is that our method is slower for testing equivalence than existing containment checkers, but this is perhaps not surprising as our approach addresses the more difficult problem of first computing a canonical form for both input queries, and further considers congruence rather than the more trivial case of equivalence where variable names are fixed. Furthermore, once these canonical forms are computed, equivalence classes of congruent queries can be found in constant time using simple data structures (e.g., (ideal) hashing of the canonical query strings). If we estimate that our system is 10 times slower to canonicalise two queries than these systems can perform a pairwise check (per Fig. 10), QCan will begin to outperform these systems for partitioning a set of 11 or more queries by equivalence (or congruence); in the case of 11 queries, these systems must perform $\binom{11}{2} = 55$

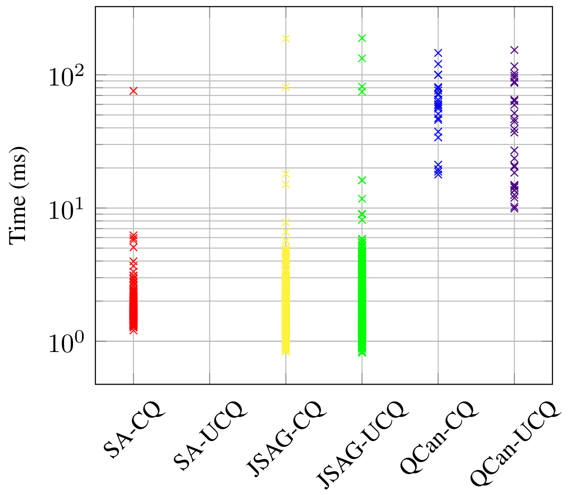


Fig. 10. Runtimes for JSAG, SA and QCan.

pairwise checks (in the worst case), while QCan will canonicalise 11 queries and partition them in constant time (in the best and the worst case). The time savings thereafter grow quadratically.

8.4. *Stress test*

With the exception of some outliers, most queries that we have evaluated thus far have been canonicalised in fractions of a second. On the other hand, we know that our algorithms are super-exponential in the worst case. Such cases may occur when we have monotone queries that are in conjunctive normal form (i.e., consisting of joins of unions), in which case our UCQ normal form can be exponential in size, upon which we perform further potentially exponential processes. In order to stress-test our method and explore such worst cases, we run the following experiment.

For this experiment, we generate queries of the form $\text{AND}(U_1, \ldots, U_m)$ where each union pattern is of the form $U_i = \text{UNION}(t_1, \ldots, t_k)$, and each triple pattern is of the form $t_j = (s_j, \texttt{:p}_j, o_j)$, where s_j and o_j are variables randomly selected from a predetermined set of $m + k$ variables, and $\texttt{:p}_j$ is an IRI randomly selected from a predetermined set of m IRIs. The UCQ normal form for this query will consist of a union of k^m BGPs, each containing m triple patterns. Finally, we project a random subset of the set of variables that appear in the query, making sure to project at least one variable.

Figure 11 shows the times for each step of the canonicalisation procedure on the synthetic UCQs. On the x-axis we increase k (the base of the exponent), while across the different series we increase m (the exponent). The y-axis is shown in log scale. We see that for the UCQ rewriting, graph construction and minimisation steps, the higher series (representing an increasing exponent) diverge further and further as the x-axis increases (representing an increasing base). On the other hand, the differences in times for canonical labelling are less pronounced since the minimisation process reduces the r-graphs significantly due to the regular construction of the queries. The slowest queries tested ($k = 9$, $m = 4$) take around 4.1 hours to canonicalise considering all steps. Increasing k and/or m further would quickly lead to unmanageable runtimes and, eventually, out-of-memory exceptions.

These results illustrate the poor worst-case behaviour of our canonicalisation procedure, particularly when considering queries with joins of many unions. However, as shown by the results presented in Section 8.2, virtually no queries in our real-world setting caused this worst-case behaviour.

9. Conclusion

In this paper, we have presented a formal semantics of SPARQL 1.1 and a canonicalisation procedure that produces a deterministic query string modulo congruence (equivalence modulo variable naming). This procedure in-

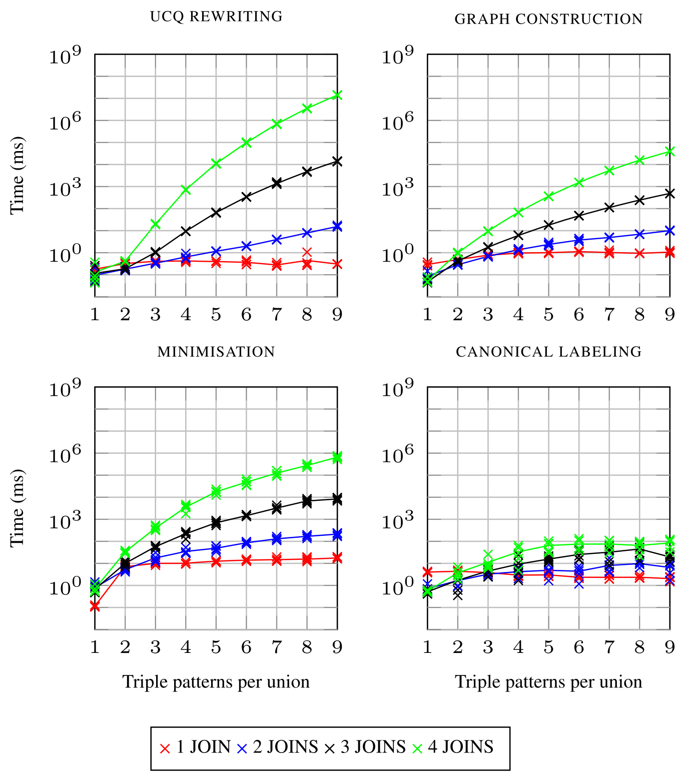


Fig. 11. Times for UCQ stress tests.

volves the application of algebraic rewritings of the query, the representation of the query as a graph, the minimisation of the query in order to remove redundancy, and finally the canonical labelling of the graph in order to produce canonical variable names based on the structure of the graph. We have proven this procedure to be sound and complete with respect to “extended monotone queries” under bag and set semantics, i.e., queries that can be rewritten to the features involving BGPs, joins, unions, projection, and distinct. We have further extended this procedure to provide sound and incomplete canonicalisation of the queries under the full SPARQL 1.1 language.

Despite the super-exponential worst-case complexity of our procedure, the experimental results indicate that our method is efficient for most queries, running in a fraction of a second – in the median case – over millions of queries from real-world logs; the slowest query to canonicalise took just over a minute. Such results are achieved because most real-world queries tend to be relatively small and simple. In this analysis, we determined that the canonical labelling is the step that takes the longest time on average. We further found that the `UNION` feature is the most costly to canonicalise in general, with `OPTIONAL` also proving costly in some cases. Comparing the performance of our method for finding equivalent queries versus existing containment checkers, we find that our method is indeed slower, but produces a canonical form that thereafter allows for constant-time detection of congruent queries in large sets of queries. Running stress-tests over queries featuring joins of unions of increasing size, we have confirmed that our procedure quickly becomes inefficient, taking hours to canonicalise four joins of unions with nine triple patterns each.

We have further confirmed that our procedure allows for detecting additional congruent queries over real-world logs versus baseline methods. We first observed that the vast majority of congruent queries are actually syntactically identical in terms of raw query strings, likely due to clients reissuing precisely the same query in order to intermittently refresh results. Of our canonical procedures, canonical labelling is the most important for finding additional congruent queries. On the other hand, minimisation and algebraic rewritings – though necessary to ensure completeness for monotone queries – lead to finding only a very small fraction of additional congruent queries. This would tend to suggest that in a practical caching system, where completeness can be traded for efficiency, it may be sufficient to apply canonical labelling without algebraic rewritings and minimisation. However, minimisation may be useful in cases where completeness is an important criterion. Also, in certain setting, queries with redundancy may be automatically generated; an interesting use-case along these lines is that of ontology-based data access (OBDA) [63], where rewritings may produce queries (typically UCQs) with redundancies that are inefficient to evaluate over large graphs.

With respect to future directions, a natural continuation of this work is to explore larger fragments of SPARQL 1.1 for which sound and complete canonicalisation can be performed. In particular, we have already begun to investigate such procedures for a fragment akin to UC2RPQs. At first glance, this should be analogous to the minimisation and canonicalisation of basic graph patterns and unions, where property paths are represented as automata and we can check for containment of automata to find redundant path patterns. However, we have already found that this requires entailment of paths with inverses, which is not as straightforward as checking for containment [34].

SPARQL 1.2 is on the horizon. We believe that the formal semantics of SPARQL 1.1 defined herein may serve as a useful reference for the standardisation effort. If the semantics of `FILTER (NOT) EXISTS` is clarified as part of SPARQL 1.2, it would also be of interest to (partially) capture equivalences between the negation features of SPARQL. With respect to new features, our canonicalisation procedure should be extensible to SPARQL 1.2 following similar processes, though such canonicalisation cannot be complete.

Finally, we are currently exploring use-cases for our canonicalisation procedure in the context of two ongoing projects: one relating to caching, and one relating to query answering. Regarding caching, we have seen that most congruences in real-world query logs are exact (syntactic) duplicates; however, rather than considering congruencies between full queries, a more promising approach for caching is to mine common sub-queries, where canonicalisation can be used for such purposes. In the context of question answering, we can also use canonicalisation in order to normalise training data for sequence-to-sequence models that translate natural language to SPARQL queries. An important future line of research is then to explore and evaluate the benefits of SPARQL canonicalisation in the context of these and further use-cases.

Online material We provide code and other material online at: http://github.com/RittoShadow/QCan. A demo is available at: http://qcan.dcc.uchile.cl.

Acknowledgements

This work was supported by Fondecyt Grant No. 1181896 and by ANID – Millennium Science Initiative Program – Code ICN17_002. We would also like to thank the reviewers of the conference and journal versions of this paper for useful comments that helped to improve this work.

Appendix. Proofs

A.1. Proofs for Section 6

Herein we provide the proofs for the lemmas and theorems enumerated in the body of the paper.

A.1.1. Proof for Lemma 6.1

First assume that Q does not contain a literal subject. Define G to be the result of replacing all variables in Q with an IRI, say `:x`. Observe that G is a valid RDF graph. Define D to be the dataset with the default graph G. Let μ denote the solution such that $\mathrm{dom}(\mu) = \mathrm{vars}\, Q$ and $\mu(v) =$ `:x` for all $v \in \mathrm{dom}(\mu)$. Observe that $\mu \in Q(D)$. Hence there exists a dataset D such that $Q(D)$ is non-empty: Q is satisfiable.

Next assume that Q contains a literal subject. Take any blank node mapping α and solution mapping μ. Observe that $\mu(\alpha(Q))$ will still contain a literal subject, and hence for any RDF graph G, blank node mapping α and solution mapping μ, it holds that $\mu(\alpha(Q)) \nsubseteq G$ as G cannot contain a triple with a literal subject. Hence Q cannot have any solution over any dataset, Q is unsatisfiable, and the result holds.

A.1.2. Proof for Lemma 6.2

Recall: $\text{UNION}(Q_1, \dots, Q_n)(D) = \bigcup_{i=1}^{n} Q_i(D)$ (with union being bag or set union depending on the semantics). The result is (always) empty if and only if all $Q_i(D)$ are (always) empty. Removing any Q_k such that $Q_k(D)$ is (always) empty will not affect the results of the query. Thus the result holds.

A.1.3. Proof for Lemma 6.3

Take any Q_i ($1 \leqslant i \leqslant n$). Given any dataset D, for any solution $\mu \in Q_i(D)$, there must exist a corresponding solution $\lambda_i(\mu) \in \lambda_i(Q_i)(D)$ with the same multiplicity, where $\text{dom}(\lambda_i(\mu)) = \lambda_i(\text{dom}(\mu))$, and $\lambda_i(\mu)(\lambda_i(v)) = \mu(v)$, i.e., a solution that is the same but with variables renamed per λ_i. Furthermore, $\text{SELECT}_V(Q_i)(D) = \text{SELECT}_V(\lambda_i(Q_i))(D)$ as by definition λ_i does not rewrite variables in V. We now have:

$$
\begin{aligned}
&\text{SELECT}_V\big(\text{UNION}(Q_1, \dots, Q_n)\big)\\
&\quad\equiv \text{UNION}\big(\text{SELECT}_V(Q_1), \dots, \text{SELECT}_V(Q_n)\big)\\
&\quad\equiv \text{UNION}\big(\text{SELECT}_V\big(\lambda_1(Q_1)\big), \dots, \text{SELECT}_V\big(\lambda_n(Q_n)\big)\big)\\
&\quad\equiv \text{SELECT}_V\big(\text{UNION}\big(\lambda_1(Q_1), \dots, \lambda_n(Q_n)\big)\big)
\end{aligned}
$$

which concludes the proof.

A.1.4. Proof for Lemma 6.4

Let $V = \text{pvars}(Q)$ denote the possible variables of Q; then, for any dataset D, if $\mu \in Q(D)$ then $\text{dom}(\mu) \subseteq V$. For each solution $\mu \in Q(D)$, the projection $\text{SELECT}_{V'}(Q)(D)$ produces a solution μ' such that $\text{dom}(\mu') \subseteq V' \cap \text{dom}(\mu)$ and $\mu \sim \mu'$, while the projection $\text{SELECT}_{V' \cup V''}(Q)$ produces a solution μ'' such that $\text{dom}(\mu'') = (V' \cup V'') \cap \text{dom}(\mu)$ and $\mu \sim \mu''$. But since $\text{dom}(\mu) \subseteq V$ and $V \cap V'' = \emptyset$, this means that $\text{dom}(\mu) \cap V'' = \emptyset$. Hence $\text{dom}(\mu'') = (V' \cup V'') \cap \text{dom}(\mu) = V' \cap \text{dom}(\mu) = \text{dom}(\mu')$. Further given that $\mu \sim \mu'$, $\mu \sim \mu''$, $\text{dom}(\mu') = \text{dom}(\mu'') \subseteq \text{dom}(\mu)$, we can conclude that $\mu' = \mu''$. We have thus shown a one-to-one mapping from solutions of the form μ' to μ'' such that $\mu' = \mu''$, and thus the result holds.

A.1.5. Proof for Lemma 6.5

First we prove that if $\text{vars}\, Q = V$ and $\text{bnodes}(Q) = \emptyset$, then $\text{DISTINCT}(\text{SELECT}_V(Q)) \equiv \text{SELECT}_V(Q)$. It suffices to show that $\text{SELECT}_{\text{vars}\, Q}(Q)$ cannot produce duplicate results, which we will now prove by contradiction. Assume a dataset D such that there exists a solution μ where $\text{SELECT}_{\text{vars}\, Q}(Q)(D)(\mu) > 1$; i.e., the solution appears more than once. Since all variables are projected, $\text{SELECT}_{\text{vars}\, Q}(Q) \equiv Q$. For μ to be duplicated, there must then exist multiple blank node mappings α such that $\text{dom}(\alpha) = \text{bnodes}(Q)$ and $\mu(\alpha(Q)) \subseteq D$ (see Table 8), but since $\text{bnodes}(Q) = \emptyset$, there is only the single empty mapping α, and hence we have a contradiction. The case $\text{vars}\, Q \subsetneq V$ follows from this result and Lemma 6.4.

Next we prove that if $\text{vars}\, Q \not\subseteq V$ or $\text{bnodes}(Q) \neq \emptyset$, and Q is satisfiable, then $\text{DISTINCT}(\text{SELECT}_V(Q)) \not\equiv \text{SELECT}_V(Q)$. It suffices to show that there exists a dataset D for which $\text{SELECT}_V(Q)$ can produce duplicates if $V \neq \text{vars}\, Q$ or $\text{bnodes}(Q) \neq \emptyset$. Let μ denote a solution mapping such that $\text{dom}(\mu) = V$ and for all $v \in \text{dom}(\mu)$, $\mu(v) = \texttt{:x}$. Let μ' and μ'' denote two solution mappings such that $\text{dom}(\mu') = \text{dom}(\mu'') = \text{vars}\, Q \setminus V$ and for all $v \in \text{vars}\, Q \setminus V$, $\mu'(v) = \texttt{:y}$ and $\mu''(v) = \texttt{:z}$. Let α' and α'' denote two blank node mappings such that $\text{dom}(\alpha') = \text{dom}(\alpha'') = \text{bnodes}(Q)$ and for all $b \in \text{bnodes}(Q)$, $\alpha'(b) = \texttt{:y}$ and $\alpha''(b) = \texttt{:z}$. Consider a dataset D whose default graph is defined as $\mu(\mu'(\alpha'(Q))) \cup \mu(\mu''(\alpha''(Q)))$; this is a valid RDF graph as Q is satisfiable and thus does not contain literal subjects. We see that $\text{SELECT}_V(Q)(D)(\mu) \geqslant 2$, where the mapping to $\texttt{:y}$ and the mapping to $\texttt{:z}$ (be they blank node mappings or solution mappings) are counted in the multiplicity of μ. Thus there exists a dataset D for which $\text{SELECT}_V(Q)$ can produce duplicate solutions, which concludes the proof.

A.1.6. Proof for Lemma 6.6

As before, we first prove that if blank nodes are not present ($\text{bnodes}(Q) = \emptyset$), all variables are projected ($\text{vars}\, Q \subseteq V$) and no BGP has the same set of variables ($\text{vars}\, Q_i \neq \text{vars}\, Q_j$ for all $1 \leqslant i < j \leqslant n$), then $\text{SELECT}_V(\text{UNION}(Q_1, \dots, Q_n))$ cannot produce duplicate results. We know from Lemma 6.5 that if blank nodes are not present and all variables are projected, then no individual BGP in $Q_1, \dots, Q_n$ can produce duplicate results.

Hence we are left to check for duplicates produced by unions of BGPs. We will assume for the purposes of contradiction that there exists a dataset D and a solution μ such that $\mu \in Q_i(D)$ and $\mu \in Q_j(D)$ where $1 \leqslant i < j \leqslant n$. However, $\mu \in Q_i(D)$ implies that $\mathrm{dom}(\mu) = \mathrm{vars}\, Q_i$, while $\mathrm{dom}(\mu) = \mathrm{vars}\, Q_j$ implies that $\mathrm{dom}(\mu) = \mathrm{vars}\, Q_j$. Given the assumption that $\mathrm{vars}\, Q_i \neq \mathrm{vars}\, Q_j$, it follows that $\mathrm{dom}(\mu) \neq \mathrm{dom}(\mu)$: a contradiction. It then holds as a consequence that if $\mathrm{vars}\, Q = V$, $\mathrm{bnodes}(Q) = \emptyset$ and $\mathrm{vars}\, Q_i \neq \mathrm{vars}\, Q_j$ for all $1 \leqslant i < j \leqslant n$, then $\mathrm{DISTINCT}(\mathrm{SELECT}_V(\mathrm{UNION}(Q_1, \dots, Q_n))) \equiv \mathrm{SELECT}_V(\mathrm{UNION}(Q_1, \dots, Q_n))$. The special case of $\mathrm{vars}\, Q \subsetneq V$ follows from this result and Lemma 6.4.

In the other direction, we are left to show that if $\mathrm{vars}\, Q \nsubseteq V$, or $\mathrm{bnodes}(Q) \neq \emptyset$, or there exist $1 \leqslant i < j \leqslant n$ such that $\mathrm{vars}\, Q_i = \mathrm{vars}\, Q_j$, then it follows that $\mathrm{DISTINCT}(\mathrm{SELECT}_V(\mathrm{UNION}(Q_1, \dots, Q_n))) \not\equiv \mathrm{SELECT}_V(\mathrm{UNION}(Q_1, \dots, Q_n))$. First, if $\mathrm{vars}\, Q \nsubseteq V$ or $\mathrm{bnodes}(Q) \neq \emptyset$, then Lemma 6.5 tells us that an individual basic graph pattern with a blank node or non-projected variable can produce duplicates. Hence we assume that $\mathrm{vars}\, Q \subseteq V$ and $\mathrm{bnodes}(Q) = \emptyset$, and show that if there exists $1 \leqslant i < j \leqslant n$ such that $\mathrm{vars}\, Q_i = \mathrm{vars}\, Q_j$, then duplicates can always arise for the query $\mathrm{SELECT}_V(\mathrm{UNION}(Q_1, \dots, Q_n))$. Let μ be a solution such that $\mathrm{dom}(\mu) = \mathrm{vars}\, Q_i = \mathrm{vars}\, Q_j$ and $\mu(v) = \texttt{:x}$ for all $v \in \mathrm{dom}(\mu)$. Consider a dataset D whose default graph is defined as $\mu(Q_i) \cup \mu(Q_j)$; again this is an RDF graph as Q_i and Q_j are assumed to be satisfiable. Now $\mu \in Q_i(D)$ and $\mu \in Q_j(D)$, and since all variables are projected, we conclude that μ will be duplicated in $\mathrm{SELECT}_V(\mathrm{UNION}(Q_1, \dots, Q_n))(D)$. The result holds.

A.1.7. *Proof for Lemma 6.7*

This follows from the fact that each step preserves the congruence of Q, as follows:

1. property path elimination: by definition, Table 9;
2. union normalisation: proven by Pérez et al. [44];
3. unsatisfiability normalisation: Lemmas 6.1, 6.2;
4. variable normalisation: Lemmas 6.3, 6.4;
5. set vs. bag normalisation: Lemmas 6.5, 6.6.

Since congruence is an equivalence relation, it is transitive, and thus the composition of multiple steps where each preserves congruence also preserves congruence. The result thus holds.

A.1.8. *Proof for Lemma 6.8*

By definition, we have that $\mathrm{R}^-(\mathrm{R}(Q)) \simeq Q$, where $\mathrm{R}^-(\cdot)$ relies on a one-to-one mapping of blank nodes to variables (namely ξ). Since $\mathrm{L}(\cdot)$ performs a one-to-one mapping of blank nodes to blank nodes in $\mathrm{R}(Q)$, thus producing an isomorphic graph to $\mathrm{R}(G)$, we can conclude that $\mathrm{R}^-(\mathrm{L}(\mathrm{R}(Q)))$ produces a query that is isomorphic to $\mathrm{R}^-(\mathrm{R}(Q))$. Hence we have that $\mathrm{R}^-(\mathrm{L}(\mathrm{R}(Q))) \simeq \mathrm{R}^-(\mathrm{R}(Q)) \simeq Q$. Further given that isomorphism implies congruence, we have that $\mathrm{R}^-(\mathrm{L}(\mathrm{R}(Q))) \cong \mathrm{R}^-(\mathrm{R}(Q)) \cong Q$. The result then holds per the transitivity of congruence.

A.1.9. *Proof for Lemma 6.9*

We consider two cases.

SET SEMANTICS: Minimising CQs by computing their cores is a classical technique based on the idea that two CQs are equivalent if and only if they are homomorphically equivalent (with corresponding projected variables [11]). Likewise the minimisation of UCQs is covered by Sagiv and Yannakakis [49], who (unlike in the relational algebra but analogous to SPARQL) allow UCQs with existential variables; however, their framework assumes that each CQ covers all projected variables. Hence the only gap that remains is the minimisation of SPARQL UCQs where BGPs may not contain all projected variables. This result is quite direct since for a set of variables V, two BGPs Q_1 and Q_2 such that $\mathrm{vars}\, Q_1 \cap V \neq \mathrm{vars}\, Q_2 \cap V$, and any dataset D, it holds that $\mathrm{SELECT}_V(Q_1)(D) \cap \mathrm{SELECT}_V(Q_2)(D) = \emptyset$ since for any solution $\mu_1 \in \mathrm{SELECT}_V(Q_1)(D)$ it holds that $\mathrm{dom}(\mu_1) = \mathrm{vars}\, Q_1 \cap V$, while for any solution $\mu_2 \in \mathrm{SELECT}_V(Q_2)(D)$ it holds that $\mathrm{dom}(\mu_2) = \mathrm{vars}\, Q_2 \cap V$, and $\mathrm{vars}\, Q_1 \cap V \neq \mathrm{vars}\, Q_2 \cap V$. Hence, checking containment within partitions of BPGs formed by the projected variables they contain does not miss containments. The result for set semantics then follows from Sagiv and Yannakakis [49].

BAG SEMANTICS: UCQs are not minimised; thus the result follows directly from $\mathrm{R}^-(\mathrm{R}(Q)) \cong Q$.

A.1.10. Proof for Theorem 6.1

The result holds as a direct corollary of Lemmas 6.7, 6.8, 6.9, and the transitivity of congruence, which is an equivalence relation.

A.1.11. Proof for Lemma 6.10

If Q_1 and Q_2 are unsatisfiable, then per Lemmas 6.1, 6.2, $\textsc{u}(Q_1) = Q_\emptyset$ and $\textsc{u}(Q_2) = Q_\emptyset$, recalling that $Q_\emptyset$ denotes the canonical unsatisfiable query. Thus $\textsc{u}(Q_1) = \textsc{u}(Q_2)$ and the result holds per premise 2 of Remark 6.1 since equality implies isomorphism.

A.1.12. Proof for Lemma 6.11

Since Q'_1 and Q'_2 are satisfiable, and evaluated under set semantics, we know from the result of Chandra and Merlin [11] that $Q'_1 \equiv Q'_2$ if and only if both are homomorphically equivalent with respect to homomorphisms that are the identity on projected variables. Now $\textsc{m}(\textsc{r}(\textsc{u}(Q'_1)))$ computes the core of each BGP, which is known to be unique modulo isomorphism [22]. Thus if $Q'_1 \equiv Q'_2$ and both are satisfiable, we have that $\textsc{m}(\textsc{r}(\textsc{u}(Q'_1))) \simeq \textsc{m}(\textsc{r}(\textsc{u}(Q'_1)))$. On the other hand, if $Q'_1 \cong Q'_2$, we know that there exists a variable renaming such that $\rho(Q'_1) \equiv Q'_2$; combining this with the fact that $\textsc{m}(\textsc{r}(\textsc{u}(Q'_1))) \simeq \textsc{m}(\textsc{r}(\textsc{u}(\rho(Q'_1))))$, and the fact that congruence is an equivalence relation, we know that $Q'_1 \cong Q'_2$ implies $\textsc{m}(\textsc{r}(\textsc{u}(Q'_1))) \simeq \textsc{m}(\textsc{r}(\textsc{u}(Q'_2)))$. The result then holds from premise 4 of Remark 6.1.

A.1.13. Proof for Lemma 6.12

Given a satisfiable BGP Q, and $\textsc{select}_V(Q)$ such that $V \setminus \text{vars}\,Q \neq \emptyset$, then $\textsc{u}(\textsc{select}_V(Q))$ will remove the unbound variables $(V \setminus \text{vars}\,Q)$ from V per Lemma 6.4, and thus $\textsc{u}(\textsc{select}_V(Q)) = \textsc{u}(\textsc{select}_{V\cap\text{vars}\,Q}(Q))$. As part of $\textsc{u}(Q)$, during union normalisation, blank nodes are rewritten to variables. This leaves us with cases where $V \subseteq \text{vars}\,Q$ and Q does not contain blank nodes.

If $V_1 = \text{vars}\,Q_1$, then Q'_1 cannot return duplicates (Lemma 6.5), and since $Q'_1 \cong Q'_2$, then Q'_2 cannot return duplicates, and thus $V_2 = \text{vars}\,Q_2$. Thus $\textsc{u}(Q'_1)$ and $\textsc{u}(Q'_2)$ will add distinct in both cases, and the result follows from Lemma 6.11.

This leaves us with the case that $V_1 \subsetneq \text{vars}\,Q_1$. In this case, Q'_1 will return duplicates for certain datasets (Lemma 6.5), and must be evaluated under bag semantics. Given that $Q'_1 \cong Q'_2$, this likewise means that Q'_2 returns duplicates. Under bag semantics, and assuming that Q'_1 and Q'_2 are satisfiable, then Theorem 5.2 of Chaudhuri and Vardi [12] tells us that $Q'_1 \equiv Q'_2$ if and only if $Q'_1 \simeq Q'_2$ with an isomorphism that is the identity on projected variables. Noting that $Q'_1 \cong Q'_2$ implies that there exists a variable renaming ρ such that $\rho(Q'_1) \equiv Q'_2$, it follows that there exists ρ such that $\rho(Q'_1) \simeq Q'_2$, or put more simply, that $Q'_1 \simeq Q'_2$ (without the restriction on projected variables). The result then follows from premise 1 of Remark 6.1.

A.1.14. Proof for Lemma 6.13

We show that given a satisfiable UCQ Q evaluated under set semantics, the minimisation function $\textsc{m}(\textsc{r}(\textsc{u}(Q)))$ will produce an r-graph corresponding to a minimal UCQ that is unique, modulo isomorphism, for the set of UCQs congruent to Q. The minimisation of CQs (i.e., unary UCQs) is covered by Lemma 6.11. The minimisation of (non-unary) UCQs is based on Corollary 4 of Sagiv and Yannakakis [49], where we minimise the UCQ while maintaining this equivalence relation, more specifically, such that each BGP in the input UCQ will be contained in some BGP of the output UCQ (with a containment homomorphism that is the identity on projected variables). Note that this minimisation includes the removal of all unsatisfiable BGPs (per Lemma 6.2; if all are removed, then Lemma 6.10 applies as the UCQ is unsatisfiable). There are, however, two non-deterministic elements to consider in this minimisation:

- The containment only considers projected variables as fixed. Hence the naming of other variables (and blank nodes) in BGPs does not matter. However, this issue is resolved prior to minimisation by $\textsc{u}(\cdot)$, which maps blank nodes to fresh (non-projected) variables, and then renames non-projected variables in each BGP to fresh variables, per Lemma 6.3, such that the naming of variables is deterministic modulo isomorphism.
- We non-deterministically choose one BGP from each quotient set of equivalent BGPs with the same projected variables. However, since BGPs were previously minimised, all equivalent BGPs are isomorphic, and hence the choice is deterministic modulo isomorphism.

Thus, given a satisfiable UCQ Q evaluated under set semantics, $\textsc{m}(\textsc{r}(\textsc{u}(Q)))$ will produce an r-graph corresponding to a minimal UCQ that is unique, modulo isomorphism, for the set of UCQs congruent to Q. Returning to the claim, we note that Q_1 and Q_2 are satisfiable UCQs evaluated under set semantics (with distinct), that $Q_1 \cong Q_2$, and thus we have that $\textsc{m}(\textsc{r}(\textsc{u}(Q_1))) \simeq \textsc{m}(\textsc{r}(\textsc{u}(Q_2)))$: the result holds per premise 4 of Remark 6.1.

A.1.15. Proof for Lemma 6.14

Under bag semantics, $\textsc{u}(\cdot)$ removes all unsatisfiable operands from the UCQ and $\textsc{m}(\cdot)$ acts as the identity (no minimisation is applied). Per the results for CQs, any minimisation of BGPs (aside from the implicit removal of duplicate triple patterns) will reduce the multiplicity of results on some datasets. Likewise removing satisfiable BGPs will reduce the multiplicities for any dataset where the removed BGP generates solutions. This leaves one source of non-determinism (the same as in the case for set semantics) per Lemma 6.3: that non-projected variables across BGPs may have the same label whereas the query is equivalent if they have distinct labels. As before for set semantics, $\textsc{u}(\cdot)$, maps blank nodes to fresh (non-projected) variables in the case of bag semantics. This implies that if $Q_1 \cong Q_2$ then – letting Q'_1 and Q'_2 denote the result of removing unsatisfiable BGPs from Q_1 and Q_2 and distinguishing variables per Lemma 6.3 – $Q'_1 \simeq Q'_2$.

There are then two cases to consider:

1. Either Q_1 or Q_2 cannot return duplicates, per the conditions of Lemma 6.6, but given that $Q_1 \cong Q_2$, then this implies that both Q_1 and Q_2 cannot return duplicates, which means that both satisfy the conditions of Lemma 6.6, and thus both will have distinct invoked by $\textsc{u}(\cdot)$.
2. Both Q_1 and Q_2 may return duplicates, i.e., they do not satisfy the conditions of Lemma 6.6, and in neither case will distinct be invoked by $\textsc{u}(\cdot)$.

Thus it holds that $\textsc{u}(Q_1) \simeq \textsc{u}(Q_2)$, satisfying premise 2 of Remark 6.1, and the result follows.

A.1.16. Proof for Lemma 6.15

Since Q' cannot return duplicates and $Q \cong Q'$, it holds that Q cannot return duplicates, and hence Q must satisfy the conditions of Lemma 6.5. Thus $\textsc{u}(Q)$ will add distinct, and we have that $\textsc{u}(Q) \simeq \textsc{u}(Q')$. The result then follows per premise 2 of Remark 6.1.

A.1.17. Proof for Theorem 6.2

First we remark that for any EMQ Q, the first steps of $\textsc{u}(Q)$ – property path and union normalisation – yield a UCQ. We denote by Q'_1 and Q'_2 the UCQs derived from Q_1 and Q_2. We now consider the cases:

1. If Q'_1 or Q'_2 are unsatisfiable, then both are unsatisfiable, and the result holds from Lemma 6.10.
2. Otherwise (Q'_1 and Q'_2 are satisfiable):
   (a) If Q'_1 and Q'_2 both use distinct, the result holds from Lemma 6.13.
   (b) If neither Q'_1 nor Q'_2 use distinct, the result holds from Lemma 6.14.
   (c) If Q'_1 uses distinct, and Q'_2 does not, then $Q'_1 \cong Q'_2$ implies that Q'_2 cannot produce duplicates (since Q'_1 cannot). From this it follows that $Q'_1 \cong Q'_2 \cong \textsc{distinct}(Q'_2)$. Now given that $Q'_1 \cong \textsc{distinct}(Q'_2)$, it follows from Lemma 6.13 that $\textsc{r}^-(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q'_1))))) = \textsc{r}^-(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(\textsc{distinct}(Q'_2))))))$ (noting that Q'_1 and $\textsc{distinct}(Q'_2)$ use distinct). Further given that $Q'_2 \cong \textsc{distinct}(Q'_2)$, it follows from Lemma 6.15 that $\textsc{r}^-(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q'_2))))) = \textsc{r}^-(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(\textsc{distinct}(Q'_2))))))$. We have $\textsc{r}^-(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q'_1))))) = \textsc{r}^-(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q'_2)))))$.
   (d) Otherwise, if Q'_1 does not use distinct, and Q'_2 uses distinct, the result follows from the previous case and the symmetry of congruence.

This concludes the proof.

A.1.18. Proof for Theorem 6.3

Let Q'_1 denote $\textsc{r}^-(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q_1)))))$, and Q'_2 denote $\textsc{r}^-(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q_2)))))$.

$Q'_1 = Q'_2$ implies $Q_1 \cong Q_2$: follows from Theorem 6.1 (soundness), which tells us that $Q_1 \cong Q'_1$ and $Q'_2 \cong Q_2$, from which we have that $Q_1 \cong Q'_1 = Q'_2 \cong Q_2$, and thus that $Q_1 \cong Q_2$ by transitivity.

$Q_1 \cong Q_2$ implies $Q'_1 = Q'_2$: is given in Theorem 6.2 (completeness).

A.2. Proofs for Section 7

A.2.1. Proof for Theorem 7.1

The result holds from observing that given an EMQ Q, each step of the process returns precisely the same result as in the case of monotone canonicalisation. In particular, $\textsc{a}(Q)$ applies filter normalisation and local variable normalisation in addition to $\textsc{u}(\cdot)$, but neither filters nor local variables appear in EMQs. Extensions to other functions do not affect EMQs in any way. Hence given an EMQ Q, it holds that $\textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{a}(Q))))) = \textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q)))))$. The result then follows from Theorem 6.3.

A.2.2. Proof for Theorem 7.2

We show that each step preserves congruence.

In $\textsc{a}(Q)$ we apply filter normalisation, local variable normalisation, and UCQ normalisation. Each step preserves query equivalence, and thus congruence.

Next we compute the r-graph, and apply minimisation. However, if the query is not in UCQ normal form, we only apply minimisation on BGPs and UBGPs contained in the query, considering any variables external to the (U)BGP as being "projected", and thus fixed. More formally, taking a UBGP Q' inside a larger query Q, and letting V' denote the variables of Q used both inside and outside Q, observe that we can replace Q' with $\textsc{select}_{V'}(Q')$ inside Q without changing the semantics of Q as variables of vars Q' not in V' are not used elsewhere in Q, and since Q is a query, it must contain a SELECT, ASK, CONSTRUCT or DESCRIBE clause, whose results will not change if a variable not mentioned in the clause is projected away. Now since $\textsc{select}_{V'}(Q')$ is a UCQ, the minimisation process preserves congruence per Theorem 7.2.

Regarding the r-graph, by definition $\textsc{r}^{-}(\textsc{r}(Q)) = Q$, with the exception of property paths, but in the latter case it is clear that $\textsc{r}^{-}(\textsc{r}(Q)) \cong Q$ since the r-graph representation of RPQs relies on well-known automata techniques – Thompson's construction, subset expansion, Hopcroft's algorithm and state elimination – that will produce an equivalent RPQ (similar automata-based techniques were also used by Kostylev et al. [34] for their analysis of the containment of property paths). Thus $\textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{a}(Q))))) \cong \textsc{r}^{-}(\textsc{m}(\textsc{r}(\textsc{a}(Q)))) \cong \textsc{r}^{-}(\textsc{r}(\textsc{a}(Q))) \cong \textsc{r}^{-}(\textsc{r}(Q)) \cong Q$, and the result holds per the transitivity of the $\cong$ relation.

References

[1] F.N. Afrati, M. Damigos and M. Gergatsoulis, Query containment under bag and bag-set semantics, *Inf. Process. Lett.* **110**(10) (2010), 360–369. doi:10.1016/j.ipl.2010.02.017.

[2] F. Alkhateeb, J.-F. Baget and J. Euzenat, Extending SPARQL with regular expression patterns (for querying RDF), *Web Semantics* **7**(2) (2009), 57–73. doi:10.1016/j.websem.2009.02.002.

[3] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter and D. Vrgoč, Foundations of modern query languages for graph databases, *ACM Computing Surveys* **50**(5) (2017), 68. doi:10.1145/3104031.

[4] C.B. Aranda, A. Hogan, J. Umbrich and P. Vandenbussche, SPARQL Web-querying infrastructure: Ready for action? in: *International Semantic Web Conference (ISWC)*, Vol. 8219, Springer, 2013, pp. 277–293. doi:10.1007/978-3-642-41338-4_18.

[5] M. Arenas and J. Pérez, Federation and navigation in SPARQL 1.1, in: *Reasoning Web Summer School*, LNCS, Vol. 7487, Springer, 2012, pp. 78–111. doi:10.1007/978-3-642-33158-9_3.

[6] M. Arenas and M. Ugarte, Designing a query language for RDF: Marrying open and closed worlds, *ACM Trans. Database Syst.* **42**(4) (2017), 21:1–21:46. doi:10.1145/3129247.

[7] A. Bonifati, W. Martens and T. Timm, An analytical study of large SPARQL query logs, *VLDB J.* **29**(2–3) (2020), 655–679. doi:10.1007/s00778-019-00558-9.

[8] J.A. Brzozowski and E.J. McCluskey, Signal flow graph techniques for sequential circuit state diagrams, in: *IEEE Transactions on Electronic Computers*, 1963, pp. 67–76. doi:10.1109/PGEC.1963.263416.

[9] R. Castillo and U. Leser, Selecting materialized views for RDF data, in: *International Conference on Web Engineering (ICWE), Workshops*, F. Daniel and F.M. Facca, eds, Lecture Notes in Computer Science, Vol. 6385, Springer, 2010, pp. 126–137. doi:10.1007/978-3-642-16985-4_12.

[10] N. Chakraborty, D. Lukovnikov, G. Maheshwari, P. Trivedi, J. Lehmann and A. Fischer, Introduction to Neural Network based Approaches for Question Answering over Knowledge Graphs, 2019, CoRR, http://arxiv.org/abs/1907.09361 abs/1907.09361.

[11] A.K. Chandra and P.M. Merlin, Optimal implementation of conjunctive queries in relational data bases, in: *Symposium on Theory of Computing (STOC)*, J.E. Hopcroft, E.P. Friedman and M.A. Harrison, eds, ACM, 1977, pp. 77–90. doi:10.1145/800105.803397.

[12] S. Chaudhuri and M.Y. Vardi, *Optimization of Real Conjunctive Queries, in: Principles of Database Systems (PODS)*, ACM Press, 1993, pp. 59–70. doi:10.1145/153850.153856.

[13] M.W. Chekol, J. Euzenat, P. Genevès and N. Layaïda, *SPARQL Query Containment Under SHI Axioms, in: AAAI Conference on Artificial Intelligence*, AAAI Press, 2012.

[14] M.W. Chekol, J. Euzenat, P. Genevès and N. Layaïda, Evaluating and benchmarking SPARQL query containment solvers, in: *International Semantic Web Conference (ISWC)*, H. Alani, L. Kagal, A. Fokoue, P. Groth, C. Biemann, J.X. Parreira, L. Aroyo, N.F. Noy, C. Welty and K. Janowicz, eds, Lecture Notes in Computer Science, Vol. 8219, Springer, 2013, pp. 408–423. doi:10.1007/978-3-642-41338-4_26.

[15] S. Chu, B. Murphy, J. Roesch, A. Cheung and D. Suciu, Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries, *Proc. VLDB Endow.* **11**(11) (2018), 1482–1495. doi:10.14778/3236187.3236200.

[16] S. Chu, C. Wang, K. Weitz and A. Cheung, Cosette: An automated prover for SQL, in: *Conference on Innovative Data Systems Research (CIDR)*, 2017, www.cidrdb.org.

[17] S. Cohen, W. Nutt and A. Serebrenik, Rewriting aggregate queries using views, in: *SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, ACM Press, 1999, pp. 155–166. doi:10.1145/303976.303992.

[18] R. Cyganiak, A relational algebra for SPARQL, 2005, http://shiftleft.com/mirrors/www.hpl.hp.com/techreports/2005/HPL-2005-170.pdf.

[19] R. Cyganiak, D. Wood and M. Lanthaler, *RDF 1.1 Concepts and Abstract Syntax*, 2014. http://www.w3.org/TR/rdf11-concepts/.

[20] M. Dürst and M. Suignard, in: *Internationalized Resource Identifiers (IRIs)*, Vol. 3987, 2005, pp. 1–46. https://tools.ietf.org/html/rfc3987. doi:10.17487/RFC3987.

[21] H. Glaser, I. Millard and A. Jaffri, RKBExplorer.com: A knowledge driven infrastructure for linked data providers, in: *European Semantic Web Conference (ESWC)*, Lecture Notes in Computer Science, Vol. 5021, Springer, 2008, pp. 797–801. doi:10.1007/978-3-540-68234-9_61.

[22] C. Gutierrez, C.A. Hurtado, A.O. Mendelzon and J. Pérez, Foundations of Semantic Web databases, *J. Comput. Syst. Sci.* **77**(3) (2011), 520–541. doi:10.1016/j.jcss.2010.04.009.

[23] M.M. Haklay and P. Weber, OpenStreetMap: User-generated street maps, *IEEE Pervasive Comput.* **7**(4) (2008), 12–18. doi:10.1109/MPRV.2008.80.

[24] S. Harris, A. Seaborne and E. Prud'hommeaux, 2013, SPARQL 1.1 Query Language, http://www.w3.org/TR/sparql11-query/.

[25] P. Hayes and P.F. Patel-Schneider, RDF 1.1 Semantics, 2014, http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/.

[26] T. Heath and C. Bizer, *Linked Data: Evolving the Web into a Global Data Space*, Vol. 1, Morgan & Claypool, 2011, pp. 1–136. ISBN 9781608454310.

[27] D. Hernández, C. Gutiérrez and R. Angles, The Problem of Correlation and Substitution in SPARQL – Extended Version, 2018, CoRR, http://arxiv.org/abs/1801.04387 abs/1801.04387.

[28] A. Hogan, Canonical forms for isomorphic and equivalent RDF graphs: Algorithms for leaning and labelling blank nodes, *ACM TOW* **11**(4) (2017), 22:1–22:62. doi:10.1145/3068333.

[29] J. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: *Theory of Machines and Computations*, Elsevier, 1971, pp. 189–196. doi:10.1016/B978-0-12-417750-5.50022-1.

[30] Y.E. Ioannidis and R. Ramakrishnan, Containment of conjunctive queries: Beyond relations as sets, *ACM Trans. Database Syst.* **20**(3) (1995), 288–324. doi:10.1145/211414.211419.

[31] T.A. Junttila and P. Kaski, Engineering an efficient canonical labeling tool for large and sparse graphs, in: *Workshop on Algorithm Engineering and Experiments (ALENEX)*, SIAM, 2007. doi:10.1137/1.9781611972870.13.

[32] M. Kaminski, E.V. Kostylev and B.C. Grau, Query nesting, assignment, and aggregation in SPARQL 1.1, *ACM Trans. Database Syst.* **42**(3) (2017), 17:1–17:46. doi:10.1145/3083898.

[33] E. Kharlamov, D. Hovland, M.G. Skjæveland, D. Bilidas, E. Jiménez-Ruiz, G. Xiao, A. Soylu, D. Lanti, M. Rezk, D. Zheleznyakov, M. Giese, H. Lie, Y.E. Ioannidis, Y. Kotidis, M. Koubarakis and A. Waaler, Ontology based data access in statoil, *J. Web Semant.* **44** (2017), 3–36. doi:10.1016/j.websem.2017.05.005.

[34] E.V. Kostylev, J.L. Reutter, M. Romero and D. Vrgoc, SPARQL with property paths, in: *International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science (LNCS), Vol. 9366, Springer, 2015, pp. 3–18. doi:10.1007/978-3-319-25007-6_1.

[35] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P.N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer and C. Bizer, DBpedia – a large-scale, multilingual knowledge base extracted from Wikipedia, *Semantic Web* **6**(2) (2015), 167–195. doi:10.3233/SW-140134.

[36] A. Letelier, J. Pérez, R. Pichler and S. Skritek, Static analysis and optimization of semantic web queries, *ACM Trans. Database Syst.* **38**(4) (2013), 25:1–25:45. doi:10.1145/2500130.

[37] S. Malyshev, M. Krötzsch, L. González, J. Gonsior and A. Bielefeldt, Getting the most out of wikidata: Semantic technology usage in Wikipedia's knowledge graph, in: *Proceedings, Part II*, The Semantic Web – ISWC 2018 – 17th International Semantic Web Conference, Monterey, CA, USA, October 8–12, 2018, 2018, pp. 376–394. doi:10.1007/978-3-030-00668-6_23.

[38] B. McBride, Jena: A Semantic Web toolkit, *IEEE Internet Computing* **6**(6) (2002), 55–59. doi:10.1109/MIC.2002.1067737.

[39] K. Möller, T. Heath, S. Handschuh and J. Domingue, *Recipes for Semantic Web Dog Food – the ESWC and ISWC Metadata Projects, in: International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science, Vol. 4825, Springer, 2007, pp. 802–815. doi:10.1007/978-3-540-76298-0_58.

[40] A.G. Nuzzolese, A.L. Gentile, V. Presutti and A. Gangemi, Conference linked data: The ScholarlyData project, in: International Semantic Web Conference (ISWC), *Lecture Notes in Computer Science* **9982** (2016), 150–158. doi:10.1007/978-3-319-46547-0_16.

[41] N. Papailiou, D. Tsoumakos, P. Karras and N. Koziris, Graph-aware, workload-adaptive SPARQL query caching, in: *ACM SIGMOD International Conference on Management of Data*, T.K. Sellis, S.B. Davidson and Z.G. Ives, eds, ACM, 2015, pp. 1777–1792. doi:10.1145/2723372.2723714.

[42] P.F. Patel-Schneider and D. Martin, EXISTStential aspects of SPARQL, in: *ISWC 2016 Posters & Demonstrations Track, CEUR Workshop Proceedings*, Vol. 1690, CEUR-WS.org, 2016.

[43] J. Pérez, M. Arenas and C. Gutiérrez, *Semantics and Complexity of SPARQL, in: International Semantic Web Conference (ISWC)*, LNCS, Vol. 4273, Springer, 2006, pp. 30–43. doi:10.1007/11926078_3.

[44] J. Pérez, M. Arenas and C. Gutierrez, Semantics and complexity of SPARQL, *ACM Trans. Database Syst.* **34**(3) (2009). doi:10.1145/1567274.1567278.

[45] R. Pichler and S. Skritek, Containment and equivalence of well-designed SPARQL, in: *Principles of Database Systems (PODS)*, ACM, 2014, pp. 39–50. doi:10.1145/2594538.2594542.

[46] A. Polleres, From SPARQL to rules (and back), in: *International Conference on World Wide Web (WWW)*, C.L. Williamson, M.E. Zurko, P.F. Patel-Schneider and P.J. Shenoy, eds, 2007, pp. 787–796. doi:10.1145/1242572.1242679.

[47] A. Polleres and J.P. Wallner, On the relation between SPARQL1.1 and answer set programming, *J. Appl. Non Class. Logics* **23**(1–2) (2013), 159–212. doi:10.1080/11663081.2013.798992.

[48] E. Prud'hommeaux and C. Buil-Aranda, SPARQL 1.1 Federated Query, 2013, https://www.w3.org/TR/sparql11-federated-query/.

[49] Y. Sagiv and M. Yannakakis, Equivalences among relational expressions with the union and difference operators, *J. ACM* **27**(4) (1980), 633–655. doi:10.1145/322217.322221.

[50] J. Salas and A. Hogan, Canonicalisation of monotone SPARQL queries, in: *International Semantic Web Conference (ISWC)*, D. Vrandecic, K. Bontcheva, M.C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L. Kaffee and E. Simperl, eds, Lecture Notes in Computer Science, Vol. 11136, Springer, 2018, pp. 600–616. doi:10.1007/978-3-030-00671-6_35.

[51] J. Salas and A. Hogan, Canonicalisation of Monotone SPARQL Queries, http://aidanhogan.com/qcan/extended.pdf.

[52] M. Saleem, M.I. Ali, A. Hogan, Q. Mehmood and A.N. Ngomo, *LSQ: The Linked SPARQL Queries Dataset*, International Semantic Web Conference (ISWC), Vol. 9367, Springer, 2015, pp. 261–269. doi:10.1007/978-3-319-25010-6_15.

[53] M. Schmidt, M. Meier and G. Lausen, Foundations of SPARQL query optimization, in: *International Conference on Database Theory (ICDT)*, L. Segoufin, ed., ACM, 2010, pp. 4–33. doi:10.1145/1804669.1804675.

[54] G. Schreiber and Y. Raimond, RDF 1.1 Primer, 2014, http://www.w3.org/TR/rdf11-primer/.

[55] A. Seaborne and P.F. Patel-Schneider, SPARQL EXISTS report, 2019.

[56] C. Stadler, J. Lehmann, K. Höffner and S. Auer, LinkedGeoData: A core for a web of spatial open data, *Semantic Web* **3**(4) (2012), 333–354. doi:10.3233/SW-2011-0052.

[57] C. Stadler, M. Saleem, A.N. Ngomo and J. Lehmann, Efficiently pinpointing SPARQL query containments, in: *International Conference on Web Engineering (ICWE)*, T. Mikkonen, R. Klamma and J. Hernández, eds, Lecture Notes in Computer Science, Vol. 10845, Springer, 2018, pp. 210–224. doi:10.1007/978-3-319-91662-0_16.

[58] C. Stadler, M. Saleem, A.N. Ngomo and J. Lehmann, Efficiently pinpointing SPARQL query containments, in: *International Conference Web Engineering (ICWE)*, Lecture Notes in Computer Science, Vol. 10845, Springer, 2018, pp. 210–224. doi:10.1007/978-3-319-91662-0_16.

[59] K. Thompson, Regular expression search algorithm, *Commun. ACM* **11**(6) (1968), 419–422. doi:10.1145/363347.363387.

[60] B. Trakhtenbrot, The impossibility of an algorithm for the decidability problem on finite classes, in: *Proceedings of the USSR Academy of Sciences*, Vol. 70, 1950, pp. 569–572.

[61] D. Vrandecic and M. Krötzsch, Wikidata: A free collaborative knowledgebase, *Commun. ACM* **57**(10) (2014), 78–85. doi:10.1145/2629489.

[62] G.T. Williams and J. Weaver, Enabling fine-grained HTTP caching of SPARQL query results, in: *The Semantic Web – ISWC 2011 – 10th International Semantic Web Conference, Proceedings, Part I*, Bonn, Germany, October 23–27, 2011, L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N.F. Noy and E. Blomqvist, eds, 2011, pp. 762–777. doi:10.1007/978-3-642-25073-6_48.

[63] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati and M. Zakharyaschev, Ontology-based data access: A survey, in: *International Joint Conference on Artificial Intelligence (IJCAI)*, ijcai.org, 2018, pp. 5511–5519. doi:10.24963/ijcai.2018/777.

[64] X. Zhang, M. Wang, M. Saleem, A.N. Ngomo, G. Qi and H. Wang, Revealing secrets in SPARQL session level, in: *International Semantic Web Conference (ISWC)*, LNCS, Vol. 12506, Springer, 2020, pp. 672–690. doi:10.1007/978-3-030-62419-4_38.

[65] Q. Zhou, J. Arulraj, S.B. Navathe, W. Harris and D. Xu, Automated verification of query equivalence using satisfiability modulo theories, *Proc. VLDB Endow.* **12**(11) (2019), 1276–1288. doi:10.14778/3342263.3342267.