Canonicalisation of Monotone SPARQL Queries

Jaime Salas and Aidan Hogan(✉)

IMFD Chile and Department of Computer Science,
University of Chile, Santiago, Chile
jsalas@dcc.uchile.cl, aidhog@gmail.com

Abstract. Caching in the context of expressive query languages such as SPARQL is complicated by the difficulty of detecting equivalent queries: deciding if two conjunctive queries are equivalent is NP-complete, where adding further query features makes the problem undecidable. Despite this complexity, in this paper we propose an algorithm that performs syntactic canonicalisation of SPARQL queries such that the answers for the canonicalised query will not change versus the original. We can guarantee that the canonicalisation of two queries within a core fragment of SPARQL (monotone queries with select, project, join and union) is equal if and only if the two queries are equivalent; we also support other SPARQL features but with a weaker soundness guarantee: that the (partially) canonicalised query is equivalent to the input query. Despite the fact that canonicalisation must be harder than the equivalence problem, we show the algorithm to be practical for real-world queries taken from SPARQL endpoint logs, and further show that it detects more equivalent queries than when compared with purely syntactic methods. We also present the results of experiments over synthetic queries designed to stress-test the canonicalisation method, highlighting difficult cases.

1 Introduction

SPARQL endpoints often encounter performance problems in practice: in a survey of hundreds of public SPARQL endpoints, Buil-Aranda et al. [2] found that many such services have mixed reliability and performance, often returning errors, timeouts or partial results. This is not surprising: SPARQL is an expressive query language that encapsulates and extends the relational algebra, where even the simplified decision problem of verifying if a given solution is contained in the answers of a given SPARQL query for a given database is known to be PSPACE-complete [17] (combined complexity). Furthermore, evaluating SPARQL queries may involve an exponential number of (intermediate) results. Hence, rather than aiming to efficiently support all queries over all database instances for all users, the goal is rather to continuously improve performance: to increase the throughput of the most common types of queries answered.

An obvious means by which to increase throughput of query processing is to re-use work done for previous queries when answering future queries by *caching*

© Springer Nature Switzerland AG 2018
D. Vrandečić et al. (Eds.): ISWC 2018, LNCS 11136, pp. 600–616, 2018.
https://doi.org/10.1007/978-3-030-00671-6_35

results. In the context of caching for SPARQL, however, there are some significant complications. While many engines may apply low-level caches to avoid, e.g., repeated index accesses, generating answers from such data can still require a lot of higher-level query processing. On the other hand, caching at the level of queries or subqueries is greatly complicated by the fact that a given abstract query can be expressed in myriad equivalent ways in SPARQL.

Addressing the latter challenge, in this paper we propose a method by which SPARQL queries can be *canonicalised*, where the canonicalised version of two queries Q_1 and Q_2 will be (syntactically) identical if Q_1 and Q_2 are *equivalent*: having the same results for any dataset. Furthermore, we say that two queries Q_1 and Q_2 are *congruent* if and only if they are equivalent modulo variable names, meaning we can rewrite the variables of Q_2 in a one-to-one manner to generate a query equivalent to Q_1; our proposed canonicalisation method then aims to give the same output for queries Q_1 and Q_2 if and only if they are congruent, which will allow us to find additional queries useful for applications such as caching.

Example 1. Consider two queries Q_A and Q_B asking for names of aunts:

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

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

Both queries are congruent: if we rewrite the variable `?n` to `?z` in Q_B, then both queries are equivalent and will return the same results for any RDF dataset. Canonicalisation aims to rewrite both queries to the same syntactic form. □

Our main use-case for canonicalisation is to improve *caching* for SPARQL endpoints: by capturing knowledge about query congruence, canonicalisation can increase the hit rate for a cache of (sub-)queries [16]. Furthermore, canonicalisation may be useful for *analysis of SPARQL logs*: finding repeated/congruent queries without pair-wise equivalence checks; *query processing*: where optimisations can be applied over canonical/normal forms; and so forth.

A fundamental challenge for canonicalising SPARQL queries is the high computational complexity that it entails. More specifically, the QUERY EQUIVALENCE PROBLEM takes two queries Q_1 and Q_2 and returns true if and only if they return the same answers for any database instance. In the case of SPARQL, this problem is NP-complete even when simply permitting joins (conjunctive queries). Even worse, the problem becomes undecidable when features such as projection and optional matches are combined [18]. Canonicalisation is then at least as hard as the equivalence problem, meaning it will likewise be intractable for even simple fragments and undecidable when considering the full SPARQL language.

We thus propose a canonicalisation procedure that does not change the semantics of an input query (i.e., is *correct*) but may miss congruent queries (i.e., is *incomplete*) for certain features. We deem such guarantees to be sufficient for use-cases where completeness is not a strong requirement, as in the case of caching where missing a congruent query will require re-executing the query (which would have to be done in any case). For *monotone queries* [19] in a core SPARQL fragment, we provide both correctness and completeness guarantees.

The procedure we propose is based on first converting SPARQL queries to a graph-based (RDF) algebraic representation. We then initially apply canonical labelling to the graph to consistently name variables, thereafter converting the graph back to a SPARQL query following a fixed syntactic ordering. The resulting query then represents the output of a baseline canonicalisation procedure for SPARQL. To support further SPARQL features such as `UNION`, we extend this procedure by applying normal forms and minimisation over the intermediate algebraic graph prior to its canonicalisation. Currently we focus on canonicalising `SELECT` queries from SPARQL 1.0. However, our canonicalisation techniques can be extended to other types of queries (`ASK`, `CONSTRUCT`, `DESCRIBE`) as well as the extended features of SPARQL 1.1 (including aggregation, property paths, etc.) while maintaining correctness guarantees; this is left to future work.

Extended Version: An online version of this paper provides additional definitions, proofs, and experimental results [20].

2 Preliminaries

RDF: We first introduce the RDF data model, as well as notions of isomorphism and equivalence relevant to the canonicalisation procedure discussed later.

Terms and Graphs. RDF assumes three pairwise disjoint sets of terms: *IRIs*: $\mathbf{I}$, *literals* $\mathbf{L}$ and *blank nodes* $\mathbf{B}$. An *RDF triple* (s, p, o) is composed of three terms – called *subject*, *predicate* and *object* – where $s \in \mathbf{IB}$, $p \in \mathbf{I}$ and $o \in \mathbf{ILB}$.[1] A finite set of RDF triples is called an *RDF graph* $G \subseteq \mathbf{IB} \times \mathbf{I} \times \mathbf{IBL}$.

Isomorphism. Blank nodes are defined as existential variables [10] where two RDF graphs differing only in blank node labels are thus considered *isomorphic* [7]. Formally, let $\mu : \mathbf{IBL} \to \mathbf{IBL}$ denote a mapping of RDF terms to RDF terms such that μ is the identity on $\mathbf{IL}$ ($\mu(x) = x$ for all $x \in \mathbf{IL}$); we call μ a *blank node mapping*; if μ maps blank nodes to blank nodes in a one-to-one manner, we call it a *blank node bijection*. Let $\mu(G)$ denote the image of an RDF graph G under μ (applying μ to each term in G). Two RDF graphs G_1 and G_2 are defined as isomorphic – denoted $G_1 \cong G_2$ – if and only if there exists a blank node bijection μ such that $\mu(G_1) = G_2$. Given two RDF graphs, the problem of determining if they are isomorphic is GI-complete [11], meaning the problem is in the same complexity class as the standard GRAPH ISOMORPHISM PROBLEM.

Equivalence. The *equivalence* relation captures the idea that two RDF graphs entail each other [10]. Two RDF graphs G_1 and G_2 are *equivalent* – denoted $G_1 \equiv G_2$ – if and only if there exists two blank node mappings μ_1 and μ_2 such that $\mu_1(G_1) \subseteq G_2$ and $\mu_2(G_2) \subseteq G_1$ [8]. A graph may be equivalent to a smaller graph (due to redundancy). We thus say that an RDF graph G is *lean* if it does not have a proper subset $G' \subset G$ such that $G \equiv G'$; otherwise

[1] We use, e.g., $\mathbf{IBL}$ as a shortcut for $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$.

we can say that it is *non-lean*. Furthermore, we can define the *core* of a graph G as a lean graph G' such that $G \equiv G'$; the core of a graph is known to be unique modulo isomorphism [8]. Determining equivalence between RDF graphs is known to be NP-complete [8]. Determining if a graph G is lean is known to be coNP-complete [8]. Finally, determining if a graph G' is the core of a second graph G is known to be DP-complete [8].

Graph Canonicalisation. Our method for canonicalising SPARQL queries involves representing the query as an RDF graph, applying canonicalisation techniques over that graph, and mapping the canonical graph back to a SPARQL query. As such, our query canonicalisation method relies on an existing graph canonicalisation framework for RDF graphs called BLABEL [12]; this framework offers a sound and complete method to canonicalise graphs with respect to isomorphism (iCAN(G)) or equivalence (eCAN(G)). Both methods have exponential worst-case behaviour; as discussed, the underlying problems are intractable.

SPARQL. We now provide preliminaries for the SPARQL query language [9]. For brevity, our definitions focus on SPARQL *monotone queries* (MQs) [19] – permitting selection $(=, \wedge, \vee)$[2], join, union and projection – for which we can offer sound and complete canonicalisation.

Syntax. Let $\mathbf{V}$ denote a set of query variables disjoint with $\mathbf{IBL}$. We define the abstract syntax of a SPARQL MQ as follows:

1. A *triple pattern* t is a member of the set $\mathbf{VIB} \times \mathbf{VI} \times \mathbf{VIBL}$ (i.e., an RDF triple allowing variables in any position). A triple pattern is a *query pattern*.
2. If both Q_1 and Q_2 are query patterns, then $[Q_1 \text{ AND } Q_2]$, and $[Q_1 \text{ UNION } Q_2]$ are also query patterns.
3. If Q is a query pattern and V is a set of variables such that for all $v \in V$, v appears in some triple pattern contained in Q, then $\text{SELECT}_V(Q)$ is a *query*.[3]

Blank nodes in SPARQL queries are considered to be non-distinguished query variables where we will assume they have been replaced with fresh query variables. Per the final definition, we currently do not support subqueries and assume, w.l.o.g., that all queries have a projection $\text{SELECT}_V(Q)$.

Algebra. We will now define an algebra for such queries. A *solution* μ is a partial mapping from variables in $\mathbf{V}$ appearing in the query to constants from $\mathbf{IBL}$ appearing in the data. Let $\text{dom}(\mu)$ denote the variables for which μ is defined. We say that two mappings μ_1 and μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, when $\mu_1(v) = \mu_2(v)$ for every $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$. Letting M, M_1 and M_2 denote sets of solutions, we define the algebra as follows:

$$
\begin{aligned}
M_1 \bowtie M_2 &:= \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2, \mu_1 \sim \mu_2\} \\
M_1 \cup M_2 &:= \{\mu \mid \mu \in M_1 \text{ or } \mu \in M_2\} \\
\pi_V(M) &:= \{\mu' \mid \exists \mu \in M : \mu' \subseteq \mu, \text{dom}(\mu') = V \cap \text{dom}(\mu)\}
\end{aligned}
$$

[2] This is expressed by placing constants in triple patterns.
[3] Note that `SELECT *` is equivalent to returning all variables (or omitting the feature).

Union is defined here in the SPARQL fashion as a union of mappings, rather than relational algebra union: the former can be applied over solution mappings with different domains, while the latter does not allow this.

Semantics. Letting Q denote an MQ pattern in the abstract syntax, we denote the evaluation of Q over an RDF graph G as $Q(G)$. Before defining $Q(G)$, first let t denote a triple pattern; then by $\mathbf{V}(t)$ we denote the set of variables appearing in t and by $\mu(t)$ we denote the image of t under a solution μ. Finally, we can define $Q(G)$ recursively as follows:

$$\begin{aligned} t(G) &:= \{\mu \mid \mu(t) \in G, \mathrm{dom}(\mu) = \mathbf{V}(t)\} \\ [Q_1 \text{ AND } Q_2](G) &:= Q_1(G) \bowtie Q_2(G) \\ [Q_1 \text{ UNION } Q_2](G) &:= Q_1(G) \cup Q_2(G) \\ \text{SELECT}_V(Q)(G) &:= \pi_V(Q(G)) \end{aligned}$$

Set vs. Bag. The previous definitions assume a *set semantics* for query answering, meaning that no duplicate mappings are returned as solutions [17]. However, the SPARQL standard, by default, considers a *bag* (aka. *multiset*) *semantics* for query answering [9], where the cardinality of a solution in the results captures information about how many times the query pattern matched the underlying dataset [1]. We thus use the extended syntax $\text{SELECT}_V^{\Delta}(Q)$, where $\Delta = \texttt{true}$ indicates set semantics and $\Delta = \texttt{false}$ indicates bag semantics.

Containment and Equivalence. Query containment asks: *given two queries Q_1 and Q_2, does it hold that $Q_1(G) \subseteq Q_2(G)$ for all possible RDF graphs G*? If so, we say that Q_2 *contains* Q_1, which we denote by the relation $Q_1 \sqsubseteq Q_2$. On the other hand, query equivalence asks, *given two queries Q_1 and Q_2, does it hold that $Q_1(G) = Q_2(G)$ for all possible RDF graphs G*? In other words, Q_1 and Q_2 are equivalent if and only if Q_1 and Q_2 contain each other. If so, we say that $Q_1 \equiv Q_2$. In this paper, we relax the equivalence notion to ignore labelling of variables; more formally, let $\nu : \mathbf{V} \to \mathbf{V}$ be a one-to-one mapping of variables and, slightly abusing notation, let $\nu(Q)$ denote the image of Q under ν (rewriting variables in Q wrt. ν); we say that Q_1 and Q_2 are *congruent* (denoted $Q_1 \cong Q_2$) if and only if there exists ν such that $Q_1 \equiv \nu(Q_2)$. An example of such query congruence was provided in Example 1.

The complexity of query containment and equivalence vary from NP-complete when just AND is allowed (with triple patterns), upwards to UNDECIDABLE once, e.g., projection and optional matches are added [18]. For MQs, containment and equivalence are NP-complete for the related query class of *Unions of Conjunctive Queries* (UCQs) [19], which allow the same features as MQs but disallow joins over unions. Interestingly, though MQs and UCQs are equivalent query classes – i.e., for any UCQ there is an equivalent MQ and vice-versa – containment and equivalence for MQs jumps to Π_2^P-complete [19]. Intuitively this is because MQs are more succinct than UCQs; for example, to find a path of length n where each node is of type A or B, we can create an MQ of size $O(n)$,

but it requires a UCQ of size $O(2^n)$. We consider MQ*s* since real-world SPARQL queries may arbitrarily nest joins and unions (canonicalisation will rewrite them to UCQ*s*).

Most of the above results have been developed under set semantics. In terms of bag semantics, we can consider an analogous containment problem: that the answers of Q_1 are a *subbag* of the answers of Q_2, meaning that the multiplicity of an answer in Q_1 is always less-than-or-equals the multiplicity of the same answer in Q_2. In fact, the decidability of this problem remains an open question [4]; on the other hand, the equivalence problem is GI-complete [4], and thus in fact probably *easier* than the case for set semantics (assuming GI $\neq$ NP): under bag semantics, conjunctive queries cannot have redundancy, so intuitively speaking we can test a form of isomorphism between the two queries.

3 Related Work

Various works have presented complexity results for query containment and equivalence of SPARQL [5,13,14,18,23,24]. With respect to implementations, only one dedicated library has been released to check whether or not two SPARQL queries are equivalent: SPARQL Algebra [14]. The problem of determining equivalence of SPARQL queries can, however, be solved by reductions to related problems, where Chekol et al. [6] have used a μ-calculus solver and an XPath-equivalence checker to implement SPARQL equivalence checks. Recently Saleem et al. [22] compared these SPARQL query containment methods using a benchmark based on real-world query logs; we use these same logs in our evaluation. These works do not deal with canonicalisation; using an equivalence checker would require quadratic pairwise checks to determine all equivalences in a set or stream of queries; hence they are impractical for a use-case such as caching.

To the best of our knowledge, little work has been done specifically on canonicalisation of SPARQL queries. In analyses of logs, some authors [3,21] have proposed some syntactic canonicalisation methods – such as normalising whitespace or using a SPARQL library to format the query – that do manage to detect some duplicates, but not more complex cases such as per Example 1. Rather the most similar work to ours (to the best of our knowledge) is the SPARQL caching system proposed by Papailiou et al. [16], which uses a canonical labelling algorithm (specifically Bliss) to assign consistent labels to variables, allowing to recall isomorphic graph patterns from the cache for SPARQL queries. However, their work does not consider factoring out redundancy caused by query operators (aka. *minimisation*), and hence they would not capture equivalences as in the case of Example 1. In general, our work focuses on canonicalisation of queries whereas the work of Papailiou et al. [16] is rather focused on caching; compared to them we capture a much broader notion of query equivalence than their approach based solely on canonical labelling of query variables. It is worth noting that we are not aware of similar methods for canonicalising SQL queries.

4 Query Canonicalisation

Our approach for canonicalising SPARQL MQs involves representing the query as an RDF graph, performing a canonicalisation of the RDF graph (including the application of algebraic rewritings, minimisation and canonical labelling), ultimately mapping the resulting graph back to a final canonical SPARQL UCQ.

4.1 Representational Graph for UCQs

The MQ class is closed under join and union (see Q_A, Example 1). As the first query normalisation step, we will convert MQ queries to UCQs of the form $\mathsf{select}_V^{\Delta}(\mathsf{union}(\{\mathsf{and}(\{Q_1^1, \dots Q_m^1\}), \dots, \mathsf{and}(\{Q_1^k, \dots Q_n^k\})\}))$ following a standard DNF-style expansion (we refer to the extended version for more details [20]). The output UCQ may be exponential in size. Thereafter, given such a UCQ, we define its *representational graph* (or R-graph for short) as follows.

Definition 1. *Let $\beta()$ denote a function that returns a fresh blank node and $\beta(x)$ a function that returns a blank node unique to x. Let $\iota(\cdot)$ denote an id function such that if $x \in \mathbf{IL}$, then $\iota(x) = x$; otherwise if $x \in \mathbf{VB}$, then $\iota(x) = \beta(x)$. Finally, let Q be a* UCQ*; we define* R(Q)*, the* R*-graph of Q, as follows:*

- *If Q is a triple pattern (s, p, o), then $\iota(Q)$ is set as $\beta()$ and*

$$\mathrm{R}(Q) = \{(\iota(Q), :\mathtt{s}, \iota(s)), (\iota(Q), :\mathtt{p}, \iota(p)), (\iota(Q), :\mathtt{o}, \iota(o)), (\iota(Q), \mathtt{a}, :\mathtt{TP})\}$$

- *If Q is $\mathsf{and}(\{Q_1, \dots, Q_n\})$, then $\iota(Q)$ is set as $\beta()$ and*

$$\begin{aligned}\mathrm{R}(Q) &= \{(\iota(Q), :\mathtt{arg}, \iota(Q_1)), \dots, (\iota(Q), :\mathtt{arg}, \iota(Q_n)), (\iota(Q), \mathtt{a}, :\mathtt{And})\} \\ &\cup \mathrm{R}(Q_1) \cup \dots \cup \mathrm{R}(Q_n)\end{aligned}$$

- *If Q is $\mathsf{union}(\{Q_1, \dots, Q_n\})$, then $\iota(Q)$ is set as $\beta()$ and*

$$\begin{aligned}\mathrm{R}(Q) &= \{(\iota(Q), :\mathtt{arg}, \iota(Q_1)), \dots, (\iota(Q), :\mathtt{arg}, \iota(Q_n)), (\iota(Q), \mathtt{a}, :\mathtt{Union})\} \\ &\cup \mathrm{R}(Q_1) \cup \dots \cup \mathrm{R}(Q_n)\end{aligned}$$

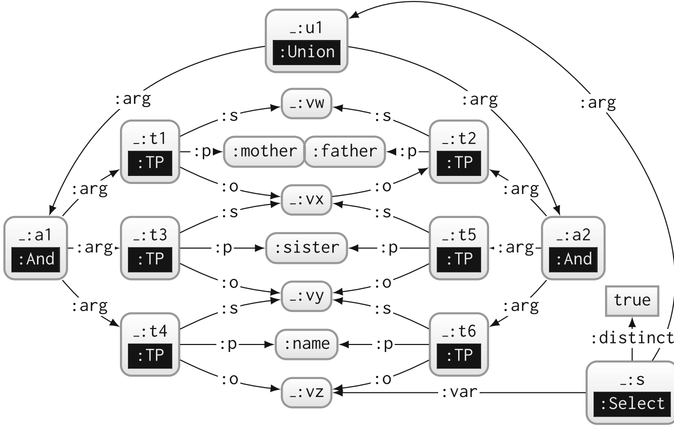
- *If Q is $\mathsf{select}_V^{\Delta}(Q_1)$, then $\iota(Q)$ is set as $\beta()$ and*

$$\begin{aligned}\mathrm{R}(Q) &= \{(\iota(Q), :\mathtt{arg}, \iota(Q_1)), (\iota(Q), :\mathtt{distinct}, \Delta), (\iota(Q), \mathtt{a}, :\mathtt{Select})\} \\ &\cup \{(\iota(Q), :\mathtt{var}, \iota(v)) \mid v \in V\} \cup \mathrm{R}(Q_1)\end{aligned}$$

*where "*a*" abbreviates* rdf : type *and Δ is a boolean datatype literal.* □

Example 2. Here we provide an example of the R-graph for query Q_A and Q_B in Example 1: the R-graph has the same structure for both queries assuming that a UCQ normal form is applied beforehand (to Q_A in particular). For clarity, we

embed the types of nodes into the nodes themselves; e.g., the uppermost node expands to _:u1 —rdf:type→ :Union.



Due to the application of UCQ normal forms, we have a projection, over a union, over a set of joins, where each join involves one or more triple patterns. □

We also define the inverse $\mathrm{R}^{-}(\mathrm{R}(Q))$, mapping an R-graph back to a UCQ query, such that $\mathrm{R}^{-}(\mathrm{R}(Q))$ is congruent to the Q [20].

4.2 Projection with Union

Unlike the relational algebra, SPARQL MQs allow unions of query patterns whose sets of variables are not equal. This may give rise to existential variables, which in turn can lead to further equivalences that must be considered [19].

Example 3. Returning to Example 1, consider a query $Q_C \equiv Q_B$, a minor variant of Q_B using different non-projected variables in the union:

```
SELECT DISTINCT ?n WHERE { { ?a :name ?n . ?c :mother ?m . ?m :sister ?a . }
    UNION { ?a :name ?n . ?c :father ?f . ?f :sister ?a . } }
```

Such unions are permitted in SPARQL. Likewise we could rename both occurrences of `?a` on the left of the union in Q_C without changing the solutions since `?a` is not projected. Any correspondences between non-projected variables across a union are thus syntactic and do not affect the semantics of the query. □

We thus distinguish the blank node representing every non-projected variable in each CQ of the R-graph produced previously. Letting G denote $\mathrm{R}(Q)$, we define the CQ roots of G as $\mathsf{cq}(G) = \{y \mid (y, \mathtt{a}, \mathtt{:And}) \in G\}$. Given a term r and a graph G, we define $G[r]$ as the sub-graph of G rooted in r, defined recursively as $G[r]_0 = \{(s,p,o) \in G \mid s = r\}$, $G[r]_i = \{(s,p,o) \in G \mid \exists x, y : (x,y,s) \in G[r]_{i-1}\} \cup G[r]_{i-1}$, with $G[r] = G[r]_n$ such that $G[r]_n = G[r]_{n+1}$ (the fixpoint).

We denote the blank nodes representing variables in G by $\mathsf{var}(G) = \{v \in \mathbf{B} \mid \exists (s,p) : (s,p,v) \in G \wedge p \in \{\mathtt{:s}, \mathtt{:p}, \mathtt{:o}\}\}$, and we denote the blank nodes representing unprojected variables in G by $\mathsf{uvar}(G) = \{v \in \mathsf{var}(G) \mid \nexists s : (s, \mathtt{:var}, v) \in G\}$. Finally we denote the blank nodes representing projected variables in G by $\mathsf{pvar}(G) = \mathsf{var}(G) \setminus \mathsf{uvar}(G)$. We can now define how variables are distinguished.

Definition 2. *Let G denote $\text{R}(Q)$ for a* UCQ Q*. We define the variable distinguishing function* $\text{D}(G)$ *as follows. If there does not exist a blank node x such that* $(x, \texttt{a}, \texttt{:Union}) \in G$*, then* $\text{D}(G) = G$*. Otherwise if such a blank node exists, we define* $\text{D}(G) = \{(s, p, \delta(o)) \mid (s, p, o) \in G\}$*, where* $\delta(o) = o$ *if* $o \notin \mathsf{uvar}(G)$*; otherwise* $\delta(o) = \beta(r, o)$ *such that* $r \in \mathsf{cq}(G)$ *and* $(s, p, o) \in G[r]$*.* □

In other words, $\text{D}(G)$ creates a fresh blank node for each non-projected variable appearing in the representation of a CQ in G as previously motivated.

4.3 Minimisation

Under set semantics, UCQs may contain redundancy whereby, for the purposes of canonicalisation, we will apply *minimisation* to remove redundant triple patterns while maintaining query equivalence. After applying UCQ normalisation, the R-graph now represents a UCQ of the form $(Q, V) := (Q_1 \cup \ldots \cup Q_n, V)$, with each $Q_1, \ldots, Q_n$ being a CQ and V being the set of projected variables. Under set semantics, we then first remove *intra*-CQ *redundancy* from the individual CQs; thereafter we remove *inter*-CQ *redundancy* from the overall UCQ.

Bag Semantics. We briefly note that if projection with bag semantics is selected, the UCQ can only contain one (syntactic) form of redundancy: exact duplicate triple patterns in the same CQ. Any other form of redundancy mentioned previously – be it intra-CQ or inter-CQ redundancy – will affect the multiplicity of results [4]. Hence if bag semantics is selected, we do not apply any redundancy elimination other than removing duplicate triple patterns in CQs.

Set-Semantics/CQs. We now minimise the individual CQs of the R-graph by computing the core of the sub-graph induced by each CQ independently. But before computing the core, we must ground projected variables to avoid their removal during minimisation. Along these lines, let G denote an R-graph $\text{D}(\text{R}(Q))$ of Q. We define the grounding of projected variables as follows: $L(G) = \{(s, p, \lambda(o)) \mid (s, p, o) \in G\}$, where if o denotes a projected variable, $\lambda(o) = \texttt{:o}$ for $\texttt{:o}$ a fresh IRI computed for o; otherwise $\lambda(o) = o$. We assume for brevity that variable IRIs created by λ can be distinguished from other IRIs. Finally, let $\mathsf{core}(G)$ denote the core of G. We can then minimise each CQ as follows.

Definition 3. *Let G denote* $\text{D}(\text{R}(Q))$*. We define the* CQ*-minimisation of G as* $\text{C}(G) = \{\mathsf{core}(L(G[x])) \mid x \in \mathsf{cq}(G)\}$*. We call $C \in \text{C}(G)$ a CQ core.* □

Example 4. Consider the following query, Q_D:

```
SELECT DISTINCT ?z WHERE {
  { ?w :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 }
```

This query is congruent to the previous queries Q_A, Q_B, Q_C. After applying UCQ normal forms, we end up with the following R-graph for Q_D:

```
SELECT DISTINCT ?z WHERE {
  { ?w1 :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 CQ (the graph induced by the CQ node with type :And and any node reachable from that node in the directed R-graph). Figure 1 depicts the sub-R-graph representing the third CQ (omitting the :And-typed root node for clarity since it will not affect computing the core). The dashed sub-graph will be removed from the core per the map: {_:vx3/_:vd3, _:t35/_:t32, _:t33/_:t32, _:vp3/:sister, _:ve3/_:vy3, _:t34/_:t36, _:vf3/:vz, ...}, 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 {
  { ?w1 :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 CQs. However, we are still left with (inter-CQ) redundancy between the first and third CQs. □

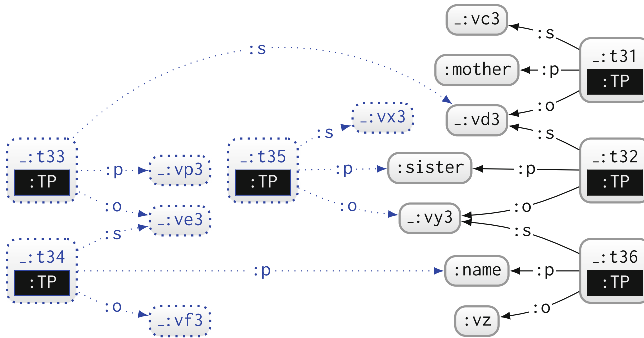


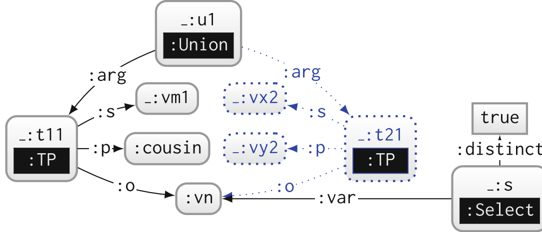
Fig. 1. R-graph of a CQ showing minimisation by leaning

Set Semantics/UCQs. After minimising individual CQs, we may still be left with a union containing redundant CQs as highlighted by Example 4. Hence we must now apply a higher-level minimisation of redundant CQs. While it may be tempting to simply compute the core of the entire R-graph – as would work for Example 4 and, indeed, as would also work for unions in the relational algebra – unfortunately SPARQL union again raises some non-trivial complications [19].

Example 5. Consider the following (unusual) query:

```
SELECT DISTINCT ?n WHERE { { ?m :cousin ?n . } UNION { ?x ?y ?n . } }
```

If we were to compute the core over the R-graph for the entire UCQ, we would remove the second CQ as follows:



This would leave us with the following query:

```
SELECT DISTINCT ?n WHERE { ?m :cousin ?n . }
```

But this has changed the query semantics where we lose non-cousin values. □

Instead, we must check containment between pairs of CQs [19]. Let $(Q, V) := (Q_1 \cup \ldots \cup Q_n, V)$ denote the UCQ under analysis. We need to remove from Q:

1. all Q_i $(1 \leq i \leq n)$ such that there exists Q_j $(1 \leq j < i \leq n)$ such that $\mathsf{select}_V(Q_i) \equiv \mathsf{select}_V(Q_j)$; *and*
2. all Q_i $(1 \leq i \leq n)$ where there exists Q_j $(1 \leq j \leq n)$ such that $\mathsf{select}_V(Q_i) \sqsubset \mathsf{select}_V(Q_j)$ (i.e., proper containment where $\mathsf{select}_V(Q_i) \not\equiv \mathsf{select}_V(Q_j)$);

The former condition removes all but one CQ from each group of equivalent CQs while the latter condition removes all CQs that are properly contained in another CQ. With respect to SPARQL union, note that these definitions apply to cases where CQs have different variables. More explicitly, let $V_1, \ldots, V_n$ denote the projected variables appearing in $Q_1, \ldots, Q_n$, respectively. Observe that $\mathsf{select}_{V_i}(Q_i) \sqsubseteq \mathsf{select}_{V_j}(Q_j)$ can only hold if $V_i = V_j$: assume without loss of generality that $v \in V_i \setminus V_j$, where v must then generate unbounds in V_j, creating a mapping μ, $v \in \mathrm{dom}(\mu)$, that can never appear in V_i.[4]

To implement condition (1), let us first assume that all CQs contain all projection variables such that no unbounds can be returned. Note that in the previous step we have computed the cores of CQs in C(G) and hence it is sufficient to check for isomorphism between them; we can thus take the current R-graph G_i for each Q_i and apply iso-canonicalisation of G_i [12], removing any other Q_j $(j > i)$ whose G_j is isomorphic. Thereafter, to implement condition (2), we can check if there exists a blank node mapping μ such that $\mu(G_j) \subseteq G_i$, for $i \neq j$ (which is equivalent to checking *simple entailment*: $G_i \models G_j$ [8]).

[4] We assume that CQs without variables may generate an empty mapping ($\{\mu\}$ with $\mathrm{dom}(\mu) = \emptyset$) if the CQ is contained in the data, or no mapping ($\{\}$) otherwise. This means we will not remove such CQs (unless they are precisely equal to another CQ) as they will generate a tuple of unbounds in the results if and only if the data match.

Now we drop the assumption that all CQs contain all variables in V, meaning that we can generate unbounds. To resolve such cases, we can partition $\{Q_1, \ldots, Q_n\}$ into various sets of CQs based on the projected variables they contain, and then apply equivalence and containment checks in each part.

Definition 4. *Let* $\text{C}(G) = \{C_1, \ldots, C_n\}$ *denote the* CQ *cores of* $G = \text{D}(\text{R}(Q))$*. A* CQ *core* C_i *is in* $\text{E}(G)$ *iff* $C_i \in \text{C}(G)$ *and there does not exist a* CQ *core* $C_j \in \text{C}(G)$ *(*$i \neq j$*) such that:* $\mathsf{pvar}(C_i) = \mathsf{pvar}(C_j)$*; and* $C_i \cong C_j$ *with* $j < i$ *or* $C_j \models C_i$*.* □

Definition 5. *Let* $\text{E}(G) = \{C_1, \ldots, C_n\}$ *denote the minimal* CQ *cores of* $G = \text{D}(\text{R}(Q))$*. Let* $P = \{(s, p, o) \in G \mid \exists (s, \mathtt{a}, \mathtt{:Select}) \in G\}$ *and* $U = \{(s, p, o) \in G \mid \exists (s, \mathtt{a}, \mathtt{:Union}) \in G,$ and $p = \mathtt{:arg}$ implies $\exists C \in \text{E}(G) : \{o\} = \mathsf{cq}(C)\}$*. We define the minimisation of* G *as* $\text{M}(G) = \bigcup_{G' \in \text{E}(G)} L^{-}(G') \cup P \cup U$*, where* $L^{-}(G')$ *denotes the replacement of variable IRIs with their original blank nodes.* □

The result is an R-graph representing a redundancy-free UCQ.

4.4 Canonical Labelling and Query Generation

We take the minimal R-graph $\text{E}(G)$ generated by the previous methods and apply the iso-canonicalisation method $\text{iCan}(\text{E}(G))$ to generate canonical labels for the blank nodes in $\text{E}(G)$; having normalised the UCQ algebra and removed redundancy, applying this process will finally abstract away the naming of variables in the original query from the R-graph. Then we are left to map from the R-graph back to a query, which we do by applying $\text{R}^{-}(\text{iCan}(\text{E}(G)))$; in $\text{R}^{-}(\cdot)$, we order triple patterns in CQs, CQs in UCQs and variables in the projection lexicographically. The result is the final canonicalised UCQ in SPARQL syntax. Soundness and completeness results for MQs are given in the extended version [20].

4.5 Other Features

We can represent other (non-MQ) features of SPARQL (e.g., filters, optional, etc.) as an R-graph in an analogous manner to that presented here; thereafter, we can apply canonical labelling over that graph without affecting the semantics of the underlying query. However, we must be cautious with UCQ rewriting and minimisation techniques. Currently in queries with non-UCQ features, we detect subqueries that are UCQs (i.e., use only join and union) and apply normalisation only on those UCQ subqueries considering any variable also used outside the UCQ as a virtual projected variable. Combined with canonical labelling, this provides a cautious (i.e., sound but incomplete) canonicalisation of non-MQ queries.

4.6 Implementation

We implement the described canonicalisation procedure using two main libraries: Jena for parsing and executing SPARQL queries; and Blabel for computing the core of RDF graphs and applying canonical labelling. The containment checks

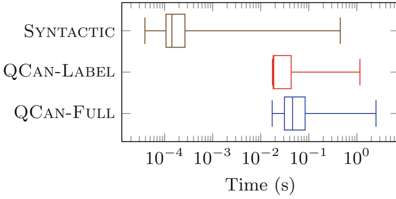


Fig. 2. Runtimes for LSQ queries

Table 1. High-level results for canonicalising LSQ queries, including the total time taken and (max) duplicates (**D.**) found

Algorithm	Time (s)	D.	Max.D.	Queries
Syntactic	211	3,960	12	768,618
QCan-Label	28,066	10,722	40	768,618
QCan-Full	77,022	10,722	40	768,618

over CQs are implemented using SPARQL ASK queries (with Jena). In the following, we refer to our system as QCan: Query CANonicalisation. Source code is available at https://github.com/RittoShadow/QCan, while a simple online demo can be found at http://qcan.dcc.uchile.cl/.

5 Evaluation

We now evaluate the proposed canonicalisation procedure for monotone SPARQL queries. In particular, the main research questions to be empirically assessed are as follows. RQ1: *How is the performance of canonicalisation?* RQ2: *How many additional duplicate queries can the canonicalisation process expect to find versus baseline syntactic methods in a real-world setting?* To address these questions, we present two experimental settings. In the first setting, we apply our canonicalisation method over queries from the Linked SPARQL Queries (LSQ) dataset [21], which contains queries taken from the logs of four public SPARQL endpoints. In the second setting, we create a benchmark of more difficult synthetic queries designed to stress-test the process. All experiments were run on a single machine with two Intel Xeon E5-2609 V3 CPUs and 32 GB of RAM running Debian v.7.11.

5.1 Real-World Setting

In the first setting, we perform experiments over queries from endpoint logs taken from the LSQ dataset [21], where we extract the unique strings for `SELECT` queries that could be parsed successfully by Jena (i.e., that were syntactically valid), resulting in 768,618 queries (see the extended version [20] for details). Over these queries, we then apply three experiments for increasingly complete and expensive canonicalisation, as follows. Syntactic: We pass the query through the Jena SPARQL parser and serialiser, parsing the query into an abstract algebra and then writing the algebraic query back to a SPARQL query. QCan-Label: We parse the query, applying canonical labelling to the query variables and reordering triple patterns according to the order of the canonical labels. QCan-Full: We apply the entire canonicalisation procedure, including parsing, labelling, UCQ rewriting, minimisation, etc. We can now address our research questions.

(RQ1:) Per Table 1, canonicalising with QCan-Label is 127 times slower than the baseline Syntactic method, while QCan-Full is 365 times slower

than Syntactic and 2.7 times slower than QCan-Label; however, even for the slowest method QCan-Full, the mean canonicalisation time per query is a relatively modest 100 ms. In more detail, Fig. 2 provides boxplots for the runtimes over the queries; we see that most queries under the Syntactic canonicalisation generally take around 0.1–0.3 ms, while most queries under QCan-Label and QCan-Full take 10–100 ms. We did, however, find queries requiring longer: approximately 2.5 s in isolated worst cases for QCan-Full.

(RQ2:) Canonicalising with QCan-Label finds 2.7 times more duplicates than the baseline Syntactic method. On the other hand, canonicalising with QCan-Full finds no more duplicates than QCan-Label: we believe that this observation can be explained by the relatively low ratio of true MQ queries in the logs [20], and the improbability of finding redundant patterns in real queries. The largest set of duplicate queries found was 12 in the case of Syntactic and 40 in the case of QCan-Label and QCan-Full.

5.2 Synthetic Setting

Many queries found in the LSQ dataset are quite simple to canonicalise. In order to see how the proposed canonicalisation methods perform for more complex queries, we propose two categories of synthetic query: the first category is designed to test the canonicalisation of CQs, particularly the canonical labelling and intra-CQ minimisation steps; the second category is designed to test the canonicalisation of UCQs, particularly the UCQ rewriting and inter-CQ minimisation steps. Both aim at testing performance rather than duplicates found.

Synthetic CQ Setting. In order to test the minimisation of CQs, we select difficult cases for the canonical labelling and core computation of graphs [12]. More specifically, we select the following three (undirected) graph schemas:

2D grids: For $k \geq 2$, the k-2D-grid contains k^2 nodes, each with a coordinate $(x, y) \in \mathbb{N}^2_{1...k}$, where nodes with distance one are connected; the result is a graph with $2(k^2 - k)$ edges.

3D grids: For $k \geq 2$, the k-3D-grid contains k^3 nodes, each with a coordinate $(x, y, z) \in \mathbb{N}^3_{1...k}$, where nodes with distance one are connected; the result is a graph with $3(k^3 - k^2)$ edges.

Miyazaki: This class of graphs was designed by Miyazaki [15] to enforce a worst-case exponential behaviour in Nauty-style canonical labelling algorithms. For k, each graph has $20k$ nodes and $30k$ edges.

To create CQs from these graphs, we represent each edge in the undirected graph by a pair of triple patterns $(v_i, :\mathtt{p}, v_j)$, $(v_j, :\mathtt{p}, v_i)$, with $v_i, v_j \in \mathbf{V}$ and $:\mathtt{p}$ a fixed IRI for all edges. In order to ensure that the canonicalisation involves CQ minimisation, we enclose the graph pattern in a `SELECT DISTINCT` v query, which provides the most challenging case for canonicalisation: applying set semantics and projecting (and thus "fixing") a single query variable v. We then run the Full canonicalisation feature, which for CQs involves computing the core

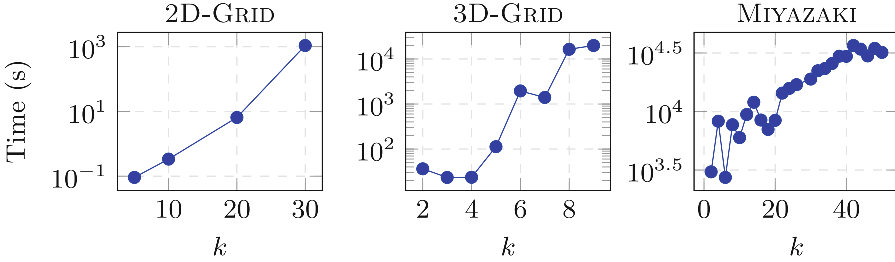


Fig. 3. Runtimes for threes types of synthetic CQs

of the R-graph and applying canonical labelling. Note that under minimisation, 2D-Grid and 3D-Grid graphs collapse down to a core with a single undirected edge, while Miyazaki graphs collapse down to a core with a 3-cycle.

In Fig. 3 we present the runtimes of the canonicalisation procedure, where we highlight that the y-axis is presented in log scale. We see that instances of 2D-Grid for $k \leq 10$ can be canonicalised in under a second. Beyond that, the performance of canonicalisation lengthens to seconds, minutes and even hours.

Synthetic MQ Setting. We also performed tests creating MQs in CNF (joins of unions) of the form $(t_{1,1} \cup \ldots \cup t_{1,n}) \bowtie \ldots \bowtie (t_{m,1} \cup \ldots \cup t_{m,n})$, where m is the number of joins, n is the number of unions, and $t_{i,j}$ is a triple pattern sampled (with replacement) from a k-clique of triples with a fixed predicate (such that $k = m + n$) to stress-test the performance of the canonicalisation procedure, where each such query will be rewritten to a query of size $O(n^m)$. Detailed results are available in the extended paper [20]; in summary, QCan-Full succeeds up to $m = 4$, $n = 8$, taking about 7.4 h, or $m = 8$, $n = 2$, taking 3 min; for values of $m = 8$, $n = 4$ and beyond, canonicalisation fails.

6 Conclusions

This paper describes a method for canonicalising SPARQL (1.0) queries considering both set and bag semantics. This canonicalisation procedure – which is sound for all queries and complete for monotone queries – obviates the need to perform pairwise containment/equivalence checks in a list/stream of queries and rather allows for using standard indexing techniques to find congruent queries. The main use-cases we foresee are query caching, optimisation and log analysis.

Our method is based on (1) representing the SPARQL query as an RDF graph, over which are applied (2) algebraic UCQ rewritings, (3 – in the case of set semantics) intra-CQ and inter-CQ normalisation, (4) canonical labelling of variables and ordering of query syntax, before finally (5) converting the graph back to a canonical SPARQL query. As such, by representing the query as a graph, our method leverages existing graph canonicalisation frameworks [12].

Though the worst-case complexity of the algorithm is doubly-exponential, experiments show that canonicalisation is feasible for a large collection of real-world SPARQL queries taken from endpoint logs. Furthermore, we show that the number of duplicates detected doubles over baseline syntactic methods. In more challenging experiments involving synthetic settings, however, we quickly start to encounter doubly-exponential behaviour, where the canonicalisation method starts to reach its practical limits. Still, our experiments for real-world queries suggests that such difficult cases do not arise often in practice.

In future work, we plan to extend our methods to consider other query features of SPARQL (1.1), such as subqueries, property paths, negation, and so forth; we also intend to investigate further into the popular `OPTIONAL` operator.

Acknowledgements. The work was supported by the Millennium Institute for Foundational Research on Data (IMFD) and by Fondecyt Grant No. 1181896.

References

1. Angles, R., Gutierrez, C.: The multiset semantics of SPARQL patterns. In: Groth, P., et al. (eds.) ISWC 2016. LNCS, vol. 9981, pp. 20–36. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46523-4_2
2. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.-Y.: SPARQL web-querying infrastructure: ready for action? In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8219, pp. 277–293. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41338-4_18
3. Arias Gallego, M., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. In: Usage Analysis and the Web of Data (USEWOD) (2011)
4. Chaudhuri, S., Vardi, M.Y.: Optimization of real conjunctive queries. In: Principles of Database Systems (PODS), pp. 59–70. ACM Press (1993)
5. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: SPARQL query containment under SHI axioms. In: AAAI Conference on Artificial Intelligence (2012)
6. Wudage Chekol, M., Euzenat, J., Genevès, P., Layaïda, N.: Evaluating and benchmarking SPARQL query containment solvers. In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8219, pp. 408–423. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41338-4_26
7. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, February 2014. http://www.w3.org/TR/rdf11-concepts/
8. Gutierrez, C., Hurtado, C.A., Mendelzon, A.O., Pérez, J.: Foundations of semantic web databases. J. Comput. Syst. Sci. **77**(3), 520–541 (2011)
9. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C Recommendation, March 2013. http://www.w3.org/TR/sparql11-query/
10. Hayes, P., Patel-Schneider, P.F.: RDF 1.1 Semantics. W3C Recommendation, February 2014. http://www.w3.org/TR/rdf11-mt/
11. Hogan, A.: Skolemising blank nodes while preserving isomorphism. In: World Wide Web Conference (WWW), pp. 430–440. ACM (2015)
12. Hogan, A.: Canonical forms for isomorphic and equivalent RDF graphs: algorithms for leaning and labelling blank nodes. ACM TWeb **11**(4), 22:1–22:62 (2017)

13. Kaminski, M., Kostylev, E.V.: Beyond well-designed SPARQL. In: International Conference on Database Theory (ICDT), pp. 5:1–5:18 (2016)
14. Letelier, A., Pérez, J., Pichler, R., Skritek, S.: Static analysis and optimization of semantic web queries. ACM Trans. Database Syst. **38**(4), 25:1–25:45 (2013)
15. Miyazaki, T.: The complexity of McKay's canonical labeling algorithm. In: Groups and Computation, II, pp. 239–256 (1997)
16. Papailiou, N., Tsoumakos, D., Karras, P., Koziris, N.: Graph-aware, workload-adaptive SPARQL query caching. In: ACM SIGMOD International Conference on Management of Data, pp. 1777–1792. ACM (2015)
17. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16:1–16:45 (2009)
18. Pichler, R., Skritek, S.: Containment and equivalence of well-designed SPARQL. In: Principles of Database Systems (PODS), pp. 39–50 (2014)
19. Sagiv, Y., Yannakakis, M.: Equivalences among relational expressions with the union and difference operators. J. ACM **27**(4), 633–655 (1980)
20. Salas, J., Hogan, A.: Canonicalisation of monotone SPARQL queries. Technical report. http://aidanhogan.com/qcan/extended.pdf
21. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.-C.N.: LSQ: the linked SPARQL queries dataset. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 261–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_15
22. Saleem, M., Stadler, C., Mehmood, Q., Lehmann, J., Ngomo, A.N.: SQCFramework: SPARQL query containment benchmark generation framework. In: Knowledge Capture Conference (K-CAP), pp. 28:1–28:8 (2017)
23. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: International Conference on Database Theory (ICDT), pp. 4–33. ACM (2010)
24. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking database representations of RDF/S stores. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 685–701. Springer, Heidelberg (2005). https://doi.org/10.1007/11574620_49