Canonicalisation of Monotone SPARQL Queries (*Extended Version*)

Jaime Salas and Aidan Hogan

IMFD Chile & Department of Computer Science, University of Chile

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 [18] (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* 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 [17]. 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 [19]. 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* [20] 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: This extended version provides additional definitions, proofs and results in the Appendix.

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 [11] where two RDF graphs differing only in blank node labels are thus considered *isomorphic* [8]. 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 [12], 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 [11]. 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 $\mu_2(G_2) \subseteq G_1$ [9]. 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 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 [9]. Determining equivalence between RDF graphs is known to be NP-complete [9]. Determining if a graph G is lean is known to

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

be coNP-complete [9]. Finally, determining if a graph G' is the core of a second graph G is known to be DP-complete [9].

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 [13]; 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 [10]. For brevity, our definitions focus on SPARQL *monotone queries* (MQs) [20] – 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:

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

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.

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

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 [18]. However, the SPARQL standard, by default, considers a *bag* (aka. *multiset*) *semantics* for query answering [10], 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 $\text{SELECT}^{\Delta}_V(Q)$, where $\Delta = \texttt{true}$ indicates set semantics and $\Delta = \texttt{false}$ 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 [19]. For MQs, containment and equivalence are NP-complete for the related query class of *Unions of Conjunctive Queries* (UCQs) [20], which allow the same features as MQs but disallow joins over unions. Interestingly, though MQs and UCQs are equivalent query classes – for any UCQ there is an equivalent MQ and vice-versa – containment and equivalence for MQs jumps to Π^P_2-complete [20]. 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 [5]; on the other hand, the equivalence problem is GI-complete [5], 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 [25,24,6,15,19,14]. With respect to implementations, only one dedicated library has been released to check whether or not two SPARQL queries are equivalent: SPARQL Algebra [15]. The problem of determining equivalence of SPARQL queries can, however, be solved by reductions to related problems, where Chekol et al. [7] have used a μ-calculus solver and an XPath-equivalence checker to implement SPARQL equivalence checks. Recently Saleem et al. [23] 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 [3,22] 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. [17], 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. [17] 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

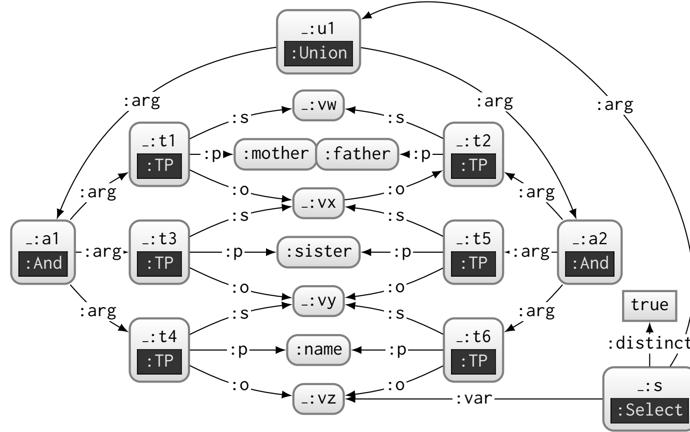
The MQ class is closed under join and union (see Q_A, Example 1). As the first query normalisation step, we will convert MQ queries to UCQs of the form $\mathsf{select}^{\Delta}_{V}(\mathsf{union}(\{\mathsf{and}(\{Q^1_1, \ldots Q^1_m\}), \ldots, \mathsf{and}(\{Q^k_1, \ldots Q^k_n\})\}))$ following a standard DNF-style expansion (we refer to Appendix A for more details). 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 $\mathrm{R}(Q)$, the R-graph of Q, as follows:*

- *If Q is a triple pattern (s, p, o), then $\iota(Q)$ is initialised as $\beta()$ and*
  $\mathrm{R}(Q) = \{(\iota(Q), \texttt{:s}, \iota(s)), (\iota(Q), \texttt{:p}, \iota(p)), (\iota(Q), \texttt{:o}, \iota(o)), (\iota(Q), \mathsf{a}, \texttt{:TP})\}$
- *If Q is $\mathsf{and}(\{Q_1, \ldots, Q_n\})$, then $\iota(Q)$ is initialised as $\beta()$ and*
  $\mathrm{R}(Q) = \{(\iota(Q), \texttt{:arg}, \iota(Q_1)), \ldots, (\iota(Q), \texttt{:arg}, \iota(Q_n)), (\iota(Q), \mathsf{a}, \texttt{:And})\}$
- *If Q is $\mathsf{union}(\{Q_1, \ldots, Q_n\})$, then $\iota(Q)$ is initialised as $\beta()$ and*
  $\mathrm{R}(Q) = \{(\iota(Q), \texttt{:arg}, \iota(Q_1)), \ldots, (\iota(Q), \texttt{:arg}, \iota(Q_n)), (\iota(Q), \mathsf{a}, \texttt{:Union})\}$
- *If Q is $\mathsf{select}^{\Delta}_{V}(Q_1)$, then $\iota(Q)$ is initialised as $\beta()$ and*
  $\mathrm{R}(Q) = \{(\iota(Q), \texttt{:arg}, \iota(Q_1)), (\iota(Q), \texttt{:distinct}, \Delta), (\iota(Q), \mathsf{a}, \texttt{:Select})\}$
  $\cup \{(\iota(Q), \texttt{:var}, \iota(v)) \mid v \in V\}$

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; this results in a query equivalent to the input query (see Appendix B).

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 [20].

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 $\mathsf{cq}(G) = \{y \mid (y, \mathsf{a}, \mathsf{:And}) \in G\}$. 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 $\mathsf{var}(G) = \{v \in \mathbf{B} \mid \exists(s,p) : (s,p,v) \in G \wedge p \in \{\mathsf{:s}, \mathsf{:p}, \mathsf{:o}\}\}$, and we denote the blank nodes representing unprojected variables in G by $\mathsf{uvar}(G) = \{v \in \mathsf{var}(G) \mid \nexists s : (s, \mathsf{:var}, v) \in G\}$. 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 $\mathrm{R}(Q)$ for a UCQ Q. We define the variable distinguishing function $\mathrm{D}(G)$ as follows. If there does not exist a blank node x such that $(x, \mathsf{a}, \mathsf{:Union}) \in G$, then $\mathrm{D}(G) = G$. Otherwise if such a blank node exists, we define $\mathrm{D}(G) = \{(s,p,\delta(o)) \mid (s,p,o) \in G\}$, 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, $\mathrm{D}(G)$ 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 [5]. 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 $\textsc{d}(\textsc{r}(Q))$ 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, $\lambda(o) =$ `:o` for `: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 $\textsc{d}(\textsc{r}(Q))$. We define the* CQ*-minimisation of G as $\textsc{c}(G) = \{\mathsf{core}(L(G[x])) \mid x \in \mathsf{cq}(G)\}$. We call $C \in \textsc{c}(G)$ 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 . } }
```

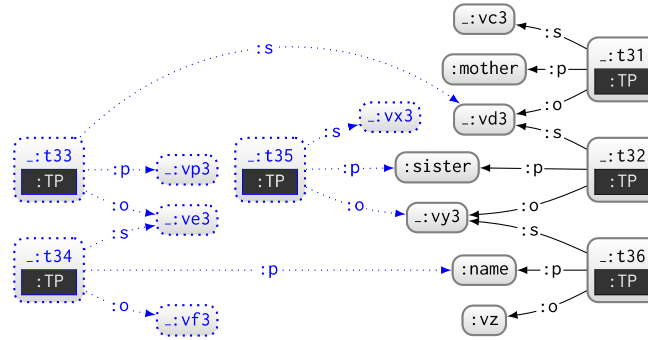


Fig. 1. R-graph of a CQ showing minimisation by leaning

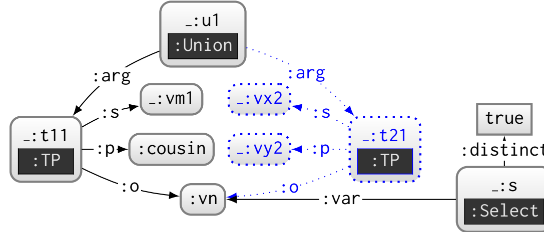
We see that the projected variable is preserved in all CQs. However, we are still left with (inter-CQ) redundancy between the first and third CQs. □

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 [20].

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 [20]. 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 $\mathrm{C}(G)$ 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 [13], removing any other Q_j ($j > i$) whose G_j is isomorphic. Thereafter, to implement condition (2), we can check if 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*: $G_i \models G_j$ [9]).

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

Definition 5. *Let $\mathrm{E}(G) = \{C_1, \ldots, C_n\}$ denote the minimal CQ cores of $G = \mathrm{D}(\mathrm{R}(Q))$. Let $P = \{(s,p,o) \in G \mid \exists(s, \mathsf{a}, \texttt{:Select}) \in G\}$ and $U = \{(s,p,o) \in G \mid \exists(s, \mathsf{a}, \texttt{:Union}) \in G$, and $p = \texttt{:arg}$ implies $\exists C \in \mathrm{E}(G) : \{o\} = \mathsf{cq}(C)\}$. We define the minimisation of G as $\mathrm{M}(G) = \bigcup_{G' \in \mathrm{E}(G)} L^-(G') \cup P \cup U$, 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 $\mathrm{E}(G)$ generated by the previous methods and apply the iso-canonicalisation method $\mathrm{ICan}(\mathrm{E}(G))$ to generate canonical labels for the blank nodes in G; having normalised the UCQ algebra and removed redundancy, applying this process will finally abstract away the naming of variables in the

[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 iff the data match.

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 Appendix D.

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 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 [22], 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 32GB 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 [22], where we extract the unique strings for `SELECT`

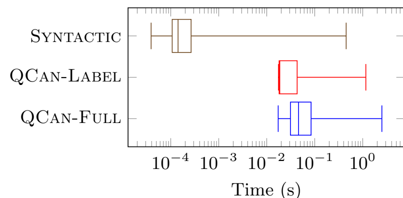


Fig. 2. Runtimes for LSQ queries

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

queries that could be parsed successfully by Jena (i.e., that were syntactically valid), resulting in 768,618 queries (see Appendix E for more 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, Figure 2 provides boxplots for the runtimes over the queries; we see that most queries under the Syntactic canonicalisation generally take around 0.1–0.3ms, while most queries under QCan-Label and QCan-Full take 10–100 ms. We did, however, find queries requiring longer: approximately 2.5 seconds 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, 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.

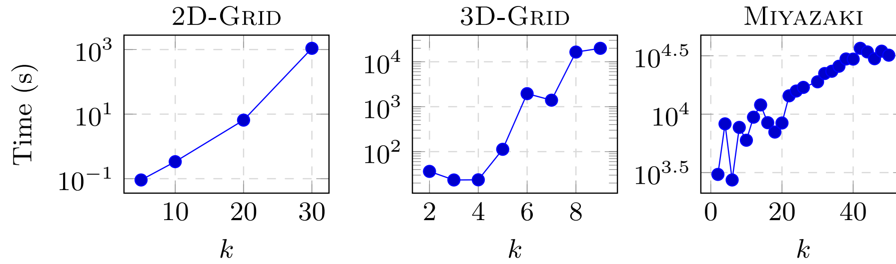


Fig. 3. Runtimes for threes types of synthetic CQs

Synthetic CQ setting In order to test the minimisation of CQs, we select difficult cases for the canonical labelling and core computation of graphs [13]. 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 [16] 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 $(v_i, \texttt{:p}, v_j)$, $(v_j, \texttt{:p}, v_i)$, with $v_i, v_j \in \mathbf{V}$ and `: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 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 Figure 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 Appendix F; in summary, QCAN-FULL succeeds up to

$m = 4$, $n = 8$, taking about 7.4 hours, or $m = 8$, $n = 2$, taking 3 minutes; 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 [13].

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 also 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: International Semantic Web Conference (ISWC). pp. 20–36. Springer (2016)
2. Aranda, C.B., Hogan, A., Umbrich, J., Vandenbussche, P.: SPARQL web-querying infrastructure: Ready for action? In: International Semantic Web Conference (ISWC). pp. 277–293 (2013)
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. Chandra, A.K., Merlin, P.M.: Optimal Implementation of Conjunctive Queries in Relational Data Bases. In: ACM Symposium on Theory of Computing (STOC). pp. 77–90 (1977)
5. Chaudhuri, S., Vardi, M.Y.: Optimization of *Real* Conjunctive Queries. In: Principles of Database Systems (PODS). pp. 59–70. ACM Press (1993)

6. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: SPARQL query containment under SHI axioms. In: AAAI Conference on Artificial Intelligence (2012)
7. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: Evaluating and benchmarking SPARQL query containment solvers. In: International Semantic Web Conference (ISWC). pp. 408–423. Springer (2013)
8. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (Feb 2014), http://www.w3.org/TR/rdf11-concepts/
9. Gutierrez, C., Hurtado, C.A., Mendelzon, A.O., Pérez, J.: Foundations of Semantic Web databases. J. Comput. Syst. Sci. 77(3), 520–541 (2011)
10. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C Recommendation (Mar 2013), http://www.w3.org/TR/sparql11-query/
11. Hayes, P., Patel-Schneider, P.F.: RDF 1.1 Semantics. W3C Recommendation (Feb 2014), http://www.w3.org/TR/rdf11-mt/
12. Hogan, A.: Skolemising Blank Nodes while Preserving Isomorphism. In: World Wide Web Conference (WWW). pp. 430–440. ACM (2015)
13. Hogan, A.: Canonical forms for isomorphic and equivalent RDF graphs: Algorithms for leaning and labelling blank nodes. ACM TWeb 11(4) (2017)
14. Kaminski, M., Kostylev, E.V.: Beyond well-designed SPARQL. In: International Conference on Database Theory (ICDT). pp. 5:1–5:18 (2016)
15. 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)
16. Miyazaki, T.: The Complexity of McKay's Canonical Labeling Algorithm. In: Groups and Computation, II. pp. 239–256 (1997)
17. 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)
18. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34(3) (2009)
19. Pichler, R., Skritek, S.: Containment and equivalence of well-designed SPARQL. In: Principles of Database Systems (PODS). pp. 39–50 (2014)
20. Sagiv, Y., Yannakakis, M.: Equivalences among relational expressions with the union and difference operators. J. ACM 27(4), 633–655 (1980)
21. Salas, J., Hogan, A.: Canonicalisation of Monotone SPARQL Queries. Technical Report, http://aidanhogan.com/qcan/extended.pdf
22. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: the linked SPARQL queries dataset. In: International Semantic Web Conference (ISWC) (2015)
23. 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)
24. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: International Conference on Database Theory (ICDT). pp. 4–33. ACM (2010)
25. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking database representations of RDF/S stores. In: International Semantic Web Conference (ISWC). pp. 685–701 (2005)

A Algebra of UCQs and UCQ-normalisation of MQs

We first define the syntax of a UCQ:

Definition 6. *The syntax of a* UCQ *query is as follows:*

1. *If* $\{t_1, \ldots, t_n\}$ *is a set of triple patterns* ($n \geq 1$)*, then* $\mathsf{and}(\{t_1, \ldots, t_n\})$ *is a query pattern called a conjunctive query (*CQ*) pattern.*
2. *If* $\{C_1, \ldots, C_n\}$ *is a bag of* CQ *patterns* ($n \geq 1$)*, then* $\mathsf{union}(\{C_1, \ldots, C_n\})$ *is a (*UCQ*) query pattern.*
3. *If* Q *is a* UCQ *query pattern and* V *is a set of variables such that for all* $v \in V$*,* v *appears in some pattern contained in* Q*, then* $\mathsf{select}_V^{\Delta}(Q)$ *is a query.* □

If Q is a query in UCQ syntax, we denote by $Q(G)$ the evaluation of Q over G, which we now define.

Definition 7. *Letting* Q *be a query in* UCQ *syntax and* G *an RDF graphs, then the semantics of a* UCQ *query is defined as follows:*

$$\begin{aligned}
t(G) &:= \{\mu \mid \mu(t) \in G, \mathrm{dom}(\mu) = \mathbf{V}(t)\} \\
\mathsf{and}(\{t\})(G) &:= t(G) \\
\mathsf{and}(\{t_1, \ldots, t_n\})(G) &:= t_1(G) \bowtie \ldots \bowtie t_n(G) \\
\mathsf{union}(\{C_1, \ldots, C_n\})(G) &:= C_1(G) \cup \ldots \cup C_n(G) \\
\mathsf{select}_V^{\Delta}(Q)(G) &:= \pi_V(Q(G))
\end{aligned}$$

where μ *denotes a query solution,* $\{t_1, \ldots, t_n\}$ *denotes a set of triple patterns, and* $\{C_1, \ldots, C_n\}$ *denotes a bag of CQs, with each* C_i *(for* $1 \leq i \leq n$*) being of the form* $\mathsf{and}(T_i)$*, where* T_i *is a set of triple patterns.* □

As before, $\mathsf{select}_V^{\Delta}(Q)$ then allows to state whether or not the query should be evaluated under set or bag semantics. In case that $\Delta = \mathsf{false}$, then $\cup$ is defined as bag-union, adding the multiplicity of repeated results.

The UCQ syntax has the same expressivity as monotone queries: for every monotone query Q, there is a UCQ Q' such that $Q \equiv Q'$ (follows from [18, Prop. 3.8]). While in practical systems queries are often expressed in the monotone syntax (which can be more concise per Example 1), the UCQ syntax is far more restricted, which leads us closer to a semantically-equivalent normal form. Hence we now describe a procedure to convert from a monotone query to a UCQ query.

Definition 8. *For an* MQ Q*, we define the* UCQ *rewriting of* Q*, denoted* $\mathrm{U}(Q)$*, as follows:*

1. *If* Q *is a triple pattern* t*, then* $\mathrm{U}(Q) = \mathsf{and}(\{t\})$*.*
2. *If* Q *is* $[Q_1 \text{ AND } Q_2]$*, then* $\mathrm{U}(Q) = \mathrm{U}^*([\mathrm{U}(Q_1) \text{ AND } \mathrm{U}(Q_2)])$*.*
3. *If* Q *is* $[Q_1 \text{ UNION } Q_2]$*, then* $\mathrm{U}(Q) = \mathrm{U}^*([\mathrm{U}(Q_1) \text{ UNION } \mathrm{U}(Q_2)])$*.*
4. *If* Q *is* $\text{SELECT}_V^{\Delta}(Q')$*, then* $\mathrm{U}(Q) = \mathsf{select}_V^{\Delta}(\mathrm{U}(Q'))$

where, in turn, $\text{U}^*(Q)$ *is defined as follows (where* T *denotes a set of triple patterns and* $\mathbb{C}$ *denotes a bag of CQs of the form* $\{\mathsf{and}(T_1), \ldots, \mathsf{and}(T_n)\}$*):*

5. *If* Q *is* $[\mathsf{and}(T_1)\ \text{AND}\ \mathsf{and}(T_2)]$ *then* $\text{U}^*(Q) = \mathsf{and}(T_1 \cup T_2)$*.*
6. *If* Q *is* $[\mathsf{and}(T)\ \text{AND}\ \mathsf{union}(\mathbb{C})]$ *or* $[\mathsf{union}(\mathbb{C})\ \text{AND}\ \mathsf{and}(T)]$ *then* $\text{U}^*(Q) = \mathsf{union}(\{\mathsf{and}(T \cup T') \mid \mathsf{and}(T') \in \mathbb{C}\})$*.*
7. *If* Q *is* $[\mathsf{union}(\mathbb{C}_1)\ \text{AND}\ \mathsf{union}(\mathbb{C}_2)]$ *then* $\text{U}^*(Q) = \mathsf{union}(\{\mathsf{and}(T_1 \cup T_2) \mid \mathsf{and}(T_1) \in \mathbb{C}_1 \wedge \mathsf{and}(T_2) \in \mathbb{C}_2\})$*.*
8. *If* Q *is* $[\mathsf{union}(\mathbb{C}_1)\ \text{UNION}\ \mathsf{union}(\mathbb{C}_2)]$ *then* $\text{U}^*(Q) = \mathsf{union}(\mathbb{C}_1 \cup \mathbb{C}_2)$*.*
9. *If* Q *is* $[\mathsf{and}(T)\ \text{UNION}\ \mathsf{union}(\mathbb{C})]$ *or* $[\mathsf{union}(\mathbb{C})\ \text{UNION}\ \mathsf{and}(T)]$ *then* $\text{U}^*(Q) = \mathsf{union}(\{\mathsf{and}(T)\} \cup \mathbb{C})$*.*
10. *If* Q *is* $[\mathsf{and}(T_1)\ \text{UNION}\ \mathsf{and}(T_2)]$ *then* $\text{U}^*(Q) = \mathsf{union}(\{\mathsf{and}(T_1), \mathsf{and}(T_2)\})$*.*

In the case of bag semantics, for point (7), the multiplicity of $\mathsf{and}(T_1 \cup T_2)$ *is given by* $\mathbb{C}_1(T_1) \times \mathbb{C}_2(T_2)$*, where* $\mathbb{C}_1(T_1)$ *and* $\mathbb{C}_2(T_2)$ *denote the multiplicities of* T_1 *and* T_2 *in their respective bags; for (8) bag union is applied for* $\mathbb{C}_1 \cup \mathbb{C}_2$ *adding multiplicities. Under set semantics, set union can be applied throughout.* □

Example 6. Referring back to Example 1, Q_B corresponds to an equivalent version of Q_A in UCQ normal form where we have a union over joins, but never a join over unions. □

Lemma 1. *Given a monotone query* Q*, then* $Q \equiv \text{U}(Q)$*.*

Proof. By inspection on the rewriting steps of Definition 8: (1) $\mathsf{and}(\{t\})(G)$ is directly defined as $t(G)$; (2) this is a recursive step, (3) this is a recursive step, (4) the evaluation of projection does not change from MQ to UCQ. Regarding the recursive steps (2,3), note that (5,8,9,10) follow from the commutativity and associativity of joins and unions. Regarding (6,7) for set semantics, note that (7) follows for set semantics from the normal form for SPARQL proved in [18, Lemma 2.5], while (6) is a special sub-case of (7) with a (virtual) union of a single CQ on one side. Regarding (6) for bag semantics, the multiplicities of join solutions for SPARQL are defined as the sum of the product of the multiplicity of the compatible solutions on both sides of the join generating that join solution [1], which the defined multiplicity preserves; on the other hand, regarding (7) for bag semantics, the multiplicities of SPARQL union are the sum of the multiplicities of the solutions from both sides [1], which the defined multiplicity preserves. Regarding recursive steps, the proof is concluded by induction on the structure of the individual rewriting steps in Definition 8. □

We highlight that given an MQ Q, the UCQ resulting from $\text{U}(Q)$ can be exponential in the number of (non-unique) triple patterns in the query. The worst case is given by an input MQ query of the form $Q = [Q_1\ \text{AND}\ [Q_2\ \text{AND}\ [\ldots\ \text{AND}\ Q_m]]]$ where, for $1 \leq i \leq m$, $Q_i = [t_{i,1}\ \text{UNION}\ [t_{i,2}\ \text{UNION}\ [\ldots\ \text{UNION}\ t_{i,n}]]]$. Note that the number of non-unique triple patterns in Q is of the order $O(m \times n)$. However, $\text{U}(Q)$ expands to the UCQ $\mathsf{union}(\{\mathsf{and}(\{t_1, \ldots, t_m\}) \mid t_1 \in Q_1 \wedge \ldots \wedge t_m \in Q_m\})$. With n ways to choose any of $t_1, \ldots t_m$, this query contains n^m triple patterns. This is analogous to the exponential explosion of DNF rewritings.

Please note that for brevity, the body of the paper assumes that Q has been written to $\text{U}(Q)$ as a prior step (i.e., the $\text{U}(\cdot)$ is left implicit).

B Inverse Mapping from an R-graph

We now define the inverse mapping of an R-graph to a query. For brevity, we will define the case that the R-graph is in UCQ normal form and will be mapped back to a UCQ; however, the inverse mapping can be extended naturally for other forms of R-graph and other forms of queries.

Definition 9. *Let G denote the R-graph $\mathrm{R}(\mathrm{U}(Q))$ for a monotone query Q. Let $\top_G \in \mathbf{IBL}$ be the root of G such that there does not exist (s, p) such that $(s, p, \top_G) \in G$. Further let $G[z]$ denote the sub-graph of G rooted at z (see Section 4.2). Finally, let $\nu : \mathbf{ILB} \to \mathbf{ILV}$ be a function that is the identity on $\mathbf{IL}$ (i.e., for all $x \in \mathbf{IL}$, $\nu(x) = x$) and that maps $\mathbf{B}$ to $\mathbf{V}$ in a one-to-one manner. We then define the inverse R-graph operation $\mathrm{R}^-(G)$ to a UCQ query as follows:*

- *If there exists (s, p, o) such that*
  $\{(z, \mathsf{a}, \mathsf{:TP}), (z, \mathsf{:s}, s), (z, \mathsf{:p}, p), (z, \mathsf{:o}, o)\} \subseteq G$ *where* $z = \top_G$,
  then $\mathrm{R}^-(G) = (\nu(s), \nu(p), \nu(o))$.
- *If there exists $\{x_1, \ldots, x_n\}$ such that*
  $\{(z, \mathsf{a}, \mathsf{:And}), (z, \mathsf{:arg}, x_1), \ldots, (z, \mathsf{:arg}, x_n)\} \subseteq G$ *where* $z = \top_G$
  and there does not exist $(z, \mathsf{:arg}, x) \in G$ *such that* $x \notin \{x_1, \ldots, x_n\}$
  then $\mathrm{R}^-(G) = \mathsf{and}(\{\mathrm{R}^-(G[x_1]), \ldots, \mathrm{R}^-(G[x_n])\})$.
- *If there exists $\{x_1, \ldots, x_n\}$ such that*
  $\{(z, \mathsf{a}, \mathsf{:Union}), (z, \mathsf{:arg}, x_1), \ldots, (z, \mathsf{:arg}, x_n)\} \subseteq G$ *where* $z = \top_G$
  and there does not exist $(z, \mathsf{:arg}, x) \in G$ *such that* $x \notin \{x_1, \ldots, x_n\}$
  then $\mathrm{R}^-(G) = \mathsf{union}(\{\mathrm{R}^-(G[x_1]), \ldots, \mathrm{R}^-(G[x_n])\})$.
- *If there exists $(x, \Delta, \{v_1, \ldots, v_n\})$ (for $n \geq 1$) such that*
  $\{(z, \mathsf{a}, \mathsf{:Select}), (z, \mathsf{:arg}, x), (z, \mathsf{:distinct}, \Delta),$
  $(z, \mathsf{:var}, v_1), \ldots, (z, \mathsf{:var}, v_n)\} \subseteq G$ *where* $z = \top_G$
  and there does not exist $(z, \mathsf{:var}, v) \in G$ *such that* $v \notin \{v_1, \ldots, v_n\}$
  then $\mathrm{R}^-(G) = \mathsf{select}^{\Delta}_{\{\nu(v_1), \ldots, \nu(v_n)\}}(\mathrm{R}^-(G[x]))$. □

Lemma 2. *For a UCQ Q, it holds that $\mathrm{R}^-(\mathrm{R}(Q)) \cong Q$.*

Proof. Variables in $\mathrm{R}^-(\mathrm{R}(Q))$ are generated by $\nu(\beta(v))$ for all $v \in Q$; since both ν and β are one-to-one, then $\nu \circ \beta$ is one-to-one on $\mathbf{V}$ and thus variables in $\mathrm{R}^-(\mathrm{R}(Q))$ map one-to-one to variables in Q. With respect to the structure of the query, by inspection, each sub-graph $\mathrm{R}(Q)[x]$ induced by Definition 9 is rooted by $\iota(Q')$ in Definition 1 for Q' a subquery of Q and the triples matched by Definition 9 correspond to those generated by Definition 1. We conclude that $\mathrm{R}^-(\mathrm{R}(Q))$ preserves the structure of Q modulo variable names. Finally, by definition, a one-to-one change in variable names does not affect the congruence relation, where $\nu \circ \beta$ witnesses the congruence $\mathrm{R}^-(\mathrm{R}(Q)) \cong Q$. □

Lemma 3. *For an MQ Q, it holds that $\mathrm{R}^-(\mathrm{R}(\mathrm{U}(Q))) \cong Q$.*

Proof. For the MQ Q, Lemma 1 implies that $\mathrm{U}(Q) \equiv Q$. Lemma 2 further implies that $\mathrm{R}^-(\mathrm{R}(\mathrm{U}(Q))) \cong \mathrm{U}(Q)$ since $\mathrm{U}(Q)$ is a UCQ. Given $\mathrm{R}^-(\mathrm{R}(\mathrm{U}(Q))) \cong \mathrm{U}(Q) \equiv Q$, we can then conclude that $\mathrm{R}^-(\mathrm{R}(\mathrm{U}(Q))) \cong Q$. □

We remark that the inverse mapping $\text{R}^{-}(\cdot)$ generates a query in abstract UCQ-syntax, which contains three types of unordered elements: an unordered set of projected variables, an unordered set of triple patterns in each CQ, and an unordered bag of CQs in the UCQ. If we wish to create a canonical query in concrete SPARQL syntax (i.e., a SPARQL query string), we must define and apply a deterministic ordering within each such set/bag. Later we will apply canonical labelling of blank nodes/variables, which can be used to provide a deterministic ordering of variables that captures isomorphism. With this ordering of variables, we can order the set of projected variables. Next we assume a total syntactic ordering over **VIL** (again, **B** is replaced by **V** in the query); with this, we can define a lexicographical ordering of triple patterns (ordered by subject, then predicate, then object), which allows us to order the elements of CQs. Finally, we can define an ordering of sets of triple patterns such that $T_1 \leq T_2$ if and only if $T_1 \subseteq T_2$ or $\min(T_1 \setminus T_2) < \min(T_2 \setminus T_1)$; this allows us to order CQs inside UCQs. With this, we can apply a deterministic ordering of all elements in the UCQ, allowing us to generate a canonical query in concrete SPARQL syntax.

C UCQ Minimisation Examples

We consider some examples to illustrate the minimisation process for UCQs focusing on the removal of redundant CQs. In the first example we consider, all CQs contain all projected variables (and possibly other non-projected variables).

Example 7. Consider the following example of a UCQ:

```
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 . }
}
```

If we consider the first two CQs, 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, which means that during minimisation, one of these CQs would have been removed. Temporarily grounding `:vn` before minimisation ensures they are no longer isomorphic. On the other hand, the R-graphs of the second and third CQ 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.

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

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

... and applying it to the sub-R-graph representing the first CQ:

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

This evaluates as true and hence the first CQ is removed. Likewise the fourth CQ maps to the fifth CQ and hence the fifth CQ 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. □

We provide a second example where, this time, some CQs do not contain all projected variables.

Example 8. Consider the following extended example:

```
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 $(Q_1 \cup \ldots \cup Q_6, V)$ denote this UCQ, respectively, with V_i denoting the projected variables in each CQ (e.g., $V_3 = \{\texttt{?v}\}$). If we partition the set of CQs by their projected variables V_i, we will end up with three sets of CQs as follows: $\{\{Q_1, Q_2\}, \{Q_3, Q_4\}, \{Q_5, Q_6\}\}$ partitioned 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 CQs can return multiple results, none containing an unbound; the third CQ will return the same results 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 both variables ?v and ?w if and only if the RDF graph is not empty. □

D Soundness and completeness

We provide some soundness and completeness results for the proposed canonicalisation scheme. We first give some initial results.

To begin, we note that distinguishing variables does not alter the semantics of the query underlying the R-graph:

Lemma 4. *For an* MQ Q*, it holds that* $\mathrm{R}^{-}(\mathrm{D}(\mathrm{R}(\mathrm{U}(Q)))) \cong Q$*.* □

Proof. The function $\text{D}(\cdot)$ rewrites variables in each CQ that are not projected away to fresh variables. Observing that for any RDF graph G, the expression $\pi_V(\nu_1(Q_1)(G) \cup ... \cup \nu_n(Q_n)(G))$ is equivalent to $\pi_V(Q_1(G) \cup ... \cup Q_n(G))$ so long as each $\nu_1, \ldots, \nu_n$ is a one-to-one variable mapping that is the identity on elements of V, we can conclude that $\text{R}^-(\text{D}(\text{R}(\text{U}(Q)))) \equiv \text{R}^-(\text{R}(\text{U}(Q)))$: the function $\text{D}(\cdot)$ has no effect on solutions. Lemma 3 further states that $\text{R}^-(\text{R}(\text{U}(Q))) \cong Q$, and hence we can conclude that $\text{R}^-(\text{D}(\text{R}(\text{U}(Q)))) \cong Q$. □

The following result establishes soundness: i.e., that the proposed canonicalisation procedure does not change the query results of the input query.

Theorem 1. *For an* MQ Q*, it holds that* $\text{R}^-(\text{ICan}(\text{M}(\text{D}(\text{R}(\text{U}(Q)))))) \cong Q$*.*

Proof. The relation $\text{R}^-(\text{ICan}(\text{D}(\text{R}(\text{U}(Q))))) \cong Q$ holds from Lemma 4, and the fact that $\text{ICan}(\cdot)$ relabels blank nodes in a one-to-one fashion, thus renaming variables in the output query in a one-to-one fashion (which by definition does not affect the $\cong$ relation). We are thus left to show that the minimisation of CQs and UCQs does not affect the semantics of the input query.

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 [4]. Likewise the minimisation of UCQs is covered by Sagiv and Yannakakis [20], 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 CQs may not contain all projected variables. This result is quite direct since a CQ C_1 cannot be contained in a CQ C_2 if they cover different projected variables: assuming that v is a projected variable appearing in C_1 but not in C_2, then v always maps to `UNBOUND` in C_2 (i.e., v is not in the domain of solutions for C_2), which can never occur for C_1, and hence C_1 cannot be contained in C_2, nor vice versa (in fact, they cannot share any solutions!). Hence checking containment within the CQ partitions formed by the projected variables they contain does not miss containments. The result for set semantics then follows from Sagiv and Yannakakis [20].

BAG SEMANTICS: neither CQs nor UCQs are minimised, and hence the result follows directly from Lemma 4. □

Finally we establish completeness: that for any two MQs, they are congruent if and only if their canonicalised queries are equal.

Theorem 2. *For two* MQ*s* Q_1 *and* Q_2*, it holds that* $Q_1 \cong Q_2$ *if and only if* $\text{R}^-(\text{ICan}(\text{M}(\text{D}(\text{R}(\text{U}(Q_1)))))) = \text{R}^-(\text{ICan}(\text{M}(\text{D}(\text{R}(\text{U}(Q_2))))))$*.*

Proof. SET SEMANTICS: The main idea is that given a monotone query Q, we show that the minimisation function $\text{M}(\cdot)$ will produce a minimal UCQ Q' that is unique for the set of monotone queries congruent to Q (i.e., equivalent modulo isomorphism to Q). For the minimisation of CQs, this follows directly from the fact that we compute their minimal form as the core of a graph, which is known to be unique modulo isomorphism [4,9]. For the minimisation of UCQs, we have

a non-deterministic choice when we choose one CQ from each quotient set of equivalent CQs with the same projected variables; however, since the CQs were previously minimised, all such equivalent CQs are isomorphic, and hence the choice is deterministic modulo isomorphism. For the last part, we need to show that the final UCQ produced is minimal. While we remove all CQs contained in another CQ – and all but one CQ in each quotient set of equivalent CQs – we are left to rule out the case that the results of a CQ Q_i may be contained in the SPARQL union of the results of two CQs Q_j and Q_k but where none of the queries are contained individually in the other (i.e., $Q_a \not\sqsubseteq Q_b$ for $a, b \in \{i, j, k\}$ and $a \neq b$). To rule out this case, we can consider the classical technique of producing a canonical database [4] for Q_i, which is produced by replacing every variable in Q_i with a unique fresh IRI; clearly there is then at least one solution for Q_i over the canonical database (mapping variables to their surrogate IRIs). Now if we consider the canonical databases for Q_i, we know that there cannot be a mapping (homomorphism) from Q_j nor Q_k to this database since otherwise containment would hold (since, e.g., $Q_i \sqsubseteq Q_j$ if and only if there is a homomorphism from Q_j to Q_i [4]). Hence the result of Q_i over this database cannot be contained in the result of $Q_j \cup Q_k$ over the same, which serves as a counter-example to the containment $Q_i \sqsubseteq Q_j \cup Q_k$. This concludes the argument that the minimisation of UCQs is indeed minimal and unique modulo isomorphism for equivalent MQs.

BAG SEMANTICS: The result for bag semantics follows from the fact that no minimisation (neither CQ nor UCQ) is possible since removing any triple pattern from a CQ or any CQ from a UCQ would directly affect the multiplicity of final results [5]. Hence the UCQ is unique modulo isomorphism for equivalent MQs.

CONCLUSION: We have argued that $\mathrm{M}(\mathrm{D}(\mathrm{R}(\mathrm{U}(Q))))$ represents a minimal query that is unique modulo isomorphism for equivalent MQs (under set or bag semantics). The subsequent application of $\mathrm{iCan}(\cdot)$ thereafter produces a canonical query with respect to isomorphism, which is thus unique for congruent MQs. Finally, let Q'_1 denote $\mathrm{R}^{-}(\mathrm{iCan}(\mathrm{M}(\mathrm{D}(\mathrm{R}(\mathrm{U}(Q_1))))))$ and let Q'_2 denote $\mathrm{R}^{-}(\mathrm{iCan}(\mathrm{M}(\mathrm{D}(\mathrm{R}(\mathrm{U}(Q_2))))))$. If $Q_1 \cong Q_2$, then $Q'_1 = Q'_2$ since the rewritten queries are unique for the set of all congruent MQs. On the other hand, Theorem 1 implies $Q'_1 \cong Q_1$ and $Q'_2 \cong Q_2$; as a result, if $Q'_1 = Q'_2$, then $Q_1 \cong Q_2$. □

E LSQ Query Features

Table 2 provides an overview of these LSQ queries, first counting queries that contain individual features, and second counting the combinations of UCQ features in queries that feature (at least) `UNION`. In terms of UCQ features, we see that `JOIN` and `DISTINCT` are used frequently, whereas `UNION` is used relatively infrequently. We also observe that `OPTIONAL` and `FILTER` are very commonly used features: though it would be undecidable to consider canonicalisation of UCQs with such additional features, we see that it is important to at least be able to offer incomplete canonicalisation of such queries.

Table 2. Number of LSQ queries with individual features (left) and combinations of UCQ features involving `UNION` (right; U=`UNION`, D=`DISTINCT`, J=`JOIN`, *=*other*)

Feature	Queries
`DISTINCT`	143,522
`JOIN`	309,087
`UNION`	34,282
Projection	665,956
`FILTER`	181,606
`OPTIONAL`	282,700
Named graph features	234,860
Solution Modifiers	5,810
Unsupported	1,046

Comb.	Queries
U	1,480
UD	1,462
UJ	1,902
UDJ	372
U*	6,127
UJ*	199
UD*	168
UDJ*	22,572

F Synthetic MQ Experiments

We consider synthetic queries for testing the performance of canonicalising MQs, including the application of UCQ normal forms and intra-CQ minimisation technique (as well as inter-CQ minimisation and canonical labelling).

More specifically, we create synthetic MQs 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 the k-clique such that $k = m + n$. Observe that such queries are not in UCQ/DNF normal form (union of joins) but rather in UCQ/CNF normal form (join of unions). Hence we will need to apply UCQ normalisation, minimisation and canonical labelling. In particular, the resulting initial UCQ will be of size n^m. Furthermore, note that by sampling from the k-clique with replacement, we may end up with redundancy in the CQs and UCQs that will be subject to minimisation.

We run these experiments for $m = 2^i$ and $n = 2^j$ for $(i, j) \in \mathbb{N}^2_{0\ldots5}$, generating five queries for each (m, n) value. Figure 4 presents the runtimes for canonicalisation of these synthetic UCQs, where each graph represents a particular m value, with the x axis presenting the various n values. From these results, we can see that we run into performance issues already for $(4, 4)$, where producing a UCQ of size $4^4 = 64$, the subsequent exponential processes – minimisation and canonicalisation – deteriorate in performance; other results for $(4, 8)$ and $(8, 4)$ were not achieved. These synthetic cases highlight the double-exponential behaviour of the canonicalisation algorithm. However, we conjecture that any canonicalisation algorithm for MQs will be double-exponential, where the Π^P_2-complete result of Sagiv and Yannakakis [20] for MQ equivalence establishes at least the need for an exponential algorithm (unless P = NP) since canonicalisation must

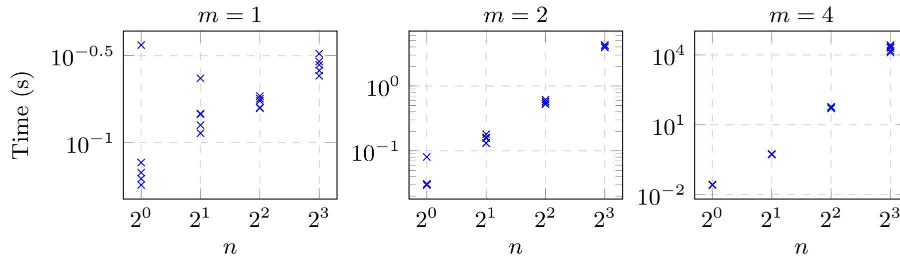


Fig. 4. Runtimes for synthetic UCQs for varying m and n

then be Π_2^P-hard; we further conjecture that an initial exponential rewriting of the query to a particular given normal form is required for canonicalisation.[5]

It is important to highlight, however, that not all queries with four joins of four unions would lead to such behaviour; the regularity of these synthetic (and very artificial) queries makes them a particularly challenging case for minimisation and canonicalisation. Again, experiments over real-world queries suggest that canonicalisation is feasible for the types of queries that users tend to ask.

[5] If we rather considered a rewriting to UCQs, the resulting "dual" rewriting would also be exponential. Only an adaptive rewriting procedure – one that depends on the characteristics of the particular query – could avoid the double-exponent.