QCan: Normalising Congruent SPARQL Queries

Jaime Salas and Aidan Hogan

IMFD Chile & Department of Computer Science, University of Chile

Abstract. We demonstrate a system to *canonicalise* (aka. *normalise*) SPARQL queries for use-cases such as caching, query log analysis, query minimisation, signing queries, etc. Our canonicalisation method deterministically rewrites a given input query to an equivalent canonical form such that the results for two queries are syntactically (string) equal if and only if they give the same results on any database, modulo variable names. The method is sound and complete for a monotone fragment of SPARQL with selection (equalities), projection, join and union under both set and bag semantics. Considering other SPARQL features (e.g., optional, filter, graph, etc.), the underlying equivalence problem becomes undecidable, where we currently rather support a best-effort canonicalisation for other SPARQL 1.0. features. We demonstrate a prototype of our canonicalisation framework, provide example rewritings, and discuss limitations, use-cases and future work.
Demo link: http://qcan.dcc.uchile.cl

1 Introduction

Though there are hundreds of SPARQL endpoints now available on the Web, many suffer from performance problems, including unavailability, variance in performance, timeouts, partial results, and so forth [1]. A lot of attention has thus been given to improving the performance of such query services, either in terms of exploring query optimisations, better indexing schemes, etc. Such works aim to improve the efficiency of executing a particular query in isolation.

However, a more recent trend in the literature is to consider in more detail the past queries that such services have received from users and to try to optimise these queries in particular. One direction in which such research has gone is to analyse SPARQL query logs to understand what kinds of queries users typically ask in terms of the features used, their underlying structure, the number of results they return, and so forth [2,11,4,3]. Such analyses help to understand the most important aspects of the SPARQL query language to optimise in practice. Another direction is to consider caching methods that reuse static results from previous queries to help process later queries [12,7]; such caching methods thus aim to reduce the overall workload of the service.

Such works are complicated by one key issue: the same "abstract query" can be expressed in myriad ways in the SPARQL query language, with possible variations in both the syntax and operators used. More formally, letting Q be a query and D be a dataset, we denote by $Q(D)$ the results of applying Q over

D. Now letting Q_1 and Q_2 be two SPARQL queries, we say that Q_1 and Q_2 are *equivalent*, denoted $Q_1 \equiv Q_2$ if and only if $Q_1(D) = Q_2(D)$ for any dataset D. This means that Q_1 and Q_2 do not differ in semantics – in terms of what results they will generate from a(ny) dataset – only in how they are expressed. Thus, for example, in a caching scenario, though we have already stored the results for Q_1, we may not hit the cache for Q_2 if we do not consider equivalence, even though the results are reusable (leaving aside changes to the underlying data).

Furthermore, the results for SPARQL queries are mappings from variables to constants and thus two queries that differ only in variable names may not be equivalent since their results may differ. Hence we arrive at another relation between queries: we say that Q_1 and Q_2 are *congruent*, denoted $Q_1 \cong Q_2$, if and only if there exists a one-to-one rewriting of variables $\nu : V \to V$ in Q_1 such that $\nu(Q_1)(D) \equiv Q_2(D)$; in other words, Q_1 and Q_2 are equivalent under some one-to-one variable rewriting. This notion of query congruence would allow us to capture further useful queries in, for example, a caching scenario.

Example 1. Consider two queries Q_A and Q_B asking for grandparents:

```
SELECT DISTINCT ?y WHERE {
 { ?w :mother ?x . } UNION { ?w :father ?x. }
 { ?x :mother ?y . } UNION { ?x :father ?y. }
 ?a ?b ?c. ?c ?d ?y . # redundant
}
```

```
SELECT DISTINCT ?c WHERE {
 { ?a :mother ?b . ?b :mother ?c }
 UNION  { ?a :mother ?b . ?b :father ?c }
 UNION  { ?a :father ?b . ?b :mother ?c }
 UNION  { ?a :father ?b . ?b :father ?c } }
```

Both queries are congruent: they return the same results for any RDF dataset (modulo variable names). Note that the line marked redundant in Q_A could be removed without changing the semantics of the query. □

While detecting cases of query equivalence/congruence could be helpful for various use-cases, such a procedure entails a very high computational complexity. More specifically, deciding if two queries Q_1 and Q_2 are equivalent, in the case of SPARQL, 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 added [9]. Furthermore, consider a log or stream of n queries issued to an endpoint. Traditional methods for deciding query equivalence take two queries and return `true` if they are equivalent, or `false` otherwise. Using such a decision procedure, we would be forced to perform $O(n^2)$ comparisons to find all equivalent queries. This would clearly become prohibitively costly for endpoints that have to deal with millions of queries [11], *even if* checking individual query pairs does not observe the exponential worst-cases constructed in theory. Hence we see some of the challenges associated with detecting equivalent/congruent queries in settings such as caching.

2 QCan: Query Canonicalisation

While the previous discussion paints a pessimistic view of the possibility of ever having a procedure that can detect equivalent/congruent queries in a practical setting such as caching, we highlight two observations that may give us hope: (1)

many real-world queries are quite simple, where the sorts of worst-cases predicated in theory seem unlikely to occur in current practice [4]; (2) we can avoid quadratic pairwise checks by using a canonicalisation procedure that facilitates indexing equivalent queries by their normalised string or hash value.

Along these lines, in our paper accepted for the ISWC Research Track [10], we proposed a canonicalisation procedure for *monotone SPARQL queries* – queries with selection, projection, join and union under set or bag semantics – such that the canonical form of a query $\text{can}(Q)$ is congruent to Q, and the canonical form of two queries Q_1 and Q_2 are equal ($\text{can}(Q_1) = \text{can}(Q_2)$) if and only if Q_1 and Q_2 are congruent. We proved this procedure to be sound and complete for the aforementioned fragment of monotone SPARQL queries and through experiments on real and synthetic queries, we showed it (1) to be efficient for queries found in practice, taking milliseconds on average; (2) to be capable of finding more duplicate queries than a baseline syntactic normalisation; (3) to eventually break on harder synthetic cases that provoke a doubly-exponential behaviour. Rather than reject queries using other features of SPARQL (1.0), we discussed how the procedure can be extended to perform a best-effort (sound but incomplete) canonicalisation of other features of SPARQL.

The canonicalisation procedure is founded on representing the algebra of a SPARQL query in a simpler structure that we can then normalise (following techniques known from theoretic works for deciding query equivalence). Specifically, we represent a SPARQL query as an RDF graph and then apply normalisation steps on the RDF graph;[1] in this graph, we represent query variables as blank nodes. Over the RDF graph representation, our first step is to apply a Union of Conjunctive Query (UCQ) normalisation: in SPARQL one can express unions of joins, joins of unions, or any nesting between; hence we first apply a (potentially exponential) rewriting of the query such that it is strictly a union of multiple Conjunctive Queries (CQs; aka. BGP), where each CQ is a join of multiple triple patterns; this UCQ rewriting will effectively rewrite Q_A in Example 1 to a query of a similar form to Q_B. Next, under set semantics, as shown for Q_A in Example 1, a query may contain redundant patterns, where we apply a *minimisation* step: in each CQ of the UCQ computed in the previous stage, we remove redundant variables (this is done by *leaning* the RDF sub-graph corresponding to the CQ), where we then remove redundant CQs from the UCQ. Finally, we apply a canonical labelling (modulo isomorphism) of the blank nodes, which deterministically labels the query variables [6]. The last step is to apply a deterministic syntactic mapping from the RDF graph back to a SPARQL query.

For space reasons, we have sketched our canonicalisation procedure. For further details – including related work, detailed examples, formalisation, proofs and experiments – we refer to the extended version of the full paper [10]. We highlight, however, that while there exist frameworks to decide query equivalence, we are not aware of any framework that *canonicalises* queries in this

[1] This is not dissimilar to the idea of SPIN [5], though we require special attention to ensure correspondences between RDF [6] and SPARQL equivalence relations.

manner (for SPARQL or even for SQL); the closest work we are aware of is that of Papailiou et al. [7], who rewrite query variables modulo isomorphism.

3 QCan Demo

In the Poster and Demo track, we plan to provide a demo of our QCan system, publicly available here: `http://qcan.dcc.uchile.cl`. The demo is quite straightforward: given an input query on the left, it computes and displays the normalised canonical version of the query on the right. Our main motivation is to be able to give practical hands-on examples of how the canonicalisation process works to those who may have use for it, and to highlight the current types of queries supported, what are the current limitations (limited support for filters, no support for SPARQL 1.1), etc. We would also be keen to discuss use-cases for this system and what requirements they might imply.

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

References

1. 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)
2. Arias Gallego, M., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. In: USEWOD (2011)
3. Bielefeldt, A., Gonsior, J., Krötzsch, M.: Practical Linked Data Access via SPARQL: The Case of Wikidata. In: LDOW (2018)
4. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. PVLDB 11(2), 149–161 (2017)
5. Fürber, C., Hepp, M.: Using SPARQL and SPIN for data quality management on the semantic web. In: Business Information Systems (BIS). pp. 35–46 (2010)
6. Hogan, A.: Canonical forms for isomorphic and equivalent RDF graphs: Algorithms for leaning and labelling blank nodes. ACM TOW 11(4) (2017)
7. 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)
8. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34(3) (2009)
9. Pichler, R., Skritek, S.: Containment and equivalence of well-designed SPARQL. In: Principles of Database Systems (PODS). pp. 39–50 (2014)
10. Salas, J., Hogan, A.: Canonicalisation of Monotone SPARQL Queries. ISWC (Accepted) (2018), see extended version: `http://aidanhogan.com/qcan/extended.pdf` (under revision for the camera-ready version)
11. 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)
12. Williams, G.T., Weaver, J.: Enabling fine-grained HTTP caching of SPARQL query results. In: International Semantic Web Conference (ISWC). pp. 762–777 (2011)