Canonicalisation of SPARQL 1.1 Queries

Jaime Salas
supervised by Aidan Hogan
Universidad de Chile
Santiago, Chile

ABSTRACT

SPARQL is the standard query language for RDF as stated by the W3C. It is a highly expressive querying language that contains the standard operations based on set algebra, as well as navigational operations found in graph querying languages. Because of this, there are various ways to represent the same query, which may lead to redundancy in applications of the Semantic Web such as caching systems. We propose a canonicalisation method as a solution, such that all (monotone) congruent queries will have the same canonical form. Despite the theoretical complexity of this problem, our experiments show a good performance over real-world queries. Finally, we anticipate applications in caching, log analysis, query optimisation, etc.

CCS CONCEPTS

• **Theory of computation → Database query processing and optimization (theory)**; Complexity classes; Problems, reductions and completeness; *Data structures and algorithms for data management*; Regular languages; • **Information systems → Query languages**.

KEYWORDS

sparql, canonicalisation, normalisation, minimisation, query, monotone, canonical

ACM Reference Format:
Jaime Salas, supervised by Aidan Hogan. 2022. Canonicalisation of SPARQL 1.1 Queries. In *Companion Proceedings of the Web Conference 2022 (WWW '22 Companion), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3487553.3524191

1 INTRODUCTION

The Semantic Web is an evolution of the World Wide Web that surfaced as a complement to the standard Web in order to allow automated processes over data found in it. This has been achieved by introducing standards, frameworks and technologies to structure data in a way machines are able to interpret.

The *Resource Description Framework* (RDF hereafter) was introduced as the standard representation of resources on the Semantic Web. It is a semi-structured representation of data that allows to model data without the constraints of a schema and with the easy integration of external sources.

SPARQL was then introduced as the standard querying language for RDF. It features the standard operations of querying languages based on set algebra such as joins, unions, projection, etc. In addition, it allowed for the querying of different datasets simultaneously. Despite this, SPARQL was missing useful features such as (explicit) negation, the aggregation of results, and results from federated queries. Therefore, SPARQL 1.1 was introduced as the new standard, which included these features. In addition, because of the graph-like structure of RDF datasets, property paths were introduced. These allowed for the querying of relations between entities of arbitrary length, as well as to express queries more succinctly.

Unfortunately, SPARQL may exhibit performance issues in some dedicated endpoints, where these services may be unavailable for a considerable amount of time [2]. Fortunately, some of these issues are partially addressed by caching systems, but are limited by the system's ability to detect duplicate queries [14].

Our work proposes a method for the canonicalisation of SPARQL 1.1 queries. This involves the computation of a canonical form for each query. This canonical form is identical for all queries that are *congruent*, that is, queries that return the same results given any RDF dataset, given an equivalent variable mapping. We note however that such a form cannot exist for the full language since it could otherwise be used to solve the first-order logic validity problem [22]. On the other hand, this problem is decidable for more limited fragments of SPARQL.

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

```
SELECT DISTINCT ?name WHERE {
  { ?aunt :name ?name .
    ?child :mother ?parent .
    ?parent :sister ?aunt . }
  UNION
  { ?aunt :name ?name .
    ?child :father ?parent .
    ?parent :sister ?aunt . }
}
```

Figure 1: Congruent queries

Figure 1 shows two congruent queries. Both queries look for the names of aunts, but the query on the right is in its *conjunctive normal form*. Additionally, they are congruent because if we rewrite the variables on the left by mapping: (`?a` → `?aunt`, `?c` → `?child`, `?p` → `?parent`, `?z` → `?name`), then they are *equivalent*.

Our goal is to formulate a function that, given an input query, returns a canonical query that is congruent to the input, and (ideally) gives the same output query for all queries congruent to the input.

We anticipate the following applications:

WWW '22 Companion, April 25–29, 2022, Virtual Event, Lyon, France

ACM ISBN 978-1-4503-9130-6/22/04…$15.00
https://doi.org/10.1145/3487553.3524191

Query caching: Papailiou et al. canonically labeled basic graph patterns in their study for caching. Our method includes a larger scope [14].

Log analysis: Large query logs are analysed to find statistics on uses of features, trends, etc, which may be used to optimise existing SPARQL endpoints. Our method may reduce the workload by computing sets of congruent queries. [1, 18]

Query optimisation: Our method may be used to process queries before execution to remove redundancies, and thereby improve the performance of systems, particularly if a query must be executed multiple times. [10]

Learning over queries: We may reduce the number of distinct queries in training sets for machine learning models by finding all unique queries and giving them a standard syntactic form, reducing the variance in the training set [3].

2 PROBLEM

Our hypothesis is that a canonical form for SPARQL 1.1 can be computed by an algorithm in a reasonable time despite the theoretical complexity of the problem. Additionally, this algorithm would be *sound* for the full language, and *complete* for a monotonic fragment of the language, which represents a large majority of real-world queries.

The algorithm is *sound* if the canonical query is congruent to the original query; the algorithm is *complete* if all congruent queries share the same canonical form. It is evident that a sound and complete algorithm for the full language that always halts may not exist, since otherwise it could be used to solve the equivalence of queries that include undecidable cases.

3 RELATED WORK

3.1 Minimisation and Normalisation

Pérez et al. [15] proved in their study that certain fragments of SPARQL 1.0 allowed a normal form. This fragment corresponds to monotone queries, which are queries that contain only combinations of joins, unions and projections. Furthermore, Schmidt et al. present in their study [19] a number of rewriting rules over features such as left joins, filters and negations, as well as an erratum on one of the results of Pérez et al., which suggested a normal form for queries containing optional patterns.

Studies have shown that the containment decision problem is undecidable for queries that contain optional patterns (left joins) under set semantics, let alone the full language [7]. Equivalence is also undecidable for queries based on the relational algebra. [23].

3.2 Canonicalisation

Hogan designed a system for the canonical labeling and leaning of RDF graphs (blabel) [8]. The canonical labeling is based on computing successive partitions over blank nodes in a deterministic manner, inspired by a NAUTY algorithm for checking graph isomorphism [13].

In regards to the canonicalisation, a few studies have mentioned the canonical labeling of variables in applications such as sub-query caching. Papailou et al. [14] utilised a canonical labeling algorithm (specifically Bliss [9]) on basic graph patterns in order to either cache results or find stored results from isomorphic queries.

3.3 Existing Systems

Because of the theoretical results that indicate that query containment, equivalence and minimisation are complex problems in the worst-case, few systems have been implemented to tackle these problems. However, by limiting the use of certain features, smaller fragments of the language have been identified where these problems are decidable, and even efficient in the general case.

For SPARQL, a number of systems have been implemented to solve the containment problem on queries under certain constraints. These are `SPARQL Algebra` [11] and the `Jena-SPARQL-API Graph -isomorphism` (`JSAG`) tool [20]. In particular, the approach in the study by Stadler et al. [20] consists in the representation of certain features of SPARQL as a directed graph in order to normalise expressions. However, it is limited to mostly monotonic features and optional patterns, whereas SPARQL 1.1 contains several other features that are standard to other known query languages. These features include: aggregations, regular path queries, negation, etc. Chekol et al. proposed a system [5] where queries are translated into formulae which are then processed by either `Alternation Free two-way MU-calculus` (`AFMU`) [21] or TreeSolver [6], an XPath-equivalence checker. Their study also compared the previously mentioned systems to `SPARQL Algebra`, and concluded that `SPARQL Algebra` often outperformed the other systems. However, it is worth noting that `SPARQL Algebra` works directly on SPARQL queries, whereas the other systems depend on a translation to formulae.

To our knowledge, little work has been done specifically on the application of minimisation and normalisation on large collections of real-world queries.

4 PROPOSED APPROACH

The objective of this work is to expand the scope of existing systems for the canonicalisation of SPARQL. This includes the addition of the newer features that were introduced as part of SPARQL 1.1. Additionally, it is desirable to include support for the normalisation of regular path queries, a challenging, but theoretically feasible process. This result may be used to form a partition of query logs into congruent equivalence classes. Finally, it is necessary to identify use-cases for the system and adapt the system to satisfy them.

Our method can be summarised in the following steps:

Query Rewriting We apply several rewriting rules in order to compute a more normal form of the query. This includes rewriting monotone parts of the query into unions of basic graph patterns, property paths into equivalent graph patterns, and renaming or eliminating variables that do not affect the results. This may be repeated iteratively until no more changes are made.

Graph Representation We construct a graph based on an algebraic expression of the query. This allows us to capture both the commutative and associative properties of certain SPARQL operators, and is necessary to use the tools needed in the following steps.

Removal of Redundancies We remove redundancies in the monotone parts of the query. These are divided into two groups: redundant basic graph patterns, and redundant triple patterns. We perform containment checks on every pair of basic graph patterns in a union, and eliminate those that are

contained in others. In addition, we compute the core of the graph that represents each basic graph pattern in order to eliminate redundant triple patterns.

Canonical Labeling We find canonical labels for the variables in the query by canonically labeling the graph that represents the query. The blank nodes are assigned canonical labels based on the topology of the graph. Thereby we assign new names for each variable based on their respective blank nodes.

5 METHODOLOGY

Literature Review: An extensive research of relevant studies is performed. This includes the studying of normalisation techniques in both abstract structures such as algebraic expressions and data structures such as graph datasets. In addition, we must research similar systems or systems that may aid in solving the proposed problem.

Design of Solution: Based on the findings of the first step, a method and a system are designed to solve the gap in the state of the art. This includes the design of data structures optimised for our application, adapting existing methods to our purposes, etc.

Implementation of Method: Development of a system that utilises the methods designed in the previous step in order to test the hypothesis. This step is repeated following feedback from the evaluation stage in order to optimise the system and address any shortcomings.

Evaluation of the System: Acquire data of the performance of the system based on execution time, number of additional duplicated queries detected, scalability, and comparisons with similar existing systems. The results from this stage may reveal issues and potential improvements that lead to a new stage of implementation.

Identification of Use-cases: In this stage, we must identify applications whose performance would be improved by the proposed method.

Integration with Other Systems: Once we have found systems that can be improved with our method, a new stage of development starts to integrate our method with the target system. This may include adapting the method to work with different structures than those envisioned.

Evaluation of the Combined System: We will need to design benchmarks and tests to evaluate the performance of the system. This step may also need feedback from real users.

6 RESULTS

In this section we describe the most prominent results of our research so far. We begin by giving an overview of the canonicalisation algorithm we have implemented. Next, we present the results of our experiments over real-world queries in order to evaluate the performance of our algorithm. Finally, we present the results of our experiments comparing the performance of our system to other systems in detecting duplicated queries in a set of SPARQL queries. A thorough explanation of this work can be found in our study [17].

6.1 Query Rewriting

By using the abstract algebra provided by Apache Jena [12], as well as several tools for the transformation of these structures, we rewrite the monotonic fragments of queries into unions of conjunctive queries (UCQs), property paths into equivalent query patterns, when possible, and local variables that are always unbound are removed. In addition, under certain circumstances, we may rewrite queries that cannot produce duplicated bindings by adding the DISTINCT keyword if it is missing. See Figure 1 for an example, where the query on the left would be rewritten into a UCQ similar to the query on the right.

6.2 Graph Representation

We utilise the data structures provided by Jena to represent the normalised abstract algebra expression of the previous step as an RDF graph called an *r-graph*.

Terms such as literals with datatypes (e.g. booleans) are represented by *literal nodes* (which we assume are in canonical form), IRIs are represented by *IRI nodes*, and blank nodes and variables are represented by *blank nodes* provided by Jena. Nodes that represent variables are labeled accordingly, whereas we assume that nodes that represent blank nodes are uniquely identified by their label.

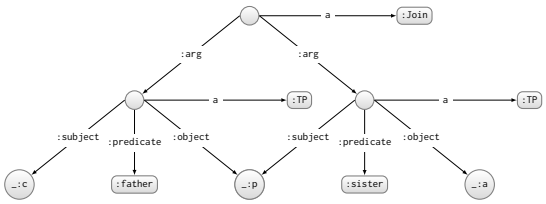

Figure 2: Representational graph of a basic graph pattern

Figure 2 shows an example of the representation graph for a basic graph pattern that searches for paternal aunts. Blank nodes are shown as circles, where they are labeled only if they represent a variable. IRIs are represented by rounded rectangles. Triple patterns such as (`_:c`, `:father`, `_:p`) are reified, meaning we represent their structure by adding a triple pattern for each component of the original triple pattern. Nodes that represent operators such as JOIN and triple patterns (TP) are typed with the `a` predicate and an appropriate IRI. Since both triple patterns in this basic graph pattern refer to the parent by a variable p, they both point to the same node representing this variable. Other operators may be represented in a similar way. This representation offers two benefits: we abstract the order of operands for operators such as JOIN and UNION, which are commutative, and we can use techniques for computing cores and canonically labeling graphs, as described in the following.

6.3 Redundancy Elimination

We apply techniques to remove redundancies within monotone parts in queries. The first type of redundancy we consider is the presence of redundant triple patterns. As we mentioned in Section 6.2, as part of our method, we represent basic graph patterns as RDF graphs where triple patterns are represented as triples with existential blank nodes instead of variables. Therefore, if a basic graph

pattern contains redundant triple patterns, its graph representation will contain redundant triples.

The second type of redundancy we consider are redundant basic graph patterns in unions. Since at this step all of the monotone components of the query have been normalised into UCQs, we simply find all union operators and list all the basic graph patterns contained within. Following this, we perform a standard UCQ minimisation [4] by checking for containment of all pairs of basic graph patterns. Finally, we eliminate all basic graph patterns that are found to be contained in another basic graph pattern.

6.4 Canonical Labeling

The last step of the algorithm consists in a canonical labeling. In order to do this, we use the *blabel* package for the canonical labeling of blank nodes in RDF graphs. This package will produce a representational graph whose blank nodes have canonical labels. We use these canonical labels to deterministically name variables.

6.5 Experiments

In order to test our algorithm in a real-world setting, we used queries from the Wikidata logs [24] available at https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en, and the LSQ dataset [18] available at https://aksw.github.io/LSQ/, which itself contains queries from DBpedia3.5.1, Linked Geo Data, Semantic Web Dog Food, and British Museum. The full results are available in our study [16].

As part of our baseline method, we translate the SPARQL queries into algebraic expressions using Apache Jena's ARQ engine. This representation effectively removes redundant elements such as whitespaces and capitalisation (e.g. queries that say `filter` in lowercase versus queries that say `FILTER` in uppercase).

6.5.1 Real World Queries. We now present the results of performing the full canonicalisation method on all the queries in our test group. The results in Figure 3 indicate that the canonical labeling of the r-graph is the step that takes the most time on average. In fact, in most cases the total time spent on the algorithm is dominated by this step. This is an expected result since the canonical labeling scales poorly with the size of the graph, and both the `SWDF` and `Wikidata` sets contain the queries with the largest r-graphs. However, it is noteworthy that the minimisation step presents the largest variation of all the steps, as can be seen in the results for the `SWDF` and `Wikidata` sets. In fact, in some occasions the runtime is determined by the minimisation time rather than the labeling time.

6.5.2 Duplicates found. For this experiment, we started with our baseline method, and in each successive iteration we added another step of the full algorithm, until we reached the full algorithm. Table 1 denotes the total number of duplicate queries found. Here, Raw indicates comparison of the query string, and Baseline indicates a method where queries are translated into algebraic expressions using Apache Jena ARQ's engine, which effectively normalises syntactic elements such as whitespaces and capitalisation (e.g. queries that say select in lowercase versus SELECT in uppercase).

The results shown in Tables 1 show that the number of duplicates found either remains the same or increases in each successive algorithm. In particular, the highest increase occurs between the

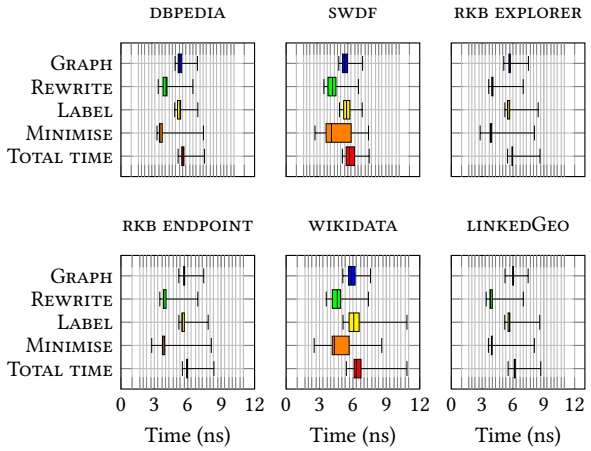

Figure 3: Runtimes for each step of the canonicalisation algorithm over the following sets of queries: DBpedia (upper left), SWDF (upper right), RKB Explorer (middle left), RKB Endpoint (middle right), Wikidata (lower left) and Linked Geo (lower right). Times are shown in logscale (10^x)

Table 1: Total number of duplicates found by each method.

Method	DBp	SWDF	REx	REn	WD	Geo
Raw	250,940	53,061	299,892	142,032	683,132	723,116
Baseline	251,283	53,263	301,419	143,523	686,453	736,331
Label	251,315	53,388	301,910	144,007	687,654	739,695
Rewrite	251,315	53,388	301,910	144,007	687,751	739,700
Full	251,315	53,388	301,911	144,007	687,760	739,702
Queries	424,362	112,470	335,833	169,617	872,555	842,794

Baseline and the Label methods for the `Wikidata` set of queries, where the number of duplicates detected increases by a few thousand. In the other query sets this increase is less. In addition, there is almost no difference in the number of duplicates found by the Label and Rewrite methods, with only a single one found in the `RKB Endpoint` query set. Furthermore, the number of duplicates found by the full algorithm is the same as the one found by the Rewrite method. However, this is an expected result since most of the queries used in these experiments are quite simple, and are therefore unlikely to contain any redundancies.

6.5.3 Comparison with Existing Systems. To our knowledge, there are currently no other systems that compute normal, canonical SPARQL queries. However, we may compare the performance of our algorithm with specialised systems based on *query containment*, namely `SPARQL Algebra`[11] and the `Jena-SPARQL-API Graph-isomorphism (JSAG)` tool [20], based on how many duplicate queries are detected in a set of queries.

Figure 4 shows the runtimes for our comparison of both containment checkers and our method. We note that there are two sets of queries: one that contains conjunctive queries without projection, and the other contains unions of conjunctive queries with projection. `SA1`, `JSAG1` and `QCan1` correspond to the tests on the set of conjunctive queries without projection, whereas `SA2`, `JSAG2` and

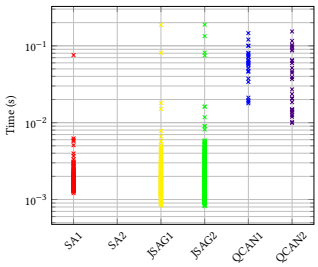

Figure 4: Runtimes for JSAG, SA and QCan

`QCan2` correspond to the tests on the set of unions of conjunctive queries with projection. However, there are no values for `SA2` because `SPARQL-Algebra` does not support queries with projection. The results indicate that both `SA` and `JSAG` take on average between a thousandth and a hundredth of a second, whereas our method takes under a tenth of a second. Therefore, our method is considerably slower than both containment checkers if we want to check for the equivalence of two queries. However, it should be noted that the number of containment checks that were needed to find all equivalent queries is quadratic on the number of queries, whereas our method is linear on the number of queries.

7 CONCLUSION

As of this time, the work on the canonicalisation of the core of SPARQL 1.1 is done, and our method has proven to be efficient in a real-world setting. The method is complete for a significant portion of SPARQL, that of monotone queries, and is sound for the complete language. Indeed, the canonical labeling already proves to significantly increase the number of duplicated queries detected, more so than the rewriting and minimisation of monotone sub-queries. As future work, we expect to resume work on the normalisation of property paths, more specifically a fragment known as *conjunctive two-way regular path queries* (C2RPQs). The decision problems of containment and equivalence of this fragment are decidable, though intractable. Our current approach simply normalises the paths in each separate path pattern, whereas our aim is to provide a form of normalisation analogous to that of UCQs. We have identified the *star* ($*$) operator (and the *plus* (+)) as the main problem because there are no simple ways to verify if one of these paths is redundant. This holds true even when limiting the paths to be of the form p^*, where p is an IRI. However, we anticipate that most C2RPQs in real settings will be able to be minimised efficiently. Regardless, even if it is not feasible, to our knowledge there are no studies that confirm this, nor are there studies that identify the largest fragment of monotone queries with property paths for which this is still feasible. Finally, and in parallel with the former work, we will integrate our system with a caching system in order to improve the cache hit rate. Currently, experiments that use our method yield positive results in only about half of the test cases. It is not known if this is due to the performance of the caching system itself or the overhead caused by our method. We are also interested in exploring other possible use-cases, as mentioned in the introduction, involving query optimisation, question answering, and more besides.

REFERENCES

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

[2] C. Buil Aranda, M. Arenas, Ó. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Semant.*, 18(1):1–17, 2013. doi: 10.1016/j.websem.2012.10.001.

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

[4] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. SPARQL query containment under SHI axioms. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 2012.

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

[6] P. Genevès, N. Layaïda, and V. Knyttl. Xml reasoning solver user manual. 2011.

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

[8] A. Hogan. Skolemising Blank Nodes while Preserving Isomorphism. In *World Wide Web Conference (WWW)*, pages 430–440. ACM, 2015. doi: 10.1145/2736277.2741653.

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

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

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

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

[13] B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014. doi: 10.1016/j.jsc.2013.09.003.

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

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

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

[17] J. Salas and A. Hogan. Semantics and Canonicalisation of SPARQL 1.1. *Semantic Web Journal*, 2022. doi: 10.3233/SW-212871. (To appear).

[18] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo. LSQ: the linked SPARQL queries dataset. In *International Semantic Web Conference (ISWC)*, volume 9367, pages 261–269. Springer, 2015. doi: 10.1007/978-3-319-25010-6_15. URL https://doi.org/10.1007/978-3-319-25010-6_15.

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

[20] C. Stadler, M. Saleem, A. N. Ngomo, and J. Lehmann. Efficiently Pinpointing SPARQL Query Containments. In *International Conference Web Engineering (ICWE)*, volume 10845 of *Lecture Notes in Computer Science*, pages 210–224. Springer, 2018. doi: 10.1007/978-3-319-91662-0_16.

[21] Y. Tanabe, K. Takahashi, M. Yamamoto, A. Tozawa, and M. Hagiya. A decision procedure for the alternation-free two-way modal μ-calculus. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 277–291. Springer, 2005.

[22] B. Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. In *Proceedings of the USSR Academy of Sciences*, volume 70, pages 569–572, 1950.

[23] B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70(4):569–572, 1950.

[24] D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.