UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CANONICALISATION OF SPARQL 1.1 QUERIES

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN COMPUTACIÓN

JAIME OSVALDO SALAS TREJO

PROFESOR GUÍA:
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
ANGELA BONIFATI
DOMAGOJ VRGOC
JUAN REUTTER DE LA MAZA

SANTIAGO DE CHILE
2023

Resumen

Canonicalización de consultas SPARQL 1.1

SPARQL es el lenguaje de consulta estándar para RDF de acuerdo al *World Wide Web Consortium* (W3C). Es un lenguaje de consulta altamente expresivo que contiene las operaciones estándares del álgebra relacional tales como los *joins*, la unión, los *outer joins*, etc. además de operaciones propias de los lenguajes de consulta para grafos. Debido a esto, existen varias maneras de representar la misma consulta que no son triviales de determinar, lo cual puede causar ineficiencias en aplicaciones de la Web Semántica. Por ejemplo, un sistema de caché podría funcionar más eficientemente si fuese capaz de detectar estas consultas equivalentes. Se propone una técnica de *canonicalización* como una solución a este problema, de manera que se computa una forma canónica para las consultas SPARQL donde todas las consultas son equivalentes módulo nombres de variables (o congruentes) a su forma canónica, y para un subconjunto del lenguaje, todas las consultas congruentes tienen la misma forma canónica. Se describen en detalle los pasos que componen este método de canonicalización: la normalización algebraica, la representación de la consulta como un grafo RDF, la minimización de las componentes monótonas de la consulta, y la rotulación canónica de las variables. A pesar de la alta complejidad teórica del problema, los experimentos demuestran el buen comportamiento del método en consultas enviadas por usuarios reales. Finalmente, se discuten posibles casos de uso en sistemas de caché, análisis de consultas, *query benchmarking*, etc.

Abstract

SPARQL is the standard query language for RDF as recommended by the World Wide Web Consortium (W3C). It is a highly expressive query language that contains the standard operations based on set algebra such as joins, unions, outer joins, etc. 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 redundancies in applications of the Semantic Web such as caching systems, where cached results could be returned if the systems were capable of detecting these duplicates. We propose a canonicalisation technique as a solution, where we compute a canonical form for SPARQL queries such that all queries are equivalent modulo variable names (congruent) to their canonical form, and where for a subset of the language, all congruent queries will have the same canonical form. We describe in full detail the steps that comprise our canonicalisation method: the algebraic rewriting, the representation of the query as an RDF graph, the minimisation of monotone sub-queries, and the canonical labelling. Despite the theoretical complexity of this problem, our experiments show good performance over real-world queries. Finally, we anticipate applications in caching, log analysis, query benchmarking, etc.

Table of Content

1 Introduction **1**

1.1 Problem 2

1.2 Hypothesis 5

1.2.1 Goals 6

1.3 Thesis Structure 7

2 Background **9**

2.1 Graph Theory 9

2.2 RDF 10

2.2.1 Semantics of RDF 13

2.2.2 Isomorphism in RDF 14

2.2.3 Entailment and Equivalence in RDF 14

2.2.4 Canonicalisation of RDF graphs 16

2.3 SPARQL 19

2.3.1 Abstract Syntax 19

2.3.2 Solution mappings 19

2.3.3 Algebra 20

2.3.4 Semantics 20

2.3.5 Set or Bag Semantics 21

2.3.6 Classes of Queries 21

2.3.7 Safe Variables 21

2.3.8 Other features 22

3 Related Work **24**

3.1 Rules for Minimisation and Normalisation 24

3.2 Systems for RDF and SPARQL Canonicalisation 24

3.3 Systems for SPARQL Containment 25

3.4 Other Query Languages 25

4 SPARQL 1.1 **27**

4.1 Syntax 27

4.2 Semantics 28

4.2.1 Set Semantics 28

4.2.2 Bag Semantics 32

4.3 Default and Named Graphs 33

4.4 Assignments to Variables 34

4.5 Aggregation 35

4.6 Negation 37

4.7 Federated Queries 38

4.8 Query Containment and Equivalence 39

5 Canonicalisation of SPARQL queries **41**

5.1 Monotone Queries 41

5.1.1 Normalisation 42

5.1.2 Graph Representation 53

5.1.3 Redundancy-free Monotone Queries 56

5.1.4 Canonical Labelling 63

5.1.5 Inverse Mapping 63

5.2 SPARQL 1.1 65

5.2.1 Normalisation 65

5.2.2 Graph Representation 67
5.3 Property Path Normalisation 75

6 Soundness and Completeness **78**

6.1 Monotone Queries 78
6.1.1 Soundness 78
6.1.2 Completeness 80
6.1.3 Complexity 85
6.2 SPARQL 1.1 87
6.2.1 Soundness 87
6.2.2 Incompleteness 88
6.2.3 Complexity 91

7 Towards UC2RPQs **92**

7.1 Normalisation 92
7.1.1 Expansion 93
7.1.2 Collapse 93
7.1.3 Collapse then Expand, or Expand then Collapse 94
7.2 Minimisation 96
7.2.1 Compute the Transitive Closure 96
7.2.2 RPQ-aware Minimisation 98
7.2.3 2RPQ Containment and Equivalence 101
7.2.4 Union Expansion 106
7.3 Discussion 107

8 Implementation **108**

8.1 Dependencies 108
8.2 Project Structure 109
8.2.1 Builder 111

8.2.2 Data 111
8.2.3 Generate 112
8.2.4 Main 112
8.2.5 Parsers 113
8.2.6 Paths 113
8.2.7 Transformer 113
8.2.8 Visitor 114
8.3 Canonical Labelling 114
8.4 Serialisation of Representational Graphs 115
8.5 Property Paths and Automata 115
8.6 Testing 115

9 Results and Evaluation **117**
9.1 Queries used 117
9.1.1 Real-world Queries 117
9.1.2 Synthetic Queries 119
9.2 Machine Specifications 121
9.3 Results 122
9.3.1 Real-world Queries 122
9.3.2 Synthetic Queries 127
9.4 Comparison with Other Systems 129

10 Use-cases **132**
10.1 Query Caching 132
10.2 Automated Queries 133
10.3 Benchmarking Queries 133
10.4 Query Taxonomy 134

11 Conclusion **138**

11.1 Summary 138
11.2 Key Results 139
11.3 Discussion 140
11.4 Future Work 141

Bibliography **147**

List of Figures

1.1 An example of two equivalent SPARQL queries 2

1.2 An example of two congruent SPARQL queries. 3

1.3 An example of two equivalent queries with property paths. 4

2.1 An example of two isomorphic graphs. 9

2.2 An example of an RDF graph about Pokemon Yellow. 12

2.3 Ground graph example 12

2.4 Example of a reification of a triple. Here `_:t` is used to denote the triple. . . 13

2.5 An example of three RDF graphs that are semantically equivalent. 15

2.6 Example of labelling process 17

2.7 Example of labelling process (continued) 18

2.8 Example of a query that contains an `ORDER BY` clause. 22

2.9 Example of a query that contains `LIMIT` and `OFFSET` 23

4.1 An example of a query that contains a `FROM` clause. 33

4.2 An example of a query that contains `FROM NAMED` and `GRAPH` clauses. 34

4.3 Example of a query with a `BIND` expression. 34

4.4 Example of a query with inline values. 35

4.5 Example of a SPARQL query with aggregation. 36

4.6 Example of queries featuring negation using `MINUS` (left) and `FILTER NOT EXISTS` (right). 37

4.7 An example of a SPARQL query that includes a federated query. 38

5.1 An example of two equivalent UCQs. 45

5.2 An example of a query that projects an unbound projected variable. 47

5.3 An example of an unsatisfiable BGP. 48

5.4 An example of two equivalent UCQs. Note that the UCQ on the left contains an unsatisfiable CQ. 50

5.5 An example of two equivalent BGPs. 50

5.6 Illustration of SPARQL terms: blank nodes and variables (left), IRIs (center), and literals (right) 53

5.7 Representation of a Triple Pattern 54

5.8 Visualisation of the graph representation of a `JOIN` expression. 54

5.9 Visualisation of the graph representation of a `UNION` expression. 55

5.10 Visualisation of the distributive Property of `JOIN` and `UNION`. 55

5.11 Visualisation of the graph representation of an explicit projection $P = \text{SELECT}_V(Q)$. 56

5.12 Example of an r-graph for a UCQ 57

5.13 Example of the r-graph for a CQ. 58

5.14 Example of a CQ with distinguished labels. 59

5.15 Example of a lean CQ. 60

5.16 Example of two equivalent queries, if we remove their redundancies. 60

5.17 On the other hand, these two queries are *not* equivalent 60

5.18 Example of an `ASK` query given by G_2 in the running example. 63

5.19 Example of G_1 from the running example with canonical labels. 63

5.20 Resulting r-graph 64

5.21 This shows the canonical form of the query presented in Figure 5.1. 64

5.22 An example of two equivalent `FILTER` expressions 66

5.23 An example of two equivalent queries. 66

5.24 An example of a SPARQL query with a monotone sub-query before and after removing a redundant triple pattern. 67

5.25 Visualisation of the graph representation of an `OPTIONAL` expression. 69

5.26 Visualisation of the graph representation of an ordered built-in function. . . 70

5.27 Visualisation of the graph representation of a `FILTER` expression. 70
5.28 Visualisation of the graph representation of an `(NOT) EXISTS` function. . . . 71
5.29 Visualisation of the graph representation of a `BIND` expression. 71
5.30 Visualisation of the graph representation of a graph pattern that contains a `FILTER` expression, a `BIND` expression, and a `GROUP BY` expression. 72
5.31 Representation of an `ORDER BY` clause. 72
5.32 Representation of an `OFFSET` and `LIMIT` clause. 73
5.33 Representation of a `FROM` clause. 74
5.34 Representation of a `GRAPH` expression. 74
5.35 Visualisation of the graph representation of a `VALUES` expression. 75
5.36 NFA, DFA and minimal DFA produced for the RPQ : $\mathtt{p}^*/{:}\,\mathtt{p}^*/{:}\,\mathtt{p}^*$ 77
7.1 Example of a path pattern. 92
7.2 The result the expansion of the pattern in Figure 7.1. 93
7.3 The result of the collapse of the pattern in Figure 7.1. 93
7.4 Example where expanding paths leads to more redundancies found. 94
7.5 Example where collapsing paths leads to more redundancies found. 95
7.6 One ring to rule them all. 95
7.7 The dreaded pizza 96
7.8 A problem with inverses. 97
7.9 Example of a path pattern (left). Computation of the transitive closure of the same path pattern (right). 97
7.10 Example of a path pattern (left). Computation of the transitive closure of the same path pattern (right). 97
7.11 Mapping property paths to variables 99
7.12 Creating the canonical graph 99
7.13 Minimised (core) version of Q 100
7.14 Here `?c` should collapse into `?b` or `?b` into `?c` 100
7.15 A similar case to Figure 7.14, but with one edge mapping to two 100

7.16 A 2NFA that accepts words in $fold(a)$. 103

7.17 A 2NFA that accepts words in $fold(a^{-})$. 103

7.18 A 2NFA that accepts words in $fold($:p/:p^{-}/:p$)$. 104

7.19 Example of the steps of the union expansion of a path pattern. 106

8.1 File structure of project QCan 109

8.1 File structure of project QCan (continued) 110

8.1 File structure of project QCan (continued) 111

8.2 Example of a test case for UCQ rewriting, leaning, and canonical labelling. . 116

8.3 Example of a test case for filter rewriting and canonical labelling. 116

9.1 3-2D-Grid 120

9.2 2-3D-Grid 120

9.3 6-clique 120

9.4 4-triangle 120

9.5 An example of a Miyazaki Graph (Source: http://vlsicad.eecs.umich.edu/BK/SAUCY). 120

9.6 An example of a monotone query featuring a join of two unions of four triple patterns each. 121

9.7 Runtimes for each step of the canonicalisation algorithm 123

9.8 Runtimes for each step of the canonicalisation algorithm on BGPs 124

9.9 Runtimes for synthetic CQs 127

9.10 Times for UCQ stress tests 128

9.11 Runtimes for JSAG, SA and QCan 130

10.1 A SPARQL query based on a simplified version of a real query. 134

10.2 The running example now with canonical labels. 135

10.3 The running example now in a UCQ normal form, and having pushed the optional pattern inside. 135

10.4 The running example from Figure 10.3 with the IRIs replaced with variables. 136

10.5 The running example from Figure 10.4 with all predicates replaced with a generic IRI `:p`. 136

10.6 The simplest graph pattern. 137

List of Tables

4.1 Abstract SPARQL syntax 29

4.2 SPARQL property path syntax 30

4.3 Set evaluation of graph patterns where G is an RDF graph; B is a basic graph pattern; N is a navigational graph pattern; Q, Q_1 and Q_2 are graph patterns; V is a set of variables; R is a built-in expression; v is a variable; M is a set of solution mappings; and x is an IRI 31

4.4 Bag evaluation of graph patterns where G is an RDF graph; B is a basic graph pattern; N is a navigational graph pattern. 32

4.5 Bag evaluation of navigational patterns where G is an RDF graph, N is a navigational pattern, and x is a fresh blank node 33

4.6 Evaluation of dataset modifiers where X and X' are sets of IRIs 33

4.7 Aggregation algebra under bag semantics, where $\mathfrak{M}$ and $\mathfrak{M}'$ are bags of solution mappings; V is a set of variables and v is a variable; A is an aggregation expression; and $\mathcal{M}$ is a set of solution groups; we recall that $\mathbf{M}$ is the set of all solution mappings 36

4.8 Evaluation of group-by patterns where D is a dataset, Q is a graph pattern or group-by pattern, V is a set of variables, $v_1, \ldots, v_n$ are variables, and $A, A_1, \ldots, A_n$ are aggregation expressions 36

4.9 Complexity of SPARQL tasks on core fragments (considering combined complexity for EVALUATION). We annotate with * those results that are not stated but follow directly from results presented in the respective paper. Note that -C and -H stand for -COMPLETE and -HARD, respectively. 39

5.1 Equivalences given by Pérez et al. [56] for set semantics 43

5.2 Equivalences given by Schmidt et al. [67] for set semantics 65

5.3 Definitions for representational graphs $\mathrm{R}(Q)$ of query patterns Q, where "`a`" abbreviates `rdf:type`, B is a basic graph pattern, N is a navigational graph pattern, c is an RDF term, e is a non-simple property path (not an IRI), k is a non-zero natural number, v is a variable, w is a variable or IRI, x is an IRI, y is a variable or property path, μ is a solution mapping, Δ is a boolean value, V is a set of variables, R is a built-in expression, and $\mathfrak{M}$ is a bag of solution mappings. 68

5.4 Rewrite rules for property paths to UCQs 76

5.5 Rewrite rules for property paths in *inverse normal form* 76

7.1 Solutions for Q' over the canonical graph G 99

9.1 Distribution of features in the DBpedia (DBP), SWDF, RKB Explorer (REX), RKB Endpoint (REN), Wikidata (WD) and Linked Geo (GEO) sets of queries. 118

9.2 Total number of duplicates found by each method 126

9.3 Most duplicates of a single query found by each method 126

9.4 Total number of duplicate BGPs found by each method 126

9.5 Most duplicates of a single BGP found by each method 126

9.6 UCQ features supported by SPARQL Algebra (SA), Alternating Free two-way μ-calculus (AFMU), Tree Solver (TS) and Jena-SPARQL-API Graph-isomorphism (JSAG); note that ABGP denotes Acyclic Basic Graph Patterns 129

Chapter 1

Introduction

> I'm being quoted to introduce something, but I have no idea what it is and certainly don't endorse it.
>
> *Randall Munroe*

The purpose of the Semantic Web is to adapt or extend the current World Wide Web so that certain processes that require human reasoning and deduction can be performed automatically by machines. This is currently accomplished by structuring data in certain formats that can be interpreted by machines, and be operated on with specific protocols.

One such format corresponds to the *Resource Description Framework* (referred to as RDF hereafter)[27]. According to this framework, entities and their relationships are represented as a set of *triples* containing a *subject*, a *predicate*, and an *object* [27].

In order to expand RDF to allow for the construction of simple ontologies, a schema was designed: the *Resource Description Framework Schema* (RDFS) [14]. RDFS allows for creating a basic ontology over the data in RDF format, allowing simple inferences without the need for a user. Other more expressive languages exist, such as the *Web Ontology Language* (OWL) [36], that extend the complexity of the ontologies we may define, allowing for more advanced inferences to be made.

SPARQL [35] was then designed as the standard querying language for RDF as dictated by the World Wide Web Consortium (W3C). It features most of the necessary functions to carry out queries over *RDF* such as joins, unions, outer-joins, etc. [25]. SPARQL queries are executed over a set of these graphs: a default graph, and another set of named graphs.

Despite this, SPARQL was missing useful features such as (an explicit expression for) negation and the aggregation of results. Subsequently, these features were then implemented and introduced as part of SPARQL 1.1 [35]. Furthermore, because of the graph-like structure of RDF datasets, property paths were introduced. These allowed for the querying of relations between entities denoted by paths of arbitrary length, as well as to express queries more succinctly. Additionally, SPARQL allows for expressing federated sub-queries over external

SPARQL endpoints [60].

A SPARQL endpoint is an online interface that allows users to query a knowledge base using the SPARQL language over the Web via a standard protocol. These endpoints may each contain features specially catered for the needs of their users, which can also be accessed externally by SPARQL's support of federated queries. The results of these queries are typically returned in one or more machine-processable formats.

There has been a steady growth of the usage of RDF for the representation of data and metadata on the Web. For insance, a variety of public knowledge bases, such as DBPedia [45], Wikidata [75], and LinkedGeoData [69] have been published on the Web as RDF. As a result, the demand to query these datasets has also increased. Furthermore, there are currently many SPARQL endpoints on the Web [8]. Some examples include: Wikidata and DBpedia, both SPARQL endpoints that receive a vast number of queries everyday [65].

1.1 Problem

One problem that arises with the aforementioned SPARQL endpoints is the overload caused by processing large streams of queries, some of which may be duplicated. In fact, Aranda et al. [8] have concluded that renowned endpoints, such as DBpedia, are offline on average about 4% of the time, thereby showing the importance of having methods to reduce workload on said endpoints. In the particular case of duplicate queries, the machine must often perform the exact same routine to return the same answers, instead of re-using the results of these routines.

Since SPARQL is a rich and declarative query language, which offers an array of features to express the users' needs, there may be multiple ways to express the same abstract query. Some SPARQL queries can thus be semantically identical; that is, they will yield the same results over any RDF graph. We call such queries *equivalent* (denoted as $Q \equiv Q'$ for two queries Q and Q').

```
SELECT ?movie
WHERE{
    ?actor :actsIn ?movie;
        :salary ?salary .
}
```

```
SELECT ?movie
WHERE{
    ?actor :salary ?salary ;
        :actsIn ?movie .
}
```

Figure 1.1: An example of two equivalent SPARQL queries

Figure 1.1 contains two SPARQL queries that appear to be the same because the only difference between the two is the order of the triple patterns. Since the join operator is commutative, the results are the same for both queries over any RDF graph. Therefore, these two queries are equivalent.

On the other hand, the queries presented in Figure 1.2 are not equivalent, despite the fact that they both return "the same results". This is because they contain variables with different labels, but the same semantic value. For example, `?movie` and `?film` refer to the same thing: the movies or films that `?actor` and `?actor2` acted in. Instead, we call these queries *congruent* (denoted as $Q \cong Q'$ for two queries Q and Q') because we may rename the variables so as to make both queries equivalent. As such, the queries presented in Figure 1.2 are *congruent*, because the query on the right contains a redundant variable and a different order in the triples listed, as well as a redundant triple pattern.

```
SELECT DISTINCT ?movie ?salary
WHERE{
    ?actor :actsIn ?movie; :salary ?salary .
    ?actor2 :actsIn ?movie .
}
```

```
SELECT DISTINCT ?film ?wage
WHERE{
    ?actor2 :salary ?wage; :actsIn ?film .
    ?actor :actsIn ?film, ?film2 .
}
```

Figure 1.2: An example of two congruent SPARQL queries.

One way to reduce the workload on SPARQL endpoints would be to avoid executing equivalent (or congruent) queries multiple times in a limited time span. This issue is partially solved by caching the results [76] and simply returning the cached data if there's a request for a duplicated query. However, most methods only cache syntactically equivalent queries, which limits the amount of results that can be reused, as we will miss queries that are *semantically equivalent*.

Two queries are syntactically equivalent if the syntax of the queries is the same. This means that elements such as variables must have the same names, and the order of elements must be the same. On the other hand, queries are semantically equivalent if they have the same "meaning". For queries, this means that the evaluation of the query will produce the same results over any dataset, despite having a different syntax.

Currently, more syntactical equivalences are found by computing normalised queries. These normalisations are limited to techniques such as whitespace and lower-case normalisation on the received queries to facilitate the detection of duplicates. A number of endpoints also perform a transformation process over raw queries and output an algebraic form of said query, but this still only involves syntactic normalisation. The use of algebraic expressions is effective in order to perform optimisations prior to the processing of a query [26]. Indeed, the former techniques make the detection of syntactically equivalent duplicates trivial. However, the queries presented in Figure 1.1 are identical, but contain different variable names; the queries in Figure 1.2 contain redundant variables as well; therefore the syntactically normalised forms of neither pair would be equivalent.

Ideally, we could find queries that are not only semantically equivalent, but also congruent. Assuming a static dataset, we could then trivially rewrite the variables of the previously-evaluated congruent query in order to evaluate future queries.

However, identifying whether or not two queries are semantically equivalent, or congruent, is hardly a trivial matter. For example, Figure 1.3 denotes two congruent queries, but this can only be seen if we expand the path patterns into equivalent unions and basic graph patterns, and can reason about property paths such that we realise that the results of the third `UNION` argument add no new results versus evaluating the second `UNION` argument.

```
SELECT DISTINCT ?series
WHERE {
    ?series :instanceOf/:subclassOf* :TVSeries .
    ?series :genre|:subgenre :ScienceFiction .
}
```

```
SELECT DISTINCT ?series
WHERE {
    { ?series :instanceOf ?instance .
      ?instance :subClassOf* :TVSeries .
      ?series :genre :ScienceFiction . }
    UNION
    { ?series :instanceOf ?type .
      ?type :subclassOf* :TVSeries .
      ?series :subgenre :ScienceFiction . }
    UNION
    { ?series :instanceOf :TVSeries .
      ?series :subgenre :ScienceFiction . }
}
```

Figure 1.3: An example of two equivalent queries with property paths.

Therefore, given a large set of queries, such as those sent to a given endpoint by users, identifying which queries are equivalent or congruent is an even more challenging problem.

One option would be to check each pair of queries to determine those that are equivalent or congruent. However, such a solution would require a quadratic number of pair-wise checks. Evidently, such a solution is not viable for large-scale applications because each equivalence check is computationally complex.

Instead, we look into the concept of normalisation as a solution for syntactic equivalence, as we would only need to perform this normalisation once per query. This leads us to the concept of *canonicalisation*, which means to convert an input that can have multiple representations into a single, standard form that is called its *canonical form*.

A possible solution would then be to design a general algorithm to canonicalise queries. In reference to the problem at hand, it would mean converting SPARQL queries into a theoretical canonical form such that any two queries are congruent if, and only if, they are represented by the same syntactic expression in their canonical form.

Therefore, the main goal of this thesis is to develop canonicalisation methods for SPARQL including features introduced in SPARQL 1.1. These results may then allow us, for example, to extend caching methods to support a larger number of semantically equivalent queries. This would allow endpoints to detect a larger number of equivalent queries, thereby allowing us to reuse more query answers instead of computing them repeatedly.

The idea of canonicalising every possible SPARQL query seems optimistic, to say the least. As a matter of fact, for SPARQL, the equivalence problem is NP-complete even for well-

designed queries with basic features [1]. When we include more features, such as property paths, equivalence may become even harder to identify. In fact, the equivalence decision problem for queries with property paths (similar in complexity to navigational queries) is EXPSpace-complete [15]. For the full language, equivalence is undecidable.

Given that a canonicalisation procedure could be trivially used to solve equivalence, this means that even for fragments of SPARQL, the canonicalisation problem must have a high computational complexity. Furthermore, it means that a full canonicalisation procedure is not possible for the full SPARQL language.

On the other hand, while canonicalisation of SPARQL queries has a high worst-case complexity, most queries found in logs from renowned SPARQL endpoints are quite simple [65]. Therefore, we could reduce the scope of the canonicalisation to simpler queries and use existing efficient graph isomorphism algorithms to do a best-effort canonicalisation.

Our work proposes a method for the canonicalisation of SPARQL 1.1 queries that is *sound* for the full-language, which means that the method will produce a valid and congruent canonical query for any SPARQL query. In addition, said method is *complete* for a more limited fragment of the language, which means that the method will produce the same canonical query for all congruent queries in this fragment. This complete fragment consists in queries that contain only joins, unions and projections, otherwise known as monotone queries [1].

Finally, since there are clear overlaps between monotone queries with property paths and (unions of) conjunctive regular path queries [44], we expect that a sound and partially complete canonicalisation can be developed for monotone queries that contain property paths. In fact, path queries are equivalent to monotone queries if limited to property paths containing concatenations, alternate paths and inverse paths. However, complications arise for canonicalising property paths with Kleene's star.

1.2 Hypothesis

Our hypothesis is that we can design a canonicalisation algorithm and implement a system that can compute a canonical form for real-world SPARQL queries in reasonable time (subsecond on average) despite the theoretical complexity of the problem, and that this system can find more equivalent queries in real-world logs than syntactic methods alone. Additionally, this algorithm should 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 proposed algorithm is *sound* if the process does not alter the semantics of the query, or rather 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

[1]We refer to monotone queries as a syntactic fragment of SPARQL similar to the fragment of SQL studied by Sagiv et al. [62]. We distinguish it from the semantic property of being monotonic [10], i.e., having more results when data are added. Herein, all monotone queries are monotonic, but the inverse does not necessarily hold.

halts does not exist, since otherwise it could be used to solve the equivalence of first-order logic queries, a problem that is well-known to be undecidable. Despite the fact that the algorithm may not find the finest partition of congruent equivalence classes, we anticipate an improvement over systems that apply only a syntactic analysis of large sets of queries in terms of the number of duplicated detected.

To validate this hypothesis, we follow a methodology as follows:

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 [34, 31, 44, 3, 5, 56, 67, 6, 57]. In addition, we must research similar systems or systems that may aid in solving the proposed problem [41, 66, 54, 71, 21, 22, 80, 20].

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: The next step involves the 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: Here we will acquire data about 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 of the proposed method to demonstrate its utility in practice.

It is worth noting that due to the nature of the data necessary for the testing and evaluating stages, there is no need for human participants. In fact, the data can be compiled exclusively from existing SPARQL logs from well-known endpoints.

Consequently, the suggested methodology is iterative in nature. Feedback obtained after the evaluation stages may be utilised to optimise the algorithm as far as possible in this work. The results from previous iterations can be used as baselines for subsequent iterations, for instance, to further extend the algorithms to support features commonly found in queries in the logs [13].

1.2.1 Goals

The objective of this thesis is to develop a system for the canonicalisation of SPARQL. This includes both the features of SPARQL 1.0 as well as the addition of newer features that

were introduced as part of SPARQL 1.1. Furthermore, we may extend the completeness of the algorithm to include monotone queries with inequalities. 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 those use-cases. We anticipate the following applications:

Query caching: Papailiou et al. canonically labelled basic graph patterns in their study for caching. Our method could lead to a higher cache hit rate by capturing a broader class of congruent queries [54].

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, and may help to identify recurring abstract patterns [65, 13].

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 [43].

Learning over queries: For applications that apply learning over queries – such as question answering systems that translate natural language questions to structured queries – we may reduce the number of different queries in the training sets used to train machines by finding all unique, non-congruent queries and giving them a standard syntactic form, thereby reducing the syntactic variance in the training set [17].

1.3 Thesis Structure

In this section, we describe the overall structure of this thesis.

- Chapter 2 begins by presenting the background of this study, where we describe the key concepts necessary to understand this work. These include graph theory; RDF and its components, representation of information, and its semantic properties; and SPARQL, presenting features such as joins, unions, and projection.

- In the following chapter, we present the existing work related to this study, such as on the minimisation of SPARQL queries, canonical labeling of RDF graphs, and systems that check for containment (and equivalence) of SPARQL queries. Next, we present our work on defining the entirety of SPARQL 1.1 formally, which fills a gap in the literature, and is necessary in order to address the canonicalisation problem in a principled way. This includes features that are seldom discussed such as aggregation, federated queries, and the difference between the two forms of negation (`MINUS` vs. `FILTER NOT EXISTS`).

- Following this, we present our work on the canonicalisation of SPARQL for monotone queries – for which the process is *sound* and *complete* – and then show our support for additional features in both SPARQL 1.0 and 1.1. In particular, there is an emphasis in our work on the normalisation and minimisation of property paths, and its limitations.

- Building on the previous chapter, we then present the proofs for the *soundness* and *completeness* of our method for monotone queries, as well as its computational complexity. Subsequently, we present proofs for the *soundness* of our method for the entirety of SPARQL 1.1, but show that it is *incomplete* for the full language.
- Next, we describe how our method was implemented by listing the software used, the structure of the project, how testing was managed, etc. Subsequently, we describe the experiments we designed to evaluate the performance of our method, as well as the concrete results, and further discuss what these results allow us to conclude.
- In the next chapter, we present some use-cases for our method, and some preliminary results that show how our method may improve other systems.
- Finally, we conclude and summarise the results of this study, and discuss possible future work.

Chapter 2

Background

In this chapter, we present concepts that are relevant to this study. To begin, we describe key concepts of graph theory, such as definitions, and decision problems and their complexities. Following this, we present the main elements of RDF such as triples, RDF graphs, IRIs, etc, and how certain decision problems are viewed under the framework. Finally, we present SPARQL, its features and semantics, and decision problems and their complexities.

2.1 Graph Theory

A *directed graph* $G = (V, E)$ is composed of a set of *vertexes* V and a set of ordered pairs of vertexes named *edges* $E \subseteq V \times V$. Given $(v, v') \in E$, vertex v is said to be *connected* to v'. On the other hand, an *undirected* graph $G = (V, E)$ is a graph where V is a set of unordered pairs of vertexes. In other words, if $(v, v') \in E$, then it follows that $(v', v) \in E$.

Subsequently, given two undirected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, we say these graphs are *isomorphic* if there exists a bijection $\beta : V_G \to V_H$ such that $(v, v') \in E_G$ if and only if $(\beta(v), \beta(v')) \in E_H$. In such a case, β is called an *isomorphism*, and G and H are said to be *isomorphic* (written $G \cong H$).

Figure 2.1: An example of two isomorphic graphs.

It is clear from the example in Figure 2.1 that by defining a bijection β such that $\beta(a) = f$, $\beta(b) = g$, $\beta(c) = i$, and $\beta(d) = h$, the graph on the right can be obtained by mapping each node in the graph on the left using β.

Given two graphs, the graph isomorphism problem is a decision problem that returns true if both graphs are isomorphic, and false otherwise. This problem is known to be in the NP complexity class but it is not clear if it is NP-hard. For this reason, the graph isomorphism

problem has been assigned its own complexity class – GI-complete – where GI stands for graph isomorphism. In fact, Babai has determined that it can be solved in quasipolynomial time, meaning that GI is in QP [11].

A related concept is that of *homomorphism* of graphs. Unlike an isomorphism, which is a bijection, and therefore preserves both nodes and edges, a homomorphism may map different nodes from the original graph to a single node so long as adjacency between nodes is preserved. Also, it is not necessary for the image of the first graph under the homomorphism to correspond to the second graph, but rather it can map the first graph into a sub-graph of the second graph. More formally, given two undirected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, a mapping $\mu : V_G \to V_H$ is a *homomorphism* from G to H if and only if $(v, v') \in E_G$ implies $(\mu(v), \mu(v')) \in E_H$. If there is a homomorphism from G to H and from H to G, we can call G and H *homomorphically equivalent*. In contrast to the graph isomorphism decision problem, determining if there exists a homomorphism from G to H is NP-complete.

Furthermore, a homomorphism may be used to characterise a proper sub-graph of a graph. If G is a graph and H is a sub-graph of G, then there must exist a homomorphism from H to G. Indeed, we will design a mapping function μ as follows: for every edge $(v, v') \in H$, $\mu(v) = v$ and $\mu(v') = v'$. Clearly, this covers all of the edges of H since they are a subset of the edges of G. Next, for every edge in G such that $(v'', v''') \notin H$, $\mu(v'') = v$ and $\mu(v''') = v'$ where $(v, v') \in H$. We conclude that this mapping μ is a homomorphism.

However, not all graphs allow a homomorphism from itself to a proper sub-graph of itself. Given an undirected graph G, we say it is a *core* if every homomorphism from G to itself is an isomorphism (or an *automorphism*, given that it is an isomorphism from G to itself). This means that there is no way to map nodes from G to a sub-graph of itself with fewer nodes while preserving adjacency between nodes. Furthermore, if G is a core, $G \subseteq H$, and there exists a homomorphism from H to G, we may also call G the core of H, which is unique up to isomorphism.

2.2 RDF

RDF is a standard framework for the representation of information on the Web [68]. The information is organised in a semi-structured form, where each piece of information is represented as a 3-tuple (s, p, o) known as a *triple*, where s denotes the subject, p denotes the predicate, and o denotes the object. Alternatively, we may interpret them as a node labelled s connected to a node labelled o, forming an edge $s \xrightarrow{p} o$.

Elements in RDF are known as *resources*. A resource may denote anything that exists. These resources may be referred to by an *Internationalised Resource Identifier* (IRI from this point onwards), a *literal* or a *blank node*.

IRIs are used to identify resources in a manner that is generalised and standardised globally, so as to avoid ambiguity or erroneous deductions when handling requests. They may or may not be resolvable over the Web. The resource identified by an IRI is called its referent. We define $\mathbf{I}$ as the set of all IRIs.

Herein, we use the Turtle syntax, where IRIs may be written as *prefixed names*. A prefixed name consists of a prefix label and a local name separated by a colon (:). For example, the prefix `rdf` refers to the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, so the IRI `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` may be written as `rdf:type`.

A *literal* is an element that is defined exclusively by its literal value, such as a numeric value. We define $\mathbf{L}$ as the set containing all literals. A literal also has a datatype, which denotes the range of values it may take, such as strings, numbers, dates, etc. In order to specify a datatype, RDF primarily uses XML Schema Datatypes, which are referred to by IRIs with the `xsd` prefix and the name of the datatype (for example `^^xsd:boolean` refers to the boolean datatype). If no datatype is specified, the literal is assumed to be a string.

A *blank node* is an element used to represent a value that is unknown, but is known to exist. Implementations for the handling of blank nodes are dependant on the application [48]. We define $\mathbf{B}$ as the set that contains all blank nodes. Blank nodes are written as an underscore followed by a colon, and a blank node label. For example, `_:b` refers to a blank node with `b` as its label.

A subject may be an IRI or a blank node; a predicate may only be an IRI; an object may be an IRI, a literal value or a blank node. Therefore, we may denote $\mathbf{S} = (\mathbf{I} \cup \mathbf{B})$ as the set of all subjects, $\mathbf{P} = \mathbf{I}$ as the set of all predicates, and $\mathbf{O} = (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ as the set of all objects.

An *RDF graph* is then a set of triples $G \subseteq \mathbf{S} \times \mathbf{P} \times \mathbf{O}$, which can be visualised as a directed labelled graph. To do this, assume $subjects()$, $predicates()$ and $objects()$ are functions such that $subjects(G)$, $predicates(G)$ and $objects(G)$ return all the subjects, predicates or objects, respectively, that appear in an RDF graph G. We may then define the set of vertices of G as $nodes(G) = subjects(G) \cup objects(G)$. Furthermore, the set of edges of G is determined by the set of triples, and the set of edge labels by $predicates(G)$. Finally, a *(proper) sub-graph* of an RDF graph is a (proper) subset of the triples in the graph.

Figure 2.2 shows an example of an RDF graph that describes *Pokemon Yellow* (represented by the IRI `:PokemonYellow`), its release dates in different regions, and its developer, *Game Freak* (represented by the IRI `:GameFreak`). One notable feature is that `:Japan` appears as an object in two different triples, showing that RDF graphs may often present cycles (both directed and undirected). Another feature of this graph is the presence of blank nodes to describe the release dates of *Pokemon Yellow* in different regions. This is because we are representing an n-ary relation between `:PokemonYellow`, its release dates, and each region.

However, an RDF graph doesn't necessarily contain blank nodes; triples may consist entirely of IRIs or literals (where appropriate). Such a graph is called a *ground* RDF graph. The importance of ground graphs in this study stems from the fact it does not have any ambiguities associated to blank nodes, which means that determining if two graphs are equivalent is as trivial as checking string equivalence over a concatenation of a canonical ordering and serialisation of the triples. We provide an example of a ground RDF graph in Figure 2.3.

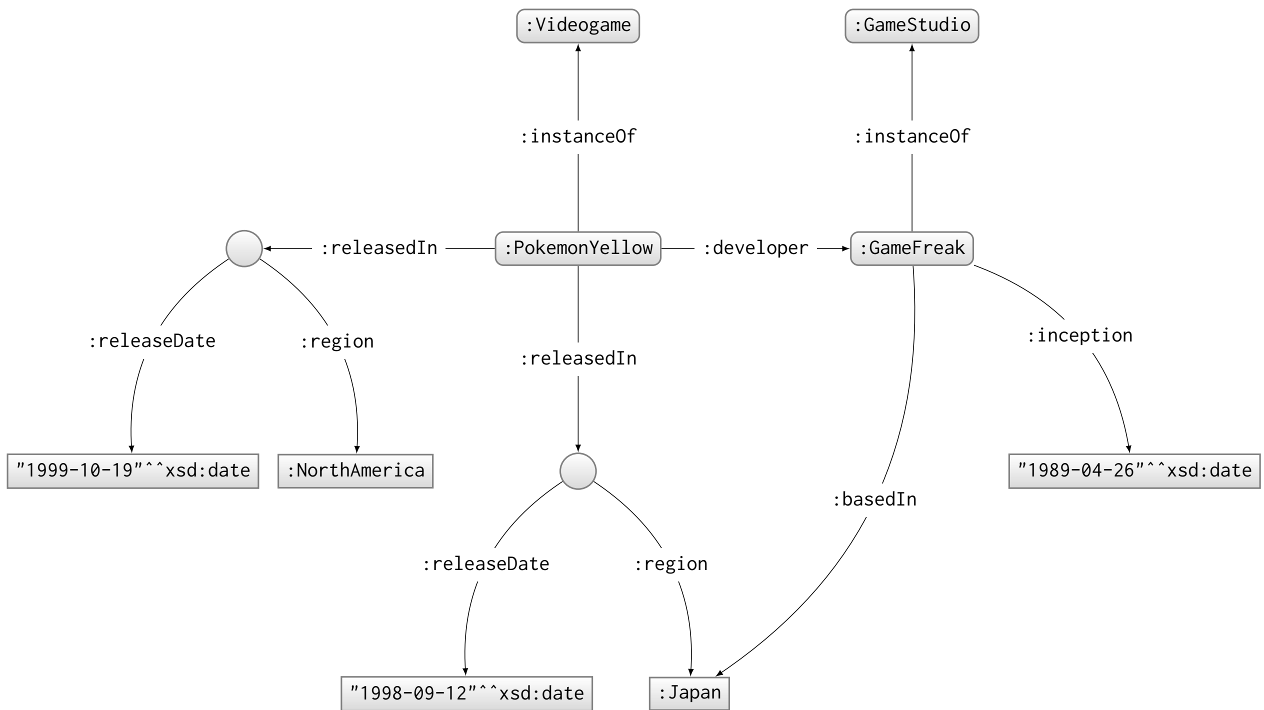


Figure 2.2: An example of an RDF graph about Pokemon Yellow.

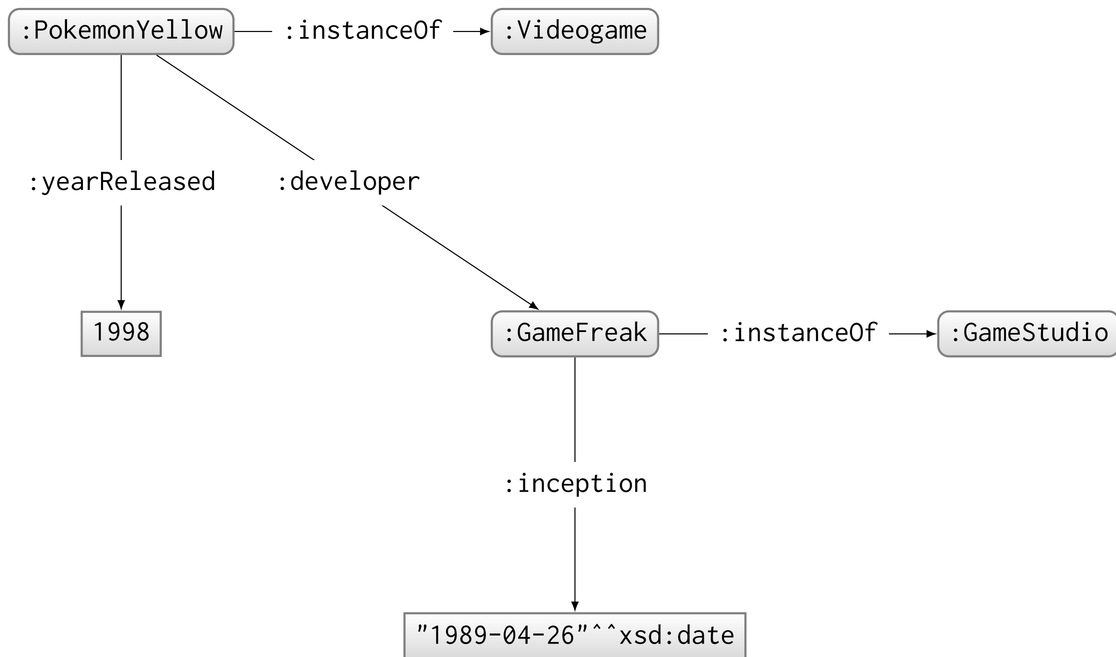


Figure 2.3: Ground graph example

There are times where one may need to state claims about triples themselves. For instance, we may need to specify when a triple was added or last modified. A way to address this is through *reification*. In this way, we represent the information of a triple by making statements about its structure and its contents. We denote the existence of the triple with a blank node or IRI (depending on the form of reification), and we add triples denoting the subject, predicate and object of the original triple. Figure 2.4 presents an example of a reified representation

of a triple (`:PokemonYellow` ,`:developer` ,`:GameFreak`) that was last updated on February 17th 2013. We note that, by default, the reified statement is not asserted; this makes sense considering that the reification may qualify the statement to be outdated, of low confidence, or outright false. In order to assert the statement, the corresponding triple must also be added to the graph (in the case of Figure 2.4, this would involve adding an edge from `:PokemonYellow` to `:GameFreak` labelled `:developer`).

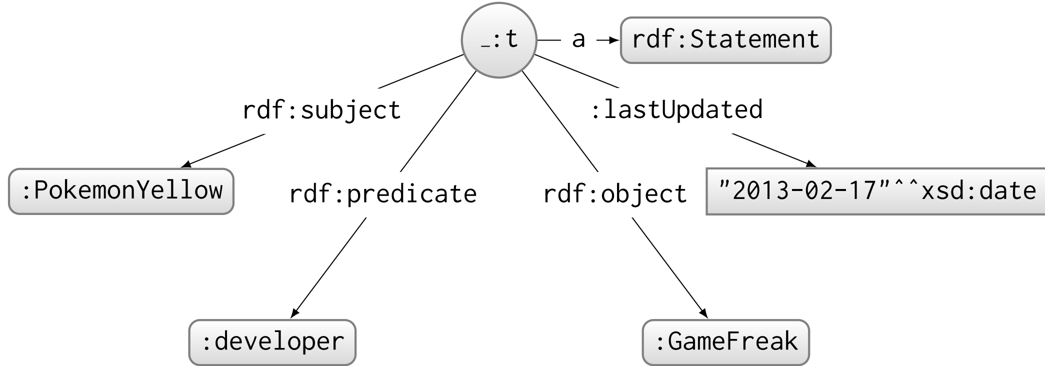


Figure 2.4: Example of a reification of a triple. Here `_:t` is used to denote the triple.

2.2.1 Semantics of RDF

An RDF graph is a set of triples. The semantics of RDF graphs is defined according to the notion of a *simple interpretation* I, which, per the W3C standard [35], consists of:

- A non-empty set R_I of resources, called the domain of I.
- A set P_I, called the set of properties of I.
- A mapping EXT_I from P_I to the powerset of $R_I \times R_I$.
- A mapping S_I from IRIs into $(R_I \cup P_I)$.
- A partial mapping L_I from literals into R_I.

In order to understand what it means for an RDF graph to be true under a simple interpretation, $EXT_I(p)$ denotes the set of pairs of resources for which the property p is true. The interpretation is then treated as a function from RDF terms, triples and graphs to resources and truth values, as follows:

- If $x \in \mathbf{L}$ then $I(x) = L_I(x)$.
- If $x \in \mathbf{I}$ then $I(x) = S_I(x)$.
- If t is a triple (s, p, o), then $I(t) = \mathit{true}$ if $I(p) \in P_I$ and $(I(s), I(o)) \in EXT_I(I(p))$. Otherwise, $I(t) = \mathit{false}$.

- If G is an RDF graph, then $I(G) = false$ if $I(t) = false$ for some triple $t \in G$. Otherwise $I(G) = true$.

Additionally, for non-ground RDF graphs, this is extended to include a mapping A from blank nodes to resources in R_I, resulting in the piecewise mapping $[I + A]$ such that $[I + A](x) = I(x)$ if x is a literal or IRI, and $[I + A](x) = A(x)$ if x is a blank node. In that case, the mapping for triples and RDF graphs is defined analogously. As a result, the truth condition for a non-ground RDF graph is: if G is an RDF graph, then $I(G) = true$ if and only if $[I + A](G) = true$ for some blank node mapping A that maps the blank nodes in G to values in R_I. In other words, this means that an RDF graph is semantically interpreted as the conjunction of the set of triples it contains, where blank nodes act as existential variables.

2.2.2 Isomorphism in RDF

Aside from the semantics of RDF, an important structural property of RDF graphs is that of *RDF isomorphism* [27]. This notion captures the relation between RDF graphs that are identical modulo blank node labels (which, we recall, denote existential variables, whose particular values do not affect interpretations of the RDF graphs).

Let $\mu : \mathbf{ILB} \rightarrow \mathbf{ILB}$ be a mapping of RDF terms to RDF terms. If μ is the identity on **IL**, that is, IRIs and literals remain the same, then it is called a *blank node mapping*. Furthermore, let its domain be denoted as $\mathrm{dom}(\mu)$ and its codomain denoted as $\mathrm{codom}(\mu)$. Such a function is applied to an RDF graph G node by node, resulting in a new graph $H = \mu(G)$ (by using $\mu(G)$ to denote the image of G under μ). The resulting graph is called an *instance* of graph G. If at least one blank node has been replaced by a name in **IL**, or two blank nodes have been mapped into the same node, it is a *proper instance*. Otherwise, if the blank nodes in $\mathrm{dom}(\mu)$ are mapped to blank nodes in $\mathrm{codom}(\mu)$ in a bijective manner, then μ is called a *blank node bijection*.

Subsequently, an RDF graph G is *isomorphic* with another RDF graph G' (denoted as $G \simeq G'$) if there exists a blank node bijection μ such that $\mu(G) = G'$. In such a case, μ is called an *isomorphism*. In other words, isomorphism indicates that the result of a one-to-one blank node mapping function over one graph is the other graph.

An *automorphism* of an RDF graph G is an isomorphism μ such that $\mu(G) = G$. In essence, it is a map of G to itself. Evidently, the identity mapping is an automorphism, and is known as a *trivial automorphism*. If μ is a mapping function different from the identity mapping, it is known as a *non-trivial automorphism*. Finally, we denote by $Aut(G)$ the set of automorphisms of G.

2.2.3 Entailment and Equivalence in RDF

While the isomorphism relation compares RDF graphs based on the structure of the graph, assuming that blank nodes are simply entities whose labels are inconsequential, then the *equivalence* relation takes into consideration the *semantics* (rather than the structure) of

RDF graphs, i.e. the meaning of the data contained in the graph. In the latter case, blank nodes are considered values that are known to exist. Therefore, they are of utmost importance when comparing the *semantic equivalence* of two graphs.

Firstly, we must address the matter of *entailment*. An RDF graph G entails another RDF graph G' if every possible interpretation I that makes G true also makes G' true. In that case, it is written $G \models G'$. Although there exist more complex entailments under ontologies such as OWL, for the purposes of this study, we shall focus on *simple entailment* derived from simple interpretations, and will further refer to it as *entailment*.

Evidently, if a graph G has been established as true under an interpretation I, it means that all of its triples are true under that interpretation. Since any sub-graph $G' \subseteq G$ can only contain triples from G, all of which are true under I, it must be true as well. In fact, an RDF graph G entails G' if, and only if, there exists a mapping μ such that $\mu(G')$ is a sub-graph of G [34].

Because of this, the entailment decision problem may be solved by checking if there exists a homomorphism from G' to G. In such a case, the homomorphism we look for is the mapping μ such that $\mu(G') \subseteq G$. This problem is known to be NP-complete because given a homomorphism μ from G' to G as a witness, we can determine if μ is a valid mapping in polynomial time, putting the problem in NP. On the other hand, we can trivially reduce traditional graph homomorphism to this problem, giving the lower bound. As a result, the entailment decision problem is also NP-complete.

An RDF graph G is *equivalent* to another RDF graph G' if G entails G', and G' entails G. In such a case, it is written $G \equiv G'$. If two RDF graphs are equivalent, it can be interpreted as their being *semantically equivalent*. This means that, under RDF semantics, they represent the same information.

In graph terminology, we can analogously say that G and G' are equivalent if and only if there exists a homomorphism from G to G' and a homomorphism from G' to G, in which case we call G and G' *homomorphically equivalent*. Thus semantic equivalence and homomorphic equivalence coincide for RDF graphs (under simple semantics), and we can generically refer to RDF graphs as being equivalent (under either notion).

Determining equivalence between RDF graphs is also NP-complete. This is because to determine equivalence, we can check for entailment in both directions [57].

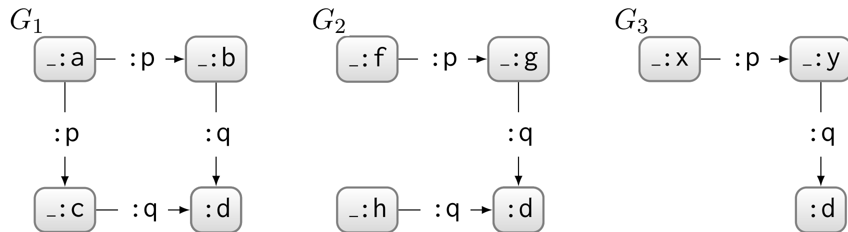


Figure 2.5: An example of three RDF graphs that are semantically equivalent.

Indeed, one can appreciate from the example in Figure 2.5 that an RDF graph can model the same information as a larger RDF graph, if they are equivalent. Taking this into

consideration, we call an RDF graph G *lean* if it does not contain a proper sub-graph G' such that $G \equiv G'$.

Determining if a graph G is lean is coNP-complete. This is because, given a graph G, a proper sub-graph G' such that $G' \equiv G$, and a homomorphism $\mu : V \to V'$ as our witness, we can verify in polynomial time that μ is a valid homomorphism from G to G', which leads us to the conclusion that G is *not* lean [34].

Finally, as a result of the previous statements, determining if a graph C is the core of a graph G is DP-complete, the complexity class of decision problems that can be solved by answering both an NP-complete problem and a coNP-complete problem. In this case, we can check if a graph C is the core of G by checking that C is equivalent to G (an NP-complete problem) and that C is lean (a coNP-complete problem) [34].

2.2.4 Canonicalisation of RDF graphs

With respect specifically to canonicalisation, to the best of our knowledge, little work has been done on SPARQL. In analyses of logs, authors have proposed very basic canonicalisation methods, such as normalising whitespace, that do manage to detect some duplicates, but not more complex cases such as in the examples provided previously. On the other hand, Papailiou et al. have proposed using isomorphism of basic graph patterns for a query caching system [54]. However, there are some recent related works on efficiently canonicalising RDF graphs that could be leveraged for the case of SPARQL [32]. The idea is that the structure of a SPARQL query can be replicated as an RDF graph, and that we can adapt this canonicalisation algorithm for SPARQL.

Hence, the technical problem we tackle in this work will be to design an algorithm to efficiently canonicalise SPARQL queries based on existing techniques for canonicalising RDF graphs. We now introduce the latter techniques for canonicalising RDF graphs.

Iso-canonicalisation of RDF graphs

An *iso-canonicalisation* of a graph G, $\text{L}(G)$ produces a graph such that $G \simeq \text{L}(G)$, and for any graph G' it follows that $\text{L}(G) = \text{L}(G')$ if and only if $G \simeq G'$ [38].

The main issue we have to tackle in this form of canonicalisation is the labelling of blank nodes. The labels of blank nodes are largely inconsequential when it comes to executing a query. However, they are of essence to verify if two graphs are isomorphic. When it comes to RDF graphs, if a graph is grounded, verifying isomorphism is simply a matter of verifying whether or not both graphs contain the same triples.

The proposed solution to deal with blank nodes is to canonically label them. This canonical labelling needs to be deterministic under graph isomorphism. Proposals for practical graph isomorphism algorithms have been studied for decades as far back as the 60's [24], and as recent as a few years ago [50]. However, the aim of this study is not to improve upon existing graph isomorphism algorithms, nor to evaluate their performance outside of

the specifics of this work. Rather, we investigate a method for SPARQL queries that uses such techniques. One such system is BLabel, which implements an algorithm for canonically labelling blank nodes in RDF graphs along these lines [38], as we now summarise.

For each name (i.e. IRI or literal) in the input graph, a unique fixed hash is assigned based on its syntactic string value. As for blank nodes they are all assigned the same initial hash. Iteratively, each blank node is assigned a new hash which is computed from the hashes of the terms of the triples it appears in. The process must be iterative because if two blank nodes are in the same triple, changes in the hash of one of them will affect the other. Thus, this is repeated until no new partitions are found. This algorithm is analogous to an RDF-specific version of the classical Weisfeiler–Lehman algorithm (WL) [2]. However, not all blank nodes may be distinguished by this algorithm.

In case there are still blank nodes with the same value after the previous process, the partitions of blank nodes are deterministically ordered based on their size, and their current hash value. From the first partition in the order, a blank node is selected non-deterministically, and distinguished by combining its hash with a fixed value. This process is then repeated until we reach a fixpoint, where no additional partitions are distinguished.

All non-deterministic choices are explored following the steps above. For each resulting graph with distinguished blank nodes, the smallest such graph in a total order is chosen as the canonical graph.

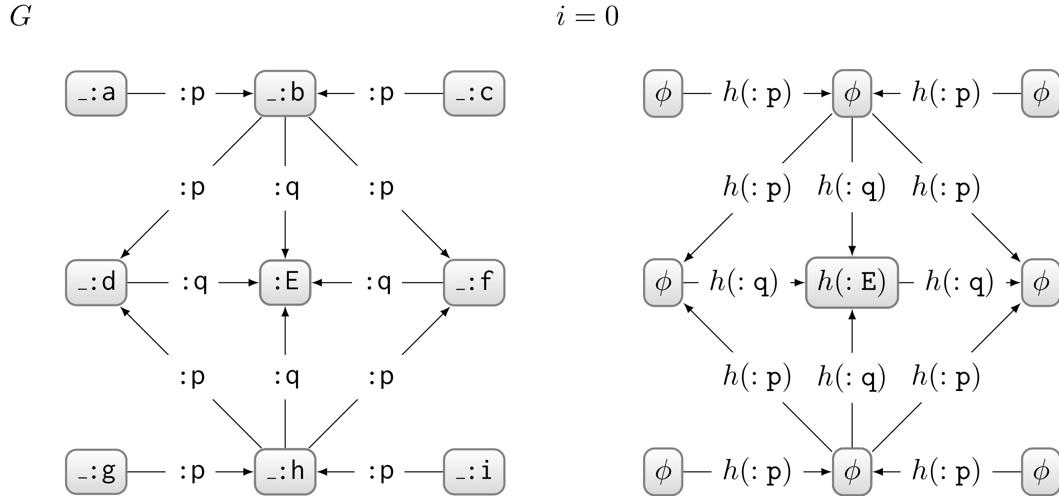
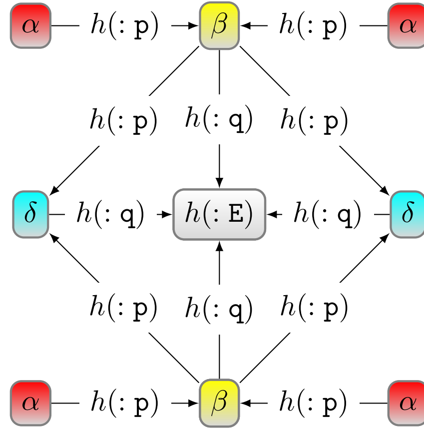


Figure 2.6: Example of labelling process

Figure 2.6 shows the initial graph G, with blank node labels from _:a to _:h with the exception of :E , which corresponds to an IRI. At the beginning ($i = 0$), all blank node labels have the same hash value ϕ. Literals and IRIs are hashed according to their syntactic values.

Figure 2.7 shows the next steps of the algorithm. At $i = 1$, blank node labels are assigned hash values depending on their adjacent nodes and edges. In the case of edges, there is a distinction between outgoing edges and ingoing edges. We can clearly appreciate a partition in blank node labels, illustrated with colours: {_:a, _:c, _:g, _:i} (coloured red), {_:b, _:h} (coloured yellow), and {_:d, _:f} (coloured blue). If run for another iteration, the hashes may change, but the partitions will not change. This is because the blank nodes in each partition

$i = 1$ $i = 2$



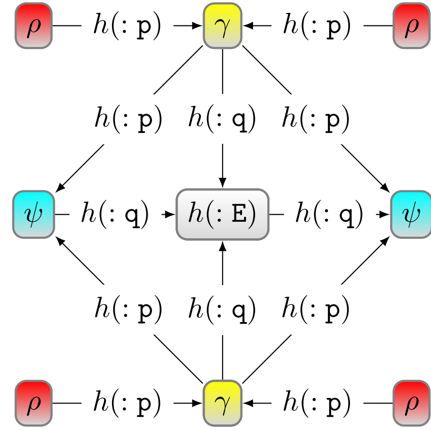


Figure 2.7: Example of labelling process (continued)

are indistinguishable by the method used, since the information used for the hashing is the same (i.e., they have the same edges). At $i = 2$, the blank node labels have changed, but the partition remains the same. At this point, the algorithm finishes since there are no differences in the partition of blank nodes.

In cases where the fixpoint of the hashing process does not distinguish all blank nodes, the hash value of a partition is used as an invariant in a NAUTY recursive search [24].

Equi-canonicalisation of RDF graphs

An equi-canonicalisation of an RDF graph G [38], $\mathrm{eCan}(G)$, produces an RDF graph such that $G \equiv \mathrm{eCan}(G)$, and given graphs G_1 and G_2, $\mathrm{eCan}(G_1) = \mathrm{eCan}(G_2)$ if and only if $G_1 \equiv G_2$. Furthermore, $\mathrm{eCan}(G)$ is lean and thus unique up to isomorphism. In other words, the equi-canonical form of graph G retains its semantic values,

We may accomplish this by computing the iso-canonicalisation of the core of G because the core of an RDF graph is equivalent to G (because it is a sub-graph of G), and is unique up to blank node labels. Subsequently, the algorithm used to compute the core of a graph G is described as follows [38]:

In the first step, an initial set of blank nodes that cannot be mapped to another term by a homomorphism (called unique blank nodes) is computed. These blank nodes are identified by the analysis of the ground terms in their direct edges. Evidently, if the ground terms are unique to a blank node, it cannot be mapped to any other blank node. In a later iteration, these unique blank nodes are *grounded*, which means that for all intents and purposes they act like an IRI or a literal, which also means that they will appear in the final core. Consequently, another term may become unique due to its direct edges. Therefore, the grounding process can be applied iteratively until no new groundings occur. If all blank nodes have been grounded, the graph is lean: this happens in the vast majority of real-world graphs [65]. In case it does not occur, the algorithm computes homomorphisms from the non-ground part of the graph into itself (*endomorphisms*) that produce proper instances of the graph, iteratively,

until no further proper instances of the graph to itself can be found. It is at this point that this algorithm may become exponential because it may need to compute all combinations of endomorphisms. However, this process can be optimised to reduce the practical cost of this algorithm, for instance, by making use of the degree of symmetry of some graphs [50].

2.3 SPARQL

Now we will introduce some key concepts of SPARQL [35], the standard query language for RDF. In this section, we introduce formal definitions specifically for SPARQL 1.0, as defined, for example, by Pérez et al. [55]. We begin by summarising the abstract syntax of queries. Thereafter we present the concept of solution mappings and the algebra of operations on sets of solution mappings. Finally, we discuss the evaluation of queries under both set and bag semantics.

2.3.1 Abstract Syntax

In this section, we describe the abstract syntax of SPARQL queries. Since SPARQL is essentially a graph matching language, the queries are based on different types of graph patterns. Therefore, we begin by defining $\mathbf{V}$ as the set that contains all variables, after which we define the simplest of graph patterns, followed by the definition of the main operations successively.

- Let t be an RDF triple that allows variables in any of its components. In that case, t is a *triple pattern*. Furthermore, a *triple pattern* is also a *query pattern*.
- If Q_1 and Q_2 are query patterns, then $[Q_1 \text{ AND } Q_2]$, $[Q_1 \text{ UNION } Q_2]$ and $[Q_1 \text{ OPT } Q_2]$ are query patterns.
- Let R be a *built-in expression* that contains elements of **VIBL** and applies comparisons, restrictions or other built-in operations. A boolean combination of such expressions is also an expression. If Q is a query pattern and R is a built-in expression, then $\text{FILTER}_R(Q)$ is also a query pattern.
- If Q is a query pattern and V is a set of variables, then $\text{SELECT}_V(Q)$ is also a query pattern.

2.3.2 Solution mappings

A solution to a SPARQL query Q is a partial mapping from the variables that appear in the query (denoted as $\text{vars}(Q)$) to terms from **IBL** appearing in the data. Such mappings will be denoted by μ. Let $\text{dom}(\mu)$ denote the variables for which a mapping μ is defined. Given $\{v_1, \ldots, v_n\} \subseteq \mathbf{V}$, and given $\{x_1 \ldots, x_n\} \subseteq \mathbf{IBL} \cup \{\bot\}$, we denote by $\{v_1/x_1, \ldots, v_n/x_n\}$ the mapping μ such that $\text{dom}(\mu) = \{v_i \mid x_i \neq \bot\}$ for $1 \leq i \leq n$ and $\mu(v_i) = x_i$ for

$v_i \in \text{dom}(\mu)$. Let μ_1 and μ_2 be mappings. They are said to be *compatible* if and only if for every $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it holds that $\mu_1(v) = \mu_2(v)$, which is denoted as $\mu_1 \sim \mu_2$. Additionally, if $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) \neq \emptyset$ then μ_1 and μ_2 are said to be *overlapping*, denoted as $\mu_1 * \mu_2$.

Given a built-in expression R, we denote by $\mu(R)$ the result of mapping the variables that appear in R (denoted as $\text{vars}(R)$) according to μ. Furthermore, we say that μ satisfies R (denoted $\mu \models R$) if $\mu(R)$ is evaluated as *true*. For example, let $R = isIRI(x)$ where $isIRI$ is a function that returns *true* if the value bound to the variable x is an IRI. Then $\mu \models R$ if and only if $\mu(x)$ is an IRI.

2.3.3 Algebra

Let M, M_1, M_2, etc. denote sets of solution mappings. Furthermore, let $\bowtie$, $\cup$, $\rhd$, and $⟕$ denote the join, union, anti-join, and left outer join operators for sets, respectively. We define the algebra of solution mappings as follows:

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

2.3.4 Semantics

The following definitions denote the evaluation of query patterns over an RDF graph G.

$$
\begin{aligned}
B(G) &:= \{\mu \mid \exists \alpha\ :\ \mu(\alpha(B)) \subseteq G,\ \text{dom}(\mu) = \text{vars}(B) \textit{ and } \text{dom}(\alpha) = \text{bnodes}(B)\ \} \\
[Q_1 \text{ AND } Q_2](G) &:= Q_1(G) \bowtie Q_2(G) \\
[Q_1 \text{ UNION } Q_2](G) &:= Q_1(G) \cup Q_2(G) \\
[Q_1 \text{ OPT } Q_2](G) &:= Q_1(G) ⟕ Q_2(G) \\
\text{FILTER}_R(Q)(G) &:= \sigma_R(Q(G)) \\
\text{SELECT}_V(Q)(G) &:= \pi_V(Q(G))
\end{aligned}
$$

2.3.5 Set or Bag Semantics

The SPARQL standard considers *bag semantics* for query answering by default. In other words, duplicate solutions are preserved and returned as solutions.

These semantics are captured using the extended query syntax:

$$\textsc{select}_V(Q)^{\Delta}, \text{ where } \Delta = \begin{cases} true \text{ for set semantics} \\ false \text{ for bag semantics} \end{cases}$$

As of SPARQL 1.0, explicit projection is limited to only be used outside all other query operators. Projection of basic graph patterns may be emulated using blank nodes as existential variables. SPARQL 1.1, as defined later, allows the projection of sub-queries.

2.3.6 Classes of Queries

In this section, we describe classes of SPARQL queries based on the features they contain, as well as the general structure of the graph patterns.

Conjunctive Queries (CQs) These are queries that contain only projections and joins of triple patterns. *Basic graph patterns* are also conjunctive queries in SPARQL where blank nodes can be used to emulate projection.

Unions of Conjunctive Queries (UCQs) These are queries that consist in projections of unions of conjunctive queries. They are analogous to first-order logic queries in a disjunctive normal form.

Monotone Queries (MQs) These are queries that contain only joins, unions, and projections. In particular, both CQs and UCQs are monotone queries [62].

2.3.7 Safe Variables

We present the notion of *safe variables* as denoted by Schmidt et al. in their study [67]. These are the variables in Q that are always bound in $Q(G)$ for any RDF graph G. Whether or not variables are safe may affect equivalence rules, which we will use later for canonicalisation.

$$
\begin{aligned}
\text{safeVars}(t) &:= \text{vars}(t)\\
\text{safeVars}([Q_1 \text{ AND } Q_2]) &:= \text{safeVars}(Q_1) \cup \text{safeVars}(Q_2)\\
\text{safeVars}([Q_1 \text{ UNION } Q_2]) &:= \text{safeVars}(Q_1) \cap \text{safeVars}(Q_2)\\
\text{safeVars}([Q_1 \text{ OPT } Q_2]) &:= \text{safeVars}(Q_1)\\
\text{safeVars}([Q_1 \text{ MINUS } Q_2]) &:= \text{safeVars}(Q_1)\\
\text{safeVars}(\text{FILTER}_R(Q)) &:= \text{safeVars}(Q)\\
\text{safeVars}(\text{SELECT}_V(Q)) &:= \text{safeVars}(Q) \cap V
\end{aligned}
$$

2.3.8 Other features

We have formalised the main features in SPARQL 1.0. Here we present other features which are not formally defined in most literature in concrete SPARQL syntax:

ORDER BY

It may be of interest to establish an order over the results of query, for instance, if we want to find the top results based on a certain value.

An ORDER BY clause determines the order of the results. It is followed by a sequence of expressions composed of an *order modifier* (either ASC or DESC) with a variable. The order modifier may be omitted, in which case it is assumed to be ordered in ascending order. Figure 2.8 presents an example in concrete SPARQL syntax that presents actors in order of being lowest-paid.

```
SELECT ?actor
WHERE{
    ?actor :actsIn ?movie;
        :salary ?salary .
}
ORDER BY ?salary
```

Figure 2.8: Example of a query that contains an `ORDER BY` clause.

LIMIT, OFFSET

If we take into account a total order over the results, as defined by an ORDER BY clause, we may wish to limit the number of results returned by a query. Otherwise, we may want to limit the number of results to reduce the workload of a machine. Regardless, SPARQL features the LIMIT clause, which is followed by a number n, and indicates that out of all the possible results, only the first n are returned. On the other hand, an OFFSET clause is also followed by a number n, and indicates that the first result it returns is the nth result.

If both LIMIT and OFFSET are used, with m and n respectively, the query will return the first m results starting from the nth result. Figure 2.9 presents an example of the usage of both LIMIT and OFFSET in concrete SPARQL syntax, finding the actors in position 51–60 in terms of being lowest paid.

```
SELECT ?actor
WHERE{
    ?actor :actsIn ?movie;
        :salary ?salary .
}
LIMIT 50
OFFSET 10
```

Figure 2.9: Example of a query that contains `LIMIT` and `OFFSET`

Chapter 3

Related Work

In this section we present and discuss some studies we encountered during the *Literature Review* stage of this work relating to normalisation and minimisation rules for SPARQL queries, followed by systems that allow for some form of canonicalisation for RDF and SPARQL, and systems that check for containment of SPARQL queries. We briefly explain their overall results, as well as their more relevant conclusions to our study.

3.1 Rules for Minimisation and Normalisation

Pérez et al. [56] 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 [67] 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 [34].

3.2 Systems for RDF and SPARQL Canonicalisation

Hogan designed a system for the canonical labeling and leaning of RDF graphs (BLABEL) [37]. 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 [50].

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. [54] utilised a canonical labeling algorithm (specifically Bliss [41]) on basic graph patterns in order to either cache

results or find stored results from isomorphic queries. However, the study uses canonical labeling rather than offering a sound and complete canonicalisation for SPARQL, which means that some equivalences and congruences of monotone queries may be missed.

3.3 Systems for SPARQL Containment

Perhaps 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 practice.

For SPARQL, a number of systems have been implemented to solve the containment problem on queries under certain constraints. These are `SPARQL Algebra` [46] and the `Jena-SPARQL-API Graph-isomorphism` (`JSAG`) tool [70]. In particular, the approach in the study by Stadler et al. [70] 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 [21] where queries are translated into formulae which are then processed by either an *Alternation Free two-way μ-calculus* ($AFMU$) decider [72] or `Tree Solver` [33], 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.

Although the aforementioned containment checkers may allow us to determine whether two queries are equivalent, this approach is not efficient if we need to find all equivalences in a large set of queries. This is because we would need to perform a quadratic number of pairwise checks in order to find all equivalences. Even then, we would not be able to find any pairs of queries that are congruent but not equivalent.

3.4 Other Query Languages

XPath is an expression language for the retrieval of nodes in XML documents. These expressions can also be visualised as *tree patterns*. It is used in query languages such as XQuery, the standard query language for XML according to the W3C. XPath is flexible and expressive enough to be used to query graph databases as well as XML documents [47].

The containment decision problem for XPath queries with constraints known as Document Type Definitions (DTDs) is decidable for a fragment that contains tests for predicates, wildcards, and searches for descendants, otherwise known as *tree patterns* [78, 53]. The same problem without DTDs is known to be in coNP-complete [51]. Furthermore, this complex-

ity does not depend on whether the patterns are evaluated over tree structures or graph databases [29].

In regards to minimisation, the XPath tree pattern minimisation decision problem is in Π_2^P in the general case [77], but there are certain fragments that allow minimisation in PTime[78].

Interestingly, unlike monotone SPARQL queries, one can construct tree patterns that contain no redundancies, but are not minimal. This means that non-redundancy does not imply minimality [28].

Chapter 4

SPARQL 1.1

While there exist various formal definitions for the core of SPARQL 1.0, as discussed in Section 2.3, to the best of our knowledge, there does not exist a similar comprehensive treatment for SPARQL 1.1 (though parts of the SPARQL 1.1 query language have been defined formally [58, 4, 9, 59, 42]).

In this chapter, we present our extended definitions for SPARQL 1.1. These acknowledge the features added in SPARQL 1.1 such as property paths, aggregation expressions, etc. Furthermore, we have filled some of the gaps in the literature such as the subtle difference between `MINUS` and `FILTER NOT EXISTS`, formalise the semantics of bags of solution mappings, and federated queries. Such features are commonly used in real-world queries [12, 8], and defining them formally is an important prerequisite before proposing canonicalisation methods. We will describe the results that are most relevant to this work, but we refer to our study for a more thorough and detailed explanation [64]. For example, for brevity, here we will exclude the formal definition of features relating to ordering results from SPARQL 1.0, for which we do not apply a special treatment during canonicalisation, but these definitions can be found in [64]. Likewise we focus on the canonicalisation and definition of `SELECT` queries, where definitions for `ASK`, `CONSTRUCT` and `DESCRIBE` can rather be found in [64].

4.1 Syntax

We now introduce an abstract syntax for SPARQL 1.1 queries, which is defined recursively, and which extends the abstract syntax for SPARQL 1.0 defined in Section 2.3.1. We present the following preliminaries, followed by the definitions in Table 4.1.

- A *triple pattern* $t = (s, p, o)$ is a member of the set $\mathbf{VIBL} \times \mathbf{VI} \times \mathbf{VIBL}$.
- A *basic graph pattern* $B = \{t_1, \ldots, t_n\}$ is a set of triple patterns.
- A *path pattern* $p = (s, e, o)$ is a member of the set $\mathbf{VIBL} \times \mathbf{P} \times \mathbf{VIBL}$ where $\mathbf{P}$ is the set of *property paths* defined in Table 4.2.

- A *navigational graph pattern* $N = \{p_1, \ldots, p_n\}$ is a set of path patterns and triple patterns with variables as predicates.
- We denote by $\text{vars}(B)$ the set of variables used in each triple pattern in B. Similarly, we denote by $\text{vars}(N)$ the set of variables used in each path pattern and triple pattern in N. Additionally, we denote by $\text{paths}(N)$ the set of property paths in N.
- A term in **VIBL** is also known as a *built-in expression* R. Let $\mathbf{R}$ denote the set of all built-in expressions, ε denote an *error* and $\perp$ denote an *unbound value*. A function ϕ is a *built-in function* that receives a tuple of built-in expressions as input, and, when evaluated, outputs a built-in expression, an error message (ε), or an unbound value ($\perp$). Then $\phi(R_1, \ldots, R_n)$ is also a built-in expression. The set of variables used in R is denoted by $\text{vars}(R)$. For example, let R be the expression `isBlank(?v)`, where `isBlank()` is a built-in function that returns *true* if the built-in expression (`?v` in this case) is bound to a blank node.
- A function $\psi : ((\mathbf{R})) \to \mathbf{R} \cup \{\varepsilon, \perp\}$ is an *aggregation function* that takes bags of tuples of built-in expressions as input and outputs a built-in expression, an error message (ε), or an unbound value ($\perp$). Then $\psi(R_1, \ldots, R_n)$ is an *aggregation expression*. For example, an aggregate expression `AVG(?v1)` will return the average value of the bag of results bound to `?v1` (assuming they are numeric).

4.2 Semantics

In this section, we describe the evaluation of SPARQL queries over RDF graphs under set semantics and bag semantics, where the output is a set of solution mappings, and bag (or multiset) of solution mappings, respectively.

4.2.1 Set Semantics

The semantics of a SPARQL query Q is defined in terms of its evaluation over a SPARQL dataset, which typically first involves transforming the dataset into *solution mappings*. We first describe the set semantics of SPARQL 1.1, which works with sets of solutions, and builds upon the definitions for SPARQL 1.0 described in Section 2.3.2.

Let M, M_1 and M_2 denote sets of mappings. We define an algebra consisting of the natural-join ($\bowtie$), union ($\cup$), semi-join ($\ltimes$), anti-join ($\rhd$), left-outer-join ($⟕$), minus ($-$), projection (π), selection (σ), and bind (β) operators, respectively, as follows:

Table 4.1: Abstract SPARQL syntax

- B is a basic graph pattern.	∴ B is a query pattern on vars(B).
- N is a navigational graph pattern.	∴ N is a query pattern on vars(N).
- Q_1 is a query pattern on V_1. - Q_2 is a query pattern on V_2.	∴ $[Q_1 \text{ AND } Q_2]$ is a query pattern on $V_1 \cup V_2$; ∴ $[Q_1 \text{ UNION } Q_2]$ is a query pattern on $V_1 \cup V_2$; ∴ $[Q_1 \text{ OPT } Q_2]$ is a query pattern on $V_1 \cup V_2$; ∴ $[Q_1 \text{ MINUS } Q_2]$ is a query pattern on V_1.
- Q is a query pattern on V. - Q_1 is a query pattern on V_1. - Q_2 is a query pattern on V_2. - v is a variable not in V. - R is a built-in expression.	∴ $\text{FILTER}_R(Q)$ is a query pattern on V; ∴ $[Q_1 \text{ FE } Q_2]$ is a query pattern on V_1; ∴ $[Q_1 \text{ FNE } Q_2]$ is a query pattern on V_1; ∴ $\text{BIND}_{R,v}(Q)$ is a query pattern on $V \cup \{v\}$.
- Q is a query pattern on V. - $\mathfrak{M}$ is a bag of solution mappings on $V_{\mathfrak{M}} = \bigcup_{\mu \in \mathfrak{M}} \text{dom}(\mu)$.	∴ $\text{VALUES}_{\mathfrak{M}}(Q)$ is a query pattern on $V \cup V_{\mathfrak{M}}$.
- Q is a graph pattern on V. - x is an IRI. - v is a variable.	∴ $\text{GRAPH}_x(Q)$ is a graph pattern on V. ∴ $\text{GRAPH}_v(Q)$ is a graph pattern on $V \cup \{v\}$.
- Q_1 is a graph pattern on V_1. - Q_2 is a graph pattern on V_2. - x is an IRI. - Δ is a boolean value.	∴ $[Q_1 \text{ SERVICE}_x^\Delta Q_2]$ is a graph pattern on $V_1 \cup V_2$.
- Q is a graph pattern on V. - Q' is a group-by pattern on (V', V). - V'' is a set of variables. - A is an aggregation expression. - Λ is a (possibly empty) set of pairs $\{(A_1, v_1), \ldots, (A_n, v_n)\}$, where $A_1, \ldots, A_n$ are aggregation expressions, $v_1, \ldots, v_n$ are variables not appearing in $V \cup V'$ such that $v_i \neq v_j$ for $1 \leq i < j \leq n$, and where $\text{vars}(\Lambda) = \{v_1, \ldots, v_n\}$.	∴ Q is a group-by pattern on $(\emptyset, V)$. ∴ Q' is a graph pattern on V'. ∴ $\text{GROUP}_{V''}(Q)$ is a group-by pattern on (V'', V). ∴ $\text{HAVING}_A(Q')$ is a group-by pattern on (V', V). ∴ $\text{AGG}_\Lambda(Q')$ is a graph pattern on $(V' \cup \text{vars}(\Lambda), V)$.
- Q is a query but not a from query. - X and X' are sets of IRIs.	∴ $\text{FROM}_{X,X'}(Q)$ is a from query and a query.

Table 4.2: SPARQL property path syntax

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

$$
\begin{aligned}
M_1 \bowtie M_2 &:= \{\ \mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\} \\
M_1 \cup M_2 &:= \{\ \mu \mid \mu \in M_1 \text{ or } \mu \in M_2\} \\
M_1 \ltimes M_2 &:= \{\ \mu_1 \in M_1 \mid \exists \mu_2 \in M_2 : \mu_1 \sim \mu_2\} \\
M_1 \rhd M_2 &:= \{\ \mu_1 \in M_1 \mid \nexists \mu_2 \in M_2 : \mu_1 \sim \mu_2\} \\
M_1 ⟕ M_2 &:= (M_1 \bowtie M_2) \cup (M_1 \rhd M_2) \\
M_1 - M_2 &:= \{\ \mu_1 \in M_1 \mid \nexists \mu_2 \in M_2 : \mu_1 \sim \mu_2 \text{ and } \mu_1 * \mu_2\} \\
\pi_V(M) &:= \{\ \mu' \mid \exists \mu \in M : \mu' \subseteq \mu \text{ and } \mathrm{dom}(\mu') = V \cap \mathrm{dom}(\mu)\} \\
\sigma_R(M) &:= \{\ \mu \in M \mid \mu \models R\} \\
\beta_{R,v}(M) &:= \{\ \mu \cup \langle v/\mu(R)\rangle \mid \mu \in M, v \notin \mathrm{dom}(\mu) \text{ and } \mu(R) \neq \varepsilon\}
\end{aligned}
$$

Letting Q denote a 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 $\mathrm{vars}(t)$ and $\mathrm{bnodes}(t)$ we denote the set of variables and blank nodes appearing in t, respectively. Then, we denote by $\mu(t)$ the image of t under a solution μ, and in a similar fashion, we define $\alpha(t)$ as the image of t under a blank node mapping α; the difference between these two lies in that although blank nodes may act as variables, they cannot generate bindings.

On the other hand, in order to define the evaluation of a navigation graph pattern N, we must first define the evaluation of property paths; a property path evaluated on an RDF graph returns sets of pairs of elements connected by sequences of predicates that match the regular expression defined by the property path. We denote the set of terms appearing as subjects or predicates in an RDF graph G by $\mathrm{so}(G)$; more specifically $\mathrm{so}(G) := \{x \mid \exists p, y : (x,p,y) \in G$ or $(y,p,x) \in G\}$. Then, the evaluation of a property path e over G (denoted $e(G)$) is defined as follows:

$$
\begin{aligned}
p(G) &:= \{(s,o) \mid (s,p,o) \in G\} \\
!p(G) &:= \{(s,o) \mid \exists q : (s,q,o) \in G \text{ and } q \neq p\} \\
\hat{}e(G) &:= \{(s,o) \mid (o,s) \in e(G)\} \\
e_1/e_2(G) &:= \{(x,z) \mid \exists y : (x,y) \in e_1(G) \text{ and } (y,z) \in e_2(G)\} \\
e_1|e_2(G) &:= e_1(G) \cup e_2(G) \\
e+(G) &:= \{(y_1, y_{n+1}) \mid \text{ for } 1 \leq i \leq n : \exists (y_i, y_{i+1}) \in e(G)\} \\
e*(G) &:= e+(G) \cup \mathrm{so}(G) \\
e?(G) &:= e(G) \cup \mathrm{so}(G) \\
!(p_1|\ldots|p_k|\hat{}p_{k+1}|\ldots|\hat{}p_n)(G) &:= !(p_1|\ldots|p_k)(G) \cup !(\hat{}p_{k+1}|\ldots|\hat{}p_n)(G)
\end{aligned}
$$

Thus, given a navigational graph pattern N and RDF graph G, we define the *path graph* G^N as the set $\{(s,e,o) \mid e \in paths(N) \text{ and } (s,o) \in e(G)\}$.

Finally, we can define $Q(G)$ recursively as shown in Table 4.3:

Table 4.3: Set evaluation of graph patterns where G is an RDF graph; B is a basic graph pattern; N is a navigational graph pattern; Q, Q_1 and Q_2 are graph patterns; V is a set of variables; R is a built-in expression; v is a variable; M is a set of solution mappings; and x is an IRI

$B(G)$	$:=$	$\{\mu \mid \exists\alpha : \mu(\alpha(B)) \subseteq G_D \text{ and } \mathrm{dom}(\mu) = \mathrm{vars}(B) \text{ and } \mathrm{dom}(\alpha) = \mathrm{bnodes}(B)\}$
$N(G)$	$:=$	$\{\mu \mid \exists\alpha : \mu(\alpha(N)) \subseteq G_D^N \cup G_D \text{ and } \mathrm{dom}(\mu) = \mathrm{vars}(N) \text{ and } \mathrm{dom}(\alpha) = \mathrm{bnodes}(N)\}$
$[Q_1 \text{ AND } Q_2](G)$	$:=$	$Q_1(G) \bowtie Q_2(G)$
$[Q_1 \text{ UNION } Q_2](G)$	$:=$	$Q_1(G) \cup Q_2(G)$
$[Q_1 \text{ MINUS } Q_2](G)$	$:=$	$Q_1(G) - Q_2(G)$
$[Q_1 \text{ OPT } Q_2](G)$	$:=$	$Q_1(G) ⟕ Q_2(G)$
$\text{SELECT}_V(Q)(G)$	$:=$	$\pi_V(Q(G))$
$\text{FILTER}_R(Q)(G)$	$:=$	$\sigma_R(Q(G))$

4.2.2 Bag Semantics

The evaluation of SPARQL queries under bag semantics is similar to the evaluation under set semantics, but operates over and returns bags of solution mappings, instead of sets of solution mappings. This means that the multiplicities of the solution mappings is considered as part of the evaluation. Therefore, we must define the algebra of bags of solution mappings analogous to the definition presented in Section 4.2.1.

Let $\mathfrak{M}$, $\mathfrak{M}_1$ and $\mathfrak{M}_2$ denote bags of solution mappings. We define an algebra consisting of the join, union, semi-join, anti-join, difference, left-outer-join, selection and projection operators, respectively, as follows: note that we use Iverson bracket notation where $[\,\phi\,] = 1$ if and only if ϕ is *true*; otherwise, if ϕ is false or undefined, then $[\,\phi\,] = 0$.

$$
\begin{aligned}
\mathfrak{M}_1 \bowtie \mathfrak{M}_2(\mu) &:= \sum_{\mu_1,\mu_2} \mathfrak{M}_1(\mu_1) \cdot \mathfrak{M}_2(\mu_2) \cdot [\,\mu_1 \cup \mu_2 = \mu\,] \\
\mathfrak{M}_1 \cup \mathfrak{M}_2(\mu) &:= \mathfrak{M}_1(\mu) + \mathfrak{M}_2(\mu) \\
\mathfrak{M}_1 \ltimes \mathfrak{M}_2(\mu) &:= \mathfrak{M}_1(\mu) \cdot [\,\text{there exists } \mu' \in \mathfrak{M}_2 \text{ such that } \mu \sim \mu'\,] \\
\mathfrak{M}_1 \rhd \mathfrak{M}_2(\mu) &:= \mathfrak{M}_1(\mu) \cdot [\,\text{there does not exist } \mu' \in \mathfrak{M}_2 \text{ such that } \mu \sim \mu'\,] \\
\mathfrak{M}_1 - \mathfrak{M}_2(\mu) &:= \mathfrak{M}_1(\mu) \cdot [\,\text{there does not exist } \mu' \in \mathfrak{M}_2 \text{ such that } \mu \sim \mu' \text{ and } \mu * \mu'\,] \\
\mathfrak{M}_1 ⟕ \mathfrak{M}_2(\mu) &:= ((\mathfrak{M}_1 \bowtie \mathfrak{M}_2) \cup (\mathfrak{M}_1 \rhd \mathfrak{M}_2))(\mu) \\
\pi_V(\mathfrak{M})(\mu) &:= \sum_{\mu'} \mathfrak{M}(\mu') \cdot [\,\mathrm{dom}(\mu') = V \cap \mathrm{dom}(\mu) \text{ and } \mu \sim \mu'\,] \\
\sigma_R(\mathfrak{M})(\mu) &:= \mathfrak{M}(\mu) \cdot [\,\mu \models R\,] \\
\beta_{R,v}(\mathfrak{M})(\mu) &:= \sum_{\mu'} \mathfrak{M}(\mu') \cdot [\,\mu' \cup \{v/R(\mu')\} = \mu\,]
\end{aligned}
$$

The bag evaluation of SPARQL queries remains largely the same, other than the evaluation of basic graph patterns and navigational patterns:

Table 4.4: Bag evaluation of graph patterns where G is an RDF graph; B is a basic graph pattern; N is a navigational graph pattern.

$B(G)(\mu)$	$:=$	$\lvert\{\alpha \mid \mu(\alpha(B)) \subseteq G \text{ and } \mathrm{dom}(\alpha) = \mathrm{bnodes}(B)\}\rvert \cdot [\mathrm{dom}(\mu) = \mathrm{vars}(B)]$
$N(G)(\mu)$	$:=$	$\lvert\{\alpha \mid \mu(\alpha(N)) \subseteq G^N \text{ and } \mathrm{dom}(\alpha) = \mathrm{bnodes}(N)\}\rvert \cdot [\mathrm{dom}(\mu) = \mathrm{vars}(N)]$

However, some property paths may be rewritten into equivalent (unions of) basic graph patterns, which may affect the multiplicities of the results. This is because the evaluation of property paths is done under set semantics, while the evaluation of operators into which the property paths are rewritten are done under bag semantics. Table 4.5 indicates the evaluation of navigational graph patterns..

Table 4.5: Bag evaluation of navigational patterns where G is an RDF graph, N is a navigational pattern, and x is a fresh blank node

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

4.3 Default and Named Graphs

Thus far we've defined the semantics of SPARQL 1.1 queries over a single RDF graph. However, SPARQL allows us to operate over several RDF graphs by defining a *SPARQL dataset*. Formally, a *SPARQL dataset* $D := (G, \{(x_1, G_1), \ldots, (x_n, G_n)\})$ is a pair of an RDF graph G called the *default graph*, and a set of *named graphs* of the form (x_i, G_i), where x_i is a unique IRI (called a *graph name*) and G_i is an RDF graph. We denote by G_D the default graph G of D and by $D^* = \{(x_1, G_1), \ldots, (x_n, G_n)\}$ the set of all named graphs in D. We further denote by $G_{D[x_i]}$ the graph G_i such that $(x_i, G_i) \in D^*$ or the empty graph if x_i does not appear as a graph name in D^*.

In concrete SPARQL syntax, the `FROM` and `FROM NAMED` clauses establish the default graph and the set of named graphs, respectively. These clauses are followed by IRIs that denote RDF datasets. Figure 4.1 provides an example of the use of the `FROM` clause in concrete SPARQL syntax, where the basic graph pattern in the `WHERE` clause will now only generate answers from the RDF graph named `http://example.org/a`.

```
SELECT ?actor
FROM <http://example.org/a>
WHERE{
    ?actor :actsIn ?movie;
        :salary ?salary .
}
```

Figure 4.1: An example of a query that contains a `FROM` clause.

Queries are evaluated on a SPARQL dataset D, where dataset modifiers allow for changing the dataset considered for query evaluation. First, let X and X' denote (possibly empty or overlapping) sets of IRIs and let D denote a SPARQL dataset. We then denote by $D(X, X') := (\biguplus_{x \in X} G_{D[x]}, \{(x', G_{D[x']}) \mid x' \in X'\})$ a new dataset formed from D by merging all named graphs of D named by X to form the default graph of $D(X, X')$, and by selecting all named graphs of D named by X' as the named graphs of $D(X, X')$. We define the semantics of dataset modifiers in Table 4.6.

Table 4.6: Evaluation of dataset modifiers where X and X' are sets of IRIs

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

Once a set of named graphs has been defined, they can be accessed with the `GRAPH` clause. A `GRAPH` clause is followed immediately by either an IRI that points to a specific external

graph or a variable that maps to each of the named graphs. Finally, it is followed by a sub-query that is executed over each named graph. Figure 4.2 presents a query that defines three named graphs, and then defines a sub-query that looks for actors in each of the named graphs.

```
SELECT ?actor
FROM NAMED <http://example.org/a>
FROM NAMED <http://example.org/b>
FROM NAMED <http://example.org/c>
WHERE{
    GRAPH ?g
    { ?actor :actsIn ?movie;
        :salary ?salary . }
}
```

Figure 4.2: An example of a query that contains `FROM NAMED` and `GRAPH` clauses.

We now define two operators that allow for generating query-specific datasets:

$$\text{GRAPH}_x(Q)(D) := Q(\langle G_{D[x]}, D_N \rangle)$$
$$\text{GRAPH}_v(Q)(D) := \bigcup_{(x_i, G_i) \in D_N} \beta_{x_i,v}(Q(\langle G_i, D_N \rangle))$$

With respect to the evaluation of the previously defined query features, these were defined on RDF graphs rather than RDF datasets. In the case of RDF datasets, such features are simply evaluated on the default graph of the dataset, unless a different active graph is selected via the use of `FROM` or `GRAPH`, in which case they are evaluated on the active graph.

4.4 Assignments to Variables

SPARQL allows us to assign specific values to variables that may or may not depend on the evaluation of the query. This can be achieved by the use of the `BIND` or the `VALUES` keywords.

The `BIND` keyword allows us to assign built-in expressions to variables. In the example presented in Figure 4.3 we assign the difference between the actors' date of birth and the release date of the movies they acted in in order to determine the actors' age at the release of the movie.

```
SELECT ?actor ?movie ?age
WHERE{
    ?actor :actsIn ?movie .
    ?actor :dateOfBirth ?dob .
    ?movie :releaseDate ?date .
    BIND ((?date - ?dob) AS ?age)
}
```

Figure 4.3: Example of a query with a `BIND` expression.

Let Q be a query, G a dataset, and R a built-in expression that is being assigned to a variable v. We define the semantics of `BIND` as follows:

$$\text{BIND}_{R,v}(Q)(G) := \beta_{R,v}(Q(G))$$

On the other hand, SPARQL allows us to explicitly define in-line values by use of the `VALUES` keyword. In the example presented in Figure 4.4 we are looking for actors who have acted in horror, science fiction, and fantasy movies. While this could be accomplished similarly by a `UNION` and explicitly declaring each value in a triple pattern, this is sometimes used for optimisations such as in caching systems and federated query systems [9].

```
SELECT ?actor
WHERE {
    ?actor :actsIn ?movie .
    ?movie :genre ?genre .
    VALUES ?genre { :Horror :ScienceFiction :Fantasy }
}
```

Figure 4.4: Example of a query with inline values.

The `VALUES` keyword allows us to explicitly define a set of solution mappings M in a SPARQL query, that may be visualised as a table of results where each row is a solution mapping μ, and each mapping contains a set of variables and the values assigned to said variable. These values are either literal values, IRIs or a special value `UNDEF` that denotes an unbound value.

Let Q denote a query pattern that is evaluated in a dataset G. The set of solution mappings M defined as part of the `VALUES` clause is then joined with the evaluation of Q over G as follows:

$$\text{VALUES}_M(Q)(G) := Q(G) \bowtie M$$

4.5 Aggregation

Aggregations expressions allow us to perform operations on bags of results instead of individual results. Let $(\mu, \mathfrak{M})$ denote a *solution group*, where μ is a solution mapping called the *key of the solution group*, and $\mathfrak{M}$ is a bag of solution mappings called the *bag of the solution group* such that each solution in the bag of the solution group $\mathfrak{M}$ is compatible with the key of the solution mapping μ. Group-by patterns then return a set of solution groups $\mathcal{M} := \{(\mu_1, \mathfrak{M}_1), \ldots, (\mu_n, \mathfrak{M}_n)\}$ as their output. In concrete SPARQL syntax, we use the `GROUP BY` expression to group sets of solution mappings based on partial results.

In the example presented in Figure 4.5 we look for actors and the box office of all movies they acted in. We group these by the actors while we compute the average box office for each

```
SELECT ?actor (AVG(?value) AS ?net)
WHERE{
    ?actor :actsIn ?movie .
    ?movie :boxOffice ?value .
} GROUP BY ?actor
```

Figure 4.5: Example of a SPARQL query with aggregation.

of them. In this query, `?actor` defines the key of the solution group, and thus exists outside the scope of the grouping. On the other hand, variables such as `?value` may not be used outside the grouping unless it has been used inside an *aggregate function* such as *average* (`AVG`). Internally, this is the same as binding an expression (in this instance `AVG(?value)`) to an anonymous variable. We now define the semantics of group-by patterns along with the AGG graph pattern, which allows for converting a set of solution groups to a set of solution mappings.

Table 4.7: Aggregation algebra under bag semantics, where $\mathfrak{M}$ and $\mathfrak{M}'$ are bags of solution mappings; V is a set of variables and v is a variable; A is an aggregation expression; and $\mathcal{M}$ is a set of solution groups; we recall that $\mathbf{M}$ is the set of all solution mappings

$$\begin{aligned}
\gamma_V(\mathfrak{M}) &:= \{(\mu, \mathfrak{M}') \mid \mu \in \pi_V(\mathfrak{M}) \text{ and } \forall \mu' \in \mathbf{M} : \mathfrak{M}'(\mu') = \mathfrak{M}(\mu') \cdot [\,\pi_V(\{\mu'\}) = \{\mu\}\,]\} \\
\sigma'_A(\mathcal{M}) &:= \{(\mu, \mathfrak{M}) \in \mathcal{M} \mid \mathfrak{M} \models A\} \\
\beta'_{A,v}(\mathcal{M}) &:= \{\ (\mu \cup \{v/A(\mathfrak{M})\}, \beta_{A(\mathfrak{M}),v}(\mathfrak{M})) \mid (\mu, \mathfrak{M}) \in \mathcal{M}\} \\
\zeta(\mathcal{M}) &:= \{\mu \mid (\mu, \mathfrak{M}) \in \mathcal{M}\}
\end{aligned}$$

In Table 4.7, we present an algebra for aggregation, which includes the grouping operator described earlier for inducing a set of solution groups from a set or bag of solution (e.g., grouping by `?actor`), a selection operator for filtering solution groups that do not satisfy a given aggregate condition (e.g., filter groups whose maximum `?salary` is over a certain value), a bind operator that stores the result of an aggregation operator to a fresh variable (e.g., assign `MAX(?salary)` to a variable `?max`), and finally a flatten operator that extracts the key of each solution group into a "flat" set of solution mappings (e.g., extract solution mappings for `?max`). Table 4.8 then shows the evaluation of queries that contain these features using the algebra presented in Table 4.7.

Table 4.8: Evaluation of group-by patterns where D is a dataset, Q is a graph pattern or group-by pattern, V is a set of variables, $v_1, \ldots, v_n$ are variables, and $A, A_1, \ldots, A_n$ are aggregation expressions

$$\begin{aligned}
\textsc{group}_V(Q)(D) &:= \gamma_V(Q(D)) \\
\textsc{having}_A(Q)(D) &:= \sigma'_A(Q(D)) \\
\textsc{agg}_{\{(A_1,v_1),\ldots,(A_n,v_n)\}}(Q)(D) &:= \zeta(\beta'_{A_1,v_1}(\ldots(\beta'_{A_n,v_n}(Q(D)))\ldots))
\end{aligned}$$

note: $\textsc{agg}_{\{\}}(Q)(D) := \zeta(Q(D))$

4.6 Negation

Although negation could be accomplished in SPARQL 1.0 using a combination of optional graph patterns and filter expressions, it was only natural to include explicit expressions for negation. In fact, SPARQL 1.1 introduced two distinct forms of negation: `MINUS` and `FILTER (NOT) EXISTS`.

```
SELECT ?actor ?salary
WHERE{
    ?actor :salary ?salary .
    MINUS
    { ?movie :genre :Horror . }
}
```

```
SELECT ?actor ?salary
WHERE{
    ?actor :salary ?salary .
    FILTER NOT EXISTS
    { ?movie :genre :Horror }
}
```

Figure 4.6: Example of queries featuring negation using `MINUS` (left) and `FILTER NOT EXISTS` (right).

Figure 4.6 contains two queries that look for actors who have not acted in horror movies, and their salaries. Unfortunately the user forgot to add a triple pattern to link actors to movies, but this helps us to illustrate the difference between `MINUS` and `FILTER NOT EXISTS`. We note that despite the fact that both express negation, both expressions are not equivalent. In fact, the `MINUS` expression will have no effect on the results of the query on the left because there is no overlap between the solutions for `?actor` and `?salary`, and `?movie`. On the other hand, the `FILTER NOT EXISTS` expression will return every result for `?actor` and `?salary`, unless the graph contains a triple describing a movie of the horror genre, in which case the results are empty. This has to do with the fact that `FILTER NOT EXISTS` requires mappings that are compatible, and removes results if the mapping is not empty, whereas `MINUS` further requires overlapping mappings on the right in order to remove mappings from the left (if no solution on the right has a variable in common with a given solution on the left, the solution on the left is kept). We now define these negation features for the abstract syntax considering set semantics:

$$
\begin{aligned}
[Q_1 \,\text{FE}\, Q_2](D) &:= \{\mu \in Q_1(D) \mid (\mu(Q_2))(D) \neq \emptyset\} \\
[Q_1 \,\text{FNE}\, Q_2](D) &:= \{\mu \in Q_1(D) \mid (\mu(Q_2))(D) = \emptyset\} \\
[Q_1 \,\text{MINUS}\, Q_2](D) &:= Q_1(D) - Q_2(D)
\end{aligned}
$$

While the multiplicity of $\mathfrak{M}_1 - \mathfrak{M}_2$ was defined in Section 4.2.2, we are left to define the multiplicity for the `FILTER (NOT) EXISTS` cases:

$$
\begin{aligned}
[Q_1 \,\text{FE}\, Q_2](D)(\mu) &:= Q_1(D)(\mu) \cdot [(\mu(Q_2))(D) \neq \emptyset] \\
[Q_1 \,\text{FNE}\, Q_2](D)(\mu) &:= Q_1(D)(\mu) \cdot [(\mu(Q_2))(D) = \emptyset]
\end{aligned}
$$

The observant reader may note that $\mu(Q_2)$ in the previous definitions for `FILTER (NOT) EXISTS` is not well-defined for all cases. For example, if Q_2 contains a `BIND` clause to a variable

$v \in \text{dom}(\mu)$, then replacing v with $\mu(v)$ in Q_2 renders $\mu(Q_2)$ syntactically invalid. This is a known issue with SPARQL 1.1 that would require inventing new semantics to address, and for which there is no natural solution. We refer to [64, Section 3.10] for more discussion.

4.7 Federated Queries

Federated queries allow us to run queries on remote SPARQL endpoints, and join the results of that (federated) query with the results of our local query. For example, the query in Figure 4.7 finds all actors who have acted in a horror movie in our default graph. Then, assuming there exists a remote SPARQL endpoint at `http://some-sparql-endpoint.com` that contains information on taxation, the query looks for information on taxes paid by each actor in the year 2019. The results from the federated query are joined with the results from the local query.

```
SELECT ?actor ?value
WHERE {
    ?actor :actsIn ?movie .
    ?movie :genre :Horror .
    SERVICE <http://some-sparql-endpoint.com>
    { ?actor :taxed ?taxation .
      ?taxation :year 2019^^xsd:year .
      ?taxation :value ?value . }
}
```

Figure 4.7: An example of a SPARQL query that includes a federated query.

In order to define this feature, we denote by ω a *federation mapping* from IRIs to services such that, given an IRI $x \in \mathbf{I}$, then $\omega(x)$ returns the service $\mathcal{S}$ hosted at x or returns ε in the case that no service exists or can be retrieved. We denote by $\mathcal{S}.Q(D_{\mathcal{S}})$ the evaluation of a query Q on a remote service $\mathcal{S}$. When a service of the query evaluation is not indicated (e.g., $Q(D)$), we assume that it is evaluated on the local service. Finally, we define that $\varepsilon.Q(D_\varepsilon)$ invokes a query-level error ε – i.e., the evaluation of the entire query fails – while $\varepsilon.Q^*(D_\varepsilon)$ returns a set with the empty solution mapping $\{\mu_\emptyset\}$.[1]

We can then define the evaluation of federated queries in abstract syntax as follows, where the first definition throws an error if the remove SPARQL query service fails, while the latter returns the results for the local query (Q_1) if the service fails. The latter then corresponds to the use of the `SILENT` modifier in the SPARQL 1.1 concrete syntax.

$$[Q_1 \,\text{SERVICE}_x^{\texttt{false}}\, Q_2](D) := Q_1(D) \bowtie \omega(x).Q_2(D_{\omega(x)})$$
$$[Q_1 \,\text{SERVICE}_x^{\texttt{true}}\, Q_2](D) := Q_1(D) \bowtie \omega(x).Q_2^*(D_{\omega(x)})$$

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

4.8 Query Containment and Equivalence

Query containment and equivalence are problems that have been well studied in the literature. Some notable results for the purposes of this study are presented in Table 4.9. Here, we present the complexities under both set and bag semantics for the following classes of SPARQL queries: *conjunctive queries* (CQs) and *conjunctive path queries* (CPQs) are basic graph patterns and navigational graph patterns, respectively; *unions of conjunctive queries* (UCQs) and *unions of conjunctive path queries* (UCPQs) are unions of basic graph patterns and navigational graph patterns, respectively; *monotone queries* (MQs) and *monotone path queries* (MPQs) are queries that contain joins and unions of basic graph patterns and navigational graph patterns, respectively[2]; *non-monotone queries* (NMQs) and *non-monotone path queries* (NMPQs) are queries that contain joins and unions, as well as negation of basic graph patterns and navigational graph patterns, respectively. In particular, both C(P)Qs and UC(P)Qs are monotone (path) queries, and thus subsets of M(P)Qs.

Table 4.9: Complexity of SPARQL tasks on core fragments (considering combined complexity for EVALUATION). We annotate with * those results that are not stated but follow directly from results presented in the respective paper. Note that -C and -H stand for -COMPLETE and -HARD, respectively.

	Set semantics			
	EVALUATION	CONTAINMENT	EQUIVALENCE	CONGRUENCE
CQ	NP-C [56]	NP-C [18]*	NP-C [18]*	NP-C
UCQ	NP-C [56]	NP-C [18]*	NP-C [18]*	NP-C
MQ	NP-C	Π_2^P-C [62]*	Π_2^P-C [62]*	Π_2^P-C
NMQ	PSPACE-C [56]	Undecidable [74]*	Undecidable [74]*	Undecidable
CPQ	NP-C [44]	EXPSPACE-C [44]	NP-H, EXPSPACE	NP-H, EXPSPACE
UCPQ	NP-C [44]	EXPSPACE-C [44]	NP-H, EXPSPACE	NP-H, EXPSPACE
MPQ	NP-H [44]*	EXPSPACE-H [44]*	Π_2^P-H	Π_2^P-H
NMPQ	PSPACE-H	Undecidable	Undecidable	Undecidable
Full	PSPACE-H	Undecidable	Undecidable	Undecidable
	Bag semantics			
	EVALUATION	CONTAINMENT	EQUIVALENCE	CONGRUENCE
CQ	NP-C	NP-H, *Decidability open*	GI-C [19]*	GI-C
UCQ	NP-C	Undecidable [40]*	GI-C [23]*	GI-C
MQ	NP-C	Undecidable	GI-H	GI-H
NMQ	PSPACE-C	Undecidable	Undecidable [57]*	Undecidable
Full	PSPACE-H	Undecidable	Undecidable	Undecidable

The results presented in Table 4.9 show that the canonicalisation is a hard problem because it may solve the congruence decision problem, which is NP-complete even for conjunctive queries. On the other hand, we see that the problem is decidable even for MPQs, even though it is EXPSPACE-complete. Further studies reveal that this complexity may be

[2]Note that unlike UCQs and UCPQs, they permit joins of unions, and can thus express certain queries exponentially more concisely.

lower for MPQs under certain restrictions [31]. Intuitively this is because, under set semantics, we must additionally check for redundancies with (U)CQs , while under bag semantics, such “redundancies” can no longer be removed without affecting equivalence as this would change the multiplicity of results. Thus, under bag semantics, it suffices to check that the queries are “isomorphic”, which will be discussed in Chapter 5.

A notable result is that the complexity of the congruence decision problem for (U)CQs under bag semantics is GI-complete, meaning that it is as hard as the graph isomorphism decision problem, but may in fact be easier than the analogous problems under set semantics (if GI $\neq$ NP).

Chapter 5

Canonicalisation of SPARQL queries

The general objective of this thesis is to research methods for canonicalising SPARQL queries and to design and implement a practical algorithm that takes as input a large list of SPARQL queries and can identify subsets of equivalent (or congruent) queries.

The proposed algorithm aims to accomplish the following:

- Support the majority of real-world SPARQL queries.
- Capture more equivalences than baseline, syntactic methods.
- Be executed efficiently and at the scale of millions of queries.

In this chapter, we will first describe our method for the fragment of SPARQL known as monotone queries, for which the canonicalisation process is *sound* and *complete*. Following this, we will describe the method for the full language, where the process is sound but *incomplete*. In particular, we will describe the partial canonicalisation we have achieved in conjunctive queries with property paths.

5.1 Monotone Queries

In this section, we describe in detail the different steps that comprise the canonicalisation method for monotone queries. 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. Thereby we assign new names for each variable in a manner that is canonical modulo isomorphism.

We now describe each of the steps in greater detail.

5.1.1 Normalisation

In this section, we describe the normalisation process of our method. This step encompasses several transformations and removal of redundancies. Most of these transformations are results of equivalences that have been well studied in the literature [34, 67].

Algorithm 1 Pseudo-algorithm for the normalisation of monotone SPARQL queries.

Input: A monotone SPARQL query Q
Output: A normalised query $Q' = \text{U}(Q)$ such that $Q' \equiv Q$
  function U(Q)
    $Q \leftarrow$ UNIONNORMALISATION(Q)
    $Q \leftarrow$ VARIABLENORMALISATION(Q)
    $Q \leftarrow$ UNSATISFIABILITYNORMALISATION(Q)
    $Q \leftarrow$ SETVSBAGNORMALISATION(Q)
    return Q
  end function

Algorithm 1 presents pseudo-code describing the steps that comprise the normalisation process. To summarise, we first begin by describing the rewriting of monotone parts of queries into UCQs. Subsequently, we describe cases where, under certain circumstances, we may rewrite monotone queries that cannot produce duplicated bindings by adding the `DISTINCT` keyword (if missing). Following this, we describe the cases in which certain variables are renamed in order to differentiate variables with the same name but that share no semantic value (denoted as *non-correlated variables*). Next, we describe the cases in which certain variables may be removed altogether without affecting the semantic value of the query. Finally, we describe cases where certain CQs and UCQs may never produce results (are *unsatisfiable*), and may be rewritten into a standard, canonical *unsatisfiable* CQ.

UCQ Normalisation

In this section, we describe the rules for rewriting monotone queries into UCQs. One of the first results we consider is that joins of triple patterns can be rewritten to equivalent BGPs. That is, a query pattern $[t \text{ AND } t']$ where t and t' are triple patterns can be rewritten to a BGP $B = \{t, t'\}$. In addition, in order to avoid potential issues with identical blank nodes matching different terms, blank nodes are rewritten into fresh variables that do not appear elsewhere in the query. Let $\eta : \mathbf{B} \rightarrow \mathbf{V}$ denote a one-to-one mapping of blank nodes to variables, and let $\eta(B)$ denote the basic graph pattern $\{\eta(t), \eta(t')\}$. This leads to the following equivalence (under both set and bag semantics):

$$\eta(B) \equiv \text{SELECT}_{\text{vars}(B)}([\eta(t) \text{ AND } \eta(t')])$$

Table 5.1: Equivalences given by Pérez et al. [56] for set semantics

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

i.e., a BGP B is equivalent to the result of rewriting its blank nodes to fresh variables and joining the individual triple patterns, finally projecting only the variables originally in B.

Thereafter, Pérez et al. [56] prove a number of equivalences involving joins and unions, as shown in Table 5.1. These known results give rise to a UCQ normal form for monotone queries. More specifically, given a pattern $[Q_1 \text{ AND } [Q_2 \text{ UNION } Q_3]]$, we can rewrite this to $[[Q_1 \text{ AND } Q_2] \text{ UNION } [Q_1 \text{ AND } Q_3]]$; in other words, we translate unions of joins to (equivalent) patterns involving joins of unions. Furthermore, we can abstract the commutativity and associativity of both joins and unions by introducing two new operators that accept multiple query patterns as an argument:

$$\text{AND}(Q_1, \ldots, Q_n) \equiv [Q_1 \text{ AND } [\ldots \text{ AND } Q_n]]$$
$$\text{UNION}(Q_1, \ldots, Q_n) \equiv [Q_1 \text{ UNION } [\ldots \text{ UNION } Q_n]]$$

The UCQ normal form for monotone queries is then of the form $\text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))$ (optionally with a distinct operator), where each Q_i $(1 \leq i \leq n)$ is of the form $\text{AND}(Q_{i1}, \ldots, Q_{im})$.

Algorithm 2 shows pseudo-code for the UCQ normalisation of monotone SPARQL queries recursively. A downside of this process is that the UCQ normal form may be of exponential size on the number of patterns on the original monotone query; in the worst case, a pattern of the form $\text{AND}(\text{UNION}(t_{11}, \ldots, t_{1m}), \ldots, \text{UNION}(t_{n1}, \ldots, t_{nm}))$ would be rewritten to a union of m^n BGPs (for each combination of t_{ij} and t_{kl}), with each BGP containing n triple patterns. On the other hand, the advantage of this approach is that once we have the query in UCQ

Algorithm 2 Pseudo-algorithm for the UCQ normalisation of monotone queries.

Input: A monotone SPARQL query Q

Output: A UCQ Q' such that $Q' \equiv Q$

  function unionNormalisation(Q)
    if $Q = (s, p, o)$ **then** ▷ Q is a triple pattern.
      return Q
    else if $Q = \text{SELECT}_V(Q')$ **then**
      return $\text{SELECT}_V(\text{unionNormalisation}(Q'))$
    else if $Q = [Q_1 \text{ AND } Q_2]$ **then**
      $Q' \leftarrow \text{unionNormalisation}(Q_1)$
      $Q'' \leftarrow \text{unionNormalisation}(Q_2)$
      if $Q' = [Q'_1 \text{ UNION } Q'_2]$ and $Q'' = [Q'_3 \text{ UNION } Q'_4]$ **then**
        return $[[[Q'_1 \text{ AND } Q'_3] \text{ UNION } [Q'_2 \text{ AND } Q'_3]] \text{ UNION } [[Q'_1 \text{ AND } Q'_4] \text{ UNION } [Q'_2 \text{ AND } Q'_4]]]$
      else if $Q' = [Q'_1 \text{ UNION } Q'_2]$ **then**
        return $[[Q'_1 \text{ AND } Q''] \text{ UNION } [Q'_2 \text{ AND } Q'']]$
      else if $Q'' = [Q''_1 \text{ UNION } Q''_2]$ **then**
        return $[[Q''_1 \text{ AND } Q'] \text{ UNION } [Q''_2 \text{ AND } Q']]$
      else
        return $[Q' \text{ AND } Q'']$
      end if
    else if $Q = [Q_1 \text{ UNION } Q_2]$ **then**
      $Q' \leftarrow \text{unionNormalisation}(Q_1)$
      $Q'' \leftarrow \text{unionNormalisation}(Q_2)$
      return $[Q' \text{ UNION } Q'']$
    end if
  end function

normal form, we have a more regular structure for the query and can apply known techniques to minimise it (i.e., to remove redundancies from it).

Renaming Variables

According to the standard SPARQL definitions, unions of query patterns with differing sets of variables are allowed; this leads to more equivalence cases that must be considered.

On the other hand, the same variable may sometimes appear in multiple query scopes such that the variable (once projected away) does not need to be bound to the same term in a solution.

```
SELECT DISTINCT ?m
WHERE{
{ ?a :title ?m . ?a :actor ?p . ?d :directed ?a .}
UNION
{ ?a :title ?m . ?a ?r ?p . ?d :directed ?a .}
}
```

```
SELECT DISTINCT ?m
WHERE{
{ ?a :title ?m . ?a :actor ?p . ?d :directed ?a .}
UNION
{ ?b :title ?m . ?b ?r ?p . ?d :directed ?b .}
}
```

Figure 5.1: An example of two equivalent UCQs.

Evidently, the two queries in Figure 5.1 are equivalent since if we look at each sub-query individually, we obtain a list of all directors, and the movies they have directed and their titles. However, since in the right query, the variable which maps to the movies has a different label in each sub-query (`?a` in one sub-query, and `?b` in another), our current approach registers them as distinct queries. This is because, despite the fact that in the query on the left variable `?a` appears in both sub-queries, because `?a` is not projected outside the sub-query, it belongs to two different scopes.

The scenario presented leads to a *false correlation* between both instances of variable `?a`. In such a case, we say that those variables are not correlated. We identify the following two cases where this issue may arise:[1]

- We call a variable v *local* to a query pattern Q on V (see Table 4.1) if $v \in \text{vars}(Q)$ and $v \notin V$. A variable v *local* to Q is not correlated with other appearances of v outside of Q.
- We call a variable v a *union variable* if it appears in a union $Q = \text{UNION}(Q_1, \ldots, Q_n)$. A union variable $v \in \text{vars}(Q_i)$ does not correlate with other appearances of $v \in \text{vars}(Q_j)$ where $i \neq j$ unless v is correlated outside of Q.

As a consequence, our method may fail to identify congruence between queries that contain these non-correlated variables, and queries that contain the same structure, but where the variables are distinguished. To address this, we process these queries to differentiate

[1] We recall that SPARQL does not allow the same blank nodes to appear in multiple graph patterns; otherwise such blank nodes would need to be addressed in a similar manner.

all non-correlated variables such that only variables that are correlated may share the same name.

The solution we have devised for UCQs is to distinguish the variables in each CQ by labelling each non-projected variable (or local variable) depending on the CQ they belong to. This is formalised in Lemma 5.1.1:

Lemma 5.1.1. Let $Q = \text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))$ denote a UCQ. Let $\lambda_1, \ldots, \lambda_n$ denote variable-to-variable mappings such that for $1 \leq i \leq n$, $\lambda_i : \mathbf{V} \to \mathbf{V}$ where $\text{dom}(\lambda_i) = \text{vars}(Q_i) \setminus V$, and, for all $v \in \text{dom}(\lambda_i)$, it holds that $\lambda_i(v) \notin \text{vars}(Q)$ and there does not exist λ_j $(1 \leq i < j \leq n)$ and $v' \in \text{dom}(\lambda_j)$ such that $\lambda_i(v) = \lambda_j(v')$. In other words, each variable-to-variable mapping rewrites each non-projected variable of each union operand to a fresh variable. Then the following equivalence holds:

$$\text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n)) \equiv \\ \text{SELECT}_V(\text{UNION}(\lambda_1(Q_1), \ldots, \lambda_n(Q_n)))$$

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

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

which concludes the proof. □

Removing Variables that are Always Unbound

Next we apply a simple rule to remove variables that are always unbound in projections.

Lemma 5.1.2. Let Q be a query pattern on V, let V' be a set of variables, and let V'' be a set of variables such that $V \cap V'' = \emptyset$. It holds that $\text{SELECT}_{V'}(Q) \equiv \text{SELECT}_{V' \cup V''}(Q)$.

For example, the query in Figure 5.2 has three projected variables: `?m`, `?n` and `?o`. However, `?o` appears nowhere else in the query, which means that it cannot produce results. Therefore, as part of our normalisation, we eliminate variable `?o` from the projection.

Proof. Let $V = \text{pvars}(Q)$ denote the possible variables of Q, defined to be the set of variables $v \in \text{vars}(Q)$ such that there exists a dataset D and a solution $\mu \in Q(D)$ where $v \in \text{dom}(\mu)$. Then, for such a dataset D, if $\mu \in Q(D)$ then $\text{dom}(\mu) \subseteq V$. For each solution $\mu \in Q(D)$, the projection $\text{SELECT}_{V'}(Q)(D)$ produces a solution μ' such that $\text{dom}(\mu') \subseteq V' \cap \text{dom}(\mu)$ and $\mu \sim \mu'$, while the projection $\text{SELECT}_{V' \cup V''}(Q)$ produces a solution μ'' such that $\text{dom}(\mu'') =$

```
SELECT DISTINCT ?m ?n ?o
WHERE{
  { ?a a :Movie .
  ?a :title ?m .
  ?a :genre :Horror . }
  UNION
  { ?a a :Movie .
  ?a :sequel ?b .
  ?b :title ?n .
  ?a :genre :Horror . }
}
```

Figure 5.2: An example of a query that projects an unbound projected variable.

$(V' \cup V'') \cap \mathrm{dom}(\mu)$ and $\mu \sim \mu''$. But since $\mathrm{dom}(\mu) \subseteq V$ and $V \cap V'' = \emptyset$, this means that $\mathrm{dom}(\mu) \cap V'' = \emptyset$. Hence $\mathrm{dom}(\mu'') = (V' \cup V'') \cap \mathrm{dom}(\mu) = V' \cap \mathrm{dom}(\mu) = \mathrm{dom}(\mu')$. Further given that $\mu \sim \mu'$, $\mu \sim \mu''$, $\mathrm{dom}(\mu') = \mathrm{dom}(\mu'') \subseteq \mathrm{dom}(\mu)$, we can conclude that $\mu' = \mu''$. We have thus shown a one-to-one mapping from solutions of the form μ' to μ'' such that $\mu' = \mu''$, and thus the result holds. □

Finally, we present in Algorithm 3 pseudo-code that depicts how this may be implemented. Recall that λ_i is a variable to variable mapping that rewrites non-projected variables to fresh variables unique to the union operand Q_i. We additionally remove variables in V that do not appear in the query.

Algorithm 3 Pseudo-algorithm for renaming non-correlated variables of UCQs.

Input: A UCQ $Q = \textsc{select}_V(\textsc{union}(Q_1, \ldots, Q_n))$
Output: A UCQ Q' where local variables are uniquely named, and such that $Q' \equiv Q$.

```
function VARIABLENORMALISATION(Q)
    V' ← V
    for all Q_i ∈ {Q_1, ..., Q_n} do
        for all v ∈ vars(Q_i) do
            if v ∉ V then
                v ← λ_i(v)
            else
                V' ← V' \ {v}
            end if
        end for
    end for
    V ← V \ V'
    return SELECT_V(UNION(Q_1, ..., Q_n))
end function
```

Unsatisfiable BGPs

While in the general case, we are not able to verify preemptively if a graph pattern is satisfiable or not, there is a trivial case for (U)CQs [64]. This is when a BGP contains a triple pattern with a literal value in the subject position, which is allowed in SPARQL but not in

RDF (as of the time of this study). Therefore, it is not possible for such a pattern to yield any results when evaluated. We state the following Lemma:

Lemma 5.1.3. Let Q denote a BGP. Q is unsatisfiable if and only if it contains a literal subject.

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

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

```
SELECT DISTINCT ?m ?n ?o
WHERE{
  ?a a :Movie .
  ?a :title ?m .
  "Genre" :genre :Horror . }
```

Figure 5.3: An example of an unsatisfiable BGP.

Figure 5.3 contains a BGP with a triple pattern with a literal value in the subject position. Evidently, there are no graphs in which this will yield results because RDF does not allow literal values in the subject position. As a result, our approach is to replace any unsatisfiable BGP with a *canonical* unsatisfiable BGP ($Q_\perp$) so all unsatisfiable BGPs have the same canonical form.

Finally, we must also address what happens with unsatisfiable BGPs in UCQs. We state the following Lemma:

Lemma 5.1.4. A UCQ $Q = \textsc{union}(Q_1, Q_2, \cdots, Q_n)$ is unsatisfiable if and only if each Q_i (where $i \in \{1, \ldots, n\}$) is unsatisfiable. Furthermore, a UCQ $Q = \textsc{union}(Q_1, \cdots, Q_k, \cdots, Q_n)$ where one of its CQs Q_k is unsatisfiable is equivalent to
$Q' = \textsc{union}(Q_1, \cdots, Q_{k-1}, Q_{k+1}, \cdots, Q_n)$ where Q_k has been removed.

PROOF. The evaluation of a UCQ $\textsc{union}(Q_1, \ldots, Q_n)(D)$ is the union of the evaluation of each CQ $\bigcup_{i=1}^{n} Q_i(D)$ (with union being bag or set union depending on the semantics). The result is (always) empty if and only if all $Q_i(D)$ are (always) empty. Removing any Q_k such that $Q_k(D)$ is (always) empty will not affect the results of the query. Thus the result holds. $\square$

An example of this is presented in Figure 5.4, where the left query is a UCQ that contains an unsatisfiable CQ. As we have stated, this is equivalent to the same UCQ obtained by removing the unsatisfiable CQ.

Algorithm 4 Pseudo-algorithms for the normalisation of unsatisfiable UCQs.

Input: A BGP $Q = \{t_1, \cdots, t_n\}$
Output: Returns $Q_\perp$ if Q is unsatisfiable or Q otherwise.

```
function isUnsatisfiableBGP(Q)
    for all t_i ∈ Q do
        if t_i has a literal subject then
            return Q_⊥
        end if
    end for
    return Q
end function
```

Input: A UCQ $Q = \textsc{select}_V(\textsc{union}(Q_1, \ldots, Q_n))$
Output: Removes all unsatisfiable CQs in Q or returns $Q_\perp$ if Q is unsatisfiable.

```
function unsatisfiabilityNormalisation(Q)
    S ← {}
    for all Q_i ∈ {Q_1, ..., Q_n} do
        if isUnsatisfiableBGP(Q) ≠ Q_⊥ then
            S ← S ∪ {Q_i}
        end if
    end for
    if S is not empty then
        return select_V(union(S))
    else
        return Q_⊥
    end if
end function
```

```
SELECT DISTINCT ?m
WHERE{
  { ?a a :Movie .
  ?a :title ?m .
  "Genre" :genre :Horror . }
  UNION
  { ?a a :Movie .
  ?a :title ?m .
  ?a :genre :Horror . }
  }
```

```
SELECT DISTINCT ?m
WHERE{
  ?a a :Movie .
  ?a :title ?m .
  ?a :genre :Horror .
  }
```

Figure 5.4: An example of two equivalent UCQs. Note that the UCQ on the left contains an unsatisfiable CQ.

Algorithm 4 presents pseudo-code for the normalisation of unsatisfiable UCQs as described earlier. Here, we have defined a function ISUNSATISFIABLEBGP that given a query Q returns a normal unsatisfiable query $Q_\perp$ if Q is unsatisfiable, or Q otherwise. The next function UNSATISFIABILITYNORMALISATION takes a UCQ Q then uses the previous function to check each BGP in Q for satisfiability, and returns the union of all satisfiable BGPs, or $Q_\perp$ if none are satisfiable.

Set vs. Bag Semantics

In this section, we describe how in certain cases, the presence of the `DISTINCT` (or `REDUCED`) keyword does not affect the multiplicity of the solutions that are generated by a query because no duplicates can occur. We identify and normalise two specific cases – relating to UCQs – where such cases occur.

```
SELECT ?a ?m
WHERE{
  ?a a :Movie .
  ?a :title ?m .
  ?a :genre :Horror .
  }
```

```
SELECT DISTINCT ?a ?m
WHERE{
  ?a a :Movie .
  ?a :title ?m .
  ?a :genre :Horror .
  }
```

Figure 5.5: An example of two equivalent BGPs.

Figure 5.5 shows two queries that contain the same triple patterns, and only differ in that the rightmost one contains the `DISTINCT` keyword, whereas the leftmost one does not. Despite this difference, both queries are equivalent because neither can return duplicates.

Lemma 5.1.5. Let Q denote a basic graph pattern. It holds that:

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

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

Or more simply, this means that a basic graph pattern that projects every variable (or lacks an explicit projection) cannot produce duplicate results as long as it contains no blank nodes. Therefore, it is equivalent to the same query where the `DISTINCT` keyword is specified.

PROOF. First we prove that given a basic graph pattern Q, if $\text{vars}(Q) = V$ and $\text{bnodes}(Q) = \emptyset$, then $\text{DISTINCT}(\text{SELECT}_V(Q)) \equiv \text{SELECT}_V(Q)$. It suffices to show that $\text{SELECT}_{\text{vars}(Q)}(Q)$ cannot produce duplicate results, which we will now prove by contradiction. Assume a dataset D such that there exists a solution μ where $\text{SELECT}_{\text{vars}(Q)}(Q)(D)(\mu) > 1$; i.e., the solution appears more than once. Since all variables are projected, $\text{SELECT}_{\text{vars}(Q)}(Q) \equiv Q$. For μ to be duplicated, there must then exist multiple blank node mappings α such that $\text{dom}(\alpha) = \text{bnodes}(Q)$ and $\mu(\alpha(Q)) \subseteq D$ (see Table 4.4), but since $\text{bnodes}(Q) = \emptyset$, there is only the single empty mapping α, and hence we have a contradiction. The case $\text{vars}(Q) \subsetneq V$ follows from this result and Lemma 5.1.2.

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

Following this, we may extend the following Lemma for UCQs:

Lemma 5.1.6. Let $Q_1, \ldots, Q_n$ denote basic graph patterns and let $Q = Q_1 \cup \ldots \cup Q_n$ denote the set union of their triple patterns. It holds that:

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

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

PROOF. As before, we first prove that if blank nodes are not present ($\text{bnodes}(Q) = \emptyset$), all variables are projected ($\text{vars}(Q) \subseteq V$) and no BGP has the same set of variables ($\text{vars}(Q_i) \neq \text{vars}(Q_j)$ for all $1 \leq i < j \leq n$), then $\text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))$ cannot produce duplicate results. We know from Lemma 5.1.5 that if blank nodes are not present and all variables are projected, then no individual BGP in $Q_1, \ldots, Q_n$ can produce duplicate results. Hence we are left to check for duplicates produced by unions of BGPs. We will assume for the purposes of contradiction that there exists a dataset D and a solution μ such that $\mu \in Q_i(D)$ and $\mu \in Q_j(D)$ where $1 \leq i < j \leq n$. However, $\mu \in Q_i(D)$ implies that $\text{dom}(\mu) = \text{vars}(Q_i)$, while $\text{dom}(\mu) = \text{vars}(Q_j)$ implies that $\text{dom}(\mu) = \text{vars}(Q_j)$. Given the assumption that $\text{vars}(Q_i) \neq \text{vars}(Q_j)$, it follows that $\text{dom}(\mu) \neq \text{dom}(\mu)$: a contradiction. It then holds as a consequence that if $\text{vars}(Q) = V$, $\text{bnodes}(Q) = \emptyset$ and $\text{vars}(Q_i) \neq \text{vars}(Q_j)$ for all $1 \leq i < j \leq n$, then $\text{DISTINCT}(\text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))) \equiv \text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))$. The special case of $\text{vars}(Q) \subsetneq V$ follows from this result and Lemma 5.1.2.

In the other direction, we are left to show that if $\text{vars}(Q) \nsubseteq V$, or $\text{bnodes}(Q) \neq \emptyset$, or there exist $1 \leq i < j \leq n$ such that $\text{vars}(Q_i) = \text{vars}(Q_j)$, then it follows that $\text{DISTINCT}(\text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))) \not\equiv \text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))$.
First, if $\text{vars}(Q) \nsubseteq V$ or $\text{bnodes}(Q) \neq \emptyset$, then Lemma 5.1.5 tells us that an individual basic graph pattern with a blank node or non-projected variable can produce duplicates. Hence we assume that $\text{vars}(Q) \subseteq V$ and $\text{bnodes}(Q) = \emptyset$, and show that if there exists $1 \leq i < j \leq n$ such that $\text{vars}(Q_i) = \text{vars}(Q_j)$, then duplicates can always arise for the query $\text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))$. Let μ be a solution such that $\text{dom}(\mu) = \text{vars}(Q_i) = \text{vars}(Q_j)$ and $\mu(v) = \texttt{:x}$ for all $v \in \text{dom}(\mu)$. Consider a dataset D whose default graph is defined as $\mu(Q_i) \cup \mu(Q_j)$; again this is an RDF graph as Q_i and Q_j are assumed to be satisfiable. Now $\mu \in Q_i(D)$ and $\mu \in Q_j(D)$, and since all variables are projected, we conclude that μ will be duplicated in $\text{SELECT}_V(\text{UNION}(Q_1, \ldots, Q_n))(D)$. The result holds. □

As a result of this, a UCQ where no two CQs share the same set of projected variables cannot produce duplicate results. Algorithm 5 presents pseudo-code showing how we may check if a UCQ can produce duplicates.

Algorithm 5 Pseudo-algorithm for determining if a UCQ can produce duplicates.

```
Input: A UCQ Q = SELECT_V(UNION(Q_1, ..., Q_n))
Output: A UCQ Q' = DISTINCT(Q) if Q cannot produce duplicates, or Q otherwise.
  function setVsBagNormalisation(Q)
    if vars(Q) \ V ≠ ∅ then                    ▷ i.e., Q contains a non-projected variable.
      return Q
    else if bnodes(Q) ≠ ∅ then                 ▷ i.e., Q contains blank nodes.
      return Q
    else
      for all Q_i ∈ {Q_1, ..., Q_n} do
        for all Q_j ≠ Q_i do
          if vars(Q_i) ∩ V = vars(Q_j) ∩ V then    ▷ i.e., Q_i and Q_j have the same
  projected variables.
            return Q
          end if
        end for
      end for
    end if
    return DISTINCT(Q)
  end function
```

Algorithm 5 shows pseudo-code to verify if a UCQ (without `DISTINCT`) can produce duplicate results or not, and adds the `DISTINCT` keyword if not. First, we must check if Q contains any variables that are not projected or blank nodes. If neither is the case, then we perform pair-wise checks to find if any distinct BGPs contain the same projected variables. If none are found, then we return $\text{DISTINCT}(Q)$.

5.1.2 Graph Representation

In this section, we describe the representation of queries as *representational graphs* (*r-graphs*). One of the advantages of this representation is that it captures the associative property of joins and unions. Furthermore, the minimisation and canonical labeling steps depend on this representation. We will use $\mathrm{R}(Q)$ to denote the r-graph representing Q, i.e., an RDF graph describing Q.

Terms

SPARQL queries contain terms from **ILBV** while RDF graphs contain terms from **ILB**. Terms from **V** will be mapped to blank nodes to facilitate canonical labelling. In practice, blank nodes in SPARQL queries act as variables, except that they cannot be projected from the query. SPARQL syntax does not allow the projection of said terms, hence no work is done to distinguish variables from blank nodes in the query. Therefore we represent both **B** and **V** in SPARQL as **B** in RDF.

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

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

Figure 5.6: Illustration of SPARQL terms: blank nodes and variables (left), IRIs (center), and literals (right)

We assume that natural numbers and boolean values produce datatype literals in canonical form (for example, we assume that $\iota(2) =$ `"2"^^xsd:integer` rather than, say, `"+02"^^xsd:integer`). We then define the representation graphs for a variety of query features in Table 5.3.

In all of the following examples, nodes with a dashed outline denote other query graphs. We also use an example prefix : whose value is not important in this study. The conventions used for graphically representing different types of RDF terms are illustrated in Figure 5.6.

Triple patterns

In this section, we describe the representation of triple patterns as graphs. Our representation is a form of reification, except that it is limited to single triples and uses a different typing system.

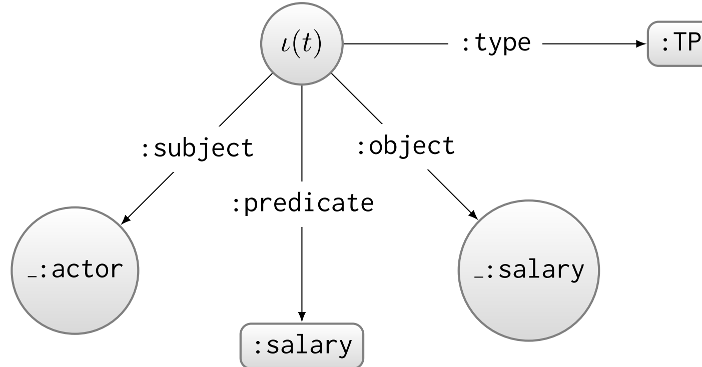


Figure 5.7: Representation of a Triple Pattern

Figure 5.7 illustrates our representation of a triple pattern from Figure 1.1. There are two blank nodes that correspond to the variables in the triple denoted by circles, and a blank node ($\iota(t)$) that represents the structure as a whole. The predicate of the triple is an IRI, which is represented as a rounded rectangle. Finally, there is a node that indicates that this graph represents a triple pattern (`:TP`).

Operators

To represent operators such as joins and unions, triples are created for each operand with a blank node as the subject, the IRI `:arg` as the predicate, and the operand as the object. One other triple is used to indicate the type of the operator: either `:Join` or `:Union` . One particular advantage of using the same predicate for every operand is that it captures the commutativity of both operations in SPARQL, as they are unordered.

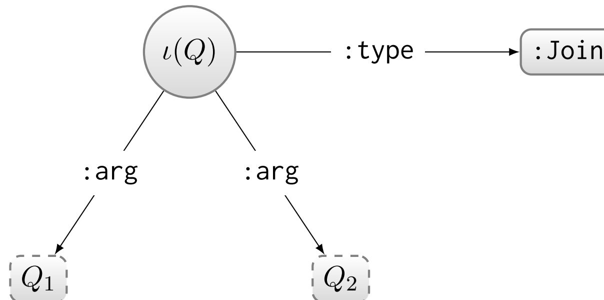


Figure 5.8: Visualisation of the graph representation of a `JOIN` expression.

Figure 5.8 illustrates the representation of an expression $Q = Q_1 \bowtie Q_2$ where Q_1 and Q_2 are sub-queries, which are denoted by dashed rectangles. These dashed rectangles are replaced by their corresponding graph representation, where the edge indicated is linked to

the blank node of the highest-level operator. For instance, if Q_1 were the triple pattern represented by the graph in Figure 5.7, then the node labelled $\iota(Q)$ would be linked to the node labelled $\iota(t)$. Note that Q_1 and Q_2 may be other operators or a triple pattern as seen in Figure 5.7. An advantage of this structure is that because graph edges are unordered, we have already captured the commutative property of join.

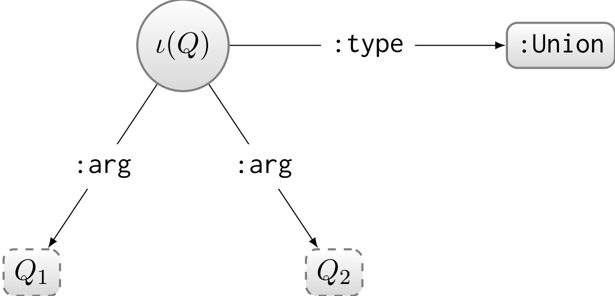


Figure 5.9: Visualisation of the graph representation of a UNION expression.

Figure 5.9 illustrates the representation of an expression $Q = Q_1 \cup Q_2$, which is analogous to the join operator presented in Figure 5.8. The one key difference is the type node that denotes that this structure is of type `:Union` .

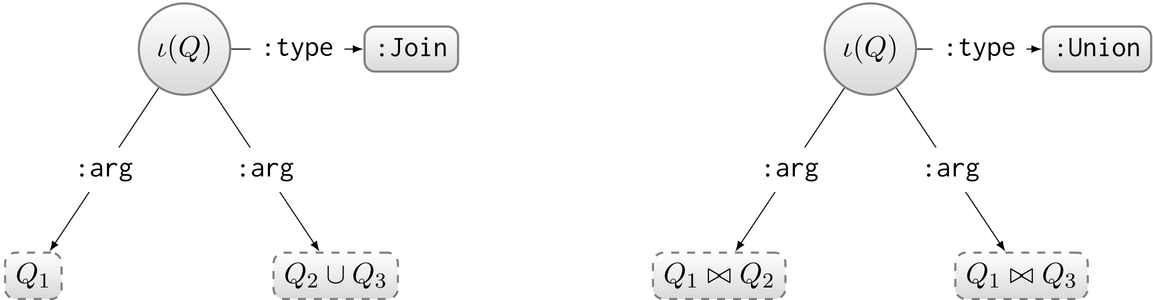


Figure 5.10: Visualisation of the distributive Property of JOIN and UNION.

Figure 5.10 illustrates an example of how the application of the distributive property of join over union should affect a graph. The graph on the left corresponds to a conjunction of disjunctive queries (i.e. a join of unions), whereas the graph on the right shows an equivalent query in the form we wish to represent: a disjunction of conjunctive queries (i.e. a union of joins).

Projection

A projection is defined by the projected variables and whether it considers set or bag semantics. Therefore, we have represented a projection as a blank node that is connected to: all projected variables with an `:arg` predicate; a boolean literal node with a `:distinct` predicate denoting whether it contains the `DISTINCT` keyword or not; and an `:op` predicate pointing to the first operation of the r-graph denoting the body of the query. We also include a representation for solution modifiers (`ORDER BY`, `LIMIT`, `OFFSET`) via the `:mod` predicate, though not part of the monotone query fragment.

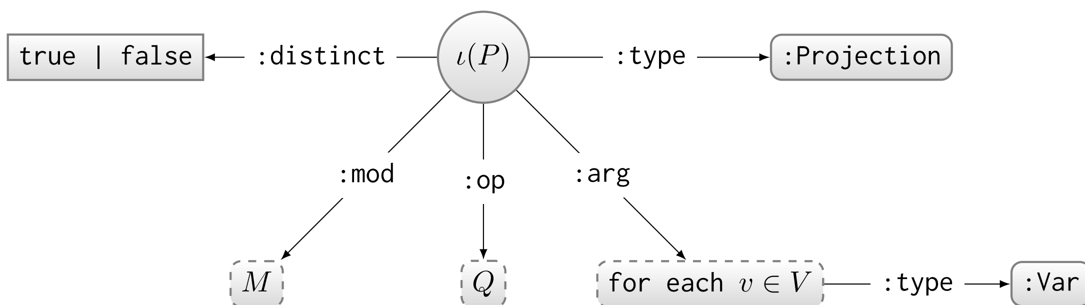


Figure 5.11: Visualisation of the graph representation of an explicit projection $P = \text{SELECT}_V(Q)$.

Figure 5.11 illustrates the representation of a projection, as described above. A projection such as this, assuming $V = \{$?a, ?b$\}$, may read as:

```
SELECT DISTINCT ?a ?b
```

Note that in such a case, the literal node would have the value *true*. The dashed circle represents a variable number of blank nodes that are constructed according to the rule inside it; in this case, there is a blank node created for each variable v in the set of projected variables V. For instance, there would be a blank node for _:a and another for _:b.

With this, we have defined the representations of the SPARQL operators for which we have implemented a full canonicalisation algorithm. We note that this process does not alter the semantics of the query in any way. Finally, given a monotone query Q, where $[a\ \text{AND}\ n]$d $[a\ \text{UNION}\ c]$cept sets rather than pairs of arguments, and because this process maps each component of the query in a one-to-one manner, and no two distinct components are represented by the same term, the graph representation function $\text{R}(Q)$ is a bijection, and $\text{R}^{-}(\text{R}(Q)) = Q$ where we can construct a mapping $\text{R}^{-}()$ that is the inverse function of $\text{R}()$.

5.1.3 Redundancy-free Monotone Queries

As we have stated, when queries are evaluated under set semantics, equivalence between queries may be preserved if we remove certain redundant triples. Evidently, for the purpose of designing a normal form, such triples must also be removed.

In this section, we explain how to remove all of these *redundancies* from the r-graphs. At this point, all monotone fragments have already been transformed into normal UCQs; therefore the query consists in a single union operator with either triple patterns or conjunctive queries as arguments.

The redundancy elimination step considers two types of redundancies: redundant triple patterns in basic graph patterns, and redundant basic graph patterns in unions.

Redundancy-free CQs

The first type of redundancy we consider are redundant triple patterns. As we mentioned in Section 5.1.2, 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 as well. As explained in Section 2.2, a *lean* RDF graph does not contain any redundant triples. This also means that the core of the graph representation of a basic pattern contains no redundant triples. Therefore, it is clear that if we retrieve its basic graph pattern, it will not contain any redundant triple patterns.

Our approach will consist of the following steps:

- Remove the redundancies of each individual conjunctive query
- Compute the canonical form of the union of the reduced conjunctive queries

These steps will effectively remove the redundancy between conjunctive queries. In the following description of the solution, we may use *branch* to refer to each individual conjunctive query. We base our discussion on the example UCQ of Figure 5.1.

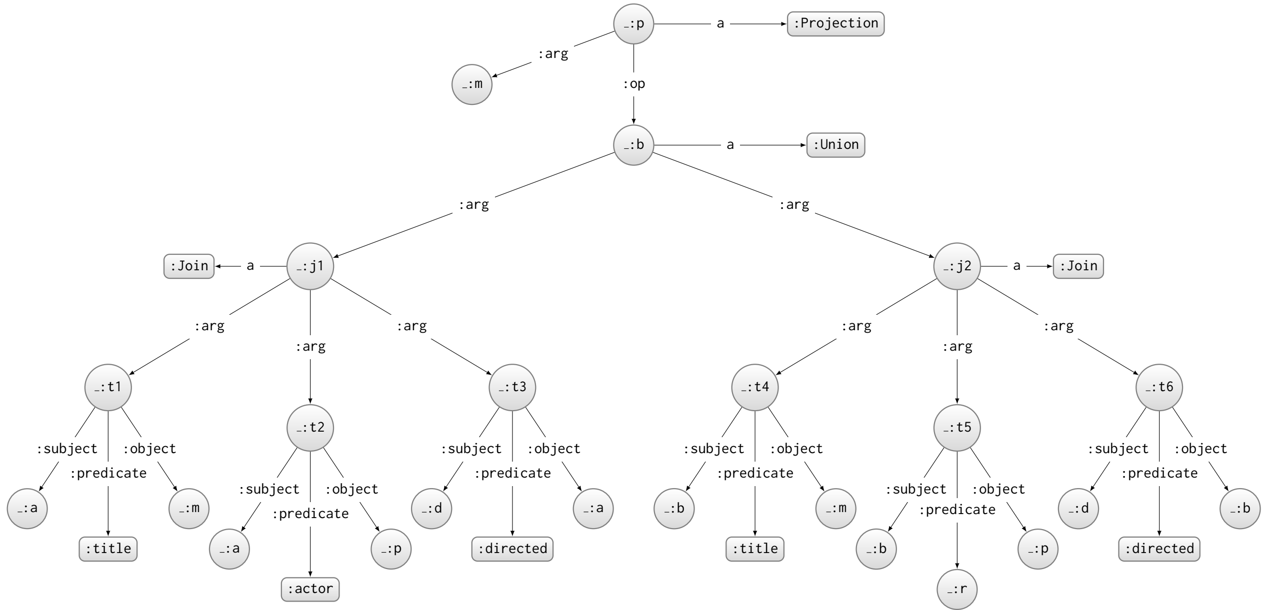


Figure 5.12: Example of an r-graph for a UCQ

Note that for readability, certain nodes are presented multiples time (for example `_:a` in Figure 5.12).

To begin, we identify two main components of the query based on the union operator node: the inner graph which contains all the conjunctions, and the outer graph which contains all other components, including: the type of projection, the variables that are projected, solution modifiers; all of which are relevant to identify variables whose labels must be preserved throughout this process.

We continue by identifying projected variables that we cannot remove during minimisation. Once we have identified these variables, we must *ground* them by defining literal nodes that contain the literal string value of each blank node label, thus ensuring that they will not be removed. These literal nodes will later be used to restore its original label.

The inner graph is then divided into each of its branches. For each branch, we will work with a graph corresponding to the union operator node with the individual branch as an argument. What is not shown in these diagrams is that the nodes that correspond to these variables are typed as `:Var`, unlike non-projected variable nodes.

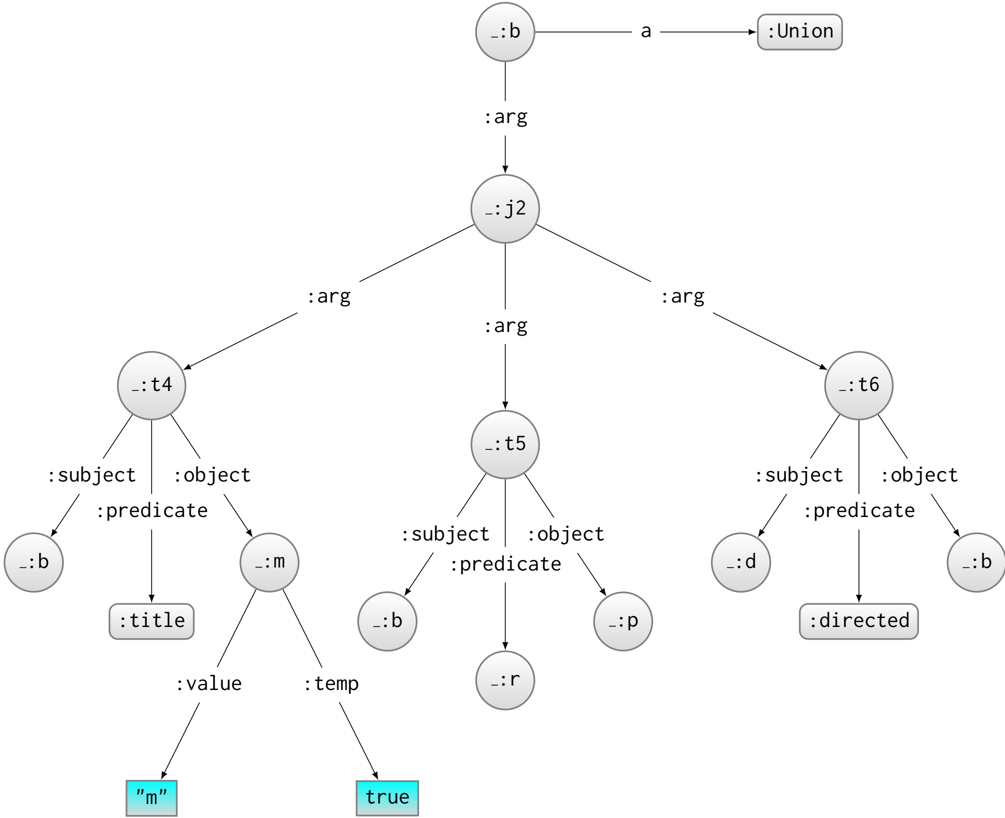


Figure 5.13: Example of the r-graph for a CQ.

Figure 5.13 illustrates the r-graph corresponding to a single branch (CQ). Note that each of the individual branches contains a `:Union` node which will be used in a later step to merge all the branches into a single union node. The variable `?m` is grounded with a literal value as it is projected by the UCQ, and thus it must be preserved during minimisation.

New blank nodes are created for every variable node that is neither projected, nor otherwise appears outside the scope of this part of the query. These blank nodes are labelled based on its branch and its original label. In the following figure, this is exemplified by the concatenation of a number and the original labels.

In Figure 5.14 we can observe the same r-graph as the one in Figure 5.13, except for the newly labelled nodes (shown in yellow). Note that the node corresponding to variable `?m` remains the same because it is projected.

Finally, we compute the core of the graph for each newly labelled branch, and store them

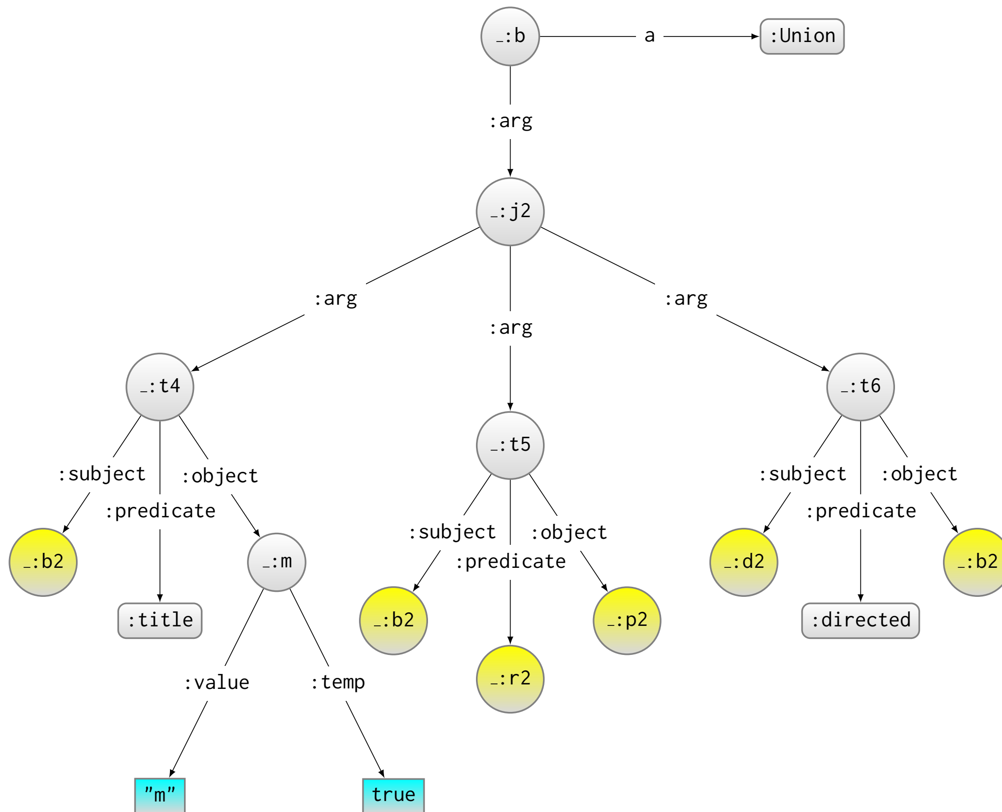


Figure 5.14: Example of a CQ with distinguished labels.

in an array for the next step. This is shown in Figure 5.15 where the sub-graph representing the triple pattern (?b2, ?r2, ?p2) has been removed.

Redundancy-free UCQs

While it may be tempting to perform a union over all the new branches and finish the process, we must consider there may be redundant basic graph patterns in this union. In a UCQ under set semantics, a basic graph pattern B_i is redundant if there exists another basic graph pattern B_j such that $B_i \sqsubseteq B_j$. This is evident because any results for B_i also appear in B_j (no matter the graph), so B_i adds no more results.

So far, our solution eliminates redundancies in CQs by finding the core of a graph. This works perfectly when considering only conjunctions, since when regarding conjunctions, a more specific query will take precedence over a more general one. However, this is not the case for unions; let us consider the queries presented in Figure 5.16, both of which are equivalent to each other:

Computing the core of the r-graph of the query on the left in Figure 5.16 will yield the query on the right, because the triple pattern (?a, ?b, ?m) is absent from the core.

On the other hand, if we were to compute the core of the r-graph of the queries in Figure 5.17, both queries would have the same canonical form. This would change the semantics of

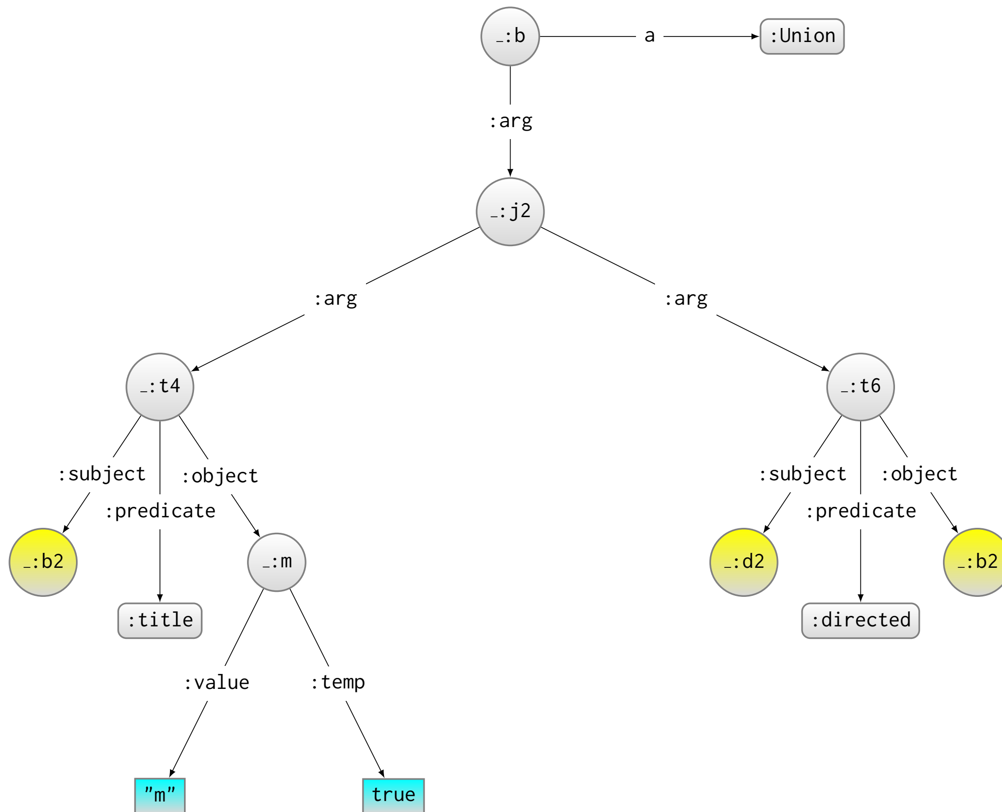


Figure 5.15: Example of a lean CQ.

the first query. This is because in the case of union, the more specific CQ is the redundant one, therefore we should drop it to avoid redundancy. In Figure 5.17, in the query on the left the more specific query is (`?a`, `:title`, `?m`), which is what we should drop.

As such, in the case of UCQs, it is not sufficient to compute the core of the r-graph of the UCQ, but rather we need to verify if there exists a case where one of the branches of the UCQ entails another. Furthermore, we should only compare CQs that contain the same set of

```
SELECT DISTINCT ?m
WHERE{
 ?b :title ?m .
 ?b ?r ?p .
 ?d :directed ?b .
}
```

```
SELECT DISTINCT ?m
WHERE{
 ?b :title ?m .
 ?d :directed ?b .
}
```

Figure 5.16: Example of two equivalent queries, if we remove their redundancies.

```
SELECT DISTINCT ?m
WHERE{
 { ?a :title ?m }
 UNION
 { ?a ?b ?m }
}
```

```
SELECT DISTINCT ?m
WHERE{
 ?a :title ?m .
}
```

Figure 5.17: On the other hand, these two queries are *not* equivalent

projected variables, as otherwise we will never find equivalences. Therefore, after computing *equivalence sets* of CQs based on the projected variables they contain $\{\mathcal{Q}_1, \ldots, \mathcal{Q}_k\}$ where each $\mathcal{Q}_i$ is a set of CQs with the same projected variables, we may do the following:

Let $\mathcal{Q} = (\{Q_1, \cdots, Q_n\}, V)$ be the current equivalence set, where V denotes the projected variables of this partition. The following must be removed from $\mathcal{Q}$:

- Q_i $(1 \leq i \leq n)$ if there exists Q_j $(1 \leq j < i \leq n)$ such that $\text{SELECT}_V(Q_i) \equiv \text{SELECT}_V(Q_j)$
- Q_i $(1 \leq i \leq n)$ if there exists Q_j $(1 \leq i \leq n)$ such that $\text{SELECT}_V(Q_i) \sqsubset \text{SELECT}_V(Q_j)$ (a proper containment)

The first condition removes all queries apart from one for each equivalence set. This ensures that no information is lost in this process. The second condition removes all queries that are properly contained in another.

In order to do this, we perform pairwise containment checks of all basic graph patterns. This is done by querying Q_j over the representational graph of Q_i in order to find if there exists a homomorphism from Q_j to Q_i. It is a well-known result that if there exists a homomorphism from Q_j to Q_i then $Q_i \sqsubseteq Q_j$ [20]. Finally, we eliminate all basic graph patterns that are found to be contained in another basic graph pattern.

Algorithm 6 shows pseudo-code for the minimisation of UCQs. We assume there exists a function partitionByProjectedVars that given a set of BGPs forms a partition of equivalence classes based on their projected variables, and a function ground that given a query Q and a set of variables V grounds all variables in Q that are contained in V. In addition, we define a function removeRedundantBGPs that given a set of BGPs removes all redundant BGPs (i.e. is contained in a different BGP) or all but one of each group of equivalent BGPs (e.g., if Q_1, Q_2 and Q_3 are equivalent, then Q_2 and Q_3 are removed). All BGPs that are compared within a partition contain the same projected variables, and have been minimised previously. To ascertain entailment, we perform the following procedure: given G_i and G_j representing the BGPs Q_i and Q_j on different branches, where we want to check if $Q_i \sqsubseteq Q_j$, we evaluate Q_j as a Boolean query (in SPARQL, by replacing the `SELECT` with an `ASK` clause) over G_i. If it holds true, we conclude that G_i entails G_j, and therefore we add Q_i to the list of redundant queries. Although the number of checks needed is quadratic on the number of BGPs, in practice we can reduce the number by checking if a BGP has already been found to be redundant. For example, if Q_i is known to be redundant, then evaluating $Q_i \subseteq Q_j$ would provide no information on whether Q_j is redundant or not. On the other hand, evaluating $Q_j \subseteq Q_i$ would be enough to conclude that Q_j is redundant. Given the previous functions, UCQMinimisation then iterates over each equivalence class, removes redundant BGPs, then returns the union of all remaining BGPs.

Figure 5.18 illustrates the canonically labelled `ASK` query form of the r-graph shown in Figure 5.13 (denoted as G_2), after leaning; canonical labels are shown in Greek letters. On the other hand, the set of triples in Figure 5.19 correspond to the triples contained in the first CQ from the left query shown in Figure 5.1 with canonical labels (denoted as G_1). In practice, both the query and the graph would be in reified form (to handle variable predicates), but for illustration purposes we show both in non-reified form; we likewise replace the projected

Algorithm 6 Pseudo-algorithm for the minimisation of UCQs.

Input: A UCQ $Q = \textsc{select}_V(\textsc{union}(Q_1, \ldots, Q_n))$
Output: A minimised UCQ Q'.

```
function UCQMinimisation(Q)
    S ← {}
    for all ({Q_i1, ..., Q_ik}, V_i) ∈ partitionByProjectedVars(Q_1, ···, Q_n) do
        S ← S ∪ removeRedundantBGPs({Q_i1, ..., Q_ik}, V_i)
    end for
    return union(S)
end function
```

Input: A set of BGPs $Q = \{Q_1, \ldots, Q_n\}$ that contain the projected variables in V.
Output: A set of BGPs Q' where no two distinct BGPs contain one another.

```
function removeRedundantBGPs(Q, V)
    R ← {}                                   ▷ This set contains all redundant BGPs.
    for all i ∈ {1, ..., n} do
        for all j > i do
            C_i ← l(r(ground(Q_i, V)))
            C_j ← l(r(ground(Q_j, V)))
            if C_i ≡ C_j then
                R ← R ∪ {Q_j}
            else if Q_i ∉ R and C_i ⊏ C_j then
                R ← R ∪ {Q_i}
            else if Q_j ∉ R and C_j ⊑ C_i then
                R ← R ∪ {Q_j}
            end if
        end for
    end for
    Q' ← Q \ R
    return Q'
end function
```

```
ASK
WHERE{
    ?π :title "m" .
    ?ν :directed ?π .
}
```

Figure 5.18: Example of an ASK query given by G_2 in the running example.

```
_:λ :title "m" .
_:λ :actor _:ϕ .
_:δ :directed _:λ .
```

Figure 5.19: Example of G_1 from the running example with canonical labels.

variable with a constant to show that it must be mapped to itself. Subsequently, the query in Figure 5.18 will be evaluated over the triple set in Figure 5.19 and return *true* because there exists a solution mapping for the query $\mu = \{(?\pi \to _{:}\lambda), (?\nu \to _{:}\delta)\}$. This in turn implies that G_2 is entailed by G_1, so as a result, G_1 will be removed from the equivalence set $\mathcal{G}$. Note that the canonical labels in Figure 5.18 and Figure 5.19 don't have to be the same for the process to work. Once we have all the unique conjunctive queries, we can unite them under a single union operator node.

5.1.4 Canonical Labelling

The second-last step of the canonicalisation process consists of applying a canonical labelling to the blank nodes of the RDF graph output from the previous process [38]. Specifically, given an RDF graph G, we apply the canonical labelling function $\text{L}(\cdot)$ described in Section 2.2.4 such that $\text{L}(G) \simeq G$ and for all RDF graphs G', it holds that $G \simeq G'$ if and only if $\text{L}(G) = \text{L}(G')$; in other words, $\text{L}(\cdot)$ bijectively relabels the blank nodes of G in a manner that is deterministic modulo isomorphism, meaning that any isomorphic graph will be assigned the same labels. This is used to assign a deterministic labelling of query variables represented in the r-graph as blank nodes; other blank nodes presenting query operators will also be labelled as part of the process but their canonical labels are not used.

Figure 5.20 illustrates the resulting r-graph for the query on the left of Figure 5.1 after all the processes we have described. All blank nodes have canonical labels –denoted by Greek letters– and the redundant branch has been removed. Since there is only a single conjunctive query, the `:Union` node has been removed, as it is also redundant.

5.1.5 Inverse Mapping

There are no more transformations to be done over the r-graphs at this point. We can now obtain a canonical query by applying the inverse of the operations used to construct the r-graphs. More specifically, we define an inverse r-mapping, denoted $\text{R}^{-}(G)$, to be a partial mapping from RDF graphs to query expressions such that $\text{R}^{-}(\text{R}(Q)) = Q$; i.e. converting Q

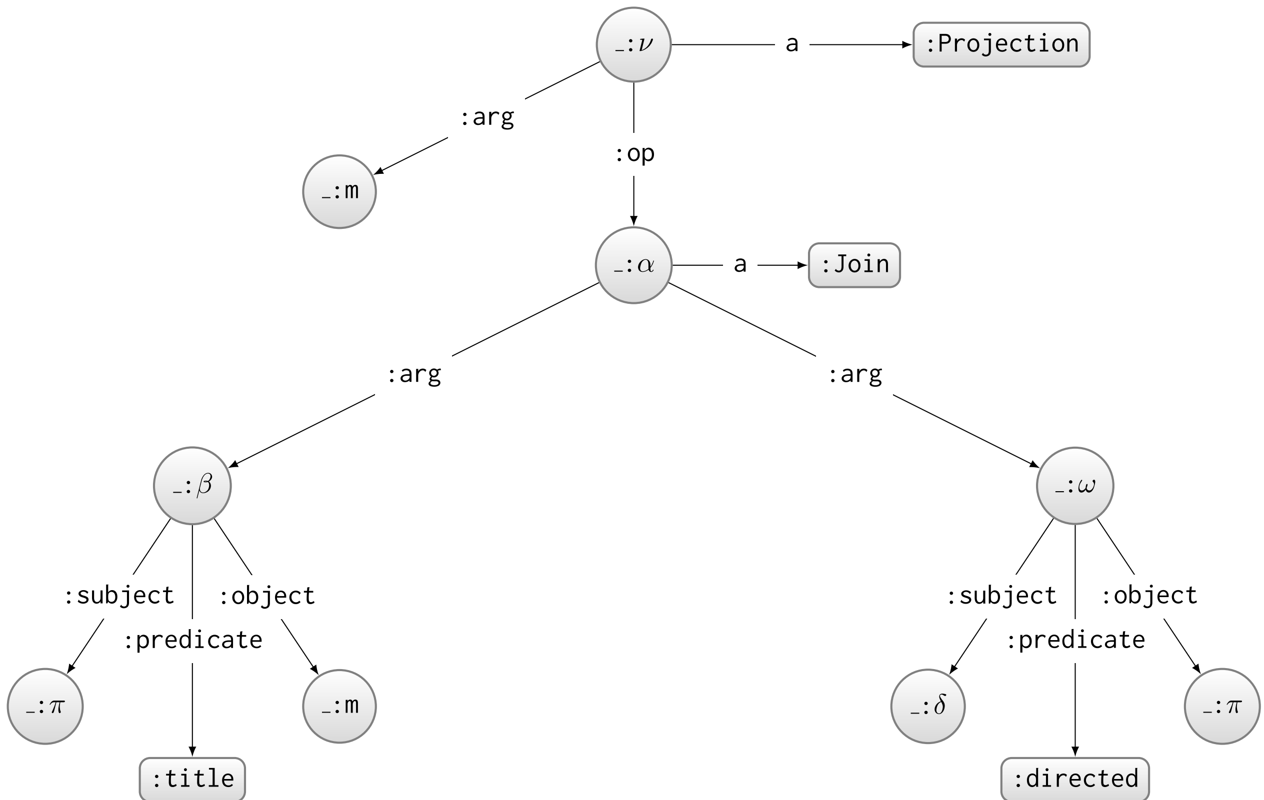


Figure 5.20: Resulting r-graph

to its r-graph and then applying the inverse r-mapping yields the query Q again.[2]

To arrive at a canonical concrete syntax, we order the operands of commutative operators using a syntactic ordering on the canonicalised elements, and then serialise these operands in their lexicographical order. To illustrate, Figure 5.21 shows the result of the full canonicalisation of our running example. This then concludes the canonicalisation of MQs.

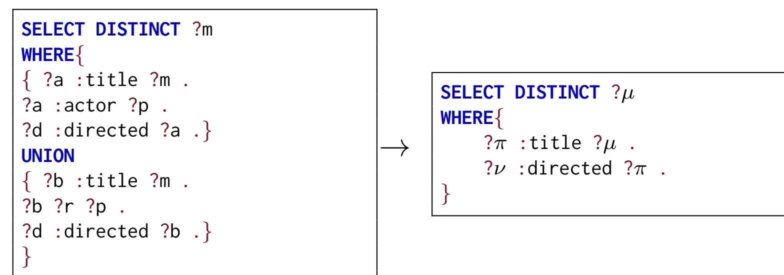


Figure 5.21: This shows the canonical form of the query presented in Figure 5.1.

[2]Here we assume the use of UNION(·), etc., to abstract away the ordering of operands of commutative operators.

5.2 SPARQL 1.1

In this section, we describe the additional steps of the canonicalisation method for features other than joins, unions and projections. We note that these techniques retain the soundness of the canonicalisation method because the semantics are not changed in any step. However, as we will explain in the next chapter in Section 6.2.2, the resulting canonical form will not capture all possible congruences. Nevertheless, this is seen as a "best-effort" canonicalisation, seeing as we may detect additional congruences that may be missed by standard techniques.

5.2.1 Normalisation

In this section, we present some additional rules for the rewriting of query patterns into equivalent, normal patterns. Most of the rules used in the following have been studied and defined thoroughly in the work [67] of Schmidt et al. A summary of the most important results is presented in Table 5.2.

Table 5.2: Equivalences given by Schmidt et al. [67] for set semantics

Minus is left distributive	$[[Q_1 \text{ UNION } Q_2] \text{ MINUS } Q_3] \equiv [[Q_1 \text{ MINUS } Q_3] \text{ UNION } [Q_2 \text{ MINUS } Q_3]]$
Optional is left distributive	$[[Q_1 \text{ UNION } Q_2] \text{ OPT } Q_3] \equiv [[Q_1 \text{ OPT } Q_3] \text{ UNION } [Q_2 \text{ OPT } Q_3]]$
Nested filters can be combined	$\text{FILTER}_{R_1 \wedge R_2}(Q) \equiv \text{FILTER}_{R_1}(\text{FILTER}_{R_2}(Q))$
Unions with filters can be combined	$\text{FILTER}_{R_1 \vee R_2}(Q) \equiv [\text{FILTER}_{R_1}(Q) \text{ UNION } \text{FILTER}_{R_2}(Q)]$
Filters on unions can be pushed in	$\text{FILTER}_{R}([Q_1 \text{ UNION } Q_2]) \equiv [\text{FILTER}_{R}(Q_1) \text{ UNION } \text{FILTER}_{R}(Q_2)]$
These equivalences hold if $\text{vars}(R) \subseteq \text{safeVars}(Q_1)$	:
	$\text{FILTER}_{R}([Q_1 \text{ AND } Q_2]) \equiv [\text{FILTER}_{R}(Q_1) \text{ AND } Q_2]$
	$\text{FILTER}_{R}([Q_1 \text{ OPT } Q_2]) \equiv [\text{FILTER}_{R}(Q_1) \text{ OPT } Q_2]$
	$\text{FILTER}_{R}([Q_1 \text{ MINUS } Q_2]) \equiv [\text{FILTER}_{R}(Q_1) \text{ MINUS } Q_2]$

Filter Normalisation

One of the most notable results presented in Table 5.2 is that filter expressions allow a significant number of rewrites as long as the expressions contain only *safe variables*, as described in Section 2.3.7. In addition, because of these rules, we may generalise monotone query patterns with multiple filter expressions into a single, combined filter expression over the query pattern. An example of this can be seen in Figure 5.22.

Non-correlated Variables

Similar to the case of UCQs as mentioned in Section 5.1.1, there are cases with variables in `FILTER (NOT) EXISTS` and `MINUS` patterns that may generate false equivalences.

We call a variable v *local* to a query pattern Q on V (see Table 4.1) if $v \in \text{vars}(Q)$ and $v \notin V$. A variable v *local* to Q is not correlated with other appearances of v outside of Q.

```
?x ?y ?z .
FILTER( ?x > 0 )
FILTER( isBlank(?z))
```

```
?x ?y ?z .
FILTER( ?x > 0 && isBlank(?z) )
```

Figure 5.22: An example of two equivalent `FILTER` expressions

Variables that appear only inside a `FILTER (NOT) EXISTS` or `MINUS` expression are denoted as *local variables* because they only exist inside the scope of the expression. Therefore, any instance of the variable outside the `FILTER (NOT) EXISTS` or `MINUS` expression will have a different semantic meaning.

```
SELECT DISTINCT ?m
WHERE{
{ ?a a :Movie .
  ?a :title ?m .
  ?a :genre :Horror .
MINUS { ?a :sequel ?b . ?b :title ?n . } }
UNION
{ ?a a :Movie .
  ?a :title ?m .
  ?a :genre :Comedy .
MINUS { ?a :sequel ?b . ?b :title ?n . } }
}
```

```
SELECT DISTINCT ?m
WHERE{
{ ?a a :Movie .
  ?a :title ?m .
  ?a :genre :Horror .
MINUS { ?a :sequel ?b . ?b :title ?n . } }
UNION
{ ?b a :Movie .
  ?b :title ?m .
  ?b :genre :Comedy .
MINUS { ?b :sequel ?a . ?a :title ?o . } }
}
```

Figure 5.23: An example of two equivalent queries.

In the example presented in Figure 5.23 we have a query that finds the titles of horror and comedy movies, except those that have sequels. We note that, similar to the example in Figure 5.1, the query on the left has variable `?a` on both sides of the union bound to results of type `:Movie` , and variable `?b` bound to the sequels. However, the query on the right inverts the use of these variables, and yet is equivalent to the query on the left. Additionally, the name of the sequel of comedies is also bound to different variables in the second `MINUS` clause of both queries (`?n` and `?o`, respectively) but this again does not affect the equivalence of the queries.

UCQ Normalisation

The UCQ normalisation rules presented in Section 5.1.1 can also be applied to all monotone sub-queries contained in full SPARQL queries. However, when it comes to rewriting union variables or otherwise non-correlated variables, we must not rewrite *any* variables that appear both inside and outside the monotone sub-query, instead of only projected variables.

In the example shown in Figure 5.24 we can appreciate that the query pattern in the

```
SELECT DISTINCT ?m ?n
WHERE {
  ?a a :Movie .
  ?a :title ?m .
  ?a :directedBy ?d
  ?a :genre :Horror .
  OPTIONAL
    {   ?c :sequel ?b .
        ?a :sequel ?b .
        ?b :title ?n .
        ?b :directedBy ?d . }
}
```

```
SELECT DISTINCT ?m ?n
WHERE {
  ?a a :Movie .
  ?a :title ?m .
  ?a :directedBy ?d
  ?a :genre :Horror .
  OPTIONAL
    {   ?a :sequel ?b .
        ?b :title ?n .
        ?b :directedBy ?d . }
}
```

Figure 5.24: An example of a SPARQL query with a monotone sub-query before and after removing a redundant triple pattern.

optional part of the `OPTIONAL` graph pattern in the left query is a BGP with a redundant triple pattern. However, it is clear that the minimisation must eliminate the triple pattern (?`c`, `:sequel`, ?`b`) rather than the one with `?a` instead. This is because `?a` appears outside the scope of this graph pattern even if neither `?a` nor `?c` are projected.

5.2.2 Graph Representation

In this section, we describe the graph representation of features other than those that appear in monotone queries (i.e. joins, unions and projections). Table 5.3 contains a summary of the definitions for the representational graphs of all features that are supported in SPARQL 1.1.

Table 5.3: Definitions for representational graphs $\text{R}(Q)$ of query patterns Q, where "a" abbreviates `rdf:type`, B is a basic graph pattern, N is a navigational graph pattern, c is an RDF term, e is a non-simple property path (not an IRI), k is a non-zero natural number, v is a variable, w is a variable or IRI, x is an IRI, y is a variable or property path, μ is a solution mapping, Δ is a boolean value, V is a set of variables, R is a built-in expression, and $\mathfrak{M}$ is a bag of solution mappings.

$\cdot$	$\text{R}(\cdot)$
B	$\{(\iota(\cdot), \texttt{a}, \texttt{:Join})\} \cup \bigcup_{(s,p,o)\in B}\{(\iota(\cdot), \texttt{:arg}, \iota((s,p,o))\} \cup \text{R}((s,p,o))$
N	$\{(\iota(\cdot), \texttt{a}, \texttt{:Join})\} \cup \bigcup_{(s,y,o)\in N}\{(\iota(\cdot), \texttt{:arg}, \iota((s,y,o))\} \cup \text{R}((s,y,o))$
$\text{AND}(\{Q_1,\ldots,Q_n\})$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q_1)),\ldots,(\iota(\cdot), \texttt{:arg}, \iota(Q_n)), (\iota(\cdot), \texttt{a}, \texttt{:Join})\} \cup \text{R}(Q_1) \cup \ldots \cup \text{R}(Q_n)$
$\text{UNION}(\{Q_1,\ldots,Q_n\})$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q_1)),\ldots,(\iota(\cdot), \texttt{:arg}, \iota(Q_n)), (\iota(\cdot), \texttt{a}, \texttt{:Union})\} \cup \text{R}(Q_1) \cup \ldots \cup \text{R}(Q_n)$
$[Q_1 \text{ MINUS } Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{a}, \texttt{:Minus})\} \cup \text{R}(Q_1) \cup \text{R}(Q_2)$
$[Q_1 \text{ OPT } Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{a}, \texttt{:Optional})\} \cup \text{R}(Q_1) \cup \text{R}(Q_2)$
$\text{FILTER}_R(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:exp}, \iota(R)), (\iota(\cdot), \texttt{a}, \texttt{:Filter})\} \cup \text{R}(Q') \cup \text{R}(R)$
$[Q_1 \text{ FE } Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{a}, \texttt{:Exists})\} \cup \text{R}(Q_1) \cup \text{R}(Q_2)$
$[Q_1 \text{ FNE } Q_2]$	$\{(\iota(\cdot), \texttt{:left}, \iota(Q_1)), (\iota(\cdot), \texttt{:right}, \iota(Q_2)), (\iota(\cdot), \texttt{a}, \texttt{:NotExists})\} \cup \text{R}(Q_1) \cup \text{R}(Q_2)$
$\text{BIND}_{R,v}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:exp}, \iota(R)), (\iota(\cdot), \texttt{:var}, \iota(v)), (\iota(\cdot), \texttt{a}, \texttt{:Bind})\} \cup \text{R}(Q') \cup \text{R}(R)$
$\text{VALUES}_{\mathfrak{M}}(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:vals}, \iota(\mathfrak{M})), (\iota(\cdot), \texttt{a}, \texttt{:Values})\} \cup \text{R}(Q') \cup \text{R}(\mathfrak{M})$
$\text{SELECT}_V(Q')$	$\{(\iota(\cdot), \texttt{:arg}, \iota(Q')), (\iota(\cdot), \texttt{:vars}, \iota(V)), (\iota(\cdot), \texttt{a}, \texttt{:Select})\} \cup \text{R}(Q') \cup \text{R}(V)$
(s,p,o)	$\{(\iota(\cdot), \texttt{:s}, \iota(s)), (\iota(\cdot), \texttt{:p}, \iota(p)), (\iota(\cdot), \texttt{:o}, \iota(o)), (\iota(\cdot), \texttt{a}, \texttt{:TP})\}$
(s,e,o)	$\{(\iota(\cdot), \texttt{:s}, \iota(s)), (\iota(\cdot), \texttt{:p}, \iota(e)), (\iota(\cdot), \texttt{:o}, \iota(o)), (\iota(\cdot), \texttt{a}, \texttt{:TP})\} \cup \text{R}(e)$
e	see Section 5.3
V	$\{(\iota(\cdot), \texttt{a}, \texttt{:VarSet})\} \cup (\bigcup_{v\in V}\{(\iota(\cdot), \texttt{:el}, \iota(v))\}$
X	$\{(\iota(\cdot), \texttt{a}, \texttt{:IriSet})\} \cup (\bigcup_{x\in X}\{(\iota(\cdot), \texttt{:el}, x)\}$
$\mathfrak{M}$	$\{(\iota(\cdot), \texttt{a}, \texttt{:SolBag})\} \cup (\bigcup_{\mu\in\mathfrak{M}}\{(\iota(\cdot), \texttt{:sol}, \iota((\mu, \mathfrak{M}(\mu))))\} \cup \text{R}((\mu, \mathfrak{M}(\mu))))$
(μ,k)	$\{(\iota(\cdot), \texttt{a}, \texttt{:Binding}), (\iota(\cdot), \texttt{:num}, \iota(k))\} \cup (\bigcup_{v\in\text{dom}(\mu)}\{(\iota(\cdot), \texttt{:el}, \iota((v,\mu(v))))\} \cup \text{R}((v,\mu(v)))$
(v,c)	$\{(\iota(\cdot), \texttt{:var}, \iota(v)), (\iota(\cdot), \texttt{:val}, \iota(c))\}$
R	see Section 5.2.2

In a similar manner as we defined for MQs in Section 5.1.2, we may extend the inverse mapping function $\text{R}^{-}()$ by applying the inverse of Table 5.3, where canonical blank nodes in RDF term or variable positions (specifically, the objects of triples in the r-graph with predicate `:subject`, `:predicate`, `:object`, or `:arg`) are mapped to canonical variables or blank nodes using a fixed, total, one-to-one mapping $\lambda^{-} : \mathbf{B} \to \mathbf{VB}$ [63].

The following subsections will show the representation of these features in greater detail.

Optional

Optional terms, otherwise known as left outer join operators, are represented by a blank node and two triples: one denoting the left operand, and the other the right operand. It is important to distinguish one from another because `OPTIONAL` is not a commutative property.

Figure 5.25 illustrates our representation of a query pattern Q of the form $Q = [Q_1 \text{ OPT } Q_2]$. Our representation can only provide a partial canonicalisation for this feature because `OPTIONAL` patterns can create unbound variables, and indirectly model negation. As a result, equivalence can become undecidable, as we will explain in Section 6.2.2.

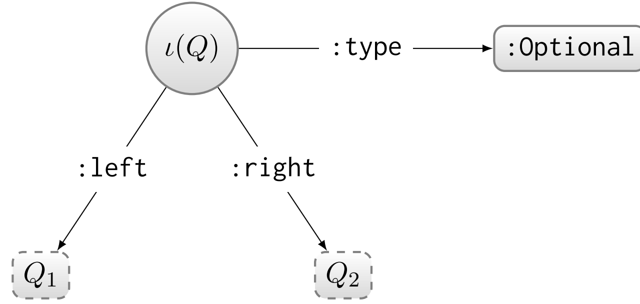


Figure 5.25: Visualisation of the graph representation of an `OPTIONAL` expression.

Built-in Expressions

In this section, we describe the graph representation of features that use built-in expressions. These are filter expressions, binding expressions, and aggregate expressions. Filters operators are more complicated to represent than binary operators such as joins or unions, because they contain terms that are essentially functions of various orders that may or not be commutative.

We recall that a term in **VIBL** is a *built-in expression*. If a built-in expression R is simply a term $R \in \mathbf{VIBL}$, then we use $\iota(R)$ to represent the expression, where $\mathrm{R}(R) = \emptyset$. Otherwise, if $R = \phi(R_1, \dots, R_n)$ where ϕ is a built-in function, then:

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

where $\iota(\phi)$ is an IRI that is assumed to uniquely identify the function, and $\iota(R)$, $\iota(R_i)$ are fresh blank nodes. If ϕ has more than one argument and is not commutative then we add a triple $(\iota(R_i), \mathtt{:ord}, \iota(i))$ for each R_i to its representation.

Analogous to how we represent `JOIN` and `UNION` expressions – an operator node connected to its operands by `:arg` edges – we represent the conjunctions (and) and disjunctions (or) of filter expressions. The challenge here is that these must also be normalised. The commutativity of "and" and "or" is captured in the same manner as `JOIN` and `UNION`, as explained in Section 5.1.1.

Built-in functions –such as `isBlank` and `isBound`– as well as equality or inequality symbols are represented as blank nodes connected by a `value:` predicate to a literal node containing the literal value of the function or symbol. This representation has been chosen because it makes the transformation back to a query much easier, but it may just as easily be represented by an IRI. Figure 5.26 shows the representation of a generic, ordered built-in function.

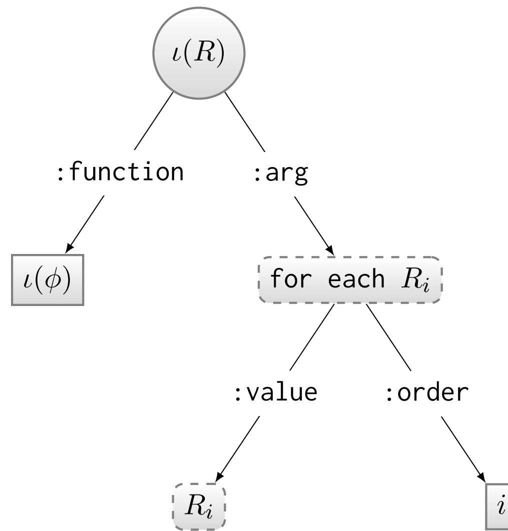


Figure 5.26: Visualisation of the graph representation of an ordered built-in function.

Filter

The most common use of built-in expressions is when said expression is declared with the `FILTER` keyword. In such a case, this is represented as shown in Figure 5.27.

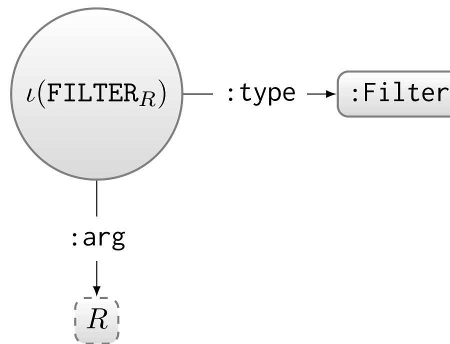


Figure 5.27: Visualisation of the graph representation of a `FILTER` expression.

However, two particular instances of `FILTER` expressions are those with the `EXISTS` and `NOT EXISTS` functions. These functions take query patterns instead of other built-in expressions, where the query patterns are represented in the standard way.

Binding Variables

The `BIND` operator allows for the assignment of built-in expressions to variables, which are then treated as part of the solution mappings of the query pattern they belong to. The example presented in Figure 5.29 shows the representation of a binding of a built-in expression R to a variable v.

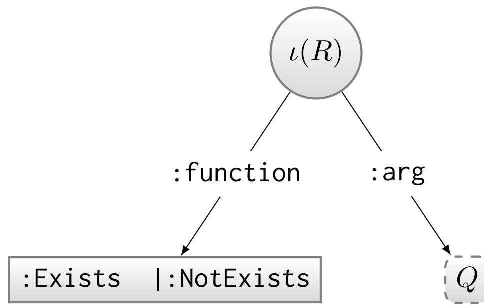


Figure 5.28: Visualisation of the graph representation of an `(NOT) EXISTS` function.

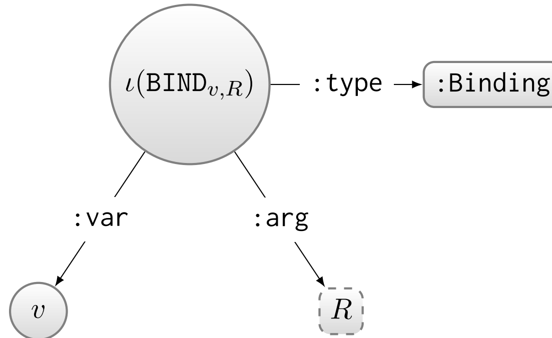


Figure 5.29: Visualisation of the graph representation of a `BIND` expression.

Aggregation

Figure 5.30 shows the representation of a query pattern that contains a `GROUP BY` and `HAVING` expression. We create a new `BIND` expression for every term stated in the `GROUP BY` expression. Each one of these expressions is bound to an anonymous variable that is only used internally. For instance, in the case of `GROUP BY ?actor, AVG(?salary)`, an expression `BIND(??1 AS ?actor)` and an expression `BIND(??2 AS AVG(?salary))` would be added. Each of these individual `BIND` expressions would be then handled as any built-in expression.

Solution Modifiers

As is only natural for applications that need to extract data, we may sometimes need to return results in either ascending or descending order. SPARQL provides the `ORDER BY` clause to return ordered results. Let $V = \{v_1, ..., v_n\}$ be the set of variables we want to order the results of a query by.

Figure 5.31 illustrates the representation of an `ORDER BY` clause. This r-graph is connected to the projection node of a query Q by a `:mod` edge. The dashed circle represents that there is a blank node for each variable in V. Each of these blank nodes has a literal node connected to it by an `:direction` edge that indicates if the variable is ordered by ascending or descending order; a variable node connected by an `:var` edge; and also a literal node that indicates the order priority of the variable.

In some cases, we may only want to return a certain number of results, or even return

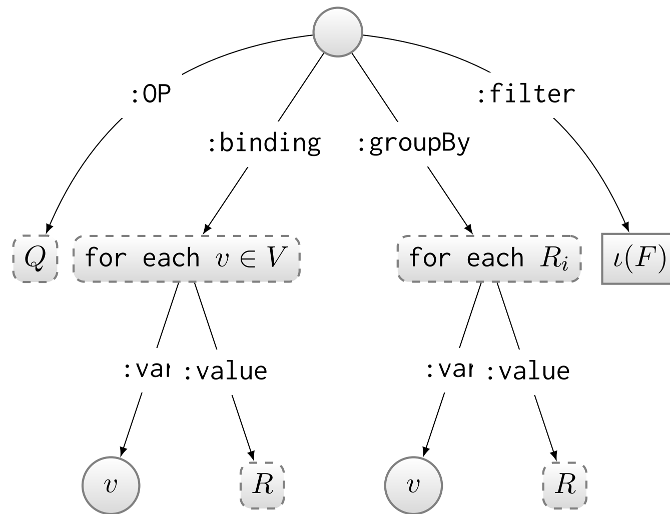


Figure 5.30: Visualisation of the graph representation of a graph pattern that contains a `FILTER` expression, a `BIND` expression, and a `GROUP BY` expression.

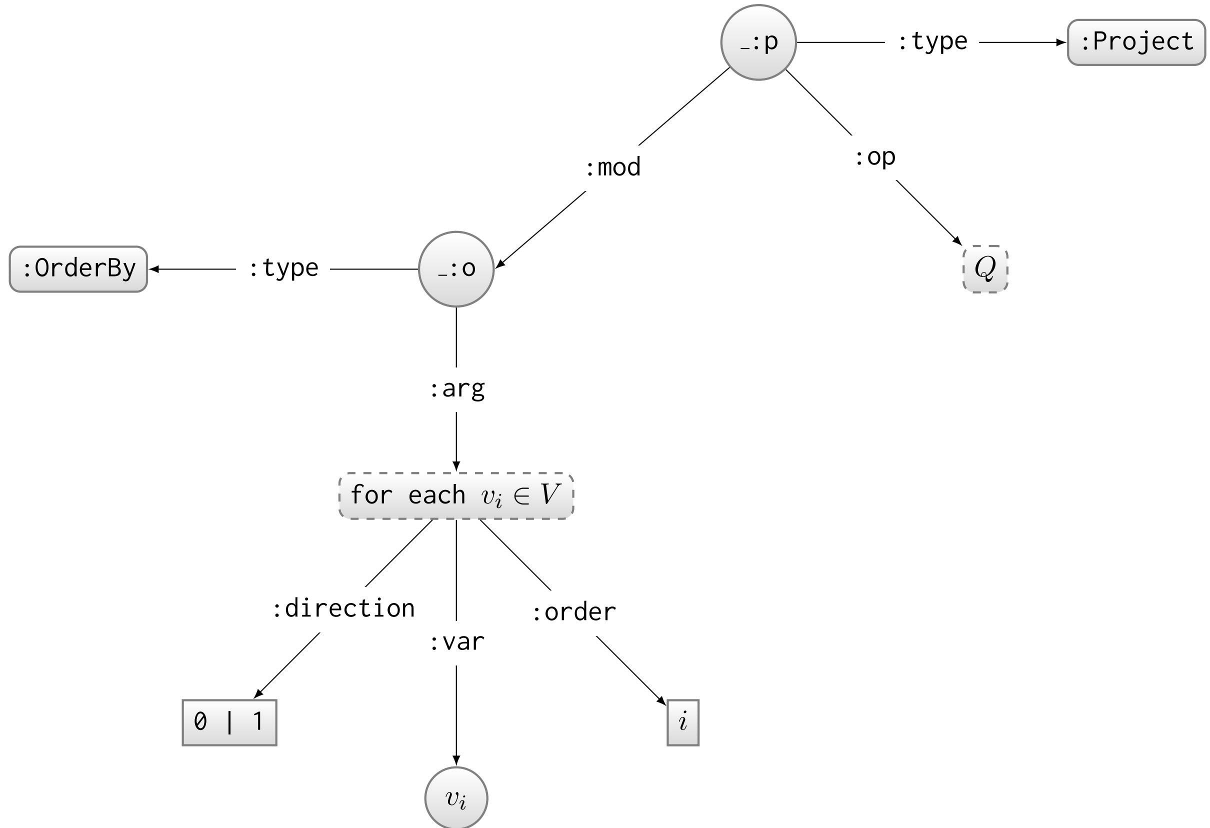


Figure 5.31: Representation of an `ORDER BY` clause.

results starting from a certain position. SPARQL provides the `LIMIT` and `OFFSET` keywords for each of the aforementioned cases, respectively. Both expressions are then (optionally) followed by an integer. If a `LIMIT` clause is stated while the `OFFSET` clause is omitted, then we return values from the initial position by default. On the other hand, if an `OFFSET` clause is stated while the `LIMIT` clause is omitted, then all results are returned starting from the

position given in the `OFFSET` clause.

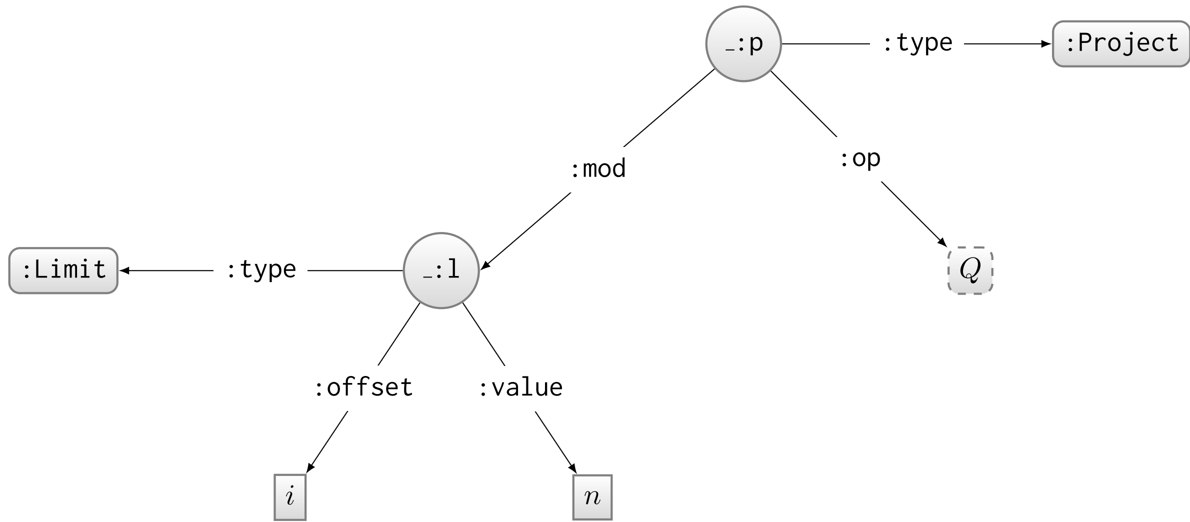


Figure 5.32: Representation of an `OFFSET` and `LIMIT` clause.

Figure 5.32 illustrates the representation of an `OFFSET` clause and a `LIMIT` clause. The r-graph is connected to the main query in an analogous manner to the one shown in Figure 5.31. It contains a literal node that corresponds to the value v in the `OFFSET` clause and another for the value n in the `LIMIT` clause.

Named Graphs

Assume we define a SPARQL dataset D which comprises both a set of graph names X_D referred to in `FROM` clauses, and a set of IRIs X_N mentioned in `FROM NAMED` clauses that denote named graphs. The default graph will be the result of the merge of the graphs mentioned in the `FROM` clauses. If there are none, the default graph is an empty graph. The query pattern described in Q will only match the graphs specified in X_N if the keyword `GRAPH` is used.

Figure 5.33 illustrates the representation of a query that contains both a `FROM` clause and a `FROM NAMED` clause. In this example, the `FROM NAMED` node is connected to the `FROM` node by an `:op` edge; if there is no `FROM NAMED` clause, then its node and its corresponding nodes will be missing; if there is no `FROM` clause, then the `FROM NAMED` clause will be connected to the projection node directly. Both the `FROM` and `FROM NAMED` nodes have an IRI node for each graph IRI contained in G_D and G_N, respectively.

Figure 5.34 illustrates the representation of a `GRAPH` clause. This r-graph is connected to either a blank node –if the graph is identified by a variable– or an IRI node –if the graph is a named graph IRI. It is also connected to a query Q that corresponds to the query which is to be matched to the graph pattern described in the `GRAPH` clause.

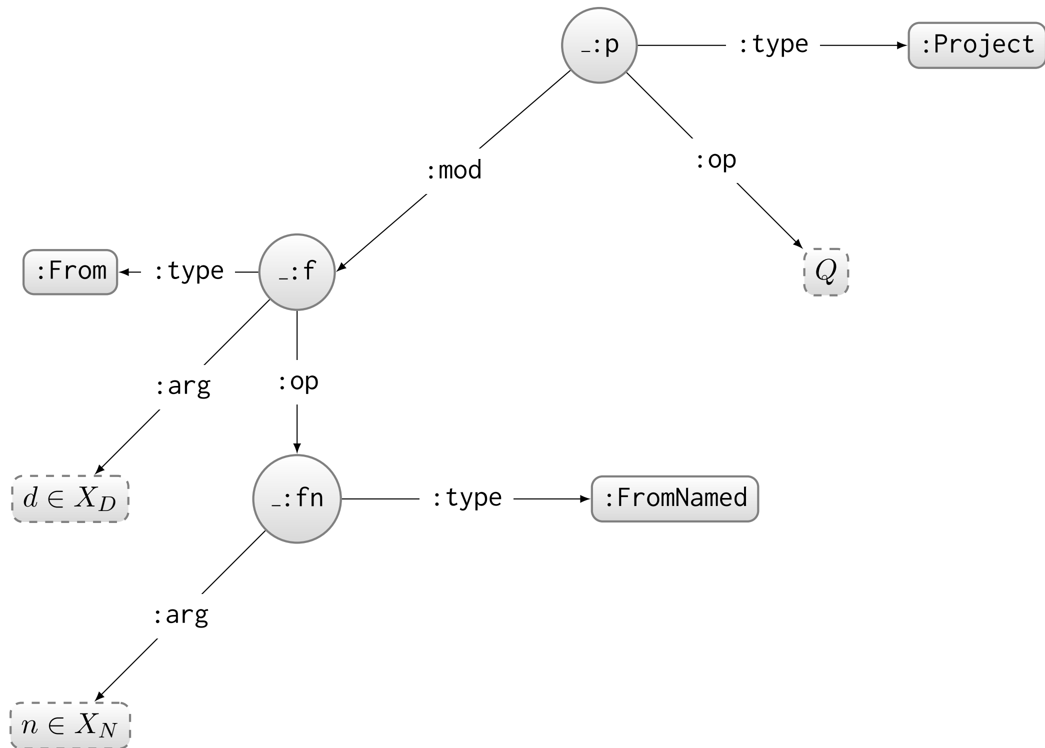


Figure 5.33: Representation of a `FROM` clause.

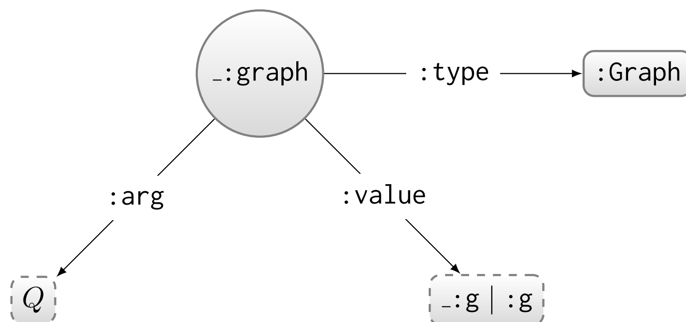


Figure 5.34: Representation of a `GRAPH` expression.

Inline Values

As stated previously, the VALUES keyword allows us to assign multiple values to one or more variables at once. We note once again that a set of solution mappings M may be visualised as a table where each row is a solution mapping μ; each column represents a variable v; and each cell contains the value of v for the solution mapping μ. We may represent this table and its (unordered) rows and columns as follows:

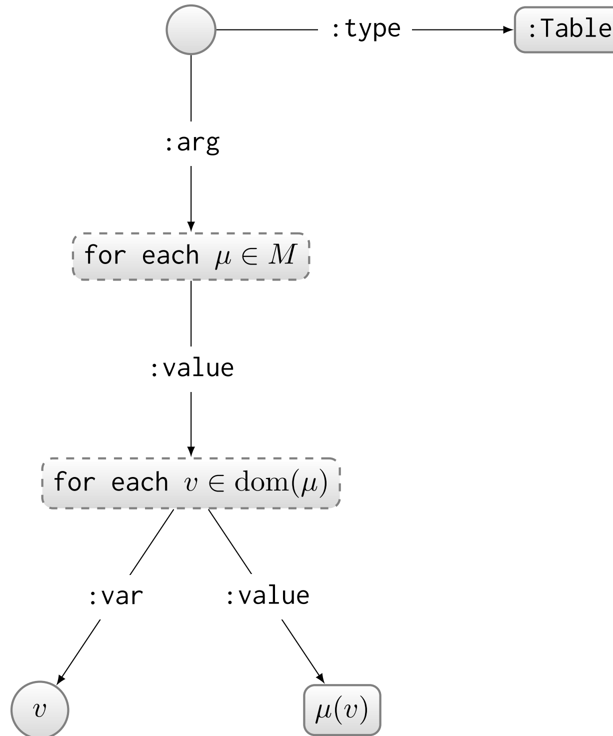


Figure 5.35: Visualisation of the graph representation of a VALUES expression.

5.3 Property Path Normalisation

An important part of this study has been to cover as many real-world SPARQL queries as possible. In order to do this, we need to identify the largest fragment of SPARQL such that our canonicalisation method is both sound and complete.

As we have shown in Table 4.9, decision problems such as the equivalence of (U)CPQs belong to the EXPSPACE-complete [44] complexity class, which means that it is a decidable problem, albeit a very complex one. Therefore, our goal would be to find a fragment for which sound and complete canonicalisation is not only possible, but also hopefully efficient for real-world queries.

Property paths are triples that allow expressions similar to regular expressions in the predicate positions. Due to this similarity, and because regular expressions and finite automata have the same expressivity, we have chosen to represent property paths as graphs

that resemble finite automata. This approach has the benefit of being able to apply known normal forms for finite automata, providing partial canonicalisation.

In this section, we describe a number of normalisations applied to property paths. These normalisations allow us to detect a slightly larger fragment of monotone queries with property paths, namely those that contain features that can be rewritten into UCQs.

Property Paths to UCQs

$(x, e_1/e_2/\dots/e_n, y)$	$\{(x, e_1, b_1), (b_1, e_2, b_2), \dots, (b_{n-1}, e_n, y)\}$
(x, e^-, y)	(y, e, x)
$(x, e_1\|\dots\|e_n, y)$	$\{(x, e_1, y)\}$ `UNION` ... `UNION` $\{(x, e_n, y)\}$

Table 5.4: Rewrite rules for property paths to UCQs

Table 5.4 contains rewrite rules used to transform property paths into equivalent UCQs. These rules rewrite any property paths with concatenation, union and inverses into UCQs. In fact, this is how the semantics of these features are defined in SPARQL (under bag semantics). When we are able to rewrite into UCQs, we can use use our existing algorithm to canonicalise the result. Evidently, the main limitation of this approach is being unable to remove redundant paths of zero/one or one/more. Furthermore, there is nothing that can be done for paths that contain expressions inside Kleene stars, etc (e.g. $((p/q)|r)^*$).

Inverse Normal Form

Inverse paths allow us to rewrite queries into other equivalent ones, sometimes as simply as swapping the positions of subject and object in path patterns (taken care of previously with rewrite rules). But other inverses might be inside other operators. Therefore, we need to find a way to normalise these queries despite the existence of inverse paths.

$(p^-)^-$	p
$(p^*)^-$	$(p^-)^*$
$(p_1/p_2)^-$	p_2^-/p_1^-
$(p_1\|p_2)^-$	$p_1^-\|p_2^-$

Table 5.5: Rewrite rules for property paths in *inverse normal form*

By following the rewrite rules described in Table 5.5 (which try to push inverses inwards to create inverses of IRIs only), we can transform any property path into an equivalent property path to its *inverse normal form*. This means that inverse operators ($^-$) will only appear next to IRIs, and therefore all property paths are operations (concatenation, union, Kleene star) on IRIs or their inverse.

Property Path Normal Form

Given an property path, we construct a *non-deterministic finite automaton* (NFA) based on Thompson's algorithm for transforming regular expressions to NFAs [73]. We then convert this NFA into a *deterministic finite automaton* (DFA) by a standard superset expansion; this transformation is exponential on the number of states of the NFA. Finally, we perform a minimisation of the DFA using Hopcroft's algorithm [39], which produces a minimal DFA such that all regular expressions that express the same language will produce the same DFA. We provide a simple example to illustrate this process.

Example 5.3.1. Consider an RPQ $:\texttt{p}^*/:\texttt{p}^*/:\texttt{p}^*$. In Figure 5.36 we provide an example of the corresponding NFA produced by Thompson's algorithm, the DFA produced by superset expansion, and the minimal DFA produced by Hopcroft's algorithm.

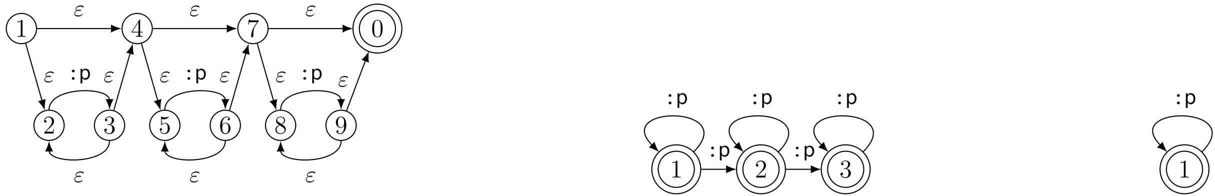


Figure 5.36: NFA, DFA and minimal DFA produced for the RPQ $:\texttt{p}^*/:\texttt{p}^*/:\texttt{p}^*$

We use the resulting DFA as the r-graph representation of the property path, thus capturing additional equivalences and congruences.

The normalisation techniques described in this section apply to property paths, but may miss equivalences of (U)C(2)RPQs due to the multiple ways in which such queries can be equivalently expressed, and in particular due to the interaction between similar features of (U)CQs and (2)RPQs. We will discuss this issue in more detail in Chapter 7, where we describe our efforts to push the bounds of canonicalisation towards covering UC2RPQs.

Chapter 6

Soundness and Completeness

In this chapter, we prove the soundness, completeness and complexity of our method for monotone queries. Subsequently, we show that for the full language soundness is preserved, but completeness is not. These results and proofs have been published in our study [64].

6.1 Monotone Queries

In this section, we present the proofs for the soundness and completeness of our method for monotone queries. We recall that this fragment corresponds to queries that contain only projection, joins, and unions.

6.1.1 Soundness

Given a monotone query Q, we denote by $\mathrm{U}(Q)$ the process described in Section 5.1.1 involving the application of:

1. union normalisation
2. unsatisfiability normalisation
3. variable normalisation
4. set vs. bag normalisation

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

Lemma 6.1.1. For an MQ Q, it holds that:

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

Proof. This follows from the fact that each step shown in Algorithm 1 preserves the congruence of Q, as follows:

1. union normalisation: proven by Pérez et al [56];
2. unsatisfiability normalisation: Lemmas 5.1.3, 5.1.4;
3. variable normalisation: Lemmas 5.1.1, 5.1.2;
4. set vs. bag normalisation: Lemmas 5.1.5, 5.1.6.

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

Following this, we show that the canonical labeling of blank nodes, denoted by $\textsc{l}(G)$ for an RDF graph G, also preserves congruence:

Lemma 6.1.2. Given a UCQ Q, it holds that:

$$\textsc{r}^{-}(\textsc{l}(\textsc{r}(Q))) \cong Q\,.$$

Proof. Note that queries Q_1 and Q_2 are said to be *isomorphic* (denoted as $Q_1 \simeq Q_2$) if there exists a one-to-one variable mapping $\rho : \mathbf{V} \to \mathbf{V}$ such that $\rho(Q_1) = Q_2$, and is the identity on projected variables. Then, by definition, we have that $\textsc{r}^{-}(\textsc{r}(Q)) = Q$, and thus $\textsc{r}^{-}(\textsc{r}(Q)) \simeq Q$, where $\textsc{r}^{-}(\cdot)$ relies on a one-to-one mapping of blank nodes to variables (namely ξ). Recalling that $\textsc{l}(\cdot)$ (described in Section 2.2.4) performs a one-to-one mapping of blank nodes to blank nodes in $\textsc{r}(Q)$, thus producing an isomorphic graph to $\textsc{r}(Q)$, we can conclude that $\textsc{r}^{-}(\textsc{l}(\textsc{r}(Q)))$ produces a query that is isomorphic to $\textsc{r}^{-}(\textsc{r}(Q))$. Hence we have that $\textsc{r}^{-}(\textsc{l}(\textsc{r}(Q))) \simeq \textsc{r}^{-}(\textsc{r}(Q)) \simeq Q$. Further given that isomorphism implies congruence, we have that $\textsc{r}^{-}(\textsc{l}(\textsc{r}(Q))) \cong \textsc{r}^{-}(\textsc{r}(Q)) \cong Q$. The result then holds per the transitivity of congruence. □

Finally we prove that the minimisation of UCQs through their r-graphs preserves congruence. Given a UCQ Q being evaluated under set semantics (with distinct), we denote by $\textsc{m}(Q)$ the result of minimising the UCQ, involving the two procedures:

1. BGP minimisation (Section 5.1.3);
2. union minimisation (Section 5.1.3).

Given a UCQ Q being evaluated under bag semantics (without distinct), we define that $\textsc{m}(Q) = Q$. If bag semantics is selected, the UCQ can only contain a syntactic form of redundancy: exact duplicate triple patterns in the same BGP, which are implicitly removed since we model BGPs as sets of triple patterns. Any other form of redundancy mentioned previously – be it within or across BGPs – will affect the multiplicity of results [19]. Hence if bag semantics is selected, we do not apply any redundancy elimination other than removing duplicate triple patterns in BGPs.

Lemma 6.1.3. Given a UCQ Q, it holds that:

$$\text{R}^-(\text{M}(\text{R}(Q))) \cong Q\,.$$

Proof. We only consider set semantics, because under bag semantics, UCQs are not minimised.

Set semantics: Minimising CQs by computing their cores is a well-known technique based on the idea that two CQs are equivalent if and only if they are homomorphically equivalent (with corresponding projected variables [18]). Likewise the minimisation of UCQs is covered by Sagiv and Yannakakis [62], who (unlike in the relational algebra but analogous to SPARQL) allow UCQs with existential variables; however, their framework assumes that each CQ covers all projected variables. Hence the only gap that remains is the minimisation of SPARQL UCQs where BGPs may not contain all projected variables. This result is quite direct since for a set of variables V, two BGPs Q_1 and Q_2 such that $\text{vars}(Q_1) \cap V \neq \text{vars}(Q_2) \cap V$, and any dataset D, it holds that $\text{SELECT}_V(Q_1)(D) \cap \text{SELECT}_V(Q_2)(D) = \phi$, per the same arguments used to prove Lemma 5.1.5. Hence checking containment within the partitions formed by the projected variables they contain does not miss containments. The result for set semantics then follows from Sagiv and Yannakakis [62]. □

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

Theorem 6.1.1. For an MQ Q, it holds that:

$$\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q))))) \cong Q\,.$$

Proof. The result holds as a direct corollary of Lemmas 6.1.1, 6.1.2, 6.1.3, and the transitivity of congruence, which is an equivalence relation. □

6.1.2 Completeness

In order to prove the completeness of our method for MQs, we need to show that $Q_1 \cong Q_2$ if and only if $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_1))))) = \text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_2)))))$.

First, the claim that $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_1))))) = \text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_2)))))$ implies $Q_1 \cong Q_2$ follows almost directly from the soundness result, which tells us that $Q_1 \cong \text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_1)))))$ and $Q_2 \cong \text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_2)))))$. Evidently, this equality relation implies congruence, and by transitivity, we may conclude that $Q_1 \cong Q_2$.

Therefore, to conclude that our method is complete for MQs, we must prove that $Q_1 \cong Q_2$ implies $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_1'))))) = \text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_2')))))$. First we note that if $Q_1 = Q_2$, then the result holds as each step in $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(\cdot)))))$ is deterministic, so we must instead look at the cases where $Q_1 \neq Q_2$ but $Q_1 \cong Q_2$.

We outline useful relations to prove the completeness of our method for MQs.

Lemma 6.1.4. The following hold:

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

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

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

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

Lemma 6.1.5. Let Q_1 and Q_2 denote UCQs. If Q_1 and Q_2 are unsatisfiable (which implies $Q_1 \cong Q_2$), then:

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

Proof. If Q_1 or Q_2 is unsatisfiable, and $Q_1 \cong Q_2$, this implies that both Q_1 and Q_2 are unsatisfiable. As a result of Lemmas 5.1.3, 5.1.4, $\text{U}(Q_1) = Q_\emptyset$ and $\text{U}(Q_2) = Q_\emptyset$, recalling that $Q_\emptyset$ denotes the canonical unsatisfiable query. Thus $\text{U}(Q_1) = \text{U}(Q_2)$ and the result holds per premise 2 of Remark 6.1.4 since equality implies isomorphism. □

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

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

Lemma 6.1.6. Let Q_1 and Q_2 denote satisfiable BGPs and V_1 and V_2 sets of variables. Further let $Q_1' = \text{DISTINCT}(\text{SELECT}_{V_1}(Q_1))$ and likewise let $Q_2' = \text{DISTINCT}(\text{SELECT}_{V_2}(Q_2))$. If $Q_1' \cong Q_2'$ then

$$\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_1'))))) = \text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_2'))))) \,.$$

Proof. Since Q_1' and Q_2' are satisfiable, and evaluated under set semantics, we know from the result of Chandra and Merlin [18] that $Q_1' \equiv Q_2'$ if and only if both are homomorphically equivalent with respect to a homomorphism that is the identity on projected variables.

Now $\text{M}(\text{R}(\text{U}(Q'_1)))$ computes the core of each BGP, which is known to be unique modulo isomorphism [34]. Thus if $Q'_1 \equiv Q'_2$ and both are satisfiable, we have that $\text{M}(\text{R}(\text{U}(Q'_1))) \simeq \text{M}(\text{R}(\text{U}(Q'_1)))$. On the other hand, if $Q'_1 \cong Q'_2$, we know that there exists a variable renaming such that $\rho(Q'_1) \equiv Q'_2$; combining this with the fact that $\text{M}(\text{R}(\text{U}(Q'_1))) \simeq \text{M}(\text{R}(\text{U}(\rho(Q'_1))))$, and the fact that congruence is an equivalence relation, we know that $Q'_1 \cong Q'_2$ implies $\text{M}(\text{R}(\text{U}(Q'_1))) \simeq \text{M}(\text{R}(\text{U}(Q'_2)))$. The result then holds from premise 4 of Remark 6.1.4. □

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

Lemma 6.1.7. Let Q_1 and Q_2 denote satisfiable BGPs and V_1 and V_2 sets of variables. Further let $Q'_1 = \text{SELECT}_{V_1}(Q_1)$ and $Q'_2 = \text{SELECT}_{V_2}(Q_2)$. If $Q'_1 \cong Q'_2$ then

$$\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q'_1))))) = \text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q'_2))))) .$$

Proof. Given a satisfiable BGP Q, and $\text{SELECT}_V(Q)$ such that $V \setminus \text{vars}(Q) \neq \emptyset$, then $\text{U}(\text{SELECT}_V(Q))$ will remove the unbound variables ($V \setminus \text{vars}(Q)$) from V per Lemma 5.1.2, and therefore
$\text{U}(\text{SELECT}_V(Q)) = \text{U}(\text{SELECT}_{V \cap \text{vars}(Q)}(Q))$. As part of $\text{U}(Q)$, during union normalisation, blank nodes are rewritten to variables. This leaves us with cases where $V \subseteq \text{vars}(Q)$ and Q does not contain blank nodes.

If $V_1 = \text{vars}(Q_1)$, then Q'_1 cannot return duplicates (Lemma 5.1.5), and since $Q'_1 \cong Q'_2$, then Q'_2 cannot return duplicates, and thus $V_2 = \text{vars}(Q_2)$. Thus $\text{U}(Q'_1)$ and $\text{U}(Q'_2)$ will add distinct in both cases, and the result follows from Lemma 6.1.6.

This leaves us with the case that $V_1 \subsetneq \text{vars}(Q_1)$. In this case, Q'_1 will return duplicates for certain datasets (Lemma 5.1.5), and must be evaluated under bag semantics. Given that $Q'_1 \cong Q'_2$, this likewise means that Q'_2 returns duplicates. Under bag semantics, and assuming that Q'_1 and Q'_2 are satisfiable, then Theorem 5.2 of Chaudhuri and Vardi [19] tells us that $Q'_1 \equiv Q'_2$ if and only if $Q'_1 \simeq Q'_2$ with an isomorphism that is the identity on projected variables. Noting that $Q'_1 \cong Q'_2$ implies that there exists a variable renaming ρ such that $\rho(Q'_1) \equiv Q'_2$, it follows that there exists ρ such that $\rho(Q'_1) \simeq Q'_2$, or put more simply, that $Q'_1 \simeq Q'_2$ (without the restriction on projected variables). The result then follows from premise 1 of Remark 6.1.4. □

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

Lemma 6.1.8. Let Q_1 and Q_2 denote satisfiable UCQs with distinct. If $Q_1 \cong Q_2$ then

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

Proof. We show that given a satisfiable UCQ Q evaluated under set semantics, the minimisation function $\text{M}(Q)$ will produce an r-graph corresponding to a minimal UCQ that is unique, modulo isomorphism, for the set of UCQs congruent to Q. The minimisation of CQs (i.e., unary UCQs) is covered by Lemma 6.1.6. The minimisation of (non-unary) UCQs is based on Corollary 4 of Sagiv and Yannakakis [62], where we minimise the UCQ while maintaining this equivalence relation, more specifically, such that each BGP in the input UCQ

will be contained in some BGP of the output UCQ (with a containment homomorphism that is the identity on projected variables). Note that this minimisation includes the removal of all unsatisfiable BGPs (per Lemma 5.1.4; if all are removed, then Lemma 6.1.5 applies as the UCQ is unsatisfiable). There are, however, two non-deterministic elements to consider in this minimisation:

- The containment only considers projected variables as fixed. Hence the naming of other variables (and blank nodes) in BGPs does not matter. However, this issue is resolved prior to minimisation by $\text{U}(\cdot)$, which maps blank nodes to fresh (non-projected) variables, and then renames non-projected variables in each BGP to fresh variables, per Lemma 5.1.1, such that the naming of variables is deterministic modulo isomorphism.
- We non-deterministically choose one BGP from each quotient set of equivalent BGPs with the same projected variables. However, since BGPs were previously minimised, all equivalent BGPs are isomorphic, and hence the choice is deterministic modulo isomorphism.

Thus, given a satisfiable UCQ Q evaluated under set semantics, $\text{M}(Q)$ will produce an r-graph corresponding to a minimal UCQ that is unique, modulo isomorphism, for the set of UCQs congruent to Q. Returning to the claim, we note that Q_1 and Q_2 are satisfiable UCQs evaluated under set semantics (with distinct), that $Q_1 \cong Q_2$, and thus we have that $\text{M}(Q_1) \simeq \text{M}(Q_2)$: the result holds per premise 4 of Remark 6.1.4. □

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

Lemma 6.1.9. Let Q_1 and Q_2 denote satisfiable UCQs without distinct. If $Q_1 \cong Q_2$ then

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

Proof. Under bag semantics, observe that $\text{U}(\cdot)$ removes all unsatisfiable operands from the UCQ and $\text{M}(\cdot)$ acts as the identity (no minimisation is applied). Per the results for CQs, any minimisation of BGPs (aside from the implicit removal of duplicate triple patterns) will reduce the multiplicity of results on some datasets. Likewise removing satisfiable BGPs will reduce the multiplicities for any dataset where the removed BGP generates solutions. This leaves one source of non-determinism (the same as in the case for set semantics) per Lemma 5.1.1: that non-projected variables across BGPs may have the same label whereas the query is equivalent if they have distinct labels. As before for set semantics, $\text{U}(\cdot)$, maps blank nodes to fresh (non-projected) variables in the case of bag semantics. This implies that if $Q_1 \cong Q_2$ then – letting Q_1' and Q_2' denote the result of removing unsatisfiable BGPs from Q_1 and Q_2 and distinguishing variables per Lemma 5.1.1 – $Q_1' \simeq Q_2'$.

There are then two cases to consider:

1. Either Q_1 or Q_2 cannot return duplicates, per the conditions of Lemma 5.1.6, but given that $Q_1 \cong Q_2$, then this implies that both Q_1 and Q_2 cannot return duplicates, which means that both satisfy the conditions of Lemma 5.1.6, and thus both will have distinct invoked by $\text{U}(\cdot)$.

2. Both Q_1 and Q_2 may return duplicates, i.e., they do not satisfy the conditions of Lemma 5.1.6, and in neither case will distinct be invoked by $\textsc{u}(\cdot)$.

Thus we can conclude that $\textsc{u}(Q_1) \simeq \textsc{u}(Q_2)$, satisfying premise 2 of Remark 6.1.4, from which the result follows. □

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

Lemma 6.1.10. Let Q denote a satisfiable UCQ without distinct. Let $Q' = \textsc{distinct}(Q)$. If $Q \cong Q'$, then:

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

Proof. Since Q cannot return duplicates and $Q \cong Q'$, it holds that Q' cannot return duplicates, and hence it must satisfy the conditions of Lemma 5.1.5. Thus $\textsc{u}(Q')$ will add distinct, and we have that $\textsc{u}(Q) \simeq \textsc{u}(Q')$. The result then follows per premise 2 of Remark 6.1.4. □

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

Theorem 6.1.2. Given two MQs Q_1 and Q_2, if $Q_1 \cong Q_2$ then

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

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

1. If Q_1' or Q_2' are unsatisfiable, then the result holds from Lemma 5.1.4.
2. Otherwise (Q_1' and Q_2' are satisfiable):
   (a) If Q_1' and Q_2' use distinct, the result holds from Lemma 6.1.8.
   (b) If neither Q_1' nor Q_2' use distinct, the result holds from Lemma 6.1.9.
   (c) If Q_1' uses distinct, and Q_2' does not, then $Q_1' \cong Q_2'$ implies that Q_2' cannot produce duplicates (since Q_1' cannot). From this it follows that $Q_1' \cong Q_2' \cong \textsc{distinct}(Q_2')$. Now given that $Q_1' \cong \textsc{distinct}(Q_2')$, it follows from Lemma 6.1.8 that $\textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q_1'))))) = \textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(\textsc{distinct}(Q_2'))))))$ (noting that Q_1' and $\textsc{distinct}(Q_2')$ use distinct). Further given that $Q_2' \cong \textsc{distinct}(Q_2')$, it follows from Lemma 6.1.10 that $\textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q_2'))))) = \textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(\textsc{distinct}(Q_2'))))))$. We then have $\textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q_1'))))) = \textsc{r}^{-}(\textsc{l}(\textsc{m}(\textsc{r}(\textsc{u}(Q_2')))))$.
   (d) Otherwise, if Q_1' does not use distinct, and Q_2' uses distinct, the result follows from the previous case and the symmetry of congruence.

This concludes the proof. □

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

Theorem 6.1.3. Given two EMQs Q_1 and Q_2, it holds that $Q_1 \cong Q_2$ if and only if

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

Proof. Let Q_1' denote $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_1)))))$, and Q_2' denote $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(Q_2)))))$.

$Q_1' = Q_2'$ implies $Q_1 \cong Q_2$: follows from Theorem 6.1.1 (soundness), which tells us that $Q_1 \cong Q_1'$ and $Q_2' \cong Q_2$, from which we have that $Q_1 \cong Q_1' = Q_2' \cong Q_2$, and thus that $Q_1 \cong Q_2$.

$Q_1 \cong Q_2$ implies $Q_1' = Q_2'$: is given in Theorem 6.1.2 (completeness). ☐

6.1.3 Complexity

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

With respect to the complexity of the algorithm $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{U}(\cdot)))))$, for simplicity we will assume as input an MQ Q such that all projected variables are contained in the query[1], which will allow us to consider the complexity at the level of triple patterns. We will denote by n the number of triple patterns in Q.

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

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

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

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

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

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

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

With respect to $\mathrm{M}(\cdot)$, first we consider BGP minimisation, which requires computing the core of each BGP's r-graph G. Letting j denote the number of unique subject and objects in G being minimised, which is also an upper bound for the number of blank nodes, we will assume a brute-force $O(j^j)$ algorithm that searches over every mapping of blank nodes to terms in G, looking for the one that maps to the fewest unique terms (this mapping indicates the core [38]). Note that the number of triples in the r-graph for each BGP is bounded by $O(n)$, and so is the number of unique subjects and objects. Furthermore, the number of BGPs is bounded by $O(3^{n/3})$. Thus the cost of minimising all BGPs is $O(3^{n/3}n^{cn})$ for some constant $c > 1$. We must also check containment between each pair of BGP r-graphs (G', G'') in order to apply UCQ minimisation. Again, assuming the number of subjects and objects in $G' \cup G''$ to be bounded by j, we can assume a brute-force $O(j^j)$ algorithm that considers all mappings. Given $O(3^{n/3})$ BPGs, we have $O((3^{n/3})^2) = O(3^{2n/3})$ pairs of BGPs to check, giving us a cost of $O(3^{2n/3}n^{cn})$. Adding both BGP and UCQ minimisation costs, we have $O(n^{cn}(3^{n/3} + 3^{2n/3})) = O(n^{cn}3^{2n/3})$ for the $\mathrm{M}(\cdot)$ step in $\mathrm{M}(\mathrm{R}(\mathrm{U}(\cdot)))$. We can then reduce $O(n^{cn}3^{2n/3})$ to $O(2^{cn \log n})$ by converting both bases to 2 and removing the constant factors.[2]

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

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

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

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

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

Overall, this complexity assumes worst cases that we expect to be rare in practice, and our analysis assumes brute-force methods for finding homomorphisms, computing cores, labelling blank nodes, etc., whereas we use more optimised methods. For example, the exponentially-sized UCQ r-graphs form a tree-like structure connecting each BGP, where it would be possible to canonically label this structure in a more efficient manner than suggested by this

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

worst-case analysis. Thus, though the method has a high computational cost, this does not necessarily imply that it will be impractical for real-world queries. Still, we can conclude that the difficult cases for canonicalisation are represented by input queries with joins of unions, and that minimisation and canonical labelling will likely have high overhead. We will discuss this further in the context of experiments presented in Chapter 9.

6.2 SPARQL 1.1

In this section, we present proof for the soundness of our method for the entirety of SPARQL 1.1. In addition, we provide proof for the *incompleteness* of this method for the full language.

6.2.1 Soundness

Next we show that the process is sound for queries in the full SPARQL 1.1 query language. We use $\text{A}(\cdot)$ to denote the application of filter normalisation (Section 5.2.1), local variable normalisation (Section 5.2.1), and property path normalisation (Section 5.3), in addition to the UCQ normalisation.

Lemma 6.2.1. For a SPARQL 1.1 query Q, it holds that:

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

PROOF. We show that each step preserves congruence.

In $\text{A}(Q)$ we apply filter normalisation (Section 5.2.1), local variable normalisation (Section 5.2.1), and property path normalisation (Section 5.3), in addition to the UCQ normalisation. Each step preserves query equivalence, and thus congruence.

Next we compute the r-graph, and apply minimisation. However, if the query is not in UCQ normal form, we only apply minimisation on BGPs and unions of BGPs (UBGPs) contained in the query, considering any variables external to the (U)BGP as being "projected", and thus fixed. More formally, taking a UBGP Q' inside a larger query Q, and letting V' denote the variables of Q used both inside and outside Q, observe that we can replace Q' with $\text{SELECT}_{V'}(Q')$ inside Q without changing the semantics of Q as variables not in V' are not used elsewhere in Q, and since Q is a query, it must contain a SELECT, ASK, CONSTRUCT or DESCRIBE clause, whose results will not change if a variable not mentioned in the clause is projected away. Now since $\text{SELECT}_{V'}(Q')$ is a UCQ, the minimisation process preserves congruence per Theorem 6.2.1.

Regarding the r-graph, we assume that this (and the inverse r-mapping) has been defined such that $\text{R}^{-}(\text{R}(Q)) = Q$, with the exception of property paths, but in this case it is clear that $\text{R}^{-}(\text{R}(Q)) \cong Q$ since the r-graph representation of RPQs relies on well-known automata techniques – Thompson's construction, subset expansion, Hopcroft's algorithm and state elimination – that will produce an equivalent RPQ (similar automata-based techniques were also used by Kostylev et al. [44] for their analysis of the containment of property paths).

Thus $\text{R}^-(\text{L}(\text{M}(\text{R}(\text{A}(Q))))) \cong \text{R}^-(\text{M}(\text{R}(\text{A}(Q)))) \cong \text{R}^-(\text{R}(\text{A}(Q))) \cong \text{R}^-(\text{R}(Q)) \cong Q$, from which the result holds. □

6.2.2 Incompleteness

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

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

Example 6.2.1. Consider the following example:

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

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

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

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

This query is unsatisfiable because predicates must be IRIs, and IRIs must always contain a colon (to separate the scheme from the hierarchical path) [30].

Next we consider issues with property paths.

Example 6.2.2. Consider the following example:

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

Clearly this is equivalent to:

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

But also to:

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

Currently we rewrite concatenation, inverse and disjunction in paths (not appearing within a recursive expression) to UCQ features. This means that we currently capture equivalence between the first and second query, but not the first and third. We will discuss this in more detail in Chapter 7.

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

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

This is equivalent to:

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

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

Incompleteness can also occur due to negation, which is complicated by the ambiguities surrounding `NOT EXISTS`[64]. We have postponed algebraic rewritings involving negation until this issue is officially resolved.

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

Example 6.2.3. Consider the query Q_1:

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

Since `AND` is commutative, this is equivalent to Q_2:

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

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

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

and, analogously, for Q_2:

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

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

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

1. Use query logs or other empirical methods to determine more common cases that this query canonicalisation framework may miss and implement targeted methods to deal with such cases.

2. Extend the query fragment for which sound and complete canonicalisation is possible; an interesting goal, for example, would be to explore MQs with full property paths (such queries are similar to C2RPQs [44], for which containment and related problems are decidable). We will discuss work in this direction in Chapter 7.

6.2.3 Complexity

In terms of the complexity of (partially) canonicalising queries with these additional features, if we assume that the size of the input query Q is in $O(n)$, where n is the number of unique triple patterns and path patterns in Q, then the complexity remains bounded by that of canonicalising monotone queries: $O((cn3^{n/3})!)$ for some constant $c > 0$. This assumes that features of the query that do not involve triple or path patterns (e.g., `BIND`, `VALUES`, `FILTER`, etc.) are of bounded size and bounded in number, while path expressions are of bounded length. This may be an oversimplification.

We may rather consider the "token length" of the query, which is the number of terminals – RDF terms, variables, keywords, etc. – appearing in the syntax of the query. In this case, we must additionally consider the costs of computing a normal form for RPQs. Given an RPQ of length l, which includes the number of non-parenthetical symbols (including IRIs, `*`, `+`, `|`, `/`) Thompson's construction creates an NFA of size $O(l)$. Subset expansion may then create a DFA with $O(2^l)$ states that remains exponential after minimisation. This will result in an exponentially-sized representation of the RPQ.

Minimisation of DFAs is achieved using Hopcroft's algorithm [39], which partitions states into equivalence classes based on their behaviour for input sequences. This algorithm has a worst-case scenario of $O((2^l)s\log(2^l))$ on a DFA with $O(2^l)$ states and an alphabet of size s. However, the size of the alphabet is bound by the length of the RPQ l, so this complexity is reduced to $O(2^l l^2)$.

Furthermore, canonical labelling will occur on this representation, where assuming again a brute-force method in the order of $O(b!)$ for b the number of blank nodes, we now have a complexity of $O(2^l!)$ for canonicalising the RPQ representation graph of a property path of length l, and given that we may have n such property paths, where n is the number of triple patterns and path expressions, this generates a cost of $O((cn2^l)!)$ for canonically labelling a navigational graph pattern, which we can add to the cost for monotone queries: $O((cn2^l)! + (cn3^{n/3})!)$, where l is the length of the longest property path, n is the number of triple patterns and path patterns, and $c > 0$ is some constant to account for the additional "syntactic" blank nodes that appear in the r-graph.

Chapter 7

Towards UC2RPQs

In this chapter we discuss some techniques that we have explored in order to compute a canonical form for UC2RPQs in a sound and complete way. This would allow us to canonicalise (most) BGPs with property paths, which are common in real-world queries [13]. We know that containment (and thus equivalence) is decidable for this fragment [3, 16], and so it may be possible to design a procedure to compute a canonical form of queries in this fragment. However, we have not yet achieved this goal. This chapter sketches some canonicalisation techniques for UC2RPQs that we have explored, how they are incomplete, and what appear to be the difficult cases. A sound and complete canonicalisation procedure for this fragment remains an open question that would be interesting to address in future work.

7.1 Normalisation

The fact that certain property paths may be rewritten into equivalent monotone queries presents us with a dilemma: should we rewrite property paths into monotone queries, thereby producing queries with more graph patterns (expansion); or should we rewrite monotone queries into equivalent property paths, which results in fewer graph patterns with more complex property paths (collapse)? We'll take Figure 7.1 as a running example. We assume that only ?x is projected under set semantics (denoted by underlining).

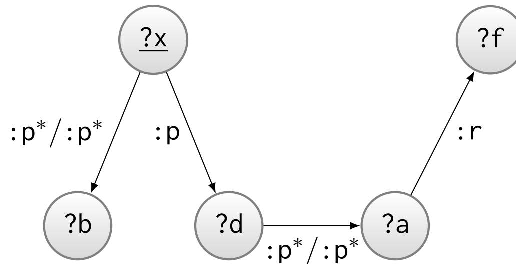


Figure 7.1: Example of a path pattern.

7.1.1 Expansion

Expansion refers to rewriting property paths into equivalent basic graph patterns or path patterns with Kleene stars according to the rules we have presented in Section 5.3. We show the result of expansion of the running example in Figure 7.2 where we replace concatenations in path expressions with intermediate variables. This form is useful for getting rid of the redundant edge from variable ?x to variable ?b (we have code for computing the core to perform this type of minimisation), but not for identifying that `:p`*/`:p`* can be simplified to `:p`*, and thus that the variables `?c` and `?e` we just created is redundant.

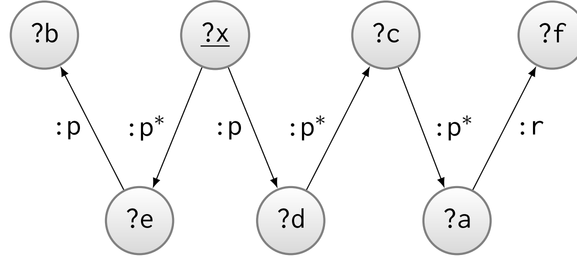


Figure 7.2: The result the expansion of the pattern in Figure 7.1.

7.1.2 Collapse

On the other hand, we may choose to reduce basic graph patterns to equivalent path patterns, or concatenate different path patterns into a single, larger path pattern, when possible. For instance, if we have (`?x`, p_1, `?y`) and (`?y`, p_2, `?z`) as path patterns, we may rewrite both into a single path pattern (`?x`, p_1/p_2, `?z`). We refer to this as *collapsing* patterns. To do this, we define *path variables* as non-distinguished (not projected, not used elsewhere) variables with a degree of 2 (i.e. it only appears twice in the same graph pattern) such as `?y` in the previous example. A path variable can be rewritten into a concatenation of a single path pattern (`?x`, p_1/p_2, `?z`). By doing this, we effectively remove `?y` from the query without affecting the results of the query pattern. This is known as *collapsing*. We show the result of collapsing the running example in Figure 7.3. If we collapse and then apply the automata-based property path normal form discussed in Section 5.3, this is great for normalising the path `:p/:p`*`/:p`*`/:r` to something like `:p/:p`*`/:r`. But, unlike in the case of the expansion, when we try to minimise, we now need to solve `:p/:p`*`/:p`*`/:r` $\models$ `:p` to understand if we should remove node b (and its edge) or not; by $e_1 \models e_2$, we denote the entailment of paths, i.e., that, for any node n_1 in any RDF graph, if there exists a node n_2 such that there exists a path from n_1 to n_2 matching e_1, then this implies the existence of a node n_3 such that there exists a path from n_1 to n_3 matching e_2.

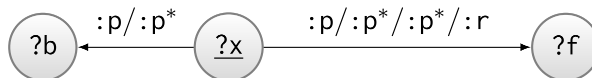


Figure 7.3: The result of the collapse of the pattern in Figure 7.1.

7.1.3 Collapse then Expand, or Expand then Collapse

Whether we decide to expand then collapse all paths, or collapse then expand, there are certain equivalences we may not detect. Figure 7.4 presents an example where expanding makes minimisation simple. Figure 7.5 presents a similar example where collapsing first makes mores sense, since minimisation is not needed once the property paths are normalised.

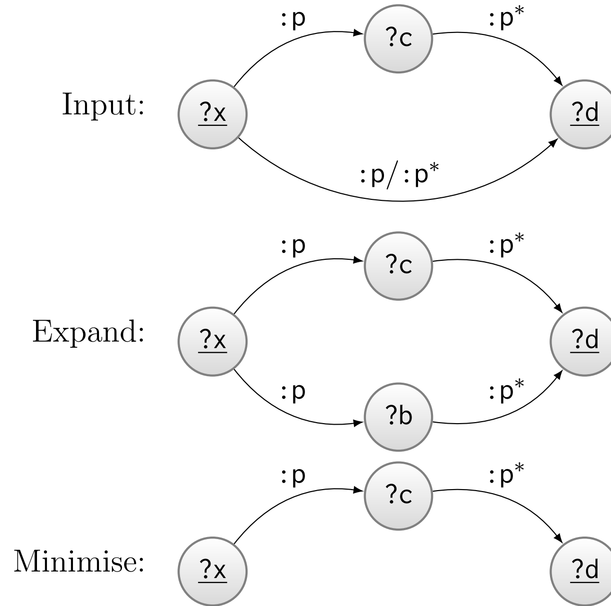


Figure 7.4: Example where expanding paths leads to more redundancies found.

We have also thought about alternating between collapsing and expanding in order to detect more equivalences, but it seems there are cases where a fixpoint will never be reached. (We can perhaps halt when a similar state is reached after a collapse or expansion.) It is not clear if such a method will yield a complete canonicalisation procedure.

In the example shown in Figure 7.6 we can see a loop of paths of `:p`*. Intuitively, we can see that this is equivalent to a self-loop from node `?a` (the only projected variable) to itself of `:p`* because all paths in the loop are of length 0 or more. To compute this, we can collapse it into a self-loop of `:p*/:p*/:p*/:p*/:p*/:p*`, which we could then minimise and find that it is equivalent to `:p`*. We can achieve this by collapsing and then normalising the aforementioned property path; we could then remove the pattern (`?a`, `:p*`, `?a`) if needed as a special case (a self-loop with *). However, in the example shown in Figure 7.7, which is equivalent to (`?a`, `:q`, `?x`), we cannot follow the same approach mentioned above for the example in Figure 7.6 because only variable `?d` has a degree of 2. Hence collapsing will just get rid of variable `?d`, leaving (`?c`, `:p*`, `?e`). Also expanding does nothing. We don't know what do with this pizza. It seems to require a special type of minimisation of graph patterns that can deal with the zero case. We'll look at some methods in Section 7.2.

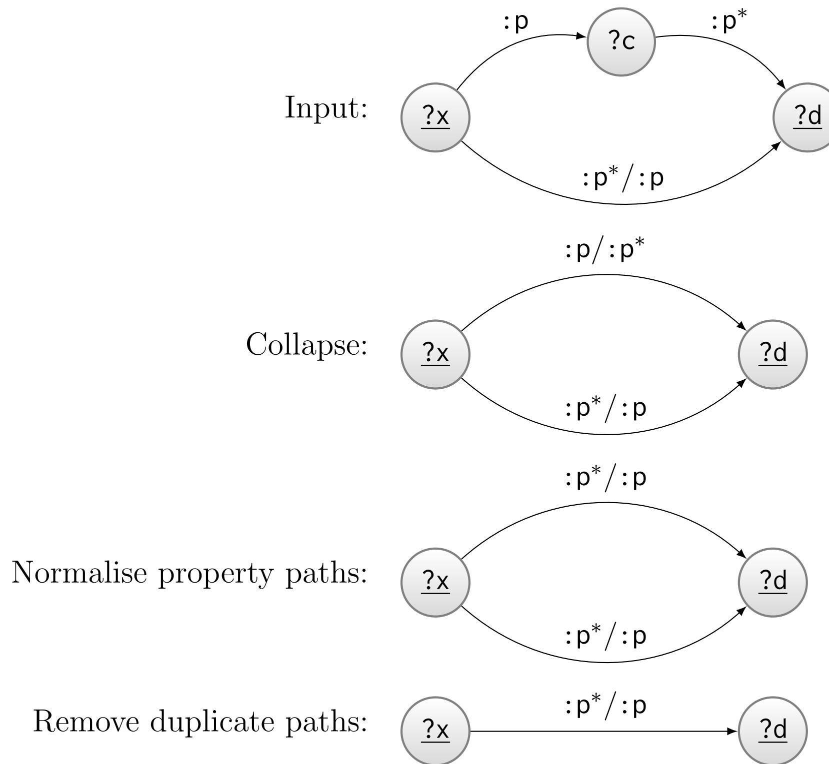


Figure 7.5: Example where collapsing paths leads to more redundancies found.

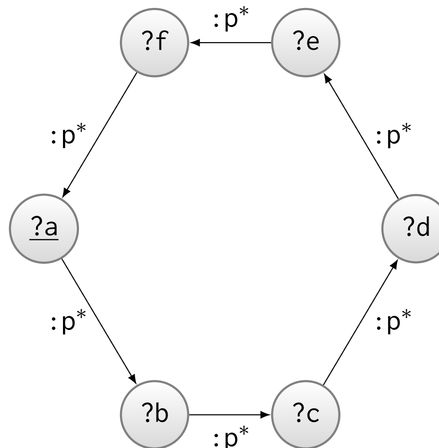


Figure 7.6: One ring to rule them all.

Zero-Length Entailments

There are some special cases relating to zero-length paths and entailments we would need to clean up.

It is possible to find path patterns of the form (?x, :p*, ?y) in graph patterns. Notably, if variable ?y is not projected and not mentioned elsewhere, then variable ?x will match every node in the subject position (every node is connected by a path of length 0 to itself). Therefore, the aforementioned path pattern may be removed safely as long as ?x appears in

a node position (subject or object) elsewhere.

Likewise something like (?a, :p*, ?a) can be removed so long as ?a appears in a node position elsewhere.

In a case like (?x, :p/:p*, ?y), we can also often simplify this to (?x, :p, ?y) via entailment (this would happen per the first case if we expand).

Inverse Entailments

There are some inverse cases that the aforementioned methods will not capture. Take for example the relatively simple case shown in Figure 7.8, where e^+ is just a shortcut for e/e^*. The example is in inverse normal form. This is equivalent to (?a, :p, ?b), but since we do not cover entailments involving inverses, here we have nothing to help us. There is nothing to expand nor collapse, and property path normalisation considers :p and :p⁻ to be unrelated constants. It seems we might need 2-way automata (and a normal form for them) to cover such cases. We will return to this issue in Section 7.2.3.

7.2 Minimisation

In this section we describe our more general attempts to minimise CPQs, the cases of equivalences it may detect, and their limitations.

7.2.1 Compute the Transitive Closure

So far, the presence of paths of length zero or more limit our ability to find a normal form for SPARQL queries with property paths. Later in this work, we discuss methods to deal with redundant paths of zero or more by removing them from the query (minimisation). However,

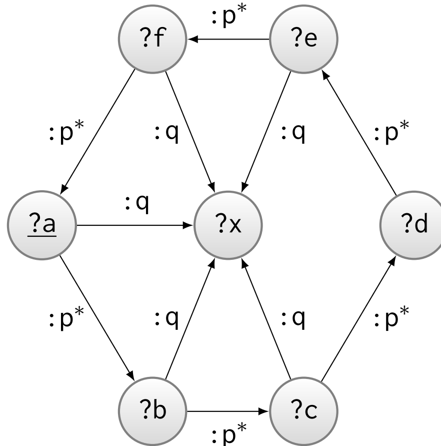


Figure 7.7: The dreaded pizza

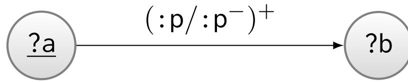


Figure 7.8: A problem with inverses.

we also considered a way to compute a normal form by instead *adding* more property paths. More precisely, we add paths of length zero or more (e.g. `:p`*) between all pairs of nodes n_1 and n_2 such that n_2 is reachable from n_1 by following paths of length zero or more, until we exhaust all possible transitions. We show examples of this in Figures 7.9 and 7.10. Of course, the result of this will not be a minimal form, but it may be a normal form.

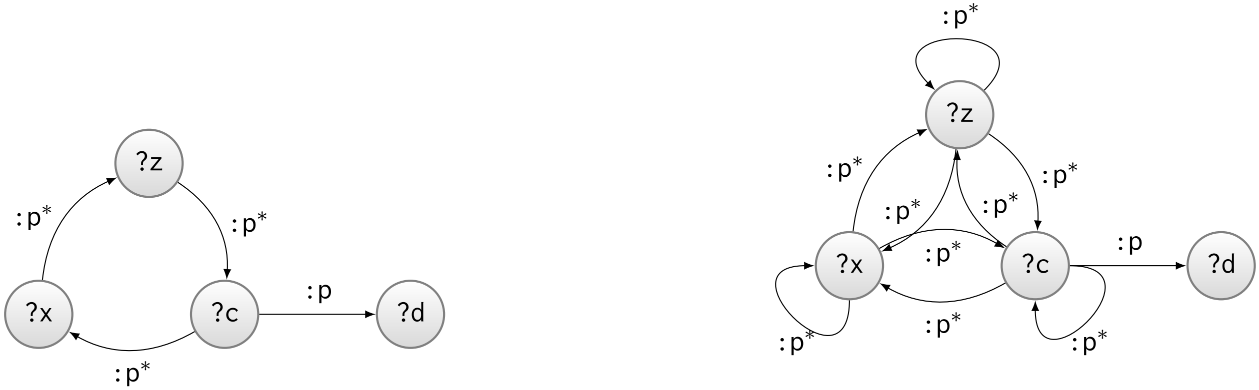


Figure 7.9: Example of a path pattern (left). Computation of the transitive closure of the same path pattern (right).

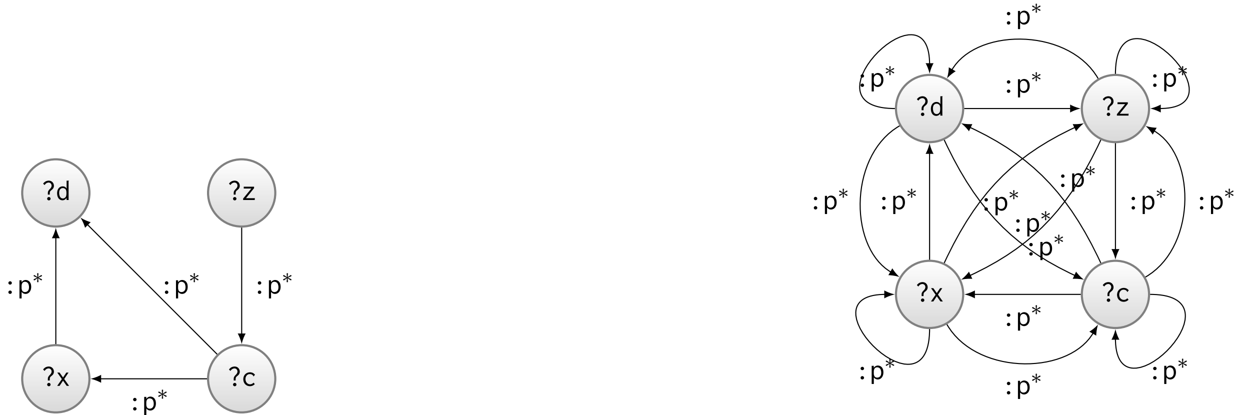


Figure 7.10: Example of a path pattern (left). Computation of the transitive closure of the same path pattern (right).

The idea is to avoid the need to remove redundant paths by computing in a closure similar to a chase to get to a least model.

Limitations Perhaps this could be applied after some form of minimisation to address corner cases, but in general does not seem to be all too promising in that if we have redundant

nodes, then we'll just be adding a bunch of paths to them that are unnecessary. For example, in Figure 7.10, if we project a single variable (any variable), then the entire query should collapse to a self-loop on that variable with :p* rather than the mess on the right. (On the other hand, if all four variables are projected, it is not clear what a normal form should look like, and maybe the monstrosity on the right is something to aim for.)

Another limitation is that, unlike in the case of minimisation, computing the closure produces a canonicalised query that is highly redundant, and may thus be less useful in practice for certain use-cases; it could be possible to apply a minimisation step after the closure to address this.

7.2.2 RPQ-aware Minimisation

The techniques we have defined so far aim to compute a normal form. However, they do not address the presence of redundant path patterns in C(2)RPQs in a general way. In this section we describe some of the techniques we have looked at to find redundancies in C2RPQs in a more general way, aiming at finding a core that takes into consideration entailment between property paths.

The RPQ-aware minimisation approach parallels the computation of the core of a graph to remove redundant nodes due to the presence of homomorphisms from the graph to a proper sub-graph of itself. We have implemented this for CQs/BGPs without property paths in a fairly standard and simple way. Given a CQ/BGP Q, we create a graph G that is the canonical graph for that query (the same structure). We then evaluate $Q(G)$, and we select the solution $\mu \in Q(G)$ that maps to the fewest constants in G, and maps all projected variables to themselves. We then rewrite Q per μ. So for example, if $Q = \{(\underline{?a}, :p, ?b), (\underline{?a}, :p, :x), (\underline{?a}, ?c, ?d)\}$, we will generate a graph $G = \{(v:a, :p, v:b), (v:a, :p, :x), (v:a, v:c, v:d)\}$, where nodes prefixed with v: denote variables, and evaluate $Q(G)$. We choose the solution $\mu \in Q(G)$ that maps <u>?a</u> to v:a (since it's a projected variable, μ should be the identity for it), and maps to the fewest unique constants with v:. In this case, the solution would be $\mu = \{\underline{?a}/v:a, ?b/:x, ?c/:p, ?d/:x\}$. This maps to only one variable node with v: and is the identity for projected variables. In Q, we can then map ?b $\rightarrow$:x, c $\rightarrow$:p, d $\rightarrow$:x per that solution, resulting in the minimised, equivalent query $Q' = \{(\underline{?a}, :p, :x)\}$.

In order to generalise this to handle property paths, we may need to be able to map a triple pattern like $(\underline{?a}, :p^*, ?y)$ into $(\underline{?a}, :p, :x)$, which entails it. Along these lines, we replace all (non-trivial) path patterns with a triple pattern that contains the same subject and object, but replaces the path expression with a variable unique to that triple pattern. By doing this, we generate candidates for which we need to check containment/entailment.

Given Q as input, Figure 7.11 shows how that we have mapped :p* to ?v1, and :q* to ?v2 to create Q'. Figure 7.12 shows how we create the canonical graph G from Q by replacing variables and path expressions with proxy IRIs that encode them. Table 7.1 shows the solutions of $Q'(G)$. Assume that the only projected variable is <u>?a</u>. We now:

1. Discard any results from $Q'(G)$ that are not the identity for projected variables. In this

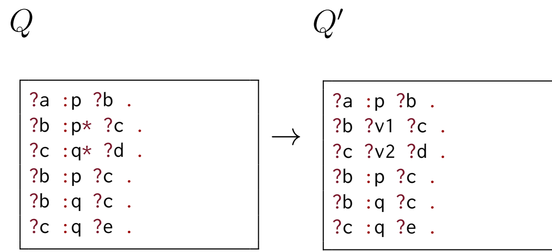


Figure 7.11: Mapping property paths to variables

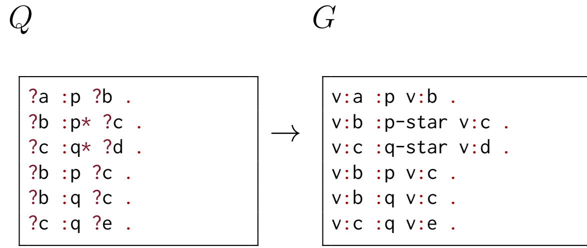


Figure 7.12: Creating the canonical graph

case, no solution is discarded as every solution maps `?a` to `v:a`.

2. Discard any results from $Q'(G)$ where any proxy variable for a path e maps to a proxy IRI for a path e' such that $e' \not\sqsubseteq e$. In this case, we discard the last two solutions of Table 7.1, as we cannot map `?v1` (representing `:p`$*$) to `:q` since `:q` $\not\sqsubseteq$ `:p`$*$.

3. Take the remaining solution that maps to the fewest unique proxy IRIs (as the solution that will get rid of the most variables and paths).[1]

?a	?b	?c	?d	?e	?v1 (:p*)	?v2 (:q*)
v:a	v:b	v:c	v:d	v:e	:p-star	:q-star
v:a	v:b	v:c	v:d	v:e	:p	:q-star
v:a	v:b	v:c	v:e	v:e	:p-star	:q
v:a	v:b	v:c	v:e	v:e	:p	:q
v:a	v:b	v:c	v:d	v:e	:q	:q-star
v:a	v:b	v:c	v:e	v:e	:q	:q

Table 7.1: Solutions for Q' over the canonical graph G

In this case, the selected solution will be the fourth one in Table 7.1, which maps to four unique proxy IRIs. Rewriting the query according to the selected solution should give us the minimised version of the query, as shown for this case in Figure 7.13.

Limitations A limitation with this approach is the dreaded pizza, presented in Figure 7.7. Again, we would like to minimise this to a self-loop on `?a` with `:p`*, and a link to `?x` from `?a` with `:q` Ȧ possible solution to this issue is to add self-loops into the canonical graph. For

[1]Admittedly, it's not clear that there cannot be ties here?

```
?a :p ?b .
?b :p ?c .
?b :q ?c .
?c :q ?e .
```

Figure 7.13: Minimised (core) version of Q

example, in this case, if we add (?a, :p-star, ?a), which does not change the semantics of the query, then this method will result in the expected minimisation.

Another concern is the inverse case, where this method cannot reverse the direction of an edge when computing answers. However, it may be possible to address this through an inverse rewriting, inverse normal forms, etc. In the worst case, we could add the inverses for all edges explicitly to the canonical graph.

A final concern is the case where we need to map one property path into two or more edges in the canonical graph. Take for example Figure 7.15, where the ?c node and its incident edges are redundant (under set semantics at least). We would need a way to deal with such cases. In fact, if we add a self-loop on ?d with :p*, then this would be okay (?c would then map to ?d). This may or may not be sufficient. In other cases, we might need to apply the transitive closure of the canonical graph, per Section 7.2.1.

Another question is what sort of relation we should we checking for paths that are "mappable" to one another per the homomorphism. In general, we should of course be mapping more general into more specific, but it is not clear in which cases this relation is specifically containment, or rather some form of entailment (of 2RPQs).

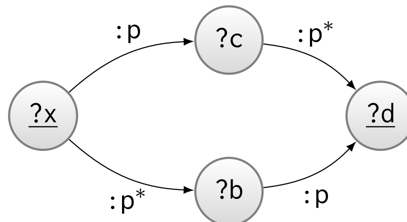


Figure 7.14: Here ?c should collapse into ?b or ?b into ?c

Figure 7.14 shows an RPQ for which we may collapse ?b into ?c or vice-versa. This is effectively a consequence of the fact that both :p/:p* and :p*/:p are both valid and equivalent ways of rewriting :p+. It is not clear which of the two should be preferred.

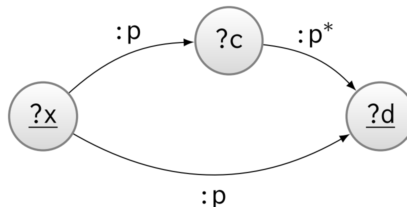


Figure 7.15: A similar case to Figure 7.14, but with one edge mapping to two

7.2.3 2RPQ Containment and Equivalence

Part of the minimisation of UCQs includes checking for containment between CQs, and the computation of the cores of CQs in order to find redundancies. In a similar manner, adapting this method to C2RPQs requires the finding of redundancies. However, containment of property paths is sometimes not a sufficient condition for determining whether an RPQ is redundant or not. Instead, we must check for *entailment* of property paths. For instance, if there exists a result for `?x` yielded by (`?x`, `:p`, `?y`), then this means that there exists a result for `?x` yielded by (`?x`, `:p/:p`$^-$, `?y`), even if neither expression is contained in the other. We also need to deal with 2RPQs.

A 2RPQ Q is defined over a set of symbols (or *alphabet*) Σ, and expressed as either a regular expression or finite-state automaton over $\Sigma^{\pm} = \Sigma \cup \{r^- \mid r \in \Sigma\}$ (i.e. the set of all symbols in Σ or their inverses). Then, $L(Q)$ denotes the regular language defined by Q. Finally, we say that a sequence of symbols in $\Sigma^{\pm}$ $w = w_1 w_2 \cdots w_n$ *conforms* to Q if and only if $w \in L(Q)$. In particular, property paths without negated property sets are analogous to 2RPQs over the set of all IRIs.

Folding

The entailment and containment decision problems are harder for regular path queries with inverse because regular path queries can be "folded" into shorter paths. Formally, we say that a sequence of symbols $v = v_1 v_2 \cdots v_m$ *folds* onto $u = u_1 u_2 \cdots u_n$ $(m \leq n)$ if there is a sequence $i_0, \cdots, i_m$ of positive integers between 0 and n such that:

- $i_0 = 0$ and $i_m = n$, and
- for $0 \leq j < m$, either:
  - $i_{j+1} = i_j + 1$ and $v_{j+1} = u_{i_j+1}$, or
  - $i_{j+1} = i_j - 1$ and $v_{j+1} = {u_{i_{j+1}}}^-$

For example (`?x`, $p/p^-/p$, `?y`) can be folded into (`?x`, p, `?y`) via the sequence $i = 0, 1, 0, 1$, and thus the latter is contained in the former. Calvanese et al. [16] then define the following for containment of 2RPQs, where Q and Q' are 2RPQs, $L(Q)$ denotes the regular language defined by Q, and $fold(L)$ is a function over a regular language L such that $fold(L) = \{u \mid v \text{ folds onto } u,\ v \in L\}$.

$$Q \sqsubseteq Q' \iff L(Q) \subseteq fold(L(Q'))$$

We may construct a *two-way non-deterministic automaton* that accepts the language of $fold(L(Q))$.

Two-Way Automata

Two-way automata denote the same language as regular expressions with inverse, but using a lower number of states and transitions than DFAs and NFAs.

A two-way non-deterministic automaton $M = (Q, S, F, \Sigma, \delta)$ is a five-tuple where Q is the set of states, S is the set of starting states, F is the set of accepting states, Σ denotes a set of binary relational symbols, and δ is a transition function. $\Sigma^{\pm}$ denotes a set of symbols that contains all symbols in Σ as well as the inverse of each symbol (i.e. whenever it's read, we go backwards). Finally, $\hat{\Sigma}$ denotes a set of symbols equal to $\Sigma^{\pm}$ plus a left-end marker ($\dashv$) and a right-end marker ($\vdash$).

The transition function $\delta : Q \times \Sigma^{\pm} \rightarrow 2^{Q\times\{-1,0,1\}}$ denotes that a run of the 2NFA reads a symbol, and depending on the current state, moves to one or many different states and (optionally) moves the pointer either one position forward or backwards. A *configuration* is pair (q, i) where q is a state, and i is an integer, and an *accepting run* of M on a word $w =\dashv w_1 \cdots w_n \vdash$ is a sequence of configurations $(q_1, i_1), \cdots, (q_m, i_m)$ where each q_j is a state in Q, and $1 \leq i_j \leq n + 1$ for $1 \leq j < m$. In addition, the initial configuration (q_1, i_1) is such that $q_1 \in S$ is a starting state, and $i_1 = 1$; the final configuration (q_m, i_m) is such that $q_m \in F$ is an accepting state, and $i_m = n + 1$; and for each configuration (q_j, i_j) it is the case that $(q_{j+1}, c) \in \delta(q_j, a_{i_j})$ such that $i_{j+1} = i_j + c$. These conditions state that any accepting run begins in a starting state, while reading the first symbol of the word, ends in a final state after reading the last symbol (w_n) and the last pointer (i_m) is at the end of string. It is evident that the sequence of integers $i_1, \cdots, i_n$ of an accepting run may be used to prove that a regular expression can be folded into another.

In the following we will look at the constructions of two-way automata for some 2RPQs, similar examples to those presented by Reutter [61]. We use the standard notation for regular expressions (and denote their property path counterpart in parentheses) where a generic symbol is denoted by a, a^{-} denotes the inverse of a, $\cdot$ denotes concatenation (/), $\cup$ denotes disjunction (|), and * denotes the Kleene star. Furthermore, q_0 denotes the initial state, q_f denotes an accepting state, and q_r denotes a rejection state (defined as a normal state, but with the intuition that the state is currently rejecting). We begin reading the first symbol w_1 after the left-end marker.

- $R = a$ (see Figure 7.16)
  - $M = (\{q_0, q_f, q_r\}, q_0, \{q_f\}, \Sigma^{\pm}, \delta)$
  - $\delta(q_0, a) = \{(q_f, 1)\}$
  - For every symbol $b \in \Sigma^{\pm}$ such that $b \neq a$: $\delta(q_0, b) = \{(q_r, -1)\}$
  - $\delta(q_r, a^{-}) = \{(q_f, 0)\}$
- $R = a^{-}$ (See Figure 7.17)
  - $M = (\{q_0, q_f, q_r\}, q_0, \{q_f\}, \Sigma^{\pm}, \delta)$
  - $\delta(q_0, a^{-}) = \{(q_f, 1), (q_r, -1)\}$
  - For every symbol $b \in \Sigma^{\pm}$ such that $b \neq a^{-}$: $\delta(q_0, b) = \{(q_r, -1)\}$

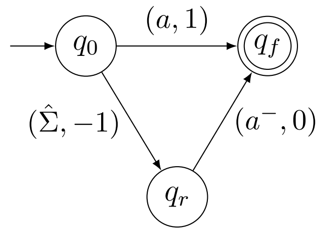


Figure 7.16: A 2NFA that accepts words in $fold(a)$.

– $\delta(q_r, a) = \{(q_f, 0)\}$

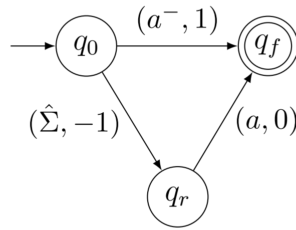


Figure 7.17: A 2NFA that accepts words in $fold(a^-)$.

- $R = R_1 \cdot R_2$
  - $M = (Q, q_0, F, \Sigma^{\pm}, \delta)$
  - $Q = \{q_0, q_f\} \cup Q^1 \cup Q^2$
  - $F = \{q_f\} \cup (F^1 - \{q_m^1\}) \cup (F^2 - \{q_m^2\})$
  - $\delta = \delta^1 \cup \delta^2$ plus for each $a \in \hat{\Sigma}$:

$$\delta(q_0, a) = \{(q_0^1, 0)\}$$
$$\delta(q_f^1, a) = \delta^1(q_f^1, a) \cup \{(q_0^2, 0)\}$$
$$\delta(q_f^2, a) = \delta^2(q_f^2, a) \cup \{(q_f, 0)\}$$

- $R = R_1 \cup R_2$
  - $M = (Q, q_0, F, \Sigma^{\pm}, \delta)$
  - $Q = \{q_0, q_f\} \cup Q^1 \cup Q^2$
  - $F = \{q_f\} \cup (F^1 - \{q_f^1\}) \cup (F^2 - \{q_f^2\})$
  - $\delta = \delta^1 \cup \delta^2$ plus for each $a \in \hat{\Sigma}$:

$$\delta(q_0, a) = \{(q_0^1, 0)\}$$
$$\delta(q_f^1, a) = \delta^1(q_f^1, a) \cup \{(q_f, 0)\}$$
$$\delta(q_f^2, a) = \delta^2(q_f^2, a) \cup \{(q_f, 0)\}$$

- $R = {R_1}^*$
  - $M = (Q, q_0, F, \Sigma^{\pm}, \delta)$
  - $Q = \{q_0, q_f\} \cup Q^1$
  - $F = \{q_f\} \cup (F^1 - \{q_f^1\}))$
  - $\delta = \delta^1$ plus for each $a \in \hat{\Sigma}$:

$$\delta(q_0, a) = \{(q_0^1, 0)\}$$
$$\delta(q_0^1, a) = \delta^1(q_0^1, a) \cup \{(q_f, 0)\}$$
$$\delta(q_f^1, a) = \delta^1(q_f^1, a) \cup \{(q_f, 0), (q_0^1, 0)\}$$

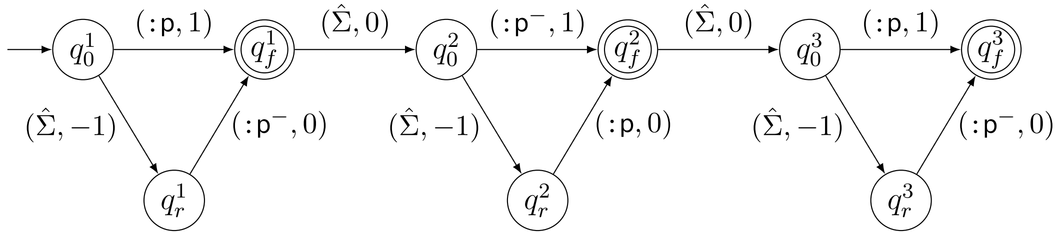


Figure 7.18: A 2NFA that accepts words in $fold(:\mathsf{p}/:\mathsf{p}^-/:\mathsf{p})$.

The 2NFA shown in Figure 7.18 accepts words in $fold(:\mathsf{p}/:\mathsf{p}^-/:\mathsf{p})$. It follows the inductive construction described earlier. It is evident that the word $w = \dashv pp^-p \vdash$ will reach an accepting state through the run $(q_0^1, 1), (q_f^1, 2), (q_0^2, 2), (q_f^2, 3), (q_0^3, 3), (q_f^3, 4), (q_f, 4)$. On the other hand, it is less evident that the word $w' = \dashv p \vdash$ will reach an accepting state. This can be done through the following run beginning with the configuration $(q_0, 1)$:

- We begin at q_0, read any symbol, and move to q_0^1.
- At state q_0^1 we once again read p, move the pointer to the right, and move to state q_f^1, leading to the configuration $(q_f^1, 2)$.
- At state q_f^1 we read any symbol, and move to q_0^2, leading to the configuration $(q_0^2, 2)$.
- At state q_0^2 we read the right-end marker $\vdash$, move the pointer to the left, and move to q_r^2, leading to the configuration $(q_r^2, 1)$.
- At state q_r^2 we read p, and move to q_f^2, leading to the configuration $(q_f^2, 1)$.
- At state q_f^2 we read p, and move to q_0^3, leading to the configuration $(q_0^3, 1)$.
- At state q_0^3 we read p, move the pointer to the right, and move to state q_f^3, leading to the configuration $(q_f^3, 2)$.
- Finally, at state q_f^3 we read the right-end marker, and move to state q_f (an accepting state), leading to the configuration $(q_f, 2)$.

It is evident that this run is an accepting run since the first configuration is $(q_0, 1)$, the final configuration is $(q_f, 2)$ where q_f is an accepting state, and every configuration is well defined by the transition function δ.

Any DFA that is equivalent to this 2NFA will have an exponential number of states compared to the number of states of the 2NFA.

After constructing automata for both Q and Q', we may construct a 2NFA that is the difference (or intersection of Q and the complement of Q'). The study by Calvanese et al. [16] presents a way to construct a "small" NFA that accepts this language, and has a size that is exponential on the size of the 2NFA used to represent Q or Q'. We now outline these steps:

- Construct NFAs M and M' such that $L(M) = L(Q)$ and $L(M') = L(Q')$.
- Construct a 2NFA M'' such that $L(M'') = fold(L(M'))$.
- Construct an NFA $\overline{M''}$ such that $L(\overline{M''} = (\Sigma^{\pm})^* - L(M''))$ (i.e. the complement of M'').
- Construct an NFA $M^{\times} = M' \times \overline{M''}$ such that $L(M^{\times}) = L(Q) - fold(L(M'))$.
- Check if there is a path from a start state to a final state in $M^{\times}$.

This would allow us to determine if an RPQ is contained in a different RPQ, which would most likely mean that it is redundant and may be removed. This may be used in the process described in Section 7.2.2 instead of the basic NFA containment mentioned there, which may help us compute the "core" of an RPQ. A brute-force approach may remove triple or path patterns in an RPQ Q, and then check if the resulting RPQ Q' is equivalent to Q by using the aforementioned automata approach.

7.2.4 Union Expansion

In a completely different direction, we can view a path pattern (x, e^+, y) as an infinite union of basic graph patterns corresponding to $(x, e, y) \cup (x, e/e, y) \cup \ldots \cup (x, e/\ldots/e, y) \cup \ldots$. Our idea was that such an expansion of path expressions into an infinite union would allow us to re-use our UCQ method to find a minimisation. In addition, we believe that we would only need to expand up to a certain point in order to test for containment. This allows us to find cases where non-recursive paths or sub-BGPs are contained in recursive paths.

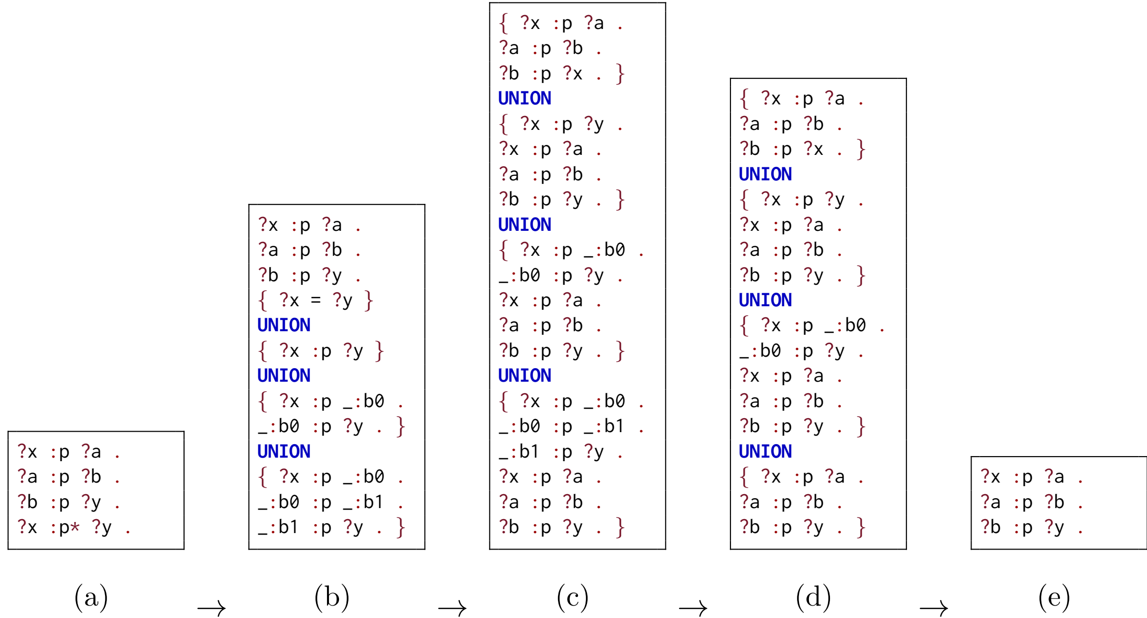


Figure 7.19: Example of the steps of the union expansion of a path pattern.

In the example presented in Figure 7.19 we begin with the path pattern seen in Figure 7.19a. The next step is shown in Figure 7.19b where we have expanded the path of zero or more `:p` into a union of up to 4 patterns (from 0 to 3) because the pattern contains 3 triple patterns. The pattern $\{$`?x` $=$ `?y`$\}$ denotes a pattern where `?x` and `?y` are mapped to the same nodes when expanded into CQs as part of the UCQ expansion (the zero-length path case). This can be seen in Figure 7.19c where we see the result of the UCQ expansion of the pattern in Figure 7.19b, and the first CQ shows that `?y` has been replaced by `?x`; the other CQs each show cases where the paths between `?x` and `?y` are of lengths between 1 and 3. Following this, the pattern in Figure 7.19d shows the result of minimising each CQ of the pattern in Figure 7.19c. Finally, Figure 7.19e shows the final result of the UCQ minimisation (which we already have), where only a single CQ remains because it is contained by all other CQs in the previous step. The result shows that the path pattern (`?x`, `:p`*, `?y`) was redundant.

Limitations One issue that we encountered is the case where the path is of length 0 (i.e. $(\mathtt{x}, p^*, \mathtt{y})$ has cases where $\mathtt{x} = \mathtt{y}$), which cannot be expressed directly as an MQ. In that case, we may rewrite the basic graph pattern by replacing all instances of `?x` with `?y` and vice-versa, but it gets messy. Also it's not clear how far we need to expand the infinite union, nor how to handle containment between recursive patterns. Finally, we would need to rewrite

the expansion back to the recursive case to restore equivalence.

7.3 Discussion

In this chapter, we have discussed several techniques that we have considered for the canonicalisation of UC2RPQ-style queries. However, in each case, we have managed to construct counterexamples that show the technique to fall short of complete canonicalisation. Nevertheless, such techniques can be useful for partial but sound canonicalisation. This leaves an interesting open question of whether or not a canonical form for UC2RPQs is possible to construct, and if so, how.

Chapter 8

Implementation

In this chapter we describe the tools – mainly software – that were used to implement the structures, representations and techniques described in the previous chapter. A better understanding of the implementation is important in the context of performance measures that will be presented later in Chapter 9. These steps are summarised as follows:

1. Parsing queries and constructing r-graphs
2. Normalisation of r-graphs
3. Canonical Labelling
4. Generation of queries

We also include a section describing some of the test cases which were used to test the correctness of certain transformations.

8.1 Dependencies

The implementation makes use of Apache Jena [49] version 3.17, a framework for the development of Semantic Web and Linked Data applications. It is used to represent queries in an algebraic form, perform transformations over these algebraic queries, traverse their structure, build RDF graphs in order to represent these queries, and serialise these graphs into valid SPARQL queries.

In addition, the project is structured as a Maven project [1], which manages its dependencies, and makes the project easier to build and include for other applications.

[1] https://maven.apache.org/

8.2 Project Structure

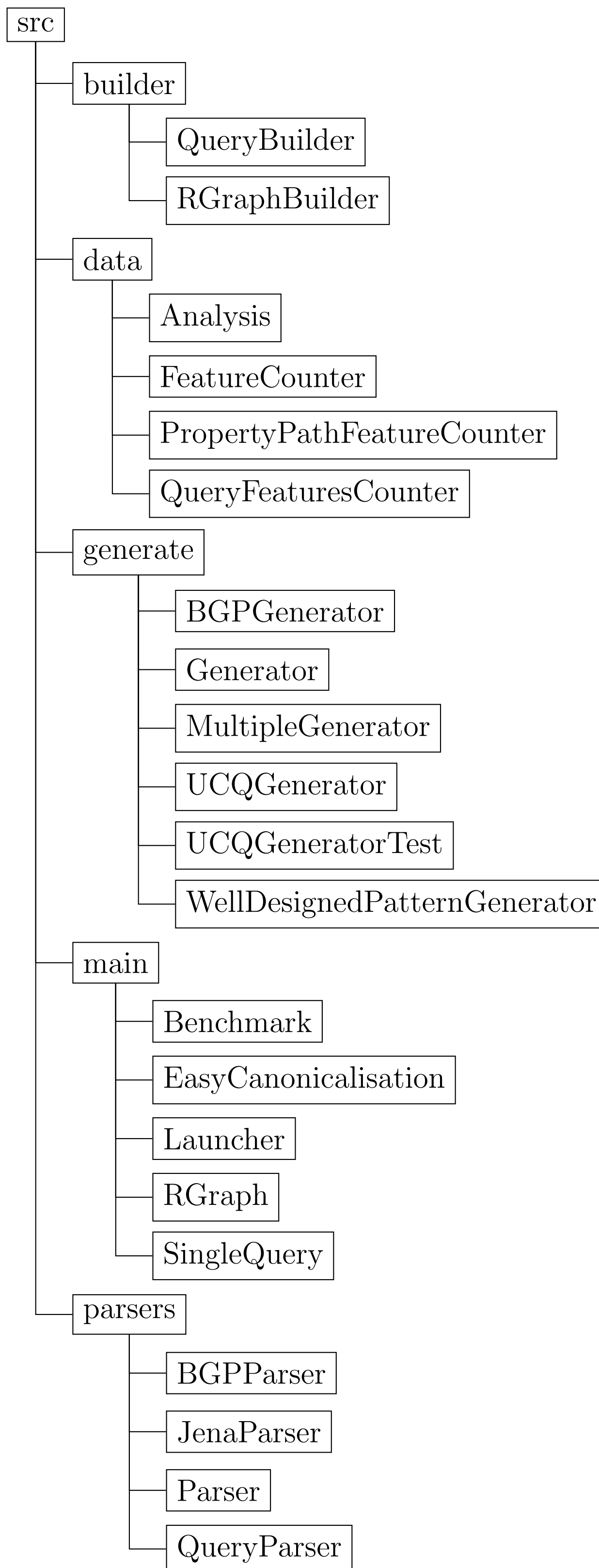


Figure 8.1: File structure of project QCan

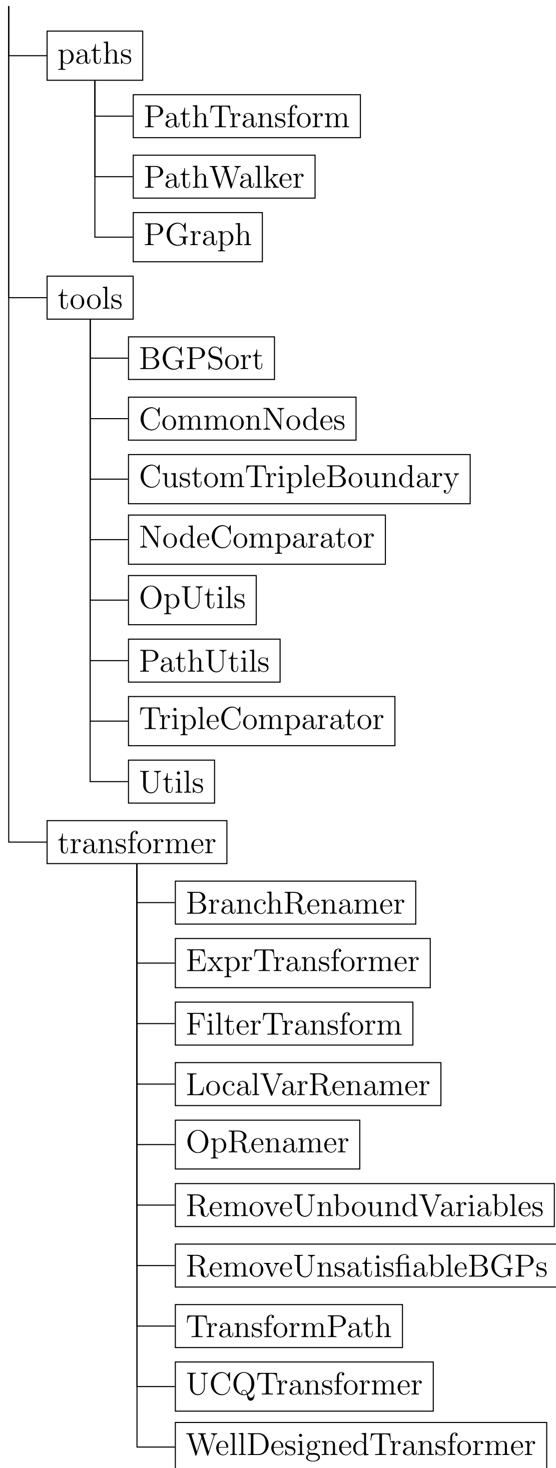


Figure 8.1: File structure of project QCan (continued)

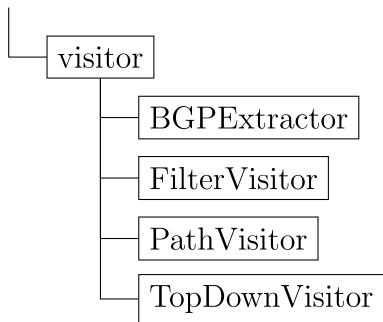


Figure 8.1: File structure of project QCan (continued)

Figure 8.1 displays the general structure of the Java classes of this project. We will now explain in greater detail each package and the classes they comprise.

8.2.1 Builder

This package contains classes used to build data structures, namely SPARQL queries and r-graphs.

Query Builder This class is used to serialise an `RGraph` object into a `Op` object as provided by Jena, an algebraic expression. This can subsequently be used to form a query in `String` form.

RGraph Builder This class is an implementation of the `OpVisitor` interface provided by Jena, which is used to traverse an `Op` object in a bottom-up fashion. By traversing the structure, an equivalent `RGraph` object is built.

8.2.2 Data

This package contains classes used to analyse sets of queries for later experiments.

Analysis This class is used to read the files written after our experiments, and automatically calculate relevant statistics such as minimum values, median values, etc.

Feature Counter This class is an implementation of the `OpVisitor` interface provided by Jena. It is used to find all features contained in a single query.

Property Path Feature Counter This class is an implementation of the `PathVisitor` interface provided by Jena. It is used to find all property path features contained in a single query, such as Kleene stars, negated property sets, etc.

Query Features Counter This class makes use of the `FeatureCounter` and `PropertyPathFeatureCounter` classes to count all occurrences of SPARQL features in a set of queries.

8.2.3 Generate

This package contains classes used to generate artificial queries for benchmarking.

Generator An abstract class used as a template for other classes in this package.

BGP Generator This class is used to generate basic graph patterns with structures based on well-known classes of graphs, such as cliques, lattices, etc.

Multiple BGP Generator This class traverses a directory, and generates basic graph patterns based on each file in the directory.

UCQ Generator This class is used to generate UCQs from monotone queries.

UCQ Generator Test This class is used to run experiments on queries generated by the `UCQ Generator` class, and record the results.

Well-designed Pattern Generator This class is used to generate well-designed optional patterns with random variable names.

8.2.4 Main

This package contains classes we deem to be crucial to the project.

Benchmark The main class used to run our experiments over large sets of queries. These generate files that contain the runtimes for each query, and also a file with the distribution of queries under the congruence class equivalence.

Easy Canonicalisation A simple class used to either canonicalise a single query provided as part of the input, or alternatively a file containing queries. In addition, one can choose to skip certain parts of the canonicalisation algorithm.

Launcher A class whose only purpose is to receive commands as input, and pass those commands to the appropriate class launcher. This is because JAR files usually define a single class as its main class.

R-graph A class that encapsulates the functions used to build an r-graph, serialise different data structures into an r-graph, perform operations over r-graphs, etc.

Single Query A class that encapsulates the functions used to canonicalise SPARQL queries. It also contains fields for measuring the time it takes to construct an r-graph, and the time it takes to output the canonical form of the query. Other fields include: the number of triples patterns in the r-graph, the number of nodes in the r-graph, and the number of variable nodes in the r-graph; there are two instances of these fields: one for the r-graph before any leaning, and another for after.

8.2.5 Parsers

Parser This class serves as a template for the other parser classes. It contains methods used to read text files that contain queries line by line. In addition, it stores the unique queries, unsupported queries, etc.

BGP Parser This class extends the `Parser` class in order to read queries, find all basic graph patterns it contains, canonicalise them, and store the results.

Jena Parser This class extends the `Parser` class in order to read queries, transform them into `Op` objects, and store the results.

Query Parser This class extends the `Parser` class, and receives a number of boolean values as input to determine what steps of the canonicalisation method are taken: normalisation, graph representation, minimisation, and canonical labeling. Furthermore, this class takes a text file that contains many SPARQL queries, and calls multiple instances of `SingleQuery` for each query. It writes into a text file each of the fields in `SingleQuery` separated by tabs.

8.2.6 Paths

This package contains classes used to transform, traverse, and normalise property paths.

Path Transform This class is used to transform property paths according to the rules we defined in Section 5.3.

Path Walker This class implements the `PathVisitor` interface provided by Jena in order to traverse `Path` objects, and construct a `PGraph` object that represents an NFA.

PGraph A class that encapsulates the functions used to build NFAs and DFAs, minimise DFAs, serialise DFAs into property paths, perform operations over other automata, etc.

8.2.7 Transformer

This package contains classes used to apply transformations to `Op` objects, mostly in order to rewrite queries into normal forms as described in Section 5.1.1.

Branch Renamer This class is used to rename the *union variables* in UCQs, as described in Section 5.1.1.

Local Var Renamer This class is used to rename *local variables* as described in Section 5.1.1.

Expression Transformer This class normalises expressions into a disjunctive normal form.

Filter Transformer This class normalises graph patterns with filter expressions by pushing the expressions outside, following the notion of *safe variables* as described in Section 5.2.1.

Remove Unbound Variables This class removes unbound variables from projections as described in Section 5.1.1.

Remove Unsatisfiable BGPs This class replaces any unsatisfiable BGPs with a standard unsatisfiable graph pattern.

Well-designed Transformer This class rewrites well-designed optional patterns as described in Section 5.2.1.

8.2.8 Visitor

This package contains classes that traverse objects such as those of the `Op` or `Path` classes in order to analyse the contents of the objects (such as the operators in a query or property path).

BGP Extractor This class traverses `Op` objects in order to extract all BGPs that appear in the query.

Filter Visitor This class traverses `Expr` objects, and builds an `RGraph` that represents its corresponding built-in expression.

Path Visitor This class traverses `Path` objects (which represent property paths) in order to analyse the structure and features of the property path.

Top-down Visitor This is an abstract class that traverses `Op` objects in a top-down fashion, which is unlike the visitors provided by Jena.

8.3 Canonical Labelling

We use the *blabel* `https://blabel.github.io/` library for the labelling and leaning of the r-graphs [38]. This library contains several functions for finding or determining isomorphism and equivalence in RDF graphs, as well as several tools that are related to isomorphism and equivalence.

Among these tools is a canonical labelling algorithm that is deterministic and preserves isomorphism. To determine the labels, it utilises a hashing algorithm, which means that hash collisions are possible. However, these values are 128 bits in length, so collisions are very unlikely to occur. This is especially true considering the mean number of blank nodes in the r-graphs.

We utilise the canonical labelling algorithm for blank nodes provided by `blabel` to label our r-graphs. We have implemented functions to provide a representation of an r-graph that is readable by `blabel`, and vice-versa.

For the leaning of graphs we also use an algorithm provided by `blabel` which computes the core of an RDF graph.

8.4 Serialisation of Representational Graphs

After the r-graph of a query has been constructed, normalised, labelled and leaned, we need to reconstruct the canonical query represented by the r-graph. To do this, we have created the `QueryBuilder` class, which contains methods that take a `Node` object, check its type, and output an `Op` object of the corresponding type. In addition, depending on the type of the node, more recursive calls are made to nodes that are connected to the former.

This query is constructed by traversing the r-graph recursively, starting from the root node that is stored by every object of the `RGraph` class. The result of this process is an `Op` object that represents the same query as the `RGraph` object. Following this, we make use of Apache Jena's own methods to serialise the `Op` object to a `String`. Finally, we assign labels that are easier to read to each variable in the resulting query.

8.5 Property Paths and Automata

We have designed the `PGraph` class to be a graph representation of property paths similar to what the `RGraph` class is to queries. This means that `PGraph` contains functions for the construction of a NFA equivalent to a given property path. Furthermore, it contains functions for the construction of a DFA based on the aforementioned NFA, and finally a function for the minimisation of the DFA based on Hopcroft's algorithm.

The construction of the NFA is accomplished by the class `PathWalker`, an implementation of the `PathVisitor` interface, which traverses the property path in a bottom-up trajectory. As it traverses the property path, it creates fresh blank nodes that correspond to the states of the NFA, and the transitions following Thompson's algorithm for the construction of NFAs from regular expressions [73].

8.6 Testing

The testing of the software has been complicated because there are potentially infinite variations of query patterns. We cannot test for correctness and completeness for every possible query, but we have defined a number of test cases that contain queries which use the main features supported by our software. There is at least one test case defined for each feature, some of which combine multiple features.

Some of the test cases contain combinations of features that we anticipated would produce complications. For instance, the effect the presence of filter expressions would have on the UCQ transformation process.

```
SELECT DISTINCT ?z
WHERE {
  { ?c <http://ex.org/a> ?y ; ?p  ?o . }
  UNION
  { ?c <http://ex.org/b> ?y }
  ?y <http://ex.org/c> ?z
}
```

```
SELECT DISTINCT ?q4
WHERE {
  { ?q3 <http://ex.org/a> ?q0 ;
    ?q2 ?q1
  { ?q0 <http://ex.org/c> ?q4 }
  }
  UNION
  { ?q3 <http://ex.org/b>  ?q0 .
  ?q0 <http://ex.org/c>  ?q4 }
}
```

```
SELECT DISTINCT ?q0
WHERE {
  { ?q4 ?q3 ?q2 ;
    <http://ex.org/a> ?q1
  { ?q1 <http://ex.org/c>  ?q0 }
  }
  UNION
  { ?q4 <http://ex.org/b>  ?q1 .
  ?q1 <http://ex.org/c>  ?q0 }
}
```

Figure 8.2: Example of a test case for UCQ rewriting, leaning, and canonical labelling.

Figure 8.2 contains three SPARQL queries that are congruent. The original text file used for testing would contain a single query per line, but we have formatted them here differently for readability. Each one of these queries is processed fully by the canonicalisation algorithm, and the resulting canonical queries are compared to test the correctness of the leaning process.

Other test cases are *negative* in the sense that they test that two queries are *not* congruent.

```
SELECT *
WHERE {
  ?s <http://ex.org/p> ?o1 .
  ?s <http://ex.org/p> ?o2 .
  FILTER((?o2 != 1000) || REGEX(
       ?o1,"ˆa"))
}
```

```
SELECT *
WHERE {
  ?s <http://ex.org/p> ?o1 .
  ?s <http://ex.org/q> ?o2 .
  FILTER(1000 != ?o2)
  FILTER(REGEX(?o1,"ˆa"))
}
```

Figure 8.3: Example of a test case for filter rewriting and canonical labelling.

Figure 8.3 contains two queries that contain `FILTER` expressions. Though both contain the same built-in expressions, the left query looks for mappings where the values for `?o1` begin with the letter "a" or values for `?o2` different from 1000. On the other hand, the query on the right looks for mappings where both conditions are met simultaneously. Furthermore, in the query on the left `?s` is connected to `?o2` by the property `http://ex.org/p`, whereas in the query on the right, it is connected by the property `http://ex.org/q`. Because of this, this test case checks that both queries are not congruent, and will fail if both are found to be congruent.

Chapter 9

Results and Evaluation

For the purposes of this study, we define the following research questions relating to the hypothesis outlined at the outset::

- How is the performance of the canonicalisation process for real-world queries?
- Given that our algorithm is doubly-exponential, at what point does the algorithm begin to fail?
- How many more duplicate queries can be detected using our procedure versus baseline syntactic methods?

We now describe the evaluation we have designed to answer the previous questions, and then the experimental results generated, as well as a discussion of the results obtained.

9.1 Queries used

It is evident that the techniques described in Chapter 5 should be tested with SPARQL queries. However, to verify each of the questions we have postulated, we will need different types of queries. We will now describe the queries used to test each aspect of the canonicalisation process.

9.1.1 Real-world Queries

For the purpose of evaluating the performance of the software on real-world queries, a dataset from the *Linked SPARQL Queries* (*LSQ*) study was used [65]. This dataset contains queries taken from the logs of six public SPARQL endpoints.

The raw logs contained a large amount of information that was deemed unnecessary for the evaluation, since only the raw queries were relevant. A script was used over the dataset to extract the queries.

Following this, another process was carried out to filter queries that were syntactically incorrect. In this process, a query was filtered if the query parser provided by Apache Jena was unable to parse the raw query and threw an exception.

Table 9.1 shows the distribution of features in the LSQ dataset. Notable results include the fact that Wikidata contains the most variety of features. In particular, it is the only set of queries that contain federated services (`SERVICE`). Furthermore, it also contains the most instances of property paths (other logs were collected prior to the recommendation of the SPARQL 1.1 standard).

Table 9.1: Distribution of features in the DBpedia (DBp), SWDF, RKB Explorer (REx), RKB Endpoint (REn), Wikidata (WD) and Linked Geo (Geo) sets of queries.

Feature	DBp	SWDF	REx	REn	WD	Geo
BGP	424,328	112,398	335,450	169,425	861,383	842,794
Distinct	192,168	57,195	210,058	147,575	538,878	5,477
Union	89,766	32,459	5,550	1,653	92,807	49,299
Join	58,952	54,702	34,180	19,355	751,243	78,575
Service	0	0	0	0	406,990	0
Filter	200,001	4,716	14,938	8,974	65,081	125,849
Opt	31,211	27,824	23,587	9,741	343,838	19,914
Mods	4,576	60,199	80,655	40,957	321,239	412,343
Bind	0	15,307	19,641	0	71,102	0
Agg	0	10,268	16,444	0	42,777	0
Minus	0	25	192	0	5,700	0
Paths	0	37	192	96	268,530	0
Values	0	83	0	0	55,037	174
Total	424,362	112,470	335,833	169,617	872,555	842,794

We can observe that `UNION` is used far less than other features such as `OPTIONAL` and `FILTER`. In addition, as we have already mentioned, the full canonicalisation of these features would be undecidable, but since they are so widely used, we felt it was important to offer an incomplete canonicalisation of these features.

Features introduced in SPARQL 1.1 such as aggregation, property paths and federated queries are present in several queries in the set. More notably, Wikidata contains the only instances of federated queries, in over 40% of the total queries, many calling an internal labelling service particular to Wikidata. Moreover, the same set contains property paths in over 25% of all queries.

BGPs in Real Queries

In addition to the queries mentioned in the previous section, we extracted all BGPs contained in each query. This was done in order to test the performance of our algorithm in applications that need to find commonly-used basic graph patterns, such as caching systems.

9.1.2 Synthetic Queries

In order to test the limitations of our algorithm, we needed queries with specific characteristics that are unlikely to appear in real-world queries. Because of this, we have designed programs (as described in Chapter 8) that construct synthetic queries. In this section, we describe these types of queries, as well as the aspect of our algorithm they are used to test.

Synthetic BGPs

In order to better test the proper functionality of the canonicalisation process, a series of synthetic basic graph patterns were created using the classes from the `generate` package described in Section 8.2.3. These synthetic BGPs are based on well-known types of graphs, where we add a triple pattern $(v_i, \texttt{:p}, v_j)$ for every pair of adjacent nodes i and j in the graph.

To ensure that the leaning process removes the largest number of triples (i.e. to represent difficult cases), we enclose the graph pattern in a `SELECT DISTINCT` V query, where V is a set of randomly selected variables.

We describe the following types of graphs that were used on our experiments:

2D Grid: For $k \geq 2$, the k-2D-grid contains k^2 nodes, each with a coordinate $(x, y) \in \mathbb{N}^2_{1\cdots k}$, where nodes with distance one are connected; this graph has a total of $2(k^2 - k)$ edges. See an example in Figure 9.1.

3D Grid: For $k \geq 2$, the k-3D-grid contains k^3 nodes, each with a coordinate $(x, y, z) \in \mathbb{N}^3_{1\cdots k}$, where nodes with distance one are connected; this graph has a total of $3(k^3 - k^2)$ edges. See an example in Figure 9.2.

Clique: For $k \geq 2$ contains k nodes that are pairwise connected; this graph has a total of $\binom{k}{2}$ edges. See an example in Figure 9.3.

Triangular: The *line graph*[1] of the k-clique, also known as a *triangular graph*: the resulting graph has $\frac{k(k-1)}{2}$ blank nodes, $\frac{k(k-1)(k-2)}{2}$ edges. See an example in Figure 9.4.

Miyazaki: This class of graphs was designed by Miyazaki [52] to enforce a worst-case exponential behaviour in `NAUTY`-style canonical labelling algorithms. For k, each graph has $20k$ nodes and $30k$ edges. See an example in Figure 9.5.

Note that under set semantics, 2D-Grid and 3D-Grid graphs collapse to a core that consists in a single undirected edge; Miyazaki graphs collapse down to a core with a 3-cycle; finally, Clique graphs are lean, so no triples are removed.

[1]The line graph of an undirected graph is constructed by converting the edges of the original graph into nodes in the line graph and connecting the nodes in the line graph if and only if the corresponding edges in the original graph share a node.

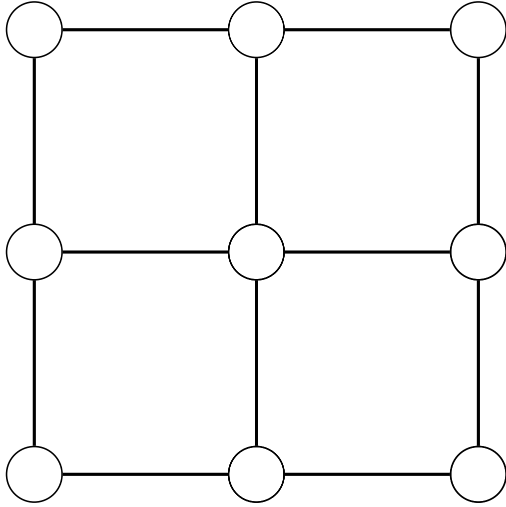


Figure 9.1: 3-2D-Grid

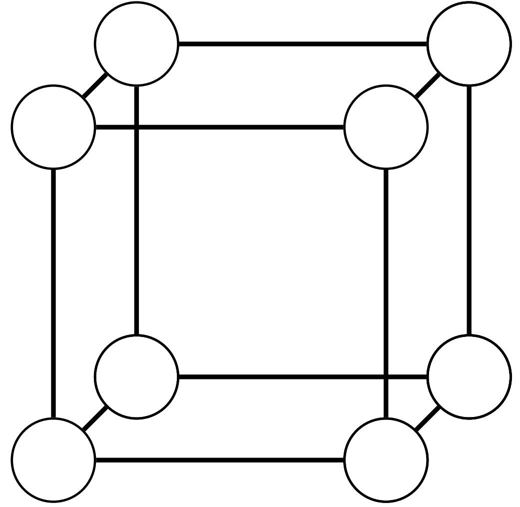


Figure 9.2: 2-3D-Grid

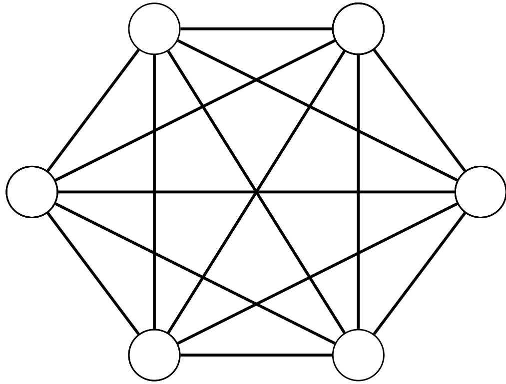


Figure 9.3: 6-clique

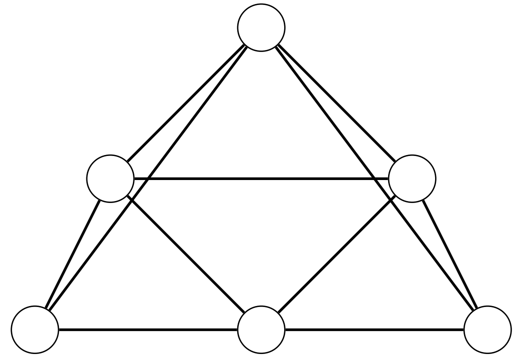


Figure 9.4: 4-triangle

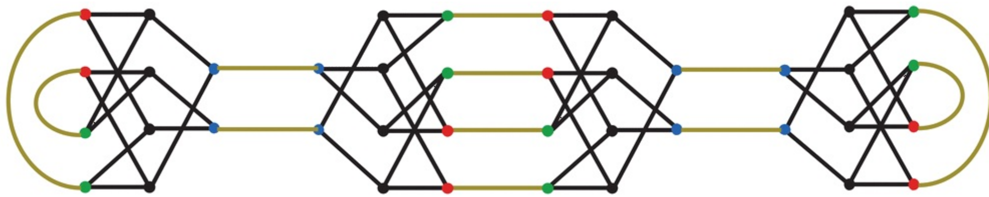


Figure 9.5: An example of a Miyazaki Graph (Source: `http://vlsicad.eecs.umich.edu/BK/SAUCY`).

Synthetic UCQs

The previous queries only represent CQs and would not test UCQ normalisation. Therefore, in order to test our method on monotone queries, we have designed synthetic UCQs using the a similar process as for the synthetic CQs; however, instead of constructing a conjunction of all the triples, we construct queries of the form $Q = (T_{11} \cup \ldots \cup T_{1m}) \bowtie (T_{21} \cup \ldots \cup T_{2m}) \bowtie \ldots \bowtie (T_{n1} \cup \ldots \cup T_{nm})$ where m is the number of unions in each disjunction, and n is the number of disjunctions. The terms T_{ij} are selected from the triples created for the synthetic queries. This process is random in the sense that we select m triples for every disjunction. This means that some of the triples may be repeated inside other disjunctions, but never in the same disjunction. An example of such a query is featured in Figure 9.6.

```
SELECT DISTINCT  ?v1
WHERE
  {   {   {   { ?v1   :p   ?v2 }
                UNION
                  { ?v1   :p   ?v3 }
              }
            UNION
              { ?v4   :p   ?v3 }
          }
        UNION
          { ?v3   :p   ?v2 }
          {   { ?v3    :p   ?v2 }
                UNION
                  { ?v3   :p   ?v2 }
              }
            UNION
              { ?v1   :p   ?v4 }
          }
        UNION
          { ?v4   :p   ?v3 }
  }
```

Figure 9.6: An example of a monotone query featuring a join of two unions of four triple patterns each.

The purpose of these artificial UCQs is to test the performance of our algorithm in difficult cases, particularly the UCQ rewriting and UCQ minimisation. For this evaluation, we consider the case when the query is in its conjunctive normal form.

9.2 Machine Specifications

Experiments were run remotely on a single machine with two Intel Xeon E5-2609 V3 CPUs and 32GB of RAM running Debian v.7.11. The max memory size was set to 10GB during execution.

9.3 Results

We now present the results of the experiments that were carried out to test the performance of our software on all of the aspects denoted by our questions.

9.3.1 Real-world Queries

We begin by showing the results of our algorithm over the real-world queries from the LSQ dataset described in Section 9.1.1, starting with the runtimes for each step of the algorithm, followed by the number of additional duplicates detected compared to our baseline method.

Runtimes

Figure 9.7 shows a box plot of the runtimes for each step of the algorithm: rewriting to a normal form (REWRITE), graph representation (GRAPH), minimisation (MINIMISE), and canonical labelling (LABEL); finally, we also show the total runtimes (TOTAL TIME). It is important to highlight that the x-axis is in logarithmic scale.

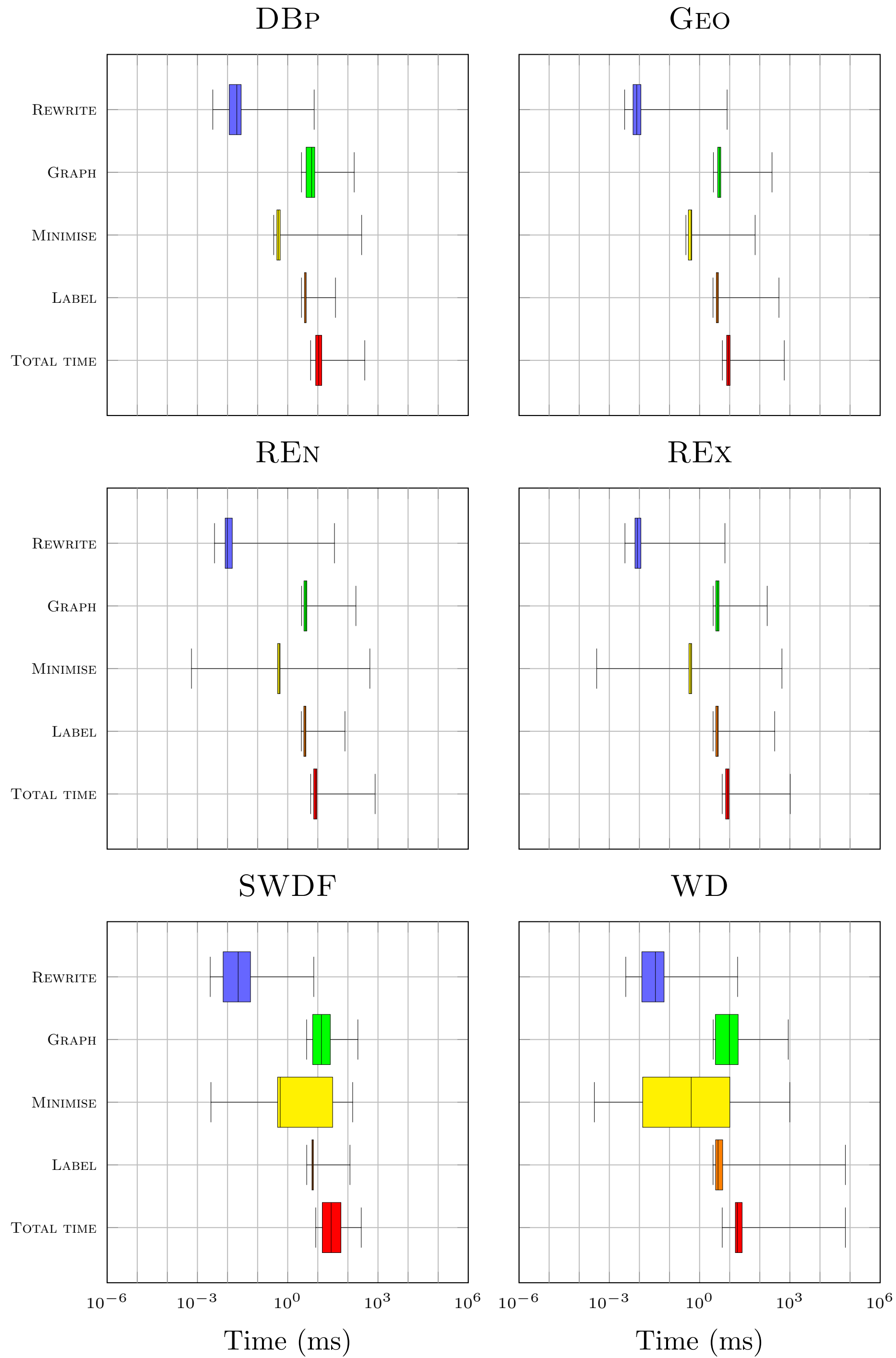


Figure 9.7: Runtimes for each step of the canonicalisation algorithm

We now present the results of performing the full canonicalisation method on all the queries in our test group. The results in Figure 9.7 indicate that the construction of the r-graph is the step that takes the most time on average, followed by the canonical labelling. This is an expected result since both the graph representation and the canonical labeling scale with the size of the graph, and both the SWDF and WD 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 WD sets.

In fact, in some occasions the runtime is dominated by the minimisation time rather than the labeling time. Despite the fact that our algorithm is doubly-exponential most individual queries were processed in just over 11ms (median). This is due to the simplicity of most real-world queries.

Following this, we show the results of the canonicalisation method over the set of all BGPs extracted from the queries in the LSQ dataset.

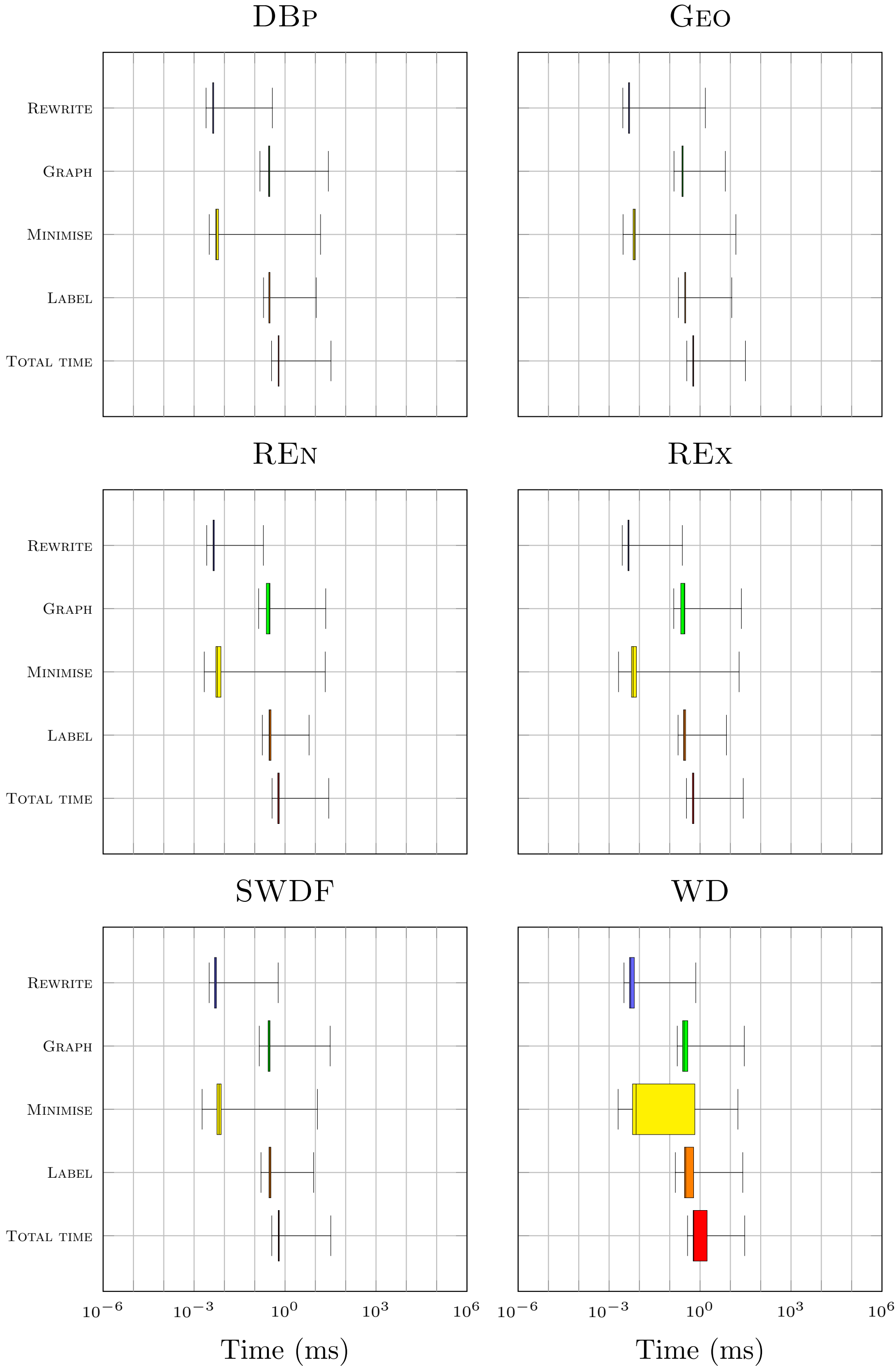


Figure 9.8: Runtimes for each step of the canonicalisation algorithm on BGPs

The results shown in Figure 9.8 present a similar trend to those in Figure 9.7, except that at no point does the canonical labelling dominate the runtime of the full process. In addition, results show that all BGPs can be processed in under a second, with at least 75% of them being processed in under a millisecond. The one notable exception to the general trend is the minimisation step for the WD set, where the variance between runtimes is much more pronounced than in any of the other sets. These results are likely explained by the fact that BGPs in real-world queries often contain very few triple patterns.

Duplicates Detected

In this section, we describe the performance of our algorithm in finding additional duplicate queries than the baseline methods we have defined. We begin by describing our rubric, and then present our results.

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.

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

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

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

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

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

Table 9.2 shows the total number of duplicates found, while Table 9.3 shows the most duplicates found of a single query (below) by each of the methods defined at the beginning of this section. It is worth noting that the results in Table 9.2 indicate that the canonical labelling is enough to detect more duplicates than our baseline method. Subsequently, the rewriting of the queries and the minimisation do not increase this number in any significant way, except for the WD set where the rewriting finds almost 100 additional duplicates. 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.

Analogously, Table 9.4 and Table 9.5 show the total number of duplicate BGPs, and the most number of duplicate BGPs, respectively, found by the methods described at the beginning of this section. Once again, we see that the canonical labelling is the step of our algorithm that finds the most additional duplicates, ranging from only 32 (DBp) up to 3,364

Table 9.2: Total number of duplicates found by each method

Query set	**Raw**	**Parse**	**Label**	**Rewrite**	**Full**
DBp	250,940	251,283	251,315	251,315	251,315
Geo	723,116	736,331	739,695	739,700	739,702
REn	142,032	143,523	144,007	144,007	144,008
REx	299,892	301,419	301,910	301,910	301,911
SWDF	53,061	53,263	53,388	53,388	53,388
WD	683,132	686,453	687,654	687,751	687,760

Table 9.3: Most duplicates of a single query found by each method

Query set	**Raw**	**Parse**	**Label**	**Rewrite**	**Full**
DBp	5,464	5,514	5,514	5,514	5,514
Geo	22,582	31,379	40,744	40,744	40,744
REn	3,814	3,814	3,814	3,814	3,814
REx	14,690	14,910	14,910	14,910	14,910
SWDF	2,388	2,633	4,938	4,938	4,938
WD	232,339	232,339	232,339	232,339	232,339

Table 9.4: Total number of duplicate BGPs found by each method

Query set	**Raw**	**Parse**	**Label**	**Rewrite**	**Full**
DBp	250,940	251,283	251,315	251,315	251,315
Geo	723,116	736,331	739,695	739,700	739,702
REn	142,032	143,523	144,007	144,007	144,008
REx	299,892	301,419	301,910	301,910	301,911
SWDF	53,061	53,263	53,388	53,388	53,388
WD	683,132	686,453	687,654	687,751	687,760

Table 9.5: Most duplicates of a single BGP found by each method

Query set	**Raw**	**Parse**	**Label**	**Rewrite**	**Full**
DBp	5,464	5,514	5,514	5,514	5,514
Geo	22,582	31,379	40,744	40,744	40,744
REn	3,814	3,814	3,814	3,814	3,814
REx	14,690	14,910	14,910	14,910	14,910
SWDF	2,388	2,633	4,938	4,938	4,938
WD	232,339	232,339	232,339	232,339	232,339

(Geo). However, the rewriting and minimisation steps find only a few additional duplicates (at most 97 for WD).

9.3.2 Synthetic Queries

Synthetic CQs

We now present the results of the experiments on synthetic CQs.

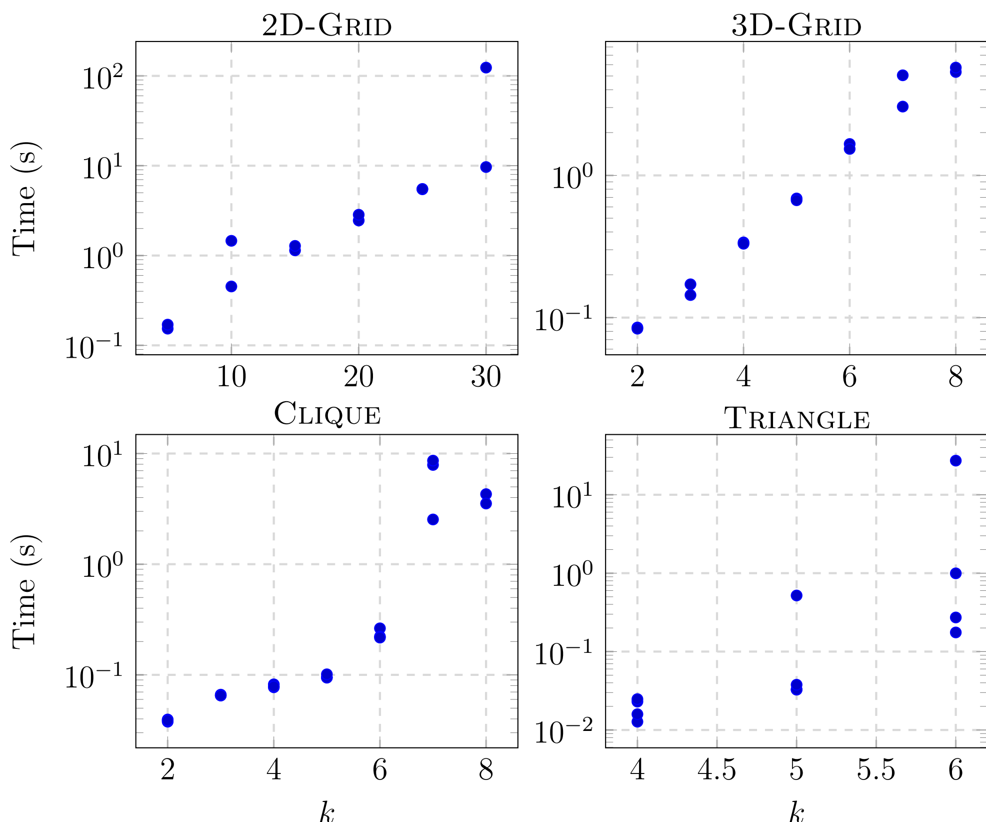


Figure 9.9: Runtimes for synthetic CQs

Figure 9.9 illustrates the runtimes – which consist in both the graph creation times and canonicalisation times – for the different types of graphs we defined previously, except for Miyazaki graphs. The latter results are missing because the algorithm would timeout for Miyazaki graphs for $k > 2$, which gave results for only a single size (249 seconds for $k = 2$). All other times are displayed in log scale. Upon analysing the data shown in Figure 9.9, we can observe that the runtimes for `2D-Grid`, `3D-Grid`, and `Triangle` graphs, increase exponentially as we increase their k value, which is an expected result since both the number of nodes and edges also increase in a linear fashion after leaning, and the canonical labeling algorithm has an exponential behaviour. On the other hand, `Clique` graphs runtimes increase by a double exponential factor, which is most likely because these graphs are already lean, and therefore the canonical labeling is done over the entire graph.

It is noteworthy that instances of `2D-Grid` for $k \leq 10$, instances of `3D-Grid` for $k \leq 4$, instances of `Clique` for $k \leq 6$, and instances of `Triangle` for $k \leq 6$ can be canonicalised in under a second. Conversely, for instances of `2D-Grid` for $k > 30$, instances of `3D-Grid` for $k > 8$, instances of `Clique` for $k > 8$, and instances of `Triangle` for $k > 6$ (and all Miyazaki graphs for $k > 2$) the algorithm would timeout.

Synthetic UCQs

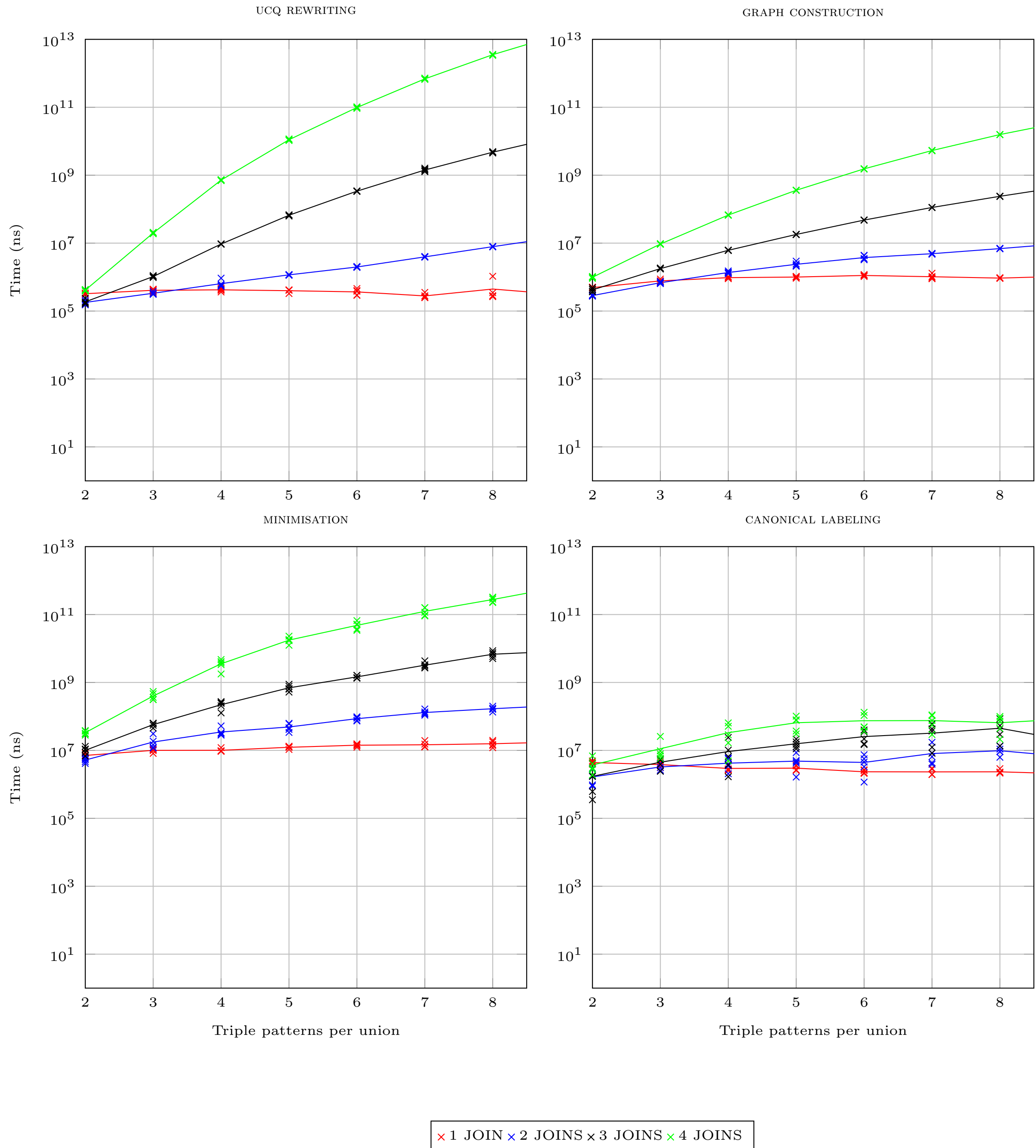


Figure 9.10: Times for UCQ stress tests

The graphs presented in Figure 9.10 show the runtimes for the synthetic UCQs, where

each has a number of conjunctions (m value) while the x-axis presents the number of unions inside each disjunctive query (n value).

It is noteworthy that increasing the value of n increases the size of the normalised form more sharply than m. Values for m vary between 1 and 64 while values for n go only up to 4 before becoming too complex to be canonicalised in mere seconds or even minutes. This makes clear that the runtimes grow exponentially, which was the expected result.

What is not considered in this experiment is the presence of multiple projections, any solution modifiers or filter expressions. We have yet to determine appropriate test cases for those features, but they are not key to the initial evaluation of this algorithm.

9.4 Comparison with Other Systems

In this section, we show the results of our experiments in comparison to other systems. Since to our knowledge, no other system exists that also canonicalises SPARQL queries, we instead compare the ability to find duplicate (equivalent) queries in a predetermined set of queries. To do this, we have found systems that check for containment between SPARQL queries.

Table 9.6: UCQ features supported by SPARQL Algebra (SA), Alternating Free two-way μ-calculus (AFMU), Tree Solver (TS) and Jena-SPARQL-API Graph-isomorphism (JSAG); note that ABGP denotes Acyclic Basic Graph Patterns

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

Table 9.6 shows the features supported by each of the containment checking systems for SPARQL found in our study. We note that none of the systems support features other than those in monotone queries.

Figure 9.11 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. Both sets of queries were obtained from the `JSAG` project, which provided these queries as a benchmark for checking containment. `SA1`, `JSAG1` and `QCan1` correspond to the tests on the set of conjunctive queries without projection, whereas `SA2`, `JSAG2` and `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

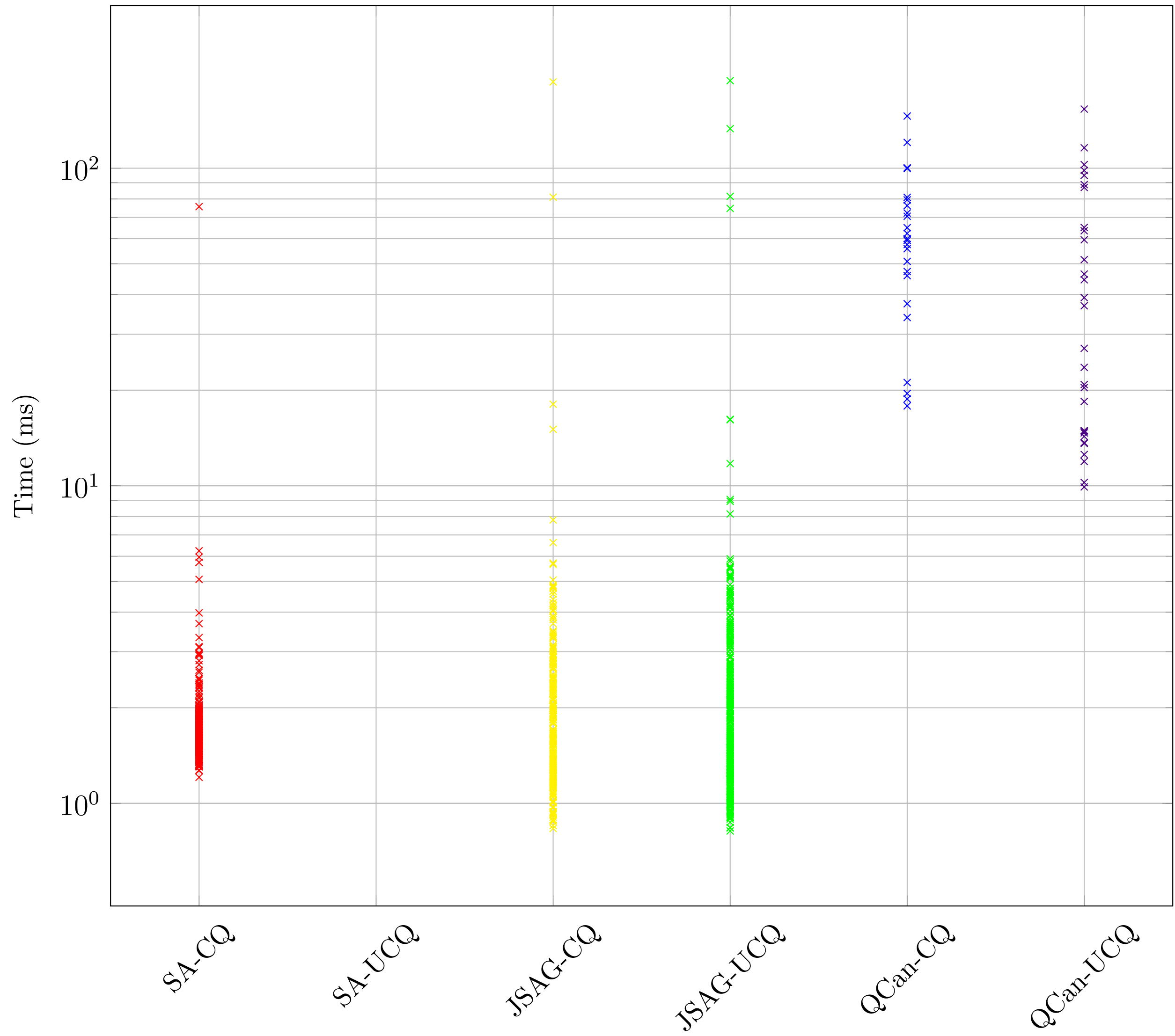


Figure 9.11: Runtimes for JSAG, SA and QCan

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.

Chapter 10

Use-cases

In this chapter, we describe some of the use-cases for which we predict our canonicalisation technique may prove useful. These range from query caching and processing query logs, both of which have already shown promising results, to more speculative use-cases such as normalising automated queries, and a *query taxonomy*. We now discuss each of these use-cases.

10.1 Query Caching

Canonicalisation may have applications in query caching systems, where it may improve the cache hit rate by detecting more congruences. We may compute the canonical form of sent queries, and store them if they haven't been seen before, or retrieve them from the cache otherwise. However, caching full queries may prove too specific, and thus there may potentially be no cache hits. Therefore, it may be more practical to cache common sub-queries rather than full queries.

Jena Caching is a caching system under development by a Master's student that stores results for BGPs found in queries to improve the answer times for future queries. This system notably caches results for additional sub-BGPs based on the BGPs that appeared in previous queries. In addition, the system uses the canonicalisation techniques and software developed in the context of this thesis. The initial idea was to compute the powerset of all triple patterns in a given BGP. In other words, if $B = \{t_1, \cdots, t_n\}$, then the results for a total of $2^n - 1$ BGPs would potentially be cached, if seen before, after canonicalisation was applied. However, this naïve approach ultimately caused such an overhead that it didn't improve the performance of the system over the baseline of evaluating the query without caching. Because of this, the caching strategy was altered to cache sub-BGPs based on the *selectivity* of each triple pattern, as computed by Jena. Triple patterns that are more selective will return fewer results than those that are less selective. To begin, the results for a basic graph pattern containing only the most selective triple pattern would be cached. Without loss of generality, let us assume that $(t_1, \cdots, t_n)$ is sorted by selectivity, where t_1 is the most selective triple pattern, and t_2 is the next most selective triple pattern that shares a variable

with a triple pattern preceding it (or simply the next most selective triple pattern if no such triple pattern exists). In the next step, the second most selective triple pattern t_2 would be added to the basic graph pattern, resulting in $B' = \{t_1, t_2\}$, and the results for B' would be checked in the cache (and potentially cached if not already in the cache, if seen before, and if chosen by the particular caching policy). Each subsequent step would add the next most selective triple pattern, until all triple patterns have been added. As a result, a total of n BGPs would be cached, instead of the $2^n - 1$ BGPs of the naïve approach. Furthermore, this approach minimises the total number of results stored because the process begins with the most selective triple pattern. In this process, our canonicalisation method is used to canonicalise these sub-BGPs in order to improve the cache hit rate.

10.2 Automated Queries

Frameworks such as the *Ontology-based Data Access* (*OBDA*) generate queries with redundant joins and unions. Our method may be used to process queries to minimise these queries. Furthermore, these queries tend to be monotone, the fragment for which our method is sound and complete. Automated query generators may produce SPARQL queries with redundant joins and unions [79].

10.3 Benchmarking Queries

The analyses of query logs is useful for several applications such as: studying the types of queries and features sent to a system, determining the data that is most frequently requested, the overall performance of a system under certain parameters, etc. However, because of the large number of queries received by these systems [7], these analyses must be done by machines. Furthermore, there is no guarantee that these queries have been processed in a way that allow for an easy analysis; in fact in our own study, several queries from the Wikidata query logs were excluded from our experiments because they were syntactically invalid, among other reasons. Our system can be used to process the queries in these logs to normalise them, reduce the variance and size of the query log by grouping all equivalent or congruent queries, and remove variable that are always unbound.

Of the applications we have mentioned, we highlight the ability to study the behaviour of queries sent to a system. This is potentially useful for the creation of query benchmarks for new systems that need to simulate cases that are likely to occur in practice. It is possible to create new queries that are based in queries that have been sent by real users to systems that serve a similar purpose.

We now illustrate an example: as part of our work on a new knowledge base for academic publications and citations, BibKG, we had to adapt queries found in query logs from Wikidata and SWDF. This process was automatised because these logs contained hundreds of thousands of queries, some of which were not useful.

SPARQL, much like RDF in Turtle syntax, defines prefixes for IRIs that make their text

forms much more human-readable. Often these prefixes refer to namespaces for IRIs of a specific service or endpoint (for example, it is standard to define the prefix `rdf` for IRIs beginning with `http://rdf.org/`).

In order to adapt these queries to this new knowledge graph, we had to map IRIs that refer to certain properties to our namespace. For instance, Wikidata uses the IRI `http://www.wikidata.org/prop/statement/P304` to refer to the number of pages of a publication, whereas BibKG uses the IRI `http://bibkg.imfd.cl/voc#pages`. Therefore, as part of our automated process, any instances of the former IRI are replaced by the latter.

However, we found that this process was not enough to produce valid SPARQL queries for experiments. Some of the queries contained projected variables that are never bound, as well as other redundancies. In addition, a significant number of the queries in these logs are duplicated instances of very simple basic graph patterns.

We found that QCan suited this work perfectly, since it addresses both of these issues as part of the canonicalisation process. By preprocessing the queries, the total number of queries decreased by up to a three-hundredth.

10.4 Query Taxonomy

Another use-case that we explored was the creation of a taxonomy of query patterns, in order to complement the creation of benchmark queries, by helping to structure and understand the contents of massive logs. Starting with high-level, more general query patterns at the root, each level of the taxonomy would add canonicalised abstractions of queries that partition the log into finer and finer equivalence classes, eventually ending in the raw input queries at the leaves.

To better illustrate this idea, we begin with the query shown in Figure 10.1:

```
SELECT DISTINCT ?resource ?uri ?wtitle ?image
WHERE {
  {
    { ?resource foaf:page <http://en.wikipedia.org/wiki/
         Calendar> }
    UNION
    { ?resource foaf:page <http://en.wikipedia.org/wiki/
         Trackback> }
  }
  ?resource foaf:page ?uri .
  ?resource rdfs:label ?wtitle .
  FILTER langMatches(lang(?wtitle), 'en').
  OPTIONAL
    { ?resource foaf:depiction ?image }
  }
```

Figure 10.1: A SPARQL query based on a simplified version of a real query.

The query shown in Figure 10.1 is based on a real query found in a log. This query finds resources in a knowledge graph whose URI are either `http://en.wikipedia.org/wiki/`

`Calendar` or `http://en.wikipedia.org/wiki/Trackback`, their labels in English, and any associated images, if available.

```
SELECT DISTINCT ?var1 ?var2 ?var3 ?var4
WHERE {
  {
    { ?var1 foaf:page <http://en.wikipedia.org/wiki/
         Calendar> }
    UNION
    { ?var1 foaf:page <http://en.wikipedia.org/wiki/
         Trackback> }
  }
  ?var1 foaf:page ?var2 .
  ?var1 rdfs:label ?var3 .
  FILTER langMatches(lang(?var3), 'en').
  OPTIONAL
    { ?var1 foaf:depiction ?var4 }
  }
```

Figure 10.2: The running example now with canonical labels.

The query shown in Figure 10.2 is the same query that appears in Figure 10.1, but with canonical labels. In this taxon, all queries with the same structure will be equivalent.

```
SELECT DISTINCT ?var1 ?var2 ?var3 ?var4
WHERE {
  { ?var1 foaf:page <http://en.wikipedia.org/wiki/
       Calendar> .
  ?var1 foaf:page ?var2 .
  ?var1 rdfs:label ?var3 .
  OPTIONAL { ?var1 foaf:depiction ?var4 } }
  UNION
  { ?var1 foaf:page <http://en.wikipedia.org/wiki/
       Trackback> .
  ?var1 foaf:page ?var2 .
  ?var1 rdfs:label ?var3 .
  OPTIONAL { ?var1 foaf:depiction ?var4 } }
  FILTER langMatches(lang(?var3), 'en')
  }
```

Figure 10.3: The running example now in a UCQ normal form, and having pushed the optional pattern inside.

The query shown in Figure 10.3 is the running example normalised according to the rules described in Sections 5.1.1 and 5.2.1. In this taxon, we will find that in addition to all the equivalences found in the previous taxon, we also find more equivalences for all queries that have this normal form.

The query shown in Figure 10.4 is the same query as the one in Figure 10.3, except for the fact that the IRIs `http://en.wikipedia.org/wiki/Calendar` or `http://en.wikipedia.org/wiki/Trackback` have been replaced with fresh variables. As a result, this taxon groups all queries that have the same overall structure. Furthermore, queries in this taxon could be used to design query plans that are independent of any specific values, where variables serve as parameters to be replaced for specific queries. For instance, variables `?var5` and `?var6` – previously associated to `http://en.wikipedia.org/wiki/Calendar` and `http://en.wikipedia.org/wiki/Trackback`, respectively – may be replaced with IRIs from other domains to a similar effect.

```
SELECT DISTINCT ?var1 ?var2 ?var3 ?var4
WHERE {
  { ?var1 foaf:page ?var5 .
  ?var1 foaf:page ?var2 .
  ?var1 rdfs:label ?var3 .
  OPTIONAL { ?var1 foaf:depiction ?var4 } }
  UNION
  { ?var1 foaf:page ?var6 .
  ?var1 foaf:page ?var2 .
  ?var1 rdfs:label ?var3 .
  OPTIONAL { ?var1 foaf:depiction ?var4 } }
  FILTER langMatches(lang(?var3), 'en')
  }
```

Figure 10.4: The running example from Figure 10.3 with the IRIs replaced with variables.

```
SELECT DISTINCT ?var1 ?var2 ?var3 ?var4
WHERE {
  { ?var1 :p ?var5 .
  ?var1 :p ?var2 .
  ?var1 :p ?var3 .
  OPTIONAL { ?var1 :p ?var4 } }
  UNION
  { ?var1 :p ?var6 .
  ?var1 :p ?var2 .
  ?var1 :p ?var3 .
  OPTIONAL { ?var1 :p ?var4 } }
  FILTER langMatches(lang(?var3), 'en')
  }
```

Figure 10.5: The running example from Figure 10.4 with all predicates replaced with a generic IRI :p.

Figure 10.5 contains the running example as shown in Figure 10.4 where all terms in the predicate position have been replaced with an identical, standard IRI `?p`. This taxon will thus group all queries based on the shape of their query patterns, i.e. if we were to draw graphs based on the basic graph patterns in these queries, the graphs would be isomorphically equivalent. Queries in this taxon may allow us to analyse the features used, the size of basic graph patterns, the connectivity of basic graph patterns, etc.

Eventually, we may strip queries from all their features, until we are left with the root of all SPARQL queries: a single, generic triple pattern of the form (`?s`, `?p`, `?o`). An example of this can be seen in Figure 10.6.

```
SELECT *
WHERE {
  ?s ?p ?o .
}
```

Figure 10.6: The simplest graph pattern.

These taxons we have presented are some of the ideas we have come up with based on some of the applications mentioned earlier in this Section. However, it is clear that other taxons may be defined that are more appropriate for other applications, and as such, there is no one taxonomy to fit all possible applications. As part of our future work, we may want to formalise the definition of *query taxons* in a way that is flexible enough to represent groups of queries independent of domain.

We believe that this idea of a query taxonomy may be useful for analysing and classifying queries in large sets of queries.

In addition, we believe that it may be used for caching query *plans* instead of query results, such as the example described in Section 10.1. This is because we may store the query plans for more general queries, leaving any variables they contain as parameters to be filled depending on the specific query. We believe that query plans for similar queries may be used instead of having to compute a new query plan.

Chapter 11

Conclusion

11.1 Summary

The RDF framework is a fundamental part of the Semantic Web, providing a standard for the representation of data in a formal suitable for the goal of the Semantic Web. Because of the significant growth in usage of the Semantic Web and Linked Data, it was necessary to access this data for both users and machines. SPARQL was then introduced as the standard query language for the retrieval of RDF data, while containing most of the features that are standard for query languages based on relational algebra such as joins, unions, filters, etc.

However, as a consequence of the expressiveness of SPARQL, the same query may be expressed in different ways. This is a problem in situations where we may want to process unique queries only once, or to group queries that are equivalent, such as in query caching systems. We propose the canonicalisation of SPARQL queries to address this problem, where we compute a canonical form for queries such that any two equivalent or congruent queries will have the same canonical form. However, because the equivalence decision problem for the full language is undecidable, we are unable to compute a canonical form that meets this requirement for all SPARQL queries. Instead, we propose a "best-effort" canonicalisation method that is *sound*, meaning that a query is congruent to its canonical form, for the full language, and *complete*, meaning that any two congruent queries will have the same canonical form, for all monotone SPARQL queries (i.e. queries that contain only joins, unions and projection). Furthermore, we propose that this canonicalisation has practical use despite its high worst-case complexity, under the assumption that most queries that are likely to occur in reality are simple.

Two important aspects of this canonicalisation are the iso-canonicalisation and equi-canonicalisation: the former is managed by a NAUTY style canonical labelling that produces labels that are identical for any two isomorphically equivalent RDF graphs; the latter is managed by the computation of the core of an RDF graph, in a process called *leaning*, which produces RDF graphs without any redundant triples.

The steps of our method have been divided in the following: the algebraic rewriting of the query, the representation of the query as an RDF graph (denoted *r-graph*), the minimi-

sation of monotonic fragments of the query, the iso-canonicalisation of the r-graph, and the serialisation of the canonicalised r-graph back into a canonical SPARQL query.

The first step transforms SPARQL queries in an algebraic form into equivalent normal forms, such as monotone sub-queries into unions of conjunctive queries (UCQs), introducing a first level of normalisation. The second step – the representation of the query as an RDF graph – serves to abstract away the order of operands in commutative operators (such as triple pattern in a basic graph pattern), transform binary distributive operators into n-ary operators (such as unions of unions into a single union), and allow us to apply iso-canonicalisation and equi-canonicalisation over the resulting graph. For the third step, the minimisation, we identify all UCQs, and for each UCQ we lean all the CQs it contains, eliminating all redundant triple patterns. After this, we partition the CQs in equivalence classes based on the projected variables they contain, and remove all redundant CQs in each partition. The remaining CQs are once again merged into a single, minimised UCQ. Subsequently, the resulting r-graph is iso-canonicalised, which allows us to determine canonical variable names that are based on the structure of the r-graph.

We have developed a system using the Apache Jena framework, BLabel package, etc., in order to provide a proof of concept of our method, by measuring the runtimes of the algorithm for real-world queries, and to provide a useful and easy-to-use system for applications that need canonicalisation (full or otherwise) of queries, such as query caching, analysis of query logs, query benchmarking, etc. Furthermore, the experiments that were carried out to prove the efficiency of the algorithm in real-world queries show that the algorithm is practical despite the theoretical complexity of the problem. In addition, the point at which the algorithm breaks and can no longer process a query in a reasonable time was never reached in practice, and instead had to be enforced using synthetic queries. Finally, the number of duplicated queries our method can detect is indeed higher than the number of duplicated queries that syntactic equivalence tools can detect.

11.2 Key Results

Our hypothesis states that we could design and implement a canonicalisation algorithm for SPARQL queries that:

- Is practical in a real-world setting, despite the high complexity of the problem.
- Is *sound* for all SPARQL queries, and *complete* for monotone queries.
- Finds more duplicates than baseline syntactic methods.

To address the first requirement, we point to the results presented in Section 9.3.1, which show that the vast majority of real-world queries were canonicalised in under 15 milliseconds, despite the fact that the algorithm has a high cost in the worst-case scenario. This is because queries that enforce a worst-case scenario appear to be very unlikely to occur in a real setting (based on currently available logs), which was made further evident by the fact that the algorithm in our experiments only failed for difficult *synthetic* cases.

To address our second hypothesis, we point to the proofs presented in Chapter 6, which show that the algorithm is sound for all SPARQL queries because each step of the canonicalisation algorithm preserves congruence, and that the algorithm is complete for monotone queries, but will miss equivalences and congruences for larger fragments of SPARQL.

Finally, to address our third hypothesis, we point to the results presented in Section 9.3.1, which show that our algorithm is able to find more duplicates in query logs than the baseline methods we defined. We highlight the fact that the majority of additional duplicates are found because of the canonical labelling of variables, whereas the minimisation of monotone fragments finds only a few additional duplicates. This is most likely because real-world queries tend to be simple, and are unlikely to contain any redundancies. On the other hand, as stated in Section 10.2, certain automated systems may produce queries that contain redundancies, so we believe our system may benefit applications that make use of said systems.

11.3 Discussion

Aside from the research contributions previously highlighted, a number of other practical contributions and artifacts have also resulted from this work:

- A system that allows for the canonicalisation of SPARQL 1.1 queries in a reasonable time.
- Test-cases to check the correctness of our canonicalisation algorithm that, though unable to prove correctness for all possible SPARQL queries, serve as a benchmark for any future works that may attempt to canonicalise SPARQL queries.
- The formalisation of the entirety of SPARQL 1.1, filling a gap in the state of art.

However, despite our best efforts, we have identified some limitations of this study:

Although the runtimes are well under a second, for applications such as query caching, it may sometimes be more efficient to run the queries again. The developer of Jena Caching has opted to turn off the minimisation step of the canonicalisation, assuming that the vast majority of additional cache hits would be a result of the canonical labelling rather than the minimisation. This is most likely the case because the sub-queries that are cached are basic graph patterns instead of full monotone queries, and are therefore not likely to contain any redundancies; also, as observed previously, queries in practice tend to be quite simple, and thus not prone to being redundant.. This suggests that further study is warranted to determine the trade-off between more complete canonicalisation (that will capture more congruences) and less complete canonicalisation that might miss some congruences, but run more efficiently. We anticipate that this trade-off may also depend on the specific application in question, rather than there being a one-fits-all answer.

Furthermore, as stated, our canonicalisation is only complete for monotone queries, which is a significant portion of SPARQL, but leaves many queries that are likely to occur in a real-world setting. These include queries with built-in expressions (such as those in `FILTER`

expressions), optional graph patterns, and property paths. Although both built-in expressions and optional graph patterns make the equivalence decision problem undecidable, there may exist certain fragments of SPARQL that are still decidable, and may allow a canonical form, for said features with certain restrictions. In addition, we have studied fragments of SPARQL with property paths that allow canonical forms, and the equivalence decision problem is decidable for this fragment, but may not be practical.

Nevertheless, these limitations present an opportunity for further research and future work.

11.4 Future Work

Perhaps we can design a form of *adaptive* canonicalisation where variations of the canonicalisation algorithm may be designed and extended based on the features contained by a specific query. For instance, if the input query is a BGP we may choose to only canonically label the query instead of minimising, such as in the case of Jena Caching mentioned earlier. Our work on the canonicalisation of SPARQL queries with property paths is incomplete, and we can see that further research can be done on this topic. We may yet find a larger fragment of SPARQL that allows property paths with certain limitations, and may still allow a canonical form. Furthermore, we have introduced the idea of a *query taxonomy*, though it is in its early stages. This idea requires a formalisation before it can be applied to new systems. Finally, the processes that comprise our canonicalisation method are not unique to SPARQL, and can therefore also be applied to other querying languages that are also widely used, and perhaps also have a larger number of simple queries than complex ones, such as SQL.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Georgii Maksimovich Adel'son-Vel'skii, Boris Yu Weisfeiler, A. A. Leman, and Igor Aleksandrovich Faradzhev. An example of a graph which has no transitive group of automorphisms. In *Doklady Akademii Nauk*, volume 185, pages 975–976. Russian Academy of Sciences, 1969.

[3] Foto N. Afrati, Matthew Damigos, and Manolis Gergatsoulis. Query containment under bag and bag-set semantics. *Inf. Process. Lett.*, 110(10):360–369, 2010.

[4] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *Web Semantics*, 7(2):57–73, 2009.

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*, 50(5), 2017.

[6] Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In *International Semantic Web Conference*, pages 114–129. Springer, 2008.

[7] Renzo Angles and Claudio Gutierrez. The Multiset Semantics of SPARQL Patterns. In *International Semantic Web Conference (ISWC)*, pages 20–36. Springer, 2016.

[8] Carlos Buil Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL web-querying infrastructure: Ready for action? In *International Semantic Web Conference (ISWC)*, pages 277–293, 2013.

[9] Marcelo Arenas and Jorge Pérez. Federation and Navigation in SPARQL 1.1. In *Reasoning Web Summer School*, volume 7487 of *LNCS*, pages 78–111. Springer, 2012.

[10] Marcelo Arenas and Martín Ugarte. Designing a query language for RDF: marrying open and closed worlds. *ACM Transactions on Database Systems (TODS)*, 42(4):1–46, 2017.

[11] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 684–697. ACM, 2016.

[12] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *PVLDB*, 11(2):149–161, 2017.

[13] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020.

[14] Dan Brickley, R.V. Guha, and Brian McBride. RDF Schema 1.1. W3C Recommendation, 2014. `http://www.w3.org/TR/rdf-schema/`.

[15] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Containment of conjunctive regular path queries with inverse. *KR*, 2000:176–185, 2000.

[16] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83, 2003.

[17] Nilesh Chakraborty, Denis Lukovnikov, Gaurav Maheshwari, Priyansh Trivedi, Jens Lehmann, and Asja Fischer. Introduction to Neural Network based Approaches for Question Answering over Knowledge Graphs. *CoRR*, abs/1907.09361, 2019.

[18] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Symposium on Theory of Computing (STOC)*, pages 77–90. ACM, 1977.

[19] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* Conjunctive Queries. In *Principles of Database Systems (PODS)*, pages 59–70. ACM Press, 1993.

[20] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. SPARQL query containment under SHI axioms. In *AAAI Conference on Artificial Intelligence*, 2012.

[21] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. Evaluating and benchmarking SPARQL query containment solvers. In *International Semantic Web Conference (ISWC)*, pages 408–423. Springer, 2013.

[22] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, 2017.

[23] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting Aggregate Queries Using Views. In *SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 155–166. ACM Press, 1999.

[24] Derek G Corneil and Calvin C Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM (JACM)*, 17(1):51–64, 1970.

[25] Richard Cyganiak. A relational algebra for SPARQL. HP Laboratories Bristol – Technical Report, September 2005. `http://shiftleft.com/mirrors/www.hpl.hp.com/techreports/2005/HPL-2005-170.pdf`.

[26] Richard Cyganiak. A relational algebra for SPARQL. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, page 35, 2005.

[27] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, February 2014. `http://www.w3.org/TR/rdf11-concepts/`.

[28] Wojciech Czerwinski, Wim Martens, Matthias Niewerth, and Pawel Parys. Minimization of tree pattern queries. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 43–54, 2016.

[29] Wojciech Czerwiński, Wim Martens, Pawel Parys, and Marcin Przybylko. The (almost) complete guide to tree pattern containment. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 117–130, 2015.

[30] Martin Dürst and Michel Suignard. Internationalized Resource Identifiers (IRIs). Request for Comments (RfC) 3987, January 2005. `https://tools.ietf.org/html/rfc3987`.

[31] Diego Figueira. Containment of UC2RPQ: the hard and easy cases. In *23rd International Conference on Database Theory (ICDT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[32] Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, editors. *International Conference on World Wide Web (WWW)*. ACM, 2015.

[33] Pierre Geneves, Nabil Layaïda, and Vojtech Knyttl. *XML Reasoning Solver User Manual*. PhD thesis, INRIA, 2011.

[34] Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Pérez. Foundations of Semantic Web databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011.

[35] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, March 2013. `http://www.w3.org/TR/sparql11-query/`.

[36] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer. W3C Recommendation, October 2009. `http://www.w3.org/TR/owl2-primer/`.

[37] Aidan Hogan. Skolemising Blank Nodes while Preserving Isomorphism. In *World Wide Web Conference (WWW)*, pages 430–440. ACM, 2015.

[38] Aidan Hogan. Canonical forms for isomorphic and equivalent RDF graphs: Algorithms for leaning and labelling blank nodes. *ACM TOW*, 11(4), 2017.

[39] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

[40] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of Conjunctive Queries: Beyond Relations as Sets. *ACM Trans. Database Syst.*, 20(3):288–324, 1995.

[41] Tommi A. Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[42] Mark Kaminski, Egor V. Kostylev, and Bernardo Cuenca Grau. Query Nesting, Assignment, and Aggregation in SPARQL 1.1. *ACM Trans. Database Syst.*, 42(3):17:1–17:46, 2017.

[43] Evgeny Kharlamov, Dag Hovland, Martin G. Skjæveland, Dimitris Bilidas, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Davide Lanti, Martin Rezk, Dmitriy Zheleznyakov, Martin Giese, Hallstein Lie, Yannis E. Ioannidis, Yannis Kotidis, Manolis Koubarakis, and Arild Waaler. Ontology Based Data Access in Statoil. *J. Web Semant.*, 44:3–36, 2017.

[44] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoc. SPARQL with Property Paths. In *International Semantic Web Conference (ISWC)*, volume 9366 of *Lecture Notes in Computer Science (LNCS)*, pages 3–18. Springer, 2015.

[45] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[46] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.*, 38(4):25:1–25:45, 2013.

[47] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graph databases with xpath. In *Proceedings of the 16th International Conference on Database Theory*, pages 129–140, 2013.

[48] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. On blank nodes. In *International semantic web conference*, pages 421–437. Springer, 2011.

[49] Brian McBride. Jena: A Semantic Web Toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[50] Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[51] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of xpath. *Journal of the ACM (JACM)*, 51(1):2–45, 2004.

[52] Takunari Miyazaki. The Complexity of McKay's Canonical Labeling Algorithm. In *Groups and Computation, II*, pages 239–256, 1997.

[53] Frank Neven and Thomas Schwentick. On the complexity of xpath containment in the presence of disjunction, dtds, and variables. *arXiv preprint cs/0606065*, 2006.

[54] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. Graph-aware, workload-adaptive SPARQL query caching. In *ACM SIGMOD International Conference on Management of Data*, pages 1777–1792. ACM, 2015.

[55] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and Complexity of SPARQL. In *International Semantic Web Conference (ISWC)*, volume 4273 of *LNCS*, pages 30–43. Springer, 2006.

[56] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[57] Reinhard Pichler and Sebastian Skritek. Containment and equivalence of well-designed SPARQL. In *Principles of Database Systems (PODS)*, pages 39–50, 2014.

[58] Axel Polleres. From SPARQL to rules (and back). In *International Conference on World Wide Web (WWW)*, pages 787–796, 2007.

[59] Axel Polleres and Johannes Peter Wallner. On the relation between SPARQL1.1 and Answer Set Programming. *J. Appl. Non Class. Logics*, 23(1–2):159–212, 2013.

[60] Eric Prud'hommeaux and Carlos Buil-Aranda. SPARQL 1.1 Federated Query. W3C Recommendation, March 2013. `https://www.w3.org/TR/sparql11-federated-query/`.

[61] Juan L Reutter. Containment of nested regular expressions. *arXiv preprint arXiv:1304.2637*, 2013.

[62] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences Among Relational Expressions with the Union and Difference Operators. *J. ACM*, 27(4):633–655, 1980.

[63] Jaime Salas and Aidan Hogan. Canonicalisation of monotone SPARQL queries. In *International Semantic Web Conference (ISWC)*, pages 600–616, 2018.

[64] Jaime Salas and Aidan Hogan. Semantics and canonicalisation of SPARQL 1.1. *Semantic Web*, (Preprint):1–65, 2021.

[65] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: the linked SPARQL queries dataset. In *International Semantic Web Conference (ISWC)*, 2015.

[66] Muhammad Saleem, Claus Stadler, Qaiser Mehmood, Jens Lehmann, and Axel-Cyrille Ngonga Ngomo. Sqcframework: SPARQL query containment benchmark generation framework. In *Knowledge Capture Conference (K-CAP)*, pages 28:1–28:8, 2017.

[67] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *International Conference on Database Theory (ICDT)*, pages 4–33, 2010.

[68] Guus Schreiber and Yves Raimond. RDF 1.1 Primer. W3C Working Group Note, June 2014. `http://www.w3.org/TR/rdf11-primer/`.

[69] Claus Stadler, Jens Lehmann, Konrad Höffner, and Sören Auer. LinkedGeoData: A core for a web of spatial open data. *Semantic Web*, 3(4):333–354, 2012.

[70] Claus Stadler, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, and Jens 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.

[71] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *International Conference on World Wide Web (WWW)*, pages 595–604. ACM, 2008.

[72] Yoshinori Tanabe, Koichi Takahashi, Mitsuharu Yamamoto, Akihiko Tozawa, and Masami 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.

[73] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[74] Boris 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.

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

[76] Gregory Todd Williams and Jesse Weaver. Enabling fine-grained HTTP caching of SPARQL query results. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, pages 762–777, 2011.

[77] Peter T Wood. Minimising simple xpath expressions. In *WebDB*, pages 13–18. Citeseer, 2001.

[78] Peter T Wood. Containment for xpath fragments under dtd constraints. In *Database Theory—ICDT 2003: 9th International Conference Siena, Italy, January 8–10, 2003 Proceedings 9*, pages 300–314. Springer, 2003.

[79] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. Ontology-based data access: A survey. International Joint Conferences on Artificial Intelligence, 2018.

[80] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. *Proc. VLDB Endow.*, 12(11):1276–1288, 2019.