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

CANONICALISATION OF SPARQL QUERIES

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN
MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERIO CIVIL EN COMPUTACIÓN

JAIME OSVALDO SALAS TREJO

PROFESOR GUÍA:
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
CLAUDIO GUTIÉRREZ GALLARDO
JORGE PÉREZ ROJAS
JUAN REUTTER DE LA MAZA

SANTIAGO DE CHILE
2018

Resumen

SPARQL es el lenguaje de consulta estándar para RDF, definido por el World Wide Web Consortium. Presentemente, hay una gran cantidad de servicios de consulta de SPARQL en la Web. También hay una gran demanda por estos servicios diariamente. Debido a la gran cantidad de consultas que deben procesar estos servicios, los procesadores sufren una sobrecarga que podría ser reducida si fuésemos capaces de detectar consultas equivalentes. Nuestra propuesta consiste en el diseño y la implementación de un algoritmo de *canonicalización* eficiente, el cual nos permitirá computar una forma canónica para las consultas. Las consultas que sean equivalentes deben tener la misma forma canónica, lo cual hará posible detectar un mayor número de consultas duplicadas. Hemos cubierto una parte significante de SPARQL 1.0 con nuestro trabajo; principalmente consultas que tengan la forma de uniones de consultas conjuntivas. El algoritmo que hemos desarrollado realiza una canonicalización completa de las consultas que contienen estas operaciones. Para las otras operaciones, realizamos una canonicalización parcial, pues estas otras operaciones son usadas popularmente en consultas reales. Hemos diseñado experimentos para probar la correctitud y el desempeño de nuestro algoritmo en comparación contra otros métodos sintácticos. Realizamos nuestros experimentos sobre consultas reales extraídas de bitácoras de bases de datos de RDF, y también sobre consultas sintéticas que diseñamos para forzar un mal comportamiento del algoritmo. Los resultados de nuestros experimentos son excitosos, pues la mayoría de las consultas reales son procesables en un tiempo corto, menor a un segundo. El número de consultas duplicadas encontradas es considerablemente superior que las encontradas por el algoritmo base definido. Finalmente, el algoritmo deja de funcionar únicamente para las consultas sintéticas diseñadas por nosotros, y de una gran complejidad, las cuales no ocurren en la práctica. El siguiente documento presenta el trabajo hecho.

Abstract

SPARQL is the standard querying language for RDF as dictated by the World Wide Web Consortium. Currently there are many SPARQL endpoints on the Web, and many users using these services daily. The vast number of queries that are computed present some problems to these endpoints, since many of said queries are repeated, but the current techniques are unable to detect these cases. We propose an efficient canonicalisation algorithm that will allow us to compute a canonical form for queries. Congruent queries (equivalent queries that return the same answers over any database up to variable renaming) will have the same canonical form, allowing us to detect duplicate queries. We have covered a significant fragment of SPARQL 1.0 with our work, namely query patterns that are unions of conjunctive queries (UCQs). The algorithm we have developed performs a full canonicalisation for queries that contain these features. For queries with other features, we have provided a "best-effort" canonicalisation, because these features are popular in real-world queries. We designed experiments to test the correctness and performance of our algorithm against syntactic normalisation methods. We evaluated the performance of our algorithm in real-world queries extracted from logs of dedicated RDF datasets, as well as synthetic queries designed to force a worst-case scenario. The results of our experiments are highly favourable in that most of the real-world queries were supported, and were able to be canonicalised in under a second. The number of duplicate queries found was also superior than the baseline algorithm we defined. Finally, the algorithm failed to canonicalise complex synthetic queries in a reasonable amount of time, but such queries are unlikely to occur in the real world. The following document presents the work done to accomplish this goal.

Agradecimientos

Quisiera agradecerle a mis papás por brindarme amor incondicional, y darme la oportunidad de estudiar hasta este punto. Quiero agradecerle a mis hermanos por hacer mi vida en casa muy disfrutable. Quiero agradecerle a la Sra. María por todo su cariño y trabajo. Le agradezco a mi familia: a mis tíos cercanos y mis abuelos, por todo el cariño que me han dado en mi vida. Le agradezco a mis amigos por haberse vuelto parte de mi vida y hacer la estadía en la universidad una muy disfrutable. Agradezco a mis numerosos animales, presentes y pasados, por haberme entretenido mucho con sus distintas personalidades y rarezas. Quiero agradecerle a mi psicóloga por toda la ayuda que me ha brindado estos últimos años. Agradezco a mi profesor guía por su gran apoyo y guía en este trabajo. No podría haber llegado a este punto sin los mencionados.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

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) [18] which consists essentially in representing data as a set of *triples* containing a *subject*, a *predicate*, and an *object* [7].

SPARQL [9] 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 data structured as RDF. SPARQL queries are executed over a set of these graphs: a default graph, and another set of named graphs.

In order to better define the semantics of RDF, a schema was designed: Resource Description Framework Schema (referred to as RDFS hereafter) [5]. RDFS allows to create basic ontologies over the data in RDF format. Other more expressive languages exist, such as the Web Ontology Language (OWL) [10].

There has been a steady growth in 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 [1], LinkedMDB [2], and Wikidata [3] have been published on the Web as RDF. As a result, the need to perform queries has also increased. There are thus currently many SPARQL endpoints on the Web [3, 2]. A SPARQL endpoint allows users to query a knowledge base using the SPARQL language. The results of these queries are typically returned in one or more machine-processable formats. To name a few examples of popular endpoints: Wikidata

[1] `http://wiki.dbpedia.org/`
[2] `http://linkedmdb.org/`
[3] `https://www.wikidata.org`

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

Figure 1.1: Example of a SPARQL query

and DBpedia, receive a vast number of queries everyday [17].

Since SPARQL is a rich and declarative query language, offering 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 equivalent; that is, they will yield the same results over any RDF graph. This can be accomplished by using syntax shortcuts or equivalent query patterns, for example.

Let us suppose we have access to an RDF dataset that contains information about entertainment such as television and film. We wish to have a list of all actors and their salaries. The query in Figure 1.1 allows us to extract that information.

```
SELECT ?actor ?wage
WHERE{
?actor ex:acts_in ?film; ex:salary ?wage .
}
```

Figure 1.2: Example of a SPARQL query

Figure 1.2 contains the SPARQL query that results from changing the names of variables `?movie` and `salary` from the query presented in Figure 1.1 to `?film` and `?wage`, respectively. This query will return the same results as the query in Figure 1.1 for any given RDF dataset, assuming the variables are renamed in the results accordingly. We call such queries that return the same results over any dataset *semantically equivalent modulo variable names*, or simply *congruent*.

Figure 1.3 also contains congruent queries, where the query on the right contains a redundant variable and a different order in the triples listed, as well as a redundant triple pattern. A formal definition of congruent queries will be presented in a later section.

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

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

Figure 1.3: Query congruence with a redundant variable

1.2 Problem

One problem that arises with the aforementioned queries is the resulting overload on the endpoints that process these queries because for every duplicated query, the machine must perform the same routine to return the same answers. In fact, Buil et al. [3] have concluded that renowned endpoints, such as DBpedia, are offline on average about 4% of the time, ergo the importance of having methods to reduce workload on said endpoints.

One way to reduce this workload would be to avoid executing repeated queries multiple times in a limited time span. This issue is partially solved by caching the results [19] and simply returning the cached data if there's a request for a duplicated query. However, most methods only cache syntactically normalised queries, which limits the amount of results that can be reused, as we now explain.

When addressing equivalence between queries, we must mention the difference between syntactic and semantic equivalence. 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 execution of the query will produce the same result over any possible dataset.

Currently, most endpoints utilise 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. However, the queries presented in Figure 1.1 and Figure 1.2 are congruent, but not equivalent; the queries of Figure 1.3 contain redundant variables; therefore the syntactically normalised forms of neither pair would be the same.

Therefore, the main goal of this thesis is to extend caching methods to support congruent queries, which captures a broader notion of semantic query equivalence. This could allow endpoints to detect a larger number of repeated queries, thereby allowing to reuse more query answers instead of computing them again.

However, identifying queries that are congruent is hardly a trivial matter. In fact, given a large set of queries such as those sent to a given endpoint by users, identifying which queries are congruent is a challenging problem. One option would be to check each pair of queries to

determine those that are congruent. However, verifying congruence for a single pair of queries may be exponential in a worst-case scenario, and we would be performing a quadratic number of said operations. Evidently, such a solution is not viable for large-scale applications: even if the exponential worst-case behaviour is never encountered for checking a pair of queries, quadratic checks would not be possible for a large set of queries.

Taking into account the concept of normalisation as a solution for syntactic equivalence, we introduce the concept of canonicalisation. To canonicalise means to convert data that can have multiple representations into a single, standard form which is called its canonical form.

That being the case, an alternative solution would 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 expression in their canonical form.

The idea of canonicalising every possible SPARQL query seems optimistic, to say the least. In fact, the problem of determining if two queries give the same answers over any database, and are thus semantically equivalent, has been well-studied in the theoretical literature. For SPARQL, the equivalence problem is NP-hard even for well-designed queries with basic features [1]. For the full language, equivalence is undecidable.

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 [17]. Therefore, we can reduce the scope of the canonicalisation to simpler queries and use existing efficient graph isomorphism algorithms to do a best-effort canonicalisation.

Proposals for practical graph isomorphism algorithms have been studied as far back as the 60's [6], and as recently as a few years ago [13]. 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.

We anticipate some of the following use-cases and optimisations:

- improving the performance of caches, where results for queries that are congruent to previously computed queries can be reused rather than having to be computed again;
- improving the analysis of query logs by finding a wider range of duplicate queries rather than with, for example, variations in the naming of variables;
- performing hashing or signing of SPARQL queries where hashes/signatures reflect the semantics of the query (the results returned) rather than the surface syntax, such that queries with the same effect have the same signature;
- simplifying the query optimisation process, where optimisations can be designed for queries in their canonical form rather than having to cover all possible variations.

1.3 Hypothesis and Methodology

Our hypothesis is that it is possible to design and implement an automated, efficient algorithm for canonicalising queries expressed in a core fragment of SPARQL based on existing algorithms for canonicalising RDF graphs, and that this fragment covers a majority of real-world queries.

The technical problem we tackle in this work will be to design said algorithm to efficiently canonicalise SPARQL queries based on existing techniques for canonicalising RDF graphs.

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

(i) Research of literature

(ii) Design

(iii) Implementation

(iv) Evaluation

(v) Iterate from point (ii)

Consequently, the suggested methodology is iterative in nature. Feedback obtained after the evaluation stage 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 of public SPARQL endpoints.

A summary of the results of this thesis is available in the following paper, which has been accepted for ISWC 2018 [12].

Chapter 2

Background

In this chapter, we provide some preliminaries relating to this thesis in the areas of graph theory, RDF definitions and properties, and SPARQL definitions and properties.

2.1 Graph Theory

RDF represents data in a graph form, and, as part of our work, we represent SPARQL queries as RDF graphs. As a result, for the purpose of this work, it is imperative we describe certain concepts of graph theory.

A graph is a combination of vertices $v \in V$ joined by edges $e \in E$. As a result, a graph is a pair of sets (V, E).

Definition 2.1.1 *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'.*

Definition 2.1.2 *An* undirected *graph $G = (V, E)$ is a graph where if $(v, v') \in E$, then it follows that $(v', v) \in E$. That is, the edges are unordered pairs of vertexes.*

Definition 2.1.3 *Given two undirected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, 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. *If graph G and graph H are isomorphic, it is written $G \cong H$.*

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`$) =$ `_:h`, the graph on the right can be obtained by mapping each node in the graph on the left using β.

The graph isomorphism decision problem is not known to belong to P nor to be NP-complete.

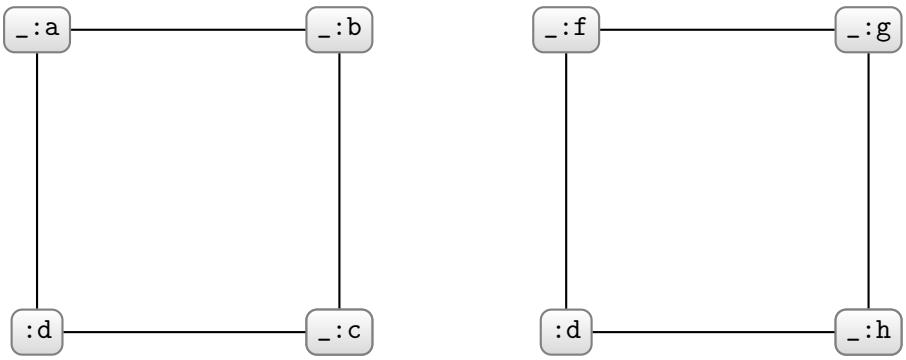

Figure 2.1: Isomorphism example

Some authors have chosen to place it in its own complexity space: GI-complete.

A more general notion than isomorphism is that of homomorphism. Both are mappings that preserve the structure of a graph. While an isomorphism must map nodes one-to-one, a homomorphism may map many nodes to one. Furthermore, while an isomorphism must "cover" the target graph, a homomorphism can map to a sub-graph of the target graph. More formally, we define graph homomorphism as follows.

Definition 2.1.4 Graph homomorphism *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* $(\mu(v), \mu(v')) \in E_H$*.*

We now indicate the computational complexity of the homomorphism problem, with the following well-known result.

Lemma 2.1.1. *Determining if there exists a homomorphism from* G *to* H *is* NP-complete*.*

2.2 RDF

RDF is a framework for the representation of information on the Web. The information is structured in a graph form, where each piece of information is represented as a triple (s, p, o), which can be visualised as a node labelled s connected to a node o by an arc labelled p.

The atomic unit of RDF is the *resource*. A resource denotes something that exists. It may be referred to by an IRI or a literal.

An *Internationalised Resource Identifier* (IRI from this point onwards) is used to identify resources in a manner that is generalised and standardised globally, 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 **I** as the set that contains all IRIs.

A *literal* is an element with a lexical expression, such as a person's name, a boolean value

or a number. A literal also has a datatype, which denotes the range of values it may take, such as strings, numbers, dates, etc. We define **L** as the set containing all literals.

Definition 2.2.1 *A* name *is any IRI or literal.*

A *blank node* is an element used to represent a value which is unknown, but it is known to exist. Implementations for the handling of blank nodes are dependant on the application. We define **B** as the set that contains all blank nodes.

Definition 2.2.2 *An RDF triple is a triple* (s, p, o) *such that: the subject* s *is an IRI or a blank node; the predicate* p *is an IRI; the object* o *is a name or a blank node. An RDF graph* G *is a set of triples.*

If there exists a predicate relation p between a subject s and object o, the triple $t = (s, p, o)$ is considered *true*.

An RDF graph is the conjunction of all of the triples it contains. It follows that an RDF graph is *true* if and only if all of its triples are *true*.

In order to facilitate notation, the union of all literal terms, IRIs and blank nodes $\mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$ will be written simply **ILB**. The same will be applied to unions of either of these sets. For instance, the union of all literal terms and IRIs is **IL**.

Definition 2.2.3 *A* sub-graph *of an RDF graph is a subset of the triples in the graph. A* proper sub-graph *is a proper subset of the triples in the graph.*

An RDF graph doesn't necessarily contain blank nodes; triples may consist entirely of IRIs or literals.

Definition 2.2.4 *A* ground *RDF graph is one that does not contain any blank nodes.*

The importance of ground graphs 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 ordering the triples and checking for string equivalence. The term *grounded* will appear in further chapters.

Figure 2.2 illustrates an RDF graph that contains three triples. All of the three triples refer to `Pokemon Yellow` as the subject; one of the triples states that the subject is an instance of a `videogame`; another triple states the name of the development team behind it (`:GameFreak`); and the last triple states that it was released in 1998. Note that `:videogame` and `:GameFreak` are IRIs, whereas 1998 is a literal value. Since all elements of the graph are either IRIs or literals, it is grounded.

If one wishes to describe triples in RDF, a form of *reification* must be used, where a resource is created to represent a triple. Let us consider a triple $t = (a, b, c)$. A *reification* of t can be expressed as a graph as shown in Figure 2.3:

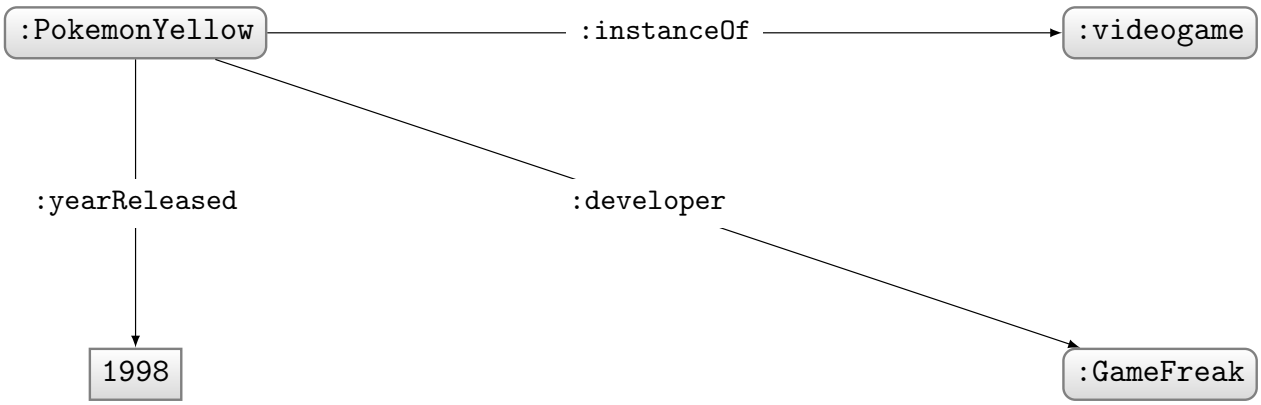

Figure 2.2: Ground graph example

```
_:t rdf:type rdf:Statement .
_:t rdf:subject :a .
_:t rdf:predicate :b .
_:t rdf:object :c .
```

Figure 2.3: Example of a reification

Where blank node `_:t` is used to identify triple t. Note that we could then make statements about the triple, such as to state a source in which it can be found.

```
_:t ex:source :s .
```

We will later use a form of reification to represent query operators.

2.2.1 Isomorphism in RDF

An RDF graph G is *isomorphic* with another RDF graph H if G and H are equal modulo blank-node labels: in other words, if the result of a one-to-one blank node mapping function over one graph is the other graph. We will state this more formally.

Definition 2.2.5 *Let* $\mu : \mathbf{ILB} \to \mathbf{ILB}$ *be a mapping of RDF terms to RDF terms. If* μ *is the identity on* **IL***, that is, if IRIs and literals remain the same, then it is called a* blank node mapping.

Such a function is applied to an RDF graph G node by node, resulting in a new graph $H = \mu(G)$ (slightly abusing notation). The resulting graph is called an *instance* of a graph G. If at least one blank node has been replaced by a name, or two blank nodes have been mapped into the same node, it is a *proper instance*.

Let μ be a blank node mapping, with its domain denoted by $dom(\mu)$ and its codomain denoted by $codom(\mu)$. If the blank nodes in $dom(\mu)$ are mapped to blank nodes in $codom(\mu)$ in a bijective manner, then μ is called a *blank node bijection*.

Definition 2.2.6 *An RDF graph G is* isomorphic *with another RDF graph H if there exists a blank node bijection μ such that $\mu(G) = H$. In such a case, μ is called an* isomorphism*.*

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*.

2.2.2 Entailment and Equivalence in RDF

RDF isomorphism only takes into account the structure of the RDF graph. On the other hand, RDF *equivalence* takes into consideration the *semantic* value of blank nodes. Blank nodes are considered values which are known to exist, but are unknown. 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 H if every possible arrangement of the world that makes G true also makes H true. When G entails H, if the truth of G is presumed or demonstrated then the truth of H is established. In such case, it is written $G \models H$. Rather than define RDF semantics formally, we remark that it is known that one can test the entailment $G \models H$ by checking if there exists a blank node mapping μ such that $\mu(H) \subseteq G$. [4]

An RDF graph G is *equivalent* to another RDF graph H if G entails H, and H entails G. In such a case, it is written $G \equiv H$.

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.

It is worth noting that ground graphs do not have any ambiguities associated to blank nodes, which means that determining if two graphs are equivalent is the same as determining if they are isomorphic. In addition, since literal terms and IRIs are unambiguous in regards to labels (i.e. two literal terms or IRIs are the same if and only if they have the same string value), determining whether a pair of ground graphs are isomorphic is reduced to ordering the triple patterns and checking for string equivalence.

Although none of the graphs shown in Figure 2.4 are isomorphic, they are all semantically equivalent. For example, $G_3 \models G_1$ is witnessed by the blank node mapping μ such that $\mu(G_1) = G_3$, $\mu(\texttt{\_:c}) = \texttt{\_:y}$, $\mu(\texttt{\_:b}) = \texttt{\_:y}$ and $\mu(\texttt{\_:a}) = \texttt{\_:x}$.

Indeed, one can appreciate that an RDF graph can model the same information as a larger RDF graph, if they are equivalent. Taking this into consideration, we will define a property of a graph for which there are no smaller RDF graphs that can model the same information.

Definition 2.2.7 *Let G be an RDF graph. We denote G as* lean *if it does not contain a proper sub-graph G' such that $G \equiv G'$. Observe that in Figure 2.4, only G_3 is lean.*

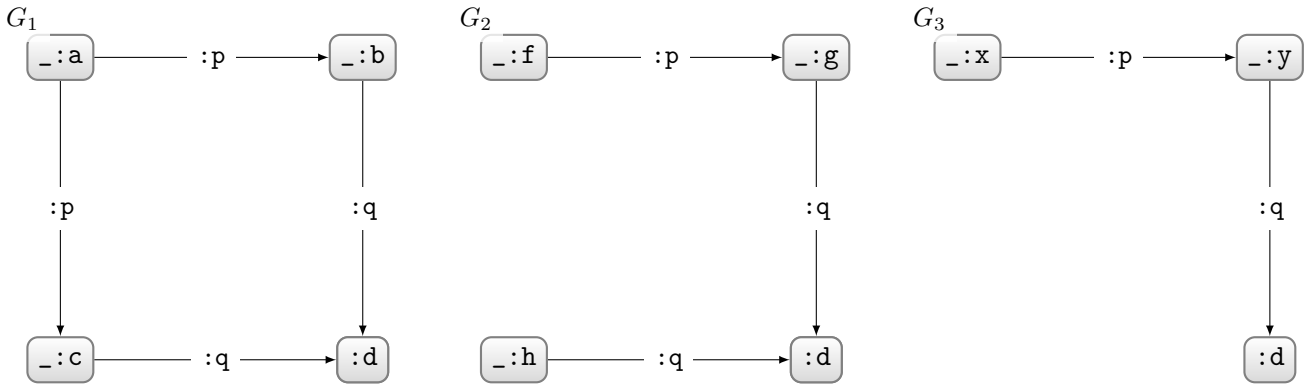

Figure 2.4: Sub-graph example

Following this, we must explain the relation between the notion of *leanness* and a *core* of an RDF graph.

Definition 2.2.8 *Let G be an RDF graph. Graph C is the core of G if and only if $C \equiv G$ and C is lean.*

In terms of the complexity of decision problems relating to equivalence, leanness and cores of RDF graphs, we can state the following.

Lemma 2.2.1. *Determining equivalence between RDF graphs is* NP-complete.

This can be reduced to two instances of the *Graph Homomorphism Problem*, which is known to be NP-complete.[4]

Lemma 2.2.2. *Determining if a graph G is lean is* coNP-complete.

Given a graph G that is not lean, this can be verified in polynomial time given a blank node mapping from G to a proper sub-graph of itself. [4]

Lemma 2.2.3. *Determining if a graph C is the core of a graph G is* DP-complete.

This problem can be reduced to verifying if both graphs are equivalent, which is NP-complete, and verifying if the witness is lean, which is coNP-complete. [4]

The full proofs for these results can be found in [4].

We will use methods to compute the core of a graph G in order to remove redundant triple patterns from queries. We denote the core of G as $\mathsf{core}(G)$ (which is unique modulo isomorphism).

2.2.3 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. However, there are some recent related works on efficiently canonicalising RDF graphs that could be leveraged for the case of SPARQL [8]. The idea is that the structure of SPARQL queries and RDF graphs are very similar and that we can adapt this algorithm for SPARQL.

Iso-canonicalisation of RDF graphs

An *iso-canonicalisation* of a graph G, $\text{ICAN}(G)$ produces a graph such that $G \cong \text{ICAN}(G)$, and for any graph G' such that $G \cong G'$ it follows that $\text{ICAN}(G) = \text{ICAN}(G')$ [9].

The main issue we have to tackle in the iso-canonicalistion process is the labelling of blank nodes. The proposed solution is the canonical labelling of blank nodes. This canonical labelling needs to be deterministic under graph isomorphism.

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 hash values are changed.

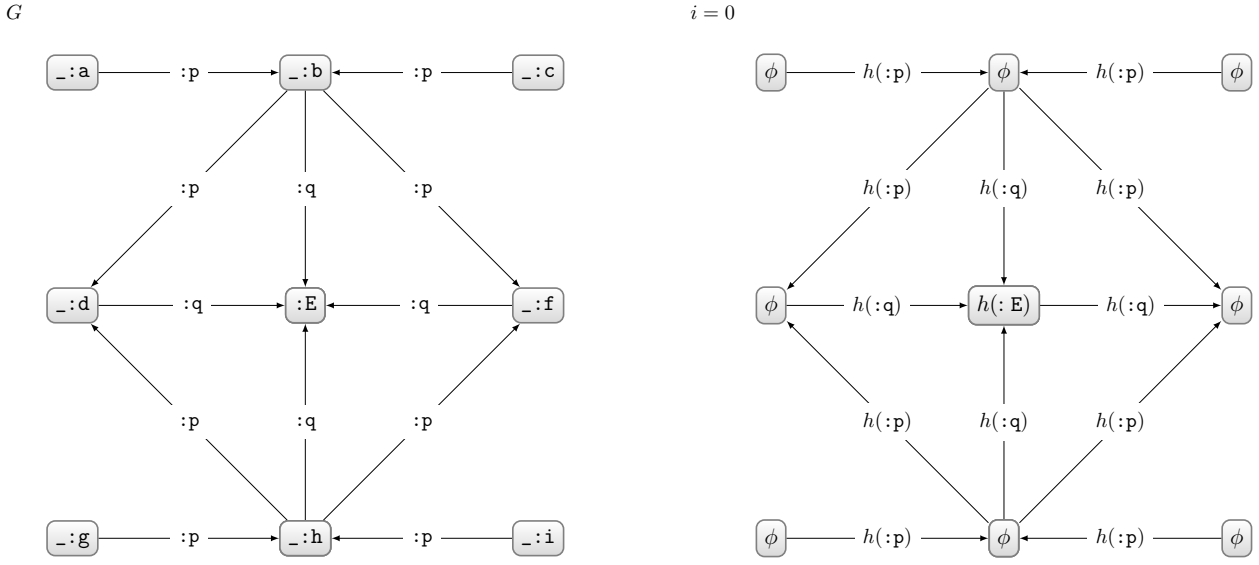

Figure 2.5: Example of labelling process

Graph G is the initial graph, with blank node labels from `_:a` to `_:h` with the exception of `:E` which corresponds to an IRI. At the beginning, all blank node labels have the same hash value ϕ. Literals and IRIs are hashed according to their values.

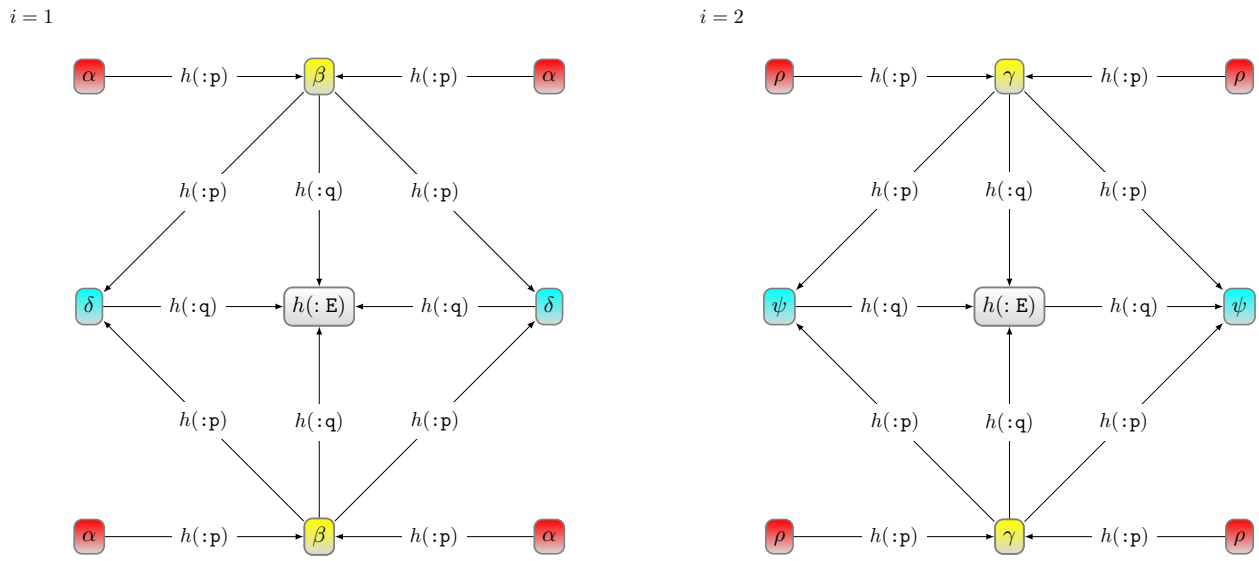

Figure 2.6: Second example of labelling process

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: $\{\texttt{\_:a}, \texttt{\_:c}, \texttt{\_:g}, \texttt{\_:i}\}$ (coloured red), $\{\texttt{\_:b}, \texttt{\_:h}\}$ (coloured yellow), and $\{\texttt{\_:d}, \texttt{\_:f}\}$ (coloured blue). This is because the blank nodes in each partition are indistinguishable by the method used, since the information used for the hashing is the same (i.e. they have the same edges). In such 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 [6]. An invariant is a property of a node in a graph such that two nodes can only be mapped by an isomorphism if they have the same invariant value.

The process continues by trying all possible deterministic options for distinguishing blank nodes in the same partition until the lowest possible graph in some ordering (e.g. alphanumerical) is found. This process is potentially exponential due to the number of possible partitions. For more details we refer to [11].

The output is an iso-canonical labelling of blank nodes for any RDF graph. We will later use this iso-canonical labelling process to assign deterministic canonical labels to the blank nodes in SPARQL queries. Since variables are represented by blank nodes, this will also assign canonical labels for variables. These canonical labels will allow us to capture a larger number of congruent queries.

Equi-canonicalisation of RDF graphs

An equi-canonicalisation of a graph G, $\text{eCan}(G)$ produces a graph such that $G \equiv \text{eCan}(G)$, in other words, the equi-canonical form of graph G retains its semantic values. Most importantly, given graphs G_1 and G_2, $\text{eCan}(G_1) = \text{eCan}(G_2)$ if and only if $G_1 \equiv G_2$. Furthermore, $\text{eCan}(G)$ is lean and thus unique up to isomorphism: in other words, it is a core of G with

canonically labelled blank nodes.

The algorithm proposed in [11] and used in this thesis to compute the core of a graph G is described:

In the first step, an initial set of blank nodes that cannot be mapped to another term by a homomorphism (i.e. they are unique) are determined. 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 term.

In a later iteration, the unique blank nodes that were determined in the first step are *grounded*, which means that for all intents and purposes they behave like an IRI or a literal term. These grounded blank nodes will appear in the final computed core.

As a consequence, other blank nodes may become unique in terms of being connected to ground blank nodes. Therefore, we may apply this grounding process 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 [17]. In case it does not occur, the algorithm computes homomorphisms from the graph into itself (*endomorphism*) until all blank nodes can be grounded. It is at this point that this algorithm may become exponential due to the number of possible homomorphisms that may be computed. However, there are certain optimisations that have been implemented, which make use of the degree of symmetry of some graphs [13]. We refer the reader to [11] for more details.

We will later use this equi-canonicalisation process to perform a minimisation of queries by removing redundant triple patterns from conjunctive queries.

2.3 SPARQL

SPARQL is in essence a graph-matching query language.

A query is composed of the following patterns:

- Let t be a *triple pattern* that allows variables in any of its components. In such a case, t is 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 F be an 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 F is a built-in expression, then $\text{FILTER}_F(Q)$ 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.

2.3.1 Algebra

Definition 2.3.1 *A* solution mapping μ *is a partial function* $\mu : \mathbf{V} \rightarrow \mathbf{IL}$

A solution to a SPARQL query is a partial mapping from variables in **V** appearing in the query to terms from **IBL** appearing in the data. Such mappings will be denoted by μ per Definition 2.3.1. Let $dom(\mu)$ denote the variables for which a mapping μ is defined.

Definition 2.3.2 *Let* μ_1 *and* μ_2 *be solution mappings. They are said to be compatible, denoted* $\mu_1 \sim \mu_2$*, if and only if for every* $v \in dom(\mu_1) \cap dom(\mu_2)$ *it holds that* $\mu_1(v) = \mu_2(v)$*.*

Let μ be a mapping and F be a built-in expression. We say that μ satisfies F, denoted $\mu \models F$ if:

- F is of the form $bound(?X)$ and $?X \in dom(\mu)$
- F is of the form $?X = c$, $?X \in dom(\mu)$ and $\mu(?X) = c$
- F is of the form $?X = ?Y$, $?X \in dom(\mu)$, $?Y \in dom(\mu)$ and $\mu(?X) = \mu(?Y)$
- F is of the form $(\neg F')$, F' is a built-in condition, and it is not the case that $\mu \models F'$
- F is of the form $(F_1 \vee F_2)$, F_1 and F_2 are built-in conditions, and $\mu \models F_1$ or $\mu \models F_2$
- F is of the form $(F_1 \wedge F_2)$, F_1 and F_2 are built-in conditions, and $\mu \models F_1$ and $\mu \models F_2$

Finally, let M, M_1 and M_2 denote sets of mappings. We define an algebra consisting of the join, union, difference, left-outer-join, selection and projection operators, respectively, as follows:

$M_1 \bowtie M_2 := \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1,\ \mu_2 \in M_2,\ \mu_1 \sim \mu_2\}$

$M_1 \cup M_2 := \{\mu \mid \mu \in M_1 \text{ or } \mu \in M_2\}$

$M_1 \setminus M_2 := \{\mu_1 \in M_1 \mid \nexists \mu_2 \in M_2 : \mu_1 \sim \mu_2\}$

$M_1 \mathbin{⟕} M_2 := (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

$\sigma_F(M) := \{\mu \in M \mid \mu \models F\}$

$\pi_V(M) := \{\mu' \mid \exists \mu \in M \ :\ \mu' \subseteq \mu,\ dom(\mu') = V \cap dom(\mu)\}$

2.3.2 Semantics

Let Q denote a SPARQL query, and $Q(G)$ the evaluation of Q over an RDF graph G. Let us also consider $V(t)$ as the set of variables that appear in triple pattern t, and $\mu(t)$ the image of t under a mapping μ. We now describe $Q(G)$ recursively:

$t(G) := \{\mu \mid \mu(t) \in G,\ dom(\mu) = V(t)\}$

$[Q_1 \text{ AND } Q_2](G) := Q_1(G) \bowtie Q_2(G)$

$[Q_1 \text{ UNION } Q_2](G) := Q_1(G) \cup Q_2(G)$

$[Q_1 \text{ OPT } Q_2](G) := Q_1(G) \text{⟕} Q_2(G)$

$\text{FILTER}_R(Q)(G) := \sigma_R(Q(G))$

$\text{SELECT}_V(Q)(G) := \pi_V(Q(G))$

2.3.3 Set or Bag Semantics

The above definitions apply assuming a *set semantics* for query answering. In other words, no duplicate mappings are returned as solutions.

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

When considering bag semantics, equivalence can be reduced to isomorphism. We need apply only a canonical labelling of the variables in each SPARQL query.

On the other hand, if we consider set semantics, then it is imperative to compute the core of each query, before ascertaining isomorphism, in order to determine if both queries are equivalent.

Therefore, this work considers both bag and set semantics.

If a query omits a projection, we assume that all variables are projected, with set semantics. As of SPARQL 1.0, projection is limited only outside all other query operators. SPARQL 1.1 allows the projection of sub-queries, but this is not considered for the scope of this work.

To consider queries with set or bag semantics, we extend our notion of a query as follows:

SELECT_V^{Δ}, where $\Delta = \begin{cases} true \text{ for set semantics} \\ false \text{ for bag semantics} \end{cases}$

In the SPARQL standard, the DISTINCT keyword can be used to select set semantics.

2.3.4 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:

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.

```
SELECT *
WHERE{
?actor ex:acts_in ?movie; ex:salary ?salary .
}
ORDER BY ?salary
```

Figure 2.7: Example of ORDER BY clause

Figure 2.7 illustrates a query pattern that contains an ORDER BY clause. In this example, the results are ordered by the values mapped to `?salary` in ascending order.

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 n^{th} result. If both LIMIT and OFFSET are used, with m and n respectively, the query will return the first m results starting from the n^{th} result.

Figure 2.8 illustrates a query pattern that contains an ORDER BY clause, a LIMIT clause and an OFFSET clause. In this example, we perform the same query as in Figure 2.7, but we limit the results to 50 starting from the 10th value (i.e. we obtain results 10 to 60 in ascending order).

```
SELECT *
WHERE{
?actor ex:acts_in ?movie; ex:salary ?salary .
}
LIMIT 50
OFFSET 10
ORDER BY ?salary
```

Figure 2.8: Example of LIMIT and OFFSET

Default and Named Graphs

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 graphs.

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

Figure 2.9: Example of FROM clause

Figure 2.9 illustrates a query pattern where we have defined the default graph as the graph named `http://example.org/a`. This query pattern will be matched with the constructed default graph and return the appropriate results.

If a set of named graphs has been defined, they can be accessed with the GRAPH clause. A GRAPH clause is followed by an IRI or a variable that maps to each of the named graphs. A GRAPH clause is also followed by a sub-query which is executed over each named graph.

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

Figure 2.10: Example of GRAPH clause

Figure 2.10 contains a query pattern where we have defined the default graph, as well as a set of named graphs. The body of the query contains a GRAPH operator with a variable

`?g` that is mapped to each of the named graphs defined in the preamble. The query inside the GRAPH operator will be executed on each of the named graphs.

2.3.5 Other Query Types

We have focused on SELECT queries, which returns tables as results. SPARQL 1.0 also supports the following query types:

CONSTRUCT This type of query returns a new RDF graph based on the results of a query pattern.

ASK This type of query returns a Boolean value indicating whether the query pattern has a solution or not.

DESCRIBE This type of query returns a single RDF graph that contains RDF data about resources. This data is not prescribed by a SPARQL query, but is determined by the SPARQL query processor instead.

However, the focus of the current work is on canonicalisation of SPARQL 1.0 SELECT queries.

2.3.6 Query Equivalence and Containment

For the next definitions, we will use the notation $Q(G)$ to denote the result of querying Q over the RDF graph G. We denote $\mathbb{G}$ as the set that contains all RDF graphs, and V as the set of variables.

Definition 2.3.3 *Let Q_1 and Q_2 be two queries. Q_1 is contained in Q_2 if and only if $\forall G \in \mathbb{G}, Q_1(G) \subseteq Q_2(G)$. This is written $Q_1 \sqsubseteq Q_2$.*

Definition 2.3.4 *Two queries Q_1 and Q_2 are equivalent, written $Q_1 \equiv Q_2$, if and only if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.*

What the above definition means in essence is that two queries are equivalent if they return the exact same results over any dataset.

Additionally, if $Q_1 \sqsubseteq Q_2$ and $Q_1 \not\equiv Q_2$, we write $Q_1 \sqsubset Q_2$.

Two queries might not be equivalent unless the variable names are rewritten. If we look back to Figures 1.1 and 1.2, we can appreciate that if we were to rewrite the variables in the query contained in Figure 1.2 as follows, we would have equivalent queries: `?movie` is mapped to `?film`; `?salary` is mapped to `?wage`.

Definition 2.3.5 *Let $\mu : \mathbf{V} \to \mathbf{V}$ be a bijective variable mapping, and let Q be a query. We will denote by $\mu(Q)$ the result of replacing each variable v in Q with $\mu(v)$.*

Definition 2.3.6 *Two queries Q_1 and Q_2 are congruent if and only if there exists a bijective variable mapping $\mu : \mathbf{V} \to \mathbf{V}$ such that $\mu(Q_1) \equiv Q_2$. This will be written $Q_1 \cong Q_2$.*

To maximise the number of redundant queries detected by our methods, we consider canonicalisation of congruent queries, disregarding the particular naming of variables.

SPARQL containment is a problem whose complexity ranges from NP-complete for queries that contain only conjunctions to undecidability for the full language [15]. Current SPARQL Containment and Equivalence tools are limited to deciding such relations for pairs of SPARQL queries. We have already discussed the limits of said techniques in Chapter 1, where we wish to avoid quadratic checks. Our proposal is thus to canonicalise SPARQL queries with respect to congruence. This method has a doubly-exponential worst-case complexity producing a potentially exponentially-sized query. Furthermore, it is only guaranteed to return the same canonical query for congruent queries using AND, UNION and SELECT. However, with respect to the complexity, most real-world queries are relatively simple, where our hypothesis is that worst-case instances will rarely be encountered in practice. With respect to other features of SPARQL for which we cannot guarantee complete canonicalisation, we will resort to a sound, best-effort approach. Though this may mean we miss congruences between queries, we do not see this as a problem for use-cases such as caching, where our goal is to increase the cache hit-rate over current methods using syntactic normalisation. Our proposed canonicalisation method is discussed in the following chapter.

Chapter 3

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 congruent queries.

The proposed algorithm must 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.

3.1 Overview

In this section, we will provide a brief overview of each step of the canonicalisation process:

Our proposed solution can be summarised in three steps:

1. Represent the SPARQL query as an RDF graph
2. Apply canonicalisation of the RDF graph
   (a) Transform the query into a union of conjunctive queries (UCQ)
   (b) Perform leaning of each conjunctive query
   (c) Remove redundant conjunctive queries
   (d) Canonically label the RDF graph
3. Convert the canonicalised graph back into a deterministically ordered SPARQL query.

3.2 Supported Queries

As we have already explained, the equivalence problem is, at the very least, a "hard" problem. This problem may vary from being NP-complete when the queries contain only conjunctions, up to undecidable if projection and left outer joins are allowed.

As a result, we have limited the scope of our canonicalisation algorithm.

The queries we support fully are traditionally known as *monotone* queries [16]. These queries contain the natural join and union features, as well as projections and the DISTINCT keyword. In cases that contain other features, we are able to detect sub-queries that only contain joins and unions, and perform canonicalisation over those sub-queries.

We guarantee soundness in our normalisation process for all features in SPARQL 1.0. This is because we do not apply any kind of minimisation over query patterns that contain features other than conjunctions or disjunctions; we only apply a canonical labelling, which cannot change the semantics of the query (modulo variable names). On the other hand the proposed algorithm is sound and complete for monotone queries. Support for SPARQL 1.1 is left for future work.

3.3 r-Graph for Monotone Queries

We first describe how we will represent a monotone SPARQL query as an RDF graph. We will denote this as an r-graph.

Definition 3.3.1 *Let $\beta()$ denote a function that returns a fresh blank node and $\beta(x)$ a function that returns a blank node unique to x. Let $\iota()$ denote an id function such that if $x \in \mathbf{IL}$, then $\iota(x) = x$; otherwise if $x \in \mathbf{VB}$, then $\iota(x) = \beta(x)$. We define $R(Q)$, the r-graph of a query Q, as follows:*

- *If Q is a triple pattern (s, p, o), then $\iota(Q) = \beta()$ and*
  $R(Q) = \{(\iota(Q), \texttt{ex:sub}, \iota(s)), (\iota(Q), \texttt{ex:pred}, \iota(p)), (\iota(Q), \texttt{ex:obj}, \iota(o)),$
  $(\iota(Q), \texttt{ex:type}, \texttt{ex:TP})\}$

- *If Q is $[Q_1 \text{ AND } Q_2]$, then $\iota(Q) = \beta()$ and*
  $R(Q) = \{(\iota(Q), \texttt{ex:arg}, \iota(Q_1)), (\iota(Q), \texttt{ex:arg}, \iota(Q_2)), (\iota(Q), \texttt{ex:type}, \texttt{ex:join})\}$

- *If Q is $[Q_1 \text{ UNION } Q_2]$, then $\iota(Q) = \beta()$ and*
  $R(Q) = \{(\iota(Q), \texttt{ex:arg}, \iota(Q_1)), (\iota(Q), \texttt{ex:arg}, \iota(Q_2)), (\iota(Q), \texttt{ex:type}, \texttt{ex:union})\}$

- *If Q is $\text{SELECT}_V^{\Delta}(Q_1)$, then $\iota(Q) = \beta()$ and*
  $R(Q) = \{(\iota(Q), \texttt{ex:OP}, \iota(Q_1)), (\iota(Q), \texttt{ex:distinct}, \Delta), (\iota(Q), \texttt{ex:type}, \texttt{ex:projection})\}$
  $\cup\{\{(\iota(Q), \texttt{ex:arg}, \iota(v)), (\iota(v), \texttt{ex:type}, \texttt{ex:var})\} \mid v \in V\}$

where Δ is a boolean datatype literal.

We will now present illustrated examples of the representations for each of the features contained in monotone queries.

3.3.1 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.

IRIs and literal terms are assigned nodes whose values correspond to its IRI or literal value respectively.

In all of the following examples, nodes with a dashed outline denote other query graphs. We also use an example prefix `ex:` whose value is not important in this study.

3.3.2 Triple Patterns

SPARQL allows triple patterns $t \in \mathbf{IBVL} \times \mathbf{IVL} \times \mathbf{IBVL}$. Our representation is a form of reification, except that it is limited to single triples and uses a different typing system.

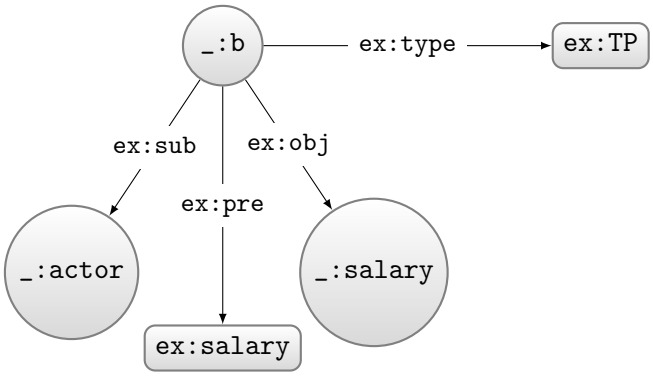

Figure 3.1: Representation of a triple pattern

Figure 3.1 illustrates our representation of a triple from Figure 1.2. There are two blank nodes that correspond to the variables in the triple –denoted by circles–, and a blank node `_:b` which 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 (`ex:TP`).

3.3.3 Joins and Unions

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

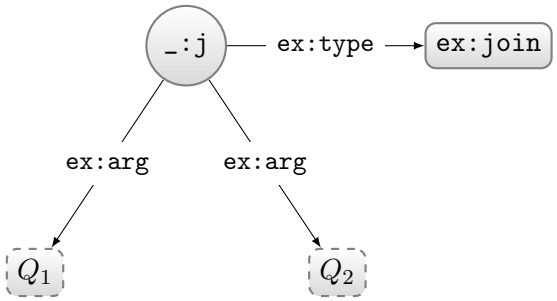

Figure 3.2: Representation of a JOIN operator

Figure 3.2 illustrates the representation of an expression $Q_1 \bowtie Q_2$ where Q_1 and Q_2 are sub-queries, which are denoted by dashed rectangles. Note that Q_1 and Q_2 may be other operators or a triple pattern as seen in Figure 3.1. An advantage of this structure is that because graph edges are unordered, we have already captured the commutative property of join.

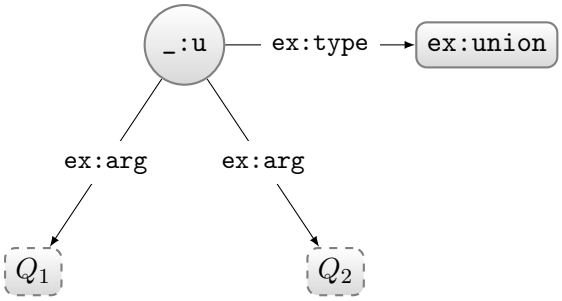

Figure 3.3: Representation of a UNION operator

Figure 3.3 illustrates the representation of an expression $Q_1 \cup Q_2$, which is analogous to the join operator presented in Figure 3.2. The one key difference is the type node that denotes that this structure is of type `ex:union`.

3.3.4 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 connected to: all projected variables with an `ex:arg` predicate; a boolean literal node with an `ex:distinct` predicate

denoting whether it contains the DISTINCT keyword; and an `ex:OP` predicate pointing to the first operation of the r-graph denoting the body of the query.

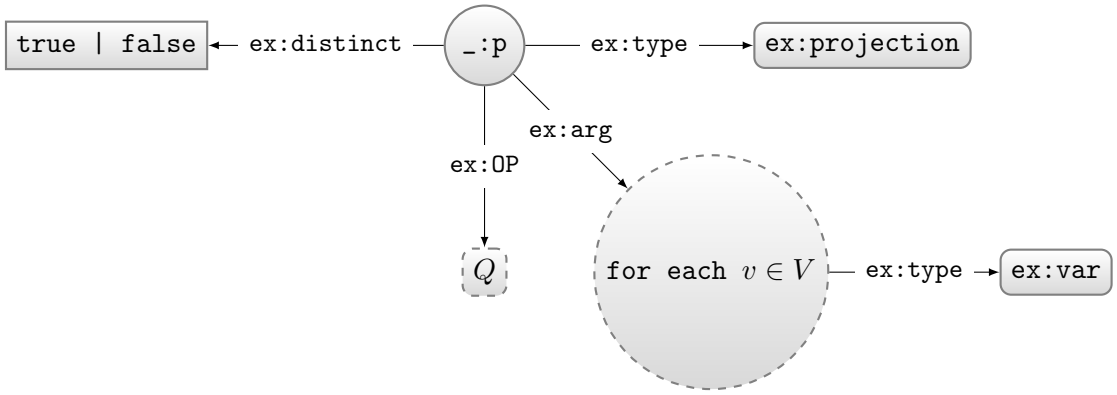

Figure 3.4: Representation of a projection

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.

With this, we have defined the representations of the monotone SPARQL operators for which we have implemented a sound and complete canonicalisation algorithm.

3.4 UCQ Normal Form

We first represent our monotone queries in disjunctive normal form (DNF). This will result in unions of conjunctive queries that later enable us to perform standard normalisation methods, which are well-known in database theory literature.

To achieve this union of conjunctive queries (UCQ) normal form, we must consider the following type of rewriting:

$$(A \cup B) \bowtie (C \cup D) = (A \bowtie C) \cup (A \bowtie D) \cup (B \bowtie C) \cup (B \bowtie D)$$

Using this property, we can rewrite any monotone query into a UCQ in DNF.

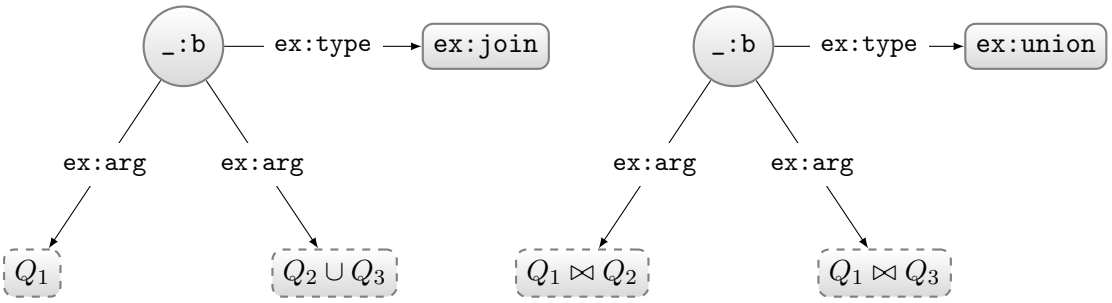

Figure 3.5: Distributive property of JOIN and UNION

Figure 3.5 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 a congruent query in the form we wish to represent: a disjunction of conjunctive queries (i.e. a union of joins).

3.4.1 Formalisation of UCQs

We will now define the syntax of UCQ queries formally.

Definition 3.4.1 *The syntax of a UCQ query is as follows:*

1. *If* $\{t_1, \ldots, t_n\}$ *is a set of triple patterns* ($n \geq 1$)*, then* $\mathsf{and}(\{t_1, \ldots, t_n\})$ *is a query pattern called a conjunctive query (CQ) pattern.*

2. *If* $\{C_1, \ldots, C_n\}$ *is a bag of CQ patterns* ($n \geq 1$)*, then* $\mathsf{union}(\{C_1, \ldots, C_n\})$ *is a (UCQ) query pattern.*

3. *If* Q *is a UCQ query pattern and* V *is a set of variables such that for all* $v \in V$*,* v *appears in some pattern contained in* Q*, then* $\mathsf{select}^{\Delta}_{V}(Q)$ *is a query.*

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

Definition 3.4.2 *Letting* Q *be a query in UCQ syntax and* G *an RDF graphs, then the semantics of a UCQ query is defined as follows:*

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

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

We have already mentioned that as part of the normalisation process, we must transform the monotone queries (MQs) into a UCQ normal form. It is a known result that for every monotone query Q, there exists a query Q' in a UCQ form such that $Q \equiv Q'$ [16].

Definition 3.4.3 *For an MQ* Q*, we define the UCQ rewriting of* Q*, denoted* $\cup(Q)$*, as follows:*

1. *If* Q *is a triple pattern* t*, then* $\cup(Q) = \mathsf{and}(\{t\})$.

2. *If* Q *is* $[Q_1 \text{ AND } Q_2]$, *then* $\text{U}(Q) = \text{U}^*([\text{U}(Q_1) \text{ AND } \text{U}(Q_2)])$.

3. *If* Q *is* $[Q_1 \text{ UNION } Q_2]$, *then* $\text{U}(Q) = \text{U}^*([\text{U}(Q_1) \text{ UNION } \text{U}(Q_2)])$.

4. *If* Q *is* $\text{SELECT}_V^{\Delta}(Q')$, *then* $\text{U}(Q) = \mathsf{select}_V^{\Delta}(\text{U}(Q'))$

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

5. *If* Q *is* $[\mathsf{and}(T_1) \text{ AND } \mathsf{and}(T_2)]$ *then* $\text{U}^*(Q) = \mathsf{and}(T_1 \cup T_2)$.

6. *If* Q *is* $[\mathsf{and}(T) \text{ AND } \mathsf{union}(\mathbb{C})]$ *or* $[\mathsf{union}(\mathbb{C}) \text{ AND } \mathsf{and}(T)]$
then $\text{U}^*(Q) = \mathsf{union}(\{\mathsf{and}(T \cup T') \mid \mathsf{and}(T') \in \mathbb{C}\})$.

7. *If* Q *is* $[\mathsf{union}(\mathbb{C}_1) \text{ AND } \mathsf{union}(\mathbb{C}_2)]$
then $\text{U}^*(Q) = \mathsf{union}(\{\mathsf{and}(T_1 \cup T_2) \mid \mathsf{and}(T_1) \in \mathbb{C}_1 \wedge \mathsf{and}(T_2) \in \mathbb{C}_2\})$.

8. *If* Q *is* $[\mathsf{union}(\mathbb{C}_1) \text{ UNION } \mathsf{union}(\mathbb{C}_2)]$ *then* $\text{U}^*(Q) = \mathsf{union}(\mathbb{C}_1 \cup \mathbb{C}_2)$.

9. *If* Q *is* $[\mathsf{and}(T) \text{ UNION } \mathsf{union}(\mathbb{C})]$ *or* $[\mathsf{union}(\mathbb{C}) \text{ UNION } \mathsf{and}(T)]$
then $\text{U}^*(Q) = \mathsf{union}(\{\mathsf{and}(T)\} \cup \mathbb{C})$.

10. *If* Q *is* $[\mathsf{and}(T_1) \text{ UNION } \mathsf{and}(T_2)]$ *then* $\text{U}^*(Q) = \mathsf{union}(\{\mathsf{and}(T_1), \mathsf{and}(T_2)\})$.

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

These rules rewrite monotone SPARQL queries into a UCQ. For example, Rule 6 performs the rewriting of a JOIN of a UNION into a UNION of JOINs as illustrated in Figure 3.5.

We will present an example of a SPARQL query to better illustrate this idea:

```
SELECT *
WHERE{
  { {?a ex:p ?b } UNION {?a ex:q ?b} } .
  { {?c ex:p ?d } UNION {?c ex:q ?d} } .
}
```

Figure 3.6: Example of a SPARQL query

Figure 3.6 provides an example of a monotone query not in UCQ formal form. On the other hand, Figure 3.7 provides the same query after UCQ normalisation.

In practice, we apply the UCQ rewriting over the r-graph of a monotone query Q. In order to better illustrate the steps of this process, in Figure 3.8 we present the r-graph corresponding to the query presented in Figure 3.9.

```
SELECT *
WHERE{
    { ?a ex:p ?b . ?c ex:p ?d }
    UNION { ?a ex:q ?b . ?c ex:p ?d . }
    UNION { ?a ex:q ?b . ?c ex:q ?d . }
    UNION { ?a ex:p ?b . ?c ex:q ?d . }
}
```

Figure 3.7: Example of a SPARQL query in UCQ form

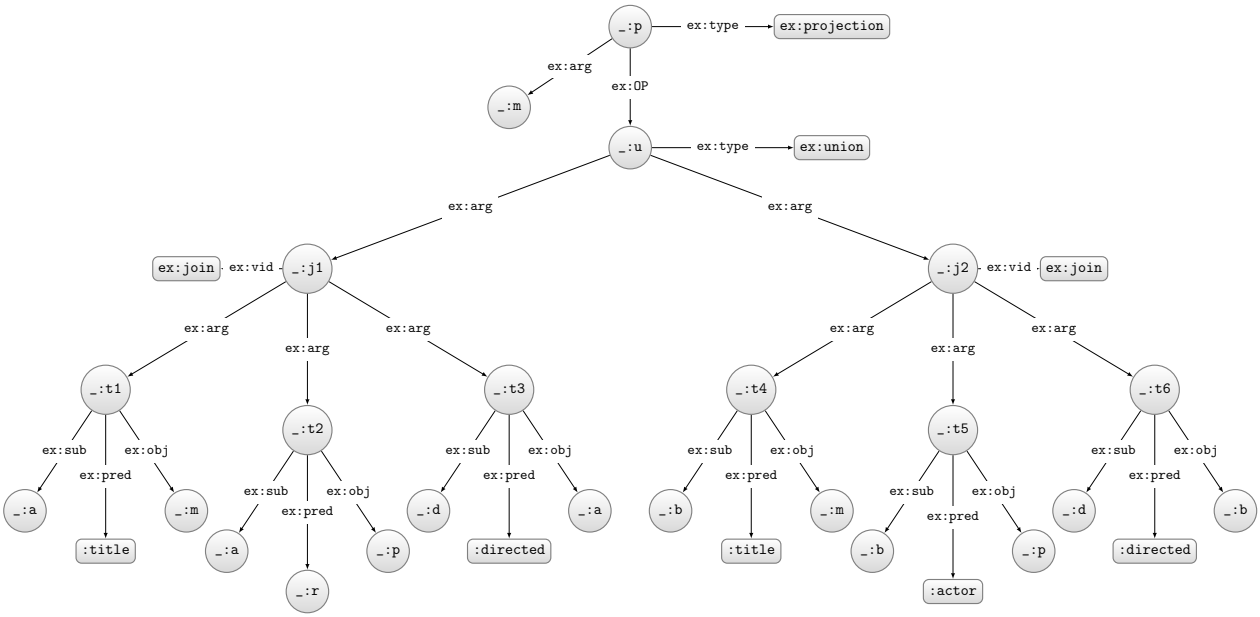

Figure 3.8: Example of a UCQ

The graph presented in Figure 3.8 contains a blank node that represents a union operator joined by an `ex:arg` predicate to its two operands. These operands are join operator nodes, both of which are connected to their respective triple patterns as described in section 3.3. Note that for readability, certain nodes are presented multiples time (for example `_:a`).

Given a monotone query Q, then $|UCQ(Q)|$ is exponential with respect to $|Q|$ in the worst case scenario, where we denote by $|Q|$ and $|UCQ(Q)|$ the number of non-unique triple patterns in query Q and $UCQ(Q)$ respectively.

3.4.2 Projection with Union

According to the standard SPARQL definitions, unions of query patterns with differing sets of variables are allowed, which is unlike the relational algebra and SQL. This distinctive feature of SPARQL proves to be a nuance when combined with projections, as it leads to more equivalence cases that need to be carefully considered.

```
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 3.9: Example of equivalent UCQs

The queries in Figure 3.9 are congruent (and equivalent) according to SPARQL semantics 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.

The solution we have devised is to distinguish the blank nodes in each conjunctive query or *branch* of the UCQ by labelling each non-projected variable depending on the branch (CQ) they belong to. Note that since the aforementioned rules have already been applied iteratively, the graph will contain at most a single union operator with each argument being a simple join; hence each such *branch* is a sub-graph of the `ex:arg` predicate of the union node.

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

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

Definition 3.4.4 *Let G denote $\text{R}(Q)$ for a UCQ Q. We define the variable distinguishing function $\text{D}(G)$ as follows. If there does not exist a blank node x such that $(x, \texttt{ex:type}, \texttt{ex:union}) \in G$, then $\text{D}(G) = G$. Otherwise if such a blank node exists, we define $\text{D}(G) = \{(s,p,\delta(o)) \mid (s,p,o) \in G\}$, where $\delta(o) = o$ if $o \notin \mathsf{uvar}(G)$; otherwise $\delta(o) = \beta(r,o)$ such that $r \in \mathsf{cq}(G)$ and $(s,p,o) \in G[r]$.* □

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

Essentially, this creates a fresh blank node for each non-projected variable appearing in

each representation of a CQ in G.

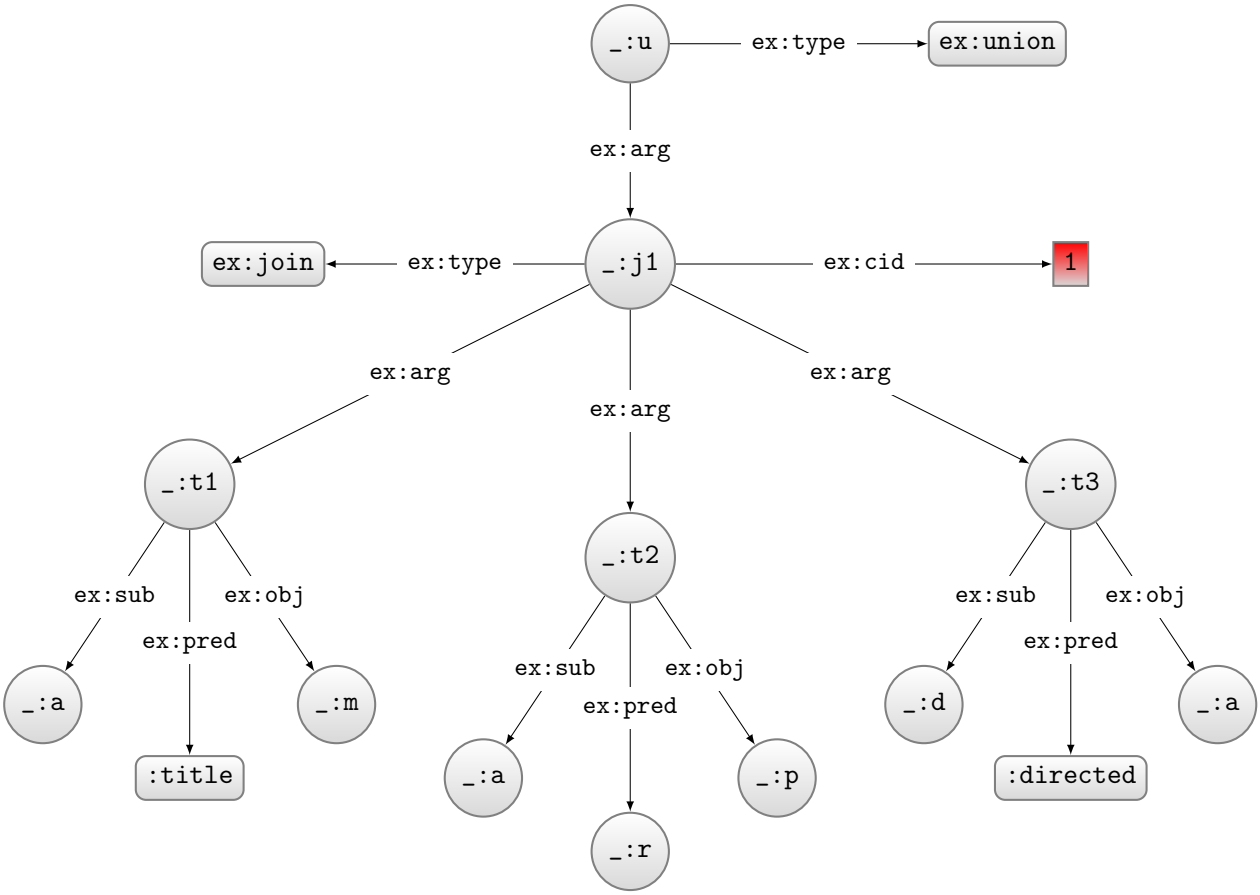

Figure 3.10: Example of a CQ

Figure 3.10 illustrates the r-graph corresponding to a single branch (CQ). Note that each of the individual branches contains a `ex:union` node which will be used in a later step to merge all the branches into a single union node. Highlighted in red is a temporary identification label for each conjunction node.

Once the identification labels for each conjunction node have been determined, new blank nodes are created for every variable node which is not projected. This is because these variables act as existentials scoped to each CQ, where having the same such variable in different CQs would lead to false correspondences in the graph's structure. These blank nodes are labelled based on the identification label of its branch and of its original node.

In Figure 3.11 we can observe the same r-graph as the one in Figure 3.10, except for the newly labelled nodes (shown in yellow). Note that the node corresponding to `_:m` remains the same because it is projected.

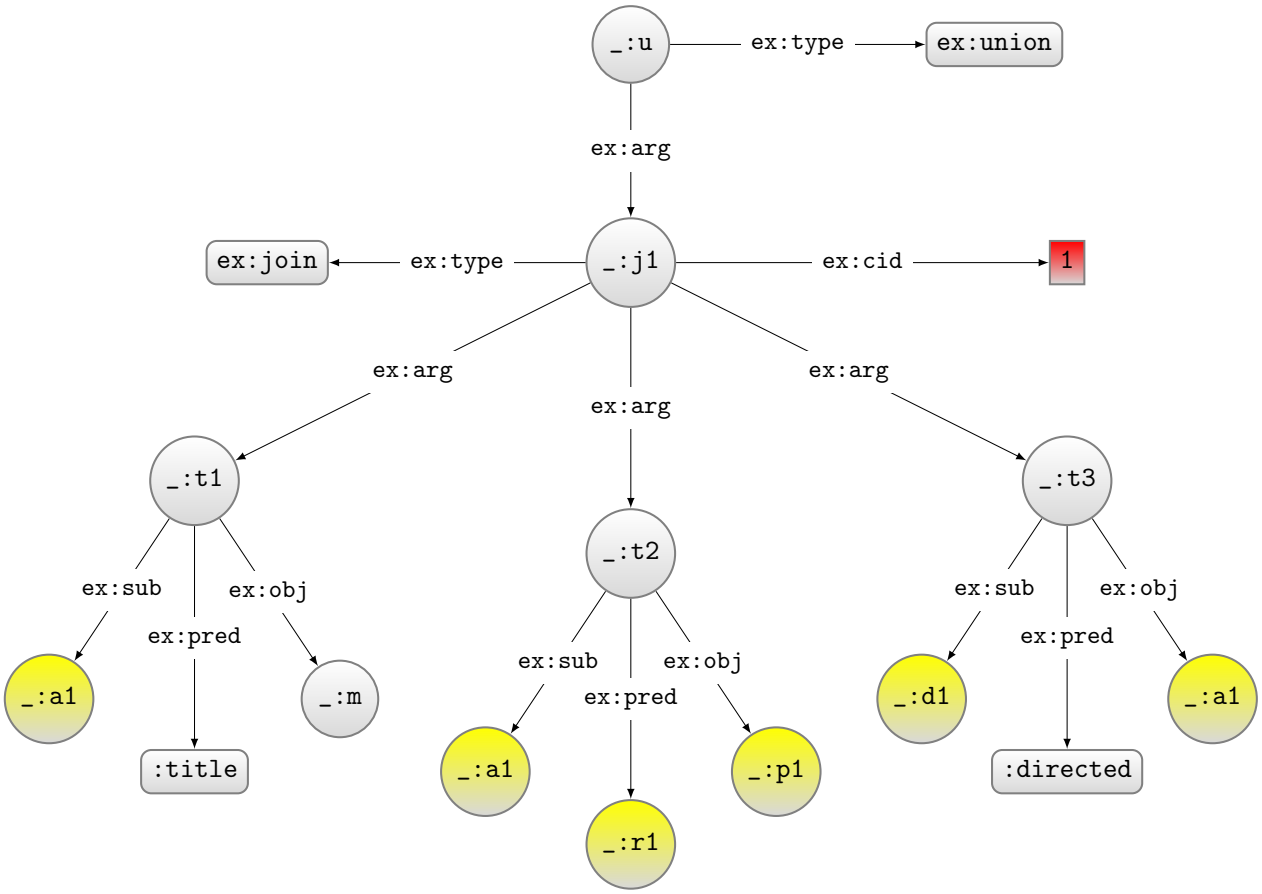

Figure 3.11: Example of a CQ with new labels

3.4.3 Redundancy-free UCQs

Where set semantics are concerned, equivalence and congruence between queries may be preserved if we remove certain triples. Evidently, for the purpose of designing a normal form, such triples must also be removed. Therefore we need to remove all of these *redundancies* from the r-graphs. This process of removing all of these redundancies from a query is known as *minimisation*. At this point, we have already produced a UCQ; therefore the query consists in a single union operator with either triple patterns or conjunctive queries as arguments.

Our next approach will consist of the following steps:

- Minimise each conjunctive query
- Minimise the union of all conjunctive queries

We will now remove the redundancy within conjunctive queries. Note that under bag semantics, no minimisation need be applied since any redundancy affects the multiplicity of solutions.

In the following description of the solution, we may use *branch* to refer to the sub-graph for each individual conjunctive query.

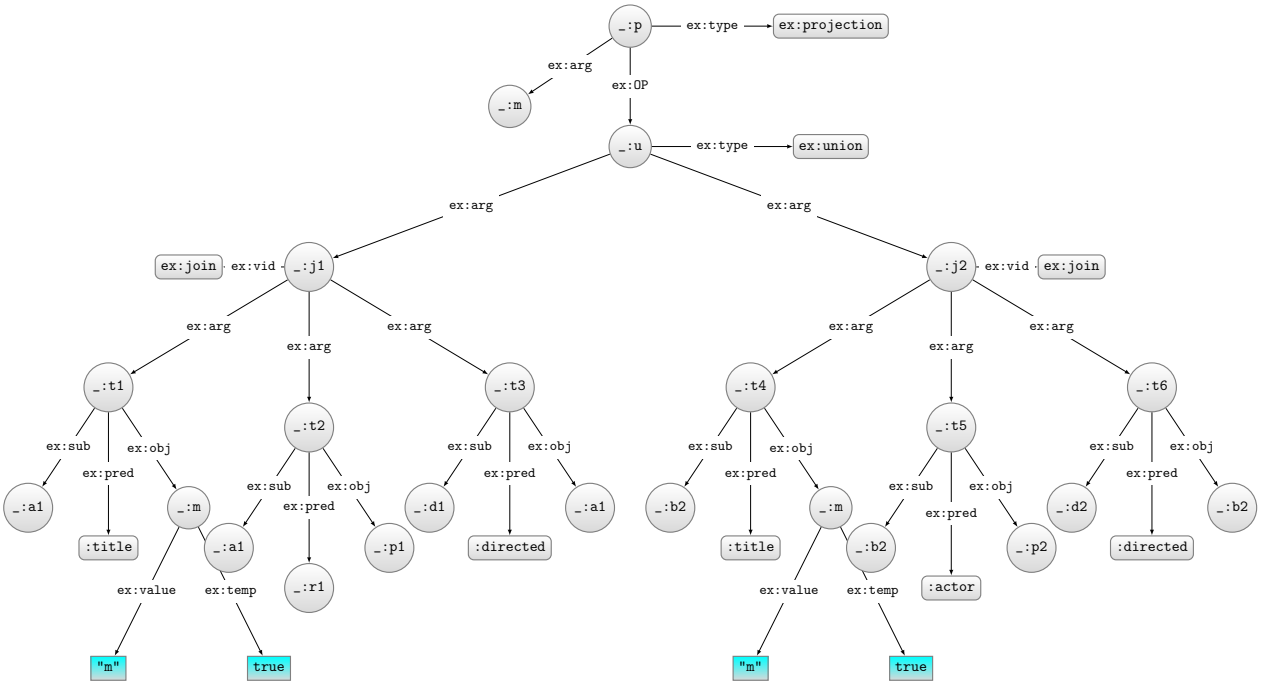

Figure 3.12: Example of a UCQ with temporary labels

We first assert that the query is of the form $Q = (Q_1 \cup \cdots \cup Q_n)$ if the first operator (aside from projection) is of type `ex:union`. The r-graph presented in Figure 3.12 is of the form that has been mentioned. Note that we keep projected variables distinguished with temporary, unique names to avoid their being removed during the following minimisation process; this can be seen for the variable `_:m` in Figure 3.12. We denote this labelling as $L(G)$.

Following this, 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 which are projected, solution modifiers; all of which are relevant to identify variables whose labels must be preserved throughout this process.

The inner graph is then divided into each of its branches that represents a CQ. For each branch, we will work with a graph corresponding to the union operator node with the individual branch as an argument. Following this, we compute the core of each branch, and store them in an array for the next step.

Definition 3.4.5 *Let G denote $\mathrm{D}(\mathrm{R}(Q))$ for a UCQ Q. We define the CQ-minimisation of G as $\mathrm{C}(G) = \{\mathsf{core}(L(G[x])) \mid x \in \mathsf{cq}(G)\}$. We call $C \in \mathrm{C}(G)$ a CQ core.*

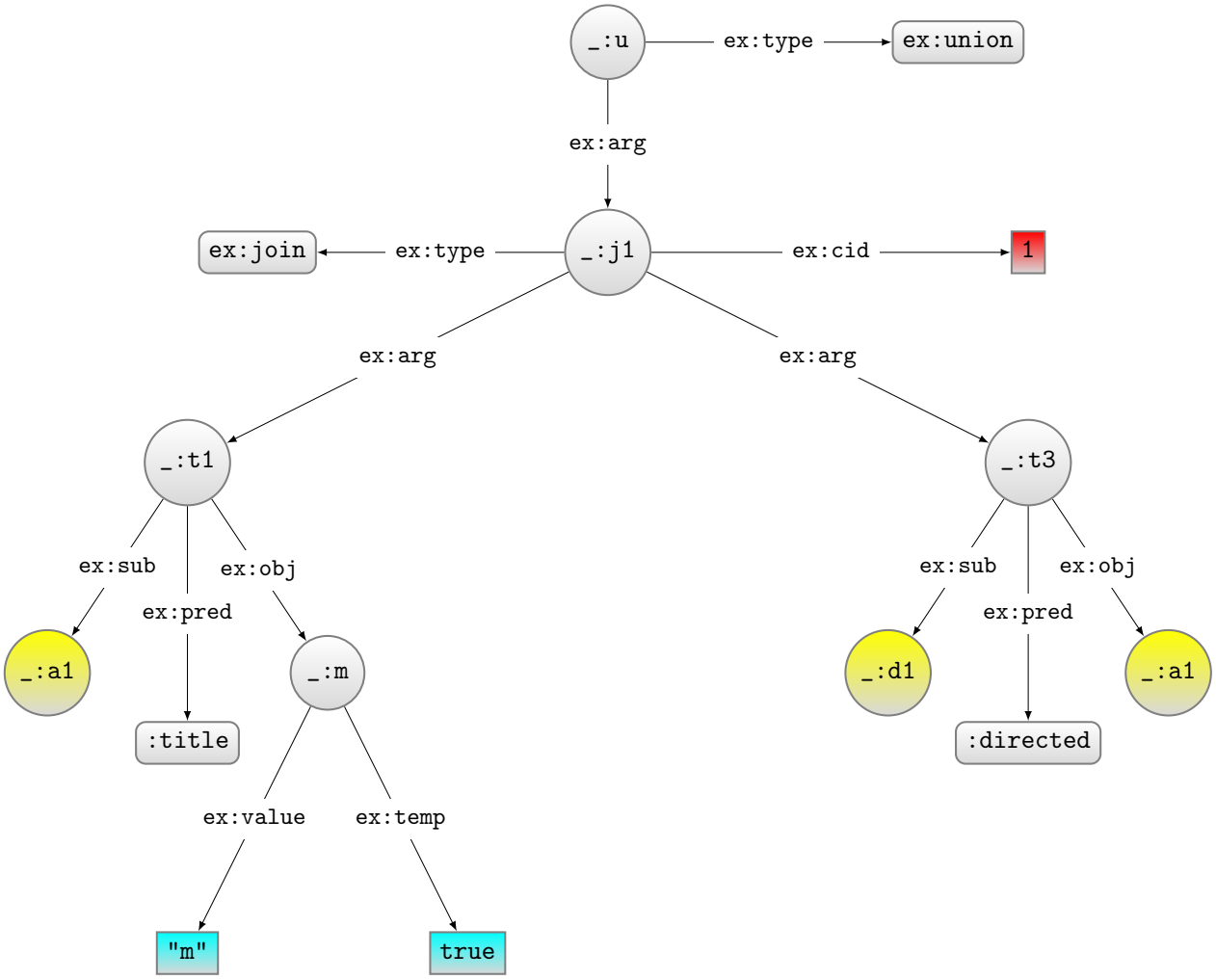

Figure 3.13: Example of the core of a CQ

Figure 3.13 illustrates the core of the CQ presented in Figure 3.11 with the projected variable `_:m` now labelled. The triple `(_:a1, _:r1, _:p1)` has been removed from the CQ because it can be mapped to triple `(_:a1, :title, _:m)`, which indicates that the first triple is redundant.

There is another kind of redundancy that must be considered: redundancy of different conjunctive queries. Let us consider the queries in Figure **??**. Evidently, these queries are congruent. The same *cannot* be said about the queries in Figure 3.15. However, both queries would have the same canonical form if the same process of redundancy reduction is applied to both r-graphs. This would change the semantics of the first query in Figure 3.15. In the case of union, the more specific query pattern is the redundant one, therefore we should drop that query pattern to avoid redundancy. In Figure 3.15, in the query on the left the more specific query pattern is $(?a, :\texttt{title}, ?m)$, but in fact this is the query pattern that should be removed.

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

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

Figure 3.14: Example of two congruent queries, if we remove redundancy.

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

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

Figure 3.15: Example of two queries which are *not* congruent

As such, we will verify if there exists a case where one of the branches of the UCQ entails another. Let $(Q, V) := (Q_1 \cup \cdots \cup Q_n, V)$ denote the UCQ under analysis, where V denotes the projected variables. The following must be removed from Q:

- $Q_i (1 \leq i \leq n)$ such that there exists $Q_j (1 \leq j < i \leq n)$ such that $\mathsf{select}^{\Delta}_{V}(Q_i) \equiv \mathsf{select}^{\Delta}_{V}(Q_j)$
- $Q_i (1 \leq i \leq n)$ where there exists $Q_j (1 \leq i \leq n)$ such that $\mathsf{select}^{\Delta}_{V}(Q_i) \sqsubset \mathsf{select}^{\Delta}_{V}(Q_j)$ (a proper containment; see Section 2.3.6).

The first condition removes all queries apart from one for each set of equivalent CQs in the UCQ. This ensures that no information is lost in this process. The second condition removes all CQs which are properly contained in another. More formally, we define this process as follows.

Definition 3.4.6 *Let* $\mathrm{C}(G) = \{C_1, \ldots, C_n\}$ *denote the CQ cores of* $G = \mathrm{D}(\mathrm{R}(Q))$ *for a UCQ* Q*. A CQ core* C_i *is in* $\mathrm{E}(G)$ *iff* $C_i \in \mathrm{C}(G)$ *and there does not exist a CQ core* $C_j \in \mathrm{C}(G)$ *(*$i \neq j$*) such that:* $\mathsf{pvar}(C_i) = \mathsf{pvar}(C_j)$*; and* $C_j \models C_i$*, or* $C_i \cong C_j$ *for* $j < i$*.* □

Definition 3.4.7 *Let* $\mathrm{E}(G) = \{C_1, \ldots, C_n\}$ *denote the minimal CQ cores of* $G = \mathrm{D}(\mathrm{R}(Q))$ *for a UCQ* Q*. Let* $P = \{(s, p, o) \in G \mid \exists(s, \texttt{ex:type}, \texttt{ex:projection}) \in G\}$ *and* $U = \{(s, p, o) \in G \mid \exists(s, \texttt{ex:type}, \texttt{ex:union}) \in G$*,* and $p = \texttt{ex:arg}$ implies $\exists C \in \mathrm{E}(G) : \{o\} = \mathsf{cq}(C)\}$*. We*

define the minimisation of G as $\textsc{m}(G) = \bigcup_{G' \in \textsc{e}(G)} L^{-}(G') \cup P \cup U$*, where* $L^{-}(G')$ *denotes removing the temporary labels on the projected variables in* G'*.* □

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

Once we have all the unique conjunctive queries, we can unite them under a single union operator node.

3.4.4 Canonical Variables

To finish the canonicalisation of the r-graph, we apply canonical labelling of the blank nodes, which will give the canonical labels of the final query variables. For this, we use the algorithm for $\textsc{ican}(G)$ described in Section 2.2.3.

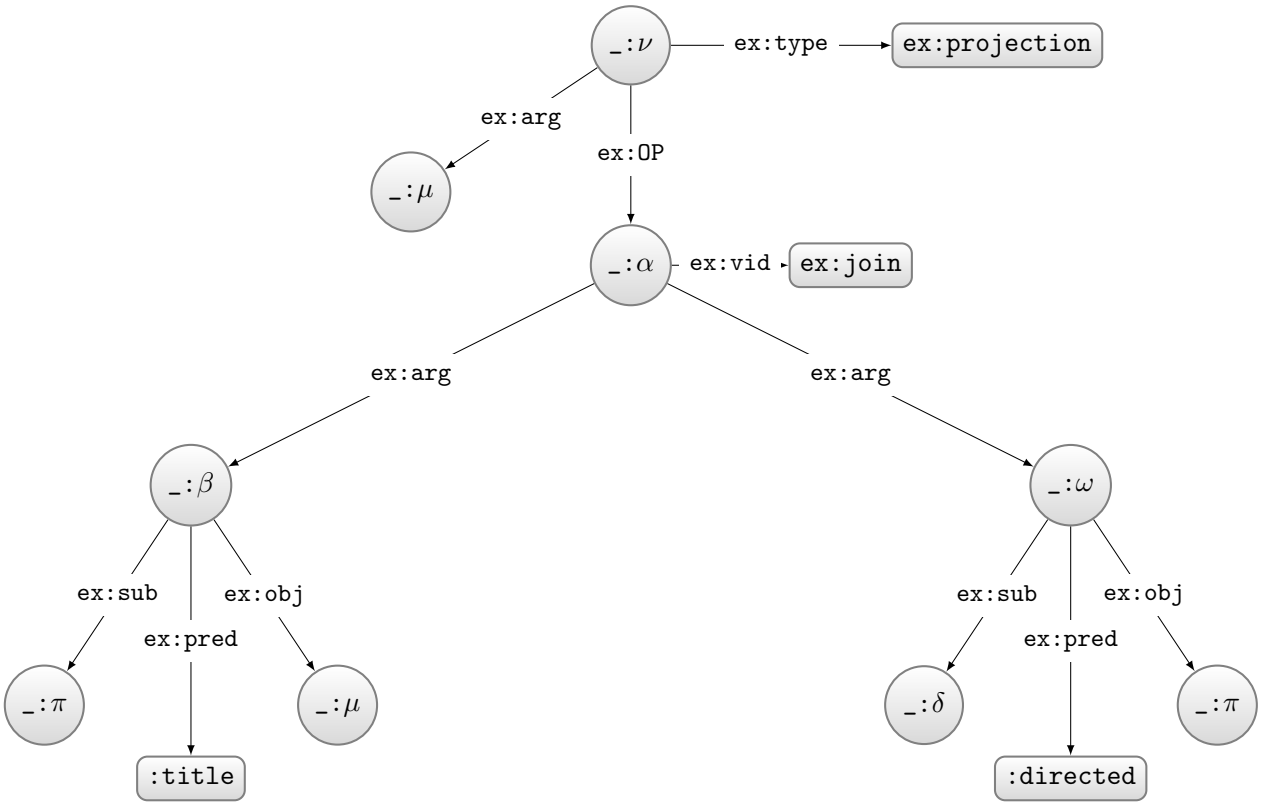

Figure 3.16: Resulting r-graph

Figure 3.16 illustrates the resulting r-graph for the running example (see Figure 3.12) 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 `ex:union` node has been removed, as it is also redundant.

There are no more transformations to be done over the r-graphs at this point. The r-graphs are identical for queries that contain supported features.

3.4.5 Canonical Query

We can now obtain a canonical query by applying the inverse of the operations used to construct the r-graphs and syntactically ordering triple patterns in each CQ, and the CQs in the UCQ. We will now formalise the process through which we retrieve the query represented by the r-graph. We define the inverse mapping of an r-graph as follows:

Definition 3.4.8 *Let G denote the r-graph $\mathrm{R}(Q)$ for a UCQ Q. Let $\top_G \in \mathbf{IBL}$ be the root of G such that there does not exist (s, p) such that $(s, p, \top_G) \in G$. Further let $G[z]$ denote the sub-graph of G rooted at z. Finally, let $\nu : \mathbf{ILB} \to \mathbf{ILV}$ be a function that is the identity on $\mathbf{IL}$ (i.e., for all $x \in \mathbf{IL}$, $\nu(x) = x$) and that maps $\mathbf{B}$ to $\mathbf{V}$ in a deterministic one-to-one manner. We then define the inverse r-graph operation $\mathrm{R}^-(G)$ to a UCQ query as follows:*

- *If there exists (s, p, o) such that*
  $\{(z, \texttt{ex:type}, \texttt{ex:TP}), (z, \texttt{ex:sub}, s), (z, \texttt{ex:pred}, p), (z, \texttt{ex:obj}, o)\} \subseteq G$ *where*
  $z = \top_G$, *then* $\mathrm{R}^-(G) = (\nu(s), \nu(p), \nu(o))$.

- *If there exists $\{x_1, \ldots, x_n\}$ such that*
  $\{(z, \texttt{ex:type}, \texttt{ex:join}), (z, \texttt{ex:arg}, x_1), \ldots, (z, \texttt{ex:arg}, x_n)\} \subseteq G$ *where* $z = \top_G$
  and there does not exist $(z, \texttt{ex:arg}, x) \in G$ *such that* $x \notin \{x_1, \ldots, x_n\}$
  then $\mathrm{R}^-(G) = \mathsf{and}(\{\mathrm{R}^-(G[x_1]), \ldots, \mathrm{R}^-(G[x_n])\})$.

- *If there exists $\{x_1, \ldots, x_n\}$ such that*
  $\{(z, \texttt{ex:type}, \texttt{ex:union}), (z, \texttt{ex:arg}, x_1), \ldots, (z, \texttt{ex:arg}, x_n)\} \subseteq G$ *where* $z = \top_G$
  and there does not exist $(z, \texttt{ex:arg}, x) \in G$ *such that* $x \notin \{x_1, \ldots, x_n\}$
  then $\mathrm{R}^-(G) = \mathsf{union}(\{\mathrm{R}^-(G[x_1]), \ldots, \mathrm{R}^-(G[x_n])\})$.

- *If there exists $(x, \Delta, \{v_1, \ldots, v_n\})$ (for $n \geq 1$) such that*
  $\{(z, \texttt{ex:type}, \texttt{ex:projection}), (z, \texttt{ex:OP}, x), (z, \texttt{ex:distinct}, \Delta),$
  $(z, \texttt{ex:arg}, v_1), \ldots, (z, \texttt{ex:arg}, v_n)\} \subseteq G$ *where* $z = \top_G$
  and there does not exist $(z, \texttt{ex:arg}, v) \in G$ *such that* $v \notin \{v_1, \ldots, v_n\}$
  then $\mathrm{R}^-(G) = \mathsf{select}^{\Delta}_{\{\nu(v_1), \ldots, \nu(v_n)\}}(\mathrm{R}^-(G[x]))$. □

The result of the previous process is a query represented in abstract UCQ syntax. Note that our goal is to provide a canonical query in SPARQL syntax.

In the UCQ syntax, there are various unordered components of the query, including the set of projected variables, sets of triple patterns in each CQ, and a bag of CQs in the UCQ. However, concrete SPARQL syntax requires a deterministic order to be imposed over such elements. First, we can order the variables in the projected set according to their canonical labels. Second, we can order triple patterns by lexicographic order (subject, then predicate, then object, assuming some total order on the set **VIL**). Third, we can order sets of triple patterns in UCQs using a typical set ordering, for example $S \leq T$ if and only if $S \subseteq T$ or the minimum element of $S \setminus T$ is less than the minimum element of $T \setminus S$. Now we can create a deterministic ordering of all elements of a UCQ, allowing us to create a canonical query in SPARQL syntax.

By applying the inverse transformation over the r-graph in Figure 3.16, and ordering its elements, the resulting query is shown in Figure 3.17.

```
SELECT ?μ
WHERE{
?π :title ?μ .
?δ :directed ?π .
}
```

Figure 3.17: Example of a canonical query for the queries in Figure 3.9

3.5 Computational Properties

In this section, we provide a formal statement of completeness: that for any two MQs, they are congruent if and only if their canonicalised queries are equal.

Theorem 3.5.1. *For two MQs Q_1 and Q_2, it holds that $Q_1 \cong Q_2$ if and only if* $\text{R}^{-}(\text{ICAN}(\text{M}(\text{D}(\text{R}(\text{U}(Q_1))))))) = \text{R}^{-}(\text{ICAN}(\text{M}(\text{D}(\text{R}(\text{U}(Q_2))))))$.

See [12] for proof of this Theorem.

With respect to computational complexity, given a monotone query Q as input, the algorithm $\text{R}^{-}(\text{ICAN}(\text{M}(\text{D}(\text{R}(\text{U}(Q_2))))))$ to produce the canonical version of Q is doubly-exponential. The first level of exponentiality is given by the UCQ rewriting, $\text{U}(Q)$. Thereafter, $\text{M}()$ and $\text{ICAN}()$ involve exponential algorithms that run on the exponentially large query $\text{U}(Q)$.

3.6 Other Features

We will now present other features for which we have provided a "best effort" canonicalisation. We have already mentioned in previous chapters that equivalence for SPARQL queries becomes undecidable with features such as OPTIONAL and FILTER. However, the features presented in this section are widely used in practice, and are fundamental for the expressive power of SPARQL.

For this reason, we represent these features in our r-graph. However, when the input query contains such features, we apply a limited canonicalisation that does not apply UCQ normal forms. Hence our method may find some equivalences for queries that contain such features, but may also fail to detect such equivalences. Furthermore, we currently only support features of SPARQL 1.0. Support for SPARQL 1.1 features such as property paths, sub-queries, etc [9] is left for future work.

3.6.1 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.

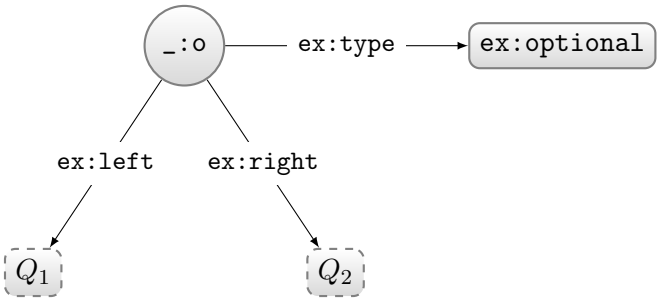

Figure 3.18: Representation of an OPTIONAL operator

Figure 3.18 illustrates our representation of a query pattern Q of the form $Q = Q_1 \bowtie Q_2$. 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 have already explained.

3.6.2 FILTER

Filter operators were much more complicated to represent than binary operators such as joins or unions. This is because they contain terms which are essentially functions.

For simplicity, we assume that there is a single filter operator for every BGP. If there are multiple filter operators, they can be reduced into a single filter operator by defining a new filter operator whose expression corresponds to the conjunction of all the expressions inside the other filter operators.

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

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

Figure 3.19: Equivalent FILTER expressions

Note that we create a fresh blank node for each value in the filter expressions –either variables, IRIs or literals– which is connected by an `ex:value` edge to a node which contains the actual value. This is necessary because certain operations such as inequalities are ordered, which means that we need to know the position of the arguments. Therefore, we add an

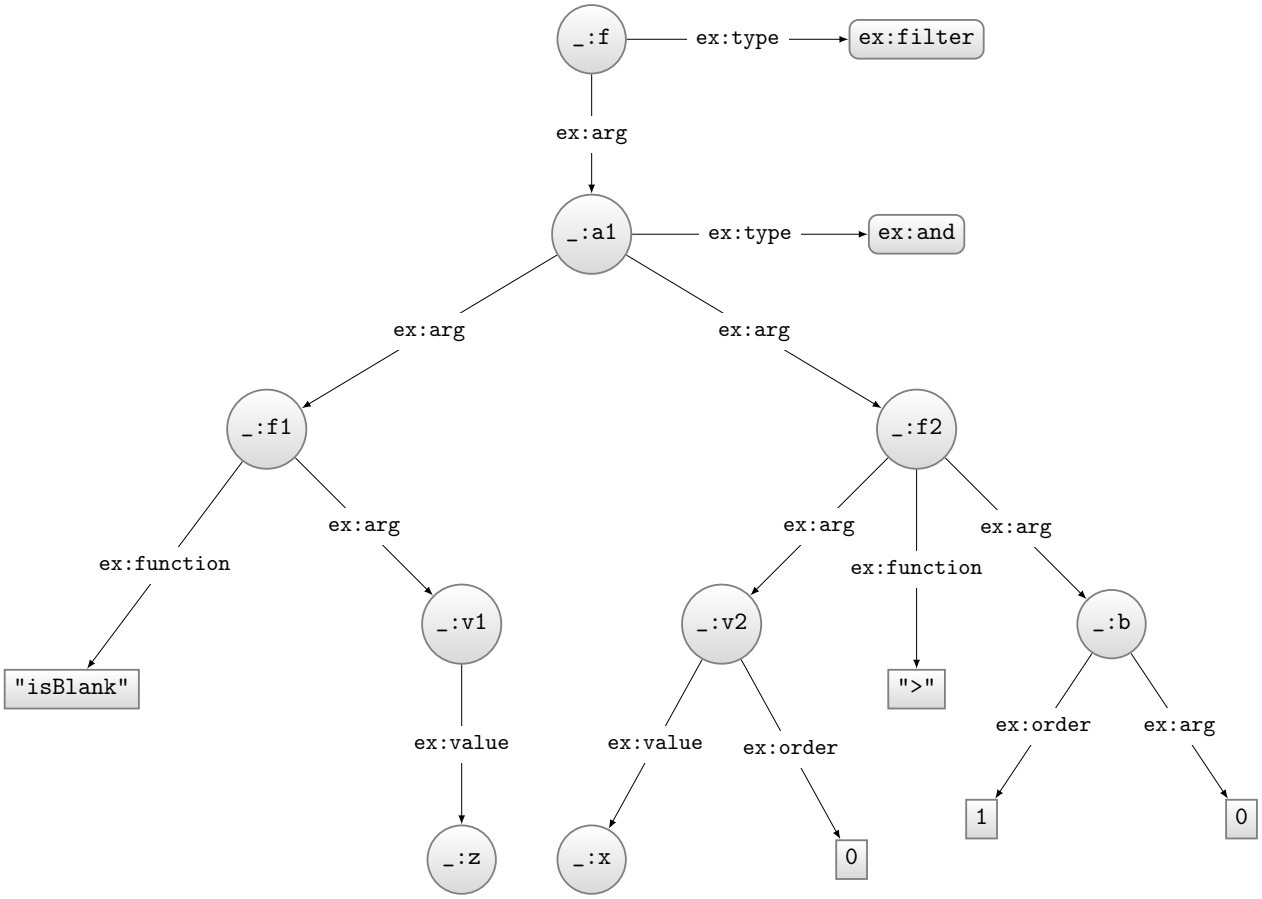

Figure 3.20: Representation of a FILTER expression

`ex:order` edge connected to a literal node whose value is either 0 or 1 (for binary operators, which are the only cases we cover) for the argument on the left or the right, respectively.

Analogous to how we represent "Join" and "Union" expressions – an operator node connected to its operands by `ex: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 previously.

Built-in functions –such as `isBlank` and `isBound`– as well as equality or inequality symbols are represented as blank nodes connected by an `ex: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 as well be represented by an IRI.

However, we do not yet support the congruence of the query patterns in Figure 3.21

The constraint defined by the filter expression in the first query effectively forces the resulting query to return all triples whose object is 10; the same as in the second query. Handling such cases is left for future work.

```
?x ?y ?z .
FILTER( ?z == 10 )
```

```
?x ?y 10 .
```

Figure 3.21: Example of an unsupported FILTER equivalence

3.6.3 ORDER BY

Let $V = \{v_1, ..., v_n\}$ be the set of variables we want to order the results of a query by.

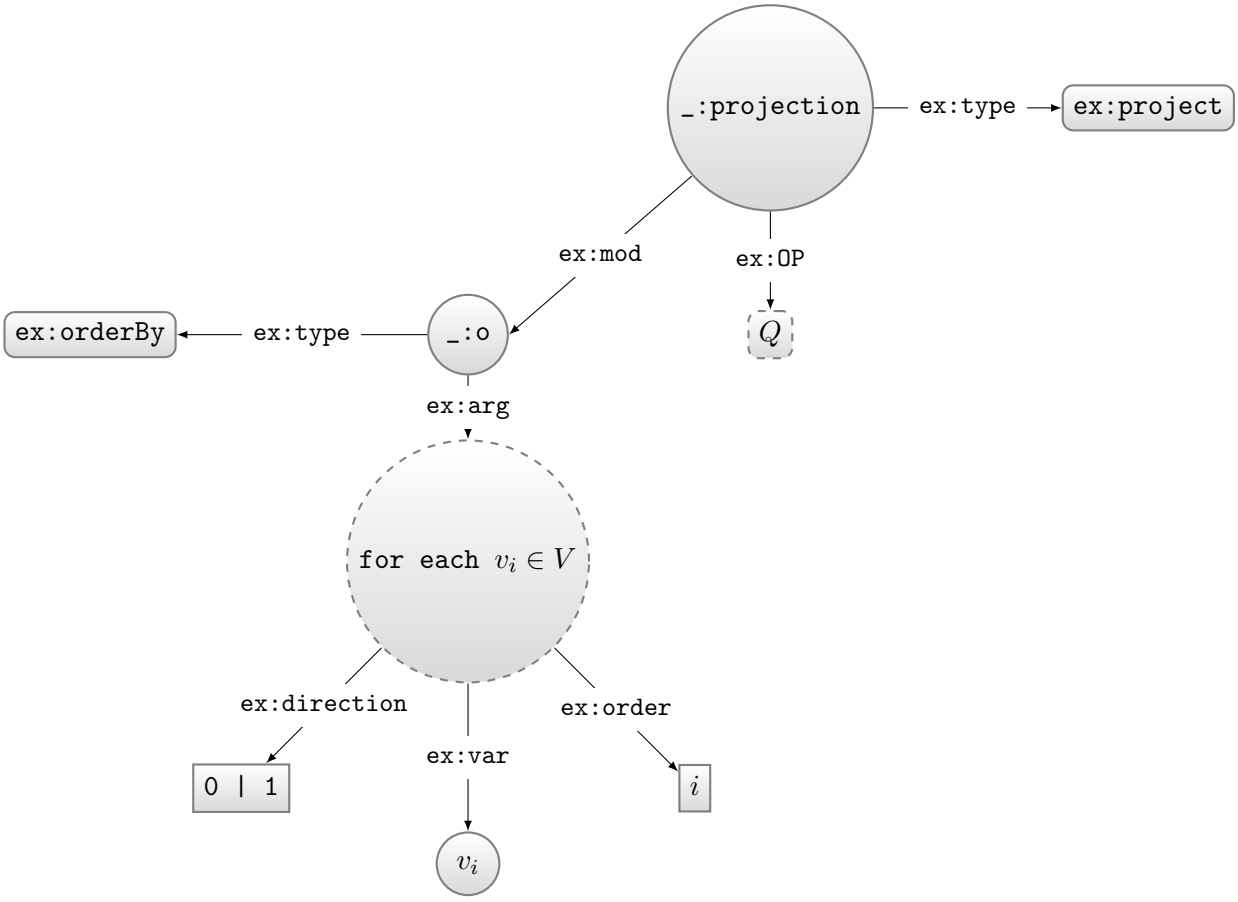

Figure 3.22: Representation of an ORDER BY clause

Figure 3.22 illustrates the representation of an ORDER BY clause. This r-graph is connected to the projection node of a query Q by an `ex: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 `ex:direction` edge which indicates if the variable is ordered by ascending or descending order; a variable node connected by an `ex:var` edge; and also a literal node which indicates the order priority of the variable.

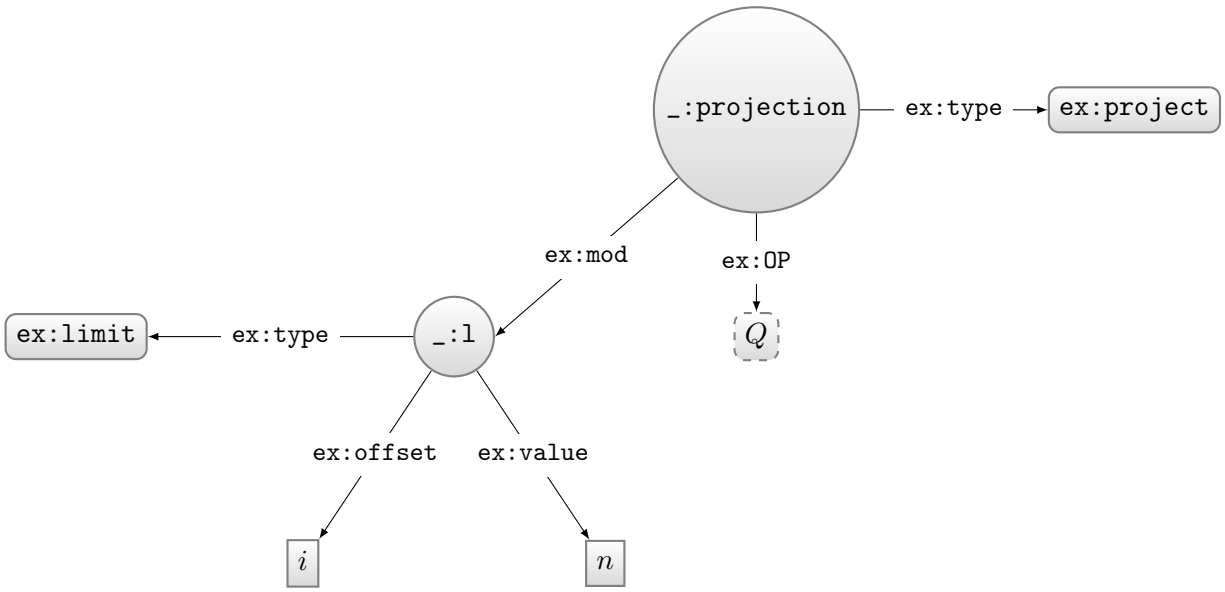

Figure 3.23: Representation of an OFFSET and LIMIT clause

3.6.4 OFFSET and LIMIT

Figure 3.23 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 3.22. 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. If an OFFSET clause is omitted, then its value is 0 by default.

3.6.5 Named Graphs

Let G be a set of graph IRIs which we want to use to define the default graph. The query pattern represented by Q will match the merge of the graphs in G. If empty, the default graph is an empty graph. Let N be a set of named graph IRIs. The query pattern described in Q will *not* match any of the graphs specified in N unless the keyword GRAPH is used, which will be represented shortly.

Figure 3.24 illustrates the representation of a query which 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 `ex: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 that selects a named graph in the SPARQL dataset.

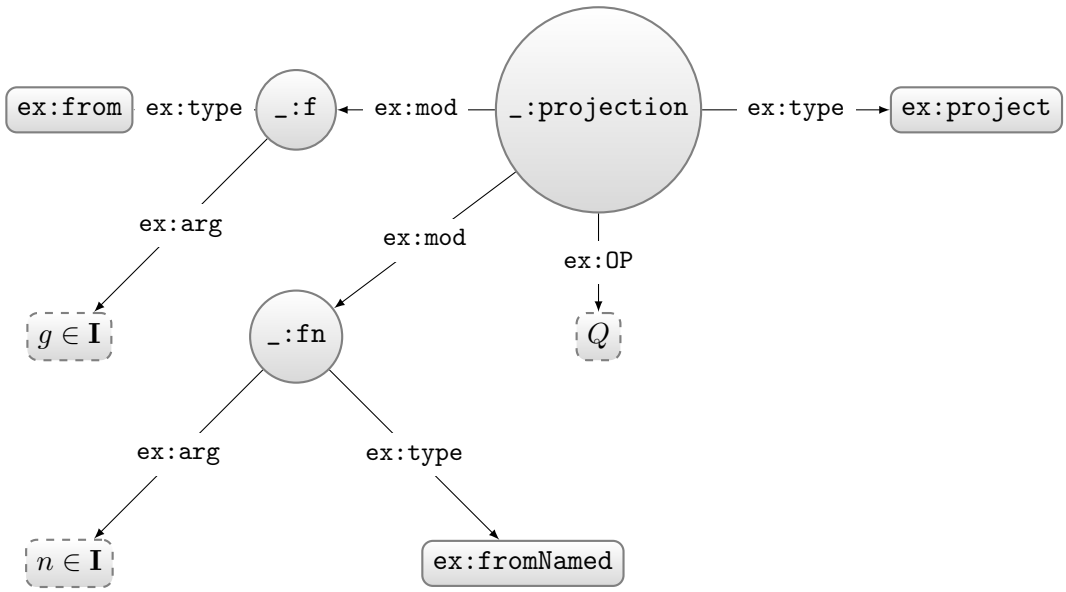

Figure 3.24: Representation of a FROM clause

3.6.6 GRAPH

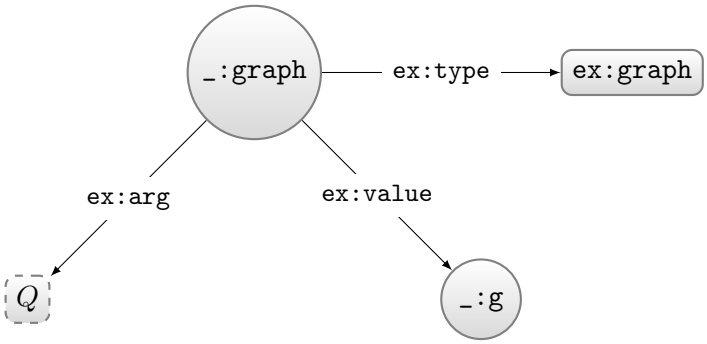

Figure 3.25: Representation of a GRAPH clause with a variable

Figure 3.25 illustrates the representation of a GRAPH clause. This r-graph is connected to either a blank node –if the graph in the GRAPH clause contains a variable– or an IRI node –if the graph in the GRAPH clause is a named graph IRI. It is also connected to a query Q which corresponds to the query which is to be matched to the graph described in the GRAPH clause.

3.6.7 Canonicalisation of Other Features

When we are presented a query that contains features discussed in this section, we do not provide a complete canonicalisation, but we provide a best-effort canonicalisation that guarantees soundness. As a result, this may fail to detect some congruent queries, but the semantics of the queries are not altered.

The idea behind this best-effort canonicalisation is to detect monotone sub-queries, and

perform the canonicalisation process over these monotone sub-queries. Variables that exist outside the sub-query are projected from the sub-query. We can use the r-graph representations of all the features that have been mentioned in this section, but we do not perform any algebraic rewritings on these features. Finally, we apply a canonical labelling over the r-graph and then retrieve the SPARQL query using an inverse mapping.

The reason we feel this best-effort canonicalisation is important is because many real-world queries contain features such as FILTER, OPTIONAL, etc, which are outside the monotone queries we have focused on. This will become evident in the experiments described in Chapter 6.

Chapter 4

Implementation

In this chapter we describe the tools –mainly software– which were used to implement the structures, representations and techniques described in the previous chapter. 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.

We now describe each step in turn:

4.1 Parsing the Query

Utilizing the toolkit provided by Jena ARQ, raw queries are parsed into an algebraic form.

The data structure used to store the algebraic form consists of nested operations. These structures are akin to binary trees where each node is an operation. The graph is created by traversing the algebraic expression, using a *visitor* class also provided by Jena ARQ, which we have extended in order to map each operation to a set of nodes as described in Chapter 3.

The r-graphs we have defined are structured as a Java class containing:

- A default graph provided by the factory class `GraphFactory` in Apache Jena.
- A `HashSet` of variables of class `Var`

- An integer to save the total number of triples in the graph.
- An instance of the class Node corresponding to the root of the r-Graph.
- Two boolean values to indicate if we are working with bag or set semantics.
- A string containing the default IRI prefix we are working with.
- An instance of every node that is used repeatedly. For instance `ex:arg` and `ex:type`.

This class contains multiple instantiations.

The one that was first designed receives a list of triples, a list of variables, and an identification number. The information of a triple is extracted as follows:

```
reifyBGP(T, i):
        G:={}
        j:=0
        for t in T:
                t:=(subject, predicate, object)
                v:=newBlankNode(i, j++)
                addTriple(subject, :subject, v, G)
                addTriple(predicate, :predicate, v, G)
                addTriple(object, :object, v, G)

addTriple(x, iri, v, G):
        if isVariable(x):
                b:=newBlankNode(x)
                c:=createTriple(v, iri, b)
        else:
                c:=createTriple(v, iri, x)
        addInto(G, c)
```

This is used when processing a BGP and its output depends on the number of triples input:

- If only one triple was input, it returns a reification of that triple, with a fresh blank node labelled tp_N (for triple pattern) with N being the identification number as its root node.
- If more that one triple was input, it returns a fresh blank node labelled $join_i$ (with i being the identification number) as its root, connected to a reification of all the triples by the predicate `ex:arg`. The triples are labelled TP_{ij} where i is the identification number of the root node, and j is a counter that increases with every triple processed.

Another instantiation function receives an instance of a Collectible class that contains the triples that form an r-graph, and outputs the r-graph. This is mostly used when we need to use the features provided by the *blabel* package, which will be described in greater detail later in this chapter.

Further algebraic operations, such as union, projection, etc, are handled in a similar manner which we do not discuss for the purposes of brevity.

4.2 Normalisation

For the UCQ normalisation processes, we utilise SPARQL Update queries with Jena ARQ. These queries allow the insertion and deletion of triples in RDF graphs, possibly based on the results of a query.

Figure 4.1 contains a SPARQL Update script used to transform a conjunction of a disjunctive query pattern and a triple pattern (i.e. a join of a union and a triple pattern) into a disjunction of conjunctive query patterns. Though both transformations are the result of associativity, we were unable to design a script which captured both cases successfully mainly because of concurrency issues. Therefore special care had to be taken in each individual case. This is the script for a query Q of the form $Q = (A \cup B) \bowtie C$; a similar script is used for queries of the form $Q = (A \cup B) \bowtie (C \cup D)$.

Sometimes, as the result of the application of the above transformation, some redundancy was added in the form of a string of nodes denoting the same operator (i.e. a union of unions or join of joins). Both union and join are associative operators, so instead of nesting multiple binary operator nodes, they could be denoted by a single operator node with multiple operands. To address this issue, other scripts had to be designed to detect these redundant cases and collapse the string of operators into a single one.

The examples shown in Figures 4.2 and 4.3 illustrate the rules used to capture the associative equivalence of join and union, respectively. In essence, if we have $(n - 1)$ nested binary join operators, they can be merged into a single n-ary join operator. This is the same for union.

Figure 4.4 contains a script used to detect *dangling* joins, where there is a join of joins, and no other type of node. In this case, we can replace the whole structure with a single instance of a join.

4.3 Canonical Labelling

We use the *blabel* `https://blabel.github.io/` library for the labelling and leaning of the r-graphs.

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

```
PREFIX ex:  <http://example.org/>

DELETE{
  ?b ?y ?b0 .
  ?b0 ex:type ex:join .
  ?b0 ex:arg ?b1 .
  ?b1 ex:type ex:union .
  ?b1 ex:arg ?q1 .
  ?b0 ex:arg ?b2 .
}
INSERT{
  _:j ex:type ex:join .
  _:j ex:arg ?q1 .
  _:j ex:arg ?b2 .
  ?b1 ex:arg _:j .
  ?b1 ex:type ex:union .
  ?b ?y ?b1 .
}
WHERE
  {

  {
  SELECT DISTINCT ?b0 ?b1 ?q1 ?b2
  WHERE{
    {
    SELECT ?b0 ?b1
        WHERE{
          ?b0 ex:type ex:join .
          ?b0 ex:arg ?b1 .
          ?b1 ex:type ex:union .
          FILTER NOT EXISTS{
            ?b0 ex:arg ?v .
            ?v ex:type ex:union .
            FILTER(?v != ?b1)
          }
          }
          LIMIT 1
      }
      ?b1 ex:arg ?q1 .
      ?b0 ex:arg ?b2 .
      FILTER( ?b1 != ?b2 )
      }
    }
    OPTIONAL {?b ?y ?b0 FILTER(?y = ex:arg || ?y = ex:OP)} .
  }
```

Figure 4.1: Example of a script used for converting a JOIN that contains a single UNION into a UNION

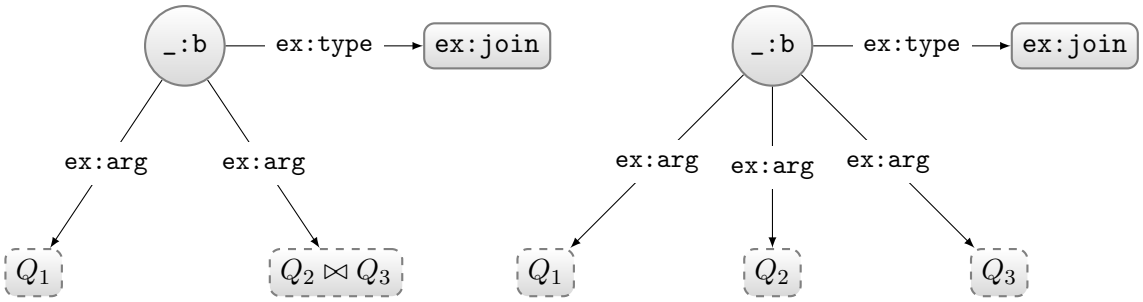

Figure 4.2: Associative property of JOIN

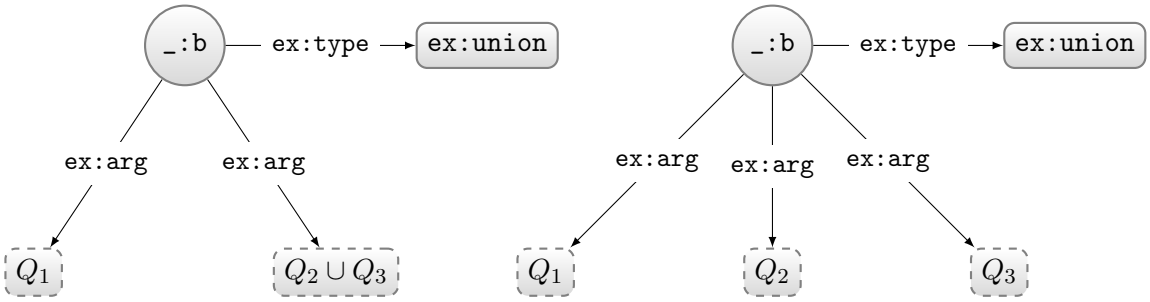

Figure 4.3: Associative property of UNION

```
PREFIX ex:  <http://example.org/>

DELETE { ?b0 ex:arg ?b1 .
    ?b1 ?p ?o }
INSERT { ?b0 ex:arg ?q }
WHERE
  {
    ?b0 ex:type ex:join .
    {
      ?b1 ex:type ex:join .
      ?b1 ex:arg ?q .
      ?q ex:type ex:TP .
      ?b1 ?p ?o
    }
    FILTER NOT EXISTS{
      ?b1 ex:arg ?q .
      ?q ex:type ?type .
      FILTER(?type != ex:TP)
    }
    ?b0 ex:arg ?b1 .
  }
```

Figure 4.4: Example of a script

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.

4.4 Minimisation

To implement minimisation, we first assume that all conjunctive queries contain the same projected variables. In the following procedure, let G_i and G_j be r-graphs. We will denote $G_i \vdash G_j$ if G_i is entailed by G_j and G_j is not entailed by G_i (see Section 2.3.6). We proceed as follows:

```
1  G = {G_i | G_i = rGraph(Q_i)}_{i=1...n}
2
3  for each G_i in G:
4          for each G_j ≠ G_i , j < i:
5                  if G_i ≡ G_j :
6                          G = G \ G_j
7
8  for each G_i in G:
9          for each G_j ≠ G_i :
10                 if G_j ⊢ G_i :
11                         G = G \ G_j
```

The result of the above algorithm is a set containing r-graphs representing all non-redundant CQs [12].

To ascertain entailment, we perform the following procedure: given G_i and G_j as different branches where we want to check if $G_j \sqsubset G_i$, we produce the query form of the r-graph G_i, replacing the projection with an ASK clause, and evaluate over G_j. If it holds true, we conclude that G_j entails G_i, therefore we add G_j (the more specific query) to the list of redundant CQs, since G_i does not entail G_j.

Figure 4.5 illustrates the ASK query form of the conjunctive query shown in Figure 3.13; it is denoted as G_i. The set of triples in Figure 4.6 correspond to the triples contained in the r-graph for the right-hand-side CQ shown in Figure 3.8, which will be denoted as G_j. The query in Figure 4.5 will be evaluated over the triple set in Figure 4.6 and return *true*, implying that G_j entails G_i. Therefore, G_j will be removed from the set of conjunctive queries. Note that the labels in Figure 4.5 and Figure 4.6 don't have to be the same for the process to

```
ASK
WHERE{
 _:j1 ex:arg _:t1 .
 _:j1 ex:arg _:t3 .
 _:t1 ex:sub _:a1 .
 _:t1 ex:pred :title .
 _:t1 ex:obj _:m .
 _:m ex:value "m" .
 _:m ex:temp "true"^^xsd:boolean .
 _:t3 ex:sub _:d1 .
 _:t3 ex:pred :directed .
 _:t3 ex:obj _:a1 .
}
```

Figure 4.5: Example of an ASK query

```
 _:j2 ex:arg _:t4 .
 _:j2 ex:arg _:t5 .
 _:j2 ex:arg _:t6 .
 _:t4 ex:sub _:a2 .
 _:t4 ex:pred :title .
 _:t4 ex:obj _:m .
 _:m ex:value "m" .
 _:m ex:temp "true"^^xsd:boolean .
 _:t5 ex:sub _:a2 .
 _:t5 ex:pred :actor .
 _:t5 ex:obj _:p2 .
 _:t6 ex:sub _:d2 .
 _:t6 ex:pred :directed .
 _:t6 ex:obj _:a2 .
```

Figure 4.6: Example of a CQ

work, because blank nodes are existential variables whose labels are irrelevant to entailment.

4.5 Generation of Queries

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.

This query is constructed by traversing the r-graph starting from the projection node. However, the preamble must be resolved before processing the actual query.

Prefixes are omitted because IRIs are stored with their full value in the r-graphs. Afterwards, we must check for any definitions for the default graph or set of named graphs.

Following this, we add the SELECT clause with the DISTINCT keyword and/or projected variables, if applicable. We order projected variables by the canonical label.

We move on to the construction of the body of the query, starting with the outermost union, join or triple pattern operator. The string is constructed recursively. The base case is when we encounter a triple pattern node.

Let $str(G)$ be a function that outputs the query represented by r-graph G. We give a few examples to illustrate this idea:

$str(G)$ where G is of the form $Q_1 \bowtie Q_2$: $str(G) = \ str(Q_1)\ .\ str(Q_2)$

$str(G)$ where G is of the form $Q_1 \cup Q_2$: $str(G) = \ str(Q_1)$ UNION $str(Q_2)$

$str(G)$ where G is of the form $Q_1 \ltimes Q_2$: $str(G) = \ str(Q_1)$ OPTIONAL $str(Q_2)$

$str(G)$ where G is a triple pattern (s, p, o): $str(G) = \ s\ p\ o$

Since G is canonical, we can first order the triples of G to ensure that this process produces a deterministic order of query patterns.

4.6 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.

Figure 4.7 contains three SPARQL queries that are equivalent. 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 ensure the correctness of the leaning process. This is an example of a positive test-case, where we test that all canonicalised queries are equivalent. We have also defined negative examples where we test that certain queries must not be equivalent under canonicalisation.

```
PREFIX ex: <http://ex.org/>
SELECT DISTINCT ?x
WHERE {
  ?x ex:p ?b .
  ?x ?q ?r .
  ?y ex:p ?b .
}

====================================

PREFIX ex: <http://ex.org/>
SELECT DISTINCT ?y
WHERE {
  ?x ex:p ?b .
  ?x ?q ?r .
  ?y ex:p ?b .
}

====================================

PREFIX ex: <http://ex.org/>
SELECT DISTINCT ?z
WHERE {
  ?z ex:p ?b .
  ?z ?q ?r .
  ?x ex:p ?b .
  ?x ?q ?r .
  ?y ex:p ?b .
}

====================================
```

Figure 4.7: Example of a positive test case for leaning and canonical variable labelling

Chapter 5

Results and Evaluation

For the purposes of this study, we define the following research questions:

- 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.

5.1 Queries Used

It is evident that the techniques described in Chapter 3 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.

5.1.1 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. This dataset contains queries taken from the logs of four 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.

Table 5.1: Number of LSQ queries with individual features (left) and combinations of UCQ features (right)

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

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

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 5.1 shows the distribution of features in the LSQ dataset. First we count the individual features contained in the queries, and then we count the combinations of UCQ features in queries where there is at least one instance of a UNION. We can observe that UNION is used far less than other features such as OPTIONAL and FILTER. However, 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 (discussed in Section 2.3.4). In terms of UCQ features, the vast majority of queries contain several other features such as DISTINCT (in approximately 18.7% of the queries), whereas queries that contain UNION and JOIN are far less common. Queries were labelled as *unsupported* if they contained any features that have not been mentioned in this work (such as SPARQL 1.1 features).

5.1.2 Synthetic CQs

Since most real-world queries turned out to be trivial to canonicalise, in order to better test the proper functionality of the canonicalisation process, a series of synthetic queries were created based on well-known types of undirected graphs.

We describe the following types of graphs which were used in 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 5.1.

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 5.2.

Clique: For $k \geq 2$, the k-clique contains k nodes that are pairwise connected; this graph has a total of $\binom{k}{2}$ edges. See an example in Figure 5.3.

Miyazaki: This class of graphs was designed by Miyazaki [14] 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 5.4.

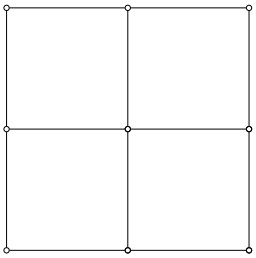

Figure 5.1: 3-2D-Grid

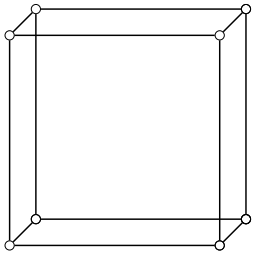

Figure 5.2: 2-3D-Grid

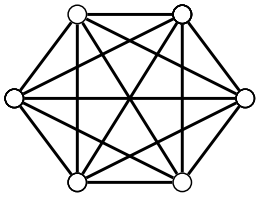

Figure 5.3: 6-clique

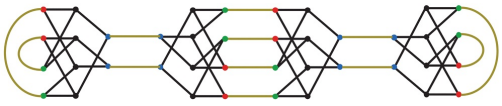

Figure 5.4: Miyazaki graph (taken from http://vlsicad.eecs.umich.edu/BK/SAUCY)

From these undirected graphs we create queries where nodes are variables connected by a fixed predicate.

Let us say that i and j are two adjacent nodes, and their respective variables are v_i and v_j. We create a triple $(v_i, : p, v_j)$ for every edge, and construct a conjunction of all such triples.

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 randomly chosen subject from all the triples. Note that under set semantics, 2D-Grid and 3D-Grid graphs collapse to a core that consists of 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.

5.1.3 Synthetic UCQs

The previous queries only represent CQs and would not test UCQ normalisation. Hence we have designed synthetic UCQs using 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.

Figure 5.5 shows a synthetic UCQ for $n = 2$ and $m = 4$

```
SELECT DISTINCT  ?v1
WHERE
  {   {   {   { ?v1  <http://example.org/p>  ?v2 }
            UNION
              { ?v1  <http://example.org/p>  ?v3 }
          }
        UNION
          { ?v4  <http://example.org/p>  ?v3 }
      }
    UNION
      { ?v3  <http://example.org/p>  ?v2 }
      {   {   { ?v1  <http://example.org/p>  ?v2 }
            UNION
              { ?v3  <http://example.org/p>  ?v2 }
          }
        UNION
          { ?v1  <http://example.org/p>  ?v4 }
      }
    UNION
      { ?v4  <http://example.org/p>  ?v3 }
  }
```

Figure 5.5: Example of a synthetic UCQ

The purpose of these queries is to test the performance of our algorithm in difficult cases, particularly the UCQ rewriting and redundancy elimination. For this evaluation, we consider the case when the input query is in its conjunctive normal form.

5.2 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.

5.3 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.

For our algorithm, we measure the time it takes to construct the r-graph as well as the time it takes to perform the canonical labelling, and the leaning process, if applicable.

We compare three variants for the detection of duplicate queries:

Syntactic We parse the queries using Jena ARQ and compare each algebraic expression.

Label We parse the queries and construct their respective r-graphs, normalise them, and apply the canonical labelling process.

Full Same as above, but we also apply leaning.

5.3.1 Real-world Queries

We begin by presenting the results of the experiments on real-world queries.

Table 5.2: High-level results for canonicalising LSQ queries, including the total time taken and duplicates found

Algorithm	Time (s)	Dup.	Max. dup.	Queries
Syntactic	211	3,960	12	768,618
Label	28,066	10,722	307	768,618
Full	77,023	10,722	307	768,618

Table 5.2 displays the results obtained for each variant. We can appreciate that on average, the Label variant takes about 132 times what the Syntactic takes. On the other hand, the Full variant takes about 2.8 times what the Label variant takes.

With regards to the number of duplicate queries found, our algorithm detects about thrice as many as baseline syntactic methods. In particular, the most duplicate queries found were at least 30 times as many as baseline syntactic methods.

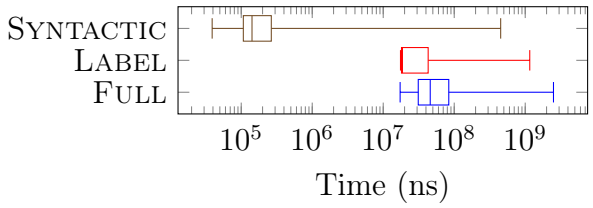

Figure 5.6: Runtimes for canonicalising LSQ queries

These slower results were expected, since our algorithm is doubly-exponential, as we have already mentioned. This isn't a major concern, however, since Figure 5.6 shows that most individual queries were processes in just under 90ms, and the variance of the runtimes is very low. This is due to the simplicity of most real-world queries.

5.3.2 Synthetic CQs

We now present the results of the experiments on synthetic CQs.

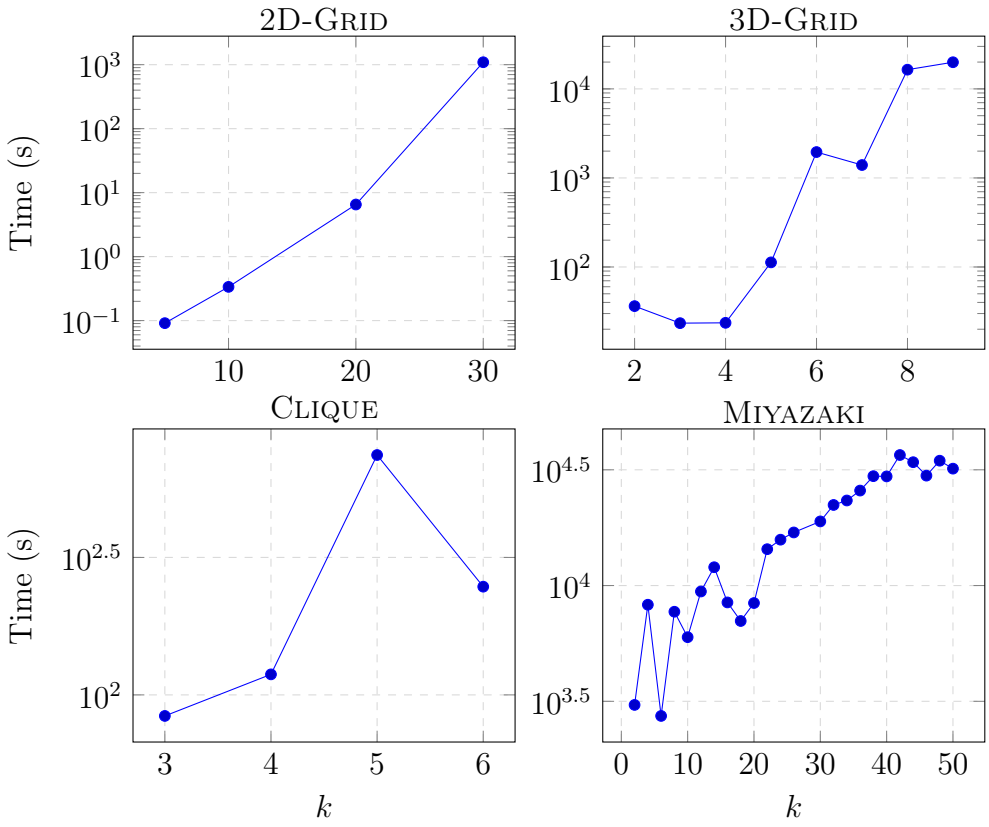

Figure 5.7: Runtimes for synthetic CQs

Figure 5.7 illustrates the runtimes –which consist in both the graph creation times and canonicalisation times– for the different types of graphs we defined previously. These times are displayed in log scale. Upon analysing the data shown in Figure 5.7, we can observe that for `2D-Grid` and `3D-Grid` graphs, runtimes increase exponentially as we increase their

k value, which is an expected result since both the number of nodes and edges also increase polynomially and the algorithms have exponential behaviour. `Miyazaki` graphs runtimes increase less quickly, and are able to be processed efficiently up to considerably higher values of k than other graphs. Finally, `Cliques` for $k \leq 6$ can be canonicalised.

5.3.3 Synthetic UCQs

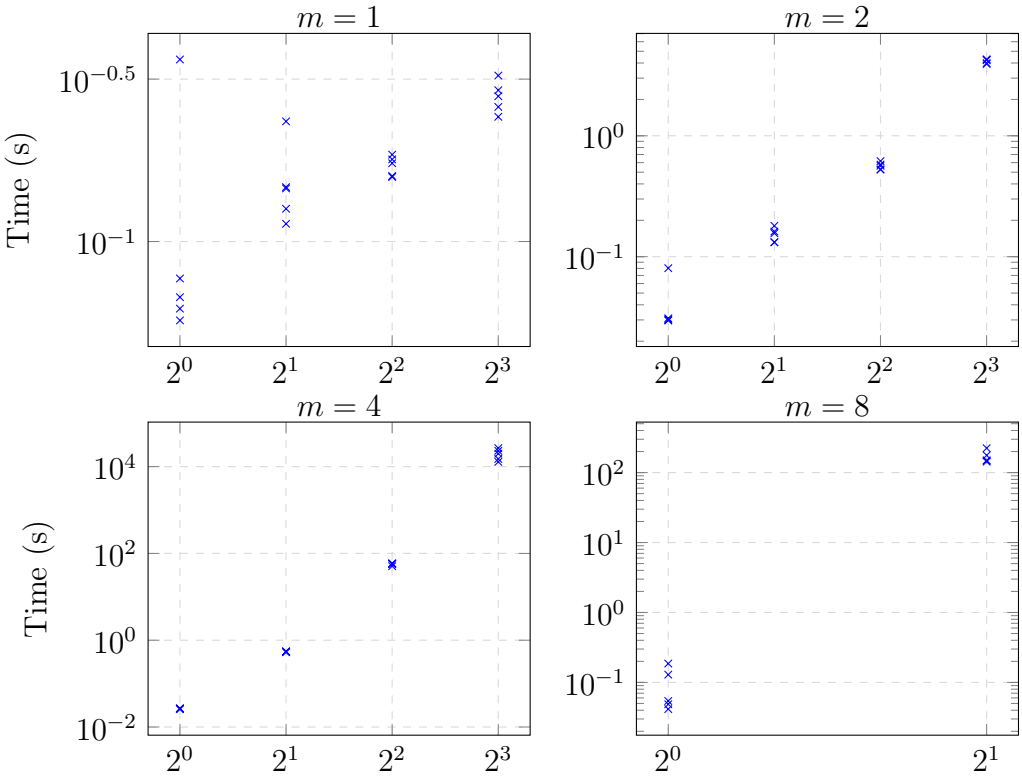

Figure 5.8: Runtimes for synthetic UCQs for varying m and n

The graphs presented in Figure 5.8 each have a number of conjunctions (m value) while the x-axis presents the number of unions inside each disjunctive query (n value). On the other hand, the graphs presented in Figure 5.9 each have a number of unions fixed, while the x-axis presents the number of conjunctions.

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 8 before becoming too complex to be canonicalised in mere seconds or even minutes. This makes it clear that the runtimes grow quickly in such cases, which was the expected result.

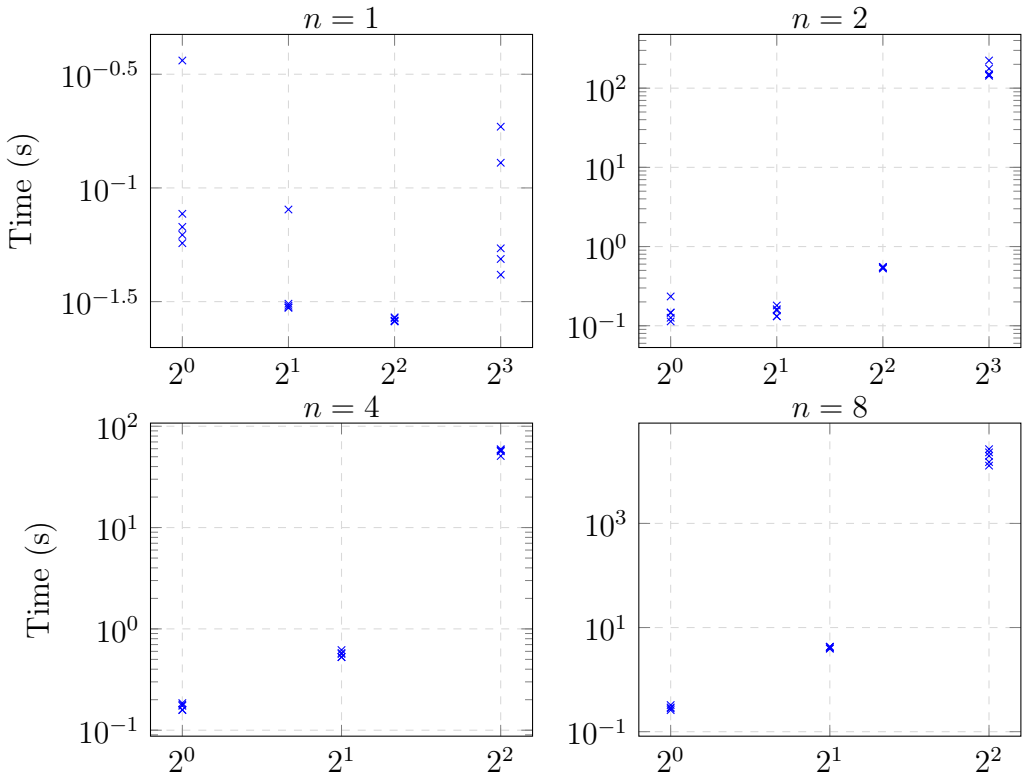

Figure 5.9: Runtimes for synthetic UCQs for varying m and n

5.4 Discussion

Now that we have presented the results, we are ready to answer the research questions we proposed at the beginning of the chapter.

How is the performance of the canonicalisation process for real-world queries?

The results indicate that most of the queries were canonicalised in under a second, which leads us to conclude that the proposed method is efficient for the majority of real-world queries.

Given that our algorithm is doubly-exponential, at what point does the algorithm begin to fail?

With regards to real-world queries, the algorithm does not fail. The experiments on synthetic queries show that the algorithm may fail for UCQs that contain over 8 conjunctions of query patterns that consist in the union of over 4 triple patterns. For synthetic queries based on well-known patterns, the algorithm begins to fail for graphs that contain over 1700 edges, which translate into query patterns that contain the same number of triple patterns.

How many more duplicate queries can be detected using our procedure versus baseline syntactic methods?

The results show that our procedure can detect over 20 times as many duplicate (congruent) queries as the baseline method we used in our experiment.

Whether or not the overhead of the canonicalisation methods would be justified in a practical caching system requires further experiments which is left for future work.

What is not considered in these synthetic experiments is the presence of multiple projections, any solution modifiers or filter expressions. We have yet to determine appropriate synthetic test cases for those features, but they are not key to the initial evaluation of this algorithm. Furthermore, such features are present in the real-world queries tested previously.

Chapter 6

Conclusion

6.1 Summary

To summarise, we have explained the problems that may arise for queries due to the lack of semantic equivalence tools used in SPARQL endpoints. We have also shown that the increasing use of RDF for the representation of information as well as SPARQL for the querying of this information, has made it more necessary to solve this issue. Following this, we have described our hypothesis, in addition to the methodology we designed to test the hypothesis. To introduce us to the solution, we have detailed certain concepts of graph theory such as sub-graphs, homomorphisms, and graph cores. These are all necessary to understand the steps we described later. We also described the two aspects of graph canonicalisation which we must consider to perform a full canonicalisation: iso-canonicalisation and equi-canonicalisation. The iso-canonicalisation is managed by a NAUTY style canonical labelling; the equi-canonicalisation is managed by the computation of the core of a graph, in a process called *leaning*.

The process for canonicalising queries has been divided into the following steps: represent the query as an RDF graph, apply a canonicalisation process to the RDF graph, and convert the canonicalised graph back into a canonical SPARQL query. The first step –the representation of the query as an RDF graph– was explained in detail, enumerating and describing all of the structures (as RDF graphs) we used to represent the main features for which we support canonicalisation. For the first part of the canonicalisation, we gave an outline of the transformations we perform on the r-graphs to capture properties of join and union operations, such as their associative and distributive properties. The next part describes the algorithm used to canonicalise UCQs, which consists in: the canonical labelling of each distinct conjunctive query, and subsequent leaning; the check for entailment of conjunctive queries in other conjunctive queries; and finally the merge of all the conjunctive queries back into a UCQ. The last part describes briefly the process used to convert the r-graph back into a canonical SPARQL query.

The implementation chapter provides a closer look at the tools –mainly software– that were used in order to design, construct and transform raw queries, starting with the parsing

of the query into an algebraic expression, using Apache Jena; followed by the normalisation process implemented with Jena ARQ's SPARQL Update engine; the canonical labelling and leaning, where we used the `blabel` package for the canonical labelling and leaning of blank nodes in RDF graphs; the generation of queries using recursive functions and the reverse of the transformations described in Chapter 3; and finally an overview of the test cases we designed to test the correctness of the main aspects of our canonicalisation algorithm.

Finally, we describe the experiments that were carried out to prove the efficiency of the algorithm in real-world queries, the point at which the algorithm breaks and can no longer process a query in a reasonable time, and the number of duplicated queries our method can detect versus the number of duplicated queries that syntactic equivalence tools can detect. We present the results of the experiments designed to prove the research questions we postulated, as well as a justification of the results obtained. As a final result, we can conclude that the algorithm works in the vast majority of real-world queries, and is capable of detecting a considerably higher number of duplicated queries than tested, syntactic normalisation tools.

6.2 Lessons Learned

Our hypothesis stated that we could build a practical canonicalisation algorithm for real-world queries, which was proven possible because many real-word queries are simple, and were able to be processed in times of a few milliseconds, as shown by the results presented.

Although we have explained why canonicalisation is a hard problem, especially considering that the equivalence problem in SPARQL becomes undecidable for the full language, we have managed to cover a majority of SPARQL queries by selecting a number of features we have identified as key to the expressive power of SPARQL.

We have also learned that hard cases are also rarely come by in practice, which was made evident by the fact that the algorithm in our experiments only failed for difficult *synthetic* cases.

This has taught us that the fact that a problem has a hard worst-case complexity does not mean that the problem cannot be tackled with a practical solution in an efficient manner if the vast majority of cases that occur in practice can be solved in a reasonable time.

6.3 Future Work

We have covered the majority of the features of SPARQL 1.0 and can already anticipate some further work.

The most logical first step would be in the form of supporting SPARQL 1.1 by extending the current canonicalisation algorithm to include features such as property paths, or sub-queries.

Other future work we can see is a fuller canonicalisation of features of SPARQL 1.0 such as OPTIONAL, where the literature already contains proposals for the simplification of the OPTIONAL equivalence problem under certain restrictions. With more research, a process before the canonicalisation algorithm may be carried out to ascertain if the OPTIONAL expressions follow these restrictions or not.

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. However, a considerably larger amount of research would have to occur to allow this.

The process can also be applied to other querying languages such as SQL, which are also widely used, and perhaps also have a larger number of simple queries than complex ones.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 0-201-53771-0. URL: http://webdam.inria.fr/Alice/.

[2] Harith Alani et al., eds. *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*. Vol. 8219. Lecture Notes in Computer Science. Springer, 2013. ISBN: 978-3-642-41337-7. DOI: 10.1007/978-3-642-41338-4. URL: http://dx.doi.org/10.1007/978-3-642-41338-4.

[3] Carlos Buil Aranda et al. "SPARQL Web-Querying Infrastructure: Ready for Action?" In: *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*. Ed. by Harith Alani et al. Vol. 8219. Lecture Notes in Computer Science. Springer, 2013, pp. 277–293. ISBN: 978-3-642-41337-7. DOI: 10.1007/978-3-642-41338-4_18. URL: http://dx.doi.org/10.1007/978-3-642-41338-4_18.

[4] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. "Foundations of RDF databases". In: *Reasoning Web International Summer School*. Springer. 2009, pp. 158–204.

[5] Dan Brickley, R.V. Guha, and Brian McBride. *RDF Schema 1.1*. W3C Recommendation. http://www.w3.org/TR/rdf-schema/. Feb. 2014.

[6] Derek G Corneil and Calvin C Gotlieb. "An efficient algorithm for graph isomorphism". In: *Journal of the ACM (JACM)* 17.1 (1970), pp. 51–64.

[7] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. http://www.w3.org/TR/rdf11-concepts/. Feb. 2014.

[8] Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, eds. *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. ACM, 2015. ISBN: 978-1-4503-3469-3. URL: http://dl.acm.org/citation.cfm?id=2736277.

[9] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. *SPARQL 1.1 Query Language*. W3C Recommendation. http://www.w3.org/TR/sparql11-query/. Mar. 2013.

[10] Pascal Hitzler et al. *OWL 2 Web Ontology Language Primer*. W3C Recommendation. http://www.w3.org/TR/owl2-primer/. Oct. 2009.

[11] Aidan Hogan. "Canonical Forms for Isomorphic and Equivalent RDF Graphs: Algorithms for Leaning and Labelling Blank Nodes". In: *ACM Transactions on the Web (TWEB)* 11.4 (2017), p. 22.

[12] Aidan Hogan and Jaime Salas. "Canonicalisation of Monotone SPARQL Queries". In: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Mon-*

terrey, California, USA, October 8-12, 2018. (to appear; see extended version online http://aidanhogan.com/qcan/extended.pdf). 2018.

[13] Brendan D McKay and Adolfo Piperno. "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112.

[14] Takunari Miyazaki. "The complexity of McKay's canonical labeling algorithm". In: *Groups and Computation II*. Vol. 28. Aer. Math. Soc.: Providence, RI. 1997, pp. 239–256.

[15] Reinhard Pichler and Sebastian Skritek. "Containment and equivalence of well-designed SPARQL". In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Richard Hull and Martin Grohe. ACM, 2014, pp. 39–50. ISBN: 978-1-4503-2375-8. DOI: 10.1145/2594538.2594542. URL: http://doi.acm.org/10.1145/2594538.2594542.

[16] Yehoshua Sagiv and Mihalis Yannakakis. "Equivalences among relational expressions with the union and difference operators". In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 633–655.

[17] Muhammad Saleem et al. "LSQ: The Linked SPARQL Queries Dataset". In: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*. Ed. by Marcelo Arenas et al. Vol. 9367. Lecture Notes in Computer Science. Springer, 2015, pp. 261–269. ISBN: 978-3-319-25009-0. DOI: 10.1007/978-3-319-25010-6_15. URL: http://dx.doi.org/10.1007/978-3-319-25010-6_15.

[18] Guus Schreiber and Yves Raimond. *RDF 1.1 Primer*. W3C Working Group Note. http://www.w3.org/TR/rdf11-primer/. June 2014.

[19] 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*. Ed. by Lora Aroyo et al. Vol. 7031. Lecture Notes in Computer Science. Springer, 2011, pp. 762–777. ISBN: 978-3-642-25072-9. DOI: 10.1007/978-3-642-25073-6_48. URL: http://dx.doi.org/10.1007/978-3-642-25073-6_48.

Appendix A

Figure A.1 is a UML diagram that shows the main classes of the software used to model r-graphs, perform the normalisation transformations, canonical labelling and graph leaning. The class `ExpandedGraph` is the class that contains the data structure that represents an r-graph, and all functions that perform the processes described in Chapter 3. The other classes are used for the experiments on real-world queries.

`SingleQuery` is a class that contains, as the name suggests, a single query. 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.

`QueryParser` is a class that 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.

`Benchmark` is a class used to execute the experiments remotely using a console.

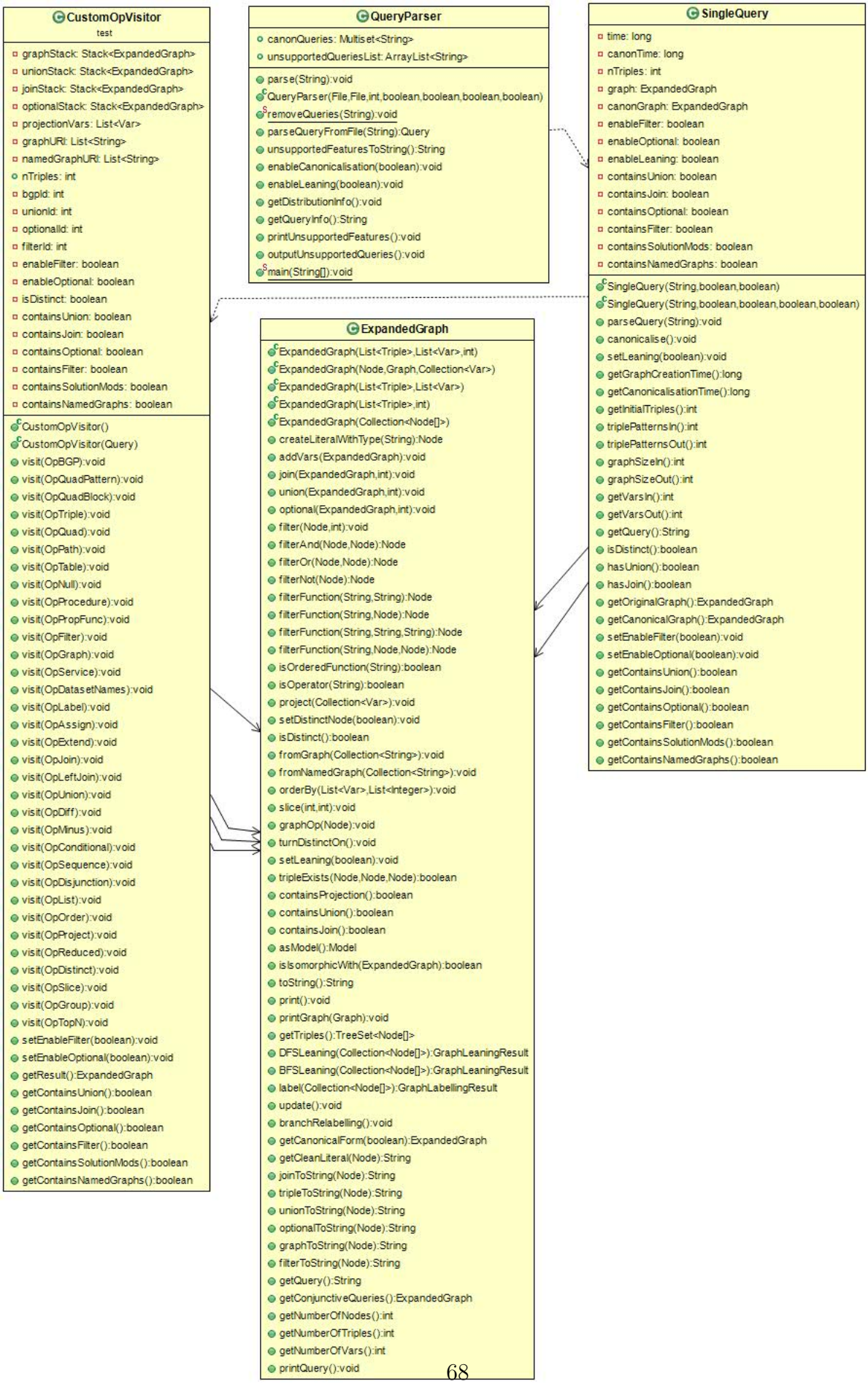

Figure A.1: UML diagram of the software