View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Birkbeck Institutional Research Online

Flexible Query Processing of SPARQL Queries

Riccardo Frosini

A Thesis presented for the degree of
Doctor of Philosophy

Knowledge Lab
Department of Computer Science and Information Systems
Birkbeck, University of London
England

December 2017

Dedicated to

the memory of my mother Lucia Lasciarrea

and my father Lanfranco Frosini.

Declaration

No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Copyright © 2017 by Riccardo Frosini.
"The copyright of this thesis rests with the author. No quotations from it should be published without the author's prior written consent and information derived from it should be acknowledged".

Flexible Query Processing of SPARQL Queries

Riccardo Frosini

Submitted for the degree of Doctor of Philosophy
December 2017

Abstract

SPARQL is the predominant language for querying RDF data, which is the standard model for representing web data and more specifically Linked Open Data (a collection of heterogeneous connected data). Datasets in RDF form can be hard to query by a user if she does not have a full knowledge of the structure of the dataset. Moreover, many datasets in Linked Data are often extracted from actual web page content which might lead to incomplete or inaccurate data.

We extend SPARQL 1.1 with two operators, APPROX and RELAX, previously introduced in the context of regular path queries. Using these operators we are able to support flexible querying over the property path queries of SPARQL 1.1. We call this new language SPARQL^{AR}.

Using SPARQL^{AR} users are able to query RDF data without fully knowing the structure of a dataset. APPROX and RELAX encapsulate different aspects of query flexibility: finding different answers and finding more answers, respectively. This means that users can access complex and heterogeneous datasets without the need to know precisely how the data is structured.

One of the open problems we address is how to combine the APPROX and RELAX operators with a pragmatic language such as SPARQL. We also devise an implementation of a system that evaluates SPARQL^{AR} queries in order to study the performance of the new language.

We begin by defining the semantics of SPARQL^{AR} and the complexity of query evaluation. We then present a query processing technique for evaluating SPARQL^{AR} queries based on a rewriting algorithm and prove its soundness and completeness. During the evaluation of a SPARQL^{AR} query we generate multiple SPARQL 1.1 queries that are evaluated against the dataset. Each such query will generate answers with a cost that indicates their distance with respect to the exact form of the original SPARQL^{AR} query.

Our prototype implementation incorporates three optimisation techniques that aim to enhance query execution performance: the first optimisation is a pre-computation technique that caches the answers of parts of the queries generated by the rewriting algorithm. These answers will then be reused to avoid the re-execution of those

sub-queries. The second optimisation utilises a summary of the dataset to discard queries that it is known will not return any answer. The third optimisation technique uses the query containment concept to discard queries whose answers would be returned by another query at the same or lower cost.

We conclude by conducting a performance study of the system on three different RDF datasets: LUBM (Lehigh University Benchmark), YAGO and DBpedia.

Acknowledgements

Many people deserve my thanks for supporting me during the PhD. My gratitude goes to my supervisor Professor Alexandra Poulovassilis for her invaluable help in shaping this work and her patient guidance. I also thank my supervisor Dr. Andrea Calí for his support and encouragement. My thanks also go to my second supervisor Professor Peter Wood, for his advices and insightful comments.

Additionally, my thanks go to all those at Birkbeck Computer Science department, particularly, Mirko Michele Dimartino. I also thank Sefano Capuzzi for his help in finding my current job.

My thanks go to Jacopo Moccia for helping me distract during some tough moments. Special thanks go to my brother Costantino for taking care of our family during my PhD. Last but not least I thank my girlfriend Martina Segnale for her support and patience particularly in the last year.

Contents

Declaration **3**

Abstract **4**

1 Introduction **17**

1.1 Contributions 23

1.2 Outline of the Thesis 25

2 Background and Related Work **26**

2.1 Background 26

2.1.1 RDF-Graph 27

2.1.2 RDF-Schema 28

2.1.3 SPARQL Syntax 31

2.1.4 SPARQL Semantics 32

2.2 Related Work 35

2.2.1 Flexible Relational Querying 35

2.2.2 Flexible XML Querying 37

2.2.3 Flexible SPARQL Querying 38

2.2.4 SPARQL Semantics, Complexity and Optimisation 40

2.3 Discussion 41

3 Syntax and Semantics of SPARQL^{AR} **42**

3.1 Syntax 42

3.2 Semantics 43

3.2.1 Adding Approximation 44

3.2.2 Adding Relaxation . . . 46
3.3 Query Answers up to a Maximum Cost . . . 50
3.4 Discussion . . . 50

4 Complexity of Query Evaluation in SPARQL^{AR} 52
4.1 Preliminaries . . . 52
4.2 Complexity of SPARQL^{AR} . . . 53
4.2.1 AND-only Queries with Filter Conditions . . . 53
4.2.2 Adding the Flexible Operators . . . 56
4.2.3 Adding UNION . . . 59
4.3 Result Summary . . . 62
4.4 Discussion . . . 63

5 Query Evaluation in SPARQL^{AR} 65
5.1 Rewriting Algorithm . . . 65
5.2 Soundness and Completenesss . . . 71
5.3 Termination . . . 84
5.4 Discussion . . . 85

6 Performance Study 87
6.1 Implementation . . . 88
6.2 Pre-Computation Optimisation . . . 89
6.3 Query Performance Study . . . 92
6.4 Discussion . . . 104

7 Optimisations 107
7.1 Summarisation Optimisation . . . 107
7.1.1 Related Work . . . 108
7.1.2 Our RDF-Graph Summary . . . 110
7.1.3 Performance Study . . . 117
7.2 Query Containment . . . 127
7.2.1 Related Work . . . 127
7.2.2 Preliminary Definitions . . . 131

7.2.3 Containment of Single-Conjunct Queries 134
7.2.4 Query containment based optimisation 138
7.2.5 Performance Study 138
7.3 Combined Optimisations 145
7.3.1 Performance Study 146
7.4 Discussion 151

8 Conclusions 154

8.1 Future Work 156

A Queries for Performance Study in Chapters 6 and 7 168

A.1 LUBM Queries 168
A.2 DBpedia Queries 172
A.3 YAGO Queries 176

B Tables of Results for Performance Study in Chapters 6 and 7 179

B.1 LUBM Results 179
B.2 DBpedia Results 184
B.3 YAGO Results 186

List of Figures

3.1 RDFS entailment rules [62] 47
3.2 Additional rules for extended reduction of an RDFS ontology [67] . . 47

6.1 SPARQL^{AR} system architecture 89
6.2 LUBM. Timings for dataset D1, with and without pre-computation optimisation. 97
6.3 LUBM. Timings for dataset D2, with and without pre-computation optimisation. 98
6.4 LUBM. Timings for dataset D3, with and without pre-computation optimisation. 99
6.5 DBpedia. Timings, with and without pre-computation optimisation. . 102
6.6 YAGO. Timings, with and without pre-computation optimisation. . . 105

7.1 LUBM. Timings for database D1. Plain coloured bars are the summarisation of size 2. Bars with diagonal lines from north-west to south-east are the summarisation of size 3. Bars with diagonal lines from south-west to north-east are the summarisation of size 3 together with pre-computation optimisation. 120
7.2 LUBM. Timings for database D2. Plain coloured bars are the summarisation of size 2. Bars with diagonal lines from north-west to south-east are the summarisation of size 3. Bars with diagonal lines from south-west to north-east are the summarisation of size 3 together with pre-computation optimisation. 121

7.3 LUBM. Timings for database D3. Plain coloured bars are the summarisation of size 2. Bars with diagonal lines from north-west to south-east are the summarisation of size 3. Bars with diagonal lines from south-west to north-east are the summarisation of size 3 together with pre-computation optimisation. 122
7.4 DBpedia. Timings, with summarisation of size 2, with and without the pre-computation optimisation. 124
7.5 YAGO timings. Plain coloured bars are the summarisation of size 2. Bars with diagonal lines from north-west to south-east are the summarisation of size 3. Bars with diagonal lines from south-west to north-east are the summarisation of size 3 together with pre-computation optimisation. 126
7.6 Containment results 132
7.7 LUBM. Timings for database D1, with query containment optimisation, with and without pre-computation optimisation. 142
7.8 LUBM. Timings for database D2, with query containment optimisation, with and without pre-computation optimisation. 143
7.9 LUBM. Timings for database D3, with query containment optimisation, with and without pre-computation optimisation. 144
7.10 DBpedia. Timings, with query containment optimisation, with and without pre-computation optimisation. 145
7.11 YAGO. Timings with query containment optimisation, with and without pre-computation optimisation. 146
7.12 LUBM. Timings for database D1, with query containment optimisation, summarisation of size 3, with and without pre-computation optimisation. 148
7.13 LUBM. Timings for database D2, with query containment optimisation, summarisation of size 3, with and without pre-computation optimisation. 149

7.14 LUBM. Timings for database D3, with query containment optimisation, summarisation of size 3, with and without pre-computation optimisation. 150
7.15 DBpedia. Timings, with query containment optimisation, summarisation of size 2, with and without pre-computation optimisation. . . . 151
7.16 YAGO. Timings, with query containment optimisation, summarisation of size 3, with and without pre-computation optimisation. 152

List of Tables

4.1 Complexity of various SPARQL^{AR} fragments previously known but with answer costs included. 62
4.2 New complexity results for three SPARQL^{AR} fragments. 62

6.1 LUBM. Number of queries generated by the rewriting algorithm, given maximum costs of 1, 2 and 3. 95
6.2 LUBM. Number of answers returned by each query, for every maximum cost, and every dataset. 96
6.3 DBpedia. Number of queries generated by the rewriting algorithm given maximum costs of 1, 2 and 3. 101
6.4 DBpedia. Number of answers returned by each query, for every maximum cost. 101
6.5 YAGO. Number of queries generated by the rewriting algorithm, given maximum costs of 1, 2 and 3. 103
6.6 YAGO. Number of answers returned by each query, for every maximum cost. 104

7.1 LUBM. Number of queries generated by the rewriting algorithm with summarisation of size 2. 119
7.2 DBpedia. Number of queries generated by the rewriting algorithm with summarisation of size 2. 123
7.3 YAGO. Number of queries generated by the rewriting algorithm given maximum costs of 1, 2 and 3 with summarisation of size 2 and of size 3.125
7.4 Containment conditions 138

7.5 LUBM. Number of queries generated by the rewriting algorithm with query containment optimisation, given maximum costs of 1, 2 and 3. . 140
7.6 DBpedia. Number of queries generated by the rewriting algorithm given maximum costs of 1, 2 and 3. 142
7.7 YAGO. Number of queries generated by the rewriting algorithm given maximum costs of 1, 2 and 3. 144
7.8 LUBM. Number of queries generated by the rewriting algorithm with summarisation of size 3 and query containment. 147
7.9 LUBM. Number of queries generated by the rewriting algorithm with summarisation of size 2 and query containment. 148
7.10 YAGO. Number of queries generated by the rewriting algorithm with summarisation size 3 and query containment. 150

B.1 Execution time in seconds of each query, for each maximum cost and each dataset. 179
B.2 Execution time of each query, for each maximum cost and each dataset, with the pre-computation optimisation 180
B.3 Execution time of each query, for each maximum cost and each dataset, using the summarisation of size 2. 180
B.4 Execution time of each query, for each maximum cost and each dataset, using the summarisation of size 3. 181
B.5 Execution time of each query, for each maximum cost and each dataset, using the summarisation of size 3 and the pre-computation optimisation. 181
B.6 Execution time of each query, for each maximum cost and each dataset, with query containment optimisation. 182
B.7 Execution time of each query, for each maximum cost and each dataset, with query containment and pre-computation optimisation. 182
B.8 Execution time of each query, for each maximum cost and each dataset, using the summarisation of size 3 and query containment. 183
B.9 Execution time of each query, for each maximum cost and each dataset, using the pre-computation optimisation, the summarisation of size 3 and query containment. 183

B.10 Execution time in seconds of each query, for each maximum cost. . . 184
B.11 Execution time of each query, for each maximum cost with, the pre-computation optimisation. 184
B.12 Execution time of each query, for each maximum cost, using the summarisation of size 2. 184
B.13 Execution time of each query, for each maximum cost, using the summarisation of size 2 and pre-computing optimisation. 185
B.14 Execution time of each query, for each maximum cost, with query containment. 185
B.15 Execution time of each query, for each maximum cost, with query containment and pre-computation optimisation. 185
B.16 Execution time of each query, for each maximum cost, with query containment and summarisation size 2. 186
B.17 Execution time of each query, for each maximum cost, with query containment and summarisation size 2 and pre-computation optimisation. 186
B.18 Execution time in seconds of each query, for each maximum cost. . . 186
B.19 Execution time of each query, for each maximum cost, with pre-computation optimisation. 187
B.20 Execution time of each query, for each maximum cost, with summarisation of size 2. 187
B.21 Execution time of each query, for each maximum cost, with summarisation of size 3. 187
B.22 Execution time of each query, for each maximum cost, with summarisation of size 3 and pre-computation optimisation. 188
B.23 Execution time of each query, for each maximum cost, with query containment. 188
B.24 Execution time of each query, for each maximum cost, with pre-computation optimisation and query containment. 188

Chapter 1

Introduction

Linked Data[1] is a way of publishing and interlinking web data using the Resource Description Framework RDF[2]. RDF data is semi-structured and machine readable. Unlike data stored in traditional databases, semi-structured data does not necessarily need to conform to a schema, but instead contains meta-data tags that allow elements of the data to be identified and described. The Linked Data initiative aims to gather web accessible data resources to which semantic meaning can be attached, as well as to provide an interlinked structure that connects different resources. According to Meusel et al. [60], semantic annotation of web pages using techniques such as Microdata (which are machine readable HTML tags) and RDFa (which is RDF data embedded into HTML pages) has increased considerably between 2010 and 2013[3]. This in turn may lead to an increase in the number of datasets being published as Linked Data in the future.

RDF describes and connects resources by the means of properties. *Resources* are represented by Uniform Resource Identifiers (URIs)[4]; blank nodes can also be used to represent resources that are known to exist but whose URI is unavailable. *Properties* denote relationships between resources and are predefined URIs (published within

---

[1] `https://www.w3.org/standards/semanticweb/data`

[2] `https://www.w3.org/RDF/`

[3] More recent results can be found in `http://webdatacommons.org/structureddata/index.html#results-2016-1`

[4] `https://www.w3.org/Addressing/URL/uri-spec.html`

RDF vocabularies). More formally, an RDF dataset is a collection of *statements* (also called *triples*) of the form $\langle subject, predicate, object \rangle$, where the *subject* is a resource (a URI or blank node), the predicate is a property (a URI) and the object is either a resource or a literal (e.g. a string, a piece of text, a number, etc...). An RDF dataset can be viewed as a graph where the subjects and objects of each statement are nodes connected by an edge representing the predicate. Properties can themselves be regarded as resources by allowing a property URI to appear as the subject of a statement.

One of the main strengths of RDF is that it can be used as a meta-modeling language. The World Wide Web Consortium (W3C)[5] has published a vocabulary known as RDF-Schema, RDFS[6], which provides a basic ontology language intended to attach semantics to RDF datasets. The RDFS vocabulary defines *Classes*, *SubClasses*, *Properties*, *SubProperties*, *Ranges and Domains* of properties. It also gives a standard vocabulary for reification and for defining *Data types*, *Collections* (i.e. lists) and *Containers* (bags, sequences). There are also two properties widely used to connect resources to a human-readable description and a name, *rdfs:comment* and *rdfs:label*, respectively.

In order to query Linked Data suitable query languages are needed. Currently, the most prominent language used to query RDF datasets is SPARQL[7]. Such a language combines graph querying with analytic query operations such as AGGREGATE and SORT which are also common in structured query languages such as SQL. Moreover, SPARQL supports distributed querying over multiple heterogeneous RDF datasets [11]. The latest extension of SPARQL (SPARQL 1.1[8]), supports *property path queries* (also known as *regular path* queries [13]) over the RDF graph. These allow users to find paths between nodes of the RDF graph whose sequence of edge labels matches a regular expression over the alphabet of edge labels. However, SPARQL does not support notions of flexible querying, apart from the OPTIONAL

[5] `https://www.w3.org/`

[6] `https://www.w3.org/TR/rdf-schema/`

[7] `https://www.w3.org/TR/rdf-sparql-query/`

[8] `http://www.w3.org/TR/sparql11-property-paths/`

operator which allows answers to be returned even if these do not match every part of the query.

This thesis investigates an extension of a fragment of SPARQL 1.1 to include flexible querying processing, focusing particularly on its property path queries. The flexible querying techniques we investigate are those of [67] where the authors proposed two flexible querying operators, APPROX and RELAX, for regular path queries. The APPROX operator edits a property path in a query by inserting, deleting or replacing properties, which makes it an ideal candidate for adding to SPARQL 1.1 since the latter supports property path queries. The RELAX operator undertakes ontology-driven relaxation, such as replacing a property by a super-property, or a class by a super-class. Many RDF datasets are provided with an ontology or schema, and hence the RELAX operator is also a natural candidate to add to SPARQL 1.1.

Flexible querying techniques have the potential to enhance users' access to complex, heterogeneous datasets, by allowing the retrieval of non-exact answers to queries that are related in some way to the exact answers. In particular, users querying Linked Data may lack full knowledge of the structure of the data, its irregularities, and the URIs used within it. Users might not know all the properties that are needed to express a valid query because of the complexity and heterogeneity of the data. Also, the meaning of properties can be misinterpreted which may lead to invalid assumptions when formulating a query. Moreover, the structure of the data, the URIs used and their classification, may also evolve over time.

The issues described above may make it difficult for users to formulate queries that precisely express their information seeking requirements. Hence, providing users with flexible querying capabilities is desirable.

Users who may want to use these flexible capabilities need to be familiar with SPARQL 1.1 itself and how the two new operators behave, i.e., that APPROX returns answers to queries that are similar to the original query, and that RELAX returns more general answers that include the answers to the original query. In practice, we expect that a visual query interface would be available, providing users with readily understandable options from which to select their query formulation, approximation and relaxation requirements. Indeed, we have implemented a proto-

type of such an interface over the SPARQL^{AR} query evaluation system described in Chapter 6.

We illustrate in the following examples how a user could use the APPROX and RELAX operators when querying an RDF dataset.

Example 1.1. Suppose a user wishes to find events that took place in London on 15th September 1940 and poses the following query on the YAGO knowledge base[9], which is derived from multiple sources such as Wikipedia[10], WordNet[11] and GeoNames[12]:

$$(x, on, \text{“15/09/1940”}) \text{ AND } (x, in, \text{“}London\text{”})$$

In the above query, x is a variable, *on* and *in* are properties, and “15/09/1940” and “London” are literals. (The above is not a complete SPARQL query, but is sufficient to illustrate the problem we address.) This query returns no results because there are no property edges named “on” or “in” in YAGO.

Suppose the user is not sure about whether the predicates used in the query are correct, and therefore decides to apply one step of APPROX to both conjuncts of the query. The query evaluation system can now replace “on” by “happenedOnDate” and “in” by “happenedIn” (which do appear in YAGO), giving the following query:

$$(x, happenedOnDate, \text{“15/09/1940”}) \text{ AND } (x, happenedIn, \text{“}London\text{”})$$

However, this query still returns no answers, since “happenedIn” does not connect event instances directly to literals such as “London”.

As the query does not return any answers, the user guesses that one of the constants may not appear in the dataset (“15/09/1940” or “London”) and hence decides to first apply RELAX to the conjunct $(x, happenedIn, \text{“}London\text{”})$ in an attempt to retrieve additional answers. The system can now relax the conjunct to $(x, type, Event)$, using knowledge encoded in YAGO that the domain of “happened-In” is *Event*. This will return all events that occurred on 15th September 1940,

[9]`http://www.mpi-inf.mpg.de/yago-naga/yago/`
[10]`https://wikipedia.org`
[11]`https://wordnet.princeton.edu/`
[12]`http://www.geonames.org`

including those occurring in London. In this particular instance only one answer is returned which is the event "Battle of Britain", but other events could in principle have been returned. So the query exhibits better recall than the original query, but possibly low precision.

Alternatively, if the user knows that the constants in the query are contained in the dataset, instead of relaxing the second triple $(x, \mathit{happenedIn}, \text{"}\mathit{London}\text{"})$ as above, the user may decide to apply a second step of approximation to it. The system can now insert the property "label" that connects URIs to their labels, yielding the following query:

$$(x, \mathit{happenedOnDate}, \text{"15/09/1940"}) \text{ AND } (x, \mathit{happenedIn/label}, \text{"}\mathit{London}\text{"})$$

This query uses the property paths extension of SPARQL 1.1, specifically the concatenation (/) operator. This query now returns the only event that occurred on 15th September 1940 in London, that is "Battle of Britain". It exhibits both better recall than the original query and also high precision.

From the previous example we can see how our flexible querying processing can allow the user to incrementally change their query so as to retrieve the required answers. The next example similarly illustrates how a user query could be incrementally edited (by the APPROX operator alone) to retrieve the required answers.

Example 1.2. Suppose the user wishes to find the geographic coordinates of the "Battle of Waterloo" event in the YAGO dataset by posing the query

$$(\langle Battle_of_Waterloo\rangle, \mathit{happenedIn}/(\mathit{hasLongitude}|\mathit{hasLatitude}), x)$$

in which angle brackets delimit a URI. This query uses the SPARQL 1.1 property path disjunction (|) operator. In the query, the property "happenedIn" is concatenated with either "hasLongitude" or "hasLatitude", thereby finding a connection between the event and its location (in our case Waterloo), and from the location to both its coordinates.

This query does not return any answers from YAGO since YAGO does not store the geographic coordinates of Waterloo. The user may therefore choose to apply one step of APPROX to the query conjunct to generate similar queries that may

return relevant answers. The system can now apply an edit to insert "isLocatedIn" after "happenedIn" which connects the URI representing Waterloo with the URI representing Belgium. The resulting query is

$$(\langle Battle_of_Waterloo\rangle, happenedIn/isLocatedIn/ \\ (hasLongitude|hasLatitude), x).$$

This query returns 16 answers that may be relevant to the user, since YAGO does store the geographic coordinates of some (unspecified) locations in Belgium, increasing recall but with possibly low precision.

Moreover, YAGO does in fact store directly the coordinates of the "Battle of Waterloo" event. An alternative edit by the system that deletes the property "happenedIn", instead of adding "isLocatedIn", gives the query

$$(\langle Battle_of_Waterloo\rangle,\ (hasLongitude|hasLatitude), x)$$

which returns the desired answers, showing both high precision and high recall.

The next example illustrates how use of the RELAX operator can find additional answers of relevance to the user:

Example 1.3. Suppose the user is interested in finding scientists who died in 1800 during a duel, and poses the following query to the DBpedia[13] dataset which is derived from the Wikipedia resource:

$$(x, subject, Duelling_Fatalities) \text{ AND } (x, deathDate, \text{"18xx-xx-xx"}) \\ \text{AND } (x, rdf:type, Scientist)$$

The value "18xx-xx-xx" is not in a format that can match the dates in the RDF dataset (because SPARQL does not support partial matching using placeholders); hence the query returns no answers. To overcome this problem, the user may decide to apply RELAX to the second conjunct, allowing the system to replace it by $(x, rdf:type, Person)$, hence dropping the constant "18xx-xx-xx". The resulting query returns the resource "Évariste Galois"[14].

[13] `http://wiki.dbpedia.org/`.

[14] We note that the resulting query contains both the conjuncts $(x, rdf:type, Person)$ and $(x, rdf:type, Scientist)$, and the fact that the second conjunct entails the first conjunct is a coincidence.

However, the user is expecting more answers to be returned by the query. Hence she decides to try to retrieve other types of persons who died during a duel, not only ones specifically recorded as being scientists. She applies RELAX also to the third conjunct, allowing the system to replace it by $(x, rdf : type, owl : Thing)$. The resulting query returns every person who died during a duel, including all the people recorded as being scientists.

In the next section we discuss the main contributions of the thesis.

1.1 Contributions

In this thesis we address the problem of *flexible querying* of Semantic Web data. We devise an extension of a fragment of the SPARQL language (more specifically SPARQL 1.1) called SPARQL^{AR} by adding to it two operators, APPROX and RELAX, which can be applied to the conjuncts of a SPARQL 1.1 query. In contrast to the work in [67], we integrate the APPROX and RELAX operators into SPARQL 1.1 which is a pragmatic language for querying RDF datasets, and moreover we define a formal semantics for SPARQL^{AR} and propose a novel evaluation algorithm based on query rewriting. We also propose three optimisation techniques for evaluating SPARQL^{AR} and we undertake an empirical study of query execution performance, neither of which were undertaken in [67].

The APPROX operator edits a conjunct by replacing, inserting or deleting properties so that the query can return different answers compared to those returned by its original form. The RELAX operator edits a query conjunct by reference to an ontology associated with the dataset being queried, replacing it with a conjunct that is less specific. The RELAX operator, therefore, generalises the query, i.e. it allows the query to return additional answers compared to those returned by its original form.

The answers retrieved by SPARQL^{AR} queries are ordered with respect to their "distance" to the exact answers. We refer to this distance as the *cost* of the answer. To encapsulate the cost of an answer, we extend the standard SPARQL semantics to include such costs. We also specify formally the semantics of RELAX and APPROX

which extends the semantics of the standard SPARQL language.

We investigate the complexity of the SPARQL^{AR} language and some of its fragments, comparing our results with the complexity studies in [2, 65] for the standard SPARQL language. We show that by including answer costs in the SPARQL^{AR} semantics the query complexity class does not change compared with SPARQL 1.1. Similarly, we show that the RELAX and APPROX operators do not change the query complexity class of our language compared to the standard SPARQL 1.1 language.

To evaluate SPARQL^{AR} queries we present an algorithm based on a rewriting procedure inspired by the work in [42, 43], however that work considers SPARQL queries without property paths and considers relaxation only, not approximation. Our algorithm generates several SPARQL 1.1 queries that are evaluated to return the answers to a SPARQL^{AR} query up to a specified cost. Generating answers at higher costs leads to greater numbers of queries being generated by the rewriting algorithm. Hence we introduce three optimisation techniques. The first optimisation technique is based on the pre-computation and caching of the answers of sub-queries.

We also investigate a second optimisation technique based on graph summarisation: given an RDF dataset G we generate a summary of G that allows us to detect and discard queries that are unsatisfiable. In contrast to the work in [5], where the authors define a summarisation technique to make RDF datasets more comprehensible by reducing their size, here we use the summarisation for enhancing query execution performance.

Our third optimisation technique is based on query containment. We devise a sufficient condition that ensures that the answers to a SPARQL^{AR} query are also returned by another SPARQL^{AR} query. By means of this technique we can discard those queries that do not return additional answers. Although the approach is similar to optimisations proposed by Gottlob et al. in [33], further investigation of query containment in the presence of APPROX and RELAX operator was required in order to apply a query containment optimisation approach to SPARQL^{AR}.

We investigate empirically how these three optimisation techniques can improve the execution of SPARQL^{AR} queries. Moreover, we show how all three techniques

can be combined to further improve query evaluation timings. Finally, we also discuss the drawbacks of these optimisation techniques when used on their own or in combination.

1.2 Outline of the Thesis

The thesis is structured as follows: In Chapter 2 we define the syntax and semantics of the fragment of the SPARQL 1.1 language that we focus on here. In the same chapter we review the current state of the art in flexible querying techniques for relational databases, XML and RDF, and other related work on SPARQL optimisation and complexity.

In Chapter 3 we introduce the syntax and semantics of our SPARQL^{AR} language. We show how its RELAX and APPROX operators lead to the computation of non-exact answers. In Chapter 4 we present a complexity study of the SPARQL^{AR} language. We compare our complexity results with the complexity of the standard SPARQL 1.1 language.

In Chapter 5 we present the rewriting algorithm that evaluates SPARQL^{AR} queries, and prove its correctness and termination properties. We examine the efficiency of the rewriting algorithm in Chapter 6 by evaluating multiple queries against three datasets: LUBM, DBpedia and YAGO. Moreover, we describe a preliminary optimisation technique based on a caching procedure which can enhance the evaluation time of SPARQL^{AR} queries.

In Chapter 7 we introduce two further optimisation techniques. We firstly define the summarisation optimisation and discuss how it can be exploited to improve query evaluation timings. We next provide a sufficient condition for query containment for SPARQL^{AR} queries. We test how both optimisations impact on the evaluation time of SPARQL^{AR} queries. In the third and final part of Chapter 7 we combine the two optimisation techniques and undertake an experimental evaluation of their combined effectiveness.

We conclude the thesis with Chapter 8 where we summarise the contributions of the thesis and discuss future work.

Chapter 2

Background and Related Work

In this chapter we introduce the syntax and semantics of a fragment of SPARQL 1.1 which forms the basis of our expanded SPARQL^{AR} language. We also review related work on flexible querying and SPARQL extensions and optimisations.

We begin by introducing the RDF framework and by defining the syntax and semantics of a fragment of SPARQL 1.1 in Section 2.1. In Section 2.2 we undertake a review of the literature, focusing on flexible querying techniques for relational databases, XML, RDF, and on work relating to SPARQL semantics, complexity and optimisation.

2.1 Background

RDF was first proposed by the W3C to enhance web content from being machine-readable to being machine-understandable by adding meta-data tagging [53]. The potential of RDF can be exploited in many ways, such as a protocol language for APIs (application programming interface) [69], a framework for representing linguistic resources [63] and, of course, the foundation for the Semantic Web [40].

In RDF, *URI*s (uniform resource identifiers) and *literals* are the fundamental data types for constructing an RDF dataset. A URI can be used as a place-holder of an entity or concept (e.g. person, place, historical event, kind of animal etc.) which is then connected to other concepts and entities. Such URIs are often web pages that can be browsed on the internet and they need to be unique within the

whole set of Linked Data on the web. A literal can be any kind of primitive data item, such as a string, a number, a date, etc.

The latest W3C specification of RDF can be found in [22] where its full set of features are defined. In [56] Marin formalises the syntax and semantics of RDF and also RDFS (see Section 2.1.2). To define the syntax and semantics of SPARQL we use some definitions from Pérez et al. [65] in which they investigate the semantics and query complexity of SPARQL, and from Gutierrez et al. [36], in which they undertake a complexity study of querying RDF/S datasets using the notion of *tableau* from the relational database literature [1].

2.1.1 RDF-Graph

An RDF dataset is a multi-graph in which pairs of URIs are connected by edges that are also labelled by URIs, drawn from a predefined set. These labels are also known as *predicates*.

For the purpose of this thesis, we modify the definition of triples from [36,56,65] by omitting blank nodes, since their use is discouraged for Linked Data because they represent a resource without specifying its name and are identified by an ID which may not be unique in the dataset [7]. We also add weights to the edges, which are needed to formalise our flexible querying semantics. Initially, these weights are all 0:

Definition 2.1 (Sets, triples and variables)**.** Assume there are pairwise disjoint infinite sets U and L of URIs and literals, respectively. An *RDF triple* is a tuple $\langle s, p, o\rangle \in U \times U \times (U \cup L)$, where s is the subject, p the predicate and o the object of the triple. Assume also an infinite set V of *variables* that is disjoint from U and L. We abbreviate any union of the sets U, L and V by concatenating their names; for instance, $UL = U \cup L$.

Definition 2.2 (RDF-graph)**.** An *RDF-graph* G is a directed graph (N, D, E) where: N is a finite set of nodes such that $N \subset UL$; D is a finite set of predicates such that $D \subset U$; E is a finite set of labelled, weighted edges of the form $\langle\langle s, p, o\rangle, c\rangle$ such that the edge source (subject) $s \in N \cap U$, the edge target (object) $o \in N$, the edge

label $p \in D$ and the edge weight c is a non-negative number.

Example 2.1. Consider the following extract of an RDF-graph $G = (N, D, E)$ relating to films:

$$
\begin{aligned}
N = &\{\text{Scoop, Woody_Allen, Love_and_Death, ``1975'', Diane_Keaton,}\\
&\text{Play_It_Again_Sam, James_Tolkan, Jessica_Harper}\}\\
D = &\{\text{hasDirector, year, actedIn}\}.\\
E = &\{\langle\langle\text{Scoop, hasDirector, Woody_Allen}\rangle, 0\rangle,\\
&\langle\langle\text{Love_and_Death, hasDirector, Woody_Allen}\rangle, 0\rangle,\\
&\langle\langle\text{Love_and_Death, year, ``1975''}\rangle, 0\rangle,\\
&\langle\langle\text{Diane_Keaton, actedIn, Love_and_Death}\rangle, 0\rangle,\\
&\langle\langle\text{Diane_Keaton, actedIn, Play_It_Again_Sam}\rangle, 0\rangle,\\
&\langle\langle\text{James_Tolkan, actedIn, Love_and_Death}\rangle, 0\rangle,\\
&\langle\langle\text{Jessica_Harper, actedIn, Love_and_Death}\rangle, 0\rangle,\\
&\langle\langle\text{Play_It_Again_Sam, hasDirector, Woody_Allen}\rangle, 0\rangle,\\
&\langle\langle\text{Woody_Allen, actedIn, Love_and_Death}\rangle, 0\rangle\}
\end{aligned}
$$

We can see from the example above that the predicate *hasDirector* connects a film with a person, the predicate *year* connects a film with a year value, and the predicate *actedIn* connects a person with a film.

2.1.2 RDF-Schema

The RDF-Schema (RDFS) was proposed by the W3C as a semantic extension of RDF to define the vocabulary used in an RDF-graph, and to describe the relationships between resources, and the relationships between resources and properties [59]. RDFS is the foundation of ontological reasoning in the Semantic Web. Other languages that extend RDF-Schema have been defined more recently, in particular the W3C language OWL[1]. Many implementations of both RDFS and OWL have been

[1] `https://www.w3.org/TR/owl-semantics/`

developed, supporting query evaluation and reasoning over RDF data [17, 48, 76].

Similarly to an RDF-graph, an RDF-Schema (RDFS) is a multi-graph with a predefined set of edge labels. With RDFS it is possible to define the classes that resources can belong to. For example a resource "cat" can belong to the the class "animal" or "feline". With RDFS is also possible to define the relationships between classes and predicates of an RDF-graph. We next define the *ontology* relating to an RDF dataset, using a fragment of the RDFS vocabulary.

Definition 2.3 (Ontology)**.** An *ontology* K is a directed graph (N_K, E_K) where each node in N_K represents either a class or a property, and each edge in E_K is labelled with a symbol from the set $\{sc, sp, dom, range\}$. These edge labels encompass a fragment of the RDFS vocabulary, namely rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain and rdfs:range, respectively.

In an RDF-graph $G = (N, D, E)$, we assume that each node in N represents an instance or a class and each edge in E a property (even though, more generally, RDF does not distinguish between instances, classes and properties; in fact, in RDF it is possible to use a property as a node of the graph). The predicate *type*, representing the RDF vocabulary rdf:type, can be used in E to connect an instance of a class to a node representing that class. In an ontology $K = (N_K, E_K)$, each node in N_K represents a class (a "class node") or a property (a "property node"). The intersection of N and N_K is contained in the set of class nodes of N_K. D is contained in the set of property nodes of N_K.

Example 2.2. We now define an ontology $K = (N_K, E_K)$ relating to the RDF-graph in Example 2.1.

$$
\begin{aligned}
N_K = &\{\text{Person, Director, Actor, Film, actedIn, hasDirector, year, Integer}\} \\
E_K = &\{\langle \text{Director, sc, Person} \rangle, \\
&\langle \text{Actor, sc, Person} \rangle, \\
&\langle \text{actedIn, domain, Actor} \rangle, \\
&\langle \text{actedIn, range, Film} \rangle,
\end{aligned}
$$

$\langle$hasDirector, domain, Film$\rangle$,

$\langle$hasDirector, range, Director$\rangle$,

$\langle$year, domain, Film$\rangle$,

$\langle$year, range, Integer$\rangle\}$

We may connect the data with the ontology by adding edges labelled with the predicate *type* to the RDF-graph G from Example 2.1, so $G = (N \cup N_K, D \cup \{type\}, E \cup E')$, where:

$$
\begin{aligned}
E' = \{ & \langle\langle \text{Woody_Allen, type, Director}\rangle, 0\rangle, \\
& \langle\langle \text{Woody_Allen, type, Actor}\rangle, 0\rangle, \\
& \langle\langle \text{Diane_Keaton, type, Actor}\rangle, 0\rangle, \\
& \langle\langle \text{James_Tolkan, type, Actor}\rangle, 0\rangle, \\
& \langle\langle \text{Jessica_Harper, type, Actor}\rangle, 0\rangle, \\
& \langle\langle \text{Scoop, type, Film}\rangle, 0\rangle, \\
& \langle\langle \text{Love_and_Death, type, Film}\rangle, 0\rangle, \\
& \langle\langle \text{Play_it_Again_Sam, type, Film}\rangle, 0\rangle\}
\end{aligned}
$$

We note that the triples in E' do not need to appear in G but can be inferred by means of the ontology K. Moreover, the triples in E' are derived by one step of inference. By applying a second step of inference we generate the following triples:

$$
\begin{aligned}
E'' = \{ & \langle\langle \text{Woody_Allen, type, Person}\rangle, 0\rangle, \\
& \langle\langle \text{Diane_Keaton, type, Person}\rangle, 0\rangle, \\
& \langle\langle \text{James_Tolkan, type, Person}\rangle, 0\rangle, \\
& \langle\langle \text{Jessica_Harper, type, Person}\rangle, 0\rangle\}
\end{aligned}
$$

No new triples are generated by any further inference steps.

2.1.3 SPARQL Syntax

Before the SPARQL language, many proposals were put forward for querying RDF-graphs, such as RQL [49], SeRQL [10] and the first W3C proposal RDQL[2] [37]. The SPARQL language was defined by the W3C to query RDF-graphs as well as extract sub-graphs and construct new graphs from RDF-graphs [65].

Drawing from [65] we start by defining the basic element in the SPARQL language: the simple triple pattern. Such elements are constructed from variables and constants that match the constants in the RDF-graph.

Definition 2.4 (Simple triple pattern)**.** A *simple triple pattern* is a tuple $\langle x, z, y \rangle \in UV \times UV \times UVL$. Given a simple triple pattern $\langle x, z, y \rangle$, $var(\langle x, z, y \rangle)$ is the set of variables occurring in it.

As in [65] we define a regular expression for the purpose of encapsulating the SPARQL 1.1 language feature called *property paths*:

Definition 2.5 (Regular expression)**.** A *regular expression* (*regexp*) $P \in RegEx(U)$ is defined as follows:

$$P := \epsilon \mid _ \mid p \mid (P_1|P_2) \mid (P_1/P_2) \mid P^*$$

where $P_1, P_2 \in RegEx(U)$ are also regexps, ϵ represents the empty regexp, $p \in U$, $_$ is a symbol that denotes the disjunction of all URIs in U, $|$ is the disjunction of two regexps, $/$ is the concatenation of two regexps, and $*$ is the concatenation of a regexp with itself zero or more times.

We use this definition of the syntax of a regular expression, from [18], because it conforms to the W3C SPARQL 1.1 syntax, with the exception of $_$ and ϵ which are not present in that syntax[3].

[2] `https://www.w3.org/Submission/RDQL/`

[3] The W3C syntax does support the 'not' operator (!) which can be used to replace the $_$ symbol by $!p$ where p is a URI that does not exist in the dataset. The W3C syntax also supports the concatenation of a regular expression pattern with itself for a fixed number of times; for example $p\{5\}$ means that p is concatenated with itself 5 times. This feature can be used to replace ϵ by $p\{0\}$ where p can be any URI.

Definition 2.6 (Regular triple pattern)**.** A *regular triple pattern* is a tuple of the form $\langle x, y, z \rangle \in UV \times RegEx(U) \times UVL$.

We note that, in contrast to a simple triple pattern, a regular triple pattern does not allow a variable in the second position. We use the term *triple pattern* for a tuple that can be either a simple triple pattern or regular triple pattern. Our query pattern syntax is also based on that of [18]:

Definition 2.7 (Query Pattern)**.** Our fragment of the SPARQL 1.1 *query pattern* syntax Q is defined as follows:

$$Q := s \mid t \mid Q_1 \text{ AND } Q_2 \mid Q_1 \text{ UNION } Q_2 \mid Q \text{ FILTER } R$$

where s is a simple triple pattern, t is a regular triple pattern, R is a SPARQL built-in condition and Q_1, Q_2 are also query patterns. We denote by $var(Q)$ the set of all variables occurring in a query pattern Q. In the W3C SPARQL syntax, a dot (.) is used for conjunction but, for greater clarity, we use AND instead.

Definition 2.8. The overall syntax of a SPARQL 1.1 query is:

$$\text{SELECT } \overrightarrow{w} \text{ WHERE } Q$$

where $\overrightarrow{w}$ is a list of variables and Q is a query pattern.

2.1.4 SPARQL Semantics

The semantics of SPARQL 1.1 are defined by means of *mappings* which represent the answers returned by a query [65]. Definitions 2.9 to 2.11 are drawn from [65]:

Definition 2.9 (Mapping)**.** A *mapping* μ from ULV to UL is a partial function $\mu : ULV \to UL$. It is assumed that $\mu(x) = x$ for all $x \in UL$, i.e. μ maps URIs and literals to themselves. The set $var(\mu)$ is the subset of V on which μ is defined. Given a triple pattern $\langle x, z, y \rangle$ and a mapping μ such that $var(\langle x, z, y \rangle) \subseteq var(\mu)$, $\mu(\langle x, z, y \rangle)$ is the triple obtained by replacing the variables in $\langle x, z, y \rangle$ by their image according to μ.

Definition 2.10 (Compatibility and Union of Mappings)**.** Two mappings μ_1 and μ_2 are said to be *compatible* if $\forall x \in var(\mu_1) \cap var(\mu_2)$, $\mu_1(x) = \mu_2(x)$. The *union* of two mappings $\mu = \mu_1 \cup \mu_2$ can be computed only if μ_1 and μ_2 are compatible. The resulting μ is a mapping such that $var(\mu) = var(\mu_1) \cup var(\mu_2)$ and: for each x in $var(\mu_1) \cap var(\mu_2)$, we have $\mu(x) = \mu_1(x) = \mu_2(x)$; for each x in $var(\mu_1)$ but not in $var(\mu_2)$, we have $\mu(x) = \mu_1(x)$; and for each x in $var(\mu_2)$ but not in $var(\mu_1)$, we have $\mu(x) = \mu_2(x)$.

Definition 2.11 (Union and Join of Sets of Mappings)**.** Given sets of mappings M_1 and M_2, their *union* and *join*, $\cup$ and $\bowtie$, are defined as follows:

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

$M_1 \bowtie M_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1 \text{ and } \mu_2 \in M_2 \text{ with } \mu_1 \text{ and } \mu_2 \text{ compatible mappings}\}$.

We next define the semantics of the fragment of SPARQL 1.1 that we extended in our SPARQL^{AR} language, described in Chapter 3. SPARQL 1.1 supports regular path queries, to which we apply the APPROX and RELAX operators.

The semantics for regular path querying are drawn from the work on the semantics of PSPARQL by Chekol et al. [18] which was an extension of the basic SPARQL language with property paths before SPARQL 1.1 was released. The semantics of a triple pattern t, with respect to a graph $G = (N, D, E)$, denoted $[\![t]\!]_G$, is defined recursively as follows:

$$[\![\langle x, \epsilon, y\rangle]\!]_G = \{\mu \mid var(\mu) = var(\langle x, \epsilon, y\rangle) \wedge \exists c \in N \,.\, \mu(x) = \mu(y) = c\} \tag{2.1.1}$$

$$[\![\langle x, z, y\rangle]\!]_G = \{\mu \mid var(\mu) = var(\langle x, z, y\rangle) \wedge \langle \mu(\langle x, z, y\rangle), 0\rangle \in E\} \tag{2.1.2}$$

$$[\![\langle x, _, y\rangle]\!]_G = \{\mu \mid var(\mu) = var(\langle x, _, y\rangle) \wedge \exists p \in D \,.\, \langle \mu(\langle x, p, y\rangle), 0\rangle \in E\} \tag{2.1.3}$$

$$[\![\langle x, P_1|P_2, y\rangle]\!]_G = [\![\langle x, P_1, y\rangle]\!]_G \cup [\![\langle x, P_2, y\rangle]\!]_G \tag{2.1.4}$$

$$[\![\langle x, P_1/P_2, y\rangle]\!]_G = [\![\langle x, P_1, z\rangle]\!]_G \bowtie [\![\langle z, P_2, y\rangle]\!]_G \tag{2.1.5}$$

$$\llbracket \langle x, P^*, y \rangle \rrbracket_G = \llbracket \langle x, \epsilon, y \rangle \rrbracket_G \cup \llbracket \langle x, P, y \rangle \rrbracket_G \cup \bigcup_{n \geq 1} \{\mu \mid \mu \in \llbracket \langle x, P, z_1 \rangle \rrbracket_G \bowtie \bowtie \llbracket \langle z_1, P, z_2 \rangle \rrbracket_G \bowtie \cdots \bowtie \llbracket \langle z_n, P, y \rangle \rrbracket_G\} \tag{2.1.6}$$

where P, P_1, P_2 are regular expressions, x and z are in UV, y is in ULV, and $z, z_1, \ldots, z_n$ are fresh variables. Equation 2.1.2 defines the semantics of a simple triple pattern (Definition 2.4). Equations 2.1.1-2.1.6, having $z \in U$ in 2.1.2, define the semantics of regular triple pattern (Definitions 2.5 and 2.6).

In SPARQL 1.1 there is also the FILTER operator (as shown in Definition 2.7) which reduces the number of answers by constraining the variables of the query by means of a condition in R. A mapping *satisfies a condition* R, denoted $\mu \models R$, according to the following [65]:

R is $x = a$: $\mu \models R$ if $x \in var(\mu)$, $a \in LU$ and $\mu(x) = a$;

R is $x = y$: $\mu \models R$ if $x, y \in var(\mu)$ and $\mu(x) = \mu(y)$;

R is $isURI(x)$: $\mu \models R$ if $x \in var(\mu)$ and $\mu(x) \in U$;

R is $isLiteral(x)$: $\mu \models R$ if $x \in var(\mu)$ and $\mu(x) \in L$;

R is $R_1 \wedge R_2$: $\mu \models R$ if $\mu \models R_1$ and $\mu \models R_2$;

R is $R_1 \vee R_2$: $\mu \models R$ if $\mu \models R_1$ or $\mu \models R_2$;

R is $\neg R_1$: $\mu \models R$ if it is not the case that $\mu \models R_1$;

The semantics of the AND, UNION and FILTER operators are as follows [65]; corresponding to the third, fourth and fifth forms of a query pattern in Definition 2.7:

$$\llbracket Q_1 \text{ AND } Q_2 \rrbracket_G = \llbracket Q_1 \rrbracket_G \bowtie \llbracket Q_2 \rrbracket_G \tag{2.1.7}$$

$$\llbracket Q_1 \text{ UNION } Q_2 \rrbracket_G = \llbracket Q_1 \rrbracket_G \cup \llbracket Q_2 \rrbracket_G \tag{2.1.8}$$

$$\llbracket Q \text{ FILTER } R \rrbracket_G = \{\mu \in \llbracket Q \rrbracket_G \mid \mu \models R\} \tag{2.1.9}$$

(The first and second forms of a query pattern are simple patterns and regular patterns, whose semantics were defined above, 2.1.1-2.1.6.) The semantics of an

overall SPARQL 1.1 query as defined in Definition 2.8 is as follows, where the projection operator $\pi_{\overrightarrow{w}}$ selects only the subsets of the mappings relating to the variables in $\overrightarrow{w}$:

$$[\![\text{SELECT } \overrightarrow{w} \text{ WHERE } Q]\!]_G = \pi_{\overrightarrow{w}}([\![Q]\!]_G) \tag{2.1.10}$$

2.2 Related Work

Currently there are two broad branches of research in the area of the Semantic Web [38]. The first one deals mostly with data management problems related to the Semantic Web, such as data serialization, query optimization, data integration, query performance, scalability and ontological reasoning. The second one deals with semantic interoperability of data on the web, i.e. linking the data and enriching it with meta-data (possibly in RDFS format), and with data quality.

The work in this thesis contributes to both these branches. We add the RELAX and APPROX operators to SPARQL 1.1 to help with semantic interoperability over RDF data that may have arisen from the integration of several heterogeneous datasets (such as YAGO), and in particular with users' querying of such data (using the RELAX operator to aid in the integration of heterogeneous datasets was also described in [44]). We also present query optimisation techniques that are useful not only in the context of flexible querying but also for graph querying in general. In particular, we will see in Chapter 7 that by applying graph summarisation techniques we are able to identify unsatisfiable queries without executing them over the dataset.

Flexible querying has been considered for many query languages and we review these works in subsections 2.2.1–2.2.4.

2.2.1 Flexible Relational Querying

An early extension of SQL with flexible query capabilities was investigated by Bosc et al. in [9]. The language, called SQLf, allows both boolean and fuzzy predicates in the WHERE clause. A fuzzy predicate can be considered as a function $P : X \to [0, 1]$, where X is the domain of the function, and the range is a value between 0 and 1.

Another flexible querying technique based on fuzzy sets is described in [8]. The authors present an extension to SQL (Soft-SQL) which permits so-called soft conditions. Such conditions tolerate degrees of under-satisfaction of a query by exploiting the flexibility offered by fuzzy set theory.

In [46] Ioannidis et al. propose a formula that measures the amount of error in the answers returned by an approximate query answering system. They also describe an approximation technique for SQL querying based on histograms. A histogram is a table that contains statistics about the attribute values in a relation. To retrieve approximate answers, a query Q is translated into a query Q' that queries histograms to retrieve the approximate answers.

Flexible querying approaches for SQL are also discussed in [71], where the authors describe a system that enables a user to issue an SQL aggregation query, see results as they are produced, and adjust the processing as the query runs.

An approximation technique for conjunctive queries on probabilistic databases is investigated in [27]. The authors associate the answers of a conjunctive query to a propositional formula in disjunctive normal form that represents the tuples of the dataset. Approximated answers are then retrieved by editing this formula. The formula can be edited by removing clauses or adding literals to a clause for a lower bound, and by adding clauses or removing literals from a clause for an upper bound.

In [32] Gaasterland proposes a query relaxation technique for relational queries where multiple queries are generated from a given query based on a graph of taxonomic relationships between predicates and constants. The user when posing a query may choose which relaxed form of the query to execute. Gaasterland also allows the user to apply constraints to the answers, i.e. the technique will not generate relaxed queries that violate the constraints posed by the user.

In [39], Heer et al. also investigate query relaxation for SQL queries. Their approach is to retrieve additional answers by widening the selection range of the query e.g., instead of returning tuples for a specified "day" value, the relaxation returns tuples for a range of days. They apply this approach to hierarchically organised data, such as geographic regions of neighbourhoods, cities, counties, and states.

Our APPROX and RELAX operators are not based on fuzzy logic as in [8,9,27]

but return potentially similar answers based on query editing and ontology relaxation. In contrast to [46] our flexible querying approach does not use data statistics to speed up query answering, but instead uses caching, graph summarisation and query containment.

2.2.2 Flexible XML Querying

A relaxation technique for the XPath query language was proposed in [28], where queries are relaxed by applying generalisation rules captured by edit operations. Since XPath queries are tree-like structures, the edit operations refer to the nodes of a query tree and are the following: labelled edge relaxation (replaces the labelled edge with one that can match any path), node deletion (a node is deleted and its children become the children of its parent), node relaxation (which replaces a node with a wildcard that matches any constant), node promotion (the node becomes a sibling of the parent node), edge cloning (the sub-tree is replicated and is connected to the destination node via an edge with the same label). Each of these edit operations has a cost. The answers are ranked in terms of a global relaxation cost that depends on the edit operations applied.

In [26] Fazinga et al. discuss an approximation technique for XPath queries which include the negation operator. In their approach, weights are added to the predicates of the XPath query which are then used to compute the satisfaction score of the answers. The evaluation algorithm proposed for executing such queries generates the answers incrementally by constructing an *operator tree* based on the XPath query. Each node in the operator tree represents an approximated query operator. The operator tree is then evaluated incrementally to return the first top-k answers.

In [55] Mandreoli et al. discuss a query approximation technique for heterogeneous XML documents. Given a collection of XML documents, a *matching schema* is built through a combination of the following: comparing the structures of the documents, the semantics of the terms in the document (by using the WordNet vocabulary[4]), and adjacency similarity, i.e. the similarity of two elements propa-

[4] `https://wordnet.princeton.edu/`

gates to their respective adjacent nodes. To compute the approximated answers, an XQuery query is rewritten based on the matching schema.

In [4] a fuzzy approach is proposed to extend the XPath query language in order to assign priorities to queries and to rank query answers. These techniques are based on fuzzy extensions of the Boolean operators. A similar approach can be found in [14], where the authors compare the structure of an XPath query and the structure of the XML being queried. To achieve this, they edit the tree-like structure of the query by replacing, inserting, deleting and permuting the nodes of the tree. Moreover, they compare the semantics of the trees by using the Wordnet vocabulary. In particular, they calculate the similarity of two nodes of the tree by using hypernyms in Wordnet.

Our approach differs from [4, 14] as we do not use fuzzy logic to retrieve additional answers. Although our query evaluation technique for query relaxation and approximation is based on a rewriting procedure similar to [26,28,55], our approach differs from these as we perform both query approximation and query relaxation. Moreover, in contrast to [26,55], we do not need to construct additional data structures for evaluating SPARQL^{AR} queries.

2.2.3 Flexible SPARQL Querying

There have been several previous proposals for applying flexible querying to the Semantic Web, mainly employing similarity measures to retrieve additional answers of possible relevance. For example, in [41] matching functions are used for constants such as strings and numbers, while in [52] an extension of SPARQL is developed, called iSPARQL, which uses three different matching functions to compute string similarity. In [23], the structure of the RDF data is exploited and a similarity measurement technique is proposed which matches paths in the RDF graph with respect to the query. Ontology-driven similarity measures are proposed in [42,43,68] which use the RDFS ontology to retrieve extra answers and assign a score to them.

In [25] methods for relaxing SPARQL-like triple pattern queries are presented. These query relaxations are produced by means of statistical language models for structured RDF data and queries. The query processing algorithms merge the results

of different relaxations into a unified results list.

Our own work builds on that reported in [44, 45, 67]. In [44], Hurtado et al. introduce the RELAX operator for RDF conjunctive queries. They investigate the complexity of evaluating such queries and show how relaxed queries can be used as a tool for data integration. In [67] the authors show how a conjunctive regular path query language can be effectively extended with approximation and relaxation techniques, using similar notions of approximation and relaxation as we use here. The work in [67] combines the concept of approximation from [45] as well as relaxation from [44].

Several prototypes that support flexible querying over RDF data have been developed [52, 66, 73]. In [52] flexible querying based on similarity, such as Levenshtein's distance [54], has been tested on different scenarios. In [66], approximation and relaxation techniques for conjunctive regular path queries have been developed in a prototype system called ApproxRelax for querying heterogeneous data arising from lifelong learners' educational and work experiences. In [73], the authors describe an implementation of a flexible querying evaluator for conjunctive regular path queries, extending the work in [66].

Instead of applying similarity matching algorithms [41,52] we extend, for the first time, a fragment of SPARQL 1.1 with APPROX and RELAX operators. In contrast to [25, 42–45, 67, 68], our focus is combining flexible query processing with the SPARQL 1.1 language, supporting both ontology-based relaxation and approximation. We undertake a complexity study and a detailed analysis of our SPARQL^{AR} language, also providing a performance study over three datasets: LUBM, DBpedia and YAGO. We also propose three optimisation techniques based on precomputation, data summarisation and query containment, and test their impact on the performance timings. In contrast to the work in [73], our evaluation approach is based on query rewriting using a standard RDF querying system as opposed to an approach based on translating regular expressions into finite automata. Early versions of our work are reported in [12, 31].

2.2.4 SPARQL Semantics, Complexity and Optimisation

Much research has have been undertaken regarding the semantics, evaluation and optimisation of the SPARQL language. In [72] two different semantics have been considered, namely the set semantics and the bag semantics (in the latter, two equivalent mappings can appear in the result set more than once). A set of algebraic equivalences used to optimise and minimise queries has also been defined. In this thesis, we adopt the semantics of SPARQL with set semantics since the answers returned by our SPARQL^{AR} query language are required to be unique.

The work in [6] shows formally that the W3C recommendation of SPARQL has the same expressive power as the relational algebra under bag semantics. This shows that the Semantic Web and relational databases are closely related.

The work in [65] investigates the semantics and complexity of SPARQL. It shows that the evaluation of SPARQL queries with graph pattern expressions, constructed using AND and FILTER operators, can be accomplished in $O(|P| \cdot |D|)$ time, where $|P|$ is the size of the graph pattern and $|D|$ is the size of the data. Adding the UNION operator, the complexity becomes NP-complete. We use and extend these results by investigating the complexity of SPARQL extended with APPROX and RELAX in Chapter 4. Evaluation of SPARQL queries that involve the AND, FILTER, and OPTIONAL operators is PSPACE-complete [72]. A central result of [72] is that the evaluation problem for every SPARQL fragment involving OPTIONAL is PSPACE-complete.

Before SPARQL 1.1 was proposed, PSPARQL extended SPARQL with regular path queries [2]. PSPARQL query containment under RDFS is studied by Chekol et al. [19, 20], who show that it can be accomplished in 2EXPTIME.

Most of the optimisation techniques proposed for SPARQL consider a subset of the language, such as queries with a single conjunct or queries where only the AND and FILTER operators can be used. For example, in [74] the authors describe a semantic query optimisation approach using their own semantic framework to capture RDFS. For query minimisation, they use the so-called back-chase algorithm that guarantees to find all minimal equivalent sub-queries. AND-only SPARQL queries, i.e. queries that allow only the AND operator and can be translated into

CQ (conjunctive queries), can be optimised by making use of Datalog rules [72]. Through these rules, the CQ is rewritten to obtain a minimal equivalent CQ. Finally, the minimal CQ is translated back into SPARQL to be executed.

In contrast to the above work, in this thesis we extend with the APPROX and RELAX operators the fragment of SPARQL 1.1 that includes the AND, FILTER and UNION operators as well as the property path feature. We study the complexity of this new language and devise a query evaluation algorithm. We propose an optimisation technique that pre-computes and caches the answers of sub-queries which can then be reused for improving query evaluation time. We also propose a second optimisation technique that enhances query performance by exploiting a summarisation of the RDF-graph. Our third optimisation technique uses the query containment property to rewrite queries in a more efficient way. Related work on graph summarisation and query containment is described in Chapter 7 and our techniques are compared with it.

2.3 Discussion

In this chapter we introduced the syntax and semantics of a fragment of SPARQL 1.1 which forms the basis of our expanded SPARQL^{AR} language. We provided an overview of relevant literature, mainly focusing on various forms of flexible querying and on the semantics, complexity and optimisations of the SPARQL query language.

In the next chapter we present our SPARQL^{AR} language, which extends the syntax and semantics of the SPARQL 1.1 fragment introduced here to include the APPROX and RELAX operators. We formally define the semantics of the extended language and illustrate usage of APPROX and RELAX through several examples.

Chapter 3

Syntax and Semantics of SPARQL^{AR}

In this thesis we extend the fragment of the SPARQL 1.1 language defined in Section 2.1 by including two more operators, APPROX and RELAX. In this chapter we define the syntax and semantics of the resulting language, which we call SPARQL^{AR}.

In Section 3.1 we define the syntax of SPARQL^{AR}. In Section 3.2 we define the semantics of SPARQL^{AR} by extending the semantics of SPARQL 1.1 defined in [18, 65]. We add a cost measure to the semantics so that the answers can be ranked, and adapt the semantics of approximation and relaxation from [67] to fit the SPARQL 1.1 setting.

3.1 Syntax

The following definition of a query pattern includes also our query approximation and relaxation operators APPROX and RELAX.

Definition 3.1 (Query Pattern). A SPARQL^{AR} *query pattern* Q is defined as follows:

$$Q := s \mid t \mid Q_1 \text{ AND } Q_2 \mid Q_1 \text{ UNION } Q_2 \mid Q \text{ FILTER } R \mid$$
$$\text{RELAX}(UV \times RegEx(U) \times UVL) \mid \text{APPROX}(UV \times RegEx(U) \times UVL)$$

where s is a simple triple pattern, t is a regular triple pattern, R is a SPARQL

built-in condition and Q_1, Q_2 are also query patterns. We denote by $var(Q)$ the set of all variables occurring in a query pattern Q.

A SPARQL^{AR} query has the form SELECT $\overrightarrow{w}$ WHERE Q, with $\overrightarrow{w} \subseteq var(Q)$. We may omit here the keyword WHERE for simplicity and write SELECT $\overrightarrow{w}\, Q$. Given $Q' = $ SELECT$\overrightarrow{w}\, Q$, the *head* of Q', $head(Q')$, is $\overrightarrow{w}$ if $\overrightarrow{w} \neq \emptyset$ and is $var(Q)$ otherwise.

3.2 Semantics

We extend the SPARQL semantics given in Section 2.1.4 in order to handle the weight/cost of edges in an RDF-Graph and the cost of applying the approximation and relaxation operators. These costs are used to rank the answers, with exact answers (of cost 0) being returned first, followed by answers with increasing costs.

We also extend the notion of SPARQL query evaluation from returning a set of mappings to returning a set of pairs of the form $\langle \mu, c \rangle$, where μ is a mapping and c is a non-negative integer that indicates the cost of the answers arising from this mapping.

We redefine the *union* and *join* of two sets of flexible query evaluation results, M_1 and M_2, which now include the answer costs (cf: Definition 2.11):

> $M_1 \cup M_2 = \{\langle \mu, c \rangle \mid \langle \mu, c_1 \rangle \in M_1$ or $\langle \mu, c_2 \rangle \in M_2$ with $c = c_1$ if $\nexists c_2.\langle \mu, c_2 \rangle \in M_2$, $c = c_2$ if $\nexists c_1.\langle \mu, c_1 \rangle \in M_1$, and $c = min(c_1, c_2)$ otherwise$\}$.
>
> $M_1 \bowtie M_2 = \{\langle \mu_1 \cup \mu_2, c_1 + c_2 \rangle \mid \langle \mu_1, c_1 \rangle \in M_1$ and $\langle \mu_2, c_2 \rangle \in M_2$ with μ_1 and μ_2 compatible mappings$\}$.

Recall from Definition 2.2 that an RDF-graph G comprises a set of nodes N, and a set of labelled weighted edges of the form $\langle \langle s, p, o \rangle, c \rangle$. We redefine the semantics of a triple pattern t with respect to a graph G, denoted $[\![t]\!]_G$, and the four operators from Section 2.1.4, i.e. AND, UNION, FILTER and SELECT, by adding the answer costs:

$$[\![\langle x, \epsilon, y \rangle]\!]_G = \{\langle \mu, 0 \rangle \mid var(\mu) = var(\langle x, \epsilon, y \rangle) \wedge$$

$$
\begin{aligned}
&\exists u \in N \,.\, \mu(x) = \mu(y) = u\} \\
[\![\langle x, z, y\rangle]\!]_G &= \{\langle \mu, c\rangle \mid var(\mu) = var(\langle x, z, y\rangle) \wedge \langle \mu(\langle x, z, y\rangle), c\rangle \in E\} \\
[\![\langle x, _, y\rangle]\!]_G &= \{\mu \mid var(\mu) = var(\langle x, _, y\rangle) \wedge \\
&\exists p \in D \,.\, \langle \mu(\langle x, p, y\rangle), c\rangle \in E\} \\
[\![\langle x, P_1|P_2, y\rangle]\!]_G &= [\![\langle x, P_1, y\rangle]\!]_G \cup [\![\langle x, P_2, y\rangle]\!]_G \\
[\![\langle x, P_1/P_2, y\rangle]\!]_G &= [\![\langle x, P_1, z\rangle]\!]_G \bowtie [\![\langle z, P_2, y\rangle]\!]_G \\
[\![\langle x, P^*, y\rangle]\!]_G &= [\![\langle x, \epsilon, y\rangle]\!]_G \cup [\![\langle x, P, y\rangle]\!]_G \cup \\
&\bigcup_{n \geq 1} \{\langle \mu, c\rangle \mid \langle \mu, c\rangle \in [\![\langle x, P, z_1\rangle]\!]_G \bowtie \\
&\bowtie [\![\langle z_1, P, z_2\rangle]\!]_G \bowtie \cdots \bowtie [\![\langle z_n, P, y\rangle]\!]_G\} \\
[\![Q_1 \text{ AND } Q_2]\!]_G &= [\![Q_1]\!]_G \bowtie [\![Q_2]\!]_G \\
[\![Q_1 \text{ UNION } Q_2]\!]_G &= [\![Q_1]\!]_G \cup [\![Q_2]\!]_G \\
[\![Q \text{ FILTER } R]\!]_G &= \{\langle \mu, c\rangle \in [\![Q]\!]_G \mid \mu \models R\} \\
[\![\text{SELECT } \overrightarrow{w}\ Q]\!]_G &= \pi_{\overrightarrow{w}}([\![Q]\!]_G)
\end{aligned}
$$

where P, P_1, P_2 are regular expression patterns, x, y, z are in ULV, and $z, z_1, \ldots, z_n$ are fresh variables.

We note that the semantics of $[\![\langle x, P_1/P_2, y\rangle]\!]$ and $[\![\langle x, P_1|P_2, y\rangle]\!]$ as well as the semantics of the four operators AND, FILTER, UNION and SELECT are left unchanged with respect to SPARQL 1.1.

3.2.1 Adding Approximation

For the APPROX operator, we apply edit operations which transform a regular expression pattern P into a new expression pattern P'. Specifically, we apply the edit operations *deletion*, *insertion* and *substitution*, defined as follows (other possible edit operations are *transposition* and *inversion*, which we leave as future work):

$A/p/B \leadsto (A/\epsilon/B)$	deletion
$A/p/B \leadsto (A/_/B)$	substitution
$A/p/B \leadsto (A/_/p/B)$	left insertion

$$A/p/B \rightsquigarrow (A/p/_/B) \qquad \text{right insertion}$$

Here, A and B denote any regular expression (including the empty expression) and the symbol $_$ represents every URI from U — so the edit operations allow us to insert any URI and substitute a URI by any other. The application of an edit operation op has a non-negative cost c_{op} associated with it.

These rules can be applied to a URI p in order to approximate it to a regular expression P. We write $p \rightsquigarrow^* P$ if a sequence of edit operations can be applied to p to derive P. The edit cost of deriving P from p, denoted $ecost(p, P)$, is the minimum cost of applying such a sequence of edit operations.

The semantics of the APPROX operator in SPARQL^{AR} is as follows:

$$
\begin{aligned}
[\![\text{APPROX}(x, p, y)]\!]_G &= [\![\langle x, p, y\rangle]\!]_G \cup \bigcup \{\langle \mu, c + ecost(p, P)\rangle \mid \\
&\qquad p \rightsquigarrow^* P \wedge \langle \mu, c\rangle \in [\![\langle x, P, y\rangle]\!]_G\} \\
[\![\text{APPROX}(x, P_1|P_2, y)]\!]_G &= [\![\text{APPROX}(x, P_1, y)]\!]_G \cup [\![\text{APPROX}(x, P_2, y)]\!]_G \\
[\![\text{APPROX}(x, P_1/P_2, y)]\!]_G &= [\![\text{APPROX}(x, P_1, z)]\!]_G \bowtie [\![\text{APPROX}(z, P_2, y)]\!]_G \\
[\![\text{APPROX}(x, P^*, y)]\!]_G &= [\![\langle x, \epsilon, y\rangle]\!]_G \cup [\![\text{APPROX}(x, P, y)]\!]_G \cup \\
&\quad \bigcup_{n \geq 1} \{\langle \mu, c\rangle \mid \langle \mu, c\rangle \in [\![\text{APPROX}(x, P, z_1)]\!]_G \bowtie \\
&\quad \bowtie [\![\text{APPROX}(z_1, P, z_2)]\!]_G \bowtie \cdots \bowtie \\
&\quad \bowtie [\![\text{APPROX}(z_n, P, y)]\!]_G\}
\end{aligned}
$$

where P, P_1, P_2 are regular expression patterns, x, y are in ULV, p is in U, and z, z_1, ..., z_n are fresh variables.

Example 3.1. Suppose that the user is looking for all discoveries made between 1700 and 1800 AD, and queries the YAGO dataset as follows:

```
SELECT ?p ?z ?y WHERE{
?p discovered ?x . ?x discoveredOnDate ?y .
APPROX(?x label ?z) .
FILTER(?y >= 1700/1/1 and ?y <= 1800/1/1)}
```

The above query is expressed in the concrete SPARQL^{AR} syntax. Terms of the form $?x$ are variables, *discovered*, *discoveredOnDate* and *label* are properties, and "1800/1/1" and "1700/1/1" are literals. Evaluating the exact form of the query (i.e. without APPROX applied to the third triple pattern) will return only the names of the discoveries z made by persons p on date y and no other information. By approximating the third triple pattern, it is possible to substitute the predicate "label" by "_". The query will then return more information concerning each discovery, such as its preferred name ($hasPreferredName$) and the Wikipedia abstract ($hasWikipediaAbstract$).

As another example, consider the following query, which is intended to return every German politician:

```
SELECT * WHERE{
    APPROX(?x isPoliticianOf ?y) .
    ?x wasBornIn/isLocatedIn* <Germany>
}
```

The exact form of this query returns no answers since the predicate "$isPoliticianOf$" only connects persons to states of the United States in YAGO. If the first triple pattern is approximated by substituting the predicate "$isPoliticianOf$" with "_", then the query will return the expected results, matching the correct predicate to retrieve the desired answers, which is "*holdsPoliticalPosition*". It will also retrieve all the other persons that are born in Germany (thus showing improved recall, but lower precision).

3.2.2 Adding Relaxation

Our RELAX operator is based on that in [44, 67] and relies on a fragment of the RDFS entailment rules known as ρDF [61]. An RDFS graph K_1 *entails* an RDFS graph K_2, denoted $K_1 \models_{RDFS} K_2$, if K_2 can be derived by applying the rules in Figure 3.1 iteratively to K_1. For the fragment of RDFS that we consider, $K_1 \models_{RDFS} K_2$ if and only if $K_2 \subseteq cl(K_1)$, with $cl(K_1)$ being the closure of the RDFS graph K_1 under these rules. Notice that if K_1 is finite then $cl(K_1)$ is also finite. The work

$$\text{Subproperty } (1)\ \frac{(a,sp,b)(b,sp,c)}{(a,sp,c)}\ (2)\frac{(a,sp,b)(x,a,y)}{(x,b,y)}$$
$$\text{Subclass } (3)\ \frac{(a,sc,b)(b,sc,c)}{(a,sc,c)}\ (4)\frac{(a,sc,b)(x,type,a)}{(x,type,b)}$$
$$\text{Typing } (5)\ \frac{(a,dom,c)(x,a,y)}{(x,type,c)}\ (6)\ \frac{(a,range,d)(x,a,y)}{(y,type,d)}$$

Figure 3.1: RDFS entailment rules [62]

$$(e1)\ \frac{(b,dom,c)(a,sp,b)}{(a,dom,c)}\ (e2)\frac{(b,range,c)(a,sp,b)}{(a,range,c)}$$
$$(e3)\ \frac{(a,dom,b)(b,sc,c)}{(a,dom,c)}\ (e4)\frac{(a,range,b)(b,sc,c)}{(a,range,c)}$$

Figure 3.2: Additional rules for extended reduction of an RDFS ontology [67]

in [44, 67] considers one integrated graph, comprising both the RDF-graph and the ontology. In this thesis we keep these graphs separate, as defined in Section 2.1.

Applying a rule in Figure 3.1 means adding a triple that is entailed by the rule to G or K. Specifically, if there are two triples t, t' that match the antecedent of a rule, then it is possible to insert the triple implied by the consequent of the rule. For example, the triple pattern $\langle x, startsExistingOnDate, y\rangle$ can be entailed from $\langle x, wasBornOnDate, y\rangle$ and $\langle wasBornOnDate, sp, startsExistingOnDate\rangle$ by applying rule 2.

The ontology K needs to be acyclic in order for relaxed queries to have unambiguous costs. The *extended reduction* of an ontology K, denoted by $extRed(K)$, is given by $cl(K) - D$, where D is defined as follows: D is the set of triples in $cl(K)$ that can be derived using rules (1) or (3) in Figure 3.1, or rules (e1), (e2), (e3) or (e4) in Figure 3.2. We note that, because cl(K) is closed with respect to the edge labels sp and sc, and also that the subgraphs induced by each of sp and sc are acyclic, the set D is uniquely defined.

Henceforth, we assume that $K = extRed(K)$, which allows *direct* relaxations to be applied to queries (see below), corresponding to the 'smallest' relaxation steps. This is necessary for associating an unambiguous cost to queries, so that query answers can then be returned to users incrementally in order of increasing cost.

If we did not use the extended reduction of the ontology K, then the relaxation steps applied would not necessarily be the "smallest" (see [44] for a detailed discus-

sion). For example, consider the following ontology $K = \{(b, dom, c), (a, sp, b), (a, dom, c)\}$, where $K \neq extRed(K)$. If we relax the triple pattern (x, a, y) with respect to K, then as a first step we could apply rule 5 to generate $(x, type, c)$. However, the same triple pattern can be generated with 2 steps of relaxation by applying rule 2 first and then rule 5 of Figure 3.1.

Following the terminology of [44], a triple pattern $\langle x, p, y\rangle$ *directly relaxes* to a triple pattern $\langle x', p', y'\rangle$ with respect to an ontology $K = extRed(K)$, denoted $\langle x, p, y\rangle \prec_i \langle x', p', y'\rangle$, if $vars(\langle x, p, y\rangle) = vars(\langle x', p', y'\rangle)$ and $\langle x', p', y'\rangle$ is derived from $\langle x, p, y\rangle$ by applying rule i from Figure 3.1. The cost of applying rule i is an integer $c_i > 0$.

A triple pattern $\langle x, p, y\rangle$ *relaxes to* a triple pattern $\langle x', p', y'\rangle$, denoted $\langle x, p, y\rangle \leq_K \langle x', p', y'\rangle$, if starting from $\langle x, p, y\rangle$ there is a sequence of direct relaxations that derives $\langle x', p', y'\rangle$. The relaxation cost of deriving $\langle x, p, y\rangle$ from $\langle x', p', y'\rangle$, denoted $rcost(\langle x, p, y\rangle, \langle x', p', y'\rangle)$, is the minimum cost of applying such a sequence of direct relaxations.

The semantics of the RELAX operator in SPARQL^{AR} are as follows:

$$\begin{aligned}
[\![\text{RELAX}(x, p, y)]\!]_{G,K} &= [\![\langle x, p, y\rangle]\!]_G \cup \{\langle \mu, c + rcost(\langle x, p, y\rangle, \langle x', p', y'\rangle)\rangle \mid \\
&\qquad \langle x, p, y\rangle \leq_K \langle x', p', y'\rangle \wedge \langle \mu, c\rangle \in [\![\langle x', p', y'\rangle]\!]_G\} \\
[\![\text{RELAX}(x, P_1|P_2, y)]\!]_{G,K} &= [\![\text{RELAX}(x, P_1, y)]\!]_{G,K} \cup [\![\text{RELAX}(x, P_2, y)]\!]_{G,K} \\
[\![\text{RELAX}(x, P_1/P_2, y)]\!]_{G,K} &= [\![\text{RELAX}(x, P_1, z)]\!]_{G,K} \bowtie [\![\text{RELAX}(z, P_2, y)]\!]_{G,K} \\
[\![\text{RELAX}(x, P^*, y)]\!]_{G,K} &= [\![\langle x, \epsilon, y\rangle]\!]_G \cup [\![\text{RELAX}(x, P, y)]\!]_{G,K} \cup \\
&\qquad \bigcup_{n \geq 1} \{\langle \mu, c\rangle \mid \langle \mu, c\rangle \in [\![\text{RELAX}(x, P, z_1)]\!]_{G,K} \bowtie \\
&\qquad [\![\text{RELAX}(z_1, P, z_2)]\!]_{G,K} \bowtie \cdots \bowtie [\![\text{RELAX}(z_n, P, y)]\!]_{G,K}\}
\end{aligned}$$

where P, P_1, P_2 are regular expression patterns, x, x', y, y' are in ULV, p, p' are in U, and z, z_1, ..., z_n are fresh variables.

Example 3.2. Consider the following portion $K = (N_K, E_K)$ of the YAGO ontology, where N_K is

$$\{hasFamilyName, hasGivenName, label, actedIn, \\ Actor, English_politicians, politician\},$$

and E_K is

$$\{(hasFamilyName, sp, label),\ (hasGivenName, sp, label),$$
$$(actedIn, domain, actor),\ (English_politicians, sc, politician)\}$$

Suppose the user is looking for the family names of all the actors who played in the film "Tea with Mussolini" and poses this query:

```
SELECT * WHERE {
?x actedIn <Tea_with_Mussolini> .
RELAX(?x hasFamilyName ?z) }
```

The exact form of the above query returns 4 answers. However, some actors have only a single name (for example Cher), or have their full name recorded using the "label" property directly. By applying relaxation to the second triple pattern using rule (2), we can replace the predicate "hasFamilyName" by "label". This causes the relaxed query to return the given names of actors in the film, recorded using the property "hasGivenName" since "hasGivenName" is a sub-property of "label" (hence returning Cher), as well as actors' full names recorded using the property "label": a total of 255 results.

As another example, suppose the user poses the following query:

```
SELECT * WHERE {
RELAX(?x type <English_politicians>) .
?x wasBornIn/isLocatedIn* <England>}
```

whose exact form returns every English politician born in England. By applying relaxation to the first triple pattern using rule (4), it is possible to replace the class *English_politicians* by *politicians*. This relaxed query will return every politician who was born in England, giving possibly additional answers of relevance to the user.

Observation 3.1. By the semantics of RELAX and APPROX, we observe that given a triple pattern $\langle x, P, y\rangle$, the following hold for every RDF-graph G and ontology K:

$$[\![\langle x, P, y\rangle]\!]_{G,K} \subseteq [\![\text{APPROX}(x, P, y)]\!]_G$$

$$[\![\langle x, P, y\rangle]\!]_{G,K} \subseteq [\![\text{RELAX}(x, P, y)]\!]_{G,K}$$

3.3 Query Answers up to a Maximum Cost

Queries with the APPROX and RELAX operators might return a very large number of answers. From a user perspective this is not practical. Hence, we use an operator $CostProj(M, c)$ to select mappings with a cost less than or equal to a given value c from a set M of mapping/cost pairs $\langle \mu, cost\rangle$.

Given a SPARQL^{AR} query Q, RDF graph G and ontology K, the semantics of Q limited to mappings with costs up to c is denoted by $[\![Q]\!]_{G,K,c}$, defined as follows:

$$[\![Q]\!]_{G,K,c} = \text{CostProj}([\![Q]\!]_{G,K}, c) = \{\langle \mu, i\rangle \mid \langle \mu, i\rangle \in [\![Q]\!]_{G,K} \wedge i \leq c\}$$

Using the above definition of CostProj, we define the semantics of query approximation *up to a cost* as follows:

$$[\![\text{APPROX}(x, p, y)]\!]_{G,K,c} = \text{CostProj}([\![\text{APPROX}(x, p, y)]\!]_{G,K}, c)$$

Similarly we define the semantics of query relaxation *up to a cost* as follows:

$$[\![\text{RELAX}(x, p, y)]\!]_{G,K,c} = \text{CostProj}([\![\text{RELAX}(x, p, y)]\!]_{G,K}, c)$$

3.4 Discussion

In this chapter we defined formally the syntax and semantics of our flexible query language SPARQL^{AR}. We added the concept of answer cost to query evaluation, which can be exploited to rank the answers of a query. We defined the semantics of the APPROX and the RELAX operators applied to SPARQL query conjuncts, and illustrated their usage through several examples.

In contrast to the work in [67], the semantics for approximation and relaxation given here expands the SPARQL 1.1 semantics given in [18, 65] whereas [67] only considered CRPQs. The work on relaxation of triple pattern queries in [44] did not apply to regular path queries and moreover did not associate a distinct cost with

each relaxation operation. On the other hand, the work in [44] allowed also other types of relaxation: dropping a triple pattern, replacing constants with variables, and breaking join dependencies.

The relaxation technique described by Huang et al. in [42, 43] although similar to ours, does not use the additional rules of Figure 3.2 to generate the smallest relaxation steps with respect to an ontology. In fact, to order their query answers, Huang et al. provide a rewriting algorithm that generates multiple queries, each with a score. The score is computed by using a similarity measure between the queries generated by the rewriting algorithm and the original query.

In the next chapter we discuss the computational complexity of various fragments of SPARQL^{AR} and compare these to the corresponding fragments of SPARQL and SPARQL 1.1.

Chapter 4

Complexity of Query Evaluation in SPARQL^{AR}

In this chapter we study the combined, data and query complexity of SPARQL^{AR}, extending the complexity results from [2,64,65,72] for standard SPARQL 1.1 queries. In [64,65,72] it is proved that the fragment of SPARQL with only the AND, UNION, FILTER and SELECT operators (or any subset which includes SELECT) is within the NP complexity class. Here, we study the complexity of the same fragments of SPARQL but with the answer cost also included in the evaluation.

We also extend the complexity results from [2] for SPARQL queries with regular expression patterns to include answer costs and our flexible query operators (APPROX and RELAX). We study the complexity of several fragments of the SPARQL^{AR} language and compare our results with those of [2].

In Section 4.1 we introduce the problem of query evaluation in SPARQL^{AR}. In Section 4.2 we study the complexity of query evaluation for various fragments of SPARQL^{AR}. In Section 4.3 we compare the complexity of SPARQL^{AR} with that of SPARQL 1.1.

4.1 Preliminaries

The combined complexity of query evaluation is based on the following decision problem, which we denote EVALUATION: Given as input a graph $G = (N, D, E)$,

an ontology K, a query Q and a pair $\langle \mu, cost \rangle$, is it the case that $\langle \mu, cost \rangle \in [\![Q]\!]_{G,K}$?

Considering data complexity, the decision problem becomes the following: Given as input a graph G, ontology K and a pair $\langle \mu, cost \rangle$, is it the case that $\langle \mu, cost \rangle \in [\![Q]\!]_{G,K}$, with Q a fixed query?

Finally, the decision problem for query complexity is the following: Given as input an ontology K, a query Q and a pair $\langle \mu, cost \rangle$, is it the case that $\langle \mu, cost \rangle \in [\![Q]\!]_{G,K}$, with G a fixed graph?

4.2 Complexity of SPARQL^{AR}

In this section, we investigate the (combined) complexity of the EVALUATION problem by incrementally adding more operators to the SPARQL^{AR} language fragment being considered.

4.2.1 AND-only Queries with Filter Conditions

We start our analysis with queries containing only the AND and FILTER operators and no regular expression patterns nor any APPROX and RELAX operators. Our proof is based on that in [64] but considers also the answer costs. The size of a query Q, denoted as $|Q|$, is equal to the sum of the size of the triple patterns in Q, where the size of a triple pattern is the number of predicates used in its regular expression pattern.

Theorem 4.1. EVALUATION can be solved in time $O(|E| \cdot |Q|)$ for queries constructed using only the AND and FILTER operators.

Proof. We give an algorithm for the EVALUATION problem that runs in polynomial time: First, for each i such that the triple pattern $\langle x, z, y \rangle_i$ is in Q, we verify that $\langle \mu(\langle x, z, y \rangle_i), 0 \rangle \in E$. If this is not the case, or if $cost \neq 0$ we return False. Otherwise we check if μ satisfies the FILTER conditions and return True or False accordingly. It is evident that the algorithm runs in polynomial time since verifying that $\langle \mu(\langle x, z, y \rangle_i), 0 \rangle \in E$ can be done in time $|E|$ and checking that μ satisfies the FILTER condition R can be done in $|R|$. □

We notice that, by comparing with the results in [64, 65, 72], query evaluation does not increase in complexity when answer costs are added to the semantics. We may also infer that the query and data complexity of the EVALUATION problem is $O(|Q|)$ and $O(|E|)$ respectively.

We now consider queries also containing the regular expression patterns supported in SPARQL 1.1. We show that there is an increase in complexity, from linear to quadratic in the size of the query, thus extending the earlier results of [2] to consider also answer costs:

Theorem 4.2. EVALUATION can be solved in time $O(|E| \cdot |Q|^2)$ for queries that may contain regular expression patterns and that are constructed using only the AND and FILTER operators.

Proof. To show this, we start by building an NFA $M_P = (S, T)$ that recognises $\mathcal{L}(P)$, the language denoted by the regular expression P appearing in one single triple pattern in Q, where S is the set of states (including s_0 and s_f representing the initial and final states respectively) and T is the set of transitions, each of cost 0. We then construct the weighted *product automaton*, H, of G and M_P as follows:

- The states of H are the Cartesian product of the set of nodes N of G and the set of states S of M_P.
- For each transition $\langle\langle s, p, s'\rangle, 0\rangle$ in M_P and each edge $\langle\langle a, p, b\rangle, cost\rangle$ in E, there is a transition $\langle\langle s, a\rangle, \langle s', b\rangle, cost\rangle$ in H.

Then we check if there exists a path from $\langle s_0, \mu(x)\rangle$ to $\langle s_f, \mu(y)\rangle$ in H. In case there is more than one path, we select one with the minimum cost using Dijkstra's algorithm. Knowing that the number of nodes in H is equal to $|N| \cdot |S|$, the number of edges is at most $|E| \cdot |T|$, and that $|T| \leq |S|^2$, the evaluation can be performed in time $O(|E| \cdot |S|^2 + |N| \cdot |S| \cdot \log(|N| \cdot |S|))$. We repeat this for all the triple patterns P in Q. $\square$

From the previous theorem we can also conclude that the query and data complexity of the EVALUATION problem are $O(|Q|^2)$ and $O(|E|)$, respectively.

We now consider queries containing the AND and SELECT operators, but without regular expression patterns or APPROX/RELAX operators, extending earlier results from [72] to consider also answer costs:

Theorem 4.3. EVALUATION is in NP-Hard for queries that do not contain regular expression patterns and that are constructed using only the AND and SELECT operators.

Proof. We first define the problem of graph 3-colourability, which is known to be NP-Complete: given a graph $\Gamma = (N_\Gamma, E_\Gamma)$ and three colours r, g, b, is it possible to assign a colour to each node in N_Γ such that no pair of nodes connected by an edge in E_Γ are of the same colour?

We next define the following RDF-graph $G = (N, D, E)$:

$$
\begin{aligned}
N =& \{r, g, b, a\} \qquad D = \{a, p\} \\
E =& \{\langle\langle r, p, g\rangle, 0\rangle, \\
& \langle\langle r, p, b\rangle, 0\rangle, \langle\langle g, p, b\rangle, 0\rangle, \langle\langle g, p, r\rangle, 0\rangle, \\
& \langle\langle b, p, r\rangle, 0\rangle, \langle\langle b, p, g\rangle, 0\rangle, \langle\langle a, a, a\rangle, 0\rangle\}
\end{aligned}
$$

Now we construct the following query Q such that there is a variable x_i corresponding to each node n_i of Γ and there is a triple pattern of the form $\langle x_i, p, x_j\rangle$ in Q if and only if there is an edge (n_i, n_j) in Γ:

$$
\begin{aligned}
Q = \text{SELECT}\, x \text{ WHERE } ((x_i, p, x_j) \text{ AND } \ldots \text{ AND} \\
(x'_i, p, x'_j) \text{ AND } (a, a, x))
\end{aligned}
$$

It is easy to verify that the graph Γ is colourable if and only if $\langle \mu, 0\rangle \in [\![Q]\!]_G$ with $\mu = \{x \to a\}$. □

We notice that in the proof the graph is of fixed size, hence we can infer that the result of Theorem 4.3 also holds for query complexity. Similarly to Theorem 4.1, the complexity of the SPARQL language does not increase when answer costs are added to the semantics.

4.2.2 Adding the Flexible Operators

We show below in Theorem 4.4 that adding the APPROX and RELAX operators does not increase the complexity class of the EVALUATION problem with respect to the fragment considered in Theorem 4.2. We first prove the following lemma, using similar techniques to those in [67].

Lemma 4.1. EVALUATION of $[\![\text{APPROX}(x, P, y)]\!]_{G,K}$ and $[\![\text{RELAX}(x, P, y)]\!]_{G,K}$ can be accomplished in polynomial time.

Premise. Given a pair $\langle \mu, cost\rangle$ we have to verify that $\langle \mu, cost\rangle \in [\![\text{APPROX}(x, P, y)]\!]_G$ or $\langle \mu, cost\rangle \in [\![\text{RELAX}(x, P, y)]\!]_G$. We start by building an NFA $M_P = (S, T)$ as described earlier. □

Approximation. An *approximate automaton* $A_P = (S, T')$ is constructed starting from M_P and adding the following additional transitions (similarly to the construction in [67]):

- For each state $s \in S$ there is a transition $\langle\langle s, _, s\rangle, \alpha\rangle$, where α is the cost of insertion.

- For each transition $\langle\langle s, p, s'\rangle, 0\rangle$ in M_P, where $p \in D$, there is a transition $\langle\langle s, \epsilon, s'\rangle, \beta\rangle$, where β is the cost of deletion.

- For each transition $\langle\langle s, p, s'\rangle, 0\rangle$ in M_P, where $p \in D$, there is a transition $\langle\langle s, _, s'\rangle, \gamma\rangle$, where γ is the cost of substitution.

We then form the weighted product automaton, H, of G and A_P as follows:

- The states of H will be the Cartesian product of the set of nodes N of G and the set of states S of A_P.

- For each transition $\langle\langle s, p, s'\rangle, cost_1\rangle$ in A_P and each edge $\langle\langle a, p, b\rangle, cost_2\rangle$ in E, there is a transition $\langle\langle s, a\rangle, \langle s', b\rangle, cost_1 + cost_2\rangle$ in H.

- For each transition $\langle\langle s, \epsilon, s'\rangle, cost\rangle$ in A_P and each node $a \in N$, there is a transition $\langle\langle s, a\rangle, \langle s', a\rangle, cost\rangle$ in H.

- For each transition $\langle\langle s, _, s'\rangle, cost_1\rangle$ in A_P and each edge $\langle\langle a, p, b\rangle, cost_2\rangle$ in E, there is a transition $\langle\langle s, a\rangle, \langle s', b\rangle, cost_1 + cost_2\rangle$ in H.

Finally we check if there exists a path from $\langle s_0, \mu(x)\rangle$ to $\langle s_f, \mu(y)\rangle$ in H. Again, if there exists more than one path we select one with minimum cost using Dijkstra's Algorithm. Knowing that the number of nodes in H is $|N| \cdot |S|$ and that the number of edges in H is at most $(|E| + |N|) \cdot |S|^2$, the evaluation can therefore be computed in $O((|E| + |N|) \cdot |S|^2 + |N| \cdot |S| \cdot \log(|N| \cdot |S|))$. $\square$

Relaxation. Given an ontology $K = extRed(K)$ we build the *relaxed automaton* $R_P = (S', T', S_0, S_f)$ starting from M_P (similarly to the construction in [67]). S_0 and S_f represent the sets of initial and final states, and S' contains every state in S plus the states in S_0 and S_f. Initially S_0 and S_f contain s_0 and s_f respectively. Each initial and final state in S_0 and S_f is labelled with either a constant or the symbol $*$; in particular, s_0 is labelled with x if x is a constant or $*$ if it is a variable and similarly s_f is labelled with y if y is a constant or $*$ if it is a variable. An *incoming* (*outgoing*) *clone* of a state s is a new state s' such that s' is an initial or final state if s is, s' has the same set of incoming (outgoing) transitions as s, and has no outgoing (incoming) transitions. Initially T' contains all the transitions in T. We recursively add states to S_0 and S_f, and transitions to T' as follows until we reach a fixpoint:

- For each transition $\langle\langle s, p, s'\rangle, cost\rangle \in T'$ and $\langle p, sp, p'\rangle \in K$ add the transition $\langle\langle s, p', s'\rangle,\ cost + \alpha\rangle$ to T', where α is the cost of applying rule 2.

- For each transition $\langle\langle s, type, s'\rangle, cost\rangle \in T'$, $s' \in S_f$ and $\langle c, sc, c'\rangle \in K$ such that s' is annotated with c add an outgoing clone s'' of s' annotated with c' to S_f and add the transition $\langle\langle s, type, s''\rangle, cost + \beta\rangle$ to T', where β is the cost of applying rule 4.

- For each transition $\langle\langle s, type^-, s'\rangle, cost\rangle \in T'$, $s \in S_0$ and $\langle c, sc, c'\rangle \in K$ such that s is annotated with c add an incoming clone s'' of s annotated with c' to S_0 and add the transition $\langle\langle s'', type^-, s'\rangle, cost + \beta\rangle$ to T', where β is the cost of applying rule 4.

- For each $\langle\langle s, p, s'\rangle, cost\rangle \in T'$, $s' \in S_f$ and $\langle p, dom, c\rangle$ such that s' is annotated with a constant, add an outgoing clone s'' of s' annotated with c to S_f, and add the transition $\langle\langle s, type, s''\rangle, cost+\gamma\rangle$ to T', where γ is the cost of applying rule 5.

- For each $\langle\langle s, p, s'\rangle, cost\rangle \in T'$, $s \in S_0$ and $\langle p, range, c\rangle$ such that s is annotated with a constant, add an incoming clone s'' of s annotated with c to S_0, and add the transition $\langle\langle s'', type^{-}, s'\rangle, cost+\delta\rangle$ to T', where δ is the cost of applying rule 6.

(We note that because queries and graphs do not contain edges labelled *sc* or *sp*, rules 1 and 3 in Figure 3.1 are inapplicable as far as query relaxation is concerned.)

We then form the weighted product automaton, H, of G and R_P as follows:

- For each node $a \in N$ of G and each state $s \in S'$ of R_P, then $\langle s, a\rangle$ is a state of H if s is labelled with either $*$ or a, or is unlabelled.

- For each transition $\langle\langle s, p, s'\rangle, cost_1\rangle$ in R_P and each edge $\langle\langle a, p, b\rangle, cost_2\rangle$ in E such that $\langle s, a\rangle$ and $\langle s', b\rangle$ are states of H, then there is a transition $\langle\langle s, a\rangle, \langle s', b\rangle, cost_1 + cost_2\rangle$ in H.

- For each transition $\langle\langle s, type^{-}, s'\rangle, cost_1\rangle$ in R_P and each edge $\langle\langle a, type, b\rangle, cost_2\rangle$ in E such that $\langle s, b\rangle$ and $\langle s', a\rangle$ are states of H, then there is a transition $\langle\langle s, b\rangle, \langle s', a\rangle, cost_1 + cost_2\rangle$ in H.

Finally we check if there exists a path from $\langle s, \mu(x)\rangle$ to $\langle s', \mu(y)\rangle$ in H, where $s \in S_0$ and $s' \in S_f$. Again, if there exists more than one path we select one with minimum cost using Dijkstra's Algorithm. Knowing that the number of nodes in H is at most $|N| \cdot |S'|$ and the number of edges in H is at most $|E| \cdot |S'|^2$, the evaluation can therefore be computed in $O(|E| \cdot |S'|^2 + |N| \cdot |S'| \cdot \log(|N| \cdot |S'|))$. □

Conclusion. We can conclude that both query approximation and query relaxation can be evaluated in polynomial time. □

From the previous lemma we can also infer that the data and query complexity of the EVALUATION problem for $[\![\text{APPROX}(x, P, y)]\!]_{G,K}$ and $[\![\text{RELAX}(x, P, y)]\!]_{G,K}$ is $O(|E|)$ and polynomial time respectively.

Theorem 4.4. EVALUATION can be solved in polynomial time for queries that may contain regular expressions and are constructed using the AND, FILTER, RELAX and APPROX operators.

Proof. We extend the EVALUATION algorithm from the proof of Theorem 4.2 and show that it runs in polynomial time: First, for each i such that the triple pattern APPROX$(x, P, y)_i$ (or RELAX$(x, P, y)_i$) is in Q, we verify that $\langle \mu_i, cost_i \rangle \in [\![\text{APPROX}(x, P, y)_i]\!]_{G,K}$ (or $[\![\text{RELAX}(x, P, y)_i]\!]_{G,K}$) for some $cost_i$ where $\mu_i = \{x \to \mu(x), y \to \mu(y)\}$. For each j such that the triple pattern $\langle x, z, y \rangle_i$ is in Q, we verify that $\langle \mu(\langle x, z, y \rangle_i), 0 \rangle \in E$. If one of these fail, or if $\sum_i cost_i \neq cost$ we return False. Otherwise we check if μ satisfies the FILTER conditions and return True or False accordingly. By Theorem 4.2 and Lemma 4.1 we know that verifying $[\![(x, P, y)_i]\!]_{G,K}$, $[\![\text{APPROX}(x, P, y)_i]\!]_{G,K}$ and $[\![\text{RELAX}(x, P, y)_i]\!]_{G,K}$ can be done in polynomial time, hence the evaluation problem can be solved in polynomial time. □

The following theorem establishes the data complexity of an extended language fragment adding also SELECT:

Theorem 4.5. EVALUATION has P-Time data complexity for queries that may contain regular expression patterns and that are constructed using the AND, FILTER, RELAX, APPROX and SELECT operators.

Proof. In order to prove this, we devise an algorithm that runs in polynomial time with respect to the size of the graph G. We start by building a new mapping μ' such that each variable $x \in var(\mu')$ appears in $var(Q)$ but not in $var(\mu)$, and to each we assign a constant from ND. We then verify in polynomial time that $\langle \mu \cup \mu', cost \rangle$ is in $[\![Q]\!]_G$. The number of mappings we can generate is $O(|ND|^{|var(Q)|})$. Since the query is fixed we can therefore say that the evaluation with respect to the data is in polynomial time. □

4.2.3 Adding UNION

We conclude our complexity study by adding the UNION operator to the language which, in addition to the previous operators, results in the SPARQL^{AR} language:

Theorem 4.6. EVALUATION is in NP for queries that may contain regular expression patterns and are constructed using the AND, FILTER, RELAX, APPROX, SELECT and UNION operators.

Proof. We give an NP algorithm, EvaluationCost shown as Algorithm 1, for the EVALUATION problem for a generic query Q containing AND, UNION and regular expression patterns. The EvaluationCost algorithm takes as input a mapping μ, a graph G and a query Q and returns a cost c. Given an evaluation $\langle \mu, c' \rangle$, and a query Q, then the EvaluationCost algorithm returns c if $\{\langle \mu, c \rangle\} \in [\![Q]\!]_G$ and $NULL$ otherwise. Finally, we need to check that c is equal to c'.

It is easy to see in the EvaluationCost algorithm that the non-deterministic step occurs when the condition $Q = Q_1$ AND Q_2 is satisfied, in which case we need to guess a decomposition of the mapping μ into μ_1 and μ_2. The number of guesses is bounded by the number of possible decompositions of μ (which is finite).

Algorithm 1: EvaluationCost

```
input  : Query Q, a mapping μ, a graph G
output: A cost value c, or NULL
if Q = t then
    if there exists a cost such that {⟨μ, cost⟩} ∈ [[t]]_G, where t is a simple
    triple pattern or an APPROX/RELAX then
        return cost;
    else
        return NULL
else if Q = Q1 AND Q2 then
    Guess a decomposition μ = μ1 ∪ μ2;
    if EvaluationCost(Q1,μ1,G) ≠ NULL and EvaluationCost(Q2,μ2,G)
    ≠ NULL then
        return EvaluationCost(Q1,μ1,G) + EvaluationCost(Q2,μ2,G);
    else
        return NULL
else if Q = Q1 UNION Q2 then
    if EvaluationCost(Q1,μ,G) = NULL then
        return EvaluationCost(Q2,μ,G);
    else if EvaluationCost(Q2,μ,G) = NULL then
        return EvaluationCost(Q1,μ,G);
    return min(EvaluationCost(Q1,μ,G), EvaluationCost(Q2,μ,G));
```

We now show that by including the SELECT operator the evaluation problem

is still in NP. Given a pair $\langle \mu, cost \rangle$ and a query SELECT $\overrightarrow{w}$ WHERE Q, where Q contains AND, UNION and SELECT, we have to check whether $\langle \mu, cost \rangle$ is in $[\![\text{SELECT } \overrightarrow{w} \text{ WHERE } Q]\!]_G$. We can guess a new mapping μ' such that $\pi_{\overrightarrow{w}}(\langle \mu', cost \rangle) = \langle \mu, cost \rangle$ and consequently check that $\langle \mu', cost \rangle \in [\![Q]\!]_G$ (which can be done in NP time with the EvaluationCost algorithm). The number of guesses is bounded by the number of variables in Q and values from G to which they can be mapped. □

Theorem 4.7. EVALUATION is NP-Complete for queries that may contain regular expression patterns and are constructed using the AND, FILTER, RELAX, APPROX, SELECT and UNION operators.

Proof. By combining Theorems 4.6 and 4.3, we know that the evaluation problem is both in NP and NP-Hard. Therefore, the evaluation problem is NP-Complete. □

The above theorem is true also for query complexity as Theorems 4.6 and 4.3 consider graphs of a fixed size. The following theorem considers the data complexity of EVALUATION for the same language:

Theorem 4.8. EVALUATION is P-Time in data complexity for queries that may contain and regular expression patterns, and that are constructed using the AND, FILTER, RELAX, APPROX, SELECT and UNION operators.

Proof. In Theorem 4.6, the non-deterministic steps (i.e. the decomposition of the mapping μ into μ_1 and μ_2 to verify that $\langle \mu, c \rangle \in [\![Q_1 \text{ AND } Q_2]\!]_G$), depend on the query Q which we assume is fixed. To verify that an evaluation $\langle \mu, 0 \rangle$ is in $[\![t]\!]_G$, with t a triple pattern of query Q, can be done in $|E|$ steps. Therefore, the evaluation can be computed in $O(|E| * |\mu|^{|Q|})$ steps.

When we include the SELECT operator we need to add a further non-deterministic step, that is, generating a new mapping μ' from μ such that $\pi_{\overrightarrow{w}}(\langle \mu', c \rangle) = \langle \mu, c \rangle$. From the proof of Theorem 4.5 we can see that this can be done in $O(|ND|^{|var(Q)|})$. Since the query is fixed, we conclude that the data complexity is polynomial time. □

Table 4.1: Complexity of various SPARQL^{AR} fragments previously known but with answer costs included.

Operators	Data Complexity	Query Complexity	Combined Complexity	Results From
AND, FILTER	$O(\|E\|)$	$O(\|Q\|)$	$O(\|E\| \cdot \|Q\|)$	Theorem 4.1
AND, FILTER, RegEx	$O(\|E\|)$	$O(\|Q\|^2)$	$O(\|E\| \cdot \|Q\|^2)$	Theorem 4.2
AND, SELECT	P-Time	NP-Complete	NP-Complete	Theorems 4.3, 4.5 and 4.6
RELAX, APPROX	$O(\|E\|)$	P-Time	P-Time	Lemma 4.1

Table 4.2: New complexity results for three SPARQL^{AR} fragments.

Operators	Data Complexity	Query Complexity	Combined Complexity	Results From
AND, FILTER, RELAX, APPROX, RegEx	$O(\|E\|)$	P-Time	P-Time	Theorem 4.4
AND, FILTER, RELAX, APPROX, RegEx, SELECT	P-Time	NP-Complete	NP-Complete	Theorems 4.3, 4.4, 4.5 and 4.6
AND, FILTER, RELAX, APPROX, RegEx, SELECT, UNION	P-Time	NP-Complete	NP-Complete	Theorems 4.6, 4.7 and 4.8

4.3 Result Summary

In Table 4.1 we show the complexity of some fragments of SPARQL^{AR} that are extensions of the work from [2, 64, 65, 67, 72]. In particular the proofs for Theorems 4.1, 4.2, 4.3 were modified from [64,65,72] so that answer costs and regular expression

patterns were taken into account. The query and combined complexity of the third row is inferred by combining Theorems 4.3 and 4.6 (as the latter holds also for the specified fragment); data complexity, on the other hand, is inferred by considering Theorem 4.5 for the specified fragment.

In Table 4.2 we list the complexity of three fragments of the SPARQL^{AR} language that have not been previously investigated. The data and query complexity of the first row can be inferred by combining the data and query complexity deduced from Theorem 4.2 with Lemma 4.1. The combined and query complexity for the fragment specified in the second row of the table is deduced by combining Theorems 4.3, 4.4 and 4.6.

The complexity of the full SPARQL^{AR} language (Theorem 4.7) is equivalent to the complexity of queries containing only the AND and SELECT. Hence, the addition of the flexible operators does not increase the worst-case complexity of query evaluation. Moreover, we notice that adding answer costs to the semantics does not increase the complexity either.

If we consider the whole of SPARQL 1.1 the reader may notice that we have excluded the OPTIONAL operator in our analysis. It is possible to add the OPTIONAL operator to SPARQL^{AR}, allowing APPROX and RELAX to be applied to triple patterns occurring within an OPTIONAL clause, with the same semantics as specified in Chapter 3. However, the complexity of SPARQL with the OPTIONAL clause is PSPACE-complete [64]. Therefore, by the results in this chapter, the complexity of SPARQL^{AR} would also increase similarly.

4.4 Discussion

In this chapter we have discussed the computational complexity of SPARQL^{AR} and some of its fragments. We have proved that the complexity class of our language is the same as that for the fragment of SPARQL 1.1 that we extend. Hence, the APPROX and RELAX operators do not worsen the overall complexity of the language. Our complexity results extend those in [2] and [72] by including answer costs within the semantics, and the APPROX and RELAX operators. We have shown

that SPARQL queries that contain the AND and SELECT operators have the same complexity class as SPARQL^{AR}. In contrast to [67] our study also included the FILTER and UNION operators.

In the next chapter we present an evaluation algorithm for SPARQL^{AR} queries which is based on query rewriting. The algorithm generates multiple SPARQL 1.1 queries that can be evaluated using any SPARQL querying system that supports property paths. We prove that our algorithm is sound and complete, and that it terminates after a finite number of steps.

Chapter 5

Query Evaluation in SPARQL^{AR}

In this chapter we discuss the algorithms we have designed for evaluating SPARQL^{AR} queries. We evaluate SPARQL^{AR} queries by making use of a *query rewriting algorithm*. A similar approach was also adopted by Huang et al. [42, 43] where they generate a tree of queries in which each parent query is a relaxed form of its child. Our work extends this previous work by considering queries with regular expression patterns and also by including the concept of query approximation.

A different query evaluation approach for APPROX and RELAX has been proposed in [73] where Selmer et al. describe a system that retrieves the answers of a flexible query by incrementally constructing and navigating a non-deterministic finite state automaton (NFA) that is the product of the query NFA and the data graph. Our approach leverages an existing SPARQL API (Jena) and hence could be readily modified to use other SPARQL query evaluation APIs.

We present our query rewriting algorithm in Section 5.1 and prove its soundness and completeness in Section 5.2. We conclude by proving the termination property of the algorithm in Section 5.3.

5.1 Rewriting Algorithm

Our query rewriting algorithm (see Algorithm 3 below) starts by considering the query Q_0, which returns the exact answers to the user's query Q, i.e. ignoring any occurrences of the APPROX and RELAX operators in Q. To keep track of which

triple patterns need to be relaxed or approximated, we label such triple patterns with A for approximation and R for relaxation.

In Algorithm 3, the function *toCQS* ("to conjunctive query set") takes as input a query Q_{AR}, and returns a set of pairs $\langle Q_i, 0\rangle$ such that $\bigcup_i [\![Q_i]\!]_G = [\![Q_{AR}]\!]_G$ and no Q_i contains the UNION operator. The function *toCQS* exploits the following equalities:

$$
\begin{aligned}
&[\![(Q_1 \text{ UNION } Q_2) \text{ AND } Q_3]\!]_G = \\
&([\![Q_1]\!]_G \cup [\![Q_2]\!]_G) \bowtie [\![Q_3]\!]_G = \\
&([\![Q_1]\!]_G \bowtie [\![Q_3]\!]_G) \cup ([\![Q_2]\!]_G \bowtie [\![Q_3]\!]_G) = \\
&([\![Q_1 \text{ AND } Q_3]\!]_G) \cup ([\![Q_2 \text{ AND } Q_3]\!]_G)
\end{aligned}
$$

We assign to the variable *oldGeneration* the set of queries returned by $toCQS(Q_0)$. For each query Q in the set *oldGeneration*, each triple pattern $\langle x, P, y\rangle$ in Q labelled with A (R), and each URI p that appears in P, we apply one step of approximation (relaxation) to p, and we assign the cost of applying that approximation (relaxation) to the resulting query. The *applyApprox* and *applyRelax* functions invoked by Algorithm 3 are shown as Algorithms 4 and 6, respectively. From each query constructed in this way, we next generate a new set of queries by applying a second step of approximation or relaxation. We continue to generate queries iteratively in this way. The cost of each query generated is the summed cost of the sequence of approximations or relaxations that have generated it. If the same query is generated more than once, only the one with the lowest cost is retained. Moreover, the set of queries generated is kept sorted by increasing cost. For practical reasons, we limit the number of queries generated by bounding the cost of queries up to a maximum value c.

In Algorithm 3, the *addTo* function takes two arguments: the first is a collection C of query/cost pairs, while the second is a single query/cost pair $\langle Q, c\rangle$. The operator adds $\langle Q, c\rangle$ to C. If C already contains a pair $\langle Q, c'\rangle$ such that $c' \geq c$, then $\langle Q, c'\rangle$ is replaced by $\langle Q, c\rangle$ in C.

To compute the query answers (see Algorithm 2) we apply an evaluation function, *eval*, to each query generated by the rewriting algorithm (in order of increasing cost

of the queries) and to each mapping returned by *eval* we assign the cost of the query. If we generate a particular mapping more than once, only the one with the lowest cost is retained. In Algorithm 2, *rewrite* denotes the query rewriting algorithm (i.e. Algorithm 3) and the set of mappings M is maintained in order of increasing cost.

Algorithm 2: Flexible Query Evaluation

---

input : Query Q; approx/relax max cost c; Graph G; Ontology K.
output: List M of mapping/cost pairs, sorted by cost.
$M := \emptyset$;
foreach $\langle Q', cost \rangle \in$ *rewrite(Q,c,K)* **do**
  foreach $\langle \mu, 0 \rangle \in$ *eval(Q′,G)* **do**
    $M := M \cup \{\langle \mu, cost \rangle\}$
return M;

.

Algorithm 3: Rewriting algorithm

---

input : Query Q_{AR}; approx/relax max cost c; Ontology K.
output: List of query/cost pairs, sorted by cost.
Q_0 := remove the APPROX and RELAX operators, and the label triple patterns of Q_{AR};
queries := *toCQS*(Q_0);
oldGeneration := *toCQS*(Q_0);
while *oldGeneration* $\neq \emptyset$ **do**
  newGeneration := $\emptyset$;
  foreach $\langle Q, cost \rangle \in$ *oldGeneration* **do**
    foreach *labelled triple pattern* $\langle x, P, y \rangle$ *in* Q **do**
      rew := $\emptyset$;
      if $\langle x, P, y \rangle$ *is labelled with* A **then**
        rew := applyApprox(Q,$\langle x, P, y \rangle$);
      else if $\langle x, P, y \rangle$ *is labelled with* R **then**
        rew := applyRelax(Q,$\langle x, P, y \rangle$,K);
      foreach $\langle Q', cost' \rangle \in$ *rew* **do**
        if $cost + cost' \leq c$ **then**
          addTo(*newGeneration*, $\langle Q', cost + cost' \rangle$);
          addTo(*queries*, $\langle Q', cost + cost' \rangle$) ; `/* The elements of` *newGeneration* `and` *queries* `are also kept sorted by increasing cost. */`
  oldGeneration := *newGeneration*;
return *queries*;

.

Algorithm 4: applyApprox

```
input  : Query Q; triple pattern ⟨x, P, y⟩_A.
output: Set S of query/cost pairs.
S := ∅;
foreach ⟨P', cost⟩ ∈ approxRegex(P) do
    Q' := replace ⟨x, P, y⟩_A by ⟨x, P', y⟩_A in Q;
    S := S ∪ {⟨Q', cost⟩};
return S;
```

. . .

Algorithm 5: approxRegex

```
input  : Regular Expression P.
output: Set T of RegEx/cost pairs.
T := ∅;
if P = _ or P = ϵ then
    return T;
else if P = p where p is a URI then
    T := T ∪ {⟨ϵ, cost_d⟩};
    T := T ∪ {⟨_, cost_s⟩};
    T := T ∪ {⟨_/p, cost_i⟩};
    T := T ∪ {⟨p/_, cost_i⟩};
else if P = P1/P2 then
    foreach ⟨P', cost⟩ ∈ approxRegex(P1) do
        T := T ∪ {⟨P'/P2, cost⟩};
    foreach ⟨P', cost⟩ ∈ approxRegex(P2) do
        T := T ∪ {⟨P1/P', cost⟩};
else if P = P1|P2 then
    foreach ⟨P', cost⟩ ∈ approxRegex(P1) do
        T := T ∪ {⟨P', cost⟩};
    foreach ⟨P', cost⟩ ∈ approxRegex(P2) do
        T := T ∪ {⟨P', cost⟩};
else if P = P1* then
    foreach ⟨P', cost⟩ ∈ approxRegex(P1) do
        T := T ∪ {⟨(P1*)/P'/(P1*), cost⟩};
return T;
```

The *applyApprox* (Algorithm 4) and *applyRelax* (Algorithm 6) functions invoke the functions *approxRegex* (Algorithm 5) and *replaceTriplePattern* (Algorithm 7),

Algorithm 6: applyRelax

input : Query Q; triple pattern $\langle x, P, y\rangle_R$ of Q; Ontology K.
output: Set S of query/cost pairs.
$S := \emptyset$;
foreach $\langle\langle x', P', y'\rangle_R, cost\rangle \in relaxTriplePattern(\langle x, P, y\rangle, K)$ **do**
    $Q' :=$ replace $\langle x, P, y\rangle_R$ by $\langle x', P', y'\rangle_R$ in Q;
    $S := S \cup \{\langle Q', cost\rangle\}$;
return S;

respectively. In Algorithm 7, z, z_1 and z_2 are fresh new variables. The *relax-TriplePattern* function might generate regular expressions containing a URI $type^-$, which are matched to edges in E by reversing the subject and the object and using the property label *type*. The predicate $type^-$ is generated when we apply rule 6 of Figure 3.1 to a triple pattern. Given a triple pattern $\langle x, a, y\rangle$ where x is a constant and is y a variable, and an ontology statement $\langle a, range, d\rangle$, we can deduce the triple pattern $\langle y, type, d\rangle$. If instead the predicate a appears in a triple pattern containing a regular expression such as $\langle x, a/b, z\rangle$ (which is equivalent to $\langle x, a, y\rangle$ AND $\langle y, b, z\rangle$), then we cannot simply replace it with $\langle y, type, d\rangle$ as the regular expression would be broken apart and two triple patterns would result. By using $\langle d, type^-, y\rangle$, we correctly construct the triple pattern $\langle d, type^-/b, z\rangle$.

In the following example, we illustrate how the rewriting algorithm works by showing the queries it generates, starting from a SPARQL^{AR} query.

Example 5.1. Consider the following ontology K (satisfying $K = extRed(K)$), which is a fragment of the YAGO knowledge base:

$$K = (\{happenedIn, placedIn, Event\},$$
$$\{\langle happenedIn, sp, placedIn\rangle,$$
$$\langle happenedIn, dom, Event\rangle\})$$

Suppose a user wishes to find every event which took place in London on 15th September 1940 and poses the following query Q:

$$\text{APPROX}(x, happenedOnDate, \text{“15/09/1940”})$$
$$\text{AND RELAX}(x, happenedIn, \text{“}London\text{”}).$$

Algorithm 7: relaxTriplePattern

```
input  : Triple pattern ⟨x, P, y⟩; Ontology K.
output: Set T of triple pattern/cost pairs.
T := ∅;
if P = p where p is a URI then
    foreach p' such that ∃(p, sp, p') ∈ E_K do
        T := T ∪ {⟨⟨x, p', y⟩, cost_2⟩};
    foreach b such that ∃(a, sc, b) ∈ E_K and p = type and y = a do
        T := T ∪ {⟨⟨x, type, b⟩, cost_4⟩};
    foreach b such that ∃(a, sc, b) ∈ E_K and p = type⁻ and x = a do
        T := T ∪ {⟨⟨b, type⁻, y⟩, cost_4⟩};
    foreach a such that ∃(p, dom, a) ∈ E_K and y is a URI or a Literal do
        T := T ∪ {⟨⟨x, type, a⟩, cost_5⟩};
    foreach a such that ∃(p, range, a) ∈ E_K and x is a URI do
        T := T ∪ {⟨⟨a, type⁻, y⟩, cost_6⟩};
else if P = P1/P2 then
    foreach ⟨⟨x', P', z⟩, cost⟩ ∈ relaxTriplePattern(⟨x, P1, z⟩) do
        T := T ∪ {⟨⟨x', P'/P2, y⟩, cost⟩};
    foreach ⟨⟨z, P', y'⟩, cost⟩ ∈ relaxTriplePattern(⟨z, P2, y⟩) do
        T := T ∪ {⟨⟨x, P1/P', y'⟩, cost⟩};
else if P = P1|P2 then
    foreach ⟨⟨x', P', y'⟩, cost⟩ ∈ relaxTriplePattern(⟨x, P1, y⟩) do
        T := T ∪ {⟨⟨x', P', y'⟩, cost⟩};
    foreach ⟨⟨x', P', y'⟩, cost⟩ ∈ relaxTriplePattern(⟨x, P2, y⟩) do
        T := T ∪ {⟨⟨x', P', y'⟩, cost⟩};
else if P = P1* then
    foreach ⟨⟨z1, P', z2⟩, cost⟩ ∈ relaxTriplePattern((⟨z1, P1, z2⟩) do
        T := T ∪ {⟨⟨x, P1*/P'/P1*, y⟩, cost⟩};
    foreach ⟨⟨x', P', z⟩, cost⟩ ∈ relaxTriplePattern((⟨x, P1, z⟩) do
        T := T ∪ {⟨⟨x', P'/P1*, y⟩, cost⟩};
    foreach ⟨⟨z, P', y'⟩, cost⟩ ∈ relaxTriplePattern((⟨z, P1, y⟩) do
        T := T ∪ {⟨⟨x, P1*/P', y'⟩, cost⟩};
return T;
```

Without applying APPROX or RELAX this query does not return any answers when evaluated on the YAGO endpoint (because "happenedIn" connects to URIs representing places and "London" is a literal, not a URI). After the first step of approximation and relaxation, the following queries are generated:

$$Q_1 = (x, \epsilon, \text{"15/09/1940"})_A \text{ AND } (x, happenedIn, \text{"}London\text{"})_R$$
$$Q_2 = (x, happenedOnDate/_, \text{"15/09/1940"})_A \text{ AND } (x, happenedIn, \text{"}London\text{"})_R$$
$$Q_3 = (x, _/happenedOnDate, \text{"15/09/1940"})_A \text{ AND } (x, happenedIn, \text{"}London\text{"})_R$$
$$Q_4 = (x, _, \text{"12/12/12"})_A \text{ AND } (x, happenedIn, \text{"}London\text{"})_R$$
$$Q_5 = (x, happenedOnDate, \text{"15/09/1940"})_A \text{ AND } (x, placedIn, \text{"}London\text{"})_R$$
$$Q_6 = (x, happenedOnDate, \text{"15/09/1940"})_A \text{ AND } (x, type, Event)_R$$

All of these also return empty results, with the exception of query Q_6 which returns every event occurring on 15/09/1940 (YAGO contains only one such event, namely "Battle of Britain").

5.2 Soundness and Completenesss

We now discuss the soundness, completeness and termination of our rewriting algorithm. As we stated earlier, the algorithm takes as one of its inputs a cost that limits the number of queries generated. Therefore the classic definitions of soundness and completeness need to be modified. Given an initial query Q we denote by $rew(Q)_c$ the set of queries generated by the rewriting algorithm whose answers have cost less than or equal to c.

Definition 5.1 (Soundness)**.** The rewriting of Q, $rew(Q)_c$, is *sound* if the following holds: $\bigcup_{Q' \in rew(Q)_c} [\![Q']\!]_{G,K} \subseteq [\![Q]\!]_{G,K,c}$ for every graph G and ontology K.

Definition 5.2 (Completeness)**.** The rewriting of Q, $rew(Q)_c$, is *complete* if the following holds: $[\![Q]\!]_{G,K,c} \subseteq \bigcup_{Q' \in rew(Q)_c} [\![Q']\!]_{G,K}$ for every graph G and ontology K.

To show the soundness and completeness of the query rewriting algorithm (in Theorem 5.1), we first require two lemmas and a corollary:

Lemma 5.1. Given four sets of evaluation results M_1, M_2, M'_1 and M'_2 such that $M_1 \subseteq M'_1$ and $M_2 \subseteq M'_2$, it holds that:

$$M_1 \cup M_2 \subseteq M'_1 \cup M'_2 \qquad (5.2.1)$$

$$M_1 \bowtie M_2 \subseteq M'_1 \bowtie M'_2 \qquad (5.2.2)$$

Proof. 5.2.1: From the definition of union, it follows that $M'_1 \cup M'_2$ contains every mapping from M_1 and M_2, and therefore the statement holds.

5.2.2: From the definition of join, $M_1 \bowtie M_2$ contains a mapping $\mu_1 \cup \mu_2$ for every pair of compatible mappings $\langle \mu_1, cost_1 \rangle \in M_1$ and $\langle \mu_2, cost_2 \rangle \in M_2$. Since M'_1 and M'_2 also contain μ_1 and μ_2, respectively, then $M'_1 \bowtie M'_2$ will contain $\mu_1 \cup \mu_2$. □

The following result follows from Lemma 5.1:

Corollary 5.1. Given four sets of evaluation results M_1, M_2, M'_1 and M'_2 such that $M_1 = M'_1$ and $M_2 = M'_2$, it holds that:

$$M_1 \cup M_2 = M'_1 \cup M'_2 \tag{5.2.3}$$

$$M_1 \bowtie M_2 = M'_1 \bowtie M'_2 \tag{5.2.4}$$

Lemma 5.2. Given SPARQL^{AR} queries Q_1 and Q_2, graph G, ontology K and cost c, the following equations hold:

$$\text{CostProj}((\llbracket Q_1 \rrbracket_{G,K} \bowtie \llbracket Q_2 \rrbracket_{G,K}), c) = \text{CostProj}((\llbracket Q_1 \rrbracket_{G,K,c} \bowtie \llbracket Q_2 \rrbracket_{G,K,c}), c)$$
$$\text{CostProj}((\llbracket Q_1 \rrbracket_{G,K} \cup \llbracket Q_2 \rrbracket_{G,K}), c) = \llbracket Q_1 \rrbracket_{G,K,c} \cup \llbracket Q_2 \rrbracket_{G,K,c}$$

Proof. Considering the right hand side (RHS) of the first equation, we know that each pair $\langle \mu, cost \rangle$ in the RHS has $cost \leq c$ and is equal to $\langle \mu_1, cost_1 \rangle \bowtie \langle \mu_2, cost_2 \rangle$, where $cost_1 \leq c$, $cost_2 \leq c$, $\langle \mu_1, cost_1 \rangle \in \llbracket Q_1 \rrbracket_{G,K}$ and $\langle \mu_2, cost_2 \rangle \in \llbracket Q_2 \rrbracket_{G,K}$. Therefore, the pair $\langle \mu, cost \rangle$ must also be contained in the left hand side (LHS) of the equation. Conversely, for each pair $\langle \mu, cost \rangle$ in the LHS, we know that $cost \leq c$ and that there must exist a pair $\langle \mu_1, cost_1 \rangle \in \llbracket Q_1 \rrbracket_{G,K}$ and a pair $\langle \mu_2, cost_2 \rangle \in \llbracket Q_2 \rrbracket_{G,K}$ such that $\langle \mu_1, cost_1 \rangle \bowtie \langle \mu_2, cost_2 \rangle = \langle \mu, cost \rangle$. Moreover, since $cost = cost_1 + cost_2$ we know that $cost_1 \leq c$ and $cost_2 \leq c$. Therefore, we can conclude that $\langle \mu, cost \rangle$ must also be contained in the RHS of the equation.

For the second equation it is easy to verify that every evaluation pair $\langle \mu, cost \rangle$ is in $\text{CostProj}(\llbracket Q_1 \rrbracket_{G,K} \cup \llbracket Q_2 \rrbracket_{G,K}, c)$ if and only if it is contained either in $\llbracket Q_1 \rrbracket_{G,K,c}$ or in $\llbracket Q_2 \rrbracket_{G,K,c}$, or in both. □

We also define the following operator *addCost* which increments the cost of a set of mappings by i:

$$\text{addCost}([\![Q]\!]_{G,K}, i) = \{\langle \mu, c+i \rangle \mid \langle \mu, c \rangle \in [\![Q]\!]_{G,K}\}$$

Theorem 5.1. The Rewriting Algorithm is sound and complete.

Proof. For ease of reading, in this proof we will replace the operators APPROX and RELAX with A and R respectively and will denote with $A/R(t)$ that we are applying either APPROX or RELAX to a triple pattern t. We divide the proof into three parts: **(1)** The first part shows that for $c \geq 0$ and relaxed or approximated triple patterns of the form $\langle x, p, y \rangle$, the algorithm is sound and complete. **(2)** The second part of the proof shows that the algorithm is sound and complete for approximated and relaxed triple patterns containing any regular expression. **(3)** Finally, we show that the algorithm is sound and complete for general queries Q, i.e. we show that the following holds for any query Q, graph G and ontology K:

$$[\![Q]\!]_{G,K,c} \subseteq \bigcup_{Q' \in rew(Q)_c} [\![Q']\!]_{G,K} \subseteq [\![Q]\!]_{G,K,c}$$

(1) In this first part we show that for any triple pattern $\langle x, p, y \rangle$ and cost $c \geq 0$ the following holds:

$$[\![A/R(x,p,y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,p,y))_c} [\![t']\!]_{G,K}$$

We show this by induction on the cost c. For the base case of $c = 0$ we need to show that:

$$[\![A/R(x,p,y)]\!]_{G,K,0} = \bigcup_{t' \in rew(A/R(x,p,y))_0} [\![t']\!]_{G,K} \qquad (5.2.5)$$

On the LHS, since the costs of applying APPROX and RELAX have cost greater than zero, the CostProj operator in the definition of $[\![A/R(x,p,y)]\!]_{G,K,0}$ (see Section 3.3) will only return the exact answers of the query, in other words it will exclude the answers generated by the APPROX and RELAX operators. On the RHS, the rewriting algorithm will not return queries with associated cost greater than 0 and therefore will just return the original query unchanged. This, when evaluated, will therefore also return the exact answers of the query. So 5.2.5 holds.

When c is greater than 0 we consider the two different cases, one for APPROX and the other for RELAX:

(a) Approximation. For approximation, we show the following by induction on the cost c:

$$[\![A(x,p,y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A(x,p,y))_c} [\![t']\!]_{G,K} \tag{5.2.6}$$

The induction hypothesis is that 5.2.6 holds for $c = i\alpha + j\beta + k\gamma$ for all $i, j, k \geq 0$, where α, β, γ are the cost of the insertion, deletion and substitution edit operations, respectively. We have already shown the base case of $i = j = k = 0$. We now show that 5.2.6 is true when one of, i, j or k is incremented by 1.

Considering the RHS of Equation 5.2.6, when we apply one step of approximation to a triple pattern the algorithm generates a set of triple patterns, that will be recursively rewritten by the algorithm. Therefore, by applying every possible edit operation to the original triple pattern (x, p, y), we have that[1]:

$$\begin{aligned}
\bigcup_{t' \in rew(A(x,p,y))_c} [\![t']\!]_{G,K} = [\![\langle x,p,y\rangle]\!]_{G,K} \cup \\
&\text{addCost}(\bigcup_{t' \in rew(A(x,_/p,y))_{c-\alpha}} [\![t']\!]_{G,K}, \alpha) \cup \\
&\text{addCost}(\bigcup_{t' \in rew(A(x,p/_,y))_{c-\alpha}} [\![t']\!]_{G,K}, \alpha) \cup \\
&\text{addCost}(\bigcup_{t' \in rew(A(x,\epsilon,y))_{c-\beta}} [\![t']\!]_{G,K}, \beta) \cup \\
&\text{addCost}(\bigcup_{t' \in rew(A(x,_,y))_{c-\gamma}} [\![t']\!]_{G,K}, \gamma)
\end{aligned}$$

Considering the LHS of Equation 5.2.6, by the semantics of approximation (see Section 3.2.1), we have that:

[1] On the RHS, addCost is incrementing the cost of the evaluation by the cost of the edit operation being applied.

$$
\begin{aligned}
[\![A(x,p,y)]\!]_{G,K,c} = [\![\langle x,p,y\rangle]\!]_{G,K} \cup \\
&\text{addCost}([\![A(x,_/p,y)]\!]_{G,K,c-\alpha},\alpha)\cup \\
&\text{addCost}([\![A(x,p/_,y)]\!]_{G,K,c-\alpha},\alpha)\cup \\
&\text{addCost}([\![A(x,\epsilon,y)]\!]_{G,K,c-\beta},\beta)\cup \\
&\text{addCost}([\![A(x,_,y)]\!]_{G,K,c-\gamma},\gamma))
\end{aligned}
$$

Combining the last two into a single equation, we therefore need to show that:

$$
\begin{gathered}
[\![\langle x,p,y\rangle]\!]_{G,K}\cup \\
\text{addCost}(\textstyle\bigcup_{t'\in rew(A(x,_/p,y))_{c-\alpha}}[\![t']\!]_{G,K},\alpha)\cup \\
\text{addCost}(\textstyle\bigcup_{t'\in rew(A(x,p/_,y))_{c-\alpha}}[\![t']\!]_{G,K},\alpha)\cup \\
\text{addCost}(\textstyle\bigcup_{t'\in rew(A(x,\epsilon,y))_{c-\beta}}[\![t']\!]_{G,K},\beta)\cup \\
\text{addCost}(\textstyle\bigcup_{t'\in rew(A(x,_,y))_{c-\gamma}}[\![t']\!]_{G,K},\gamma) \\
= \\
[\![\langle x,p,y\rangle]\!]_{G,K}\cup \\
\text{addCost}([\![A(x,_/p,y)]\!]_{G,K,c-\alpha},\alpha)\cup \\
\text{addCost}([\![A(x,p/_,y)]\!]_{G,K,c-\alpha},\alpha)\cup \\
\text{addCost}([\![A(x,\epsilon,y)]\!]_{G,K,c-\beta},\beta)\cup \\
\text{addCost}([\![A(x,_,y)]\!]_{G,K,c-\gamma},\gamma))
\end{gathered}
$$

Given Corollary 5.1 and dropping the addCost functions, it is sufficient to show that all the following equations hold individually:

$$[\![\langle x,p,y\rangle]\!]_{G,K} = [\![\langle x,p,y\rangle]\!]_{G,K} \tag{5.2.7}$$

$$\bigcup_{t'\in rew(A(x,_/p,y))_{c-\alpha}} [\![t']\!]_{G,K} = [\![A(x,_/p,y)]\!]_{G,K,c-\alpha} \tag{5.2.8}$$

$$\bigcup_{t'\in rew(A(x,p/_,y))_{c-\alpha}} [\![t']\!]_{G,K} = [\![A(x,p/_,y)]\!]_{G,K,c-\alpha} \tag{5.2.9}$$

$$\bigcup_{t'\in rew(A(x,\epsilon,y))_{c-\beta}} [\![t']\!]_{G,K} = [\![A(x,\epsilon,y)]\!]_{G,K,c-\beta} \tag{5.2.10}$$

$$\bigcup_{t' \in rew(A(x,_,y))_{c-\gamma}} [\![t']\!]_{G,K} = [\![A(x,_,y)]\!]_{G,K,c-\gamma} \tag{5.2.11}$$

Equation 5.2.7 is trivially true. Equations 5.2.10 and 5.2.11 hold since on the LHS, $rew(A(x,\epsilon,y))_{c-\beta}$ and $rew(A(x,_,y))_{c-\gamma}$ contain only (x,ϵ,y) and $(x,_,y)$ respectively, for any $c-\beta, c-\gamma \geq 0$, and on the RHS, by the semantics of approximation, we know that $[\![A(x,\epsilon,y)]\!]_{G,K} = [\![x,\epsilon,y]\!]_{G,K}$ and $[\![A(x,_,y)]\!]_{G,K} = [\![x,_,y]\!]_{G,K}$.

For Equation 5.2.8, considering the semantics of approximation with concatenation of paths, the LHS of the equation can be rewritten in the following way since we know that we will not apply any step of approximation to $A(x,_,z)$:

$$([\![\langle x,_,z\rangle]\!]_{G,K}) \bowtie (\bigcup_{t' \in rew(A(z,p,y))_{c-\alpha}} [\![t']\!]_{G,K})$$

Applying Lemma 5.2 we can rewrite the RHS of 5.2.8 to:

$$\text{CostProj}(([\![A(x,_,z)]\!]_{G,K,c-\alpha} \bowtie [\![A(z,p,y)]\!]_{G,K,c-\alpha}), c-\alpha)$$

It is possible to drop the outer CostProj since the query $[\![A(x,_,z)]\!]_{G,K,c-\alpha}$ returns only mappings with associated cost 0, obtaining:

$$[\![A(x,_,z)]\!]_{G,K,c-\alpha} \bowtie [\![A(z,p,y)]\!]_{G,K,c-\alpha}$$

Therefore we need to show that the following holds:

$$([\![\langle x,_,z\rangle]\!]_{G,K}) \bowtie (\bigcup_{t' \in rew(A(z,p,y))_{c-\alpha}} [\![t']\!]_{G,K}) = [\![A(x,_,z)]\!]_{G,K,c-\alpha} \bowtie [\![A(z,p,y)]\!]_{G,K,c-\alpha}$$

Given Corollary 5.1 it is sufficient to show that:

$$[\![\langle x,_,z\rangle]\!]_{G,K} = [\![A(x,_,z)]\!]_{G,K,c-\alpha} \tag{5.2.12}$$

$$\bigcup_{t' \in rew(A(z,p,y))_{c-\alpha}} [\![t']\!]_{G,K} = [\![A(z,p,y)]\!]_{G,K,c-\alpha} \tag{5.2.13}$$

Equation 5.2.12 holds by similar reasoning to Equation 5.2.11. Equation 5.2.13 holds by the induction hypothesis.

Equation 5.2.9 can be shown to hold by similar reasoning to Equation 5.2.8. We conclude that Equation 5.2.6 holds for every $c \geq 0$.

(b) Relaxation. For relaxation, we show the following by induction on the cost c:

$$[\![R(x,p,y)]\!]_{G,K,c} = \bigcup_{t' \in rew(R(x,p,y))_c} [\![t']\!]_{G,K} \quad (5.2.14)$$

The induction hypothesis is that 5.2.14 holds for $c = i\alpha + j\beta + k\gamma + l\delta$ for all $i, j, k, l \geq 0$, where α, β, γ, δ are the costs of the four relaxation operations arising from rules 2, 4, 5 and 6, respectively, of Figure 3.1. We have already shown the base case of $i = j = k = l = 0$. We now show that 5.2.14 holds when one of, i, j, k or l is incremented by 1. Similarly to the reasoning for approximation in part (a), we need to show the following, where $isp(p)$ is a function that returns all the immediate super properties of p, and $isc(c)$ is a function that returns all the immediate super classes of c[2]:

$$\begin{gathered}
[\![\langle x,p,y\rangle]\!]_{G,K} \cup \\
\textstyle\bigcup_{p' \in isp(p)} [\![R(x,p',y)]\!]_{G,K,c-\alpha} \cup \\
\textstyle\bigcup_{a \in isc(y)} [\![R(x,type,a)]\!]_{G,K,c-\beta} \cup \\
\textstyle\bigcup_{a \in isc(x)} [\![R(a,type^-,y)]\!]_{G,K,c-\beta} \cup \\
[\![R(x,type,a)]\!]_{G,K,c-\gamma} \cup \\
[\![R(a,type^-,y)]\!]_{G,K,c-\delta} \\
= \\
[\![\langle x,p,y\rangle]\!]_{G,K} \cup \\
\textstyle\bigcup_{p' \in isp(p)} (\bigcup_{t' \in rew(R(x,p',y))_{c-\alpha}} [\![t']\!]_{G,K}) \cup \\
\textstyle\bigcup_{a \in isc(y)} (\bigcup_{t' \in rew(R(x,type,a))_{c-\beta}} [\![t']\!]_{G,K}) \cup \\
\textstyle\bigcup_{a \in isc(x)} (\bigcup_{t' \in rew(R(a,type^-,y))_{c-\beta}} [\![t']\!]_{G,K}) \cup \\
\textstyle\bigcup_{t' \in rew(R(x,type,a))_{c-\gamma}} [\![t']\!]_{G,K} \cup \\
\textstyle\bigcup_{t' \in rew(R(a,type^-,y))_{c-\delta}} [\![t']\!]_{G,K}
\end{gathered}$$

Given Corollary 5.1 it is sufficient to show that the following equations hold individually:

[2]Here we apply rule 4 only if p is *type* and x is a constant, or p is $type^-$ y is a constant. Also we apply rule 5 only if y is a constant, and we apply rule 6 only if x is a constant.

$$[\![\langle x,p,y\rangle]\!]_{G,K} = [\![\langle x,p,y\rangle]\!]_{G,K} \tag{5.2.15}$$

$$\bigcup_{p'\in isp(p)} [\![R(x,p',y)]\!]_{G,K,c-\alpha} = \bigcup_{p'\in isp(p)} (\bigcup_{t'\in rew(R(x,p',y))_{c-\alpha}} [\![t']\!]_{G,K}) \tag{5.2.16}$$

$$\bigcup_{a\in isc(y)} [\![R(x,type,a)]\!]_{G,K,c-\beta} = \bigcup_{a\in isc(y)} (\bigcup_{t'\in rew(R(x,type,a))_{c-\beta}} [\![t']\!]_{G,K}) \tag{5.2.17}$$

$$\bigcup_{a\in isc(x)} [\![R(a,type^{-},y)]\!]_{G,K,c-\beta} \bigcup_{a\in isc(x)} (\bigcup_{t'\in rew(R(a,type^{-},y))_{c-\beta}} [\![t']\!]_{G,K}) \tag{5.2.18}$$

$$[\![R(x,type,a)]\!]_{G,K,c-\gamma} = \bigcup_{t'\in rew(R(x,type,a))_{c-\gamma}} [\![t']\!]_{G,K} \tag{5.2.19}$$

$$[\![R(a,type^{-},y)]\!]_{G,K,c-\delta} = \bigcup_{t'\in rew(R(a,type^{-},y))_{c-\delta}} [\![t']\!]_{G,K} \tag{5.2.20}$$

Equation 5.2.15 is trivially true. Equations (5.2.16-5.2.20) can be rewritten as the general case of the induction hypothesis for some $c \geq 0$. Therefore equations (5.2.16-5.2.20) hold by the induction hypothesis. We conclude that Equation 5.2.14 holds for every $c \geq 0$.

(2) Now we need to show that approxRegex and relaxTriplePattern are sound and complete for triple patterns containing any regular expression. In part (1) we have demonstrated soundness and completeness for triple patterns containing a single predicate, p:

$$[\![A/R(x,p,y)]\!]_{G,K,c} = \textstyle\bigcup_{t'\in rew(A/R(x,p,y))_c} [\![t']\!]_{G,K}$$

This is our base case. We now show soundness and completeness by structural induction, considering the three different operators used to construct a regular expression: concatenation, disjunction and Kleene-Closure.

(a) Concatenation. The induction hypothesis is that the following equations hold for any regular expressions P_1 and P_2:

$$[\![A/R(x,P_1,y)]\!]_{G,K,c} = \bigcup_{t'\in rew(A/R(x,P_1,y))_c} [\![t']\!]_{G,K} \tag{5.2.21}$$

$$[\![A/R(x,P_2,y)]\!]_{G,K,c} = \bigcup_{t'\in rew(A/R(x,P_2,y))_c} [\![t']\!]_{G,K} \tag{5.2.22}$$

We now show that the following holds:

$$[\![A/R(x,P_1/P_2,y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,P_1/P_2,y))_c} [\![t']\!]_{G,K} \tag{5.2.23}$$

When the approxRegex and relaxTriplePattern functions are passed as input a triple pattern of the form $A/R(x,P_1/P_2,y)$, this is split into two triple patterns: $A/R(x,P_1,z)$ and $A/R(z,P_2,y)$. Both of these triple patterns are passed recursively to the approxRegex and relaxTriplePattern functions which return two sets of triple patterns that will be joined with the AND operator. Therefore the RHS of Equation 5.2.23 can be written in the following way:

$$\text{CostProj}(\bigcup_{t' \in rew(A/R(x,P_1,z))_c} [\![t']\!]_{G,K} \bowtie \bigcup_{t' \in rew(A/R(z,P_2,y))_c} [\![t']\!]_{G,K}, c)$$

Given the semantics of approximation and relaxation with concatenation of paths, the LHS of Equation 5.2.23 can be written as follows:

$$\text{CostProj}([\![A/R(x,P_1,z)]\!]_{G,K} \bowtie [\![A/R(z,P_2,y)]\!]_{G,K}), c)$$

which by Lemma 5.2 is equal to:

$$\text{CostProj}([\![A/R(x,P_1,z)]\!]_{G,K,c} \bowtie [\![A/R(z,P_2,y)]\!]_{G,K,c}, c)$$

We therefore need to show that:

$$\text{CostProj}([\![A/R(x,P_1,z)]\!]_{G,K,c} \bowtie [\![A/R(z,P_2,y)]\!]_{G,K,c}, c) =$$
$$\text{CostProj}(\bigcup_{t' \in rew(A/R(x,P_1,z))_c} [\![t']\!]_{G,K} \bowtie \bigcup_{t' \in rew(A/R(z,P_2,y))_c} [\![t']\!]_{G,K}, c)$$

Dropping the outer CostProj operators on both sides of the above equation and applying Corollary 5.1, it is sufficient to show that the following equations hold individually:

$$[\![A/R(x,P_1,z)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,P_1,z))_c} [\![t']\!]_{G,K}$$
$$[\![A/R(z,P_2,y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(z,P_2,y))_c} [\![t']\!]_{G,K}$$

These equations hold by the induction hypothesis. Therefore Equation 5.2.23 holds.

(b) Disjunction. Similarly to concatenation, our induction hypothesis is that Equations 5.2.21 and 5.2.22 hold for any regular expressions P_1 and P_2. We now show that the following equation holds:

$$[\![A/R(x, P_1|P_2, y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,P_1|P_2,y))_c} [\![t']\!]_{G,K} \qquad (5.2.24)$$

When the approxRegex and relaxTriplePattern functions are passed as input a triple pattern of the form $A/R(x, P_1|P_2, y)$, this is split into two triple patterns: $A/R(x, P_1, y)$ and $A/R(x, P_2, y)$. Both of these triple patterns are passed recursively to the approxRegex and relaxTriplePattern functions which will return two sets of triple patterns that will be combined with the UNION operator. Therefore the RHS of Equation 5.2.24 can be written as follows:

$$\bigcup_{t' \in rew(A/R(x,P_1,y))_c} [\![t']\!]_{G,K} \cup \bigcup_{t' \in rew(A/R(x,P_2,y))_c} [\![t']\!]_{G,K}$$

Given the semantics of approximation and relaxation with disjunction of paths, we can write the LHS of Equation 5.2.24 as follows:

$$\text{CostProj}([\![A/R(x, P_1, y)]\!]_{G,K} \cup [\![A/R(x, P_2, y)]\!]_{G,K}, c)$$

which by Lemma 5.2 is equal to:

$$[\![A/R(x, P_1, y)]\!]_{G,K,c} \cup [\![A/R(x, P_2, y)]\!]_{G,K,c}$$

We therefore need to show that:

$$[\![A/R(x, P_1, y)]\!]_{G,K,c} \cup [\![A/R(x, P_2, y)]\!]_{G,K,c} =$$
$$\bigcup_{t' \in rew(A/R(x,P_1,y))_c} [\![t']\!]_{G,K} \cup \bigcup_{t' \in rew(A/R(x,P_2,y))_c} [\![t']\!]_{G,K}$$

By Corollary 5.1 it is sufficient to show that:

$$[\![A/R(x, P_1, y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,P_1,y))_c} [\![t']\!]_{G,K}$$
$$[\![A/R(x, P_2, y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,P_2,y))_c} [\![t']\!]_{G,K}$$

These equations hold by the induction hypothesis. Therefore Equation 5.2.24 holds.

(c) Kleene-Closure. Our induction hypothesis in this case is that

$$[\![A/R(x, P^n, y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,P^n,y))_c} [\![t']\!]_{G,K}$$

for any regular expression P and any $n \geq 0$, where P^n denotes the regular expression $P/P/\ldots/P$ in which P appears n times. For the base case of $n = 0$, where $P^n = \epsilon$, the equation is trivially true since $rew(A(x, \epsilon, y))_c$ contains only the query (x, ϵ, y). We now show that the following holds:

$$[\![A/R(x, P^{n+1}, y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,P^{n+1},y))_c} [\![t']\!]_{G,K} \tag{5.2.25}$$

The approxRegex function rewrites an approximated triple pattern containing the property path P^{n+1} in the following way: $A(x, P^i/P/P^j, y)$ for arbitrarily chosen i, j satisfying $i + j = n$. It then splits this into three triple patterns, $A(x, P^i, z_1)$, $A(z_1, P, z_2)$ and $A(z_2, P^j, y)$. Therefore, for approximation, the RHS of 5.2.25 becomes:

$$\begin{aligned} \text{CostProj}(&\bigcup_{t' \in rew(A(x,P^i,z_1))_c} [\![t']\!]_{G,K} \bowtie \\ &\bigcup_{t' \in rew(A(z_1,P,z_2))_c} [\![t']\!]_{G,K} \bowtie \\ &\bigcup_{t' \in rew(A(z_2,P^j,y))_c} [\![t']\!]_{G,K}, c) \end{aligned} \tag{5.2.26}$$

The CostProj operator here captures the behaviour of the rewriting algorithm that excludes queries with associated cost greater than c.

Knowing that $\mathcal{L}(P^i/P/P^j) = \mathcal{L}(P^{n+1})$ and by the semantics of approximation with concatenation of paths, we can write the LHS of 5.2.25 as:

$$\text{CostProj}([\![A(x, P^i, z_1)]\!]_{G,K} \bowtie [\![A(z_1, P, z_2)]\!]_{G,K} \bowtie [\![A(z_2, P^j, y)]\!]_{G,K}, c)$$

Applying Lemma 5.2, this can be further rewritten as:

$$\begin{aligned} \text{CostProj}(&[\![A(x, P^i, z_1)]\!]_{G,K,c} \bowtie \\ &[\![A(z_1, P, z_2)]\!]_{G,K,c} \bowtie \\ &[\![A(z_2, P^j, y)]\!]_{G,K,c}, c) \end{aligned} \tag{5.2.27}$$

Combining 5.2.27 and 5.2.26 and removing the outer CostProj operator on both hand sides we therefore need to show that:

$$[\![A(x,P^i,z_1)]\!]_{G,K,c} \bowtie [\![A(z_1,P,z_2)]\!]_{G,K,c} \bowtie [\![A(z_2,P^j,y)]\!]_{G,K,c} =$$
$$\bigcup_{t' \in rew(A(x,P^i,z_1))_c} [\![t']\!]_{G,K} \bowtie \bigcup_{t' \in rew(A(z_1,P,z_2))_c} [\![t']\!]_{G,K} \bowtie \bigcup_{t' \in rew(A(z_2,P^j,y))_c} [\![t']\!]_{G,K}$$

By Corollary 5.1 it is sufficient to show that the following equations hold individually:

$$[\![A(x,P^i,z_1)]\!]_{G,K,c} = \bigcup_{t' \in rew(A(x,P^i,z_1))_c} [\![t']\!]_{G,K} \tag{5.2.28}$$

$$[\![A(z_1,P,z_2)]\!]_{G,K,c} = \bigcup_{t' \in rew(A(z_1,P,z_2))_c} [\![t']\!]_{G,K} \tag{5.2.29}$$

$$[\![A(z_2,P^j,y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A(z_2,P^j,y))_c} [\![t']\!]_{G,K} \tag{5.2.30}$$

Equations (5.2.28,5.2.29,5.2.30) hold by the induction hypothesis since i and j are both less than or equal to n; therefore Equation 5.2.25 holds.

The same reasoning applies for the relaxTriplePattern function applied to a relaxed triple pattern containing the property path P^{n+1} on the RHS of 5.2.25, with the difference that it rewrites the triple pattern in 3 different ways: $R(x, P^i/P/P^j, y)$ (for arbitrarily chosen i, j satisfying $i + j = n$, $i > 0$ and $j > 0$), $R(x, P/P^n, y)$ and $R(x, P^n/P, y)$. It is possible to apply the same steps of the proof as for approxRegex, noticing that $\mathcal{L}(P^i/P/P^j) = \mathcal{L}(P/P^n) = \mathcal{L}(P^n/P)$.

(3) General queries. We now show that the algorithm is sound and complete for any query that may contain approximation and relaxation. As the base case we have the case of a query comprising a single triple pattern, which has been shown in part (2) of the proof:

$$[\![A/R(x,P,y)]\!]_{G,K,c} = \bigcup_{t' \in rew(A/R(x,P,y))_c} [\![t']\!]_{G,K}$$

Consider now a query $Q = t$ AND Q' with t being an arbitrary triple pattern of the query Q. The induction hypothesis is that:

$$[\![Q']\!]_{G,K,c} = \bigcup_{Q'' \in rew(Q')_c} [\![Q'']\!]_{G,K} \tag{5.2.31}$$

We now show that the following holds.

$$[\![Q]\!]_{G,K,c} = \bigcup_{Q'' \in rew(Q)_c} [\![Q'']\!]_{G,K} \tag{5.2.32}$$

The LHS of Equation 5.2.32 is equivalent to the following by the semantics of the AND operator:

$$\text{CostProj}([\![t]\!]_{G,K} \bowtie [\![Q']\!]_{G,K}, c)$$

Applying Lemma 5.2 we can rewrite this as follows:

$$\text{CostProj}([\![t]\!]_{G,K,c} \bowtie [\![Q']\!]_{G,K,c}, c) \tag{5.2.33}$$

For the RHS of Equation 5.2.32 we have to consider two different cases: **(a)** t is a simple triple pattern, or **(b)** t contains the RELAX or APPROX operators.

(a) We can rewrite the RHS of Equation 5.2.32 to:

$$[\![t]\!]_{G,K} \bowtie \bigcup_{Q'' \in rew(Q')_c} [\![Q'']\!]_{G,K} \tag{5.2.34}$$

Combining 5.2.33 and 5.2.34 we need to show that:

$$\text{CostProj}([\![t]\!]_{G,K,c} \bowtie [\![Q']\!]_{G,K,c}, c) = [\![t]\!]_{G,K} \bowtie \textstyle\bigcup_{Q'' \in rew(Q')_c} [\![Q'']\!]_{G,K}$$

On the LHS, we are able to drop the outer CostProj operator and also the CostProj applied to the triple pattern t since $[\![t]\!]_{G,K}$ only returns mappings with cost 0. The resulting equation is as follows:

$$[\![t]\!]_{G,K} \bowtie [\![Q']\!]_{G,K,c} = [\![t]\!]_{G,K} \bowtie \textstyle\bigcup_{Q'' \in rew(Q')_c} [\![Q'']\!]_{G,K}$$

Applying Corollary 5.1 it is sufficient to show that the following equations hold individually:

$$[\![t]\!]_{G,K} = [\![t]\!]_{G,K} \tag{5.2.35}$$

$$[\![Q']\!]_{G,K,c} = \bigcup_{Q'' \in rew(Q')_c} [\![Q'']\!]_{G,K} \tag{5.2.36}$$

Equation 5.2.35 is trivially true and Equation 5.2.36 holds by the induction hypothesis. Therefore Equation 5.2.32 holds in the case of t being a simple triple pattern.

(b) If t contains the APPROX or RELAX operators then the RHS of 5.2.32 is equal to:

$$\text{CostProj}(\bigcup_{t' \in rew(t)_c} [\![t']\!]_{G,K} \bowtie \bigcup_{Q'' \in rew(Q')_c} [\![Q'']\!]_{G,K}, c)$$

(We have added the CostProj operator on the equation in order to capture the behaviour of the rewriting algorithm that excludes queries with associated cost greater than c). Therefore, combining this with 5.2.33, we need to show that:

$$\text{CostProj}([\![t]\!]_{G,K,c} \bowtie [\![Q']\!]_{G,K,c}, c) =$$
$$\text{CostProj}((\bigcup_{t' \in rew(t)_c} [\![t']\!]_{G,K}) \bowtie (\bigcup_{Q'' \in rew(Q')_c} [\![Q'']\!]_{G,K}), c)$$

Removing the CostProj from both hand sides of the equation and applying Corollary 5.1, it is sufficient to show that the following equations hold individually:

$$[\![t]\!]_{G,K,c} = \bigcup_{t' \in rew(t)_c} [\![t']\!]_{G,K} \tag{5.2.37}$$

$$[\![Q']\!]_{G,K,c} = \bigcup_{Q'' \in rew(Q')_c} [\![Q'']\!]_{G,K} \tag{5.2.38}$$

Equation 5.2.37 holds since approxRegex and relaxTriplePattern are sound and complete as shown in step (2) of the proof. Equation 5.2.38 holds by the induction hypothesis. Therefore Equation 5.2.32 holds in the case of t containing the APPROX and RELAX operators.

□

5.3 Termination

We now show that the rewriting algorithm terminates after a finite number of steps. Moreover, by supplying a maximum cost parameter to the rewriting algorithm, we can predict the number of iterations needed to generate all the queries up to that cost:

Theorem 5.2. Given a query Q, ontology K and maximum query cost c, the rewriting algorithm terminates after at most $\lceil c/c' \rceil$ iterations, where c' is the lowest cost of an edit or relaxation operation, assuming that $c' > 0$.

Proof. The rewriting algorithm terminates when the set *oldGeneration* is empty. At the end of each cycle, *oldGeneration* is assigned the value of *newGeneration*. During each cycle, elements are added to *newGeneration* only when new queries are generated and have cost less than c, or already generated queries are generated again at a lesser cost than before (also less than c).

On each cycle of the algorithm, each query generated by *applyApprox* or *applyRelax* has cost at least c' plus the cost of the query from which it is generated. Since we start from query Q_0 which has cost 0, every query generated during the nth cycle will have cost greater than or equal to $n \cdot c'$. When $n \cdot c' > c$ the algorithm will not add any queries to *newGeneration*. Therefore, the algorithm will stop after at most $\lceil c/c' \rceil$ iterations. □

5.4 Discussion

We have presented an algorithm based on query rewriting that evaluates SPARQL^{AR} queries. We have proved that our query rewriting algorithm is correct and complete with respect to the semantics of SPARQL^{AR} queries up to a certain cost, and that it terminates after a finite number of steps.

Our approach differs from those in [42, 43] as we have devised a query evaluation algorithm that supports both approximation and relaxation, and our queries are ranked with respect to the costs associated with each APPROX and RELAX operation applied. In contrast, [42, 43] consider only query relaxation, and their query answers are ranked based on a score which is calculated by comparing the similarity between the generated query and the original query. In contrast to [73], our approach makes use of the Jena API to evaluate the SPARQL queries generated by the rewriting algorithm. Our query evaluation can therefore be modified to use other, possibly more efficient, SPARQL APIs due to its use of this API. In contrast, Selmer et al. developed their query evaluation implementation so that it directly manipulates the data graph, which is stored in the Sparksee[3] graph database.

In the next chapter we explore the performance of query evaluation in SPARQL^{AR}

[3] `http://www.sparsity-technologies.com/`

by executing three sets of queries over three datasets. We describe our implementation of SPARQL^{AR}, present an optimisation based on a caching technique, and show how this impacts on query evaluation performance.

Chapter 6

Performance Study

To test the efficiency of our rewriting algorithm we have developed a prototype that is able to execute SPARQL^{AR} queries. This chapter begins by describing the implementation and query processing performance of this prototype in Section 6.1. In Section 6.2 we describe an optimisation technique based on the pre-computation of parts of queries. In Section 6.3 we compare the query processing performance of the SPARQL^{AR} evaluator with and without the pre-computation optimisation by executing a number of queries over three different datasets of varying sizes.

A different flexible query evaluation implementation can be found in [73] where Selmer et al. describe a system to evaluate flexible queries by directly manipulating the data graph. They present a performance study over two datasets: L4All, a synthetic dataset that contains chronological records of learning and work episodes of users, and the YAGO dataset which will be described later in this chapter. They describe two optimisation techniques: a distance-aware query evaluation mode that causes their evaluation algorithm to avoid navigating parts of the graph that might return answers at a cost higher than the cost required; and a second optimisation that generates multiple NFAs from queries containing the alternation operator $||$ instead of a single NFA.

Similarly to our prototype, the flexible query implementation in [42] uses the Jena API. Their performance study uses the LUBM benchmark and they show that they are able to return the top 150 answers efficiently. In contrast, we do not limit the number of answers, and in our performance study we return all answers with

associated cost up to 3. Also, for our study we use three datasets: LUBM, YAGO and DBpedia.

6.1 Implementation

A prototype that implements the SPARQL^{AR} query evaluation algorithms described in the previous chapters has been implemented in Java. As shown in Figure 6.1, the system architecture consists of three layers: the GUI layer, the System layer, and the Data layer. The GUI layer supports user interaction with the system, allowing queries to be submitted, costs of the edit and relaxation operators to be set, data sets and ontologies to be selected, and query answers to be incrementally displayed to the user. The System layer comprises three components: the Utilities, containing classes providing the core logic of the system; the Domain Classes, providing classes relating to the construction of SPARQL^{AR} queries; and the Query Evaluator in which query rewriting, optimisation and evaluation are undertaken. The Data layer connects the system to the selected RDF dataset and ontology using the Jena API[1]; Jena library methods are used to execute SPARQL queries over the RDF dataset and to load the ontology into memory. RDF datasets are stored as a TDB database[2] and RDF-Schemas can be stored in multiple RDF formats (e.g. Turtle, N-Triple, RDF/XML).

User queries are submitted to the GUI, which invokes a method of the *SPARQL^{AR} Parser* that parses the query string and constructs an object of the class *SPARQL^{AR} Query*. The parser was built using the Java Compiler-Compiler[3] tool, that generates Java parsers from a generative grammar.

The GUI invokes the *Data/Ontology Loader*, which creates an object of the class *Data/Ontology Wrapper*, and the *Approx/Relax Constructor* which creates objects of the classes *Approx* and *Relax*. Once these objects have been initialised, they are passed to the Query Evaluator by invoking the *Rewriting Algorithm*. This generates

[1] `https://jena.apache.org`

[2] `https://jena.apache.org/documentation/tdb/.`

[3] `https://javacc.java.net/`

the set of SPARQL queries to be executed over the RDF dataset. The set of queries is passed to the *Evaluator*, which interacts with the *Optimiser* and the *Cache* to improve query performance — we discuss the Optimiser and the Cache in the next section. The *Evaluator* uses the *Jena Wrapper* to invoke Jena library methods for executing SPARQL queries over the RDF dataset. The *Jena Wrapper* also gathers the query answers and passes them to the *Answer Wrapper*. Finally, the answers are displayed by the *Answers Window*, in ranked order.

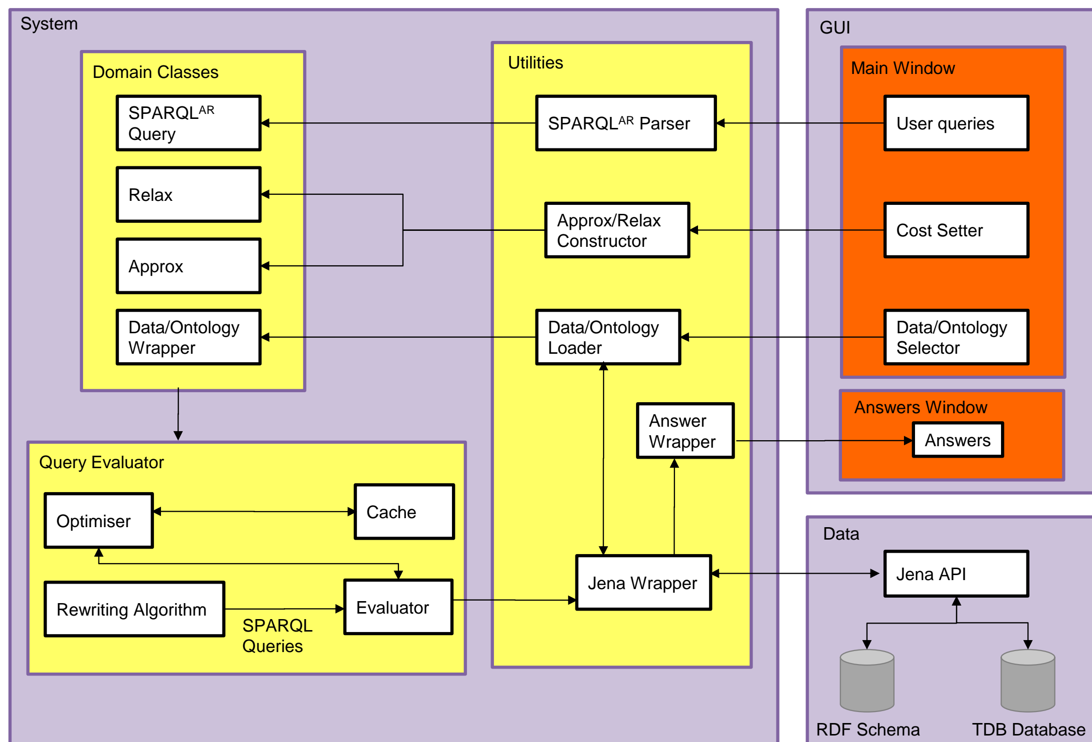


Figure 6.1: SPARQL^{AR} system architecture

6.2 Pre-Computation Optimisation

The rewriting algorithm defined in Section 5.1 generates in general an exponential number of queries with respect to the size of the initial query. We propose an optimisation technique that pre-computes parts of these queries to avoid the re-evaluation of some sub-queries.

Algorithm 8: MaxSet Function

input : a query Q.
output: Set of subqueries of Q, QS.
$QS := \emptyset$;
$Exact :=$ subset of triple patterns of Q that are not labelled with APPROX or RELAX;
foreach q *such that* $Exact \subseteq q \subset Q$ *and* q *is connected* **do**
  $QS := QS \cup q$;
return QS;

In contrast to our first version of this optimisation (see [31]) we do not explicitly separate the query into two parts, the exact and the approx/relax part, since this might lead to the calculation of the Cartesian product of sets of pre-computed answers. Instead, for each query Q generated by the rewriting algorithm, we generate a set of queries QS.

To compute this set QS we use the MaxSet function listed in Algorithm 8. Each query q generated by the MaxSet function contains all the triple patterns in the original query Q that are not approximated or relaxed. It also contains triples that are approximated or relaxed such that the following property holds: every $q' \subseteq q$ contains at least one triple pattern that shares a variable with a triple from $q - q'$. If a query q has this property then we say that it is *connected*. Therefore, when evaluating the query q, it is never the case that given $q', q'' \subseteq q$ and $q' \cup q'' = q$, we have that $eval(q', G) \bowtie eval(q'', G) = eval(q', G) \times eval(q'', G)$. In other words, we never have to calculate the Cartesian product of two sets of query answers.

Example 6.1. Given query $Q = (x_1, p_1, x_2)$ AND $(x_1, p_2, x_3)_A$ AND $(x_4, p_3, x_2)_R$ AND $(x_4, p_4, x_5)_R$, MaxSet returns the following set of subsets of triple patterns:

$$\{\{(x_1, p_1, x_2)\}, \tag{6.2.1}$$

$$\{(x_1, p_1, x_2), (x_1, p_2, x_3)_A\}, \tag{6.2.2}$$

$$\{(x_1, p_1, x_2), (x_4, p_3, x_2)_R\}, \tag{6.2.3}$$

$$\{(x_1, p_1, x_2), (x_1, p_2, x_3)_A, (x_4, p_3, x_2)_R\}, \tag{6.2.4}$$

$$\{(x_1, p_1, x_2), (x_4, p_3, x_2)_R, (x_4, p_4, x_5)_R\}\} \tag{6.2.5}$$

Each of these subsets contains the triple patterns that are not approximated or

relaxed (i.e. (x_1, p_1, x_2)). We also notice that triple pattern $(x_4, p_4, x_5)_R$ appears only together with $(x_4, p_3, x_2)_R$ as these share the variable x_4 and the variable x_2 is needed to connect to the exact triple pattern (x_1, p_1, x_2).

Algorithm 9 illustrates the optimised evaluation for SPARQL^{AR} queries using the MaxSet function. Each query generated by the rewriting algorithm is split into sub-queries QS by the MaxSet function. Each sub-query in QS is evaluated and stored in a cache. We notice that if parts of a query $q \in QS$ have already been computed then we reuse these answers to compute q. To avoid memory overflow, we place an upper limit on the size of the cache.

We compute the answers of a full query Q' arising from the rewriting algorithm using the *newEval* function which exploits the answers already computed and stored in the cache. If parts of the query have been already computed, *newEval* retrieves such answers and combines them to compute the final answers. We note that *new-Eval* might need to execute parts of Q' that are not available in the cache in order to compute the answer. We also note that, since Algorithm 8 loops over queries $Q' \subset Q$, the returned set does not include Q itself.

This technique will avoid the re-computation of sub-queries, thus speeding up the overall evaluation. In the next section we compare the performance of the rewriting algorithm with and without the pre-computation optimisation.

Algorithm 9: Flexible Query Evaluation – Optimised

---

input : Query Q; approx/relax max cost c; Graph G; Ontology K.
output: List M of mapping/cost pairs, sorted by cost.
$\pi_{\vec{w}} :=$ head of Q;
$cache := \emptyset$; `/* set of pairs of query/evaluation results */`
$M := \emptyset$;
foreach $\langle Q', cost \rangle \in rewrite(Q,c,K)$ **do**
  if *cache is not full* **then**
    foreach $Q'' \in MaxSet(Q')$ **do**
      if Q'' *is not in cache* **then**
        $cache := cache \cup \langle Q'', newEval(Q'', cache, G) \rangle$;
  foreach $\langle \mu, 0 \rangle \in newEval(Q', cache, G)$ **do**
    $M := M \cup \{\langle \mu, cost \rangle\}$;
return $\pi_{\vec{w}}(M)$;

6.3 Query Performance Study

For our query performance study we use three datasets: LUBM[4] (Lehigh University Benchmark), YAGO 3.0[5] and DBpedia[6]. The LUBM benchmark constructs datasets that describe universities, departments, professors, publications and students. By specifying the number of universities, the benchmark scales the size of the dataset. We consider three RDF datasets containing approximately 670,000, 1,300,000 and 6,700,000 triples respectively.

The YAGO dataset integrates data from Wikipedia, Geonames and Wordnet; it contains approximately 120 million triples, corresponding to a size of 10 GB in TDB format. The YAGO dataset is generated by 'scraping' the Wikipedia pages, hence the information stored might lack accuracy. In fact, the YAGO dataset also stores the accuracy of each fact (we removed these accuracy measures from the dataset for our performance study). In contrast to DBpedia, the YAGO database is not updated often and some of the Wikipedia facts (which are stored in DBpedia) are not available in YAGO.

Finally, we use the DBpedia dataset which stores only the Wikipedia facts. It is updated regularly and contains approximately 4,230,000 URLs and 62 million triples.

We defined 7 SPARQL^{AR} queries for LUBM, 5 for DBpedia and 3 for YAGO. Each query contains between 1 and 5 triple patterns, some of which are approximated or relaxed in order to retrieve the answers that the user is looking for or that might be useful to the user.

The cost of each edit and relaxation operation is set to 1. We incrementally set the maximum cost of the query answers from 1 to 3. We stop the execution of a query if the system is not able to finish the evaluation within 8 hours.

[4] `http://swat.cse.lehigh.edu/projects/lubm/`

[5] `https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`

[6] `http://wiki.dbpedia.org/`

LUBM Evaluation

For the LUBM dataset we ran the following queries (the full description of the meaning of each query and the reason why a user might apply APPROX or RELAX are given in Appendix A.1):

Q1:

```
SELECT ?x ?t WHERE{
    ?x publicationAuthor/teacherOf ?c .
    ?x publicationAuthor/teachingAssistantOf ?c .
    RELAX(?x rdf:type Article)  . APPROX(?x title ?t)
}
```

Q2:

```
SELECT ?c WHERE{
    RELAX(GraduateStudent1 mastersDegreeFrom/hasAlumnus Student25) .
    GraduateStudent1  takesCourse ?c .
    Student25 takesCourse ?c
}
```

Q3:

```
SELECT ?x ?z WHERE{
    RELAX(?x doctoralDegreeFrom University1) .
    RELAX(?x worksFor University1) .
    ?x teacherOf ?c . APPROX(?z teachingAssistantOf ?c)
}
```

Q4:

```
SELECT * WHERE{
    ?z publicationAuthor AssociateProfessor3.
    APPROX(?z publicationAuthor/advisor AssociateProfessor3)
}
```

Q5:

```
SELECT ?s ?c WHERE{
    ?x rdf:type AssistantProfessor . ?x teacherOf ?c .
    ?s takesCourse ?c . RELAX(?s rdf:type UndergraduateStudent) .
    APPROX(?s address
        "UndergraduateStudent5@Department1.University0.edu")
}
```

Q6:

```
SELECT * WHERE{
    Student1 advisor/teacherOf ?c . Student1 takesCourse ?c .
    RELAX(?c rdf:type UndergraduateCourse)
}
```

Q7:

```
SELECT ?p WHERE{
    RELAX(ResearchGroup3 subOrganizationOf* ?x) .
    RELAX(?p rdf:type AssistantProfessor) . ?p worksFor ?x  .
    Publication0 publicationAuthor ?p
}
```

We start our analysis by showing the number of queries generated by the rewriting algorithm for each query, given maximum costs 1, 2 and 3 (Table 6.1). The number of queries generated depends on the number of triple patterns that have been relaxed and approximated, as well as on the length of the property path in each triple pattern, and may increase exponentially with respect to the maximum cost. In Table 6.1, we see that query Q_4 generates the most queries for cost 3 even though it has only one approximated triple pattern. This is because its property path has a concatenation of two URIs, thus the APPROX operator leads to a larger number of queries being generated by the rewriting algorithm. For query Q_3 the large number of rewritten queries is due to the higher number of triple patterns with an APPROX or RELAX operator (three) compared to the other queries (one or two). Overall we notice that the APPROX operator generates a higher number

of queries compared to RELAX, since the latter is applicable only if the ontology contains specific rules related to the query triple pattern.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	6	4	8	9	6	2	3
2	17	8	29	37	17	2	5
3	37	11	71	112	37	2	7

Table 6.1: LUBM. Number of queries generated by the rewriting algorithm, given maximum costs of 1, 2 and 3.

Table 6.2 shows the number of answers returned by each query executed over the three versions of the LUBM database, given maximum costs 0, 1, 2 and 3. The second column of the table indicates the three datasets constructed with the LUBM benchmark where $D1$, $D2$ and $D3$ contain 5, 10 and 50 universities respectively. In the rows relating to cost 0, the answers returned by the exact form of the query are displayed. We notice that all the queries in their exact form, except for Q_4, do not return any answer with respect to any of the datasets (the explanation for this is given in Appendix A.1).

All queries, except for Q_1 and Q_3 (and Q_2 when evaluated against D_1), return more answers after one step of approximation and relaxation. For query Q_1 the evaluator returns more answers after two steps of approximation. We also notice that after the first two steps (i.e. max cost 2) we do not retrieve more answers for any query. This is mainly due to the highly structured nature of the LUBM dataset which does not contain dense connections between the URIs. Hence, the number of answers is constrained by the constants appearing in the queries.

In Figures 6.2 to 6.4 we show the execution times of the queries against the 3 LUBM datasets, with and without the pre-computation optimisation (the precise timings are listed in Table B.1 and B.2 in Appendix B). We note that the times are shown on a logarithmic scale and that the bars with the diagonal stripes show the timings of the evaluation with the pre-computation optimisation.

If we consider dataset D1 we notice that the maximum cost impacts on the

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
0	D1	0	0	0	0	0	0	0
1	D1	0	0	0	18	1	1	1
2	D1	186	0	0	18	1	1	1
3	D1	186	0	0	18	1	1	1
0	D2	0	0	0	1	0	0	0
1	D2	0	1	0	13	1	1	1
2	D2	373	1	0	13	1	1	1
3	D2	373	1	0	13	1	1	1
0	D3	0	0	0	7	0	0	0
1	D3	0	3	0	13	1	1	1
2	D3	2036	3	0	13	1	1	1
3	D3	2036	3	0	13	1	1	1

Table 6.2: LUBM. Number of answers returned by each query, for every maximum cost, and every dataset.

execution time due to the increasing number of queries generated by the rewriting algorithm, especially for queries Q_1, Q_3, Q_4 and Q_5. This is mainly due to the presence of the APPROX operator that generates a greater number of queries as the maximum cost increases.

We can see that for dataset D1 there is a small improvement in the performance when using the pre-computation optimisation. This is mainly due to the fact that for small datasets Jena is able to execute SPARQL queries efficiently. If we consider the execution of queries Q_3 and Q_5 with maximum cost set to 2 and 3, we can see that there is a reduction in time of several seconds. In general, timings for queries that can be executed in less than a second do not improve using the pre-computation optimisation. Instead, the execution time increases due to the fact that the pre-computation needs to execute and store the answers of multiple sub-queries.

For dataset D2, the execution of queries Q_3, Q_4 and Q_5 cannot be completed

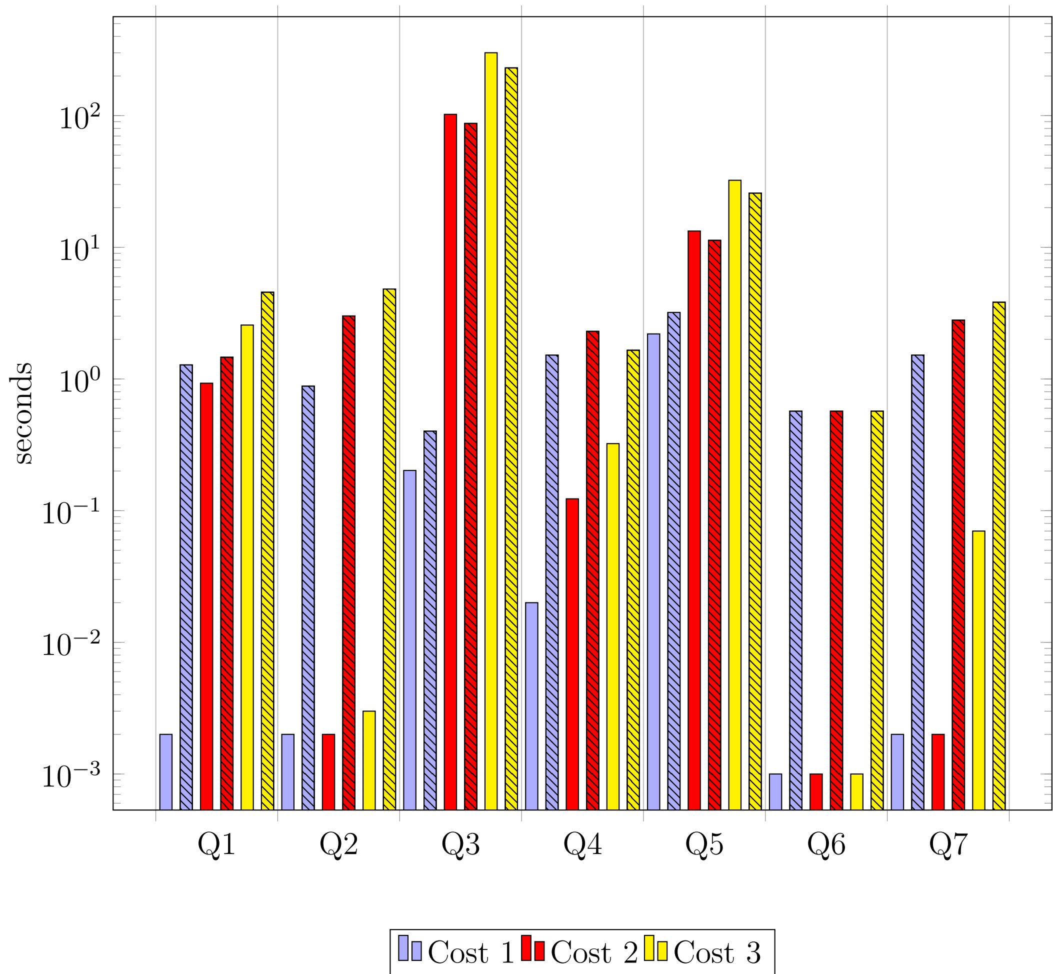


Figure 6.2: LUBM. Timings for dataset D1, with and without pre-computation optimisation.

within 8 hours without the pre-computation optimisation when maximum cost is set to 2 or 3. This is mainly due to the high number of queries that need to run and to the APPROX operator that generates queries containing _ that take a long time to evaluate. In fact after two steps of approximation, the approximated triple patterns of Q_3, Q_5 become respectively: (?z _/_ ?c), (?z _/_ "UndergraduateStudent5@Department1.University0.edu"). Evaluating such queries means that the query evaluator (Jena) has to find all possible paths of size two in the dataset, which can take a very long time due to the increased size of dataset D2. Query Q_4 has a longer property path and can generate even more complicated paths, such as (?z publicationAuthor/advisor/_/_ AssociateProfessor3), (?z publicationAuthor/_/advisor/_ AssociateProfessor3), etc. Notice that query Q_1 also contains the

APPROX operator, although the exact part of that query limits the values the variable ?x can be assigned to. (In query Q_1 the variable ?x appears in every triple. In contrast, in query Q_3 the variable ?z appears only in the approximated triple pattern. In query Q_5 all the constants are classes hence variable ?s can be instantiated to many constants.)

With dataset D3, the pre-computation optimisation is still not able to execute queries Q_3, Q_4 and Q_5 for maximum cost 1, 2 and 3. Similarly to dataset D2 the issue is due to the presence of the APPROX operator. In the next chapter we will present two additional optimisations that will enable the execution of the aforementioned queries within the 8 hour threshold.

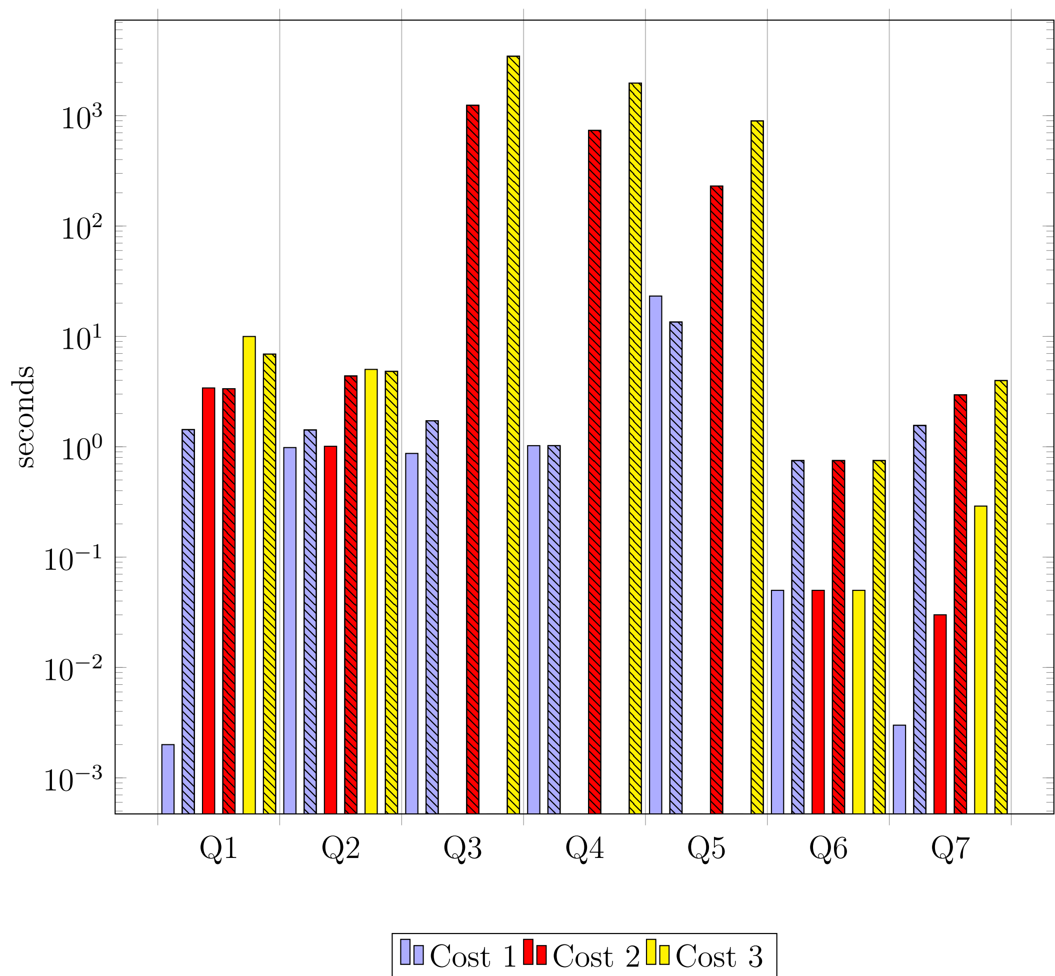


Figure 6.3: LUBM. Timings for dataset D2, with and without pre-computation optimisation.

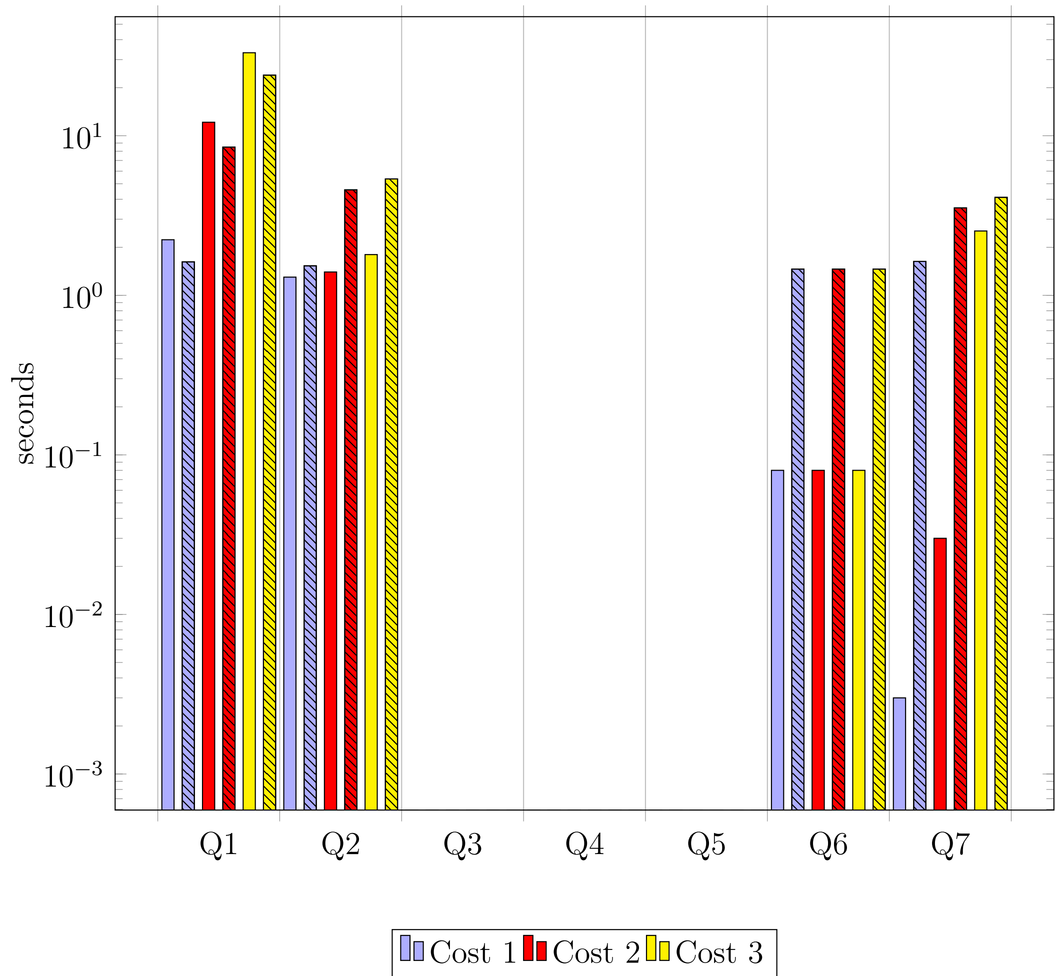


Figure 6.4: LUBM. Timings for dataset D3, with and without pre-computation optimisation.

DBpedia Evaluation

For the DBpedia dataset we ran the following queries (the full description of the meaning of each query and the reason why a user might apply APPROX or RELAX are given in Appendix A.2):

Q1:

```
SELECT ?y WHERE{
    APPROX(<The_Hobbit> subsequentWork* ?y). ?y rdf:type  Book
}
```

Q2:

```
SELECT ?x ?y WHERE{
```

```
    APPROX(?x albumBy <The_Rolling_Stones>) . ?x rdf:type Album .
    ?y album ?x . RELAX(?x recordLabel <London_Records>)
} group by ?x
```

Q3:

```
SELECT ?k ?d ?kd WHERE{
    APPROX(?k  diedIn <Battle_of_Poitiers>) .
    <Battle_of_Poitiers> date ?d . ?k deathDate  ?kd
}
```

Q4:

```
SELECT ?x ?kd WHERE{
    ?x subject Duelling_Fatalities . RELAX(?x deathDate "18xx-xx-xx") .
    RELAX(?x rdf:type Scientist)
}
```

Q5:

```
SELECT ?x ?f  WHERE{
    APPROX(12_Angry_Men_(1957_film) actor ?a) . ?x parent ?a .
    APPROX(?f actor ?x).
    RELAX(?x birthPlace New_York)
}
```

Table 6.3 shows the number of queries generated by the rewriting algorithm for these DBpedia queries. We notice that the query with the least number of rewritings is Q_4 since the APPROX operator does not appear in that query. Query Q_5 has the highest number of rewritings since it has two approximated triple patterns. Again, we notice how the APPROX operator leads to a high number of queries generated by the rewriting algorithm. Queries Q_2 and Q_3 generate the same number of rewritings since they each have only one approximated triple pattern that contains only one predicate.

Table 6.4 shows the number of answers returned by the queries given maximum costs 1, 2 and 3. The number of answers are limited for queries Q_1, Q_2 and Q_4 due

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	3	5	5	3	10
2	11	12	12	6	48
3	51	25	25	10	168

Table 6.3: DBpedia. Number of queries generated by the rewriting algorithm given maximum costs of 1, 2 and 3.

to the use of the *rdf:type* predicate that constrains the number of answers. For query Q_3 the constant URL <Battle_of_Poitiers> limits the number of answers since it has few connections with persons with a death date. Query Q_5 returns more answers than the other queries since it has no *rdf:type* predicate and moreover the RELAX operator replaces the constant <New_York> with a more generic class name.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
0	1	0	0	0	0
1	4	60	1	1	0
2	5	60	4	69	54
3	5	66	8	69	369

Table 6.4: DBpedia. Number of answers returned by each query, for every maximum cost.

We show in Figure 6.5 how the maximum cost affects the execution time of the queries. The system was not able to execute query Q_5 for cost 2 and 3 within 8 hours. This is mainly due to the high number of queries that needed to be executed. The pre-computation optimisation did have an impact on the evaluation time of several seconds for queries Q_1 with maximum cost 1 and 2, query Q_3 with maximum cost 1, 2 and 3, and query Q_5 with maximum cost 1. Since the pre-computation algorithm needs to run additional queries, the time decreases by approximately 10-20% only (see Figures B.10 and B.11 in Appendix B.2). Moreover, as the cache is limited in size this fills fast and the hit ratio decreases as the number of queries that need

to be executed increases. Further improvement in execution time could have been achieved by increasing the size of the cache so that more partial answers would have been stored and reused.

Similarly to the LUBM dataset, the pre-computation optimisation technique worsens the execution time for queries that can be executed in less than one second. It reduces the execution time slightly for queries that take more than five seconds.

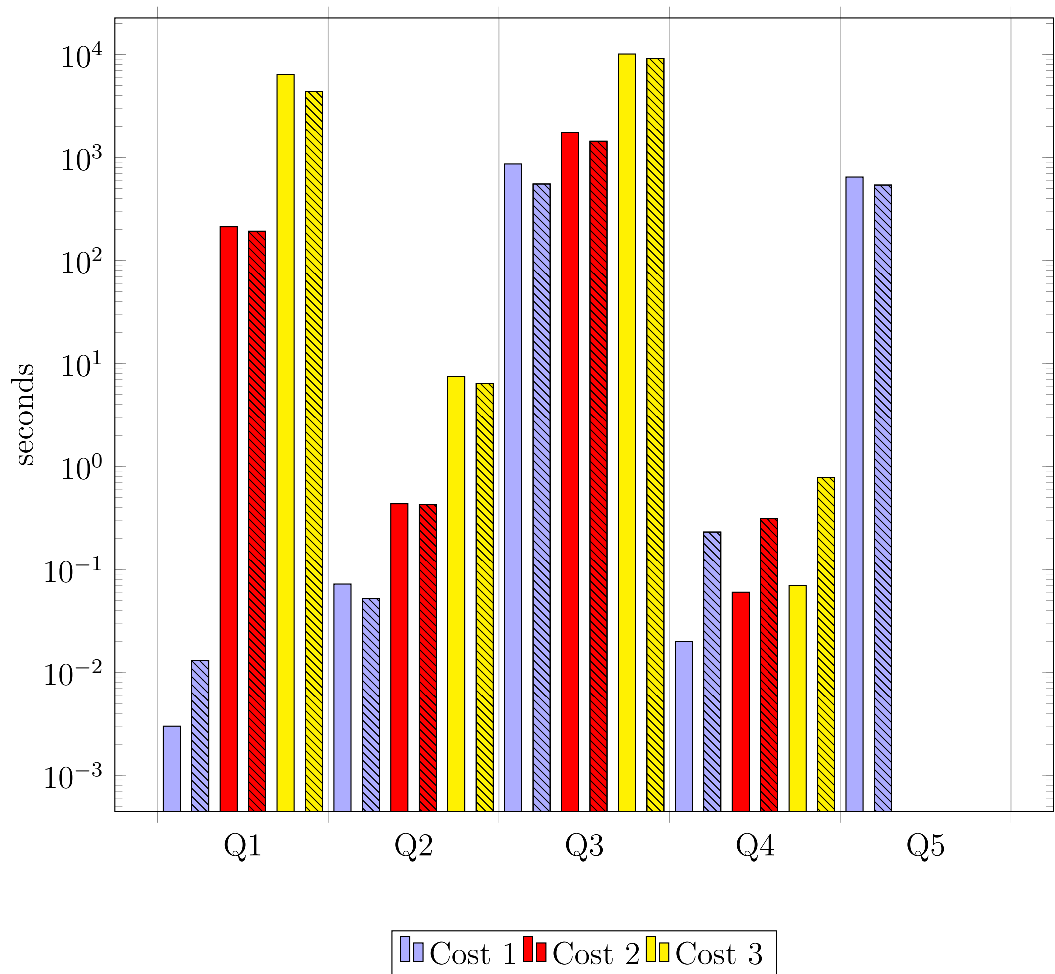


Figure 6.5: DBpedia. Timings, with and without pre-computation optimisation.

YAGO Evaluation

For the YAGO dataset we ran the following queries (the full description of the meaning of each query and the reason why a user might apply APPROX or RELAX are given in Appendix A.3):

Q1:

```
SELECT * WHERE{
    APPROX(<Battle_of_Waterloo>
        happenedIn/(hasLongitude|hasLatitude) ?x)
}
```

Q2:

```
SELECT * WHERE{
    ?x actedIn <Tea_with_Mussolini> . RELAX(?x hasFamilyName ?z)
}
```

Q3:

```
SELECT * WHERE{
    ?x rdf:type Event . ?x happenedOnDate ``1643-##-##'' .
    APPROX(?x happenedIn ``Berkshire'')
}
```

Table 6.5 shows the number of queries generated by the rewriting algorithm for the YAGO queries. Query Q_1 generates the greatest number of queries even though it has only one triple pattern which is approximated, because its single triple pattern contains a complex property path, that is the concatenation of a predicate with a disjunction of two predicates. Query Q_2 generates only 2 queries for costs 1, 2 and 3, since we can only apply the RELAX operator once to the triple pattern.

Max Cost	Q_1	Q_2	Q_3
1	12	2	5
2	60	2	12
3	199	2	25

Table 6.5: YAGO. Number of queries generated by the rewriting algorithm, given maximum costs of 1, 2 and 3.

As shown in Table 6.6 the number of answers returned by query Q_1 increases exponentially with respect to the maximum cost. This is due to the fact that the APPROX operator will find many more nodes reachable from node $\langle Battle_of_Waterloo \rangle$

when increasing the maximum cost. In contrast, for query Q_2 the number of answers does not increase as the maximum cost increases since the number of queries generated does not increase with respect to the maximum cost. Finally, for query Q_3 the number of answers is bounded by the presence of the rdf : $type$ predicate which constrains the variable $?x$ to be of type *Event*.

Max Cost	Q_1	Q_2	Q_3
0	0	4	0
1	1381	263	1
2	18584	263	1
3	116082	263	1

Table 6.6: YAGO. Number of answers returned by each query, for every maximum cost.

Figure 6.6 shows the execution times of the three YAGO queries. We see that Q_2 and Q_3 can be executed within a reasonable amount of time due to the small number of queries generated by the rewriting algorithm. On the other hand, query Q_1 with a maximum cost of 3 does not complete within the 8 hour threshold, because of the large number of queries generated by the rewriting algorithm.

The pre-computation optimisation does decrease the computation time of queries that take more than 5 seconds (Q_1 at maximum cost 2), but is still not able to execute query Q_1 with maximum cost 3. This is because Q_1 consists of a single triple pattern, hence there can be no caching of partial query results. Timings for queries Q_2 and Q_3 increase when applying the pre-computation optimisation because of the overhead of executing the sub-queries in order to cache their results.

6.4 Discussion

In this chapter we have presented the implementation of our SPARQL^{AR} prototype and have undertaken a performance analysis. We have discussed the execution times of three sets of queries against three datasets. We have seen that the number

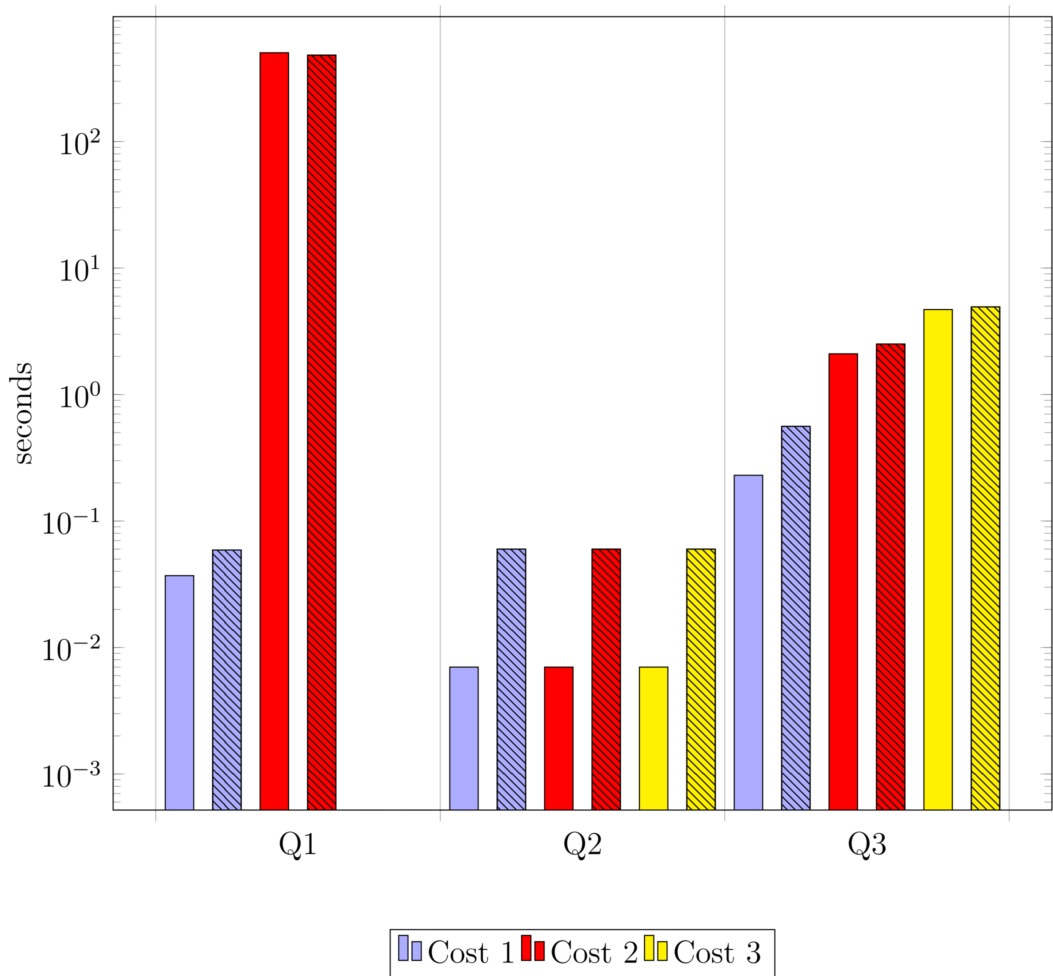


Figure 6.6: YAGO. Timings, with and without pre-computation optimisation.

of rewritten queries depends mainly on the presence of the APPROX operator and on the complexity of the property path in the approximated triple pattern.

In contrast to [73], we have described a pre-computation optimisation technique which is able to reduce the time for evaluating queries that have a large number of rewritings by caching partial answers. However, the pre-computation optimisation does not help with queries that have a lower number of rewritings, but instead increases the execution time somewhat. This is due to the additional time that the pre-computation algorithm takes to compute partial queries and store their answers in memory.

In the performance study of [42] the answers of the queries are limited to the top 150 making the query execution fast. Instead, in our performance study we tested the queries by setting the maximum cost to 3, hence the queries might return a

much higher number of answers. (In practice, our current prototype is able to limit the number of answers to be retrieved through the user interface; moreover, if the user is not satisfied by the results returned so far, she can resume the evaluation to retrieve the next set of answers.)

During our performance study we noticed that most of the queries generated by the rewriting algorithm were not returning any answers or were returning answers that had already been computed by previous queries. In the following chapter, we will discuss two more optimisations that aim to further improve the performance of the query evaluation: (1) removing queries that do not return any answers, and (2) not executing queries that return answers that have already been computed.

Chapter 7

Optimisations

In this chapter we investigate two optimisation techniques intended to improve the SPARQL^{AR} evaluation process: a summarisation optimisation and a query containment optimisation. By means of the summarisation optimisation technique we intend to achieve two goals: replace the symbol _ with a disjunction of URIs, and avoid the execution of queries that do not return any answers.

With the query containment optimisation technique we intend to reduce the number of queries that need to be executed by discarding those queries whose answer set is contained in the answer set of another query.

In Section 7.1 we discuss earlier work on RDF graph summarisation, we present our summarisation optimisation and show its impact on query evaluation timings. In Section 7.2 we discuss the query containment property for SPARQL^{AR} queries and test the effectiveness of using this for query optimisation. In Section 7.3 we combine both optimisations and evaluate their combined effect on query evaluation times.

7.1 Summarisation Optimisation

In Section 7.1.1 we first discuss related work on graph RDF summarisation. In Section 7.1.2 we present our RDF-graph summarisation approach and compare it with the work in [5] which uses a similar approach to generate RDF summaries. We show how our summaries can be exploited to improve the SPARQL^{AR} evaluation, in

particular the evaluation of queries containing the APPROX operator. In Section 7.1.3, we undertake a performance evaluation of our optimisation technique based on RDF-graph summarisation over the three datasets from Chapter 6.

7.1.1 Related Work

The problem of RDF summarisation is: given an input RDF-graph G, compute an RDF-graph S_G which summarises G, while being possibly orders of magnitude smaller than the original graph. The first work on RDF summarisation is by Cebirić et al. in [75]. Their summary helps in formulating and optimising queries, and it is able to predict if a query returns answers against the RDF-graph by querying the summary first. Starting from an RDF-graph G, the summarisation procedure collapses nodes that have at least one outgoing or ingoing edge with the same label into a single node. A disadvantage of this procedure is that it often collapses nodes which are not related at all. For example, if we consider that many nodes in a graph G have the outgoing edge *rdfs:label*, then all these nodes will collapse into one. In [75] the authors overcome this issue by partitioning the nodes of S_G by exploiting the *rdfs:label* predicate. However, they do not give more details on how the partitioning algorithm works.

In [16] a graph summary is constructed from multiple heterogeneous RDF data sources. The summary is computed by creating a graph where each node is a collection of nodes that share the same characteristics. They define two types of characteristic: *entity-based* and *class-based*. The entity-based characteristic aggregates nodes that share the same outgoing edges. The class-based characteristic instead aggregates nodes that are of the same class type. The method uses the summary to aid the user in posing queries to an RDF dataset by suggesting possible predicates and query structures.

Another type of summarisation is based on the property of *bi-simulation*[1]. In [35]

[1] A graph G is a bi-simulation of a graph G' if for each node a in G there is also a node a' in G such that a is bi-similar to a'. Nodes a and a' are bi-similar if for each outgoing edge from a to some node b, there exists an edge from a' to some node b' with the same label, such that b and b' are bi-similar.

the authors describe a system called TriAD that constructs a summary of the RDF-graph. The summarisation contains super-nodes which abstract a collection of nodes from the original graph, and super-edges that connect pairs of supernodes. A super-edge connects two super-nodes A and B if some node in A is connected to some node in B in the graph. We notice that the summarisation constructed is a bi-simulation of the original graph. The super-nodes are constructed by partitioning the graph using the METIS tool [50]. The nodes in this partition will be the super-nodes of the summarisation. Through the summarisation, TriAD does not need to explore the whole graph when querying. Instead, it traverses the summarisation to bind the query variables to the super-nodes. In this way it prunes nodes from the graph that will not be retrieved by the query.

The work in [21] describes a summarisation designed to index graphs. Their approach is similar to the $A(K)$–index [51], which indexes nodes that are bi-similar up to a path of length k in the graph[2], termed *local similarity.* The authors expand this concept by not fixing the value k. They increase the value of k incrementally and calculate the local similarity for each node.

In [15] the authors compare multiple summarisation techniques; in particular, they evaluate the trade-off between efficiency and precision, and the trade-off between precision and the ratio between the size of the original graph and the summary. The summarisations they analyse are based on node collapsing techniques, similar to the work in [75], and the bisimulation property which is also exploited in the work of [21, 35]. They define the following node collapsing techniques: two nodes collapse if they have the same set of attributes ($\sim_a$); or if they are of the same class type ($\sim_t$); or if they share at least one class type ($\sim_{st}$); or if they share the same set of incoming and outgoing attributes ($\sim_{ioa}$); or every possible combination of these.

They show that the $\sim_t$ technique produces the best size ratio but the worst precision. On the other hand, the $\sim_{ioat}$ technique, which combines $\sim_{ioa}$ and $\sim_t$, is slightly worse size-wise but is much more precise. In terms of efficiency and precision,

[2]Nodes a and a' are bi-similar up to k if for each outgoing edge from a to some node b, there exists an edge from a' to some node b' with the same label, such that b and b' are bi-similar up to $k-1$.

$\sim_t$ is the best candidate.

Finally, in [5] the authors reduce the size of an RDF-graph by agglomerating nodes into clusters using a combination of the bi-simulation property and a structure similarity property. They define nodes a and a' as being *strongly bi-similar* if for each outgoing edge from a to some node b, there exists an edge from a' to some node b' with the same label, such that b and b' are strongly bi-similar, and conversely for each outgoing edge from a' to b', there exists an edge from a to b with the same label such that b and b' are strongly bi-similar. They also define bi-similarity up to a value k: nodes a and a' are strongly bi-similar up to k if, for each outgoing edge from a to b, there exists an edge from a' to b' with the same label such that b and b' are strongly bi-similar up to $k-1$, and conversely for each outgoing edge from a' to b', there exists an edge from a to b with the same label such that b and b' are strongly bi-similar up to $k-1$. They analyse the quality and number of clusters generated using bi-simulation for 4 datasets. Their results show that for highly structured graphs strong bi-similarity can be achieved with $k = 3$.

In contrast to this previous work, our summarisation takes into account only the paths from the original RDF-graph G and we generate a deterministic automaton with only one initial state. Moreover, our summarisation is not used for indexing the nodes of G, but instead to simplify the evaluation of SPARQL queries. We construct a summarisation that may be an order of magnitude smaller than the RDF-graph G and that prevents the evaluation of queries that are known to return no answer, as well allowing us to replace the symbol $_$ by a disjunction of specific edge labels.

7.1.2 Our RDF-Graph Summary

We construct a summary of an RDF-graph, G, that is defined by an automaton, R. The summarisation satisfies $\mathcal{L}(G) \subseteq \mathcal{L}(R)$ where $\mathcal{L}(G)$ is the set of sequences of edge labels generated by G when G is viewed as an automaton in which each node is both a final and an initial state.

The summary automaton R that we define is able to recognise strings that represent paths in G up to a certain length $n \geq 2$. The states in R keep track of the last k transitions that have been traversed, for all $k < n$. So the automaton will

keep track only of the last $n-1$ states even if we have traversed more than $n-1$ states. The automaton R that recognises paths of G up to length n is constructed as follows:

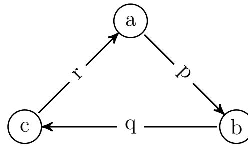
1. Initially the automaton R contains only one state, S, which is both initial and final.

2. For each $p_1p_2 \ldots p_k \in \mathcal{L}(G)$ with $k < n$, we add the new final states S_{p_1}, $\ldots$, $S_{p_1 \ldots p_k}$ to R, and also the new transitions: (S, p_1, S_{p_1}), $(S_{p_1}, p_2, S_{p_1p_2})$, $\ldots$, $(S_{p_1 \ldots p_{k-1}}, p_k, S_{p_1 \ldots p_k})$.

3. For each $p_1p_2 \ldots p_n \in \mathcal{L}(G)$ we add the transition $(S_{p_1 \ldots p_{n-1}}, p_n, S_{p_2 \ldots p_n})$.

Step 3 of the construction keeps track of the possible loops in the graph G. This is due to the fact that step 3 creates transitions between states that have been already generated by step 2. The implementation of the summarisation procedure is actually done via an automatically generated query that retrieves all the paths needed up to length n. The paths are then encoded in RDF format where the nodes are labelled as per procedure.

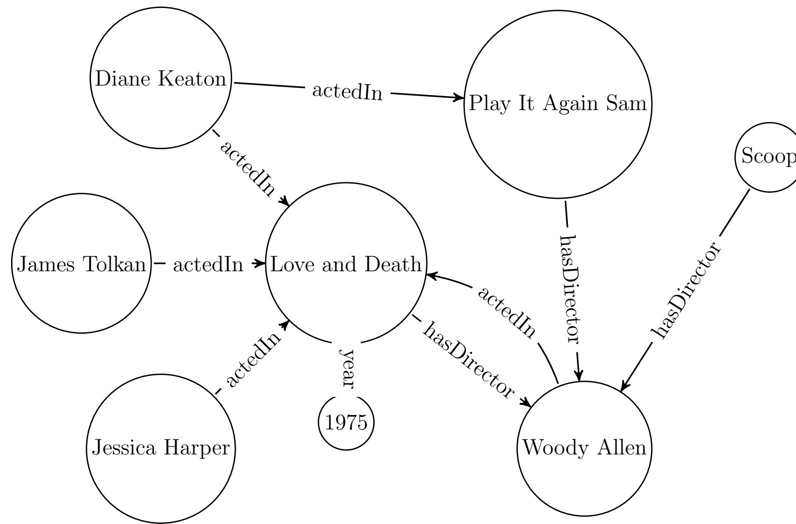
To illustrate, consider the following graph:



If we construct a summarisation R for $n = 2$ by using only steps 1 and 2 we would have the following transitions in R: (S, p, S_p), (S, q, S_q) and (S, r, S_r). Hence, the summarisation does not keep track of the loop $p/q/r$ in G. If we apply the third step of the procedure, R will now include also the following transitions: (S_r, p, S_p), (S_p, q, S_q) and (S_q, r, S_r). The final summarisation R keeps track of the loop $p/q/r$ in G.

We note that the theoretical size of the summarisation that we construct is $O(D^n)$ where D is the number distinct property labels of the graph. However, in general, RDF-graphs are sparse which results in a much smaller summarisation.

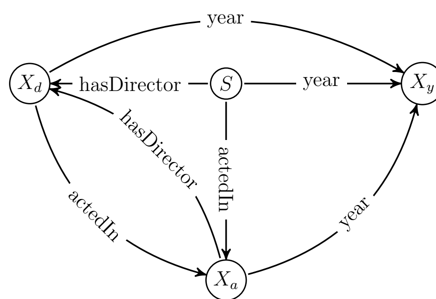
Example 7.1. To illustrate the construction of our summary, we consider the following graph G which describes a portion of a Film database:



The summarisation for $n = 2$ is constructed by retrieving the following distinct paths of up to length 2 from G:

`actedIn, year, hasDirector, actedIn/hasDirector,`
`hasDirector/actedIn, actedIn/year`

The resulting automaton R will be as follows (we use letters y, d and a to indicate the properties *year*, *hasDirector* and *actedIn* respectively):



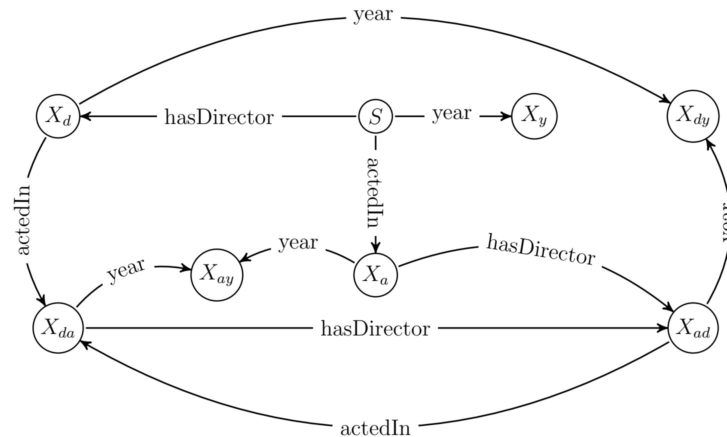
It is possible to verify that the above automaton recognises every string in $\mathcal{L}(G)$; moreover, in this particular case, $\mathcal{L}(R) = \mathcal{L}(G)$.

If we now consider $n = 3$, then we have the following paths of up to length 3:

`actedIn,year, hasDirector, actedIn/hasDirector,`
`hasDirector/actedIn, actedIn/year,`

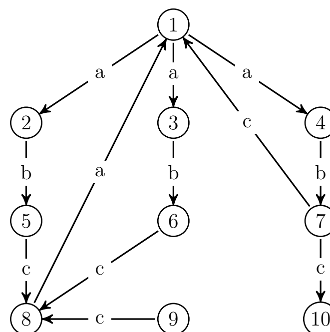
`actedIn/hasDirector/actedIn, hasDirector/actedIn/hasDirector, hasDirector/actedIn/year, actedIn/hasDirector/year`

and the resulting automaton R' will be as follows:

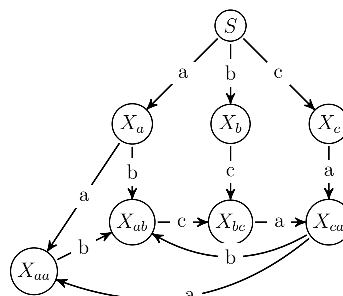


We note that this automaton, although it is equivalent to the automaton of G (i.e $\mathcal{L}(R') = \mathcal{L}(G)$), is larger than the automaton R generated for $n = 2$.

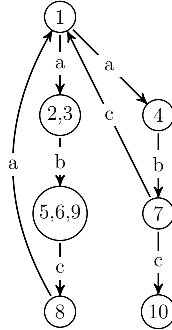
Example 7.2. Consider the following RDF-graph G taken from an example in [5]:



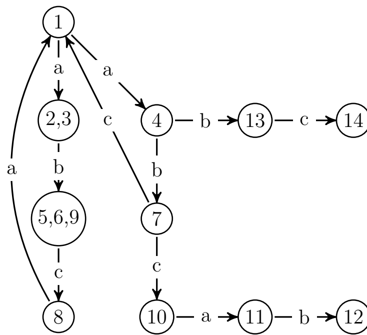
For $n = 3$, the following summarisation of G is generated:



It is possible to show that the state S can simulate every node in G but not vice-versa; for example, state 10 in G cannot simulate S. By contrast, the summarisation constructed in [5] for $n = 3$ is the following:



In this summarisation the nodes that bi-simulate each other in G are collapsed. By adding the edges $(4, b, 13)$, $(13, c, 14)$, $(10, a, 11)$ and $(11, b, 12)$ to G, we can see that our summarisation is still valid since we are not adding any new paths of length 3 to the graph. On the other hand, the summarisation from [5] will contain the new transitions as shown below, since the nodes 11, 12, 13 and 14 do not bi-simulate any other node from G:



Hence, we can conclude that although our approach is similar to that of [5], the approach in [5] generates nodes that correspond to redundant states in an automaton.

Proposition 7.1. Given a summarisation R constructed from an RDF-graph G up to path length n, where $n \geq 2$, $\mathcal{L}(G) \subseteq \mathcal{L}(R)$.

Proof. We show that the proposition is true by induction. Our base case is that for each string $s \in \mathcal{L}(G)$ of length less than or equal to n, we have that $s \in \mathcal{L}(R)$. It

is easy to show that every string $s \in \mathcal{L}(G)$ of length up to n is also in $\mathcal{L}(R)$ given steps 2 and 3 of the construction. Therefore we need to show that for every string $s' \in \mathcal{L}(G)$ of length greater than n, $s' \in \mathcal{L}(R)$. Since every node in G is both a final and initial state then for every sub-string s'', of length less than n, of s' we have that $s'' \in \mathcal{L}(G)$ and $s'' \in \mathcal{L}(R)$.

Our induction hypothesis is as follows: every string T in $\mathcal{L}(G)$ of length $n + k$, for some $k \geq 0$, is contained in $\mathcal{L}(R)$. Our induction step is to show that every string of length $n + k + 1$ in $\mathcal{L}(G)$, is also contained in $\mathcal{L}(R)$.

Consider the following string $UaTb$ of length $n + k + 1$ in $\mathcal{L}(G)$ where U is a string of length $k+1$, T is a string of length $n-2$, and a and b are strings of length 1. Since aTb is of length n then it is contained in $\mathcal{L}(G)$. Therefore, from step 3 of the construction, we also know that R contains the following transition: (S_{aT}, b, S_{Tb}). By the induction hypothesis, we know that UaT is contained in $\mathcal{L}(R)$, therefore R contains a sequence of transitions that connect the state S (the initial state) to the state S_{aT} to produce UaT. Therefore, there is a sequence of transitions in R that produces the string $UaTb$

We have shown that $\mathcal{L}(R)$ contains every sequence of strings contained in $\mathcal{L}(G)$. Therefore we can deduce that $\mathcal{L}(G) \subseteq \mathcal{L}(R)$. □

Lemma 7.1. Let R be a summarisation constructed from an RDF-graph G up to paths of length n, and Q be a SPARQL^{AR} query without the APPROX, RELAX and UNION operators. If there exists a triple pattern $(x, P, y) \in Q$ such that $\mathcal{L}(P) \cap \mathcal{L}(R) = \emptyset$, then $[\![Q]\!]_G = \emptyset$.

Proof. From Proposition 7.1 we can rewrite equation $\mathcal{L}(P) \cap \mathcal{L}(R) = \emptyset$ as follows: $\mathcal{L}(P) \cap \mathcal{L}(G) = \emptyset$. Hence, $[\![\langle x, P, y\rangle]\!]_G = \emptyset$. We notice that for any evaluation result M we have that $M \bowtie \emptyset = \emptyset$, and therefore $[\![Q]\!]_G = \emptyset$. □

The previous lemma allows us to avoid the execution of queries that do not return any answer by first testing queries against the summarisation.

We can also exploit our summarisation to replace the symbol _ with a disjunction of edge labels. Queries with the symbol _ are expensive to evaluate, since _ matches every edge label of a graph G. For example, consider the following SPARQL^{AR}

query $Q = \text{APPROX}(x, p_2/p_3, y)$ over an RDF-graph G. The rewriting algorithm will generate the following queries up to cost 1:

1. $(x, p_2/p_3, y)$
2. $(x, _/p_2/p_3, y)$
3. $(x, p_2/_/p_3, y)$
4. $(x, p_2/p_3/_, y)$
5. $(x, _/p_3, y)$
6. $(x, p_2/_, y)$
7. $(x, \epsilon/p_3, y)$
8. $(x, p_2/\epsilon, y)$

Suppose that $\mathcal{L}(G) \supseteq \{p_1, p_2, p_3, p_1p_3, p_2p_2, p_2p_3, p_2p_2p_2, p_2p_2p_3\}$ and these are all the path labels of length up to 3. Then the summary automaton R with $n = 3$ extracted from G consists of the following transitions:

$$(S, p_1, S_1), (S, p_2, S_2), (S, p_3, S_3), (S_1, p_3, S_{1,3}),$$
$$(S_2, p_2, S_{2,2}), (S_2, p_3, S_{2,3}), (S_{2,2}, p_2, S_{2,2}), (S_{2,2}, p_3, S_{2,3})$$

where every state is a final state. So we can replace the $_$ symbol as follows within the queries (1) to (8) :

1. in (2) and (3) by p_2 to give $(p_2/p_2/p_3)$ in both cases, since $p_2p_2p_3$ is the only path of length 3 ending in p_3;
2. in (5) by $(p_1|p_2)$ since p_1p_3 and p_2p_3 are the only paths of length 2 ending in p_3;
3. in (6) by $(p_2|p_3)$ since p_2p_2 and p_2p_3 are the only paths of length 2 starting with p_2;

4. in (4) we can detect that $(x, p_2/p_3/_, y)$ returns no answers since there does not exist a property path that contains p_3 followed by another URI;

5. triple patterns (1), (7) and (8) are left unchanged since these do not contain the symbol $_$.

We use Algorithm 10 to rewrite queries so that queries for which no answers will be returned are discarded. The remaining queries will then be executed by the Query Evaluation Algorithm (Algorithm 2).

Algorithm 10 rewrites each query Q, generated by the rewriting algorithm, by replacing each property path P appearing in a triple pattern in Q with the corresponding property path of the automaton computed by intersecting R with A_P, the automaton that recognises $\mathcal{L}(P)$. If at least one of the intersections is empty, then query Q does not need to be executed as it will not return any answer.

To combine the pre-computation optimisation of Chapter 6 with the summarisation optimisation, we replace the *rewrite* function in Algorithm 9, calling Algorithm 10 instead of Algorithm 3.

7.1.3 Performance Study

LUBM Evaluation

We first discuss the summarisations generated from the three LUBM datasets for the cases of $n = 2$ and $n = 3$. We generate these two summarisations from each LUBM dataset using the procedure described in Section 7.1.2. The sizes of the summarisations for the three LUBM datasets are the same, since the LUBM benchmark replicates the seed dataset multiple times and so does not create new paths as the size of the RDF-graph increases. The summarisation generated with $n = 2$ has 68 transitions (5 kilobytes). The summarisation with $n = 3$ has 122 transitions (9 kilobytes). For ease of reading, we will call the summarisation generated with $n = 2$ the *summarisation of size 2* and similarly for $n = 3$.

We first compare the number of queries generated by the rewriting algorithm with and without the summarisation optimisation. By comparing Table 6.1 (see Section 6.3) with Table 7.1 we notice that on average, fewer queries are generated using this

Algorithm 10: Rewriting of queries using the summarisation optimisation

input : Query Q; approx/relax max cost c; Graph G; Ontology K; number of paths in summary n.

output: List of pairs query/cost Qs sorted by cost.

R:=summarisation automaton of G for paths up to n;

$Qs := \emptyset$;

foreach $\langle Q', cost \rangle \in$ *rew(Q,c,K)* **do**

  toExecute:=true;

  foreach *triple pattern* $(x, P, y) \in Q'$ **do**

    $A_P :=$ the automaton that recognises $\mathcal{L}(P)$;

    P':=$A_P \cap R$;

    if $\mathcal{L}(P') = \emptyset$ **then**

      toExecute:=false

    Q':= replace (x, P, y) with (x, P', y) in Q' ;

  if *(toExecute)* **then**

    $Qs := Qs \cup \{\langle Q', cost \rangle\}$

return Qs;

optimisation technique, with the most substantial reduction being 53% for query Q_4. The only exception is Q_7 since it does not contain the APPROX operator. With a summarisation of size 3, the only further improvement is for queries Q_1 and Q_5 for maximum cost 3 where the summary results in one fewer query being generated in each case. We can conclude that using our summarisation technique we are able to reduce the number of queries: however by increasing the summarisation size from 2 to 3 there is not much improvement for the LUBM dataset.

We now show query evaluation timings using the summarisation optimisation technique with sizes 2 and 3, and also combining the pre-computation technique with the summarisation of size 3. In Figures 7.1, 7.2 and 7.3 the execution timings of the queries against datasets D1, D2 and D3 are shown. The bars with no diagonal lines represent the timings with summarisation of size 2; the bars with diagonal lines from north-west to south-east represent the timings with the summarisation of size 3; and

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	3	1	7	5	3	1	3
2	10	4	21	19	10	1	5
3	20	7	40	53	20	1	7

Table 7.1: LUBM. Number of queries generated by the rewriting algorithm with summarisation of size 2.

the bars with lines from north-east to south-west represent the timings with both the summarisation of size 3 and the pre-computation optimisation. We can see that the execution time of most queries is reduced by some orders of magnitude with respect to the simple evaluation combined with the pre-computation optimisation (Figures 6.2, 6.3 and 6.4). For example, queries Q_3, Q_4 and Q_5 are executed in less than 100 seconds for dataset D2 and maximum cost 2 and 3, using the summarisation of size 2 and 3. In contrast, the simple evaluation with and without the pre-computation optimisation either was not able to run within the 8 hours threshold or runs for more than 200 seconds. In particular, query Q_3 with maximum cost 3 is executed in 25 seconds using the summarisation of size 3 and 75 seconds using the summarisation of size 2, reducing the execution time by 99 % if we consider the evaluation with the pre-computation optimisation. For Figures 7.2 and 7.3, we also notice that we are now able to run all the queries over datasets D2 and D3 within the 8 hours threshold.

If we consider Figure 7.1, we notice that queries that originally executed in less than 10^{-1} seconds now need more than 1 second, especially when we combine the summarisation optimisation with the pre-computation optimisation. This is due to the time needed by the summarisation optimisation to rewrite each query generated by the rewriting algorithm, and the time spent executing the additional queries when using the pre-computation optimisation.

Similarly to the analysis in Chapter 6.3, we argue that the execution time of queries that take less than 5 seconds to execute does not improve when the pre-computation optimisation is used. We notice an improvement in execution time

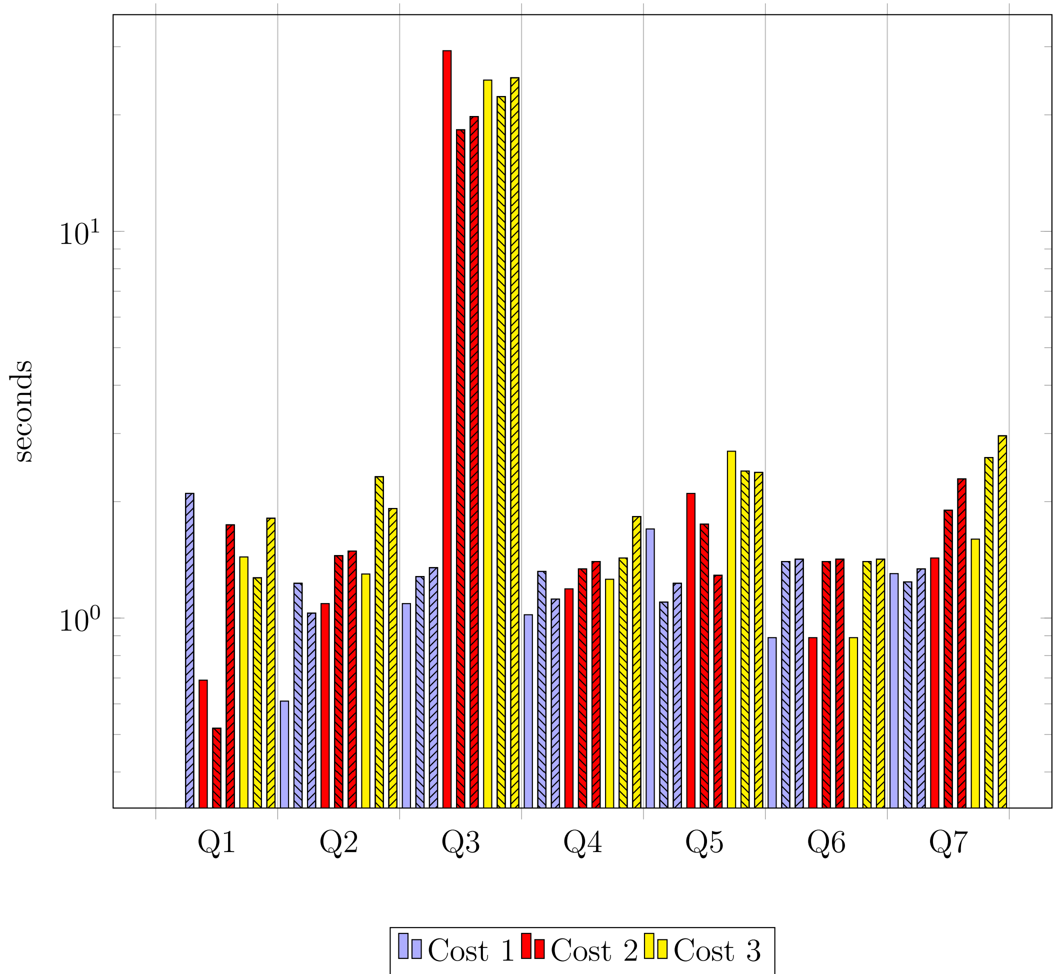


Figure 7.1: LUBM. Timings for database D1. Plain coloured bars are the summarisation of size 2. Bars with diagonal lines from north-west to south-east are the summarisation of size 3. Bars with diagonal lines from south-west to north-east are the summarisation of size 3 together with pre-computation optimisation.

when we compare the summarisation of size 2 with the summarisation of size 3. This is due to the fact that we are able to replace the symbol _ with a smaller number of edge labels when using the summarisation of size 3 since it gives a more accurate representation of the RDF dataset. Hence, we can conclude that, even though the number of queries is not reduced considerably when using the summarisation of size 3 compared to the summarisation of size 2, the execution time does improve.

We refer to the time that the evaluation algorithm spends on the optimisations described in this chapter as the *compilation time*. In general, we found that the time it takes to rewrite a query using the summarisation is between 0.001 and 2 seconds,

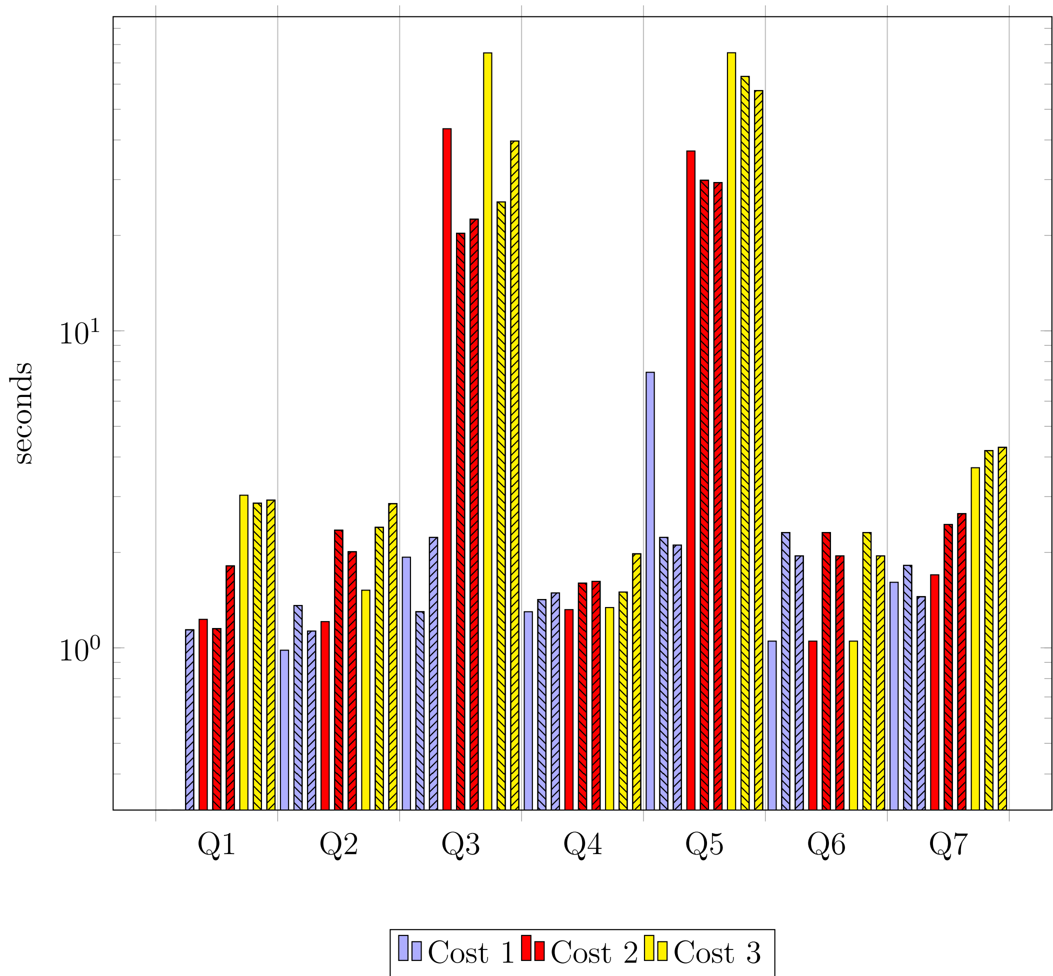


Figure 7.2: LUBM. Timings for database D2. Plain coloured bars are the summarisation of size 2. Bars with diagonal lines from north-west to south-east are the summarisation of size 3. Bars with diagonal lines from south-west to north-east are the summarisation of size 3 together with pre-computation optimisation.

depending on the number queries the rewriting algorithm generates and how often the _ symbol appears. For example, for query Q_3 over the LUBM dataset with the summarisation of size 2, the compilation time is 0.8, 1.21 and 1.67 seconds for cost 1, 2 and 3 respectively.

DBpedia Evaluation

We now discuss the evaluation of the DBpedia queries using the summarisation optimisation technique. In contrast to the analysis with LUBM, we are not able to show execution times with a summarisation of size 3. This was due to the large size of

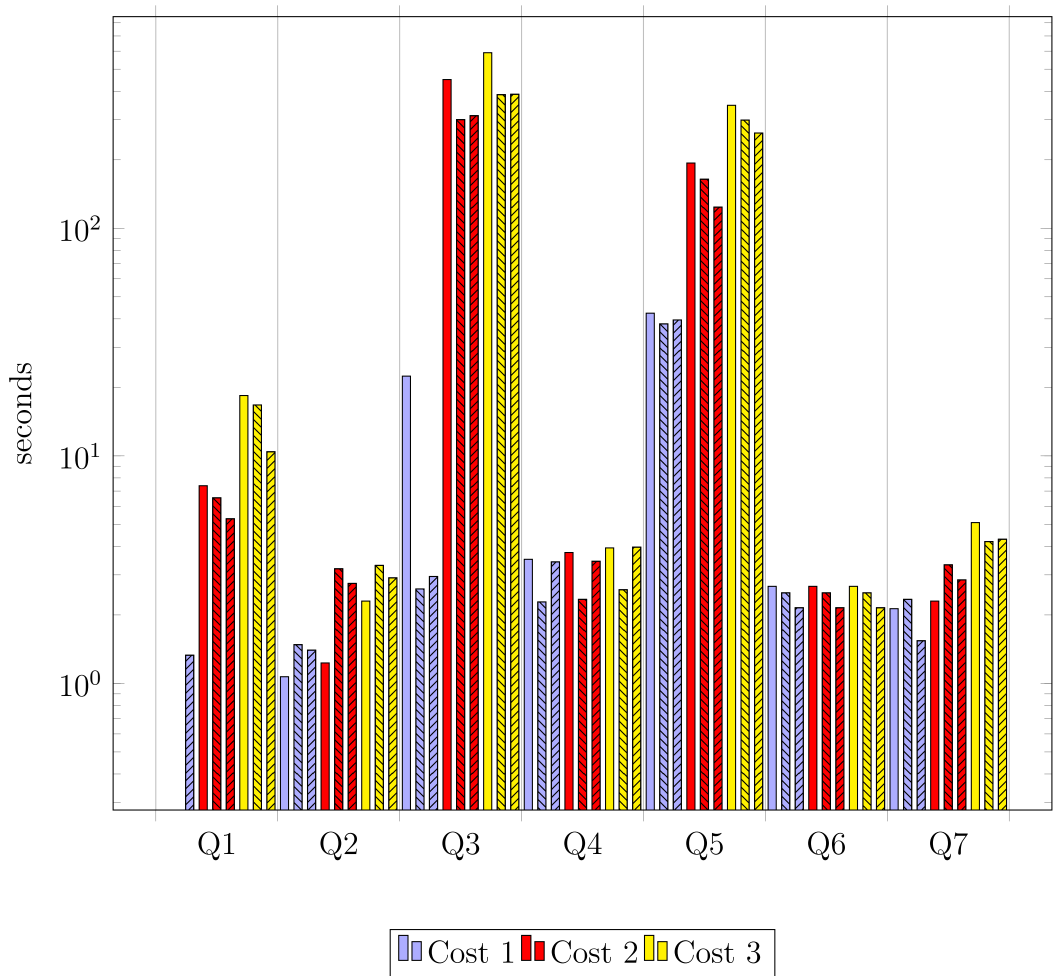


Figure 7.3: LUBM. Timings for database D3. Plain coloured bars are the summarisation of size 2. Bars with diagonal lines from north-west to south-east are the summarisation of size 3. Bars with diagonal lines from south-west to north-east are the summarisation of size 3 together with pre-computation optimisation.

the summarisation: the DBpedia summarisation of size 2 contains 71,514 transitions (4.3 megabytes), and summarisation of size 3 contains 1,109,836 transitions (67.3 megabytes).

The queries generated after rewriting with the summarisation of size 3 cause the Java Virtual Machine to crash due to the size of the queries. Therefore, we will only compare the summarisation of size 2 with and without the pre-computation optimisation.

By comparing Table 7.2 with Table 6.3 (Section 6.3) we can see that the number of queries to execute is reduced considerably for most queries. In particular, for

query Q_5 the number of queries to execute is reduced from 168 to 18 when the maximum cost is 3. Due to the size of the summarisation of size 2, we are unable to compute the number of queries generated with the summarisaion optimisation for queries Q_1 and Q_4 with cost greater than 1, since the algorithm produced queries too large to be handled by the system.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	3	2	2	1	0
2	N/A	5	5	N/A	3
3	N/A	10	10	N/A	18

Table 7.2: DBpedia. Number of queries generated by the rewriting algorithm with summarisation of size 2.

Figure 7.4 shows the execution time of the queries using the summarisation of size 2 with the pre-computation optimisation (bars with no lines) and without the pre-computation optimisation (bars with diagonal lines). We notice that (except for the queries that we were unable to execute) there is a considerable improvement in the timings when using only the summarisation optimisation. For example, we are able to execute queries Q_2, Q_3 and Q_5 in less than 3 seconds. In contrast, using the simple evaluation algorithm, the execution time of such queries is 7 seconds for Q_2, approximately 10,000 seconds for Q_5 and more than 8 hours for Q_5. When we combine it with the pre-computation optimisation, we notice that for query Q_5 and maximum cost 3 more time is required with respect to the summarisation optimisation only.

We found the compilation times to be similar to those for the queries on LUBM. For example, the compilation time for query Q_5 is 0.01, 0.8 and 2.1 seconds for cost 1, 2 and 3 respectively. The high time spent on compilation for cost 3 might be due to the fact that the number of queries the rewriting algorithm generates is rather large: 168 which is reduced to 18 with the application of the optimisation.

From our analysis we conclude that, for the DBpedia dataset, we were unable to evaluate some queries. This is mainly due to the large number of predicates and

connections that the DBpedia dataset has, which in turn leads to a large summary even for summarisation of size 2. We are still able to show that for the queries that did run, the summarisation optimisation improved the execution time of each of them.

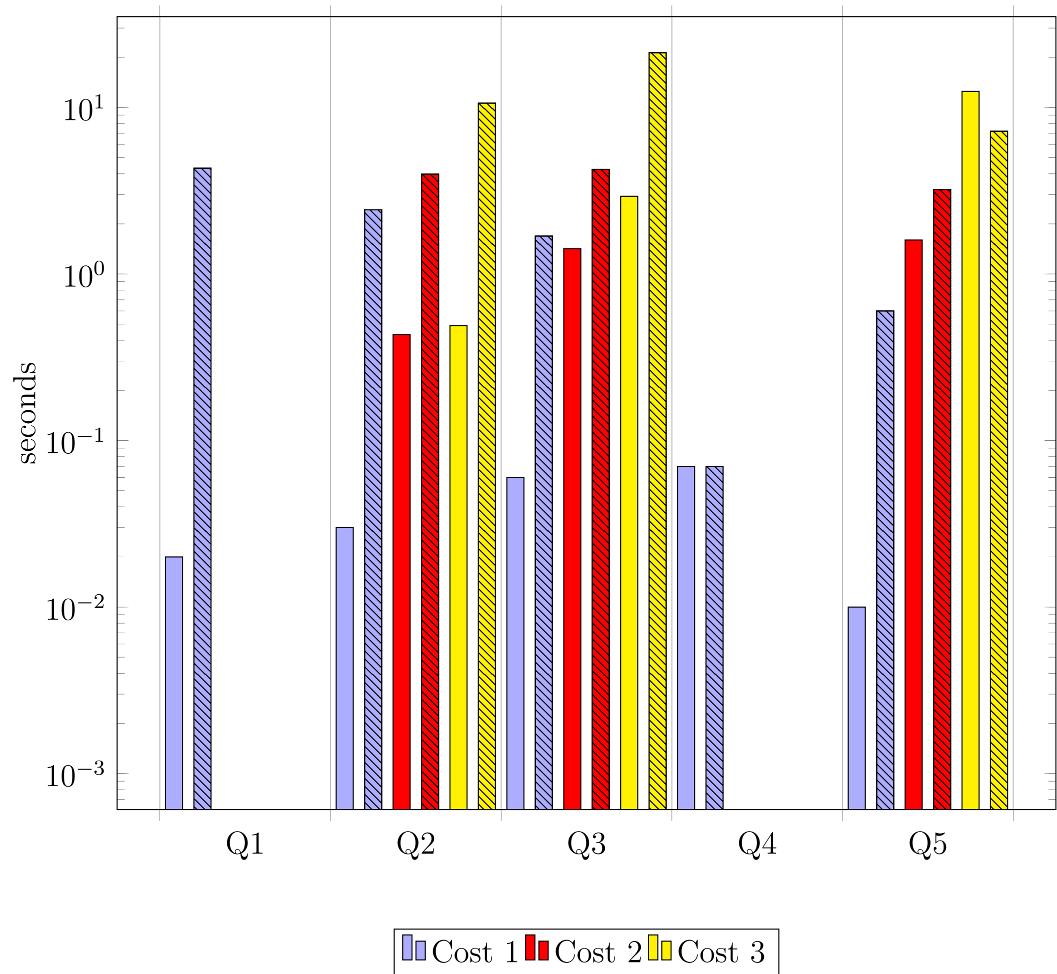


Figure 7.4: DBpedia. Timings, with summarisation of size 2, with and without the pre-computation optimisation.

YAGO Evaluation

Compared to DBpedia, the summarisations for the YAGO datasets are considerably smaller. In fact, for size 2 the summarisation generated has only 1,320 transitions (88.2 kilobytes) and for size 3 the summarisation generated has 10,528 transition (708 kilobytes).

Table 7.3 shows the number of queries generated when using the summarisation of size 2 and of size 3 (no new queries are discarded when increasing the size of the summary). Comparing Table 6.5 with Table 7.3 we see that there is no improvement for query Q_2 (since it contains only the RELAX operator) and Q_3 (since it contains only one APPROX operator with only the property "happenedIn" which can appear in long path sequences in the YAGO dataset). Similarly to two of the queries of DBpedia, we were not able to execute query Q_1 for cost 3. This was due to the size of the queries that were generated during the summarisation optimisation.

Max Cost	Q_1	Q_2	Q_3
1	10	2	5
2	46	2	12
3	N/A	2	25

Table 7.3: YAGO. Number of queries generated by the rewriting algorithm given maximum costs of 1, 2 and 3 with summarisation of size 2 and of size 3.

In Figure 7.5 the bars with no diagonal lines represent the query evaluation timings with summarisation of size 2; the bars with diagonal lines from north-west to south-east represent the timings with the summarisation of size 3; and the bars with lines from north-east to south-west represent the timings with the both the summarisation of size 3 and the pre-computation optimisation. By comparing Figures 6.6 (Chapter 6) and 7.5 we notice that the performance improved. In particular, query Q_1 runs in less than 230 seconds instead of more than 400 seconds using the simple evaluation. All the other queries now take less than two seconds to evaluate.

We notice that in most instances (except for query Q_1 with maximum cost 2) the summarisation optimisation worsens the evaluation time when increasing the summarisation from size 2 to size 3. By increasing the size of the summarisation, the optimisation algorithm takes more time to rewrite the queries since it has to compute the intersection between the summarisation and the property paths of the query. In fact compilation time is between 0.01 and 0.7 seconds for the summarisation of size 2 and 0.01 and 1.5 seconds for the summarisation of size 3. For example, query Q_3

has compilation time for the summarisation of size 2 of 0.03, 0.58 and 0.64 seconds for cost 1, 2 and 3 respectively; for the summarisation of size 3 it is 0.07, 0.62 and 1.1 seconds for cost 1, 2 and 3 respectively.

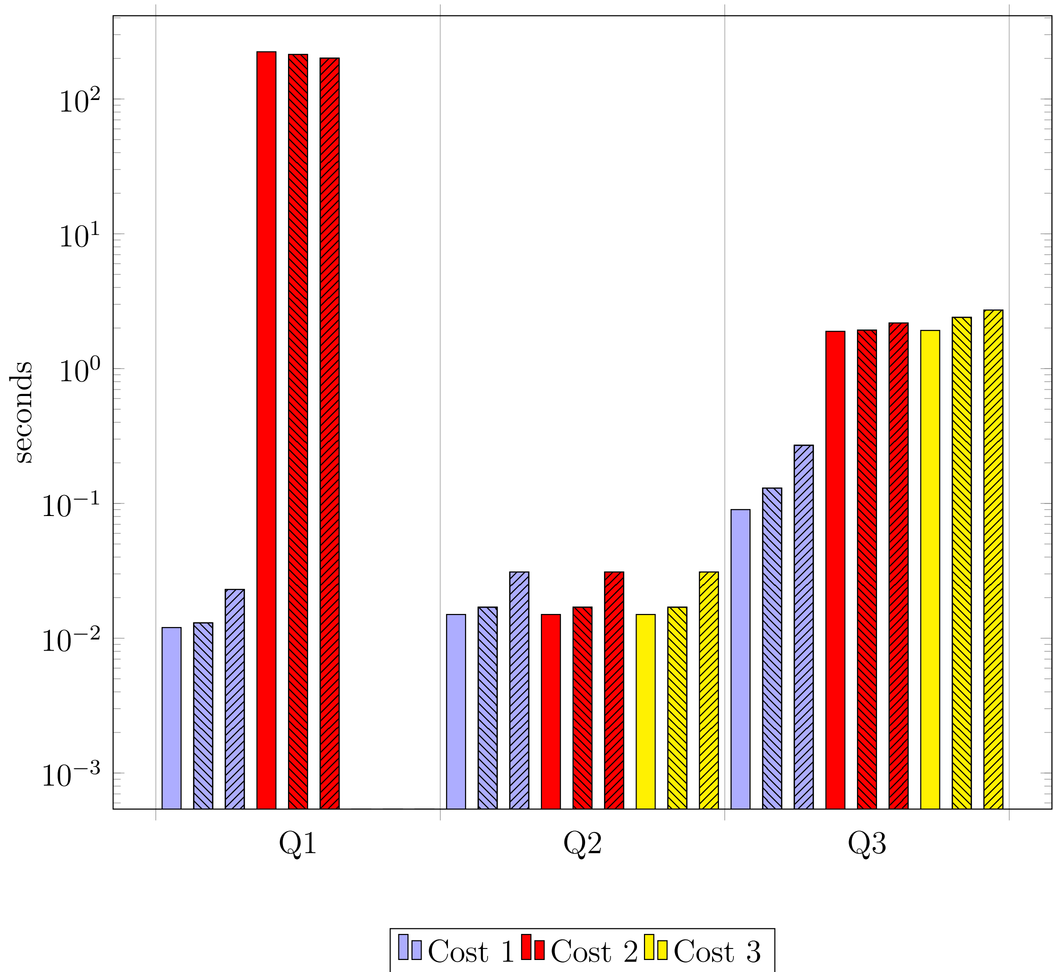


Figure 7.5: YAGO timings. Plain coloured bars are the summarisation of size 2. Bars with diagonal lines from north-west to south-east are the summarisation of size 3. Bars with diagonal lines from south-west to north-east are the summarisation of size 3 together with pre-computation optimisation.

We conclude that the summarisation optimisation technique is well suited to large sparse datasets that contain a limited number of property labels (such as YAGO or the LUBM dataset D3), and it improves the performance for queries with the APPROX operator. However, is not suitable for dense datasets that contain a large number of property labels such as DBpedia, since the summarisation generated is too large and cannot be used in practice for optimising the queries.

7.2 Query Containment

The number of queries that the rewriting algorithm generates depends on the size of the original query; therefore decreasing the size of the original query may reduce the number of generated queries, hence improving query execution times. We start by giving an example to show the intuition behind this approach. From now until the end of the chapter we assume a fixed ontology K.

Example 7.3. Consider the following query (we omit the SELECT for simplicity):

$$Q = \text{APPROX}(x, p, y) \text{ AND } \text{APPROX}(x, r, y)$$

Suppose the system applies the substitution edit operation to the second triple pattern and generates the following query:

$$Q = (x, p, y) \text{ AND } (x, _, y)$$

We notice that the first triple pattern returns only pairs of nodes that are connected by predicate p. On the other hand the second triple pattern returns every pair of nodes that are connected by the means of any predicate. Hence we are able to write the following equation $[\![\langle x, p, y\rangle]\!]_G = [\![\langle x, p, y\rangle]\!]_G \bowtie [\![\langle x, _, y\rangle]\!]_G$ and the answers returned by query Q are equivalent to the answers returned by $Q' = (x, p, y)$

A similar approach can also be adopted for relaxation based on logical inference from the ontology. In general, given a query pattern Q_1 AND Q_2, we say that the query pattern Q_1 *covers* Q_2 with respect to an ontology K if the evaluation of Q_1 is a subset of the evaluation of Q_2 for a given ontology K, for all graphs G. If Q_1 covers Q_2 then Q_1 is equivalent to Q_1 AND Q_2. It is possible to apply this technique even if $var(Q_1) \neq var(Q_2)$: if each variable in $var(Q_2)$ that is not in $var(Q_1)$ does not appear anywhere else in the query then Q_1 covers Q_2 if $[\![\text{SELECT}_{\overrightarrow{w}} Q_1]\!]_G \subseteq [\![\text{SELECT}_{\overrightarrow{w}} Q_2]\!]_G$ with $\overrightarrow{w} = var(Q_1) \cap var(Q_2)$ for all graphs G.

7.2.1 Related Work

We now discuss previous works on query containment, regular path query containment and, since in our work we focus on regular expressions and languages, also on containment between automata.

Throughout the section we use the following notation: a set of symbols Σ; a set of variables V; a weighted graph $G = (N, E)$ where $N \subseteq \Sigma$ is a finite set of nodes and E is a finite set of labelled weighted edges of the form $\langle\langle s, p, o\rangle, cost\rangle$ with $s, o \in N$, $p \in \Sigma$ and *cost* the weight of the edge. A *regular expression* $R \in RegEx$ is of the form

$$R : \epsilon \mid _ \mid l \mid (R_1.R_2) \mid (R_1|R_2) \mid R^*$$

where $R, R_1, R_2 \in RegEx$, l is any symbol from Σ, ϵ is the empty string and $_$ is the disjunction of all the symbols in Σ.

A *conjunctive regular path query* (CRPQ) Q is of the form

$$q(X)\text{:-}y_1R_1z_1, \ldots, y_nR_nz_n,$$

where each $y_iR_iz_i$, is a conjunct of the query and together these form the *body* of the query; each $y_i, z_i \in \Sigma \cup V$; each R_i is a regular expression; $q(X)$ is the *head* of the query, with $X \subseteq V$ a set of variables appearing in the body of the query.

In [3] the authors investigate the complexity of *weighted finite automata* (WFA) in different scenarios. Abusing notation, we can consider a weighted graph G as previously defined as a WFA. Given WFA A the authors in [3] define the set of all words accepted by the automaton by $\mathcal{L}(A)$. Given a word $w \in \mathcal{L}(A)$, the minimum cost of generating such a word is denoted by $\mathcal{L}_A(w)$. They define containment as follows: given two WFA A_1 and A_2, A_1 is contained in A_2 if for each word $w \in \mathcal{L}(A_1)$ then also $w \in \mathcal{L}(A_2)$ and $\mathcal{L}_{A_1}(w) \geq \mathcal{L}_{A_2}(w)$. They show that containment of automata with weights restricted to $\mathbb{N}$ and $\mathbb{Z}$ is undecidable, and therefore is also undecidable with respect to $\mathbb{Q}$. Moreover, containment is still undecidable even restricting the weights to the set $\{-1, 0, 1\}$ or $\{0, 1, 2\}$.

In [57, 58] the authors discuss the complexity of containment for *simple regular expressions*. Their simple regular expression syntax is defined as follows, where R is a simple regular expression:

$$W : l \mid (l.W)$$
$$D : W \mid (W \mid D)$$
$$D_l : l \mid (l \mid D_l)$$
$$R : l \mid l? \mid l^* \mid l^+ \mid D_l \mid D_l? \mid D_l^* \mid D_l^+ \mid D \mid D? \mid D^* \mid D^+ \mid W \mid W? \mid W^* \mid W^+$$

They make also use of following operators: $l?$ which is equivalent to $(\epsilon \mid l)$, and (l^+), which is equivalent to $(l.l^*)$. We notice that they do not allow indefinite nesting. They show the complexity with respect to inclusion and equivalence for the following regular expression fragments:

- l, l^+ — Inclusion: PTIME. Equivalence: PTIME.
- l, l^* — Inclusion: CONP-complete. Equivalence: PTIME.
- l, $l?$ — Inclusion: CONP-complete. Equivalence: PTIME.
- l, D_l^+ — Inclusion: CONP-complete. Equivalence: CONP.
- l^+, D_l — Inclusion: CONP-complete. Equivalence: CONP.
- l, W^+ — Inclusion: CONP-complete. Equivalence: CONP.
- $S - \{D_l^*, D^*\}$ — Inclusion: CONP-complete. Equivalence: CONP.
- $S - \{D^*, D_l^*, D^+, D_l^+\}$ — Inclusion: CONP-complete. Equivalence: CONP.
- l, D_l^* — Inclusion: PSPACE-complete. Equivalence: PSPACE.
- l, D_l^+ — Inclusion: PSPACE-complete. Equivalence: PSPACE.
- $S - \{D^*\}$ — Inclusion: PSPACE-complete. Equivalence: PSPACE.
- $S - \{D^*, D^+\}$ — Inclusion: PSPACE-complete. Equivalence: PSPACE.
- l, D^+ — Inclusion: PSPACE-complete. Equivalence: PSPACE.
- l, D^* — Inclusion: PSPACE-complete. Equivalence: PSPACE.
- S — Inclusion: PSPACE-complete:. Equivalence: PSPACE.

Bounding the number of occurrences of a symbol in the regular expression makes containment tractable. In fact, given a constant $k \geq 3$ which corresponds to the maximum number of occurrences of a symbol in a regular expression, the problem of containment is $O(n^k)$, with n the size of the regular expression, and therefore in

PTIME. Finally, if the regular expression is deterministic (i.e. the corresponding automaton of the regular expression is deterministic) then complexity is in PTIME.

From now on we omit the cost of the edges of a graph for simplicity. We now discuss the complexity results achieved in [30] when considering containment of conjunctive queries with regular expressions. The authors focus on CRPQ queries for semi-structured data (specifically the language StruQL). Given two CRQP queries Q_1 and Q_2, they consider that Q_1 is contained in Q_2 if for every database G the answers of Q_1 over G, $Q_1(G)$, are contained in the answers of Q_2 over G, $Q_2(G)$. They show that a query Q_1 is contained in Q_2 if there is a mapping f from Q_2 to Q_1, with $f(Q_2) = Q_1$, and f a function that maps the variables of Q_2 to the variables of Q_1. For their complexity investigation they focus on a sub-language of CRPQ, called StruQL$_0$. A StruQL$_0$ query is a CRPQ query in which all regular expressions are simple. In their formalisation, a *simple regular expression* is a regular expression of the form $r_1.r._2 \ldots r_n$ where each r_i is either $_^*$ or a constant from Σ. They show that the complexity of containment of simple regular expressions is in PTIME, and the complexity of containment of StruQL$_0$ queries is NP-complete.

Similarly to the previous work, in [13] the authors discuss the complexity of query containment with CRPQ queries but in their case they include the inverse operator (CRPQI). To do this they include in their vocabulary the inverse of every edge label, that is $\Sigma = D \cup \{p^- \mid p \in D\}$. They show that query containment of two CRPQI queries can be solved in EXPSPACE (upper bound) and that query containment for two CRPQ queries is EXPSPACE-hard (lower bound).

We now discuss the work in [34] where the authors describe a particular kind of query approximation based on *regular path queries* (i.e. CRPQs with a single conjunct). In order to compute the approximated answers they make use of a weighted transducer which transforms regular path expressions with a transformation cost. Given an RPQ query Q, a graph database G and a transducer $\mathcal{T}$ the approximate answers $ans_{\mathcal{T}}(Q, G)$ at cost k is a set of triples (a, b, k) where a and b are nodes in G such that $(a, b) \in Q'(G)$ and the query Q' is generated by the transducer $\mathcal{T}$ given input Q .

They define three different containment semantics:

- Q_1 is *approximately contained* in Q_2, $Q_1 \subseteq_{\mathcal{T}} Q_2$, if for any database G, for each $(a, b, n) \in ans_{\mathcal{T}}(Q_1, G)$ there exists $(a, b, m) \in ans_{\mathcal{T}}(Q_2, G)$ with $m \leq n$. The complexity is equivalent to the complexity of RPQ evaluation.

- Q_1 is *k-contained* in Q_2, $Q_1 \subseteq_{\mathcal{T},k} Q_2$, if it is approximately contained and we also have that $n - m \leq k$. The complexity is in PSPACE with respect to the combined size of Q_1 and Q_2, and moreover is sub-exponential with respect to k.

- Q_1 is *reliably contained* in Q_2, $Q_1 \subseteq_{\mathcal{T},\omega} Q_2$, if there exists a k such that $Q_1 \subseteq_{\mathcal{T},k} Q_2$. The complexity is PSPACE-hard.

Figure 7.6 from Sagiv et al. [24] and Deutsh et al. [70] summarises several complexity results for query containment with regular expressions.

The work presented here follows a similar approach to that in [33] where Gottlob et al. evaluate conjunctive queries under Datalog rules by the means of a rewriting algorithm. They minimise the number of queries generated by a rewriting algorithm by applying the query containment technique. In contrast to [33], our optimisation considers query containment for SPARQL with flexible operators, and does not use Datalog rules.

7.2.2 Preliminary Definitions

In this section we study query containment for SPARQL^{AR} first considering approximation then relaxation and finally both. We first define the concept of a renaming function:

Definition 7.1 (Renaming function)**.** A renaming function h from UVL to UVL is a partial function $h : UVL \rightarrow UVL$. We assume that $h(x) = x$ whenever x is either a literal in L or a URI in U. It is possible that a variable is mapped to a URI or a Literal. The *variables* of h, $var(h)$, is the subset of V where h is defined. Given a query Q such that $var(Q) \subseteq var(h)$, $h(Q)$ is the query obtained by replacing the variables in $var(Q)$ according to h.

Fragment name	Regular expressions syntax
conj. queries	$R \to l \mid R_1.R_2$
$(*)$	$R \to l \mid * \mid R_1.R_2$
$(*, \mid)$	$R \to l \mid * \mid R_1.R_2 \mid (R_1 \mid R_2)$
$(*, _)$	$R \to l \mid _ \mid * \mid R_1.R_2$
(l^*)	$R \to l \mid l^* \mid R_1.R_2$
$(*, _, l^*, \mid)$	$R \to l \mid * \mid _ \mid l^* \mid R_1.R_2 \mid (R_1 \mid R_2)$
W	$R \to l \mid _ \mid S^* \mid R_1.R_2 \mid (R_1 \mid R_2)$ $S \to l \mid _ \mid S_1.S_2$
Z	$R \to S \mid S^*$ $S \to l \mid S_1.S_2 \mid (S_1 \mid S_2)$
CRPQs	$R \to l \mid _ \mid R^* \mid R_1.R_2 \mid (R_1 \mid R_2)$

Containment of	Upper bound
conjunctive queries	NP
$(*)$-queries	NP
unions of conj. queries	Π_2^p
unions of $(*, \mid)$-queries	↑
unions of $(*, _)$-queries	↑
unions of (l^*)-queries	↑
unions of $(*, _, l^*, \mid)$-queries	Π_2^p
unions of W queries	?
unions of Z queries	↑
unions of CRPQs	EXPSPACE

Containment of	Lower bound
conjunctive queries	NP
$(*)$-queries	↓
unions of conj. queries	Π_2^p
$(*, \mid)$-queries	Π_2^p
$(*, _)$-queries	Π_2^p
(l^*)-queries	Π_2^p
$(*, _, l^*, \mid)$-queries	↓
W queries	PSPACE
Z queries	EXPSPACE
CRPQs	↓

Figure 7.6: Containment results

To gain a finer understanding of how query costs influence query containment in SPARQL^{AR}, we give a new definition that takes into account an operator *op* that compares the costs of mappings.

Definition 7.2. Given G, Q, Q' and *op*, $[\![Q]\!]_G$ is subset of $[\![Q']\!]_G$ with respect to *op*, $[\![Q]\!]_G \subseteq_{op} [\![Q']\!]_G$, if for each pair $\langle \mu, k\rangle \in [\![Q]\!]_G$ there exists a pair $\langle \mu, k'\rangle \in [\![Q']\!]_G$, such that $(k\ op\ k')$, where *op* is either $\geq$ or $=$.

From now until the end of this section we refer to the containment of two SPARQL^{AR} queries Q and Q' by $Q \subseteq_{op} Q'$, which is defined as follows:

Definition 7.3 (Query containment)**.** Given Q, Q' and *op*, Q is contained in Q' with respect to *op*, $Q \subseteq_{op} Q'$, if there exists a renaming function h such that $head(Q) = h(head(Q'))$, and $[\![Q]\!]_G \subseteq_{op} [\![h(Q')]\!]_G$ for every graph G.

In this query containment definition we use the renaming function to compare the mappings from the evaluation of the two queries in case the variables are represented by different symbols.

We make the following observation:

Observation 7.1. Given a triple pattern $\langle x, P, y\rangle$, for every graph G the following holds: $[\![\langle x, P, y\rangle]\!]_G \subseteq [\![\text{APPROX/RELAX}(x, P, y)]\!]_G$.

Before studying query containment further we introduce some necessary definitions. Given a NFA M_P, we refer to $\mathcal{L}(M_P)$ as the language recognized by M_P. This is equivalent to the language denoted by $P \in RegEx(U)$ in Section 2.1.3, i.e $\mathcal{L}(P)$. A_P denotes the NFA constructed by approximating M_P and R_P denotes the NFA constructed by relaxing M_P (see proof of Lemma 4.1).

Given $P, P' \in RegEx(U)$, we say that P' is *more expressive* than P if $\mathcal{L}(P)$ is a subset of $\mathcal{L}(P')$, i.e. $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. More formally:

Definition 7.4. Given $P, P' \in RegEx(U)$, $\mathcal{L}(P)$ is a subset of $\mathcal{L}(P')$, $\mathcal{L}(P) \subseteq \mathcal{L}(P')$, if each string that conforms to the regular expression P also conforms to P'.

Example 7.4. Given a regular expression $P_1 = p_1/p_2/(p_1|p_2)$, there are two strings that conform to it: $p_1p_2p_1$ and $p_1p_2p_2$. Considering a second regular expression

$P_2 = (p_1|p_2)^*$ it is possible to write an infinite set of strings which conform to it, including $p_1p_2p_1$ and $p_1p_2p_2$. Therefore $\mathcal{L}(P_1) \subseteq \mathcal{L}(P_2)$.

Proposition 7.2. Given two triple patterns (x, P, y) and (x', P', y') and a renaming function $h = \{x' \to x, y' \to y\}$ and an $op \in \{=, \geq\}$, $(x, P, y) \subseteq_{op} h((x', P', y'))$ if and only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$.

Proof. If is true that $(x, P, y) \subseteq_{op} h((x', P', y'))$ then it holds that $[\![x, P, y]\!]_G \subseteq_{op} [\![h(x', P', y')]\!]_G$, for every G, i.e. $[\![x, P, y]\!]_G \subseteq_{op} [\![x, P', y]\!]_G$. The latter holds only if $\mathcal{L}(G) \cap \mathcal{L}(P) \subseteq \mathcal{L}(G) \cap \mathcal{L}(P')$ for every G, and therefore only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$.

If $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ holds then for every string $s \in \mathcal{L}(P)$ then also $s \in \mathcal{L}(P')$. If we represent s as a concatenation of URIs, $P_s = p_1/p_2/\ldots/p_n$ then every answer returned by $[\![x, P_s, y]\!]_G$ for every $s \in \mathcal{L}(P)$ will also be returned by $[\![x, P', y]\!]_G$ for every G.

The proof holds for every *op* since we assume that graphs have cost 0 for all edges.

□

7.2.3 Containment of Single-Conjunct Queries

Approximation

We begin by considering queries with only one conjunct to which APPROX may be applied. The following Lemma gives a major property of the approximation operator:

Lemma 7.2. Given any graph G and any regular expression $P \in RegEx(U)$ $[\![\text{APPROX}(x, P, y)]\!]_G$ will return every mapping $\mu = \{x \to c, y \to c'\}$ such that there is a path from c to c' in G.

Proof. This can be easily verified since for every $P' \in RegEx(U)$, $\mathcal{L}(P') \subseteq \mathcal{L}(A_P)$.

□

We consider three different cases for single-conjunct SPARQL^{AR} query containment given a renaming function $h = \{x' \to x, y' \to y\}$:

$$(x, P, y) \subseteq_{op} \text{APPROX}(x', P', y') \tag{7.2.1}$$

$$\text{APPROX}(x, P, y) \subseteq_{op} (x', P', y') \tag{7.2.2}$$

$$\text{APPROX}(x, P, y) \subseteq_{op} \text{APPROX}(x', P', y') \tag{7.2.3}$$

7.2.1 op is =: is true if and only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ (by Observation 7.1). □

7.2.1 op is ≥: is true if and only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ (by Observation 7.1). □

7.2.2 op is =: is true if and only if $\mathcal{L}(A_P) = \mathcal{L}(P) \subseteq \mathcal{L}(P')$. If $\mathcal{L}(A_P) = \mathcal{L}(P) \subseteq \mathcal{L}(P')$ is true then applying any step of approximation to P the language $\mathcal{L}(P)$ will not change, therefore every evaluation in $[\![\text{APPROX}(x, P, y)]\!]_G$ is of the form $\langle \mu, 0 \rangle$. Since, $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ we can conclude that every evaluation of the form $\langle \mu, 0 \rangle \in [\![\text{APPROX}(x, P, y)]\!]_G$ is also contained in $[\![h(\langle x', P', y' \rangle)]\!]_G$.

If every pair $\langle \mu, k \rangle \in [\![\text{APPROX}(x, P, y)]\!]_G$, $\langle \mu, k \rangle$ is contained in $[\![h(\langle x', P', y' \rangle)]\!]_G$ then k has to be equal to 0 for every G. Therefore applying any step of approximation to the regular expression P the following holds: $\mathcal{L}(A_P) = \mathcal{L}(P)$; and therefore $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. □

7.2.2 op is ≥: is true if and only if $\mathcal{L}(A_P) \subseteq \mathcal{L}(P')$. □

7.2.3 op is =: is true if and only if $\mathcal{L}(P) = \mathcal{L}(P')$. The APPROX operator returns every pair of connected nodes in G, therefore two approximated triple patterns will return the same sets of mappings, with possibly different costs associated. We know by the definition of containment, with *op* being =, that for each $\langle \mu, k \rangle \in [\![\text{APPROX}(x, P, y)]\!]_G$ there exists a pair $\langle \mu, k \rangle \in [\![h(\text{APPROX}(x', P', y')]\!]_G$, for every G. Moreover, given Lemma 7.2, for each $\langle \mu', k' \rangle \in [\![h(\text{APPROX}(x', P', y'))]\!]_G$ there exists a pair $\langle \mu', k'' \rangle \in [\![\text{APPROX}(x, P, y)]\!]_G$, for every G. For the containment to hold it has to be the case that $k' = k''$ and therefore $[\![\text{APPROX}(x, P, y)]\!]_G =_= [\![h(\text{APPROX}(x', P', y'))]\!]_G$ for every G.

We can see that $[\![\text{APPROX}(x, P, y)]\!]_G =_= [\![\text{APPROX}(x', P', y')]\!]_G$ holds if and only if $[\![(x, P, y)]\!]_G =_= [\![(x', P', y')]\!]_G$ holds, and therefore also $\mathcal{L}(P) = \mathcal{L}(P')$ holds. □

7.2.3 op is ≥: is true if and only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. By the definition of containment, with *op* being ≥, and Observation 7.1, for each $\langle \mu, 0 \rangle \in [\![\langle x, P, y \rangle)]\!]_G \subseteq$

$[\![\text{APPROX}(x, P, y)]\!]_G$ there exists a pair $\langle \mu, k \rangle \in [\![h(\text{APPROX}(x', P', y'))]\!]_G$, for every G. For the containment to hold, $0 \geq k = 0$ and therefore $\langle \mu, 0 \rangle \in [\![h(\langle x', P', y' \rangle)]\!]_G$. We can conclude that $(x, P, y) \subseteq (x', P', y')$ and therefore $\mathcal{L}(P) \subseteq \mathcal{L}(P')$.

If $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ then it is possible to approximate P' and P in order to reach a common approximated regular expression A such that $\mathcal{L}(A) \subseteq \mathcal{L}(A_P) = \mathcal{L}(A'_P)$. To construct A from P' we apply fewer steps of approximation than constructing A from P since $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. Therefore, for each $\langle \mu, k \rangle \in [\![\text{APPROX}(x, P, y)]\!]_G$ there exists $\langle \mu, k' \rangle \in [\![\text{APPROX}(x', P', y')]\!]_G$ with $k \geq k'$ up to approximating A for every G, but since it is always true that $[\![\text{APPROX}(x, A, y)]\!]_G \subseteq_{\geq} [\![h(\text{APPROX}(x', A, y'))]\!]_G$, we can conclude that $\text{APPROX}(x, P, y) \subseteq_{\geq} \text{APPROX}(x', P', y')$. □

Relaxation

We again consider queries with only one conjunct. Similarly to approximation we consider three different cases for single-conjunct SPARQL^{AR} query containment given a renaming function $h = \{x' \to x, y' \to y\}$ and an ontology K:

$$(x, P, y) \subseteq_{op} \text{RELAX}(x', P', y') \tag{7.2.4}$$

$$\text{RELAX}(x, P, y) \subseteq_{op} (x', P', y') \tag{7.2.5}$$

$$\text{RELAX}(x, P, y) \subseteq_{op} \text{RELAX}(x', P', y') \tag{7.2.6}$$

7.2.4 op is =: is true if and only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ (by Observation 7.1). □

7.2.4 op is $\geq$: is true if and only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ (by Observation 7.1). □

7.2.5 op is =: is true if and only if $\mathcal{L}(R_P) = \mathcal{L}(P) \subseteq \mathcal{L}(P')$, similarly to case (2) with *op* being =. □

7.2.5 op is $\geq$: is true if and only if $\mathcal{L}(R_P) \subseteq \mathcal{L}(P')$. □

7.2.6 op is =: is true only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. By the definition of containment, with *op* being =, and Observation 7.1, for each $\langle \mu, 0 \rangle \in [\![\langle x, P, y \rangle)]\!]_G \subseteq [\![\text{RELAX}(x, P, y)]\!]_G$ there exists a pair $\langle \mu, 0 \rangle \in [\![h(\text{RELAX}(x', P', y'))]\!]_G$, for every G. Since, it has to be the case that $\langle \mu, 0 \rangle \in [\![h(\langle x', P', y' \rangle)]\!]_G$, we can conclude that $\langle x, P, y \rangle \subseteq \langle x', P', y' \rangle$ and therefore $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. □

7.2.6 op is $\geq$*:* is true only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. By the definition of containment, with *op* being $\geq$, and Observation 7.1, for each $\langle\mu, 0\rangle \in [\![\langle x, P, y\rangle)]\!]_G \subseteq [\![\text{RELAX}(x, P, y)]\!]_G$ there exists a pair $\langle\mu, k\rangle \in [\![h(\text{RELAX}(x', P', y'))]\!]_G$, for every G. Since for the containment to hold $0 \geq k = 0$ and therefore $\langle\mu, 0\rangle \in [\![h(\langle x', P', y'\rangle)]\!]_G$, we can conclude that $(x, P, y) \subseteq (x', P', y')$ and therefore $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. □

Approximation and Relaxation

We now consider queries with only one conjunct to which either RELAX or APPROX can be applied. We consider two cases for single-conjunct SPARQL^{AR} query containment given a renaming function $h = \{x' \to x, y' \to y\}$ and an ontology K:

$$\text{APPROX}(x, P, y) \subseteq_{op} \text{RELAX}(x', P', y') \tag{7.2.7}$$

$$\text{RELAX}(x, P, y) \subseteq_{op} \text{APPROX}(x', P', y') \tag{7.2.8}$$

7.2.7 op is =: is true only if $\mathcal{L}(A_P) \subseteq \mathcal{L}(R_{P'})$ and $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. □

7.2.7 op is $\geq$*:* is true only if $\mathcal{L}(A_P) \subseteq \mathcal{L}(R_{P'})$ and $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. □

7.2.8 op is =: is true only if $\mathcal{L}(R_P) \subseteq \mathcal{L}(A_{P'})$ and $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. □

7.2.8 op is $\geq$*:* is true only if $\mathcal{L}(R_P) \subseteq \mathcal{L}(A_{P'})$ and $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. By the definition of containment, with *op* being $\geq$, and Observation 7.1, for each $\langle\mu, 0\rangle \in [\![\langle x, P, y\rangle)]\!]_G \subseteq [\![\text{RELAX}(x, P, y)]\!]_G$ there exists a pair $\langle\mu, k\rangle$ such that $\langle\mu, k\rangle \in [\![h(\text{APPROX}(x', P', y'))]\!]_G$, for every G. Since for the containment to hold $0 \geq k$ then $k = 0$ and therefore $\langle\mu, 0\rangle \in [\![h(\langle x', P', y'\rangle)]\!]_G$. We conclude that $\langle x, P, y\rangle \subseteq \langle x', P', y'\rangle$ and therefore $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. □

Table 7.4 summarises the query containment conditions discussed in this Section, where t denotes a single triple pattern that is not approximated nor relaxed, A denotes an approximated triple pattern and R denotes a relaxed triple pattern. Our implementation of the query containment optimisations implements the tests in Table 7.4 relating to *op* being $\geq$ apart from those in the last three lines.

	op is $=$	op is $\geq$
$t \subseteq t$	iff $\mathcal{L}(P) \subseteq \mathcal{L}(P')$	
$t \subseteq A$	iff $\mathcal{L}(P) \subseteq \mathcal{L}(P')$	
$A \subseteq t$	iff $\mathcal{L}(A_P) = \mathcal{L}(P) \subseteq \mathcal{L}(P')$	iff $\mathcal{L}(A_P) \subseteq \mathcal{L}(P')$
$A \subseteq A$	iff $\mathcal{L}(P) = \mathcal{L}(P')$	iff $\mathcal{L}(P) \subseteq \mathcal{L}(P')$
$t \subseteq R$	iff $\mathcal{L}(P) \subseteq \mathcal{L}(P')$	
$R \subseteq t$	iff $\mathcal{L}(R_P) = \mathcal{L}(P) \subseteq \mathcal{L}(P')$	iff $\mathcal{L}(R_P) \subseteq \mathcal{L}(P')$
$R \subseteq R$	Only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$	
$A \subseteq R$	Only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ and $\mathcal{L}(A_P) \subseteq \mathcal{L}(R_{P'})$	
$R \subseteq A$	Only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ and $\mathcal{L}(R_P) \subseteq \mathcal{L}(A_{P'})$	

Table 7.4: Containment conditions

7.2.4 Query containment based optimisation

We now discuss how we exploit the containment property for query optimisation. Since the rewriting algorithm generates queries that are similar to each other and potentially return the same answers multiple times, we can check if some queries are contained in others. Algorithm 11 iterates over every query Q' generated by the rewriting algorithm and checks if there exists any other generated query Q'' that contains it. If that is the case then Q' will not be included in the set of queries to be evaluated.

Similarly to Algorithm 10, Algorithm 11 returns a set of queries that will be evaluated, hence we are able to combine the pre-computation optimisation with the query containment optimisation by simply replacing the call of the *rewrite* function in Algorithm 9 to call Algorithm 11 instead.

7.2.5 Performance Study

We now discuss the performance of SPARQL^{AR} evaluation with the query containment optimisation considering the LUBM, DBpedia and YAGO datasets.

Algorithm 11: Flexible Query Evaluation

input : Query Q; approx/relax max cost c; Graph G; Ontology K.

output: List Qs of query/cost pairs sorted by cost.

$Qs := \emptyset$;

$Q :=$ minimise(Q);

foreach $\langle Q', cost' \rangle \in$ *rew(Q,c,K)* **do**

  if *we cannot find any* $\langle Q'' cost'' \rangle \in$ *rew(Q,c,K) such that* $Q' \subseteq Q''$ *and* $cost' \geq cost''$ **then**

    $Q' :=$ minimise(Q');

    $Qs := Qs \cup \langle Q', cost' \rangle$;

return Qs;

Algorithm 12: Query Minimisation

input : Query Q.

output: minimised Q.

foreach *triple pattern* $t \in Q$ **do**

  if *there exists* $t' \in Q$ *such that* $t' \subseteq_{\geq} t$ **then**

    remove t from Q;

return Q;

LUBM Evaluation

In Table 7.5 the left-hand number in each cell is the number of queries generated by the rewriting algorithm with the query containment optimisation, and the right-hand number is the number of queries generated by the rewriting algorithm alone. In Table 7.5 we see that the number of queries that are returned by Algorithm 11 is lower in some cases than those returned by the rewriting algorithm (see Table 6.1). These queries are those that contain the APPROX operator (Q_1, Q_3, Q_4 and Q_5), since it is with such queries that the rewriting algorithm can generate multiple queries that may contain each other.

For example considering query Q_4:

```
SELECT * WHERE{
```

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	6/6	4/4	8/8	8/9	6/6	2/2	3/3
2	16/17	8/8	28/29	28/37	16/17	2/2	5/5
3	30/37	11/11	62/71	64/112	30/37	2/2	7/7

Table 7.5: LUBM. Number of queries generated by the rewriting algorithm with query containment optimisation, given maximum costs of 1, 2 and 3.

```
?z publicationAuthor AssociateProfessor3.
APPROX(?z publicationAuthor/advisor AssociateProfessor3)
}
```

The rewriting algorithm deletes the property *advisor* from the second triple pattern resulting in the following:

```
SELECT * WHERE{
?z publicationAuthor AssociateProfessor3.
?z publicationAuthor AssociateProfessor3
}
```

The rewriting algorithm also replaces the property *advisor* with the symbol (_) which results in the following:

```
SELECT * WHERE{
?z publicationAuthor AssociateProfessor3.
?z publicationAuthor/_ AssociateProfessor3
}
```

We notice that the answers returned by the second query are also returned by the first query. Moreover, both queries are generated at the same cost, hence we do not need to execute the second query.

By comparing Figures 7.7, 7.8, 7.9 with Figures 6.2, 6.3, 6.4 we notice that, for those queries that run in less than a fraction of a second using the simple evaluation,

the timing actually worsens when using the query containment optimisation. This is the case, for example, for queries Q_1, Q_2, Q_4, Q_6 and Q_7 for costs 1, 2 and 3, and dataset D1. This is due to the time it takes for Algorithm 11 to find which queries to discard when performing the query containment checks and to minimise the remaining queries. For queries that take more than one second with the simple evaluation we can see that the query containment optimisation does improve the evaluation time.

We found that the compilation time for query containment is between 0.03 seconds and 20 seconds, depending on the number and size of the queries the rewriting algorithm generates. For example, for query Q_3 the compilation time is 0.71, 4.5 and 7.8 seconds for cost 1, 2 and 3 respectively.

When the query containment optimisation is combined with the pre-computation optimisation, the execution time increases for queries that take less than 5 seconds (queries Q_1, Q_2, Q_4, Q_6 and Q_7 for all costs). However, for queries that take many seconds to execute there is an improvement of several seconds (see Q_3 for all costs). We also notice from Figure 7.9 that, in contrast to the summarisation optimisation, we are not able to execute queries Q_3, Q_4 and Q_5. This is due to the presence of the _ symbol which is expensive to execute.

DBpedia Evaluation

Applying the query containment optimisation to the DBpedia queries, we can see from Table 7.6 that there has been a reduction of the number of queries with respect to the simple rewriting algorithm (see Table 6.3), with the exception of query Q_4 which contains only the RELAX operator.

We also notice that for query Q_5, even though it contains two approximated triples, the number of queries that are discarded is considerably lower than the number discarded for query Q_4 from LUBM which contains only one approximated triple pattern. This is due to the fact that the approximated triple patterns in Q_5 contain constants or variables appearing in the head of the query hence the containment check fails to find a mapping between queries.

By comparing Figure 7.10 with Figure 6.5, we can see an overall improvement in

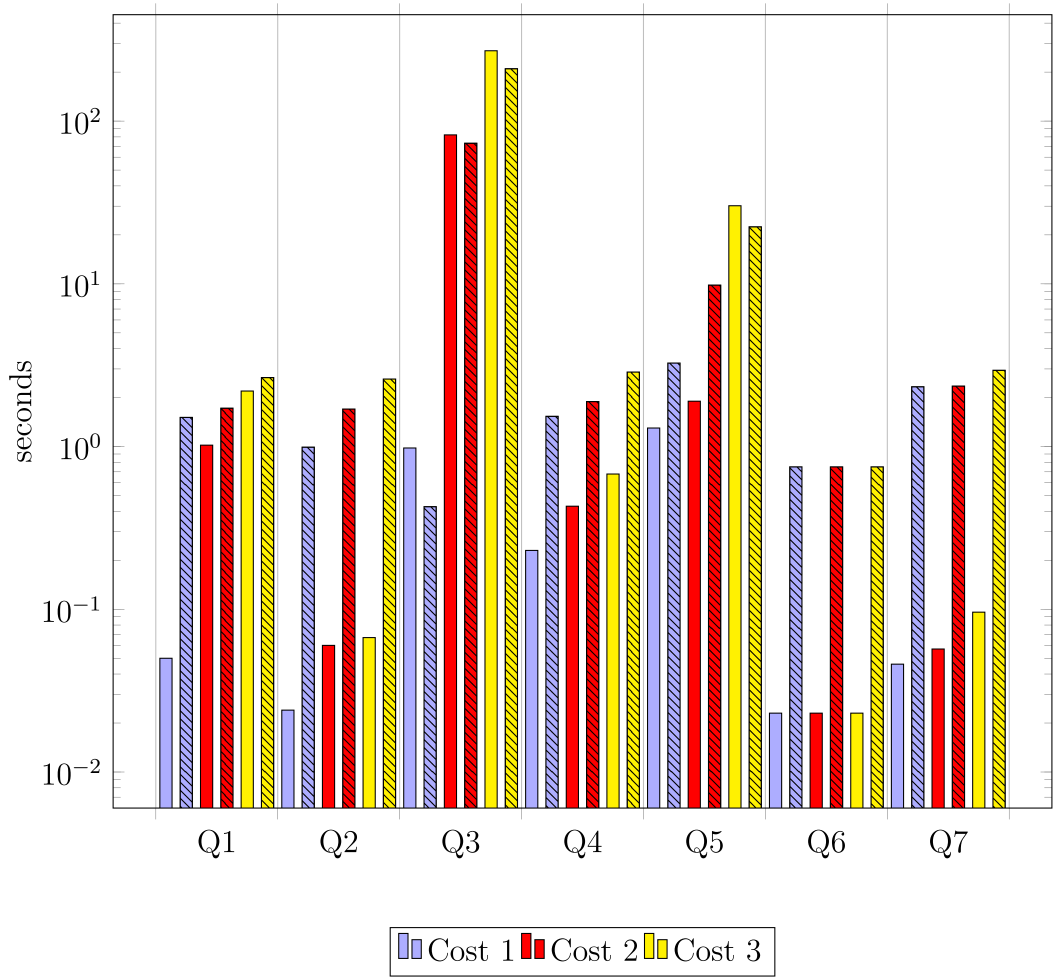


Figure 7.7: LUBM. Timings for database D1, with query containment optimisation, with and without pre-computation optimisation.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	3/3	5/5	5/5	3/3	10/10
2	7/11	11/12	11/12	6/6	46/48
3	15/51	19/25	19/25	10/10	146/168

Table 7.6: DBpedia. Number of queries generated by the rewriting algorithm given maximum costs of 1, 2 and 3.

execution time for most queries. In particular we are able to execute query Q_5 with maximum cost 2 and 3 within the 8 hour threshold since we reduced the number of queries to be executed from 48 to 46 for cost 2 and from 168 to 146 for cost 3.

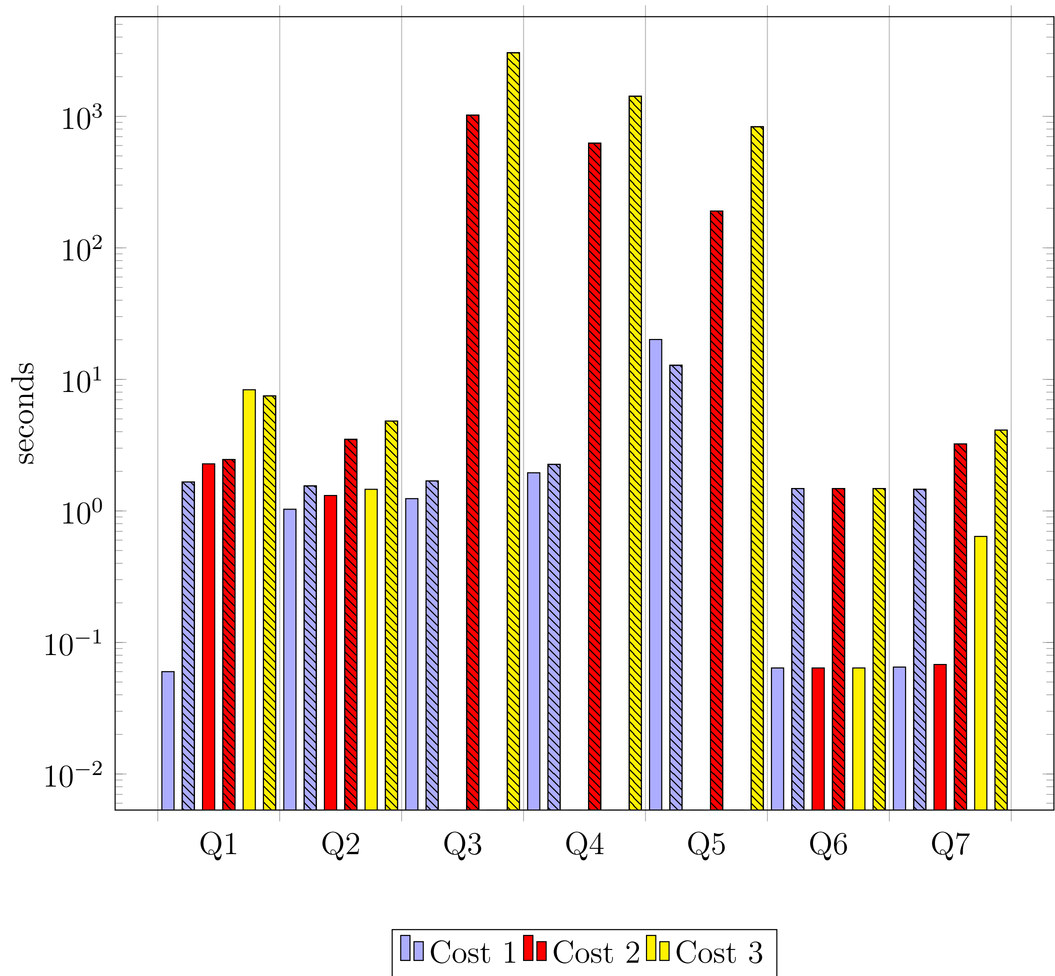


Figure 7.8: LUBM. Timings for database D2, with query containment optimisation, with and without pre-computation optimisation.

Regarding query Q_4, even though the number of queries has not changed it took more time to execute due to the overhead of computing query containments.

Similarly to LUBM the compilation time for the DBpedia queries is between 0.04 and 25 seconds. For example, the compilation time for query Q_5 is 0.3, 1.2 and 4.5 seconds for cost 1, 2 and 3 respectively.

YAGO Evaluation

Similarly to the analysis for DBpedia and LUBM, we can see from Table 7.7 that the number of queries to be evaluated is lower than with the simple rewriting algorithm. The only exception is query Q_2 which contains only one relaxed triple pattern.

By comparing Figure 7.11 with Figure 6.6 again we notice an improvement in

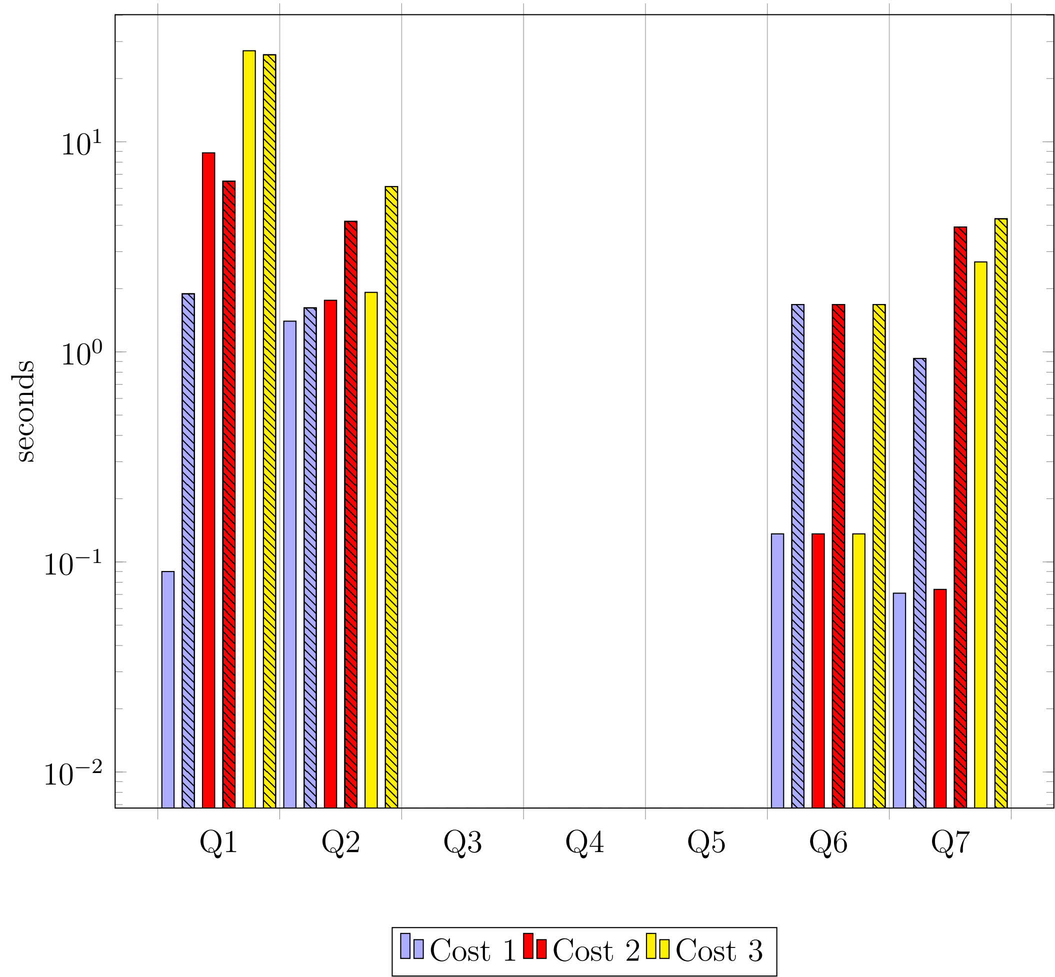


Figure 7.9: LUBM. Timings for database D3, with query containment optimisation, with and without pre-computation optimisation.

Max Cost	Q_1	Q_2	Q_3
1	12/12	2/2	5/5
2	57/60	2/2	11/12
3	166/199	2/2	19/25

Table 7.7: YAGO. Number of queries generated by the rewriting algorithm given maximum costs of 1, 2 and 3.

the execution time for Q_1 and Q_3. For example, query Q_1 for maximum cost 2 now takes only 434 seconds instead of 503. Unfortunately, we are still not able to execute query Q_1 with maximum cost 3 within the 8 hour threshold.

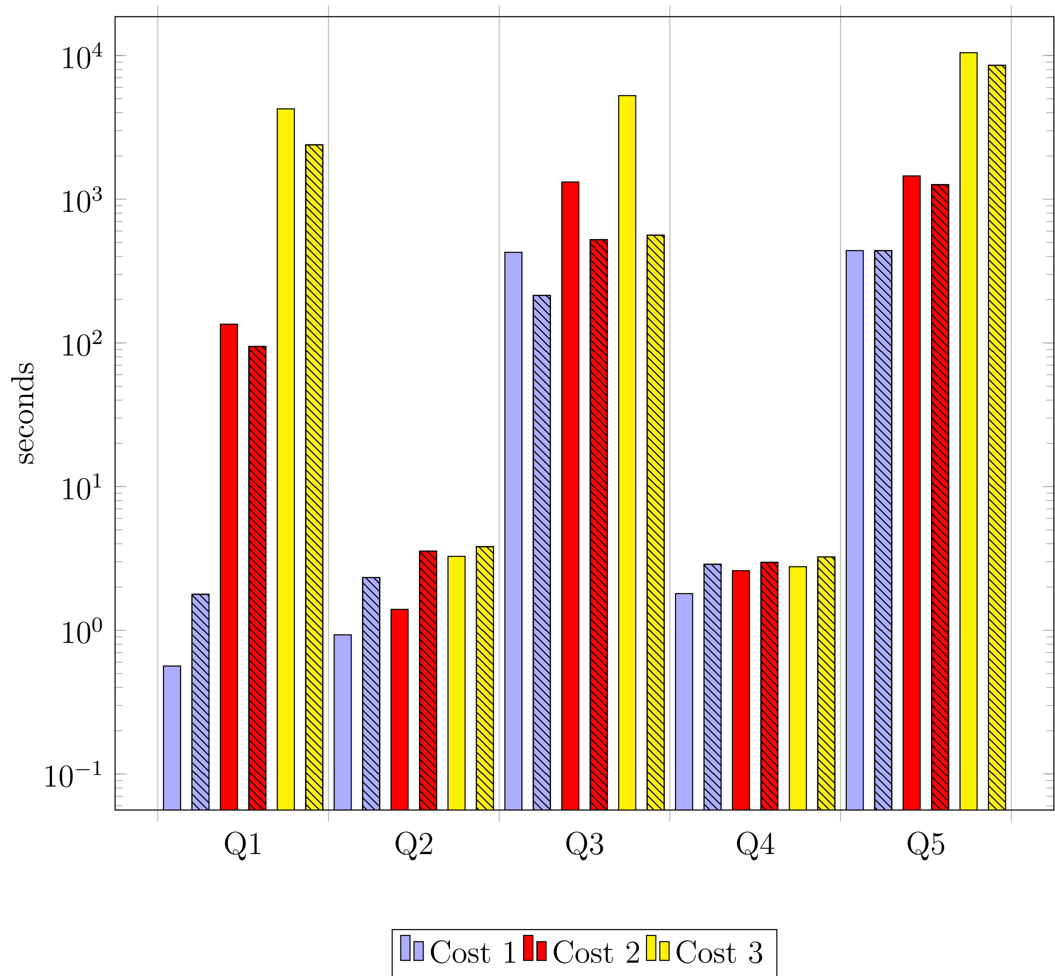


Figure 7.10: DBpedia. Timings, with query containment optimisation, with and without pre-computation optimisation.

The compilation time for the YAGO queries is approximately between 0.04 and 20 seconds. For example, for query Q_3 the compilation time is 0.1, 0.2, 0.5 seconds for cost 1, 2 and 3 respectively. These relatively low figures are mainly due to the fact that the rewriting algorithm generates a small number of queries from Q_3.

7.3 Combined Optimisations

We also tested the queries from the three datasets by combining all three optimisations: the pre-computation optimisation, the summarisation optimisation and the query containment optimisation. We are able to combine the summarisation optimisation with the query containment optimisation by replacing the *rewrite* function

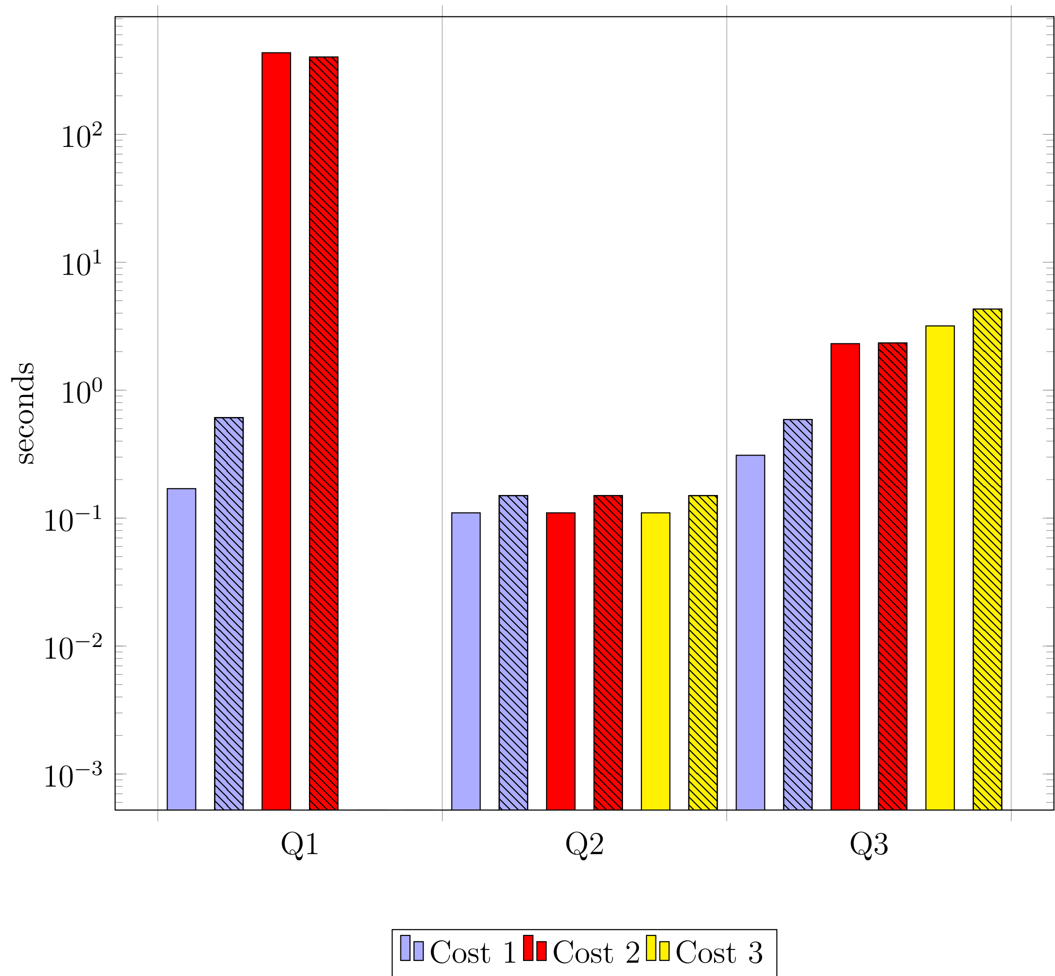


Figure 7.11: YAGO. Timings with query containment optimisation, with and without pre-computation optimisation.

in Algorithm 10 with Algorithm 11. The pre-computation optimisation is included by Algorithm 9 calling Algorithm 10 which finally calls Algorithm 11.

7.3.1 Performance Study

LUBM Evaluation

When combining the summarisation optimisation technique with query containment, we can see in Table 7.8 a further reduction in the number of queries generated with respect to the summarisation optimisation alone (see Figure 7.1). For example we reduced the number of queries generated by the rewriting algorithm for query Q_4 from 112 to 37. This leads to a further improvement in the query evaluation timings

for queries containing the APPROX operator. The summarisation optimisation technique is more effective for this kind of query since it optimises the query by editing the regular expression patterns of the query.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	2	1	7	5	2	1	2
2	7	2	20	17	7	1	3
3	14	2	36	37	14	1	4

Table 7.8: LUBM. Number of queries generated by the rewriting algorithm with summarisation of size 3 and query containment.

In Figures 7.12, 7.13, 7.14 we can see that, for most queries, the pre-computation optimisation increases the execution time. This is due to the time needed to compute the sub-queries and cache their results.

Overall, by combining all three optimisation, we reduced the timings of the LUBM queries. All the queries run in less than 20 seconds for database D1, less than 60 seconds for database D2, and less than 400 seconds for database D3.

DBpedia Evaluation

For the DBpedia dataset the number of queries generated by combining the two optimisations does not change. Hence, we do not expect to see an improvement in the query execution times. One of the issues in combining the two optimisations is that we are still not able to generate certain queries using the optimisation techniques (Q_1 and Q_4 with maximum costs 2 and 3).

By comparing Figure 7.15 with Figure 7.4 showing the execution times with the summarisation optimisation we notice that the times to execute the queries do increase when we do not use the pre-computation optimisation in both instances. This is mainly due to the time taken to compute the query containments.

When we evaluate the queries using the pre-computation optimisation the time does improve for queries Q_1, Q_2 and Q_3 when we combine the summarisation opti-

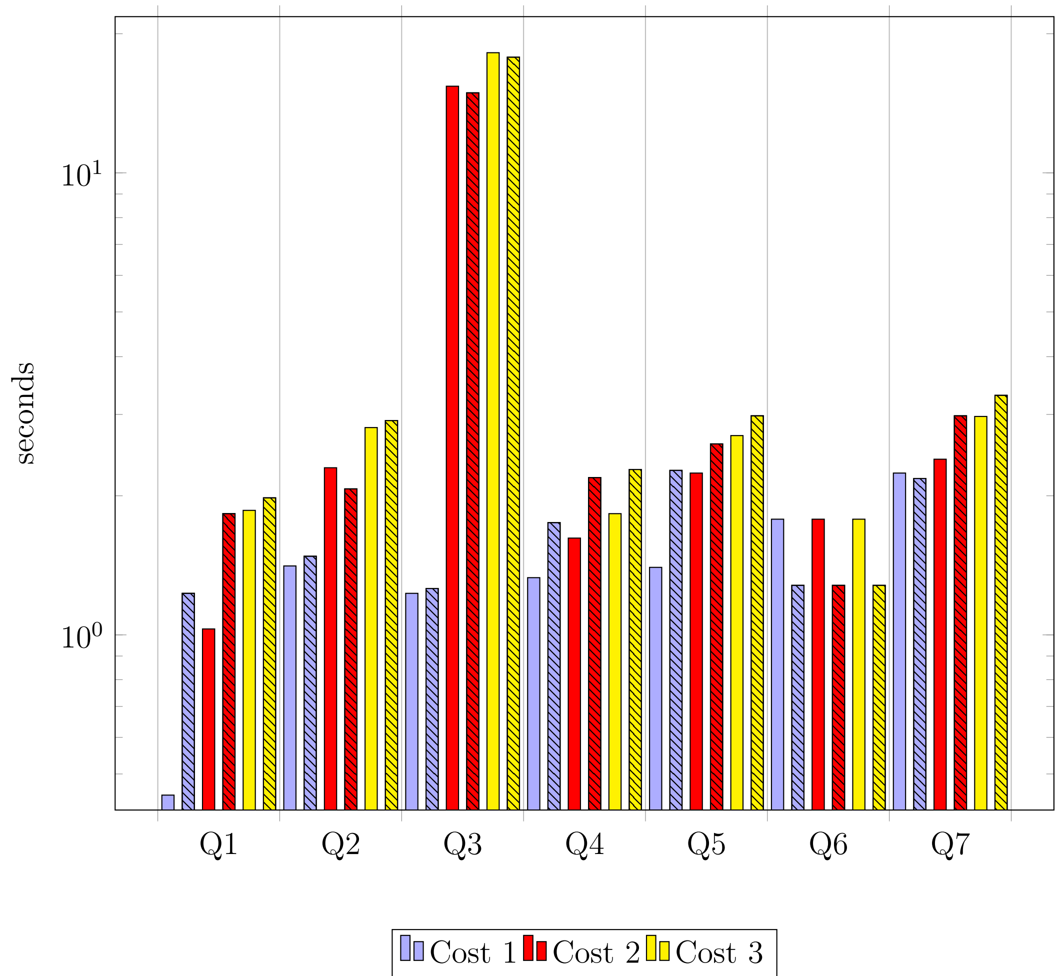


Figure 7.12: LUBM. Timings for database D1, with query containment optimisation, summarisation of size 3, with and without pre-computation optimisation.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	3	2	2	1	0
2	N/A	5	5	N/A	3
3	N/A	10	10	N/A	18

Table 7.9: LUBM. Number of queries generated by the rewriting algorithm with summarisation of size 2 and query containment.

misation and the query containment optimisation compared to the summarisation optimisation alone.

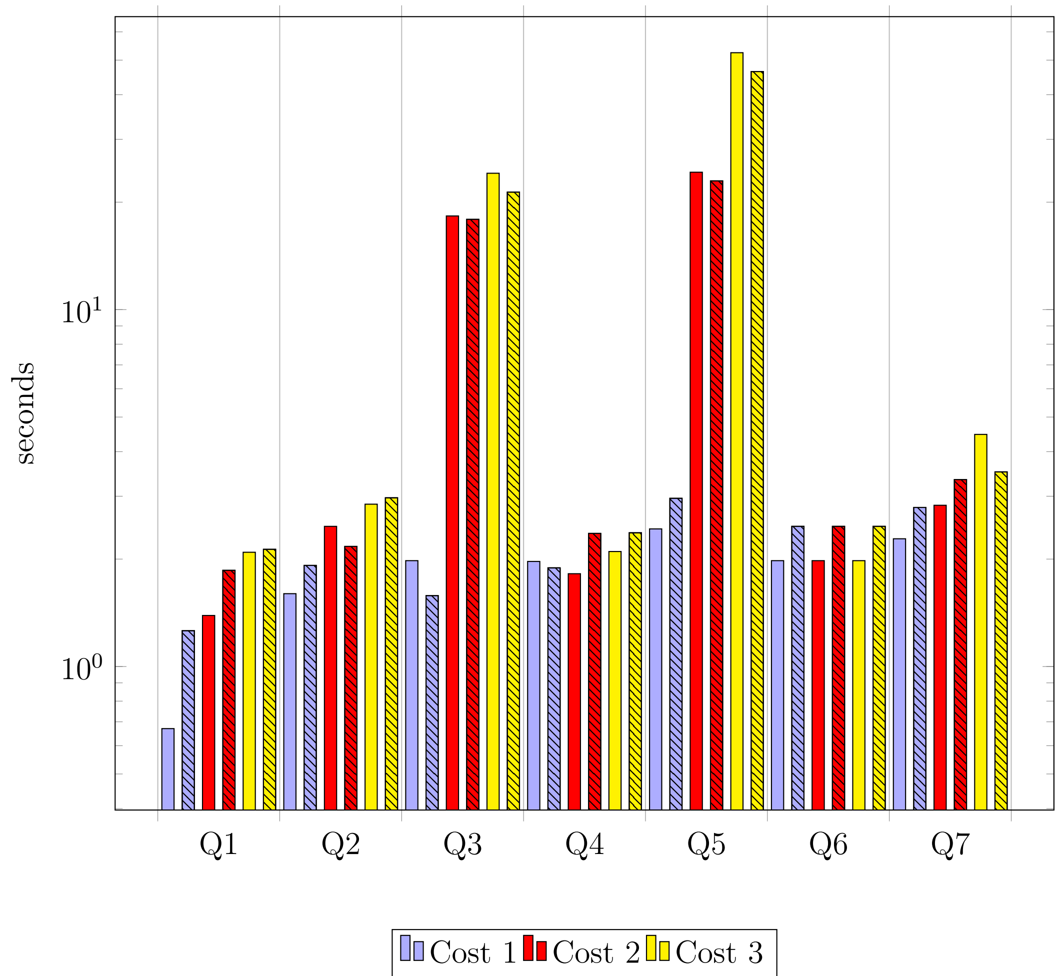


Figure 7.13: LUBM. Timings for database D2, with query containment optimisation, summarisation of size 3, with and without pre-computation optimisation.

YAGO Evaluation

In the case of the YAGO dataset we can see a slight improvement in the number of queries generated by the combined optimisations. In particular, by comparing query Q_3 from Table 7.10 with Table 7.3 we can see that there is a reduction in the number of queries. Particularly, we reduced the number of queries generated for query Q_3 from 25 to 19.

Finally, by comparing the execution times of Figure 7.16 with Figure 7.5 we notice that there is no improvement for queries Q_2 and Q_3 due to the overhead of the query containment optimisation. Regarding query Q_1 we do notice an improvement in the execution time for maximum cost 2 from 214 seconds, by using the

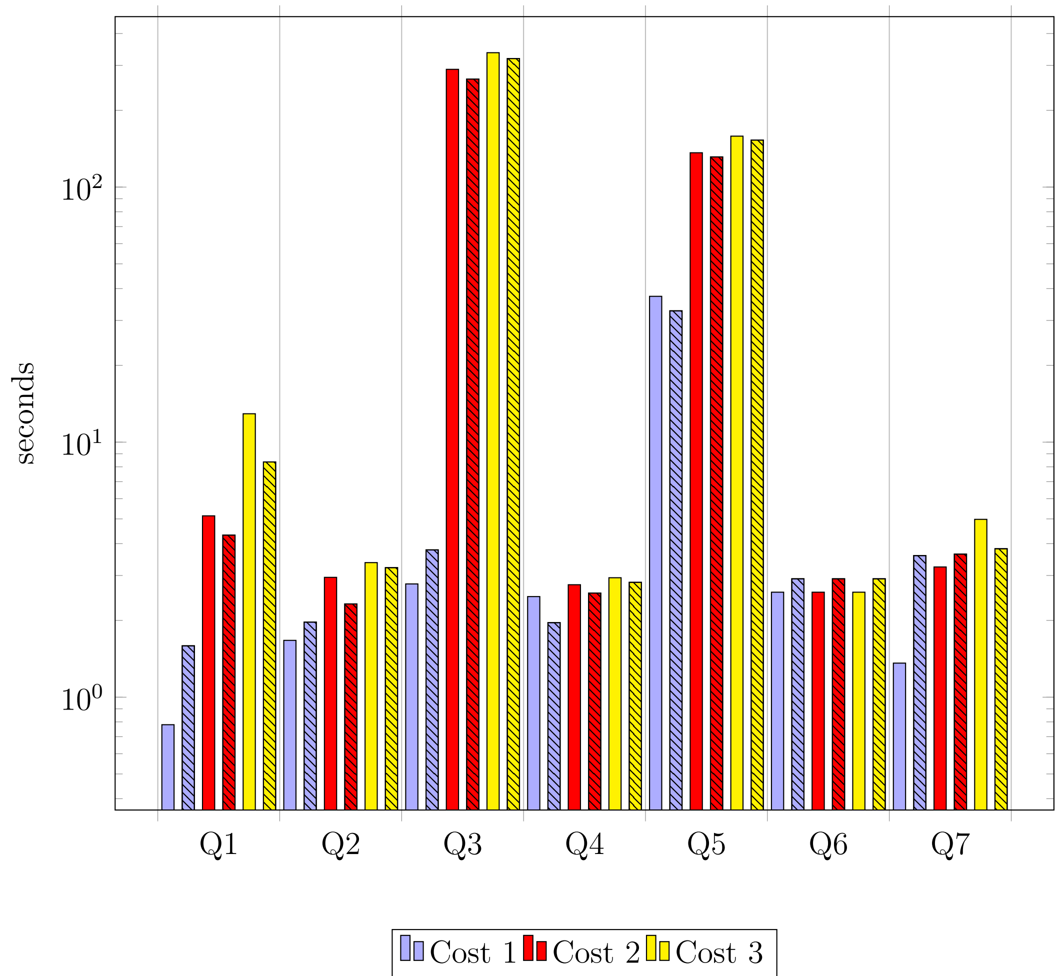


Figure 7.14: LUBM. Timings for database D3, with query containment optimisation, summarisation of size 3, with and without pre-computation optimisation.

Max Cost	Q_1	Q_2	Q_3
1	10	2	5
2	45	2	11
3	N/A	2	19

Table 7.10: YAGO. Number of queries generated by the rewriting algorithm with summarisation size 3 and query containment.

summarisation of size 3, to 173 seconds, by using both the query containment and the summarisation of size 3; and from 201, by using the summarisation of size 3 and the pre-computation optimisation, to 152, by using all three optimisations.

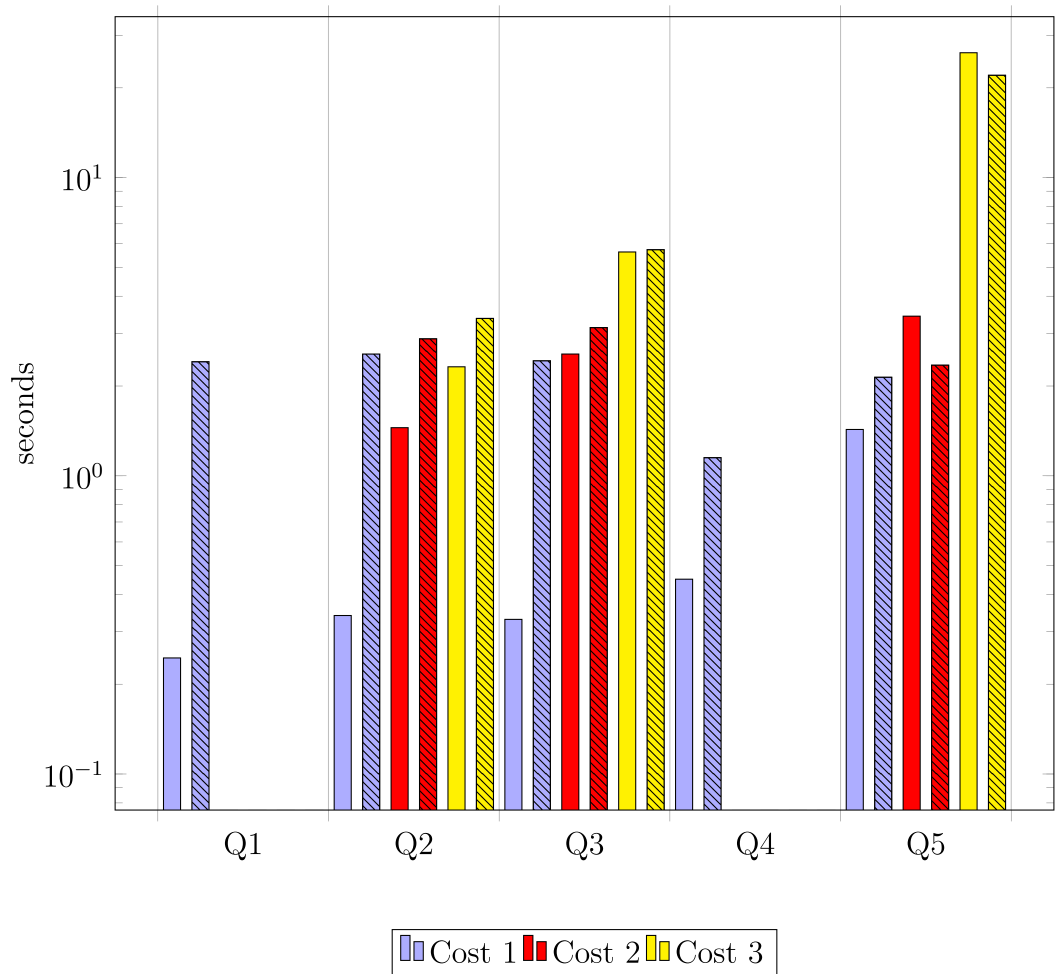


Figure 7.15: DBpedia. Timings, with query containment optimisation, summarisation of size 2, with and without pre-computation optimisation.

7.4 Discussion

We have presented in this chapter two optimisation techniques that have improved in most cases the query execution timings. By means of the summarisation optimisation we removed unsatisfiable queries and rewrote queries containing the _ symbol in such a way that they are less expensive to execute. We also removed queries that were not adding any additional answers and minimised queries by the means of the query containment technique.

In contrast to the work in [16], our summarisation technique is not used to help the user to formulate queries; moreover, their node collapsing technique may generate non-deterministic graph summaries. The summarisation in [35] tries to

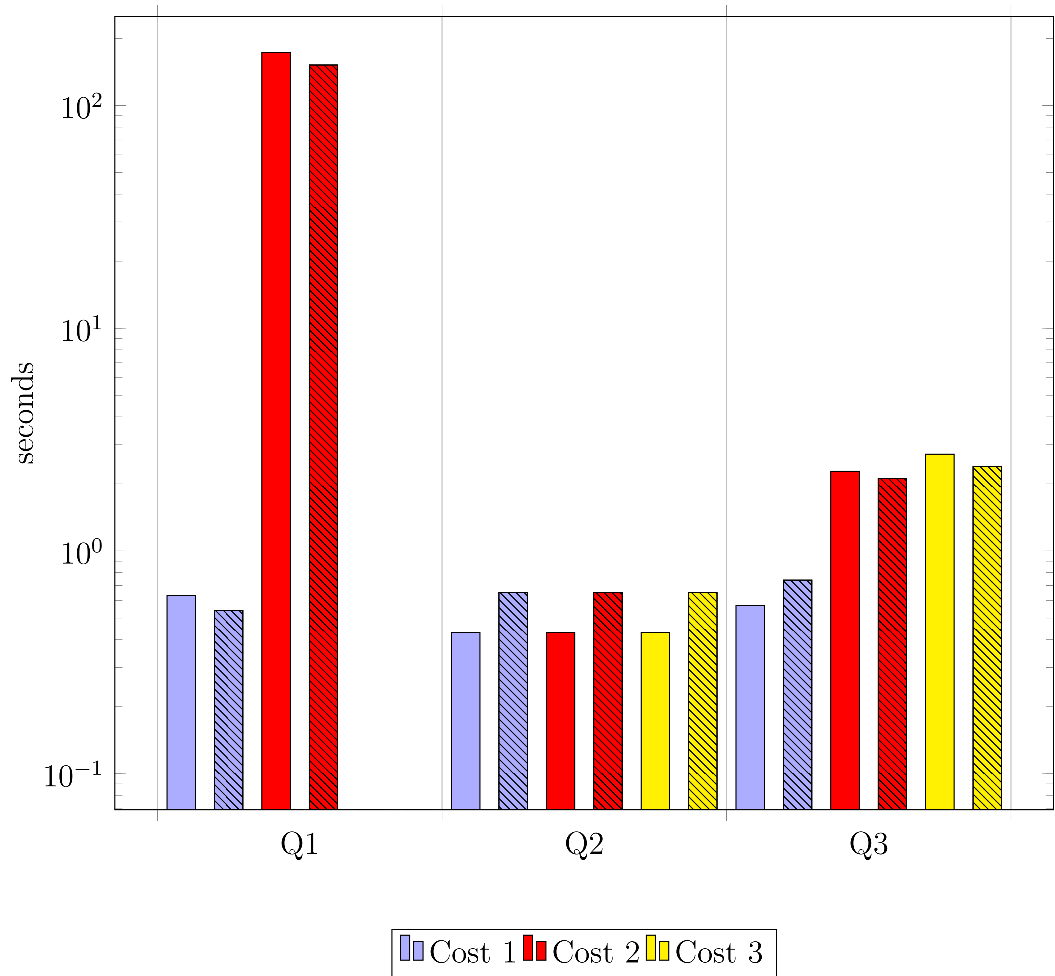


Figure 7.16: YAGO. Timings, with query containment optimisation, summarisation of size 3, with and without pre-computation optimisation.

improve query performance by pruning parts of the RDF graph. They show that their technique does improve the performance for most query instances, with the exception of single-conjunct queries, whereas in our investigation the performance does not improve when the APPROX operator is not contained in the query. In contrast to [5, 15], our work does not focus on summarisation precision and size, although this can be explored in future work.

We evaluated all the queries from Chapter 6 using both the new query optimisation techniques presented, alone and combined. Moreover, we used the precomputation technique to further improve the execution time of queries by combining it with the other two optimisation techniques.

We conclude that overall the summarisation optimisation technique does improve

the execution time for non-dense datasets and for queries that contain the APPROX operator. The query containment optimisation reduced the number of queries to be executed in many cases. Similar results can be found in [33] where the authors show the benefit of applying query elimination by using query containment techniques, although the number of queries they generate via their rewriting algorithm can grow up to 100,000.

By combining the two approaches we reduced the time to execute the worst performing queries considerably. One of the drawbacks of our combined approach is that the overhead to compute query containment and to rewrite queries into a less costly form is rather high. Hence, some queries that ran in less than one second using the simple evaluation algorithm run in more than one second when combining the optimisations. In particular, this overhead is more noticeable when including also the pre-computation technique.

Further work might include design of a query optimiser that chooses to apply one or more optimisations depending on the query structure and size. For example, single-conjunct queries do not need the pre-computation optimisation as they cannot be split into sub-queries. Similarly, the execution time of queries that contain only one triple pattern may not improve when using the query containment optimisation. The evaluation times of queries with complex query patterns or with the APPROX operator are likely to be improved by the means of the summarisation optimisation. On the other hand, this technique may not be useful when querying a dataset that has large summarisations such as DBpedia.

Chapter 8

Conclusions

We have discussed the motivation of our work by showing, with several examples, how flexible querying techniques have the potential to enhance users' access to complex, heterogeneous datasets, by allowing the retrieval of non-exact answers. We have presented our new flexible querying language for RDF called SPARQL^{AR} that extends a fragment of the standard SPARQL 1.1 query language. This new language extends the work in [67] by integrating APPROX and RELAX operators into SPARQL 1.1 which is a pragmatic language for querying RDF. We have defined the syntax and semantics of SPARQL^{AR} focusing particularly on its APPROX operator, that edits the property paths of the query by inserting, deleting and replacing properties, and its RELAX operator that allows ontology-driven relaxation.

We have extended the complexity results in [2, 65] for SPARQL by investigating the complexity of several fragments of SPARQL^{AR} and have shown that including the APPROX and RELAX operators does not increase the complexity of query evaluation compared with the standard SPARQL language.

We have implemented a query evaluation algorithm for SPARQL^{AR} based on a query rewriting technique. The algorithm was inspired by the work in [42, 43] which considers only relaxation, whereas our algorithm is able to evaluate queries containing both relaxation and approximation operators. We have shown the correctness and termination of our query rewiring algorithm and have evaluated its empirical efficiency. We have also described an optimisation technique where we pre-compute partial queries and store their answers in a cache for subsequent reuse.

We presented a performance study of our query evaluation algorithm and the pre-computation optimisation over three datasets, LUBM, DBpedia and YAGO. We noticed that the pre-computation technique does help to improve the query answering times for queries that generally take a long time to compute. But, because of the overhead needed to run the pre-computation, for the faster queries the running time actually increased.

We presented two further optimisation techniques with the aim of improving the query execution timings further. For the first of these, we presented a procedure that generates a summary of the original RDF dataset. We use a similar summarisation to that defined in [5] where the authors try to reduce the size of an RDF graph so that is more comprehensible; in contrast, our summarisation is used to enhance query evaluation performance. We showed that we are able to exploit such a summary in order to reduce the number of queries that need to be executed to evaluate a SPARQL^{AR} query. Moreover, we were able to optimise the queries that need to be executed by replacing the symbol _, which is expensive to execute since it represents the disjunction of all the predicates in the dataset, with a selected number of predicates. We discussed how this optimisation technique enhances query execution performance considerably, in particular when the query contains the APPROX operator. A major drawback of this technique is that when the dataset contains many different property labels it leads to larger summarisations, which in turn causes rewritten queries to be syntactically larger. For the DBpedia dataset in particular we were unable to test the summarisation optimisation technique with summarisations of size 3, since it caused the query to be re-written into a form that was too large to be executed by the Jena framework.

Our third optimisation technique is based on query containment, by means of which we are able to discard multiple queries that do not increase the number of answers. This optimisation technique, although based on the work in [33], was further extended by considering query containment in the presence of APPROX and RELAX operators. We have presented sufficient conditions that ensure containment of single triple patterns for SPARQL^{AR} queries. We have shown how the query containment optimisation reduces the number of queries that need to be executed,

hence reducing the overall computation time of queries. This technique reduced the execution time of, in particular, queries containing the APPROX operator. However, we were still unable to run some queries because of the presence of the _ symbol. Hence, we concluded our performance study by combining both the summarisation and the query containment optimisations and we showed improvement in the query execution time of several seconds for all three datasets.

8.1 Future Work

In Chapter 7 we suggested the need for an optimiser that chooses which optimisation technique to use based on the query structure and size. The optimiser may choose to use the summarisation optimisation technique for queries with complex regular expression patterns, or choose to use the pre-computation optimisation for queries with many triple patterns. Further optimisation of our SPARQL^{AR} prototype implementation and its interaction with the Jena API might also improve the execution time of queries.

As future work we intend to evaluate the quality of the answers returned by SPARQL^{AR}, by undertaking a user evaluation study. Our aim is to identify how many answers are considered useful by users in given information seeking scenarios after posing an initial exact query followed by a sequence of relaxation or approximation steps.

We also plan to extend our language in two ways. The first is by combining the APPROX and RELAX operators into a single operator we call FLEX that applies both the edit operations of APPROX and the RDFS entailment rules of RELAX to a single query conjunct. We intend to investigate its semantics and complexity, and how it can be used to enhance the benefits of flexible querying for users by allowing further flexibility when formulating queries.

We will also investigate another new operator that, we believe, will further help the user to query a dataset. To retrieve information from any RDF dataset, the user may start from the *label* property to find matching resources. An example query is the following:

SELECT * WHERE ?x rdfs:label "SPARQL"

This style of querying may lead to empty results due to use of erroneous capitalisation, spelling errors, redundant spaces, use of synonym words and generally lack of precision by the user within the string literal. Therefore, we aim to devise a similarity measure with respect to strings and text that enables such queries to be approximated, making use of the predicates that connect resources to literals such as *rdfs:label* and *rdfs:comment*. We intend to exploit text-similarity techniques that are used in natural language processing such as corpus-based word similarity [47].

Our similarity querying will allow both text similarity and semantic similarity. For the latter we intend to exploit an ontology reasoner. By using the ontology axioms we can infer new information that vocabularies and thesauri may not provide. These ontology axioms will be related to the domain of interest. For instance, from the resource description "Dutch Painting with sunflowers" we can infer that it is a painting painted by Van Gogh.

Another direction of research is the extension of our approximation and relaxation operators, query evaluation and query optimisation techniques to flexible *federated* query processing for SPARQL 1.1. Querying multiple data sources is potentially harder for the user, hence the APPROX and RELAX operators might aid the user to query such potentially heterogeneous RDF datasets. For example, the APPROX operator might replace the predicates of a query with predicates selected from the namespace of other RDF datasets. Similarly the RELAX operator might replace a predicate of the query with a super-predicate that generalises predicates from multiple datasets.

We intend to further extend our query containment study by investigating conditions that ensure query containment for multi-conjunct SPARQL^{AR} queries. Moreover, we will investigate in more detail the complexity of query containment for fragments of SPARQL^{AR}.

Finally, another direction of research might be to investigate the use of *path indexing* for RDF-graphs [29]. Use of path indexes may further improve our query evaluation timings, particularly for queries containing complex regular expression patterns. Our SPARQL^{AR} language may benefit from such indexes since the AP-

PROX operator in particular generates regular expression patterns with long concatenations of URIs.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *Web Semant.*, 7(2):57–73, Apr. 2009.

[3] S. Almagor, U. Boker, and O. Kupferman. What's decidable about weighted automata. In *In Automated Technology for Verification and Analysis, Lecture Notes in Computer Science*, pages 482–491, 2011.

[4] J. M. Almendros-Jiménez, A. Luna, and G. Moreno. Fuzzy XPath queries in XQuery. In *On the Move to Meaningful Internet Systems: OTM 2014 Conference Proceedings - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31*, pages 457–472, 2014.

[5] A. Alzogbi and G. Lausen. Similar structures inside RDF-Graphs. In C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas, and S. Auer, editors, *LDOW*, volume 996 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[6] R. Angles and C. Gutierrez. The expressive power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference*, ISWC '08, pages 114–129, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] C. Bizer, R. Cyganiak, and T. Heath. How to publish Linked Data on the Web. Web page, 2007. Revised 2008. Accessed 22/02/2010.

[8] G. Bordogna and G. Psaila. Customizable flexible querying in classical relational databases. In J. Galindo, editor, *Handbook of Research on Fuzzy Information Processing in Databases*, pages 191–217. IGI Global, 2008.

[9] P. Bosc and O. Pivert. SQLf: A relational database language for fuzzy querying. *Trans. Fuz Sys.*, 3(1):1–17, Feb. 1995.

[10] J. Broekstra and A. Kampman. *An RDF Query and Transformation Language*, pages 23–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[11] C. Buil-Aranda, M. Arenas, and O. Corcho. Semantics and optimization of the SPARQL 1.1 federation extension. In *Proceedings of the 8th Extended Semantic Web conference on The semanic web: research and applications - Volume Part II*, ESWC'11, pages 1–15, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] A. Calì, R. Frosini, A. Poulovassilis, and P. T. Wood. Flexible querying for SPARQL. In *On the Move to Meaningful Internet Systems: OTM 2014 Conference Proceedings - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31*, pages 473–490, 2014.

[13] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 176–185, 2000.

[14] A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi, and P. Spoletini. A fuzzy extension of the XPath query language. *J. Intell. Inf. Syst.*, 33(3):285–305, Dec. 2009.

[15] S. Campinas, R. Delbru, and G. Tummarello. Efficiency and precision tradeoffs in graph summary algorithms. In *Proceedings of the 17th International Database Engineering and Applications Symposium*, IDEAS '13, pages 38–47, New York, NY, USA, 2013. ACM.

[16] S. Campinas, T. E. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello. Introducing RDF graph summary with application to assisted SPARQL formulation.

In *Proceedings of the 23rd International Workshop on Database and Expert Systems Applications*, DEXA '12, pages 261–266, Washington, DC, USA, 2012. IEEE Computer Society.

[17] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th International World Wide Web conference*, pages 74–83. ACM, 2004.

[18] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaida. PSPARQL Query Containment. Research report, EXMO - INRIA Grenoble Rhône-Alpes / LIG Laboratoire d'Informatique de Grenoble , WAM - INRIA Grenoble Rhône-Alpes / LIG Laboratoire d'Informatique de Grenoble, June 2011.

[19] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. SPARQL query containment under RDFS entailment regime. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, IJCAR'12, pages 134–148, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. SPARQL Query Containment under SHI Axioms. Rapport de recherche, Apr. 2012.

[21] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 134–144, New York, NY, USA, 2003. ACM.

[22] W. W. W. Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

[23] R. De Virgilio, A. Maccioni, and R. Torlone. A similarity measure for approximate querying over RDF data. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 205–213, New York, NY, USA, 2013. ACM.

[24] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Revised Papers from the 8th International Workshop on Database Programming Languages*, DBPL '01, pages 21–39, London, UK, UK, 2002. Springer-Verlag.

[25] S. Elbassuoni, M. Ramanath, and G. Weikum. Query relaxation for entity-relationship search. In *Proceedings of the 8th Extended Semantic Web Conference on The Semanic Web: Research and Applications - Volume Part II*, ESWC'11, pages 62–76, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] B. Fazzinga, S. Flesca, and A. Pugliese. Top-k approximate answers to xpath queries with negation. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2561–2573, 2014.

[27] R. Fink and D. Olteanu. On the optimal approximation of queries using tractable propositional languages. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 174–185, New York, NY, USA, 2011. ACM.

[28] S. Flesca, F. Furfaro, et al. Xpath query relaxation through rewriting rules. *IEEE transactions on knowledge and data engineering*, 23(10):1583–1600, 2011.

[29] G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 636–639, 2016.

[30] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 139–148, New York, NY, USA, 1998. ACM.

[31] R. Frosini, A. Calì, A. Poulovassilis, and P. T. Wood. Flexible query processing for SPARQL. *Semantic Web*, 8(4):533–563, 2017.

[32] T. Gaasterland. Cooperative answering through controlled query relaxation. *IEEE Expert: Intelligent Systems and Their Applications*, 12(5):48–59, Sept. 1997.

[33] G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization (extended version). *CoRR*, abs/1112.0343, 2011.

[34] G. Grahne and A. Thomo. Query answering and containment for regular path queries under distortions. In D. Seipel and J. Turull-Torres, editors, *Foundations of Information and Knowledge Systems*, volume 2942 of *Lecture Notes in Computer Science*, pages 98–115. Springer Berlin Heidelberg, 2004.

[35] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Using graph summarization for join-ahead pruning in a distributed RDF engine. In *Proceedings of Semantic Web Information Management on Semantic Web Information Management*, SWIM'14, pages 41:1–41:4, New York, NY, USA, 2014. ACM.

[36] C. Gutierrez, C. A. Hurtado, and A. O. Mendelzon. Foundations of Semantic Web databases. In *Proceedings of the Twenty-third Symposium on Principles of Database Systems (PODS), June 14-16, 2004, Paris, France*, pages 95–106, 2004.

[37] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A comparison of RDF query languages. In *International Semantic Web Conference*, pages 502–517. Springer, 2004.

[38] O. Hassanzadeh, A. Kementsietsidis, and Y. Velegrakis. Data management issues on the Semantic Web. In *Proceedings of the 28th IEEE International Conference on Data Engineering*, ICDE '12, pages 1204–1206, Washington, DC, USA, 2012. IEEE Computer Society.

[39] J. Heer, M. Agrawala, and W. Willett. Generalized selection via interactive query relaxation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 959–968, New York, NY, USA, 2008. ACM.

[40] J. Hjelm. *Creating the Semantic Web with RDF: Professional Developer's Guide*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[41] A. Hogan, M. Mellotte, G. Powell, and D. Stampouli. Towards fuzzy query-relaxation for RDF. In E. Simperl, P. Cimiano, A. Polleres, O. Corcho, and V. Presutti, editors, *The Semantic Web: Research and Applications*, volume

7295 of *Lecture Notes in Computer Science*, pages 687–702. Springer Berlin Heidelberg, 2012.

[42] H. Huang and C. Liu. Query relaxation for star queries on RDF. In *Proceedings of the 11th International Conference on Web Information Systems Engineering*, WISE'10, pages 376–389, Berlin, Heidelberg, 2010. Springer-Verlag.

[43] H. Huang, C. Liu, and X. Zhou. Computing relaxed answers on RDF databases. In *Proceedings of the 9th International Conference on Web Information Systems Engineering*, WISE '08, pages 163–175, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Query relaxation in RDF. *Journal on Data Semantics*, X:31–61, 2008.

[45] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Ranking approximate answers to semantic web queries. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, pages 263–277, Berlin, Heidelberg, 2009. Springer-Verlag.

[46] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 174–185, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[47] A. Islam and D. Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Trans. Knowl. Discov. Data*, 2(2):10:1–10:25, July 2008.

[48] Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS reasoning and query answering on top of DHTs. In *International Semantic Web Conference*, pages 499–516. Springer, 2008.

[49] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *Proceedings of the 11th In-*

ternational Conference on World Wide Web, WWW '02, pages 592–603, New York, NY, USA, 2002. ACM.

[50] G. Karypis and V. Kumar. METIS – unstructured graph partitioning and sparse matrix ordering system, Version 2.0. Technical report, -, 1995.

[51] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *International Conference on Data Engineering*, pages 129–140, 2002.

[52] C. Kiefer, A. Bernstein, and M. Stocker. The fundamentals of iSPARQL: a virtual triple approach for similarity-based semantic web tasks. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference*, ISWC'07/ASWC'07, pages 295–309, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] O. Lassila, R. R. Swick, and World Wide Web Consortium. Resource description framework (RDF) model and syntax specification, 1998.

[54] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[55] F. Mandreoli, R. Martoglia, and P. Tiberio. Approximate query answering for a heterogeneous xml document base. In *International Conference on Web Information Systems Engineering*, pages 337–351. Springer, 2004.

[56] D. Marin. RDF formalization. Technical report, Santiago de Chile, Technical Report Universidad de Chile, 2004. TR/DCC-2006-8.

[57] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Proceedings of the 29th International Symposium on Mathematical Foundations of Computer Science (MFCS 2004*, pages 889–900. Springer, 2004.

[58] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM J. Comp*, 2009.

[59] B. McBride. The resource description framework (RDF) and its vocabulary description language RDFS. In S. Staab and R. Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 51–66. Springer, 2004.

[60] R. Meusel, P. Petrovski, and C. Bizer. The WebDataCommons Microdata, RDFa and Microformat Dataset Series. In *Proceedings of the 13th International Semantic Web Conference - Part I*, ISWC '14, pages 277–292, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[61] S. Muñoz, J. Pérez, and C. Gutierrez. Minimal deductive systems for RDF. In *Proceedings of the 4th European Conference on The Semantic Web: Research and Applications*, ESWC '07, pages 53–67, Berlin, Heidelberg, 2007. Springer-Verlag.

[62] P. Hayes, editor. RDF 1.1 Semantics, 2014.

[63] M. T. Pazienza, A. Stellato, and A. Turbati. Linguistic watermark 3.0: an RDF framework and a software library for bridging language and ontologies in the semantic web.

[64] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference*, ISWC'06, pages 30–43, Berlin, Heidelberg, 2006. Springer-Verlag.

[65] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, Sept. 2009.

[66] A. Poulovassilis, P. Selmer, and P. T. Wood. Flexible querying of Lifelong Learner Metadata. *IEEE Transactions on Learning Technologies*, 5(2):117–129, 2012.

[67] A. Poulovassilis and P. T. Wood. Combining approximation and relaxation in semantic web path queries. In *Proceedings of the 9th International Semantic Web Conference*, ISWC'10, pages 631–646, Berlin, Heidelberg, 2010. Springer-Verlag.

[68] B. R. K. Reddy and P. S. Kumar. Efficient approximate SPARQL querying of web of linked data. In *Uncertainty Reasoning for the Semantic Web*, volume 654 of *CEUR Workshop Proceedings*, pages 37–48. CEUR-WS.org, 2010.

[69] F. Reynolds. An RDF framework for resource discovery. In *Proceedings of the Second International Conference on Semantic Web - Volume 40*, SemWeb'01, pages 37–43, Aachen, Germany, 2001. CEUR-WS.org.

[70] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, Oct. 1980.

[71] M. Sassi, O. Tlili, and H. Ounelli. Approximate query processing for database flexible querying with aggregates. In A. Hameurlain, J. Küng, and R. Wagner, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems V*, pages 1–27. Springer-Verlag, Berlin, Heidelberg, 2012.

[72] M. Schmidt. *Foundations of SPARQL Query Optimization*. PhD thesis, Albert-Ludwigs-Universitat Freiburg, 2009.

[73] P. Selmer, A. Poulovassilis, and P. T. Wood. Implementing flexible operators for regular path queries. In *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015.*, pages 149–156, 2015.

[74] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of RDF/S query patterns. In *Proceedings of the 4th International Semantic Web Conference*, ISWC'05, pages 607–623, Berlin, Heidelberg, 2005. Springer-Verlag.

[75] v. Čebirić, F. Goasdoué, and I. Manolescu. Query-oriented summarization of RDF Graphs. *Proc. VLDB Endow.*, 8(12):2012–2015, Aug. 2015.

[76] Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan. Implementing an inference engine for RDFS/OWL constructs and user-defined rules in Oracle. In *24th IEEE International Conference on Data Engineering*, pages 1239–1248. IEEE, 2008.

Appendix A

Queries for Performance Study in Chapters 6 and 7

A.1 LUBM Queries

Q1:

```
SELECT ?x ?t WHERE{
?x publicationAuthor/teacherOf ?c .
?x publicationAuthor/teachingAssistantOf ?c .
?x rdf:type Article  . ?x title ?t
}
```

Suppose a user is looking for publications written by a teacher and a teaching assistant that teach the same course. By posing the above query, the user will not retrieve any answer. This is due to the fact that predicate *title* refers to the class *Person* and not to the class *Article*. Therefore, the user approximates the last triple pattern. The rewriting algorithm will replace, amongst other edit operations, the predicate *title* with the symbol (_). The resulting query will return multiple properties of the article (such as the publishing date, authors and publisher) including its title.

If the user is looking for other types of publication the authors wrote, she could relax the *(?x rdf:type Article)* triple. This will result in a step of relaxation that

replaces *Article* with a super-class *Publication* which includes also *UnofficialPublication*, *Specification*, *Book*, *Manual* and *Software*.

Q2:

```
SELECT ?c WHERE{
GraduateStudent1 mastersDegreeFrom/hasAlumnus Student25 .
GraduateStudent1  takesCourse ?c . Student25 takesCourse ?c
}
```

The user is looking for every course that was taken by both GraduateStudent1 (who has a Masters degree) and Student25 who is an alumnus of the university in which GraduateStudent1 has graduated. Due to the fact that GraduateStudent1 did not get a Masters degree from the university in which Student25 is an alumnus the query returns no answers. The user may choose to relax the first triple pattern. Therefore, the predicate *mastersDegreeFrom* will be replaced by *degreeFrom* which is a super-predicate of *undergraduateDegreeFrom*, *mastersDegreeFrom* and *doctoralDegreeFrom*. The new query will return many answers including those the user was looking for.

Q3:

```
SELECT ?x ?z WHERE{
?x doctoralDegreeFrom University1 . ?x worksFor University1 .
?x teacherOf ?c . ?z teachingAssistantOf ?c
}
```

The previous query returns every teacher who has a doctoral degree from University1 in which he works, along with his teaching assistant(s). Suppose the teacher the user is looking for is not returned by the previous query due to the fact that the teacher did not get a doctoral degree from University1. The user tries relaxing the first triple pattern to the following: $(?x, degreeFrom, University1)$. The new query will

return also teachers who received any kind of degree from the university in which they work for.

The original query might not return any answer also because teacher $?x$ is currently not working for *University1*. By relaxing also the second triple pattern, *(?x worksFor University1)* will be replaced by *(?x rdf:type Person)*. This new query will return every person who has a doctoral degree from *Universty1* along with the teaching assistants of his courses.

A third reason for which the query might not return any answer is that the teacher does not have a teaching assistant. Relaxing the first triple pattern yields no further answers. If however the user applies APPROX to the last triple pattern then the rewriting algorithm can replace the predicate *teachingAssistantOf* with the ϵ resulting in the triple pattern: $(?z, \epsilon, ?c)$. The resulting query will return, instead of pairs of teacher/student, pairs of teacher/course. In this particular instance it is interesting to note that the approximation operator behaves like the OPTIONAL operator.

Q4:

```
SELECT * WHERE{
?z publicationAuthor AssociateProfessor3.
?z publicationAuthor/advisor AssociateProfessor3
}
```

Suppose a user is looking for all the publications co-authored by AssociateProfessor3 and a student that she advises. The query might not return any answers due to the fact that AssociateProfessor3 may not have published any article with one of her advisees. Applying RELAX to the second triple pattern yields no further answers. Applying instead APPROX to the second triple pattern the rewriting algorithm can remove the predicate *advisor*. The new query will return every publication written by AssociateProfessor3.

If the user wants to decrease the number of answers by finding every publication written by AssociateProfessor3 and a student then, instead of approximating, the

user might relax the second triple pattern which may then be rewritten as follows: *?z publicationAuthor/rdf:type Student*. The rewriting algorithm applied RDFS entailment using the statement (*advisor rdfs:domain Student*) from the LUBM ontology.

Q5:

```
SELECT ?s ?c WHERE{
?x rdf:type AssistantProfessor . ?x teacherOf ?c .
?s takesCourse ?c . ?s rdf:type UndergraduateStudent .
?s address "UndergraduateStudent5@Department1.University0.edu"
}
```

This query returns every undergraduate student and course, such that the student has email "UndergraduateStudent5@Department1.University0.edu" and takes a course taught by an AssistantProfessor. However, the predicate *address* is not present in LUBM, therefore the query will not return any answer. By applying the APPROX operator to the fifth triple pattern, *address* will be replaced by *emailAddress*. Still the query does not return any answer since the student is a Graduate student and not an undergraduate student. By relaxing the fourth triple pattern, the user will retrieve also masters students or doctoral students with that email. Finally, the user will be able to retrieve the URI of the specified student with their courses.

Q6:

```
SELECT * WHERE{
Student1 advisor/teacherOf ?c . Student1 takesCourse ?c .
?c rdf:type UndergraduateCourse
}
```

Suppose the user is looking for every course that Student1 attends that is taught by the advisor of Student1. This query does not return any answer since LUBM does

not contain the class UndegraduateCourse. By relaxing the last triple pattern *?c rdf:type UndergraduateCourse*, the triple pattern becomes *?c rdf:type rdfs:Resource*. Since every URI is a Resource, the query will return every course attended by Student1 which is taught by her advisor.

Q7:

```
SELECT ?p WHERE{
ResearchGroup3 subOrganizationOf* ?x .
?p rdf:type AssistantProfessor . ?p worksFor ?x  .
Publication0 publicationAuthor ?p
}
```

The user is looking for every AssistantProfessor ?p who works for an organization that has ResearchGroup3 as a sub-organization such that ?p wrote the publication 'Publication0'. Consider the following two scenarios:

If the user specified the wrong research group then by relaxing the first triple pattern the triple can be replaced with *Organization rdfs:type*$^{-}$*/subOrganizationOf* ?x* using the statement *subOrganizationOf*, *rdfs:range*, *Organization*. The resulting query returns every author of the publication 'Publication0' that is also an assistant professor.

If the author the user is looking for is not an assistant professor then, by relaxing the second triple pattern, this will be rewritten to *?p rdf:type Professor*. The new query returns every author of the publication that is also a professor (this includes FullProfessor, AssistantProfessor and VisitingProfessor). A second step of relaxation will further rewrite the triple pattern to *?p rdf:type Person*.

A.2 DBpedia Queries

Q1

```
SELECT ?y WHERE{
```

```
<The_Hobbit> subsequentWork* ?y . ?y rdf:type  Book
}
```

The user is looking for every book from the Lord of the Rings, which follows the book "The Hobbit". In fact the query returns only the URI

```
<The_Lord_of_the_Rings>
```

The answer returned is correct although the user knows that "The Lord of the Rings" comprises of more books.

If the user applies RELAX to the first triple pattern then many uninteresting answers are returned. If instead the user applies APPROX to the first triple pattern, the rewriting algorithm will generate the following query:

```
SELECT ?x ?y WHERE{
<The_Hobbit> subsequentWork*/_ ?y . ?y rdf:type  Book
}
```

The new query returns every book connected to the URI of the "The Lord of the Rings" resource, including:

```
dbr:The_Return_of_the_King
dbr:The_Two_Towers
dbr:The_Fellowship_of_the_Ring
```

which are the answers the user is looking for.

Q2

```
SELECT ?x ?y WHERE{
?x albumBy <The_Rolling_Stones> . ?x rdf:type Album .
?y album ?x .?x recordLabel <London_Records>
}
```

The user is looking for all the albums published by the Rolling Stones with the London Records record label along with the songs in the album. The previous query does not return any answer since the predicate *albumBy* does not exist in DBpedia.

By approximating the first triple pattern the rewriting algorithm will replace the *albumBy* predicate with the symbol (_) which includes the predicate *artist*. The new query will return some of the albums from the London Records record label published by the Rolling Stones. However, some of the answers the user was expecting are not retrieved due to the fact that for some albums the *recordLabel* predicate is missing. By relaxing the last triple pattern this will be rewritten as:

```
?x rdf:type <Resource>
```

since DBpedia has the statement *recordLabel rdfs:domain Resource*. The new query will return every album written by the Rollings Stones, together with their songs, including the answers that were not present in the previous query.

Q3

```
SELECT ?k ?d ?kd WHERE{
?k  diedIn <Battle_of_Poitiers> .
<Battle_of_Poitiers> date ?d . ?k deathDate  ?kd
}
```

Suppose a user is looking for anyone who died during the "Battle of Poitiers" and the dates of both the battle and the person's death. The predicate *diedIn* is not present in DBpedia therefore the user will not retrieve any answers. By approximating the first triple pattern the predicate *diedIn* will be replaced by the symbol (_). The new query will return every person connected to the "Battle of Poitiers" (this occurs since the domain of *deathDate* has to be a person). The symbol (_) includes the predicate *deathPlace* which connects the resource "Battle of Poiters" to "Peter I, Duke of Bourbon" which was the answer the user was looking for.

Q4

```
SELECT ?x ?kd WHERE{
?x subject Duelling_Fatalities . ?x deathDate "18xx-xx-xx" .
?x rdf:type Scientist
}
```

A user is looking for every Scientist who died in the 1800 during a duel. The above query does not return any answer since the year "18xx-xx-xx" is not formatted correctly. The user therefore relaxes the second triple pattern which will be replaced with *?x rdf:type Person*. The query will then return the resource "Évariste Galois" amongst other answers.

However, the user was expecting additional answers. The user decides to retrieve other types of persons who died during a duel. Therefore, she relaxes also the last triple pattern which will be rewritten as *?x rdf:type owl:Thing*. This query will return every person who died during a duel including the scientist "Martin Lichtenstein".

Q5

```
SELECT ?x ?f  WHERE{
12_Angry_Men_(1957_film) actor ?a . ?x parent ?a . ?f actor ?x.
?x birthPlace New_York
}
```

Suppose a user is looking for every child born in New York to actors who played in the film "12 Angry Men". Moreover, the user is interested in the films in which these actors have played. The predicate *actor* is not present in the DBpedia dataset, so the above query returns no answers. The user, therefore, applies the APPROX operator to the first and third triple pattern.

The rewriting algorithm will replace the predicate *actor* with the predicate *starring*. The resource "New York" refers to the state, not the city, and for this reason

the query will return fewer answers than expected, since not every person in DBpedia is connected to their state with *birthPlace*. By relaxing also the last triple pattern, this will be rewritten as *(?f, rdf:type, Person)*. The resulting query will return every person who had a parent that played in "12 Angry Men".

A.3 YAGO Queries

Q1

```
SELECT * WHERE{
<Battle_of_Waterloo> happenedIn/(hasLongitude|hasLatitude) ?x
}
```

Suppose the user wants to find the geographic coordinates of the "Battle of Waterloo" event by posing the above query. This query returns no answer since YAGO does not store the geographic coordinates of Waterloo.

If we insert the predicate "isLocatedIn" after "happenedIn" then the query would have returned 16 answers including the geographic coordinates of some regions in Belgium. If instead the user approximates the triple pattern of the query then the rewriting algorithm will apply the edit operation delete to *happenedIn* from the property path. The resulting query will return the coordinates of the "Battle of Waterloo" event.

Q2

```
SELECT * WHERE{
?x actedIn <Tea_with_Mussolini> . ?x hasFamilyName ?z
}
```

Suppose the user is looking for the family names of actors who played in the film "Tea with Mussolini" by posing the above query. The query returns only a partial set of answers due to the fact that in the YAGO dataset some actors have only a first name (e.g. Cher), and others have their full name recorded using the predicate

rdfs:label. The user may relax the second triple pattern in order to retrieve more answers. This causes the relaxed query to return the given names of actors in the film, recorded using the property "hasGivenName" since "hasGivenName" is a sub-property of "label" (hence returning Cher), as well as actors' full names recorded using the property "label": a total of 255 results.

Q3

```
SELECT * WHERE{
?x rdf:type Event . ?x happenedOnDate ‘‘1643-##-##’’ .
?x happenedIn ‘‘Berkshire’’
}
```

Suppose the user wishes to find events taking place in Berkshire in 1643 and poses on the YAGO dataset the above query. The user will retrieve no answer since the predicate *happenedIn* does not connect directly to the literal "Berkshire". Therefore, the user may relax the third triple pattern. The query will be rewritten as follows:

```
SELECT * WHERE{
?x rdf:type Event . ?x happenedOnDate "1643-##-##" .
?x rdf:type Event
}
```

This query will return every event that occurred in the year 1643 including those in Berkshire.

The user instead of applying the RELAX operator may use the APPROX operator. Among other queries, the rewriting algorithm will generate the following:

```
SELECT * WHERE{
?x rdf:type Event . ?x happenedOnDate "1643-##-##" .
?x happenedIn/_ ‘‘Berkshire’’
}
```

which will return all the events that occurred in 1643 in Berkshire.

Appendix B

Tables of Results for Performance Study in Chapters 6 and 7

B.1 LUBM Results

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
0	D1	0.001	0.001	0.003	0.001	0.001	0.002	0.001
1	D1	0.002	0.002	0.202	0.02	2.2	0.001	0.002
2	D1	0.929	0.002	102.3	0.123	13.3	0.001	0.002
3	D1	2.568	0.003	300.46	0.323	32.3	0.001	0.07
0	D2	0.001	0.719	0.004	0.6	0.03	0.05	0.2
1	D2	0.002	0.983	0.87	1.023	23.2	0.05	0.003
2	D2	3.41	1.01	N/A	N/A	N/A	0.05	0.003
3	D2	9.98	1.02	N/A	N/A	N/A	0.05	0.29
0	D3	0.015	0.9	0.6	1.4	1.6	1.3	0.4
1	D3	0.23	1.3	N/A	N/A	N/A	0.08	0.03
2	D3	12.15	1.4	N/A	N/A	N/A	0.08	0.03
3	D3	33.132	1.8	N/A	N/A	N/A	0.08	2.53

Table B.1: Execution time in seconds of each query, for each maximum cost and each dataset.

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	D1	1.279	0.883	0.402	1.52	3.2	0.57	1.52
2	D1	1.463	3.01	87.3	2.3	11.3	0.57	2.8
3	D1	4.56	4.823	230.53	1.653	25.8	0.57	3.83
1	D2	1.43	1.42	1.72	2.023	13.5	0.75	1.56
2	D2	3.356	4.38	1241.94	735.1	230.24	0.75	2.953
3	D2	6.91	5.023	3457.33	1974.32	896.26	0.75	3.98
1	D3	1.62	1.53	N/A	N/A	N/A	1.46	1.63
2	D3	8.49	4.58	N/A	N/A	N/A	1.46	3.53
3	D3	23.98	5.36	N/A	N/A	N/A	1.46	4.11

Table B.2: Execution time of each query, for each maximum cost and each dataset, with the pre-computation optimisation

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	D1	0.002	0.61	1.09	1.02	1.7	0.89	1.303
2	D1	0.691	1.09	20.3	1.19	2.1	0.89	1.43
3	D1	1.439	1.3	24.6	1.26	2.7	0.89	1.6
1	D2	0.002	0.983	1.932	1.3	7.4	1.05	1.61
2	D2	1.23	1.21	43.4	1.32	36.92	1.05	1.7
3	D2	3.03	1.52	75.3	1.34	75.38	1.05	3.7
1	D3	0.002	1.07	22.4	3.51	42.35	2.67	2.13
2	D3	7.392	1.23	450.34	3.76	193.53	2.67	2.3
3	D3	18.41	2.3	590.87	3.94	347.013	2.67	5.09

Table B.3: Execution time of each query, for each maximum cost and each dataset, using the summarisation of size 2.

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	D1	0.002	1.23	1.28	1.32	1.1	1.4	1.24
2	D1	0.519	1.45	18.3	1.34	1.75	1.4	1.9
3	D1	1.271	2.32	22.3	1.43	2.4	1.4	2.6
1	D2	0.002	1.36	1.3	1.42	2.23	2.31	1.82
2	D2	1.15	2.35	20.3	1.6	29.85	2.31	2.45
3	D2	2.86	2.4	25.5	1.5	63.45	2.31	4.19
1	D3	0.002	1.48	2.6	2.28	38.0	2.5	2.34
2	D3	6.539	3.19	300.2	2.34	164.23	2.5	3.32
3	D3	16.73	3.3	386.54	2.58	298.87	2.5	4.2

Table B.4: Execution time of each query, for each maximum cost and each dataset, using the summarisation of size 3.

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	D1	1.12	1.03	1.35	1.01	1.23	1.42	1.34
2	D1	1.742	1.49	19.8	1.4	1.29	1.42	2.29
3	D1	1.812	1.92	24.94	1.83	2.38	1.42	2.96
1	D2	1.14	1.13	2.23	1.49	2.11	1.95	1.45
2	D2	1.814	2.01	22.5	1.62	29.35	1.95	2.65
3	D2	2.925	2.85	39.7	1.98	57.25	1.95	4.29
1	D3	1.33	1.4	2.95	3.42	39.54	2.15	1.54
2	D3	5.29	2.75	312.42	3.44	123.87	2.15	2.85
3	D3	10.42	2.91	387.94	3.97	262.18	2.15	4.3

Table B.5: Execution time of each query, for each maximum cost and each dataset, using the summarisation of size 3 and the pre-computation optimisation.

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	D1	0.05	0.024	0.98	0.23	1.3	0.023	0.046
2	D1	1.02	0.06	82.3	0.43	1.9	0.023	0.057
3	D1	2.194	0.067	270.67	0.678	30.2	0.023	0.096
1	D2	0.06	1.03	1.24	1.95	20.1	0.064	0.065
2	D2	2.28	1.31	N/A	N/A	N/A	0.064	0.068
3	D2	8.34	1.46	N/A	N/A	N/A	0.064	0.64
1	D3	0.09	1.4	N/A	N/A	N/A	0.136	0.071
2	D3	8.86	1.76	N/A	N/A	N/A	0.136	0.074
3	D3	27.13	1.92	N/A	N/A	N/A	0.136	2.68

Table B.6: Execution time of each query, for each maximum cost and each dataset, with query containment optimisation.

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	D1	1.51	0.991	0.427	1.533	3.26	0.75	0.233
2	D1	1.72	1.7	73.1	1.89	9.82	0.75	2.353
3	D1	2.654	2.6	209.82	2.87	22.4	0.75	2.94
1	D2	1.66	1.55	1.69	2.258	12.8	1.48	1.46
2	D2	2.46	3.5	1020.84	625.1	190.47	1.48	3.23
3	D2	7.49	4.823	3042	1421	832.98	1.48	4.12
1	D3	1.89	1.62	N/A	N/A	N/A	1.68	1.93
2	D3	6.49	4.18	N/A	N/A	N/A	1.68	3.93
3	D3	25.95	6.123	N/A	N/A	N/A	1.68	4.3

Table B.7: Execution time of each query, for each maximum cost and each dataset, with query containment and pre-computation optimisation.

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	D1	0.45	1.41	1.23	1.33	1.4	1.78	2.24
2	D1	1.03	2.3	15.4	1.62	2.24	1.78	2.4
3	D1	1.86	2.81	18.2	1.83	2.7	1.78	2.97
1	D2	0.67	1.6	1.98	1.97	2.43	1.98	2.28
2	D2	1.39	2.47	18.3	1.82	24.26	1.98	2.83
3	D2	2.09	2.85	24.1	2.1	52.45	1.98	4.47
1	D3	0.78	1.67	2.78	2.48	37.3	2.58	1.36
2	D3	5.14	2.95	289.5	2.76	136.32	2.58	3.24
3	D3	12.92	3.37	336.42	2.94	158.38	2.58	4.98

Table B.8: Execution time of each query, for each maximum cost and each dataset, using the summarisation of size 3 and query containment.

Max Cost	Dataset	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
1	D1	1.23	1.48	1.26	1.75	2.27	1.28	2.18
2	D1	1.83	2.07	14.9	2.19	2.59	1.28	2.98
3	D1	1.98	2.91	17.8	2.28	2.98	1.28	3.3
1	D2	1.26	1.92	1.58	1.89	2.96	2.47	2.79
2	D2	1.86	2.17	17.9	2.36	22.94	2.47	3.34
3	D2	2.13	2.97	21.34	2.37	46.45	2.47	3.51
1	D3	1.59	1.97	3.78	1.96	32.72	2.912	3.59
2	D3	4.32	2.32	265.2	2.56	131.27	2.912	3.64
3	D3	8.36	3.22	319.12	2.82	152.84	2.912	3.82

Table B.9: Execution time of each query, for each maximum cost and each dataset, using the pre-computation optimisation, the summarisation of size 3 and query containment.

B.2 DBpedia Results

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
0	0.002	0.004	0.002	0.003	0.005
1	0.003	0.072	864	0.02	645
2	212	0.433	1739	0.06	N/A
3	6395	7.44	10090	0.07	N/A

Table B.10: Execution time in seconds of each query, for each maximum cost.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	0.013	0.052	552	0.23	539
2	192	0.427	1437	0.31	N/A
3	4357	6.39	9128	0.78	N/A

Table B.11: Execution time of each query, for each maximum cost with, the precomputation optimisation.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	0.02	0.03	0.06	0.07	0.01
2	N/A	0.433	1.42	N/A	1.6
3	N/A	0.49	2.93	N/A	12.5

Table B.12: Execution time of each query, for each maximum cost, using the summarisation of size 2.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	4.32	2.43	1.69	0.07	0.6
2	N/A	3.98	4.25	N/A	3.22
3	N/A	10.62	21.34	N/A	7.2

Table B.13: Execution time of each query, for each maximum cost, using the summarisation of size 2 and pre-computing optimisation.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	0.564	0.93	427	1.8	439
2	135	1.398	1318	2.6	1453
3	4256	3.27	5261	2.77	10461

Table B.14: Execution time of each query, for each maximum cost, with query containment.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	1.78	2.33	214	2.88	439
2	94.5	3.56	523	2.97	1263
3	2391	3.82	562	3.24	8547

Table B.15: Execution time of each query, for each maximum cost, with query containment and pre-computation optimisation.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	0.245	0.34	0.33	0.45	1.43
2	N/A	1.45	2.56	N/A	3.43
3	N/A	2.32	5.63	N/A	26.22

Table B.16: Execution time of each query, for each maximum cost, with query containment and summarisation size 2.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	2.412	2.56	2.43	1.15	2.14
2	N/A	2.88	3.14	N/A	2.35
3	N/A	3.37	5.73	N/A	22.03

Table B.17: Execution time of each query, for each maximum cost, with query containment and summarisation size 2 and pre-computation optimisation.

B.3 YAGO Results

Max Cost	Q_1	Q_2	Q_3
0	0.005	0.002	0.001
1	0.037	0.007	0.23
2	504	0.007	2.1
3	N/A	0.007	4.7

Table B.18: Execution time in seconds of each query, for each maximum cost.

Max Cost	Q_1	Q_2	Q_3
1	0.059	0.06	0.56
2	482.6	0.06	2.51
3	N/A	0.06	4.93

Table B.19: Execution time of each query, for each maximum cost, with pre-computation optimisation.

Max Cost	Q_1	Q_2	Q_3
1	0.012	0.015	0.09
2	224	0.015	1.89
3	N/A	0.015	1.92

Table B.20: Execution time of each query, for each maximum cost, with summarisation of size 2.

Max Cost	Q_1	Q_2	Q_3
1	0.013	0.017	0.13
2	214	0.017	1.93
3	N/A	0.017	2.4

Table B.21: Execution time of each query, for each maximum cost, with summarisation of size 3.

Max Cost	Q_1	Q_2	Q_3
1	0.023	0.031	0.27
2	201	0.031	2.18
3	N/A	0.031	2.72

Table B.22: Execution time of each query, for each maximum cost, with summarisation of size 3 and pre-computation optimisation.

Max Cost	Q_1	Q_2	Q_3
1	0.17	0.11	0.31
2	434	0.11	2.31
3	N/A	0.11	3.18

Table B.23: Execution time of each query, for each maximum cost, with query containment.

Max Cost	Q_1	Q_2	Q_3
1	0.61	0.15	0.59
2	402.1	0.15	2.34
3	N/A	0.15	4.31

Table B.24: Execution time of each query, for each maximum cost, with pre-computation optimisation and query containment.

Max Cost	Q_1	Q_2	Q_3
1	0.63	0.43	0.57
2	173	0.43	2.28
3	N/A	0.43	2.72

Table B.25: Execution time of each query, for each maximum cost, with summarisation of size 3 and query containment.

Max Cost	Q_1	Q_2	Q_3
1	0.54	0.65	0.74
2	152	0.65	2.12
3	N/A	0.65	2.39

Table B.26: Execution time of each query, for each maximum cost, with precomputation optimisation, summarisation of size 3 and query containment.