Flexible Querying of Graph-Structured Data

Petra Selmer

July 2017

A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science and Information Systems
Birkbeck, University of London

Declaration

This thesis is the result of my own work, except where explicitly acknowledged in the text.

Petra Selmer

July 23, 2017

Abstract

Given the heterogeneity of complex graph data on the web, such as RDF linked data, it is likely that a user wishing to query such data will lack full knowledge of the structure of the data and of its irregularities. Hence, providing flexible querying capabilities that assist users in formulating their information-seeking requirements is highly desirable.

The query language we adopt in this thesis comprises conjunctions of regular path queries, thus encompassing recent extensions to SPARQL to allow for querying paths in graphs using regular expressions (SPARQL 1.1). To this language we add three operators: APPROX, supporting standard notions of query approximation based on edit distance; RELAX, which performs query relaxation based on RDFS inference rules; and FLEX, which simultaneously applies both approximation and relaxation to a query conjunct, providing even greater flexibility for users.

We undertake a detailed theoretical investigation of query approximation, query relaxation, and their combination. We show how these notions can be integrated into a single theoretical framework and we provide incremental evaluation algorithms that run in polynomial time in the size of the query and the data — provided the queries are acyclic and contain a fixed number of head variables — returning answers in ranked order of their 'distance' from the original query.

We motivate and describe the development of a prototype system, called Approx-Relax, that provides users with a graphical facility for incrementally constructing

their queries and that supports both query approximation and query relaxation. Restricting our focus to the user-facing features of ApproxRelax, we present a qualitative case study showing how ApproxRelax overcomes problems in a previous system in the same application domain of lifelong learning.

We then describe our techniques for implementing the extended language in the Omega system, our final implementation of query approximation and query relaxation, in which we discuss the system architecture, the data structures, data model, API and query evaluation algorithms. Subsequently, we present a performance study undertaken on two real-world datasets. Our baseline implementation performs competitively with other automaton-based approaches, and we demonstrate empirically how various optimisations can decrease execution times of queries containing APPROX and RELAX, sometimes by orders of magnitude.

Publications

1. Alexandra Poulovassilis, Petra Selmer and Peter T. Wood. **Flexible Querying of Lifelong Learner Metadata**. *IEEE Transactions on Learning Technologies* (2012), Volume 5 Issue No. 2, pp. 117 – 129

   Chapter 5

2. Petra Selmer, Alexandra Poulovassilis and Peter T. Wood. **Implementing Flexible Operators for Regular Path Queries**. In Proceedings of the *18th Joint International Conferences on Extending Database Technology/Database Theory Workshops* (EDBT/ICDT 2015), pp. 149–156

   Chapters 6 and 7

3. Alexandra Poulovassilis, Petra Selmer and Peter T. Wood. **Approximation and Relaxation of Semantic Web Path Queries**. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web* (2016), Volume 40, pp. 1 – 21

   Chapters 3, 4 and 8

Acknowledgements

I should like especially to thank my two wonderful supervisors, Professor Alexandra Poulovassilis and Professor Peter Wood, for their unfailing patience, wisdom, and guidance over the past few years. Your counsel and support have been invaluable, and I have learnt so much from you.

My sincere thanks go to Neo Technology, and my friends and colleagues there, who have over the past two years gone out of their way to support me in this endeavour.

I am grateful to Sparsity Technologies for their provision of a research licence for Sparksee for the duration of my PhD.

I owe a huge debt of thanks to my parents for instilling in me a love of learning, and, in particular, a passion for science.

Finally, I should like to thank my husband, André, without whose support, encouragement, and unending cups of tea, this thesis would never have been completed.

Contents

Abstract 3

Publications 5

Acknowledgements 6

Contents 7

List of Algorithms 11

List of Figures 12

List of Tables 14

1 Introduction 15

1.1 Background and motivation 15

1.2 Thesis contributions 21

1.3 Thesis outline 23

2 Literature Review 25

2.1 Graph-modelled data and graph query languages 25

2.1.1 Graph models 26

2.1.2 Graph query languages 28

2.1.3 SPARQL . . . 30
2.1.4 Linked and distributed data . . . 31
2.2 Keyword-based querying . . . 32
2.3 Query relaxation . . . 34
2.4 Query approximation . . . 38
2.5 Subgraph matching . . . 40
2.6 Discussion . . . 42

3 Theoretical Preliminaries 44
3.1 The data model . . . 45
3.2 The query language . . . 47
3.3 Single-conjunct queries . . . 49
3.4 Query approximation . . . 52
3.4.1 Approximate matching of single-conjunct queries . . . 52
3.4.2 Incremental evaluation of APPROX conjuncts . . . 60
3.5 Query relaxation . . . 62
3.5.1 Ontology-based relaxation of single-conjunct queries . . . 62
3.5.2 Computing the relaxed answer . . . 68
3.5.3 Incremental evaluation of RELAX conjuncts . . . 71
3.6 Multi-conjunct queries . . . 71
3.7 Summary . . . 72

4 Correctness and Complexity Results 74
4.1 Approximation of single-conjunct queries . . . 75
4.2 Incremental evaluation . . . 82
4.3 Relaxation of single-conjunct queries . . . 85
4.4 Concluding remarks . . . 92

5 The *ApproxRelax* System and a Case Study 93
5.1 Case study: Lifelong Learning . . . 94
5.2 The *ApproxRelax* system . . . 97
5.3 Comparison with *L4All*'s "What Next" . . . 108
5.4 Concluding remarks . . . 111

6 The *Omega* System **112**
6.1 System architecture . . . 113
6.2 The C5 Generic Collection library . . . 115
6.3 The Sparksee Data Model and API . . . 116
6.4 Creating data graphs in Omega . . . 117
6.5 Conjunct initialisation . . . 118
6.5.1 Construction of the automaton . . . 118
6.5.2 Initialisation . . . 119
6.6 Query conjunct evaluation . . . 123
6.6.1 The `GetNext` function . . . 124
6.6.2 The `NextStates` function . . . 125
6.6.3 The `Succ` function . . . 127
6.7 Other implementations . . . 128
6.8 Concluding remarks . . . 130

7 Query Performance Analysis **131**
7.1 The L4All evaluation . . . 132
7.1.1 Data . . . 134
7.1.2 Queries . . . 134
7.1.3 Baseline experimental results . . . 136
7.1.4 Analysis . . . 138
7.2 The YAGO evaluation . . . 142
7.2.1 Data . . . 142
7.2.2 Queries . . . 143
7.2.3 Baseline experimental results . . . 143
7.2.4 Analysis . . . 146
7.3 Performance comparison . . . 147
7.4 Approximated Regular Path Query Optimisation . . . 149
7.4.1 Path indexes . . . 149
7.4.2 Approach and methodology . . . 150
7.4.3 Experimental results . . . 152
7.4.4 Further work . . . 154
7.5 Concluding remarks . . . 154

8 The FLEX Operator **156**

8.1 Evaluation of single-conjunct FLEX queries 157

8.2 Multi-conjunct FLEX queries and comparison with APPROX/RELAX166

8.3 Concluding remarks 168

9 Conclusions and future work **169**

9.1 Thesis summary 169

9.2 Contributions 171

9.3 Directions for future work 172

Bibliography **175**

List of Algorithms

- Function Succ(s, n, A_Q, G) 60
- Function GetNext(X, R, Y, A_Q, G) 62
- Procedure Open 121
- Function GetNext(X, R, Y) 126
- Function NextStates(s) 127
- Function Succ(s, n) 128

List of Figures

1.1 A graph of a company, consisting of employees, departments they work in, and projects they worked on. 18

3.1 Example data graph G showing flight insurance data. 46
3.2 Example ontology K from the flight insurance domain. 47
3.3 Query automaton M_Q for conjunct ('FL56', $fn_1 \cdot pn_1^-, X$). 57
3.4 Fragment of approximate automaton A_Q for conjunct ('FL56', $fn_1 \cdot pn_1^-, X$). 57
3.5 A subautomaton of the product automaton H of A_Q and G. 59
3.6 RDFS Inference Rules. 63
3.7 Closure of graph G in Figure 3.1 with respect to the ontology K in Figure 3.2. 64
3.8 Additional rules used to compute the extended reduction of an RDFS ontology. 65
3.9 Relaxed automaton M_Q^K for conjunct ($Y, pn_1^-.\texttt{type}, P_1$). 70
3.10 A subautomaton of the product automaton H. 71

4.1 Automaton for the deletion of the label b in T_{p_k} (b is not the last label). 78

5.1 A fragment of Dan's timeline data and metadata. 98
5.2 A fragment of Liz's timeline data and metadata. 99

5.3 A fragment of Al's timeline data and metadata. 100
5.4 *ApproxRelax* query set-up. 102
5.5 Constructing an Educational episode query template. 103
5.6 Constructing an Occupational episode query template. 105
5.7 Viewing episode query templates. 106
5.8 Viewing the query results. 107

6.1 System architecture. 113

7.1 The L4All data graph sizes (using the closure of the data graph). . . . 135
7.2 Execution time (ms) – exact L4All queries. 138
7.3 Execution time (ms) – APPROX L4All queries. 141
7.4 Execution time (ms) – RELAX L4All queries. 141
7.5 Execution times (ms), YAGO data graph. 146

8.1 Automaton for conjunct $(\text{'FL56'}, fn_1.ppn_1.pn_1^-, Y)$. 159
8.2 Automaton for conjunct $(Y, n_1.\texttt{type}, N_1)$. 159
8.3 Automaton for $(X, e.\texttt{type}, \text{'c'})$. 161

List of Tables

5.1 Evaluation of the query 109

7.1 Characteristics of the class hierarchies in the L4All data graphs. . . . 133
7.2 Properties in the L4All data graphs other than '`type`'. 133
7.3 Characteristics of the L4All data graphs. 135
7.4 The L4All query set: Q1 - Q3 and Q8 - Q12. 137
7.5 Results for each query and L4All data graph. 139
7.6 Total initialisation and execution times (ms) for each query run over the L4All data graphs (initialisation time in italics). 140
7.7 The YAGO query set: Q1 - Q6 and Q9. 144
7.8 The number of results per query for the YAGO data graph. 145
7.9 Total initialisation and execution times (ms) for each query run over the YAGO data graph (initialisation time in italics). 145
7.10 Results of using the statistics to simulate the running of Q_1 and Q_2 as approximated queries. 153

CHAPTER 1

Introduction

1.1 Background and motivation

Graph-structured data is becoming increasingly prevalent and gaining in importance. For example, the volume of graph-structured data on the web continues to grow, most recently in the form of RDF Linked Data[1] [10]. At the time of writing, there were 570 publicly-accessible large linked datasets, spanning a variety of domains, such as the life sciences, geographical and government domains[2].

Over the past few years, graph databases [84, 119] such as Neo4j[3], Sparksee[4] and OrientDB[5] have become more pervasive in both academia and industry. They have been used in areas as diverse as social network analysis [87]; fraud detection [103]; bioinformatics [85, 86, 106]; recommendations, geospatial data, master data management, network and data centre management, authorisation and access control [119]; and, most recently, in the Panama Papers investigation[6].

Storing graph-structured data in graph databases confers several advantages [110],

[1] `http://linkeddata.org/`
[2] `http://lod-cloud.net`
[3] `http://neo4j.com/`
[4] `http://sparsity-technologies.com`
[5] `http://orientdb.com/`
[6] `http://neo4j.com/blog/icij-neo4j-unravel-panama-papers/`

including: direct and intuitive support for graph data modelling; the provision of powerful and expressive graph query languages, examples of which are Cypher[7], Gremlin[8] and SPARQL [122]; native graph storage and indexing for fast traversals, such as in Neo4j, Sparksee, GRAIL [139], SCARAB [70] and SAINT-DB [109]; and inbuilt support for core graph algorithms (for example, subgraph matching) and graph programming APIs [119].

Another factor giving the management of graph-structured data more impetus is the ever-increasing need to integrate heterogeneous data from differing data sources in order to allow different institutions and organisations to collaborate more effectively as a result of having access to shared data. For example, the authors of [83] describe the Austrian Grid project which enables Austrian research institutions to share and access graph-structured data. High-volume RDF and graph-based data repositories, such as DBPedia [11], YAGO [75, 134], Bio4j[9] and others, have been established, thanks largely to the advances in automatic data extraction from hitherto disparate data sources [33]. Indeed, the success of this endeavour is exemplified by the development of YAGO2 [61] which, drawing on data derived from Wikipedia, GeoNames[10], and WordNet [38], extends the original YAGO system with spatial and temporal extensions.

Graph-structured data comprises *nodes*, representing entities of interest, and *edges*, denoting the relationships between the entities; the relationships may be as important as the entities themselves. Graph data models allow for flexible and intuitive modelling of complex data, and, owing to the fact that they do not require a schema, they are well-suited for use with heterogeneous data. In the graph data model we consider in this thesis, an edge connects two nodes to each other and is labelled with the type of relationship between the entities it connects. For example, using data from a company domain, entities (nodes) may represent employees, departments and projects. The relationships (edges) between these entities could then comprise *isManagerOf* (denoting one person managing another), *worksIn* (indicating that an employee works in a particular department), *workedOn* (indicating that a person worked on a particular project), and *partOf* (denoting that one project

[7] `http://neo4j.com/docs/developer-manual/current/cypher/`
[8] `https://github.com/tinkerpop/gremlin/wiki/`
[9] `http://bio4j.com/`
[10] `http://geonames.org`

was part of a larger project) and so on. This graph data model allows for the modelling — and, subsequently, querying — of complex, changing data, as it easily accommodates the addition of more entities and relationships as the data domain becomes richer and more complex over time. Using our example of the company domain, entities such as 'network' can be added, along with new relationship types, such as which department uses which network. Additionally, the older data may be enriched over time by storing new types of relationships of interest between any two employees, such as the fact that one person mentored another, and whether two employees ought not to work in the same department.

Based on our company domain, Figure 1.1 shows an example of a graph G, containing nodes denoting the names of employees, departments and projects, and edges, which denote relationships between the nodes. The latter include the *worksIn* relationship between employees and departments, the *isManagerOf* relationship between two employees, the *workedOn* and *managed* relationships between employees and projects (the former relationship indicates that an employee worked on a project in some capacity, whereas the latter indicates that an employee had a managerial role within the project), and the *partOf* relationship between projects, indicating that one project is part of another, larger project. A sequence of edges — called a *path* — between two nodes denotes an indirect relationship, such as the one between 'Jane' and 'Chronos', representing the fact that 'Jane' *workedOn* the 'Zephyr' project which is *partOf* the 'Chronos' project.

Our research in this thesis focuses on flexible querying of *conjunctive regular path queries* over graph data. We assume that the graphs are directed, and an edge labelled e from a node labelled a to a node labelled b is represented by a triple (a, e, b). Referring to the data graph G in Figure 1.1, the following conjunctive query, Q_1, consisting of two conjuncts, requests all employees who have *worked on* 'Chronos' and *work in* the 'Finance' department:

$$ans(x) \;\leftarrow\; (x, workedOn, \text{`Chronos'}), (x, worksIn, \text{`Finance'})$$

In this query, x is interpreted as a node variable to which the answers — a set of nodes corresponding to the relevant employees — will be bound, while $workedOn$, $worksIn$, 'Chronos' and 'Finance' are constants. A single answer, 'Paul', is returned.

A *regular path* query (RPQ) is a query in which pairs (x, y) of nodes may be found

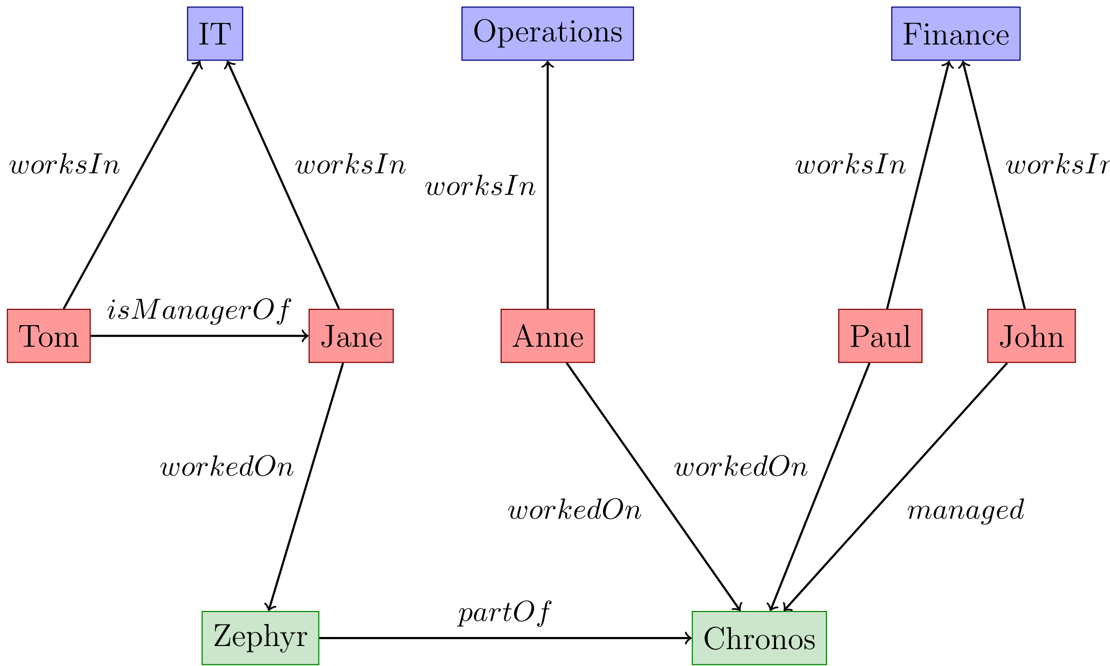

Figure 1.1: A graph of a company, consisting of employees, departments they work in, and projects they worked on.

such that there is a path from x to y whose sequence of labels matches some pattern, specified using a regular expression defined over the alphabet of edge labels [100].

Once again referring to Figure 1.1, the following RPQ, Q_2, asks for (x, y) pairs, where x is an employee and y is a project, such that x is either the *managerOf* an employee who has *workedOn* a project or has *workedOn* the project directly (the '?' operator denotes zero or one occurrence of the preceding label, and the '.' operator denotes a sequence of labels):

$$ans(x, y) \quad \leftarrow \quad (x, (isManagerOf? \cdot workedOn), y)$$

Q_2 will return the answers (Tom, Zephyr), (Jane, Zephyr), (Anne, Chronos) and (Paul, Chronos).

RPQs can be combined to form conjunctive RPQs (CRPQs), as exemplified by the following query, Q_3:

$$ans(x, y) \quad \leftarrow \quad (x, worksIn, \text{'Finance'}), (x, (isManagerOf? \cdot workedOn), y)$$

Q_3 will return the answer (Paul, Chronos).

Owing to the complexity, heterogeneity and evolution of both the data and the structure of data graphs in graph-modelled application domains and in integrated data sources such as RDF linked data, it is becoming increasingly unrealistic to expect users to be familiar with the data and its structure, and, therefore, to formulate queries that precisely match these. Indeed, a user querying such a data graph may find themselves in a position in which they are not able to obtain meaningful answers (or indeed, any answers) from the data. Moreover, the user may wish to begin from a set of initial query answers and thence proceed to further *explore* the data. Yet another scenario which may arise is that of posing queries that aim to discover how two (or more) data items are linked; for example, in the bioinformatics domain, querying the data graph in order to discover the relationships between various proteins.

Thus, the requirement that data graphs can be queried in a flexible fashion is becoming more compelling [31, 116, 140]. This means that answers that exactly match a user's query are no longer enough: it is also desirable to return answers which are in some sense *similar* to the exact answers.

There is also increasing awareness that keyword-based search alone often does not provide enough semantics or contextual structure within a query to return useful answers. For example, this has led to the recent inception of the Google Knowledge Graph [123]. The Google Knowledge Graph applies semantic search information garnered from Google's knowledge base to enhance the search results returned by Google's search engine, thus providing the user with more useful information. Two other systems seeking to combine keyword-based searching with semantic and structural information to enhance the relevance of results are QUICK [116] and XSEarch [20]; we discuss these in more detail in Chapter 2.

Moreover, the increasing recognition of the benefits and usefulness of *ontologies* for specific data domains is further driving the need for more sophisticated means of data querying that automatically take the semantics of the data into account, for example, query *relaxation* and *approximation*, as well as *ranking* of query results [15, 102, 125].

Flexible query answering through query relaxation and approximation techniques allows users both to explore the data meaningfully without necessarily being familiar

with the data and its underlying structure and also possibly to discover hitherto unknown relationships between entities; it has the potential to be beneficial in, among others, the scientific, medical and educational domains where there are large volumes of complex, heterogeneous data.

Query approximation when applied to RPQs extends the idea of regular expression matching by including a set of edit operations that may be applied to the labels making up the language of the regular expression; these include the insertion, deletion, and substitution of labels [115]. Each edit operation has an associated cost which can be configured by the user. Each application of an edit operation to a sequence of labels in the language of the regular expression within the query conjunct increases the 'distance' from the original sequence by the cost configured for the edit operation (we refer to this distance as the 'edit distance'). Answers obtained at greater distances are ranked lower than ones exactly matching the query conjunct, as well as those obtained at a smaller distance. The approximated form of a query conjunct may return quite different answers compared to the exact query conjunct.

For example, referring to Figure 1.1 and the example queries from earlier in this section, an approximated form of the query conjunct $(x, (isManagerOf? \cdot workedOn), y)$ is expressed as $APPROX(x, (isManagerOf? \cdot workedOn), y)$. This would first return all (x, y) pairs whose sequence of edge labels conforms to the regular expression $(isManagerOf? \cdot workedOn)$ at distance 0; i.e. these are the *exact* answers, which include (Tom, Zephyr), (Jane, Zephyr), (Anne, Chronos) and (Paul, Chronos). Substituting the edge label *workedOn* by, respectively, the edge label *managed*, *worksIn* or *partOf* would return all (x, y) pairs whose sequences of edge labels conform to one of the regular expressions $(isManagerOf? \cdot managed)$, $(isManagerOf? \cdot worksIn)$ or $(isManagerOf?, \cdot partOf)$. Thus, the answers (John, Chronos), (Tom, IT), (Jane, IT), (Anne, Operations), (Paul, Finance), (John, Finance) and (Zephyr, Chronos) would all be returned at the distance equivalent to substituting one edge label for another. Deleting the edge label *workedOn* to obtain the regular expression $(isManagerOf?)$ would return the answer (Tom, Jane) at the distance equivalent to deleting an edge label. By inserting either the edge label $worksIn^-$ or $workedOn^-$ to obtain, respectively, $(worksIn^- \cdot isManagerOf? \cdot workedOn)$ and $(workedOn^- \cdot isManagerOf? \cdot workedOn)$ ('$^-$' following an edge label denotes that the traversal of the edge is reversed), the answers returned would

be (IT, Zephyr), (Operations, Chronos), (Finance, Chronos), (Zephyr, Zephyr) and (Chronos, Chronos) at the distance equivalent to inserting an edge label.

Query relaxation in the context of RPQs is achieved by applying logical relaxation of a query conjunct using ontological knowledge relating to the data [115]. It has the effect of making the relaxed version of a query conjunct return progressively more general answers, with the latter being returned at an increasing 'distance' from the exact version of the query conjunct (we refer to this distance as the 'relaxation distance').

To illustrate, suppose that the data in Figure 1.1 has an associated ontology in which both *workedOn* and *managed* are subproperties of a property *projectActivity*. Then, a relaxed query conjunct $RELAX(x, managed, Chronos)$ would match one node at distance 0, namely, 'John', and also the nodes 'Paul' and 'Anne' at a relaxation distance equivalent to the cost of relaxing the property *managed* to its parent property *projectActivity*.

1.2 Thesis contributions

Our research in this thesis investigates a flexible querying mechanism for the evaluation of conjunctive regular path queries (CRPQs), in which query approximation and query relaxation are combined within a single framework, with results being returned incrementally to the user in order of increasing combined edit and relaxation distance from the original query. This research has the potential to benefit users by assisting them in formulating queries over complex graph-structured data with which they may not be fully familiar. It provides additional opportunities to users to retrieve results that are of relevance to them by, for example, returning potentially the correct answers (in cases where the original query was incorrectly formulated) or additional related answers, enabling sophisticated data discovery and insights to be made.

In this thesis, we consider a graph data model comprising a directed graph consisting of nodes and edges. This is supplemented by an ontology that represents subclass relationships between class nodes, subproperty relationships between property nodes, and domain and range relationships between property nodes and class nodes. The query language that we adopt is that of CRPQs.

Our research builds on and extends in a number of ways the work in [115], which proposed for the first time a single framework encompassing both query relaxation and query approximation for conjunctive regular path queries. Specifically, the contributions of this thesis are as follows:

- We allow each edit and relaxation operation to have a different associated cost by modifying the construction of the data structures required for the evaluation of the queries, and specifying in more detail the algorithms used for the incremental evaluation of approximated and relaxed query conjuncts.

- We provide full proofs of correctness for the constructs and algorithms used to evaluate approximated and relaxed RPQs. We show formally how the approximate answer to an RPQ can be computed in time that is polynomial in the size of the query and the data graph, with answers being returned in ranked order of their 'distance' from the original query. We also establish that the relaxed answer to an RPQ can be computed in time that is polynomial in the size of the query, the data graph and the ontology graph, again with answers being returned in ranked order.

- We present a prototype system called *ApproxRelax* and, focusing on its user-facing features, we present a qualitative case study in the domain of lifelong learning, showing how *ApproxRelax* overcomes problems of an earlier system by providing more flexible querying capabilities, resulting in answers of greater relevance being returned to the user. *ApproxRelax* implements, for the first time, ontology-based relaxation of regular path queries, as well as combined support for approximation and relaxation of CRPQs.

- We show by means of an empirical evaluation with our final system, *Omega*, how an automaton-based approach built on top of existing technology can be used to implement efficient evaluation of exact RPQs, as well as approximated and relaxed RPQs that mostly execute within a reasonable amount of time.

- We demonstrate how the use of simple graph statistics has the potential to improve the run-times of approximated queries.

- Finally, we introduce and motivate an additional FLEX operator that combines both query approximation and query relaxation into a single operator. We show that FLEX allows additional answers to be returned that cannot be obtained by applying approximation or relaxation alone. We establish that the evaluation of each RPQ whose conjunct has FLEX applied to it can be undertaken using a combination of the techniques used for approximation and relaxation, in time that is polynomial in the size of the query, the data graph and the ontology graph.

1.3 Thesis outline

This thesis is structured as follows.

In Chapter 2, we review related work on graph data models and query languages, keyword-based querying, query relaxation and approximation and subgraph matching. We conclude with a discussion comparing these works to our research.

We provide the necessary background to our research in Chapter 3, introducing the graph data model and the query language, i.e. conjunctive regular path queries (CRPQs). We illustrate approximation and relaxation of query conjuncts by means of examples, provide a formal definition of CRPQs, discuss exact matching of RPQs, and give formal definitions of approximate matching and relaxation of such queries. Finally, we discuss the evaluation of multi-conjunct CRPQs, each of whose conjuncts may be approximated or relaxed, and the complexity of query answering.

In Chapter 4, we provide formal correctness proofs for the constructs and algorithms introduced in Chapter 3. We explore the time complexity for computing the approximated and the relaxed answers to an RPQ, and show that in both cases answers can be returned in ranked order of their 'distance' from the original query in time that is polynomial in the size of the query, the data graph, and, in the case of relaxed answers, the ontology graph.

In Chapter 5, we present the *ApproxRelax* prototype system. Restricting our focus to the user-facing features of *ApproxRelax*, we detail a qualitative case study showing how *ApproxRelax* overcomes problems in a previous system in the same application domain of lifelong learning through its support for approximation and relaxation of CRPQs.

Chapter 6 describes the implementation details of the *Omega* system, our final implementation of query approximation and query relaxation, which supersedes the *ApproxRelax* system. We discuss the system architecture, data structures, data model and API, describe how data graphs are created in *Omega*, and present the query evaluation algorithms. We conclude with a review of other implementations of regular path query evaluation on graph-structured data.

In Chapter 7, we present a performance study conducted with two contrasting real-world data graphs. We establish that our baseline performance of exact RPQs is comparable to that of state-of-the-art systems, and show that, in most cases, our APPROX and RELAX queries exhibit reasonable performance. For the few queries that perform poorly, we examine the causal factors. We illustrate by means of an empirical evaluation how the availability of simple statistics relating to paths in the data graph can be used to improve the run-times of a selection of poorly-performing approximated queries.

In Chapter 8 we discuss the application of both approximation and relaxation to an individual query conjunct using one query operator, FLEX, thus providing additional flexibility for users by not requiring them to select either approximation or relaxation for a query conjunct. We describe how RPQs that have FLEX applied to them can be evaluated, providing formal proofs of correctness. Query evaluation is again accomplished in time that is polynomial in the size of the query, the data graph and the ontology graph, with answers being returned in ranked order. We also discuss the characteristics of multi-conjunct CRPQs in which conjuncts can have FLEX applied to them, considering query evaluation, complexity, and expressiveness.

Finally, Chapter 9 summarises the contributions of the thesis, giving our concluding remarks and directions for further work.

CHAPTER 2

Literature Review

We begin this chapter with setting the context for our research by reviewing literature regarding graph data models and graph query languages in Section 2.1. Keyword-based querying allows for a form of flexible querying and is discussed in Section 2.2. As query relaxation and query approximation are fundamental to our work, previous work in these areas is presented in Section 2.3 and Section 2.4, respectively.

Subgraph matching is also related to our research, i.e. retrieving a graph matching certain query conditions from a larger graph, with the former being defined as a *subgraph*. We discuss research in this area in Section 2.5.

We end with a discussion in Section 2.6 comparing our work with the approaches taken by the research reviewed.

2.1 Graph-modelled data and graph query languages

We begin by reviewing research undertaken on graph data models and query languages over the past few decades, after which we proceed to discussing more recent work.

2.1.1 Graph models

Since the 1990s, much work has been undertaken in the development of graph models and we briefly summarise some of these here; we note that a comprehensive survey of graph data models is presented by Angles and Gutierrez [7]. We begin with general graph data models and the hypergraph model, followed by semi-structured data models and RDF, and end with a summary of attributed graphs.

The GOOD (Graph Object-Oriented Data) [54] proposal by Gyssens *et al.* models the database schema and the data instances as directed labelled graphs, in which transformations of the graph are used to manipulate the data. GDM (Graph Data Model) by Hidders [59, 60] extends this work through the addition of inheritance and complex values, along with n-ary symmetric relationships. Based on GOOD, Andries *et al.* [6] propose GMOD (Graph-Oriented Object Manipulation) which models object database entities as graph-oriented user interface entities. The schema contains nodes representing abstract objects (class names) and basic types (primitive objects), with edges representing properties of the abstract objects. Both the schema and the data are represented as labelled, directed graphs, and, in contrast to GOOD, GMOD uses graph pattern matching when querying and mutating data. Güting [53] presents GraphDB, in which graphs are modelled in an object-oriented environment so that objects may be dealt with as nodes, edges and explicit paths within the graph; the original application of this model was for spatial networks. Amann and Scholl present Gram [4], a model for hypertext data consisting of a directed graph with labelled nodes and edges. In contrast to GOOD, both GraphDB and Gram allow for path queries. The GGL (Graph Database System for Genomics) model by Graves [49], arising from the biological community, takes advantage of the benefits of graph models to store and query genome maps as directed, labelled graphs, with the additional ability to model hierarchies by allowing the nodes to contain graphs.

Turning now to the hypergraph data model, Levene and Poulovassilis discuss GROOVY (Graphically Represented Object-Oriented data model with Values) [90], which uses a hypergraph model to represent an object-oriented model. Edges in a hypergraph, called hyperedges, may connect two or more nodes. This model inspired, among others, the hypernode model [88, 89, 112] which is based on a nested graph structure (this is a directed graph whose nodes may be graphs) and

can be used to typify simple, hierarchical and composite objects in order to model complex information. HyperGraphDB[1] [69] is an implementation of the hypergraph data model, such as that described above for the GROOVY model.

We now briefly discuss OEM (Object Exchange Model), XML (eXtended Markup Language) and Lore (Lightweight Object Repository), as these exemplify semi-structured data models and are thus more closely related to graph-structured models than are structured data models, such as relational and object-oriented data. OEM [104], motivated by the challenges of integrating heterogeneous, evolving data sources, allows for the flexible modelling of schemaless, complex entities using concepts — such as nesting — from object-oriented models. XML [12] introduces a standard mechanism by which data can be exchanged between Web applications. In contrast to graph models, XML has a tree-like structure. The data is self-describing and can provide for richer semantics when combined with a schema. Lore [1] is a prototype database management system allowing for the storage, updating and querying of semi-structured data. It includes (i) a data guide, encapsulating by means of edge labels the structure of the graph, thus acting as a structural summary of the database, and (ii) external objects, allowing Lore to integrate information from external data sources.

RDF (Resource Description Framework) [78] is a W3C recommendation introduced to represent metadata in a graph-structured way by describing resources and their interconnections in a flexible, extensible manner. RDF is composed of triples, each of which contains a *subject* describing the resource, a *predicate* which is a property of the resource, and an *object*, which is the value of the property. A triple is a statement of the relationship between the subject and the object, and a set of triples can be represented as a graph, with the subjects and objects being nodes, and the predicates being the edges connecting them. Triple stores — also known as RDF stores — store RDF graphs in subject-predicate-object form and are exemplified by systems such as Virtuoso[2]. Allegrograph[3] is an example of an RDF database used to build Semantic Web applications, and stores both the data and schema as triples (RDF, and RDF/S, respectively).

Attributed graphs, also widely known as *property graphs* [119], are becoming

[1] `http://www.hypergraphdb.org/`
[2] `http://virtuoso.openlinksw.com/`
[3] `http://franz.com/agraph/allegrograph/`

increasingly popular in industry and academia. Such graphs allow for any number of *attributes* — i.e. key-value pairs — to be associated with the nodes and edges. Neo4j[4] [119] is the most popular property graph database[5]. It stores graphs natively on disk and provides a framework for traversing graphs and executing graph operations, as well as a declarative query language, Cypher, which is discussed in the next section. Sparksee (formerly known as DEX) [98], another property graph database, provides APIs for different programming languages allowing for the storage and manipulation of graphs. We have used Sparksee for our implementation[6], and further details are provided in Chapter 6.

2.1.2 Graph query languages

Along with graph data models, graph query languages also have a history spanning several decades.

The languages G [24], its successor G+ [23, 99, 100] by Cruz, Mendelzon and Wood, and GraphLog [21] by Consens and Mendelzon are all based on conjunctive regular path queries (CRPQs), using a labelled, directed graph model. G matches simple paths in the graph, and G+ includes aggregation operators, such as count, sum, min and max, which allow for, among others, the computation of graph characteristics such as node degree and distances between pairs of nodes. GraphLog extends G+ through the addition of edge inversion and negation, and translates the regular path expressions within the query to Datalog for evaluation.

The query languages for GraphDB [53] and Gram [4] permit regular expressions to be defined over alternating node and edge type sequences, starting and ending with a node type. Additionally, GraphDB includes operations such as the computation of the shortest path between a pair of nodes.

GGL (Graph Database System for Genomics) [49], intended for use in the biological community, introduces a genome graph language and operators in [50] using graph matching and path matching techniques. GMOD (Graph-Oriented Object Manipulation) [6] also uses graph pattern matching techniques to both query and

[4] `http://neo4j.com/`

[5] `http://db-engines.com/en/ranking/graph+dbms` (June 2016)

[6] The choice to use Sparksee was made for purely pragmatic reasons, most notably because of their provision of a C# API.

manipulate the data.

The querying mechanism in GOOD (Graph Object-Oriented Data) [54] is based on graph transformation operations, such as additions and deletions of nodes and edges, along with a construct by which objects can be grouped according to their properties and a method to define sequences of operations. A successor to GOOD is the query language G-Log [105], operating on the GMOD [6] model. G-Log retrieves matching subgraphs by using graph-based rules specifying transformations to the schema and data along with patterns using predicates in the edges.

HML (Hypergraph Manipulation Language) is used to query and update labelled hypergraphs in the GROOVY [90] model. There are eight operators allowing for the creation and deletion of hyperedges and hypergraphs, as well as two operators allowing the hypergraph to be queried. This was extended by Poulovassilis *et al.* in Hyperlog [111, 112], a rule-based query language supporting querying and database browsing along with the ability to perform database updates.

The functional query language UnQL by Buneman *et al.* [14] uses regular expressions for querying semi-structured data. UnQL makes use of a top-down, recursive function, thus allowing for structural recursion. Remaining within the area of semi-structured data, the language Lorel by Abiteboul *et al.* [1] uses regular expressions for querying data in the Lore data model, as does the system described by Fernández *et al.* [39]. Moreover, Lore also allows for path *variables* to be used, where a path – denoted by a regular expression – can be bound to a variable. Goldman and Widom [43] use automata to evaluate regular path queries for semi-structured data, a method that we too adopt in our approach.

More recently, CRPQs are used in SPARQLeR [79], PSPARQL [3], nSPARQL [107], SPARQL 1.1 [122] and NAGA [75] — these query languages are discussed in more detail in Sections 2.1.3 and 2.1.4.

Approaches proposed for evaluating (exact) CRPQs include automaton-based approaches [43, 82], translation into Datalog or recursive SQL [22, 28, 57, 135], search-based processing [36, 82] and reachability indexing [51].

Fan *et al.* [36] discuss adding regular expressions as edge constraints on the graph patterns to be matched. The authors discuss a class of reachability queries and a class of graph patterns. For the latter, a path consisting of a restricted form of regular expression is used, which can be evaluated in cubic time.

Cypher[7], inspired by SQL, XPath and SPARQL, is a declarative, pattern-matching graph query language used by the Neo4j [119] property graph database, and it allows a restricted form of regular path queries to be expressed: the concatenation and disjunction of single edge types, as well as variable length paths, in which optional upper and lower bounds may be set. Additionally, paths are able to be returned by a Cypher query.

None of the above languages allows for query approximation or relaxation.

2.1.3 SPARQL

In recent years, SPARQL [122] has emerged as the de-facto standard for querying RDF data [78]. It has an SQL-like syntax and at the most elemental level the queries are based on graph patterns, using graph pattern matching to find solutions.

Kochut and Janik [79] introduce SPARQLeR, which extends SPARQL by using regular expressions (with various filters and conditions applied) to query the data, discover pathways in the data graph, and return paths as output; only simple (i.e. acyclic) paths are considered.

Anyanwu *et al.* [8] introduce SPARQ2L, which allows for path extraction queries from RDF data sources, by using path variables and path variable constraint expressions. Alkhateeb *et al.* [3] present an extension for regular expression patterns within SPARQL, entitled PSPARQL. Users are able to search paths within RDF graphs by using regular expression patterns (regular expressions with variables) as well as to query RDF through the use of a triple pattern. In addition, PSPARQL is not limited to finding only simple paths.

Motivated by the path-based navigational constructs within SPARQLer, SPARQ2L and PSPARQL, the most recent version of the SPARQL standard, SPARQL 1.1, introduces *property paths*, which allow paths between any two nodes to be expressed by regular expressions [9, 122].

Pérez *et al.* present nSPARQL, which adds nested regular expressions to SPARQL, and show that these are necessary to answer queries using the semantics of the RDFS vocabulary by directly traversing the RDF graph, without materialising the closure of the graph [107].

[7]`http://neo4j.com/docs/developer-manual/current/cypher/`

Using the concept of nested regular expressions from nSPARQL and XPath's ability to express nested predicates on paths, Zauner *et al.* [141] discuss RPL (RDF Path Language), which allows conditional regular expressions to be expressed over the nodes and edges appearing on paths within RDF data.

In contrast to our work, there is no query relaxation or approximation in any of the above languages.

2.1.4 Linked and distributed data

SPARQL is used widely to query linked data, and this is potentially another means by which data on the web may be accessed and queried. Hartig *et al.* [55] present concepts and algorithms to facilitate this efficiently, based on traversing RDF links in order to detect during query execution what data may be relevant to the query.

Langegger *et al.* [83] present a system providing SPARQL access to distributed data sources, the main motivating factor for the system being one of allowing scientific communities to share data. A significant component of this system is concerned with the automated registration and integration of different data sources, as well as executing federated queries across these data sources. Features of SPARQL such as graph pattern matching (including optional and alternative pattern matching) and filters have been implemented.

Kasneci *et al.* [75, 134] introduce the idea of integrating linked data and information retrieval methods to extract or infer useful, hitherto unknown relationships between data sources. The system, called YAGO, integrates information from Wikipedia and WordNet (a lexical database of English[8]). A query language supporting CRPQs for YAGO — called NAGA [76] — adopts concepts from SPARQL, but also extensions such as more expressive pattern matching and ranking. The ranking in NAGA uses statistical methods and can only rank exact matches to a given query.

[8] `http://wordnet.princeton.edu/`

2.2 Keyword-based querying

Keyword-based querying allows for a form of flexible querying. A keyword-based query is one in which the structure of the data is not considered, i.e. *any* part of the data matching the provided keyword causes the data entity to be returned (e.g. a tuple if the data source is relational; an XML node or subtree if the data source is XML; or a set of nodes if the data is graph-based). For example, if *Harry Jones* is provided as the keyword in such a query, and the data source consists of records pertaining to books, results in which *Harry Jones* appears as either author or book title will be returned. Much work has been done on keyword querying for many types of data (from structured to unstructured), which we briefly review here.

XML's XPath and XQuery languages have both recently been extended with full-text search capabilities[9]. This allows users to provide keywords — which may be words or phrases — as part of their XPath or XQuery expression. The query uses information retrieval techniques such as scoring and weighting, and is thus far more extensive than, say, a simple 'substring' search query. This means that data containing the exact keyword(s) as well as related terms will be returned.

Xu and Papakonstantinou [136] propose keyword search algorithms for XML documents that return the set of all XML subtrees containing the keywords. The trees are not ranked in any way.

Regarding keyword search in SPARQL, Elbassuoni *et al.* [33, 34] discuss the extension of SPARQL with keyword search. Such a query allows a varied number of keywords to be associated with one or more SPARQL query conjuncts when querying an RDF data repository.

Querying data via a keyword-based search is, at first glance, a guaranteed way of achieving flexibility. The lack of structural knowledge in such queries means that it solves the challenges raised by the heterogeneous nature of the data and structures, the ever-changing nature of the structures (caused by new data sources joining the 'data pool' on a regular basis) and the users' lack of knowledge of the data structures and relationships in the data.

However, one major flaw with this approach is that one is not able to apply the notion of *semantics* or *structure* within the search, which means that the answers

[9] `http://www.w3.org/TR/xquery-full-text/`

returned may not meet the user's requirements. A simple example to illustrate this is the following: using the keyword-based search technique, a user enters *Harry Potter* into a large repository consisting of books and related publications, hoping to find a book relating to feuding Scottish families in the sixteenth century ('Harry Potter' is the name of the author of a book entitled *Blood Feud: The Murrays and Gordons at War in the Age of Mary Queen of Scots*). Obviously, the multitude of books (and numerous editions thereof) relating to *Harry Potter* (the boy wizard) will be returned and overwhelm the search results needlessly and irrelevantly, from the point of the user. It is therefore clear that being able to provide *some* context within a query would be very useful (either via semantics or by structure) in expressing the end goal of the user's query more effectively.

To this end, Dong and Halevy [31] discuss the provision of a querying mechanism in which keywords along with structural requirements are combined in the query (both are optional), in order to allow users to query data sources more flexibly but also more meaningfully. The user is able to specify keywords and a structural condition — an example being 'a person with a name of *Mary* who was born in *London*' — or simply just a set of keywords. The motivation here is that the user is able to provide the notion of structure if they know it (and, indeed, this will return more meaningful answers), but, if not, they are able to revert to providing keywords only. Answers returned also include entities related to the ones containing the actual keyword(s), although these will be ranked lower. Ranking is achieved by a variety of mostly statistically-based methods.

Cohen *et al.* [20] discuss a semantic search engine, entitled XSEarch, which can be used to query XML data and rank results by integrating the notion of keywords and semantics. Answers to XSEarch queries are returned based on whether they match the keywords, whether they are semantically related, and whether the semantically-related answers are relevant to the keywords. The answers are ranked using IR techniques.

Pound *et al.* [116] recognise the problems inherent with keyword search whilst still accepting its usefulness as a flexible querying paradigm. The authors propose a solution consisting of a combination of the keyword-based technique with the more targeted structure-based technique. Their QUICK system enables a user to define a query consisting of an entity, which is described by a collection of keywords, and a

description of all the entity's relationships to other entities. The system then returns answers ranked according to various statistically-based methods.

The above proposals [20, 31, 116] all provide an additional, albeit limited, capability for the user to provide additional contextual information to facilitate the retrieval of more meaningful results. In contrast, the context of the data and its structure are integral to our research.

2.3 Query relaxation

Query relaxation is a fundamental part of our research and we present here work undertaken in this area in querying semi-structured and graph-structured data, focusing first on SPARQL and RDF, and ending with XML.

SPARQL itself allows for optional pattern matching[10]. To illustrate with an example, a user may pose a SPARQL query requesting all names and email addresses from an RDF graph, indicating that the email address is optional. This will return all names having email addresses as well as those names where the email address is not present.

User and domain preferences play an integral part in the query relaxation techniques described by Dolog *et al.* [29, 30], in which aspects such as query rewriting rules and user preferences are able to be configured and subsequently applied in order to yield the best results according to what the user deems important within a particular context. Relaxation is then carried out (making use of an ontology or schema), where more generic terms or concepts are used in subsequent — i.e. more relaxed — versions of the query. Regarding the ranking of results, their implementation uses a *divide and conquer* approach, whereby the best results of each possible combination of query re-writings is returned (in a breadth-first search).

The work by Hurtado *et al.* [66] proposes techniques for query relaxation in the setting of the RDF/S data model with respect to an RDFS vocabulary. The authors show that query relaxation can be naturally formalised using RDFS entailment. The entailment is characterised by derivation rules regarding subproperties, subclasses and typing, grounded in the semantics developed by Gutierrez *et al.* [52, 56].

Several proposals have been made in which query relaxation is based on the use

[10] `https://www.w3.org/TR/rdf-sparql-query/#OptionalMatching`

of similarity measures to retrieve additional relevant answers. For example, Hogan *et al.* [63] apply similarity functions to constants such as strings and numeric values, and in [64, 65, 118] ontology-driven similarity measures are developed, using the RDFS ontology to retrieve additional answers and assign a score to them. Elbassuoni *et al.* [35] discuss undertaking query relaxation based on statistical language models (the goal of such a model is to estimate as accurately as possible the probability distribution of sequences of words in natural language processing) for RDF data and queries. Flexible querying of RDF using SPARQL and preferences expressed as fuzzy sets is investigated by Buche *et al.* [13].

Mochol *et al.* [102] propose query relaxation using ontologies within the HR (Human Resources) domain to improve online recruitment applications for both employers and employees. Query rewriting techniques are applied to user-defined queries posed over RDF data, in which specific terms are replaced with more general terms to obtain relaxed versions of the original query, by making use of the subclass-superclass relationships within the ontology. Regarding ranking of the results, mention is made of employing semantic similarity functions to do this, but this has yet to be implemented in the system. Our approach additionally uses the *subproperty*, *domain* and *range* ontology constructs, and is applicable to any graph-structured data application.

Meng *et al.* [101] explore query relaxation by utilising data relating to the personal preferences of the user to relax the numerical and categorical constraints of the original query; for example, the original numerical range specified in a query may be expanded. The contextual preferences of the user are obtained by using association-rule mining on the database log of past queries. In contrast, our approach uses ontological constructs to achieve query relaxation.

However, techniques and ideas from the approach of Meng *et al.* could be used to enrich the relaxation operations we develop here. In particular, depending on the user's preferences, differing weights could be applied to the various ontological constructs and the relationships within a particular ontology. For example, to denote a biologically closer connection between all mammals, than, say, between a mammal and a bird, the relationship between (the superclass) *Mammal* and all its immediate subclasses, such as *Cat* and *Dog*, may be given greater weight than the relationship between *Mammal* and all *its* immediate superclasses (such as *Vertebrate*).

Elbassuoni *et al.* [33] discuss a proposed extension to SPARQL with keyword search capabilities (which was discussed earlier in this section). However, the authors also discuss the relaxation of triple patterns (RDF query conjuncts) by replacing constants with variables. The larger the number of relaxations (which is dependent on the number of keywords and constants in the original query), the lower the rank of any answers arising from the (relaxed) query. The relaxation of the keywords is achieved by employing the use of various IR-based techniques, which differs from our ontology-based approach to query relaxation.

Cedeño and Candan [19] describe a framework for cost-aware querying of weighted RDF data through predicates that express flexible paths between nodes. We discuss this work in detail in Chapter 6.

Zhou *et al.* [144] explore the idea of achieving query relaxation on Entity-Relationship data models by using *malleable schemas*, which contain multiple, overlapping definitions of data structures and attributes, and may be extended at any time. The resulting redundancies that arise are intended to capture more fully the diverse semantics of a complex, heterogeneous domain; for instance, *author* and *writer*, or *paper* and *report* are two examples where multiple terms have overlapping meanings for a particular attribute. Duplicate terms in different datasets are used to calculate correlations (and strengths thereof) between elements within the malleable schema; these are then used to relax the queries and rank the results. This approach is thus firmly grounded in a statistically-based model, in contrast to our work.

Turning now to research on query relaxation for XML data, work has been done on relaxing tree pattern queries for XML, e.g. in [5], [125] and more recently in [91]. We discuss each of these below.

Theobald *et al.* [125] present TopX, a system designed to rank retrieved XML documents without relying upon a schema (as it is very likely that, in the heterogeneous data context, many XML documents will not have accompanying schemas). The system relaxes queries by expanding the query using information from an ontology or thesaurus. Various IR-style techniques are used to achieve this, as well as to rank the answers.

Amer-Yahia *et al.* [5] discuss FleXPath, a system integrating both structural and keyword-based XML querying, much like the work in [116] described earlier

(although that work was not restricted to XML documents). The structural relaxation of the query is achieved by various methods which lead to the dropping of conditions from the original expression (which is a query tree pattern), leaf node deletion or subtree promotion — in effect, extending the search space through the removal of conditions within the XPath expression. This is integrated with full-text search capabilities. Ranking is achieved by the degree of similarity of the relaxed expression's structure to that of the original, as well as statistically-based scoring methods. FleXPath therefore still requires the user to be familiar with the structure of the XML to some extent, in order to formulate the original query.

Liu *et al.* [91] discuss relaxation of queries posed against heterogeneous XML data sources. Each data source has its own schema (a DTD, in this case), and the incoming query is relaxed based on the rules within this schema. This means the original query is relaxed differently for each data source if the sources have differing schemas. The ranking of results is achieved by ranking the relaxed queries according to how much the relaxed query differs from the original. There is the added flexibility of user configuration, whereby a user may specify weights on the relationships expressed within the schema; these are then taken into account when computing the ranking.

Yu and Jagadish [140] discuss a flexible query model aimed at XML and relational data sources. The notion of a *schema summary* is defined, which is a condensed description (containing only the most salient structures and relationships) of the full database or XML document schema; the former is, from a user's point of view, easier to understand than the latter. The schema summary is used to construct a new model called the *Meaningful Summary Query* (MSQ). Such a query requires the user only to have knowledge of the schema summary. From there, the MSQ query is evaluated by making use of schema-matching semantics. Ranking is mentioned, but only briefly. The authors state that more research to develop the ideas further is needed.

In this thesis, we build specifically on the query relaxation approach of Hurtado *et al.* [66], and combine it with query approximation within one integrated flexible querying system for CRPQs.

2.4 Query approximation

The approximate matching aspect of our research is related to a large volume of other work on query approximation for semi-structured and graph-structured data, and we now review the major relevant related work. We begin by focusing on SPARQL and RDF, followed by general graph-structured data, and then end with XML.

There have been several proposals for applying flexible querying to Semantic Web data, mostly based on the use of similarity measures to retrieve additional relevant answers. For example, similarity-based querying is the focus of Kiefer *et al.* [77], with the introduction of iSPARQL, an extension of SPARQL that supports customised similarity functions. One similarity measure used is that of edit distance between strings. However, similarity is measured with respect to the resources themselves rather than the paths connecting resources, i.e. using the edit distance between the names of the resources. In contrast to [77], our research investigates query approximation by applying a range of edit operations to the portion of the query denoting the paths in the graph to be traversed, as well as query relaxation by using the ontology associated with the data.

Turning now to approximate querying for graph-structured data, Kasneci *et al.* [74] discuss the importance of finding relationships between nodes in a graph where the nodes are connected by weighted edges. The authors present STAR, an approximation algorithm for determining relationships over large graphs. STAR returns approximations of the original (exact) query, by using the original query as a template for an optimal Steiner tree (given a collection of nodes N within a graph, a *Steiner tree* of N is a minimum-weight connected subgraph consisting of all the nodes within N). In addition, any information pertaining to the taxonomy associated with the graph data is exploited by the algorithm. This algorithm is used as part of the NAGA system [75, 76].

Mandreoli *et al.* [95] discuss flexible query answering on graph data, with ranked results. The authors present two modes of approximation. The first mode is termed *label approximation*, which essentially replaces one edge label by another. The authors suggest doing this by using linguistic information extracted from an external source, such as WordNet, and computing the ‘distance’ between the terms based on

various linguistic statistical formulae. The second mode is termed *structural approximation* and involves the establishment of 'semantic relatedness' between any pair of nodes – based on property relaxation, node connectivity, and domain and range relationships – along with an associated cost.

Yang *et al.* [138] introduce SLQ, a framework which enables schemaless and structureless querying of property graphs in which answers approximately matching the query are returned in ranked order, by means of an automatically learnt ranking model using generated training data. A set of transformation functions — comprising string (e.g. 'Anne Smith' → 'Anne'), semantic (e.g. 'lecturer' → 'teacher'), numeric (e.g. ' 20 yrs' → '22 yrs') and edge to shortest path (e.g. 'X'-'Z' → 'X' - 'Y' - 'Z') transformations — is applied to the attributes and values of nodes and edges within the original query, resulting in multiple answers that are subsequently ranked according to the ranking model.

Grahne and Thomo [46, 47] explore approximate matching of single-conjunct regular path queries using a weighted regular transducer to perform transformations to regular path queries (but not CRPQs) for approximately matching semi-structured data, with polynomial time complexity in the size of the query and the graph. Hurtado *et al.* [67] build on techniques from [46, 47, 68] to show that approximate matching of CRPQs can be undertaken in polynomial time, subject to certain assumptions which we discuss in Chapter 3. The query edit operations considered are insertions, deletions and substitutions of edge labels, inversions of edge labels (corresponding to reverse traversals of graph edges), and transpositions of adjacent labels, each with an assumed edit cost. Query results are returned incrementally to the user in order of their increasing edit distance from the original query. In other work [48], Grahne and Thomo introduce preferential RPQs where users can specify the relative importance of symbols appearing in the query by annotating them with weights. Poulovassilis and Wood [115] extend [67] by proposing combining approximation and ontology-based relaxation for CRPQs. Either approximation or relaxation can be applied to each conjunct of a CRPQ, although edge inversions and transpositions are not considered within the set of edit operations. This thesis builds on the work in [115] by allowing each edit and relaxation operation to have a different associated cost, providing full proofs of the correctness for the constructs and algorithms necessary to evaluate approximated and relaxed CRPQs, describing

a system implementation of the approximation and relaxation of CRPQs, presenting a performance study of the system, and discussing optimisations for the evaluation of such queries.

Early work on query approximation by Kanza and Sagiv [72] considered querying semi-structured data using flexible matchings which allow paths whose edge labels contain those appearing in the query to be matched. Such semantics can be captured by transposition and insertion edit operations on edge labels in our framework.

We now briefly discuss research undertaken on query approximation for XML data. Buratti and Montesi [15] discuss a system allowing approximate answers to be returned from an XML document store, by approximation of the structure within an XQuery FullText expression. The authors define the concept of a *path edit distance*, which is calculated to be the minimum number of edit operations required in order to transform one path into another. These edit operations include insertions, deletions and substitutions of path steps. Each of these has a pre-defined cost, depending on where in the corresponding node tree the transformation is to take place; so, if the transformation operation extends the search space, it is more 'expensive' than a transformation which narrows a search space (relative to the original expression). The answers are then ranked according to a 'satisfaction' ratio, which is computed using a statistical scoring method for numeric values and similarity metrics calculated from an ontology for string values. By contrast, our research considers any graph-structured data source, using regular path expressions within conjunctive path queries instead of queries posed using XQuery FullText. However, both approaches use the notion of an *edit distance* within the context of query rewriting; [15] uses it to rewrite paths within the XQuery FullText expression, whereas our approach uses it to approximate regular path queries.

2.5 Subgraph matching

Subgraph matching comprises the retrieval from a graph of subgraphs matching certain conditions. This area is relevant to our research since the algorithms proposed could potentially be utilised for improved performance of flexible CRPQ evaluation.

Approximate subgraph matching has been extensively studied recently, e.g. [37, 93, 126, 143, 146], and we discuss each of these works below.

Zhang *et al.* [143] present the SAPPER model, which sets out to find all instances of a query graph within a large graph database. The model developed uses an *edge edit distance* to return exact subgraphs as well as approximated subgraphs matching the original query but with possibly missing edges (provided the edge edit distance does not exceed a certain threshold value).

Tian and Patel [126] present the TALE method, a heuristic algorithm in which important nodes within the graph are matched first, after which the search is extended. This heuristic algorithm is not guaranteed to find all or even the best matches, unlike [143] and our framework in which all answers at a particular distance will always be returned.

Fan *et al.* [37] discuss top-k graph pattern matching on social graphs, with the eventual aim being to conduct graph pattern matching efficiently on large social datasets by developing distributed algorithms on partitioned and distributed graphs. The authors use relevance functions based on factors such as social impact and distance to compute the top-k matches.

Ma *et al.* [93] present a set of criteria preserving the topology of massive graphs. This approach rectifies problems caused by pattern matching on a data graph whose structure differs greatly from the pattern, which may result in matches that are difficult to comprehend and analyse.

Zou *et al.* [146] discuss the gStore system, in which SPARQL queries are transformed into subgraph matching queries. The flexible querying aspect of this system arises from the efficient and scalable provision of wildcard SPARQL queries — i.e. making use of the 'regex' filtering construct in SPARQL.

Varadarajan *et al.* [131] propose a framework entitled GID (Graph Information Discovery), whose aim is to facilitate user queries on hyperlinked data, and where answers are ranked according to the user's input criteria. The data model consists of a data graph and a schema graph. GID allows users to pose a series of 'filters', which may be combined. Briefly, a filter consists of various selection conditions comprising: a Boolean keywords expression (such as `keywords = 'magnesium' OR 'sulphur'`); an attribute-value pair (such as `title = 'The history of oxygen'`); a type; and, lastly, a path expression (such as `path = 'OntoGene/PubMed'`). In addition, there is a flag to denote whether to apply ranking or not (if true, then the filter also specifies which selection condition must be used as the ranking 'pivot', and ranking

is achieved by means of various statistically-based techniques). The query is thus a series of user-defined filters, which, when executed, returns a subgraph. In contrast to our research, this approach does not allow for approximations or relaxations of query paths, and the user is required to have some knowledge regarding the data structure from the outset.

2.6 Discussion

In order to provide the context in which our research is grounded, we began this chapter by reviewing both historical and current graph data models, thence proceeding to graph query languages. We reviewed historical work, and continued by discussing more current languages, such as Cypher and SPARQL. Recent work in storing and querying linked data was presented, in which the benefits conferred by flexible querying would be substantial; nonetheless, to date flexible querying has mainly used statistically-based methods.

We subsequently reviewed work on keyword-based querying, showing that its efficacy is impeded by a lack of context, as neither structure nor semantics are encompassed. A number of papers [20, 31, 116] seek to address this shortcoming by providing a limited user-defined ability to incorporate within the queries contextual and structural information.

We then surveyed recent work on the relaxation of queries posed against RDF and XML data. Hurtado *et al.* [66] propose techniques for queries posed against RDF data to be relaxed with respect to an RDFS vocabulary (the ontology), showing that query relaxation can be naturally formalised using RDFS entailment over the ontology. However, only conjunctive queries — rather than conjunctive regular path queries — are considered in [66]. Poulovassilis and Wood [115] extend [66] by considering ontology-based query relaxation over CRPQs.

We then proceeded to review current work on query approximation. One query approximation approach, complementary to our research, considered the application of customised similarity functions in an extension to SPARQL [77].

The work in [95] presents two approaches on query approximation for graph data: the replacement of edge labels using extracted information from an external linguistic source, and the establishment of the degree of relationship between two

nodes. By contrast, our approach requires no prior establishment of the degree of relationship (semantic or otherwise) between two nodes as this information is obtained automatically from the accompanying ontology, and our model additionally allows for the matching of paths with regular expressions.

Another query approximation approach for graph data considers a framework based on an automatically learnt ranking model, providing a set of transformation functions which may be applied to nodes and edges, as well as attributes thereof, within property graphs [138]. Two approaches to query approximation for XML data were discussed [15, 72], whose ideas are either captured or subsumed by our approach.

Work on subgraph matching was reviewed, and two main strands arose from this: general approximate subgraph matching using heuristics or edge edit distance, and user-defined ranking being applied to queries on hyperlinked data. This work is complementary to ours in that the techniques proposed could potentially be used to optimise the performance of flexible CRPQ evaluation.

In contrast to all the research described in this chapter, [115] builds on earlier work [66, 67] to propose a single framework encompassing both query relaxation and query approximation for queries comprising conjunctions of regular path expressions, as well as the ranking of results. This thesis extends that work by allowing each edit and relaxation operation to have a different associated cost (rather than assigning the same cost to a group of operations); modifying both the constructs and algorithms required for query evaluation; providing the algorithms in full; presenting formal proofs of the correctness and complexity of the constructs and algorithms; describing the physical implementation of the algorithms, presenting a performance analysis and discussing optimisations; and, finally, presenting an additional operator combining both query approximation and query relaxation into a single operator, along with formal proofs of correctness and complexity of the constructs used.

In the next chapter, we provide the necessary background to our research, introducing the graph-based data model, the query language, and query approximation and relaxation, building upon and extending concepts from [115].

CHAPTER 3

Theoretical Preliminaries

We begin this chapter by introducing in Section 3.1 the graph-based data model adopted in this thesis, which comprises a data graph and an ontology graph. This is followed in Section 3.2 by the definition of the query language, which supports conjunctive regular path queries (CRPQs). We illustrate approximation and relaxation of query conjuncts by means of two examples.

In Section 3.3, we give a formal definition of CRPQs and discuss exact matching of single-conjunct RPQs, providing more detailed definitions of concepts introduced in [115].

Query approximation is discussed in Section 3.4, and we build on the work in [115] by allowing each type of edit operation to have a different associated cost (rather than assigning the same cost to all edit operation types), modifying the construction of the elements required to evaluate approximated queries, and providing full algorithms.

In Section 3.5 we discuss relaxation of single-conjunct RPQs based on information from an ontology, showing how answers for a conjunct to which the RELAX operator has been applied can be computed. This, too, builds on the work in [115], but, as for query approximation, considers each type of relaxation operation to have a different associated cost.

For both query approximation and query relaxation, we show how approximated or relaxed answers can be returned to the user incrementally.

In Section 3.6, we discuss the evaluation of multi-conjunct CRPQs, each of whose conjuncts may have APPROX or RELAX applied to them. Finally, in Section 3.7, we summarise the main elements discussed in this chapter.

3.1 The data model

In this thesis we consider a general graph-structured data model comprising a directed graph $G = (V_G, E_G, \Sigma)$ and a separate ontology $K = (V_K, E_K)$. The set V_G contains nodes, each representing either an entity instance or an entity class, which we term entity nodes and class nodes, respectively. The set $E_G \subseteq V_G \times (\Sigma \cup \texttt{type}) \times V_G$ represents relationships between the members of V_G. If $e = (x, l, y) \in E_G$, then l is called the *label* of edge e. Node x is the *source* of e, while y is its *target*. We assume that the *alphabet* Σ is finite and that $\texttt{type} \notin \Sigma$.

The set V_K contains nodes, each representing either an entity class or a property; in other words, V_K is the disjoint union of two subsets V_{Class} and V_{Prop}, the subset of class nodes and the subset of property nodes, respectively.

The edges in E_K capture subclass relationships between class nodes, subproperty relationships between property nodes, and domain and range relationships between property nodes and class nodes. Hence, $E_K \subseteq V_K \times \{\texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\} \times V_K$. We assume that $\Sigma \cap \{\texttt{type}, \texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\} = V_{Prop} \cap \{\texttt{type}, \texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\} = \emptyset$, and the set of class nodes of V_G are contained in V_{Class}.

Our graph model comprises a strict subset of the RDFS vocabulary: `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`, which we abbreviate in this thesis by the symbols `type`, `sc`, `sp`, `dom`, `range`, respectively. We observe that this general graph model encompasses RDF data, except that it does not allow for the representation of RDF's 'blank' nodes (which are indeed discouraged by some authors for linked data [58]), nor for containers and reification. These and other aspects of RDF are beyond the scope of this thesis and we leave their consideration as future work.

Example 3.1. (Drawn from [114]). We now present a data graph G and an

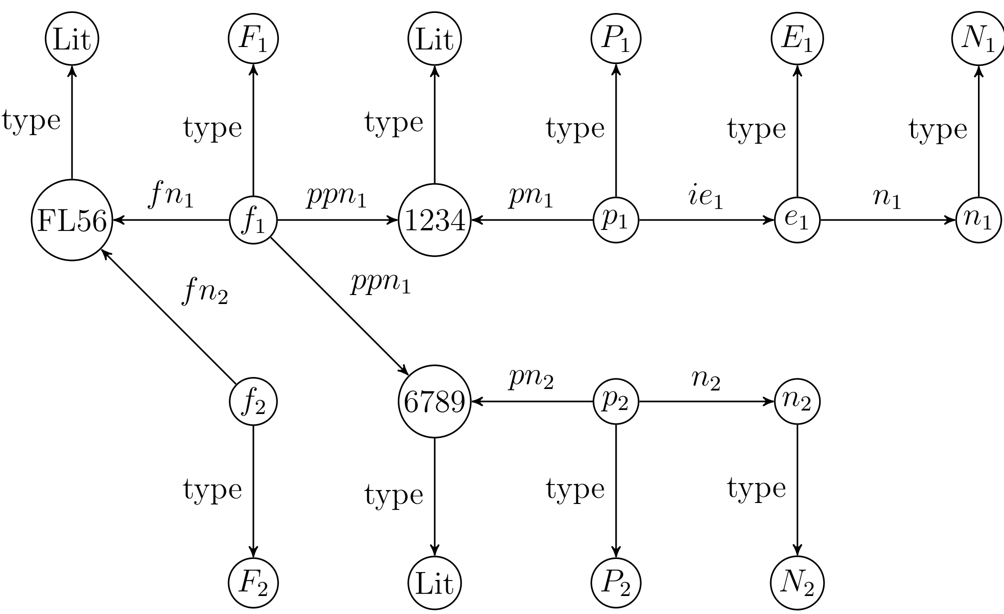

Figure 3.1: Example data graph G showing flight insurance data.

ontology K which we will henceforth use in examples in the rest of this chapter. Suppose two flight insurance companies agree to share some of their data — which is heterogeneous — under a common ontology. Figure 3.1 shows part of the merged data graph G that might arise, where F denotes $Flight$, P denotes $Person$, E denotes $Employee$ and N denotes $NationalInsuranceNumber$ (which is used in the administration of the UK National Social Security system). For simplicity here, we assume that common concepts have the same name in both the datasets and that the subscripts on the node and edge labels indicate the dataset from which each was derived (dataset 1 or dataset 2). F_1 and F_2 are classes, representing the $Flight$ class in each of the datasets. Similarly, P_1 and P_2 represent the $Person$ class; E_1 represents the $Employee$ class — which is present only in dataset 1; and N_1 and N_2 represent the class $NationalInsuranceNumber$. Of the various edge labels in Figure 3.1, fn denotes $flightNumber$, ppn denotes $passengerPassportNumber$, pn denotes $passportNumber$, ie denotes $isEmployee$ and n denotes $hasNationalInsuranceNumber$. We see from Figure 3.1 that there is some overlap in the datasets, through the literals 'FL56' and '6789' ('Lit' denotes a 'Literal'). The first of these literals is a flight number while the second is a passport number.

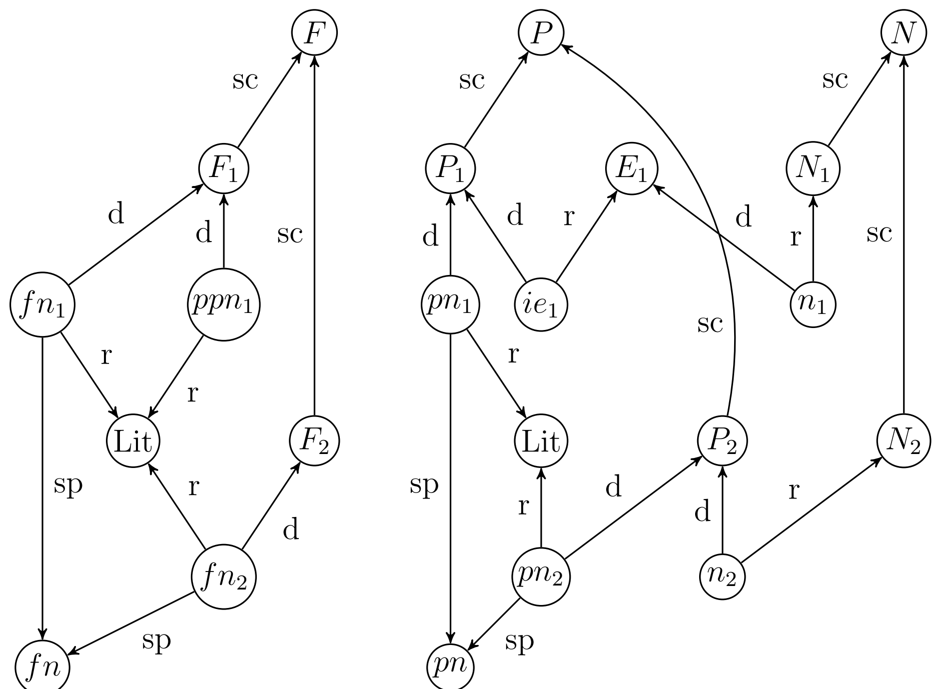

Figure 3.2: Example ontology K from the flight insurance domain.

The ontology K is given in Figure 3.2, which shows additional classes, properties, and relationships between them, that serve to integrate semantically the two datasets. In this figure, `d` denotes `dom`, `r` denotes `range`, with all other symbols having the same denotations as those given for Figure 3.1.

□

3.2 The query language

Our query language is that of *conjunctive regular path queries* (CRPQs) [17]. A CRPQ over a graph G is of the form:

$$(Z_1, \ldots, Z_m) \leftarrow (X_1, R_1, Y_1), \ldots, (X_n, R_n, Y_n) \tag{3.1}$$

where each X_i and Y_i, $1 \leq i \leq n$, is a variable or constant, each Z_i, $1 \leq i \leq m$, is a variable appearing in the body of the query, and each R_i, $1 \leq i \leq n$, is a regular expression over the alphabet of edge labels. We define regular expressions as used in this thesis in Section 3.3.

Given a CRPQ Q and graph G, let θ be a mapping from $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ to V_G such that (i) each constant is mapped to itself, and (ii) for each conjunct (X_i, R_i, Y_i), $1 \leq i \leq n$, there is a path from $\theta(X_i)$ to $\theta(Y_i)$ in G whose sequence of edge labels is in the language denoted by R_i. Let Θ be the set of such mappings. Then the (exact) answer of Q on G is $\{\theta(Z_1, \ldots, Z_m) \mid \theta \in \Theta\}$.

Let $G = (V_G, E_G, \Sigma)$ be a graph as defined in Section 3.1. We support the notion of a *semipath* [17] by allowing each edge $e = (x, l, y) \in E_G$ to be traversed both from its source x to its target y and from its target y to its source x. In order to specify the traversal from target to source, it is useful to define the *inverse* of an edge label l, denoted by l^-. Let $\Sigma^- = \{l^- \mid l \in \Sigma\}$. If $l \in \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}$, we use l^- to mean the *inverse* of l, that is, if l is a for some $a \in \Sigma \cup \{\mathtt{type}\}$, then l^- is a^-, while if l is a^- for some $a \in \Sigma \cup \{\mathtt{type}\}$, then l^- is a.

We note that our language is orthogonal to SPARQL 1.0, and is subsumed by SPARQL 1.1.

Example 3.2. (Drawn from [114]). Referring again to Figures 3.1 and 3.2, a user familiar with the first dataset may pose the following CRPQ, Q_1, in an attempt to find the passport numbers of passengers on flight number 'FL56':

$$Y \leftarrow (\text{`FL56'}, fn_1, Y), (Y, pn_1^-.type, P_1)$$

This query however returns no answers because of errors in the first conjunct $(\text{`FL56'}, fn_1, Y)$. The edge label ought to be fn_1^- instead of fn_1, indicating an incoming, rather than an outgoing, relationship from the node denoting the flight number to the adjacent node. Moreover, the edge label ppn_1 is missing. Using the query approximation techniques that we explore in this thesis, the user may pose instead the following query, Q_2, which allows the first conjunct to be approximated by including the operator APPROX:

$$Y \leftarrow APPROX(\text{`FL56'}, fn_1, Y), (Y, pn_1^-.type, P_1)$$

By replacing fn_1 by fn_1^- and inserting ppn_1 after fn_1^-, the result '1234' can now be returned (at a cost of 2α, say, assuming a cost α for each of the edit operations).

□

Example 3.3. (Drawn from [114]). Continuing with the query in the previous example, suppose the user now poses the following query, Q_3, which allows the first conjunct to be approximated and the second conjunct to be relaxed, using the operator RELAX that we explore in this thesis:

$$Y \leftarrow APPROX(\text{`FL56'}, fn_1, Y), RELAX(Y, pn_1^-.type, P_1)$$

By relaxing property pn_1 to its superproperty pn and class P_1 to its superclass P — there is a 'subproperty' edge between pn_1 and pn, and a 'subclass' edge between P_1 and P in Figure 3.2 — at a cost of 2β, say, assuming a cost β for each relaxation operation, an additional relevant result '6789' is returned at an overall cost of $2(\alpha + \beta)$.

□

3.3 Single-conjunct queries

Definition 3.1. A *single-conjunct regular path query* Q over a graph $G = (V_G, E_G, \Sigma)$ has the form:

$$vars \leftarrow (X, R, Y) \tag{3.2}$$

where X and Y are constants or variables, R is a regular expression over $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$ (see Definition 3.2 below), and $vars$ is the subset of $\{X, Y\}$ that are variables. If X or Y is a constant, then that constant must appear in V_G if Q is to return a non-empty answer on G (see Definition 3.3 below).

□

Definition 3.2. A *regular expression* R over $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$ is defined as follows:

$$R := \epsilon \mid a \mid a^- \mid _ \mid (R1 \cdot R2) \mid (R1|R2) \mid R^* \mid R^+$$

where ϵ is the empty sequence, a is any label in $\Sigma \cup \{\texttt{type}\}$ and "_" denotes the disjunction of all constants in $\Sigma \cup \{\texttt{type}\}$. "$(R1 \cdot R2)$" denotes that $R1$ is concatenated with $R2$, "$(R1|R2)$" denotes the occurrence of either $R1$ or $R2$, "R^*" denotes zero or more occurrences of R and "R^+" denotes at least one occurrence of R.

□

Definition 3.3. A *semipath* [17] p in $G = (V_G, E_G, \Sigma)$ from $x \in V_G$ to $y \in V_G$ is a sequence $(v_1, l_1, v_2, l_2, v_3, \ldots, v_n, l_n, v_{n+1})$, where $n \geq 0$, $v_1 = x$, $v_{n+1} = y$ and for each v_i, l_i, v_{i+1} either $(v_i, l_i, v_{i+1}) \in E_G$ or $(v_{i+1}, l_i^-, v_i) \in E_G$. A semipath p *conforms* to a regular expression R if the sequence of labels $l_1 \cdots l_n$ is in the language recognised by R, $L(R)$.

Given a single-conjunct regular path query Q and graph G, let θ be a mapping from $\{X, Y\}$ to V_G that maps each constant to itself. We term a mapping such as θ a (Q, G)*-matching*. A tuple $\theta(vars)$ *satisfies* Q on G if there is a semipath in G from $\theta(X)$ to $\theta(Y)$ which conforms to R. The *exact answer* of Q on G is the set of tuples which satisfy Q on G.

□

The following result on the complexity of exact query answering follows from Lemma 1 in [100].

Proposition 3.1. *Given single-conjunct regular path query Q and graph G, the exact answer of Q on G can be found in time which is polynomial in the size of Q and G.*

We now define the *triple form* of both a semipath in a graph and a sequence of labels in $L(R)$. The definition uses the notion of a *triple pattern*, which is a triple each of whose components may be either a constant or a variable. We use triple forms as a uniform syntax to which we can apply approximation and relaxation.

Definition 3.4. Let p be a semipath $(v_1, l_1, v_2, l_2, v_3, \ldots, v_n, l_n, v_{n+1})$, $n \geq 1$, in G. A *triple form* of p is a sequence of triple patterns

$$(v_1, l_1, W_1), (W_1, l_2, W_2), \ldots, (W_{n-1}, l_n, v_{n+1})$$

where $W_1, \ldots, W_{n-1}$ are distinct variables. If p is of length zero, i.e. $n = 0$, then we define p to be (v, ϵ, v) and hence the only triple form of p is (v, ϵ, v).

□

Definition 3.5. Given a query Q with single conjunct (X, R, Y), let $q = l_1 l_2 \cdots l_n$, $n \geq 1$, be a sequence of labels in $L(R)$. A *triple form* of (Q, q) is a sequence of triple

patterns

$$(X, l_1, W_1), (W_1, l_2, W_2), \ldots, (W_{n-1}, l_n, Y)$$

where $W_1, \ldots, W_{n-1}$ are distinct variables not appearing in Q. If $q = \epsilon$, then the triple form of (Q, q) is (X, ϵ, Y).

□

Definition 3.6. Let T be a sequence of triple patterns

$$(W_0, l_1, W_1), (W_1, l_2, W_2), \ldots, (W_{n-1}, l_n, W_n)$$

such that $n \geq 1$, $W_1, \ldots, W_{n-1}$ are variables, and W_0 and W_n are variables or constants. Any triple pattern (W_{i-1}, l_i, W_i) in which $l_i \in \Sigma^- \cup \{\texttt{type}^-\}$ is said to be *inverted*; otherwise the triple pattern is *non-inverted*. The *normalised form* of an inverted triple pattern (W_{i-1}, l_i, W_i) is (W_i, l_i^-, W_{i-1}), while the normalised form of a non-inverted triple pattern is the triple pattern itself. The *normalised form* of T comprises the normalised form of each triple pattern in T.

□

Example 3.4. (Drawn from [114]). Assume we have G as described in Example 3.1 and shown in Figure 3.1, and a query Q comprising the single conjunct ('FL56', $fn_1^- \cdot ppn_1, X$), where 'FL56' is a constant, X and W_1 are distinct variables, and $q = fn_1^- \cdot ppn_1$. The following is a triple form of (Q, q):

$$(\text{'FL56'}, fn_1^-, W_1), (W_1, ppn_1, X)$$

and its normalised form is:

$$(W_1, fn_1, \text{'FL56'}), (W_1, ppn_1, X)$$

□

3.4 Query approximation

In Section 3.4.1, we provide formal definitions for approximate matching of single-conjunct queries as performed by the APPROX operator. In Section 3.4.2 we describe how approximate answers can be returned to the user incrementally.

3.4.1 Approximate matching of single-conjunct queries

Approximate matching of a single-conjunct query Q against a graph G is achieved by applying edit operations to a sequence of labels in $L(R)$, where R is the regular expression used in Q. Let q be a sequence of labels in $L(R)$ and l be an arbitrary label in $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$.

An *edit* operation on q is one of the following:

(i) the *insertion* of label l into q,

(ii) the *deletion* of label l from q,

(iii) the *substitution* of some label other than l by l in q,

Each edit operation has a cost, which is a positive integer c_i, c_d, c_s, respectively; the costs may be different for different edit operations. We assume throughout that the cost of *substitution* is less than the combined cost of *insertion* and *deletion* (otherwise the substitution operation would be redundant, as it would be no more costly to achieve such an edit through an insertion and a deletion operation).

In [67], two additional edit operations were defined: (i) the *inversion* of a label in q and (ii) the *transposition* of a pair of adjacent labels in q. We note that both these operations can be subsumed semantically by the edit operations given above; *inversion* is subsumed by *substitution* in which some label $l \in \Sigma$ is replaced by l^-, and *transposition* is achieved by either applying *substitution* to the pair of labels to be transposed or by a combination of insertion and deletion of labels.

Definition 3.7. The *application of an edit operation to a sequence of triple patterns* T of the form

$$(X, l_1, W_1), (W_1, l_2, W_2), \ldots, (W_{n-1}, l_n, Y),$$

where $n \geq 1$, X and Y are variables or constants, and $W_1, \ldots, W_{n-1}$ are distinct new variables, is defined as follows:

The result of a *substitution* on T is a sequence of triple patterns

$$(X, l'_1, W_1), (W_1, l'_2, W_2), \ldots, (W_{n-1}, l'_n, Y)$$

such that there must be some $1 \leq j \leq n$ where $l_j \neq l'_j$ (l'_j has been substituted for l_j) and $l_i = l'_i$, for each $i \neq j$, $1 \leq i \leq n$.

The result of an *insertion* into T is a sequence of triple patterns

$$(X, l'_1, W_1), (W_1, l'_2, W_2), \ldots, (W_n, l'_{n+1}, Y)$$

such that there is a $1 \leq j \leq n+1$ for which $l_i = l'_i$, for each $1 \leq i \leq j-1$, and $l_i = l'_{i+1}$, for each $j+1 \leq i \leq n$ (l'_j is the inserted label).

If $n > 1$, the result of a *deletion* from T is a sequence of triple patterns

$$(X, l'_1, W_1), (W_1, l'_2, W_2), \ldots, (W_{n-2}, l'_{n-1}, Y)$$

such that there is a $1 \leq j \leq n$ for which $l_i = l'_i$, for each $1 \leq i \leq j-1$, and $l_{i+1} = l'_i$, for each $j \leq i \leq n-1$ (l_j is the deleted label). If $n = 1$, the result of a deletion from T is (X, ϵ, Y) (where l_1 is the deleted label).

If T is of the form (X, ϵ, Y), then only the insertion operation applies, the result of which is (X, l', Y), where l' is the inserted label.

□

Definition 3.8. Given graph G, semipath p in G, query Q with single conjunct (X, R, Y), (Q, G)-matching θ, sequence of labels $q \in L(R)$, triple form T_q for $(\theta(Q), q)$, and triple form T_p for p:

- we write $T_q \preceq_A T_p$, if T_q can be transformed to T_p (up to variable renaming) by a sequence of edit operations. The *cost* of the sequence of edit operations on T_q is the sum of the costs of each operation;
- the *approximation distance* from p to $(\theta(Q), q)$ is the minimum cost of any sequence of edit operations which yields T_p from T_q. The cost of the empty sequence of edit operations (so T_q is already a triple form of p) is zero. If T_q cannot be transformed to T_p, then the approximation distance is infinity;

- the *approximation distance* from p to $\theta(Q)$ is the minimum approximation distance from p to $(\theta(Q), q)$ for any sequence of labels $q \in L(R)$;
- the *approximation distance* of $\theta(Q)$, denoted $adist(\theta, Q)$, is the minimum approximation distance to $\theta(Q)$ from any semipath p in G;
- the *approximate answer* of Q on G, denoted $Q_A(G)$, is a list of pairs $(\theta(vars), adist(\theta, Q))$, where θ is a (Q, G)-matching, ranked in order of non-decreasing approximation distance;
- the *approximate top-k answer* of Q on G is a list containing the first k tuples in $Q_A(G)$.

□

In principle, we note that the edit operations could be applied *ad infinitum*. However, the semantics defined mean that answers at a minimum cost will be returned.

Example 3.5. Consider the data graph G as described in Example 3.1 and shown in Figure 3.1, and an approximated query Q:

$$X \leftarrow APPROX(\text{‘FL56’}, fn_1 \cdot pn_1^-, X)$$

where ‘FL56’ is a constant and X is a variable. A normalised triple form T of (Q, q), where $q = fn_1 \cdot pn_1^-$, is:

$$(\text{‘FL56’}, fn_1, W_1), (X, pn_1, W_1)$$

The result of substituting fn_1 by fn_1^- is the normalised triple form T':

$$(W_1, fn_1, \text{‘FL56’}), (X, pn_1, W_1)$$

The result of inserting ppn_1 between fn_1^- and pn_1^- is the normalised triple form T'':

$$(W_1, fn_1, \text{‘FL56’}), (W_1, ppn_1, W_2), (X, pn_1, W_2)$$

Thus, the answer 'p_1' would be returned at a cost of $c_s + c_i$, where c_s and c_i are the costs of the substitution and insertion operations, respectively.

□

We now describe how $Q_A(G)$ can be computed in time polynomial in the size of Q and G. A similar process was described in [67], but that paper included only sketch proofs of the theoretical results. Broadly, the steps are as follows: (1) construct a weighted *query automaton* M_Q recognising the language denoted by R using Thompson's construction (which makes use of ϵ-transitions) [2], (2) construct an *approximate automaton* A_Q, (3) construct the *product automaton* H of A_Q and G, and (4) perform shortest path traversals of H in order to find the approximate answer of Q on G. Each of the terms introduced above are defined next.

Definition 3.9. A *weighted non-deterministic finite state automaton* (NFA) M is a tuple $(S, A, \delta, S_0, S_f, \xi)$, where: S is a set of states; A is an alphabet of labels; $\delta \subseteq S \times A \times \mathbb{N} \times S$ is the transition relation; $S_0 \subseteq S$ is the set of start states; $S_f \subseteq S$ is the set of final states; and ξ is a final weight function mapping each state in S_f to a non-negative integer [32]. Given a transition $(s, a, c, t) \in \delta$, we sometimes say that the transition is *from* s *to* t and call a the label and c the cost of the transition.

We call a sequence of transitions from an initial to a final state of M a *run*. Given a sequence of labels p, a *run for* p is a sequence of transitions of the form $(s_1, a_1, w_1, s_2), \ldots, (s_{n-1}, a_{n-1}, w_{n-1}, s_n)$, where s_1 is an initial state, s_n is a final state, and $p = a_1 \cdots a_{n-1}$. The *cost* of the run is $w_1 + \cdots + w_{n-1} + \xi(s_n)$. We sometimes say that the run is *from* s_1 *to* s_n.

□

Definition 3.10. Let R be a regular expression defined over alphabet $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$. A *weighted non-deterministic finite state automaton* (NFA) M_R recognising $L(R)$ can be constructed in the same way as a normal NFA recognising $L(R)$, except that each transition and each final state has a zero weight associated with it. Formally, $M_R = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, \{s_0\}, \{s_f\}, \xi)$, where there is only one initial state s_0 and one final state s_f, and ξ maps s_f to zero.

Let Q be a single-conjunct query with conjunct (X, R, Y). The *query automaton* M_Q for Q is the same as M_R but with annotations on the initial and final states. In particular, if X (or, respectively, Y) in Q is a constant c, then s_0 (s_f) is annotated

with c; otherwise s_0 (s_f) is annotated with the wildcard symbol $*$ which matches any constant.

□

Definition 3.11. Let Q be a single-conjunct query with conjunct (X, R, Y), and $M_R = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, \{s_0\}, \{s_f\}, \xi)$ be the weighted NFA for R. We construct the *approximate automaton* A_Q *for* Q by first constructing an *approximate automaton* A_R *from* M_R. The approximate automaton A_R is constructed in a number of steps:

- First the automaton A_R^1 with *deletions* is constructed from M_R. Automaton A_R^1 is the same as M_R except that the set of transitions δ' includes all those in δ along with the set $\{(s, \epsilon, c_d, t) \mid (s, a, 0, t) \in \delta \wedge s \neq t\}$, where c_d is the cost of deletion.

- Next an automaton A_R^2 without ϵ-transitions is constructed from A_R^1, using the method of [32]. Briefly, this method first computes the ϵ*-closure* of A_R^1, which is the set of pairs of states connected by a sequence of ϵ-transitions along with the minimum summed weight for each such pair. Then $A_R^2 = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta'', \{s_0\}, S, \xi')$, where the transitions in δ'' comprise the non-epsilon transitions in δ' along with each transition (s, b, w, u) such that pair (s, t) with weight w is in the ϵ-closure and transition $(t, b, 0, u) \in \delta'$ $(b \neq \epsilon)$. Because every transition in δ' of automaton A_R^1 between two distinct states has an associated ϵ-transition, all states will be final. The final state function ξ' is defined as follows. For final state s_f, $\xi'(s_f) = \xi(s_f) = 0$. For each state $s \neq s_f$, $\xi'(s)$ is the minimum weight of the pairs (s, s_f) in the ϵ-closure.

- Thirdly, an automaton A_R^3 with *substitutions* is constructed from A_R^2. Automaton A_R^3 is the same as A_R^2 except that the transitions of A_R^3 comprise those of A_R^2 along with transitions of the form $(s, b, w + c_s, t)$, where c_s is the cost of *substitution*, for each transition $(s, a, w, t) \in \delta''$, such that $s \neq t$, and label $b \in \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$ $(b \neq a)$.

- Finally, the approximate automaton A_R is constructed from A_R^3 by including *insertions*. Automaton A_R is the same as A_R^3 except that the transitions of A_R

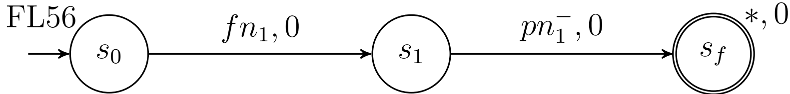

Figure 3.3: Query automaton M_Q for conjunct ('FL56', $fn_1 \cdot pn_1^-, X$).

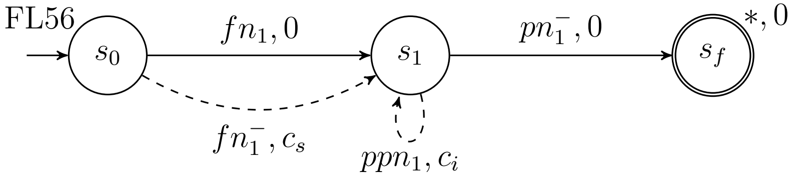

Figure 3.4: Fragment of approximate automaton A_Q for conjunct ('FL56', $fn_1 \cdot pn_1^-, X$).

comprise those of A_R^3 along with transitions of the form (s, a, c_i, s), where c_i is the cost of insertion, for each state $s \in S$ and label $a \in \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$.

The *approximate automaton* A_Q *for* Q is formed from A_R by annotating the initial and final states in A_R with the annotations from the initial and final states, respectively, in M_Q.

□

Example 3.6. (Drawn from [114]). Consider once again the data graph G shown in Figure 3.1, and the following approximated query Q:

$$X \leftarrow APPROX(\text{'FL56'}, fn_1 \cdot pn_1^-, X)$$

The query automaton M_Q for Q is shown in Figure 3.3. We see that M_Q is comprised of two transitions, each labelled with a cost of zero; the initial state, s_0, is annotated with the constant 'FL56'; and the final state, s_f, is annotated with the wildcard symbol $*$, and has a weight of zero.

A fragment of the approximate automaton A_Q for Q is shown in Figure 3.4. Two transitions — represented by the dashed lines — appear in A_Q but not in M_Q. The transition (s_0, fn_1^-, c_s, s_1) indicates that the label fn_1 has been substituted by fn_1^-, and the transition (s_1, ppn_1, c_i, s_1) indicates that the label ppn_1 has been inserted before pn_1^-; c_s and c_i denote the cost of these edit operations, respectively.

□

In Chapter 4, Lemma 4.1 shows that using automaton A_R is sufficient to find all sequences of labels generated by edit operations at an arbitrary approximation distance from a given query.

Definition 3.12. Let $A_Q = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, \{s_0\}, S, \xi)$ be an approximate automaton and $G = (V_G, E_G, \Sigma)$ a graph. We can view G as an automaton with set of states V_G, alphabet Σ, set of initial states V_G, and set of final states V_G. There is a transition from state s to state t labelled a in the automaton if and only if there is an edge $(s, a, t) \in E_G$. We can then form the *product automaton*, H, of A_Q and G. Formally, H is the weighted automaton $(T, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \sigma, I, T, \xi)$, where $I \subseteq T$ is a set of initial states and all states in T are final. The set of states T is given by $\{(s, n) \mid s \in S \wedge n \in V_G\}$. The set of transitions σ consists of transitions of the form

- $((s, n), a, c, (s', n'))$ if $(s, a, c, s') \in \delta$ and $(n, a, n') \in E_G$,
- $((s, n), a^-, c, (s', n'))$ if $(s, a^-, c, s') \in \delta$ and $(n', a, n) \in E_G$.

The set of initial states I is given by $\{(s_0, n) \mid n \in V_G\}$. We overload the use of ξ as the final weight function, carrying over the weights from final states in A_Q to those in H. The annotations on initial and final states in H are also carried over from the corresponding initial and final states in A_Q.

□

H can be viewed either as an automaton or as a graph, whichever is appropriate in a given context. When H is viewed as an automaton, we will use the terms *states*, *transitions* and *runs*; when viewed as a graph, we will use the terms *nodes*, *edges* and *paths*.

We can now define how $Q_A(G)$, the approximate answer of Q on G, can be computed:

(i) We construct the weighted NFA M_R from R, using Thompson's construction [2], and then the query automaton M_Q from M_R.

(ii) We construct the approximate automaton A_Q from M_Q.

(iii) We form the product automaton, H, of A_Q with G.

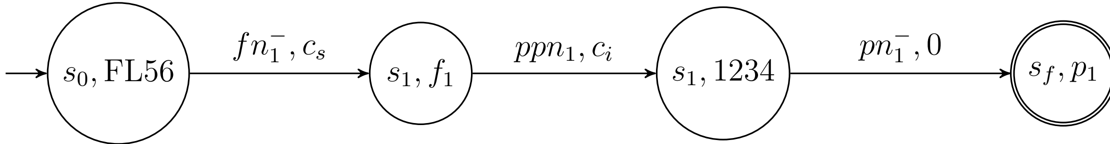

Figure 3.5: A subautomaton of the product automaton H of A_Q and G.

(iv) Let the conjunct of Q be (X, R, Y). Assume first that X is a constant u. Assume also that $u \in V_G$, for otherwise $Q_A(G)$ is empty. We perform a shortest path traversal of H starting from the initial state (s_0, u), incrementing the total cost of the path by the cost of each transition. Whenever we reach a final state (s_f, v) in H we output v, provided v *matches* the annotation on (s_f, v), along with the cost of the path. Recall that if Y is a constant the annotation on s_f will be that constant, and if Y is a variable the annotation will be the wildcard symbol $*$. Node v matches the annotation if and only if the annotation is v or $*$. Now assume X is a variable. In this case, we perform a shortest path traversal of H, outputting nodes as above, starting from state (s_0, u) for each node $u \in V_G$.

Example 3.7. Consider once again the data graph G shown in Figure 3.1, the approximated query Q and the approximate automaton A_Q from Example 3.6:

$$X \leftarrow APPROX(\text{‘FL56’}, fn_1 \cdot pn_1^-, X)$$

A subautomaton of the product automaton H of A_Q (shown in Figure 3.4) and G is shown in Figure 3.5.

For a matching θ that maps X to ‘p_1’, it is easy to see that there is one run in H from the initial state $(s_0, \mathrm{FL56})$ to the final state (s_f, p_1), with a cost of $c_s + c_i$. Hence the approximation distance of the answer ‘p_1’ is $c_s + c_i$.

□

In Chapter 4, Lemma 4.2 shows that if there is a semipath from v_0 to v_n in G, the cost of a traversal from (s_0, v_0) to (s_f, v_n) in the product automaton H corresponds to the cost of a run from s_0 to s_f in A_Q. Lemma 4.3 shows that the approximation distance from a semipath in a graph G to the matchings for a single-conjunct query Q is equal to the minimum cost of a corresponding run in H.

3.4.2 Incremental evaluation of APPROX conjuncts

The evaluation of APPROX single-conjunct queries can be accomplished 'on-demand' by incrementally constructing the edges of H as required, thus avoiding precomputation and materialisation of the entire product automaton H. This is performed by calling a function `Succ` with a node (s, n) of H. The function returns a set of transitions $\xrightarrow{d} (p, m)$, such that there is an edge in H from (s, n) to (p, m) with cost d.

We list function `Succ` below, where the function $\texttt{NextStates}(A_Q, s, a)$ returns the set of states in A_Q that can be reached from state s on reading input a, along with the cost of reaching each (the function `NextStates` is defined in Section 6.6.2).

Function Succ(s, n, A_Q, G)

Input: state s of A_Q and node n of G
Output: set of transitions which are successors of (s, n) in H

(1) $W \leftarrow \emptyset$
(2) **for** $(n, a, m) \in E_G$ *and* $(p, d) \in \texttt{NextStates}(A_Q, s, a)$ **do**
(3) add the transition $\xrightarrow{d} (p, m)$ to W
(4) **return** W

Lemma 3.1. *The transition* $\xrightarrow{d} (p, m)$ *is returned by* $\texttt{Succ}(s, n, A_Q, G)$ *iff* $(s, n) \xrightarrow{a,d} (p, m)$ *is in* $H = A_Q \times G$ *for some* $a \in (\Sigma \cup \Sigma^- \cup \{\texttt{type} \cup \texttt{type}^-\})$.

Proof. By the definition of `Succ`, the transition $\xrightarrow{d} (p, m)$ is added to W if and only if $a \in (\Sigma \cup \Sigma^- \cup \{\texttt{type} \cup \texttt{type}^-\})$, $(n, a, m) \in E_G$ and $(p, d) \in \texttt{NextStates}(A_Q, s, a)$. By the definition of H, the presence of an edge labelled a from n to m in G and of a transition labelled a from s to p in A_Q results in an edge $(s, n) \xrightarrow{a,d} (p, m)$ in H.

□

For incremental evaluation, a global set $\texttt{visited}_R$ is maintained, storing tuples of the form (v, n, s), representing the fact that node n of G was visited in state s of A_Q having started the traversal from node v. Also maintained is a global priority queue $\texttt{queue}_R$ containing tuples of the form (v, n, s, d, f), ordered by increasing values of d, where d is the approximation distance associated with visiting node n in state s having started from node v, and f is a flag denoting whether the tuple is final

or non-final, with the latter being the initial value for f. A 'final' tuple is one corresponding to a completed run in the product automaton H and thus will, when dequeued, be an answer, whereas a 'non-final' tuple is still in the process of being evaluated.

Recalling that Q has the form (X, R, Y), we begin by enqueueing the initial tuple $(v, v, s_0, 0, f)$, if X is some node v, or enqueueing a set of initial tuples otherwise, one for each node v of G. We maintain a global list $\texttt{answers}_R$ containing tuples of the form (v, n, d), where d is the smallest approximation distance of this answer tuple to Q and ordered by non-decreasing value of d. This list is used to avoid returning as an answer (v, n, d') for any $d' \geq d$. It is initialised to the empty list.

We then call function `GetNext` shown below to return the next query answer, in order of non-decreasing approximation distance from Q. We see that `GetNext` repeatedly dequeues the first tuple of $\texttt{queue}_R$, (v, n, s, d, f), adding (v, n, s) to $\texttt{visited}_R$ if the tuple is not final, until $\texttt{queue}_R$ is empty.

After dequeueing a tuple (v, n, s, d, f), we check to see whether the tuple is a final one; if not, we enqueue $(v, m, s', d + d', f)$ for each transition $\xrightarrow{d'} (s', m)$ returned by $\texttt{Succ}(s, n, A_Q, G)$ such that $(v, m, s') \notin \texttt{visited}_R$. If s is a final state, its annotation matches n, and the answer (v, n, d') has not been generated before for some d', then we add the final weight function for s to d, mark the tuple as final, and enqueue the tuple.

On the other hand, if a dequeued tuple is a final one and the answer (v, n, d') has not been generated before for some d', the triple (v, n, d) is returned after being added to $\texttt{answers}_R$.

We note that the maximum size of each of $\texttt{visited}_R$ and $\texttt{queue}_R$ is $2|R||V_G|^2$, and that the size of $\texttt{answers}_R$ will never exceed $|V_G|^2$. For $\texttt{queue}_R$ this result follows from the fact that, as in Dijkstra's shortest path algorithm, we assume that, for each combination of nodes v and n and state s, at most one tuple (v, n, s, d, f) is enqueued by using the priority queue's 'decrease key' operation.

Theorem 4.1 in Chapter 4 shows that our incremental evaluation algorithm, `GetNext`, is correct: that is, given a single-conjunct query Q and a graph G, it returns the approximate answer of Q on G.

Function GetNext(X, R, Y, A_Q, G)

Input: query conjunct (X, R, Y)
Output: triple (v, n, d), where v and n are instantiations of X and Y

```
(1)  while nonempty(queue_R) do
(2)      (v, n, s, d, f) ← dequeue(queue_R)
(3)      if f ≠ 'final' then
(4)          add (v, n, s) to visited_R
(5)          foreach -d'-> (s', m) ∈ Succ(s, n, A_Q, G) s.t. (v, m, s') ∉ visited_R do
(6)              enqueue(queue_R, (v, m, s', d + d', f))
(7)          if s is a final state s_f and its annotation matches n and
                 ∄d'.(v, n, d') ∈ answers_R then
(8)              enqueue(queue_R, (v, n, s, d + ξ[s], 'final'))
(9)      else
(10)         if ∄d'.(v, n, d') ∈ answers_R then
(11)             append (v, n, d) to answers_R
(12)             return (v, n, d)
(13) return null
```

3.5 Query relaxation

In Section 3.5.1, we provide formal definitions for relaxation of single-conjunct queries based on information from an ontology as performed by the RELAX operator. In Section 3.5.2, we describe how relaxed answers can be returned to the user incrementally.

3.5.1 Ontology-based relaxation of single-conjunct queries

The work in [66] considered ontology-based relaxation of conjunctive queries in the setting of the RDF/S data model (but not conjunctive *regular path* queries) and showed that query relaxation can be naturally formalised using *RDFS entailment*. The entailment was characterised by the derivation rules given in Figure 3.6, grounded in the semantics developed in [52, 56]. The work in [115] extended ontology-based relaxation to CRPQs, using an automaton-based approach. Here, we revisit the work of [115], giving full details and formally proving the correctness of the construction and traversal of the product automaton H of the relaxed

$$\text{(Subproperty)} \quad (1)\ \frac{(a,\texttt{sp},b)\ (b,\texttt{sp},c)}{(a,\texttt{sp},c)} \qquad (2)\ \frac{(a,\texttt{sp},b)\ (X,a,Y)}{(X,b,Y)}$$

$$\text{(Subclass)} \quad (3)\ \frac{(a,\texttt{sc},b)\ (b,\texttt{sc},c)}{(a,\texttt{sc},c)} \qquad (4)\ \frac{(a,\texttt{sc},b)\ (X,\texttt{type},a)}{(X,\texttt{type},b)}$$

$$\text{(Typing)} \quad (5)\ \frac{(a,\texttt{dom},c)\ (X,a,Y)}{(X,\texttt{type},c)} \qquad (6)\ \frac{(a,\texttt{range},c)\ (X,a,Y)}{(Y,\texttt{type},c)}$$

Figure 3.6: RDFS Inference Rules.

automaton and the data graph.

For RDF/S graphs G_1 and G_2, [66] states that $G_1 \models_{\texttt{rule}} G_2$ if G_2 can be derived from G_1 by iteratively applying the rules of Figure 3.6. The *closure* [56] of an RDF/S graph G under these rules is denoted by $\text{cl}(G)$.

In the formalisation of RDF [56], infinite sets I of IRIs and L of RDF literals are assumed. The elements in $I \cup L$ are called RDF *terms*. Given a set of variables V disjoint from I and L, a *triple pattern* is a triple $(v_1, v_2, v_3) \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)$.

As described in Section 3.1, in this thesis we assume that the data graph G and the ontology K are separate graphs, such that the nodes representing classes in V_G also appear as nodes in V_K. We also assume that query evaluation takes place on the graph given by restricting $\text{cl}(G \cup K)$ to the nodes of $V_G \cup V_K$ and the edges labelled with labels from $\Sigma \cup \{\texttt{type}\} \cup V_{Prop}$. We call this the *closure of the data graph G with respect to the ontology K* and denote it by $closure_K(G)$. For example, the closure of the data graph of Figure 3.1 with respect to the ontology in Figure 3.2 is illustrated in Figure 3.7, where F denotes $Flight$, P denotes $Person$, E denotes $Employee$, N denotes $NationalInsurancenumber$, fn denotes $flightNumber$, ppn denotes $passengerPassportNumber$, pn denotes $passportNumber$, ie denotes $isEmployee$ and n denotes $nationalInsuranceNumber$.

Terminology Note: We use the term 'closure of the data graph G' to mean 'closure of the data graph G with respect to the ontology K' if the ontology K can be inferred from the context.

As in [115], we assume that the subgraphs of the ontology K induced by edges labelled `sc` and `sp` are acyclic, and that K is equal to its *extended reduction* [66].

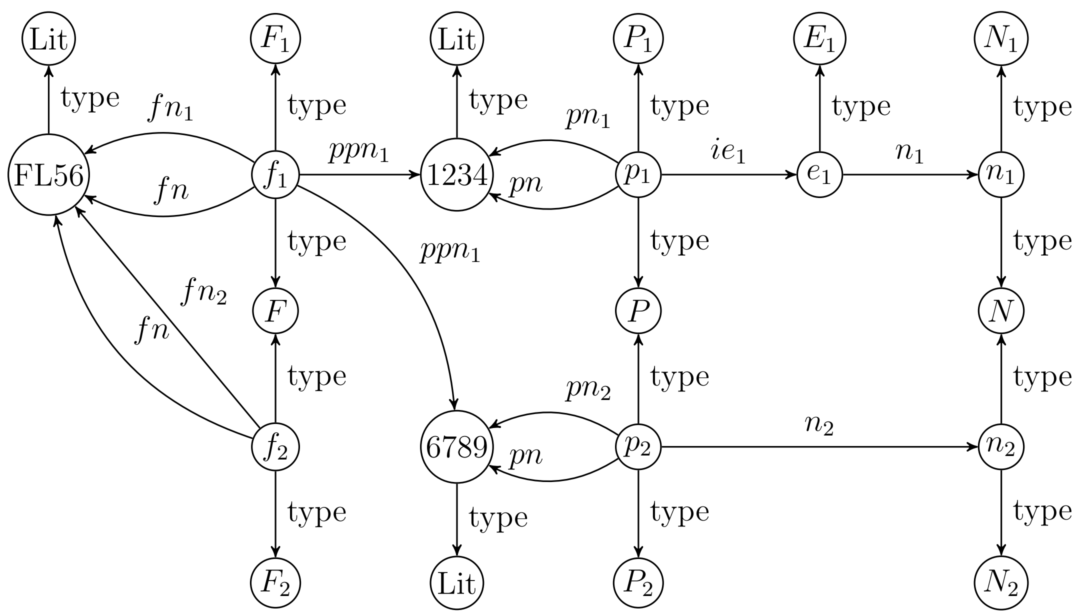

Figure 3.7: Closure of graph G in Figure 3.1 with respect to the ontology K in Figure 3.2.

These restrictions are necessary for associating an unambiguous cost with queries, so that query answers can be returned to users in order of increasing cost (see more details below).

The extended reduction of an ontology K is denoted by `extRed`(K) and is computed as follows:

(i) compute $\text{cl}(K)$;

(ii) apply the rules of Figure 3.8 in reverse until no more rules can be applied (applying a rule 'in reverse' means deleting a triple that can be deduced by the rule);

(iii) apply rules 1 and 3 of Figure 3.6 in reverse until no more rules can be applied.[1]

Using the extended reduction allows us to perform what were termed *direct* relaxations in [66], which correspond to the "smallest" relaxation steps. This is

[1] In order to generate a unique extended reduction, we alter the procedure in [66] as described in [114]: let D be the set of triples in $\text{cl}(K)$ that can be derived using rules (1) or (3) in Figure 3.6, or rules $(e1)$, $(e2)$, $(e3)$ or $(e4)$ in Figure 3.8; then `extRed`(K) is given by $\text{cl}(K) - D$. Because $\text{cl}(K)$ is closed with respect to the edge labels `sp` and `sc`, and also the subgraphs induced by each of `sp` and `sc` are acyclic, the set D is uniquely defined.

$$(e1)\ \frac{(b, \texttt{dom}, c)\ (a, \texttt{sp}, b)}{(a, \texttt{dom}, c)} \qquad (e2)\ \frac{(b, \texttt{range}, c)\ (a, \texttt{sp}, b)}{(a, \texttt{range}, c)}$$

$$(e3)\ \frac{(a, \texttt{dom}, b)\ (b, \texttt{sc}, c)}{(a, \texttt{dom}, c)} \qquad (e4)\ \frac{(a, \texttt{range}, b)\ (b, \texttt{sc}, c)}{(a, \texttt{range}, c)}$$

Figure 3.8: Additional rules used to compute the extended reduction of an RDFS ontology.

necessary for associating an unambiguous cost to query answers, so that they can be returned incrementally in order of increasing relaxation cost. In particular, we consider the cost of applying rule 2, 4, 5, or 6 of Figure 3.6 to be, respectively, c_{r2}, c_{r4}, c_{r5} or c_{r6}, each of which is a positive integer. (Since queries and data graphs cannot contain `sc` and `sp`, rules 1 and 3 are inapplicable to them, although of course they are used in computing $\text{cl}(G \cup K)$.)

The set of variables mentioned in a triple pattern t is denoted by $\texttt{var}(t)$. Let t_1 and t_2 be normalised triple patterns (see Definition 3.6) such that $t_1, t_2 \notin \text{cl}(G \cup K)$, and $\texttt{var}(t_2) = \texttt{var}(t_1)$. Then t_1 *relaxes to* t_2, denoted $t_1 \leq t_2$, [2] if $(\{t_1\} \cup G \cup K) \models_{\texttt{rule}} t_2$. Note that when applying the rules of Figure 3.6 to triple patterns, rather than (ground) triples, a, b and c must be instantiated to RDF terms, while X and Y can be instantiated to either RDF terms or variables.

Given data graph G, ontology K and triple patterns t_1 and t_2, let G_1 and G_2 be the sets of triples in the closure of G that are 'matched' by t_1 and t_2, respectively. Then it can be shown that $t_1 \leq t_2$ if and only if $(G_1 \cup K) \models_{\texttt{rule}} G_2$. From now on in this section we assume that all triple patterns have been normalised, and likewise all triple forms of queries and paths.

A *graph pattern* P is a set of triple patterns. The set of variables mentioned in P is denoted by $\texttt{var}(P)$. Let P_1 and P_2 be graph patterns such that $\texttt{var}(P_2) = \texttt{var}(P_1)$ and for all $t_1 \in P_1$ and $t_2 \in P_2$, $t_1, t_2 \notin \text{cl}(G \cup K)$. Then P_1 *relaxes to* P_2, denoted $P_1 \leq P_2$, if for all $t_1 \in P_1$ there is a $t_2 \in P_2$ such that $t_1 \leq t_2$ and for all $t_2 \in P_2$ there is a $t_1 \in P_1$ such that $t_1 \leq t_2$. The relaxation relation is reflexive and transitive.

[2]For notational simplicity we assume that the parameters G and K are implicit.

Example 3.8. (Drawn from [114]). Consider the ontology K described in Example 3.1 and shown in Figure 3.2. Let query Q comprise the single conjunct $(X, R, \text{'FL56'})$, where X is a variable, 'FL56' is a constant, and $R = (ppn_1^- \cdot fn_1)$. Recall that K contains the triples $(fn_1, \texttt{dom}, F_1)$ and $(F_1, \texttt{sc}, F)$. There is only a single $q \in L(R)$, namely $q = ppn_1^- fn_1$. Recalling the definition of a 'triple form' in Section 3.3, consider the following normalised triple form T of (Q, q):

$$(W_1, ppn_1, X), (W_1, fn_1, \text{'FL56'})$$

Notice that a triple form of (Q, q) is a graph pattern. Let P be the following graph pattern:

$$(W_1, ppn_1, X), (W_1, \texttt{type}, F)$$

Then T relaxes to P since $(W_1, fn_1, \text{'FL56'}) \leq (W_1, \texttt{type}, F_1)$ by rule 5, and $(W_1, \texttt{type}, F_1) \leq (W_1, \texttt{type}, F)$ by rule 4.

Note that, because of the requirement that variables be preserved when performing relaxation, rules 4, 5 and 6 can only be applied to the first or last triple pattern of a triple form of a sequence of labels, and even then only when a constant is present (as above). So, for example, $(ppn_1, \texttt{range}, Lit) \in K$ but we cannot apply rule 6 to the triple pattern (W_1, ppn_1, X) to relax it to $(X, \texttt{type}, Lit)$ because variable W_1 is lost in the process; similarly, we cannot apply rule 5 using $(ppn_1, \texttt{dom}, F_1)$ to the triple pattern (W_1, ppn_1, X) to relax it to $(W_1, \texttt{type}, F_1)$. We make use of this restriction in our algorithm for computing relaxed answers in Section 3.5.2 below.

□

We previously defined the exact semantics of single-conjunct regular path queries in Section 3.3. We now define the *relaxed semantics* of such queries.

Definition 3.13. Given a query Q with single conjunct (X, R, Y) and the closure of a data graph G with respect to an ontology K, $closure_K(G)$, let θ be a (Q, G)-matching. We use the notation $\theta(Q)$ to denote $(\theta(X), R, \theta(Y))$. A semipath p in $closure_K(G)$ *r-conforms* to $\theta(Q)$ if there is a $q \in L(R)$, a triple form T_q of $(\theta(Q), q)$ and a triple form T_p of p such that $T_q \leq T_p$.

□

Note that a path in $closure_K(G)$ can r-conform to a query on the basis of a triple

pattern t relaxing to a triple pattern t' such that the constants in t and t' differ, due to applications of rules 5 and 6 (for example, the triple patterns $(W_1, fn_1, \text{'FL56'})$ and $(W_1, \texttt{type}, F)$ in the previous example).

Example 3.9. (Drawn from [114].) Consider the query Q_3 from Example 3.3, the graph G of Figure 3.1 and the ontology K of Figure 3.2. Suppose that the second conjunct is used in a single-conjunct query Q_4 as follows:

$$Y \leftarrow RELAX(Y, pn_1^-.\texttt{type}, P_1)$$

Using matching θ_1 that matches Y to '1234', semipath $(\text{'1234'}, pn_1^-, p_1, \texttt{type}, P_1)$ r-conforms to $\theta_1(Q_4)$ since it matches the query exactly. Using matching θ_2 that matches Y to '6789', semipath $(\text{'6789'}, pn_2^-, p_2, \texttt{type}, P_2)$ r-conforms to $\theta_2(Q_4)$ because the normalised triple form $(W, pn_1, \text{'6789'}), (W, \texttt{type}, P_1)$ for $\theta_2(Q_4)$ relaxes to the triple form $(W, pn, \text{'6789'}), (W, \texttt{type}, P)$ which is a triple form for the semipath $(\text{'6789'}, pn^-, p_2, \texttt{type}, P)$ in $closure_K(G)$.

□

We now consider the cost of applying relaxations in order to be able to return answers ordered by increasing cost. For this we need the notion of *direct relaxation* [66]. The *direct relaxation relation*, which we denote here by $\preceq_R$, was defined in [66] to be the reflexive, transitive reduction of the relaxation relation $\leq$[3]. The *direct relaxations* of a triple pattern t (i.e., the triple patterns t' such that $t \preceq_R t'$) are the result of the smallest steps of relaxation (and the *indirect relaxations* of a triple pattern t are the triples t' such that $t \leq t'$ and $t \not\preceq_R t'$).

It is shown in [66] that a single application of each of the rules 2, 4, 5, 6 of Figure 3.6 to a triple pattern t and a triple $o \in \texttt{extRed}(K)$ yields precisely the direct relaxations of t with respect to K. We now extend this to graph patterns:

Given graph patterns P_1 and P_2, we say that P_1 *directly relaxes* to P_2, denoted $P_1 \preceq_R P_2$, if $P_1 = \{t_1\} \cup P$ and $P_2 = \{t_2\} \cup P$, for some (possibly empty) graph pattern P, and $t_1 \preceq_R t_2$. The *cost* of the direct relaxation is the cost of applying the rule that derives t_2 from t_1. The cost of a sequence of direct relaxations is the sum of the costs of each relaxation in the sequence.

[3] Once again for notational simplicity, we view the parameters G and K as being implicit.

Definition 3.14. Given ontology $K = \texttt{extRed}(K)$, graph $G = closure_K(G)$, semipath p in G, query Q with single conjunct (X, R, Y), sequence of labels $q \in L(R)$, (Q, G)-matching θ, triple form T_q for $(\theta(Q), q)$, and triple form T_p for p such that $T_q \leq T_p$:

- the *relaxation distance* from p to $(\theta(Q), q)$ is the minimum cost of any sequence of direct relaxations which yields T_p from T_q; the cost of the empty sequence of direct relaxations (so that T_q is already a triple form of p) is zero;
- the *relaxation distance* from p to $\theta(Q)$ is the minimum relaxation distance from p to $(\theta(Q), q)$ for any sequence of labels $q \in L(R)$;
- the *relaxation distance* of $\theta(Q)$, denoted $rdist(\theta, Q)$, is the minimum relaxation distance to $\theta(Q)$ from any semipath p that r-conforms to $\theta(Q)$;
- the *relaxed answer* of Q on G is a list of pairs $(\theta(vars), rdist(\theta, Q))$, such that there is a semipath in G that r-conforms to $\theta(Q)$, ranked in order of non-decreasing relaxation distance;
- the *relaxed top-k answer* of Q on G is a list containing the first k tuples in the relaxed answer of Q on G.

□

3.5.2 Computing the relaxed answer

We now describe how the relaxed answer can be computed, starting from the weighted NFA M_R that recognises $L(R)$ which was described in Section 3.4.1.

Given a query Q with single conjunct (X, R, Y), a weighted automaton $M_R = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, s_0, S_f, \xi)$ that does not contain ϵ-transitions, and ontology K such that $K = \texttt{extRed}(K)$, we construct as described below the automaton $M_R^K = (S', \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \tau, S_0, S'_f, \xi')$ of M_R with respect to K. The set of states S' includes the states in S as well as any new states defined below. S_0 and S'_f are sets of initial and final states, respectively, with S_0 including the initial state s_0 of M_R, S'_f including all final states S_f of M_R, and both possibly including additional states as defined below. We obtain the *relaxed automaton* M_Q^K by annotating each

state in S_0 and S'_f either with a constant or with the wildcard symbol $*$, depending on whether X and Y in Q are constants or variables. We recall that ξ (and initially ξ') is the final weight function mapping each state in S_f to a non-negative integer. The transition relation τ includes the transitions in δ as well as any transitions specified by the rules defined below. The rules below are applied repeatedly until no new transitions or states arise. The process terminates because of our assumption that the subgraphs of K induced by edges labelled *sc* and *sp* are acyclic:

- (rule 2) For each transition $(s, a, d, t) \in \tau$ (respectively $(s, a^-, d, t) \in \tau$) and triple $(a, \mathtt{sp}, b) \in K$, there is a transition $(s, b, d+c_{r2}, t)$ (resp. $(s, b^-, d+c_{r2}, t)$) in τ.

- (rule 4 (i)) If there is a transition $(s, \mathtt{type}, d, t) \in \tau$ where $t \in S'_f$ and $(c, \mathtt{sc}, c') \in K$ such that t is annotated with c, there is a final state t' annotated with c' in S'. State t' has the same set of outgoing transitions as t. For each transition $(s, \mathtt{type}, d, t) \in \tau$, there is a transition $(s, \mathtt{type}, d + c_{r4}, t')$ in τ.

- (rule 4 (ii)) If there is a transition $(s, \mathtt{type}^-, d, t) \in \tau$ where $s \in S_0$ and $(c, \mathtt{sc}, c') \in K$ such that s is annotated with c, there is an initial state s' annotated with c' in S'. State s' has the same set of incoming transitions as s. For each transition $(s, \mathtt{type}^-, d, t) \in \tau$, there is a transition $(s', \mathtt{type}^-, d+c_{r4}, t)$ in τ.

- (rule 5) If there is a transition $(s, a, d, t) \in \tau$ (resp. $(s, a^-, d, t) \in \tau$) where $t \in S'_f$ (resp. $s \in S_0$) and $(a, \mathtt{dom}, c) \in K$, there is a final state t' (resp. initial state s') annotated with c in S'. State t' has the same set of outgoing transitions as t (resp. s' has the same set of incoming transitions as s). For each transition $(s, a, d, t) \in \tau$ (resp. $(s, a^-, d, t) \in \tau$), there is a transition $(s, \mathtt{type}, d + c_{r5}, t')$ (resp. $(s', \mathtt{type}^-, d + c_{r5}, t)$) in τ.

- (rule 6) If there is a transition $(s, a, d, t) \in \tau$ (resp. $(s, a^-, d, t) \in \tau$) where $s \in S_0$ (resp. $t \in S'_f$) and $(a, \mathtt{range}, c) \in K$, there is an initial state s' (resp. final state t') annotated with c in S'. State s' has the same set of incoming transitions as s (resp. t' has the same set of outgoing transitions as t). For

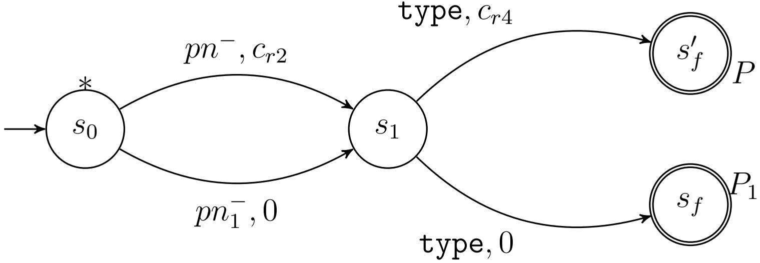

Figure 3.9: Relaxed automaton M_Q^K for conjunct $(Y, pn_1^-.\texttt{type}, P_1)$.

each transition $(s, a, d, t) \in \tau$ (resp. $(s, a^-, d, t) \in \tau$), there is a transition $(s', \texttt{type}^-, d + c_{r6}, t)$ (resp. $(s, \texttt{type}, d + c_{r6}, t')$) in τ.

We call the initial and final states specified by rules 4, 5 and 6 *cloned* states.

Example 3.10. (Drawn from [114]). Consider once again the ontology K shown in Figure 3.2 and the query Q_4 from Example 3.9:

$$Y \leftarrow RELAX(Y, pn_1^-.\texttt{type}, P_1)$$

The relaxed automaton M_Q^K initially comprises the states $\{s_0, s_1, s_f\}$ and two transitions labelled with cost zero between them, as shown in the lower part of Figure 3.9. From Figure 3.2, we see that K contains the triples $(pn_1, \texttt{sp}, pn)$ and $(P_1, \texttt{sc}, P)$. If we apply the transformation for rule 2 to the transition labelled $pn_1^-, 0$ and the triple $(pn_1, \texttt{sp}, pn) \in K$, then we add the transition labelled pn^-, c_{r2} from s_0 to s_1. If we now apply the transformation for rule 4 to the transition labelled $\texttt{type}, 0$ and the triple $(P_1, \texttt{sc}, P) \in K$, then a new final state s'_f, annotated with P is added, as is the transition labelled $\texttt{type}, c_{r4}$ from s_1 to s'_f.

Now consider the closure of the graph G shown in Figure 3.7 with respect to the ontology K shown in Figure 3.2. A subautomaton of the product automaton H of M_Q^K and $closure_K(G)$ is shown in Figure 3.10.

For a matching θ_1 that maps Y to '1234', it is easy to see that there are four runs in H from the initial state $(s_0, 1234)$ to a final state, with costs 0, c_{r2}, c_{r4} and $c_{r2} + c_{r4}$. Hence the relaxation distance of the answer '1234' is 0. For a matching θ_2 that maps Y to '6789', there is only one run in H from $(s_0, 6789)$ to a final state, with cost $c_{r2} + c_{r4}$, so the relaxation distance of the answer '6789' is $c_{r2} + c_{r4}$.

□

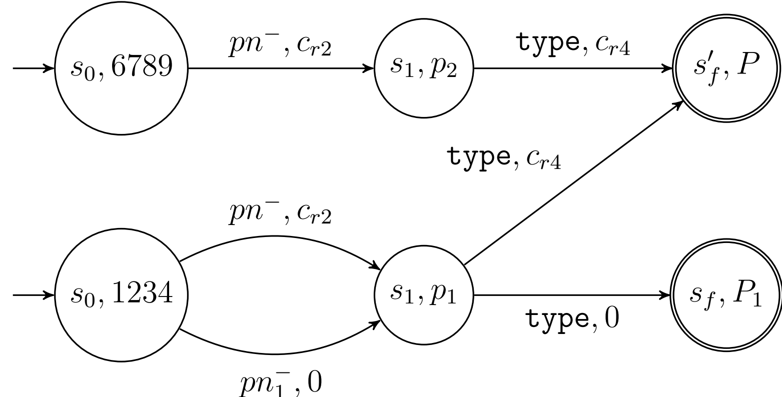

Figure 3.10: A subautomaton of the product automaton H.

In Chapter 4, Proposition 4.2 shows firstly the correctness of the construction and traversal of the product automaton H; and secondly that the relaxation distance from a semipath in a graph G to the matchings for a single-conjunct query Q is equal to the minimum cost of a run in H.

3.5.3 Incremental evaluation of RELAX conjuncts

In order to compute the relaxed answers to a single-conjunct regular path query incrementally, we can use the `GetNext` function from Section 3.4.2 along with the same initialisations of the algorithm's variables. The only difference is that the `Succ` function now uses the automaton M_Q^K rather than the automaton A_Q.

3.6 Multi-conjunct queries

We recall the general form of a CRPQ Q from Section 3.2:

$$(Z_1, \ldots, Z_m) \leftarrow (X_1, R_1, Y_1), \ldots, (X_n, R_n, Y_n)$$

where any of the conjuncts may have the APPROX or RELAX operator applied to them. Let θ be a (Q, G)-matching. If conjuncts $i_1, \ldots, i_j$, $j \leq n$, have APPROX or RELAX applied to them, the *distance* from θ to Q, $dist(\theta, Q)$, is defined as

$$dist(\theta, (X_{i_1}, R_{i_1}, Y_{i_1})) + \cdots + dist(\theta, (X_{i_j}, R_{i_j}, Y_{i_j}))$$

where $dist(\theta, (X_{i_k}, R_{i_k}, Y_{i_k}))$ is the approximation distance if conjunct i_k has APPROX applied to it and is the relaxation distance if i_k has RELAX applied to it.

Let $\theta(Z_1, \ldots, Z_m) = (a_1, \ldots, a_m)$. θ is a *minimum-distance* matching if for all (Q, G)-matchings ϕ such that $\phi(Z_1, \ldots, Z_m) = (a_1, \ldots, a_m)$, $dist(\theta, Q) \leq dist(\phi, Q)$.

The *answer* of Q on G is the list of pairs $(\theta(Z_1, \ldots, Z_m),\ dist(\theta, Q))$, for some minimum-distance matching θ, ranked in order of non-decreasing distance. The *top-k answer* of Q on G comprises the first k tuples in the answer of Q on G.

The query Q can be evaluated by joining the answers arising from the evaluation of each of its conjuncts. For each conjunct with APPROX or RELAX applied to it we can use the techniques described in previous sections to incrementally compute a relation r_i with scheme $(X_i, Y_i, Distance)$. A query evaluation tree can be constructed for Q, consisting of nodes denoting join operators and nodes representing conjuncts of Q. Since the answers for single conjuncts are ordered by non-decreasing distance, pipelined execution of any rank-join operator (see e.g. [40]) can be used to output the answers to Q in order of non-decreasing distance. If the conjuncts of Q are acyclic, the evaluation can be accomplished in polynomial time [45], since there are a fixed number of head variables in Q and we process leaf nodes (denoting conjuncts) in a bottom-up manner (beginning at the leaf nodes) by executing a sequence of joins.

3.7 Summary

We began this chapter by introducing the graph-based data model adopted in this thesis, comprising a data graph and an ontology graph. We continued by presenting the query language we adopt, based upon conjunctive regular path queries (CRPQs), providing a formal definition of CRPQs and discussing exact matching of single-conjunct RPQs.

We provided details of the algorithms for the evaluation of single-conjunct queries, along with a formal presentation of approximate matching as denoted by the APPROX operator, and we described how approximate answers can be returned to the user incrementally.

We next discussed the relaxation of single-conjunct RPQs based on information from an ontology, as well as how answers for a conjunct to which the RELAX

operator has been applied can be computed. We also described how relaxed answers can be returned to the user incrementally.

We ended by describing the evaluation of multi-conjunct CRPQs, each of whose conjuncts may have APPROX or RELAX applied to them.

In the next chapter, we provide full proofs of correctness for the constructs and algorithms introduced in this chapter.

CHAPTER 4

Correctness and Complexity Results

In this chapter, we consider the correctness and complexity of the constructs and algorithms introduced in Chapter 3.

Section 4.1 presents proofs regarding the constructs defined in Section 3.4.1. We establish the correctness of both the construction of the approximate automaton and the traversal of the product automaton, as well as the fact that the minimum cost of a run in the product automaton corresponds to the approximation distance from a semipath in a graph to the matchings for a single-conjunct query. We prove that the approximate automaton can be constructed in time that is polynomial in the size of the query, and finally establish that the approximate answer to a single-conjunct RPQ can be computed in time that is polynomial in the size of the query and the graph. In Section 4.2 we show the correctness of our incremental evaluation algorithm introduced in Section 3.4.2.

Section 4.3 contains proofs relating to the constructs defined in Section 3.5.2. We show the correctness of the construction and traversal of the product automaton, and also that the relaxation distance from a semipath in a graph to the matchings for a single-conjunct query is equal to the minimum cost of a run in the product automaton. We also establish an upper bound for the size of the relaxed automaton, and prove that the relaxed answer of a single-conjunct query on the closure of a graph

can be computed in time that is polynomial in the size of the query, the graph and the ontology. We end the chapter with some concluding remarks in Section 4.4.

For all the complexity-related proofs in this thesis, we make the assumption that any occurrence of "_" in a regular expression — denoting the disjunction of all constants in $\Sigma \cup \{\texttt{type}\}$ — has been rewritten to $a_1|a_2|...|a_n|\texttt{type}$ where $\Sigma = \{a_1, \ldots, a_n\}$.

4.1 Approximation of single-conjunct queries

All the proofs in this section relate to the constructs presented in Section 3.4.1.

We begin by showing that using the automaton A_R, formally given by $A_R = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, \{s_0\}, \{s_f\}, \xi)$, is sufficient to find all sequences of labels generated by edit operations at an approximation distance k from a given query. To do so, we make use of the concept of a *trace* of edit operations, introduced in [132]. A trace is essentially a sequence of edit operations in which order is unimportant and redundancy is not present. More specifically, edit operations are applied only to labels in the original sequence, and redundant operations (such as the insertion of some previously-deleted label) are not performed.

In the following example, an edit sequence S transforms a string A into a string B. The sequence S might be given by: delete x, insert r after y, insert s after r, substitute w by u, and delete t.

```
String A:  x  y  z  w  t
            /     \   \
String B:  y  r  s  z  u
```

A line joining the character at position i in A to position j in B denotes that $B[j]$ is derived from $A[i]$, where $i, j \geq 0$. Thus, $A[1]$ (y) and $A[2]$ (z) remain unchanged by S — meaning that $B[0]$ (y) and $B[3]$ (z) are derived directly from A — and $B[4]$ (u) is derived from $A[3]$ (w) by the application of a single substitution by S to A. Similarly, the characters x and t have been deleted from A by S, and the characters r and s have been inserted into A by S.

Using a result from [132], we can assume, without loss of generality, that in our edit sequences, all deletions come first, followed by all substitutions which are then

followed by all insertions. Any ϵ-transitions in A_R will have been removed before the application of substitution and insertion operations.

Lemma 4.1. *Let Q be a single-conjunct RPQ of the form (X, R, Y). Let A_R be the automaton constructed for R as described in Definition 3.11 of Section 3.4.1, q be a string in $L(R)$, and p be a sequence of labels in $(\Sigma \cup \Sigma^- \cup \{\texttt{type} \cup \texttt{type}^-\})^*$ corresponding to a semipath in graph G. The approximation distance from p to q is equal to the minimum cost of a run for p in A_R.*

Proof. Given graph G and (Q, G)-matching θ, let T_q be a triple form of $(\theta(Q), q)$, and T_p be a triple form for p. The approximation distance from p to q is defined as the minimum cost of any sequence of edit operations which yields T_p from T_q. The proof proceeds by induction on the number of edit operations used in any such minimum-cost edit sequence.

Basis: If no edit operations are used, then, by definition, the cost of the edit sequence is 0, and T_q is already a triple form of p. Thus, there is a run of cost 0 in A_R. Clearly, this run is of minimum cost.

Induction: For the inductive step, assume that there is an $n \geq 0$ such that, for all $m \leq n$, if m edit operations are used in a minimum-cost edit sequence of cost c which yields T_p from T_q, then the minimum cost of a run for p in A_R is c.

Now consider a sequence of labels p which requires $n + 1$ edit operations in a minimum-cost edit sequence to produce T_p from T_q. Let this edit sequence be given by $S = P_0 \preceq_A P_1 \preceq_A \cdots \preceq_A P_n \preceq_A P_{n+1}$, where $P_0 = T_q$ and $P_{n+1} = T_p$. Let the cost of a deletion, substitution and insertion be denoted by c_d, c_s and c_i, respectively. The cost of the sequence S is $k = n_d c_d + n_s c_s + n_i c_i$, where n_d, n_s and n_i are the number of deletions, substitutions and insertions, respectively, used in S for some $n_d \geq 0$, $n_s \geq 0$ and $n_i \geq 0$, and $n_d + n_s + n_i = n + 1$.

Using a result from [132], we can assume that in edit sequence S all deletions appear first, followed by all substitutions, and then all insertions. Let op_E denote the edit operation applied to P_n to yield $P_{n+1} = T_p$. The proof proceeds by considering the possible alternatives for op_E:

(1) *op_E is a deletion*: There are two cases to consider, depending on whether or not the deleted triple pattern is the last in P_n.

(i) We first consider the case in which the deleted triple pattern is not the last in

P_n. So assume that op_E deletes the triple pattern (W_{m-1}, b, W_m) in P_n to produce $P_{n+1} = T_p$, thus transforming the pair of triple patterns $t'_f = (W_{m-1}, b, W_m), (W_m, g, W_{m+1})$ in P_n into $t_f = (W_{m-1}, g, W_m)$ in P_{n+1}. Let the subsequence of sequence S up to P_n be denoted by S'. By definition, sequence S' uses n edit operations and has cost $k - c_d$. By the inductive hypothesis, there is a minimum cost run r of cost $k - c_d$ in A_R for the sequence corresponding to triple form P_n. Suppose that in run r the subsequence t'_f in P_n is matched by the transitions (s_1, b, d_1, s_2) and (s_2, g, d_2, s_3) in A_R. We will show that there is a minimum cost run of cost k in A_R which matches T_p.

Suppose that, prior to the removal of ϵ-transitions, t'_f corresponds to the sequence of transitions in A_R given by $s' = \Gamma, (s_4, b, 0, s_2), \Delta, (s_5, g, 0, s_3)$, where Γ and Δ represent sequences of ϵ-transitions. Sequence Γ represents the presence of x_1 ϵ-transitions which include y_1 ϵ-transitions each of cost c_d, representing the deletion of labels in T_q prior to b in some triple form prior to P_n in S, where $x_1 \geq 0$, $y_1 \leq x_1$, and $s_1 = s_4$ if $x_1 = 0$. Sequence Δ represents the presence of x_2 ϵ-transitions which include y_2 ϵ-transitions each of cost c_d, representing the deletion of labels from T_q between b and g in some triple form prior to P_n in S, where $x_2 \geq 0$, $y_2 \leq x_2$, and $s_2 = s_5$ if $x_2 = 0$.

After the removal of ϵ-transitions, s' is transformed to $t' = (s_1, b, d_1, s_2), (s_2, g, d_2, s_3)$ in run r, where $d_1 = y_1 c_d$ and $d_2 = y_2 c_d$, as shown in Figure 4.1. Thus, the cost of run r is $k - c_d = d_1 + d_2 + e$, where e is the cost of the remaining transitions in r. By construction, the transition $(s_4, \epsilon, c_d, s_2)$ representing the deletion of b is present in A_R, as shown in Figure 4.1.

There is a sequence of ϵ-transitions from s_1 to s_4 of cost d_1. Along with the ϵ-transition from s_4 to s_2, this means there is a sequence of ϵ-transitions from s_1 to s_2 of cost $d_1 + c_d$.

There is a sequence of ϵ-transitions from s_2 to s_5 of cost d_2. Therefore there is a path of ϵ-transitions from s_1 via s_4 and s_2 to s_5 of cost $d_1 + c_d + d_2$. As there is a transition of cost 0 labelled with g from s_5 to s_3, a transition labelled with g from s_1 to s_3 of cost $d_1 + c_d + d_2$ would have been added to A_R during the removal of ϵ-transitions (see Figure 4.1). Therefore, there is a minimum cost run of cost k in A_R which matches T_p.

(ii) We now consider the case where the deleted triple pattern *is* the last in

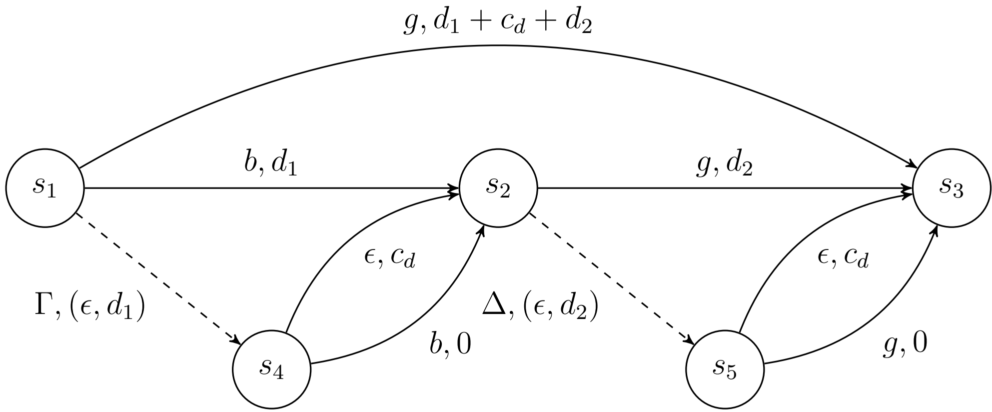

Figure 4.1: Automaton for the deletion of the label b in T_{p_k} (b is not the last label).

P_n. Assume that op_E deletes the triple pattern (W_m, g, W_{m+1}) in P_n, transforming triple form $t'_f = (W_{m-1}, b, W_m), (W_m, g, W_{m+1})$ in P_n into $t_f = (W_{m-1}, b, W_m)$ in $P_{n+1} = T_p$. As in case (i), we know by the inductive hypothesis that there is a minimum cost run r of cost $k - c_d$ in A_R for the sequence corresponding to triple form P_n. Suppose that in run r the subsequence t'_f in P_n is matched by the transitions (s_1, b, d_1, s_2) and (s_2, g, d_2, s_3) in A_R. We will show that there is a minimum cost run of cost k in A_R which matches T_p.

Suppose that prior to the removal of ϵ-transitions, t'_f is matched in A_R by the partial sequence of transitions given by s', as defined previously. After the removal of ϵ-transitions, s' is transformed to t', also as given previously. Using the same reasoning as in the case for when the deleted label is not the last in P_n, we note that $d_2 = y_2 c_d$, representing the deletion of y_2 ϵ-transitions, succeeding b and preceding g in T_q, in some sequence prior to P_n.

If there were m ϵ-transitions succeeding g in T_q, these would all have been deleted prior to P_n in sequence S'. As g is the last label, then, from the inductive hypothesis, we have that $\xi(s_3) = d_3$, where $d_3 = mc_d$. Thus, the cost of run r is $k - c_d = d_1 + d_2 + d_3 + e$ where e is the cost of the remaining transitions in r.

By construction, the transition $(s_5, \epsilon, c_d, s_3)$ representing the deletion of g is present in A_R. There is a sequence of ϵ-transitions from s_2 to s_5 of cost d_2, and, along with the ϵ-transition from s_5 to s_3 of cost c_d, this means there is a sequence of ϵ-transitions from s_2 to s_3 via s_5 of cost $d_2 + c_d$. Thus $\xi(s_2)$ is set to $d_2 + c_d + d_3$. Thus, there is a minimum cost run of cost k which matches T_p.

(2) op_E *is a substitution*: Suppose op_E replaces some triple $t' = (W_{m-1}, a, W_m)$ in P_n by $t = (W_{m-1}, b, W_m)$ in $P_{n+1} = T_p$. Hence the subsequence of sequence S up to P_n uses n edit operations and has cost $k - c_s$. By the inductive hypothesis, there is a minimum cost run r of cost $k - c_s$ in A_R for the sequence corresponding to triple form P_n.

Suppose that in run r triple t' is matched by the transition $f' = (s_1, a, d, s_2)$. Then, by construction, A_R has a transition $f = (s_1, b, d + c_s, s_2)$ matching t. Substituting f' in r by f yields a run r' in A_R with cost k which matches T_p.

(3) op_E *is an insertion*: Suppose op_E inserts a triple pattern $t = (W_{m-1}, a, W_m)$ after another triple pattern $t' = (W_{m-2}, h, W_{m-1})$ in P_n to produce $P_{n+1} = T_p$. Hence the subsequence of sequence S up to P_n uses n edit operations and has cost $k - c_i$. By the inductive hypothesis, there is a minimum cost run r of cost $k - c_i$ in A_R for the sequence corresponding to triple form P_n.

Assume that in run r triple t' is matched by the transition $f' = (s_1, h, d, s_2)$, where $d \geq 0$. By construction there is a transition (s_2, a, c_i, s_2) in A_R, and hence a sequence $(s_1, h, d, s_2), (s_2, a, c_i, s_2)$ matching t' and t. Using this sequence instead of (s_1, h, d, s_2) in r yields a run of cost k in A_R. The case for inserting a triple pattern *before* another triple pattern is analogous.

□

The next two lemmas show firstly the correctness of the traversal of the product automaton, H — formally given by $(T, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \sigma, I, T, \xi)$ — formed from the approximate automaton $A_Q = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, \{s_0\}, S, \xi)$ and a graph $G = (V_G, E_G, \Sigma)$ (as described in Definition 3.11, and in Definition 3.12 onwards in Section 3.4.1); and secondly that the approximation distance from a semipath in the graph G, to the matchings for a single-conjunct query Q is equal to the minimum cost of a corresponding run in H.

Lemma 4.2. *There is a run in H from (s_0, v_0) to (s_f, v_n) of cost k if and only if there is a semipath from v_0 to v_n in G and a run of cost k from s_0 to s_f in A_Q, for some initial state s_0 and some final state s_f in A_Q.*

Proof. There are two types of transition in H, as given in the definition of its construction. The first type of transition, $((s, n), a, c, (s', n'))$, is in H if and only if there is an edge labelled a from n to n' in G and a transition labelled a from s to s'

with a cost of c in A_Q. The second type of transition, $((s, n), a^-, c, (s', n'))$, is added to H if and only if there is an edge labelled a from n' to n in G and a transition labelled a^- from s to s' with a cost of c in A_Q.

Given the above, it is straightforward to show (by induction, for example) that there is a sequence of transitions of the form $((s_0, v_0), a_1, c_1, (s_1, v_1)), \ldots, ((s_{j-1}, v_{j-1}), a_{j-1}, c_{j-1}, (s_f, v_j))$ in H of cost $k = c_1 + \cdots + c_{j-1} + \xi[s_f]$ if and only if there is a semi-path from v_0 to v_j in G and a sequence of transitions of the form $(s_0, a_1, c_1, s_1), \ldots, (s_{j-1}, a_{j-1}, c_{j-1}, s_f)$ of cost k in A_Q.

□

Lemma 4.3. *Let θ be a matching from a query Q with single conjunct (X, R, Y) to a graph $G = (V_G, E_G, \Sigma)$, where $\theta(X) = v_0$ and $\theta(Y) = v_n$ for some $v_0, v_n \in V_G$. Let p be a semipath from v_0 to v_n in G, and H be the product automaton of A_Q and G. The approximation distance from p to $\theta(Q)$ is k if and only if k is the minimum cost of a run for the sequence of labels comprising p from (s_0, v_0) to (s_f, v_n) in H, for some initial state s_0 and some final state s_f in A_Q.*

Proof. ($\Rightarrow$) We know, by definition, that if the approximation distance from p to $\theta(Q)$ is k, then k is the *minimum* approximation distance from p to $(\theta(Q), q)$ for any sequence of labels $q \in L(R)$. For any such $q \in L(R)$, we also know, by definition, that if the approximation distance from p to $(\theta(Q), q)$ is k, then k is the minimum cost of any sequence of edit operations which yields triple form T_p from triple form T_q.

From Lemma 4.1, we know that if p has approximation distance k from q, the minimum cost of a run from s_0 to s_f for p in A_R, and hence A_Q, is k. From Lemma 4.2, we know that there is a run of cost k for p from (s_0, v_0) to (s_f, v_n) in H. There can be no run of cost less than k for p in H since this would contradict the fact that p is of approximation distance k from $\theta(Q)$.

($\Leftarrow$) Suppose that the minimum cost of a run for the sequence of labels comprising p from (s_0, v_0) to (s_f, v_n) in H, for some initial state s_0 and some final state s_f in A_Q, is k. This means that, by Lemma 4.2, A_Q has a minimum cost of k for a run for p. Hence, the minimum cost of edit operations needed to obtain T_p from T_q, for *any* $q \in L(R)$, must be k. Therefore the approximation distance from p to $\theta(Q)$ is k.

□

We next consider the complexity of approximate matching. We first prove in Lemma 4.4 that the automaton A_Q can be constructed in time polynomial in the size of the query, and then show in Proposition 4.1 that the approximate answer to a single-conjunct RPQ can be computed in time that is polynomial in the size of the query and the graph. We subsequently show that the data and query complexity is polynomial in the size of the graph and the regular expression of the query respectively, and that the space complexity is dominated by the space required by the product automaton.

Lemma 4.4. *A_Q has at most $2|R|$ states and $4|R|^2|\Sigma|$ transitions, and can be constructed in $O(|R|^3|\Sigma|)$ time.*

Proof. From [2], we have that M_R contains at most $2|R|$ states and $4|R|$ transitions. Deletions add at most $|R|$ more transitions, giving $5|R|$ transitions in total. The subsequent removal of ϵ-transitions may result in A_Q having at most $25|R|^2$ transitions. Insertions add at most $2|R||\Sigma|$ transitions, and substitutions add at most $25|R|^2|\Sigma|$ transitions. However, since a directed, labelled multi-graph with $2|R|$ nodes and $|\Sigma|$ distinct labels can have at most $4|R|^2|\Sigma|$ edges, this is also the bound for the number of transitions in A_Q. From [32], we have that the construction of A_Q can be performed in $O(|R|^3|\Sigma|)$ time.

□

Proposition 4.1. *Let $G = (V_G, E_G, \Sigma)$ be a graph and Q be a single-conjunct query using regular expression R over alphabet $\Sigma \cup \{\texttt{type}\}$. The approximate answer of Q on G can be found in time $O(|R|^2|V_G|(|R||\Sigma||E_G| + |V_G|\log(|R||V_G|)))$.*

Proof. Let A_Q be the approximate automaton constructed from R, and H be the product graph constructed from A_Q and G. Lemma 4.3 shows that traversing H correctly yields all approximate answers of Q. Lemma 4.4 tells us that A_Q has at most $2|R|$ states and $4|R|^2|\Sigma|$ transitions. Therefore H has at most $2|R||V_G|$ nodes and $4|R|^2|E_G||\Sigma|$ edges. If we assume that H is sparse (which is highly likely), then running Dijkstra's algorithm on each node of a graph with node set N and edge set A can be done in time $O(|N||A| + |N|^2 \log |N|)$. So, for graph H, the combined time complexity is $O(|R|^3|V_G||E_G||\Sigma| + |R|^2|V_G|^2 \log(|R||V_G|))$ which is

equal to $O(|R|^2|V_G|(|R||\Sigma||E_G| + |V_G|\log(|R||V_G|)))$.

□

As a corollary, it is easy to see that the data complexity is $O(|V_G||\Sigma||E_G| + |V_G|^2\log(|V_G|))$ and the query complexity is $O(|R|^3)$. The space complexity is dominated by the space requirements of H given in the proof above.

Further, we observe that the data complexity results are consistent with the results in our empirical evaluation, which is discussed in detail in our chapter on query performance analysis, in Section 7.1.4.

4.2 Incremental evaluation

The following theorem shows that our incremental evaluation algorithm, represented by `GetNext` — introduced in Section 3.4.2 — is correct: that is, given a single-conjunct query Q and a graph G, it returns the approximate answer of Q on G.

Theorem 4.1. *Let Q be a query with single conjunct (X, R, Y), and G a graph. Let* $\texttt{visited}_R$, $\texttt{queue}_R$ *and* $\texttt{answers}_R$ *be initialised as described in Section 3.4.2 and* `GetNext` *be called repeatedly until it returns null. When* `GetNext` *returns null, then (1) $(v, n, d) \in \texttt{answers}_R$ if and only if (v, n, d) is in the approximate answer of Q on G, and (2) if $\texttt{answers}_R[i] = (v, n, d)$ and $\texttt{answers}_R[j] = (v', n', d')$, for non-negative integers i and j with $i < j$, then $d \leq d'$.*

Proof. Throughout, we define H as being the product automaton of the graph G and the approximate automaton A_Q constructed for Q; and s_0 and s_f indicate an initial state and final state in A_Q, respectively.

Part (1): $(\Leftarrow)$ By Lemma 4.3, we need to show that if the minimum cost of *any* run from (s_0, v) to (s_f, n) is d, for some $v, n \in V_G$, then $(v, n, d) \in \texttt{answers}_R$.

We first show that if the minimum cost of any run in H from (s_0, v) to (s, n) (where s may or may not be a final state) is d, then tuple (v, n, s, d, f) is added to $\texttt{queue}_R$ before any tuple (v, n', s', d', f), where $d' > d$, is dequeued from $\texttt{queue}_R$. Assume that r is a minimum cost run in H from (s_0, v) to (s, n). The proof proceeds by induction on the number of transitions in r having non-zero cost.

Basis: For the base case, there are no transitions with non-zero cost in r, so the cost of r is zero. By definition we have that the tuple $(v, v, s_0, 0, f)$, possibly as

one of a set of initial tuples, is enqueued in $\texttt{queue}_R$. When one of these zero-cost tuples is dequeued at line (2), we can see, by Lemma 3.1 and the invocation of $\texttt{Succ}(s_0, v, A_Q, G)$ at line (5), that all tuples representing the successive transitions in H will be enqueued in $\texttt{queue}_R$ at line (6); each of these tuples subsequently undergoes the same process. Because run r ends with (s, n), the tuple $(v, n, s, 0, f)$, where f is 'non-final', will be added to $\texttt{queue}_R$.

Now assume that, at some point, (v, n', s', d', f), where $d' > 0$ and f is 'non-final', is added to $\texttt{queue}_R$. As $\texttt{queue}_R$ is a priority queue ordered by non-decreasing values of cost, it is straightforward to see that (v, n', s', d', f) will not be dequeued before tuple $(v, n, s, 0, f)$ is enqueued.

Induction: For the inductive step, suppose that there is an $n \geq 0$ such that, for all $m \leq n$, if a minimum cost run in H of cost k from (s_0, v) to (u, w), say, has m transitions of non-zero cost, then tuple (v, w, u, k, f) will be placed on $\texttt{queue}_R$ before any tuple (v, w', u', k', f), $k' > k$, is dequeued from $\texttt{queue}_R$.

Now consider a minimum-cost run r of cost k which contains $n + 1$ transitions with non-zero cost. Let run r be from (s_0, v) to (s, n), and let transition t be the last transition in r labelled with a cost $c > 0$. Hence, r can be viewed as run $r' \cdot r''$, where t is the first transition on r''. Clearly, the number of transitions with a non-zero cost in r' is n, and the cost of r' is $k - c$. Let (u, w) be the state in H at which r' ends and r'' starts. By the induction hypothesis, we know that the tuple $(v, w, u, k - c, f)$ will have been placed on $\texttt{queue}_R$ before any tuple $(v, w', u', k' - c, f)$, where $k' > k$, is dequeued from $\texttt{queue}_R$.

Suppose, in the worst case, that there is already a tuple (v, n', s', k', f), $k' > k$, on $\texttt{queue}_R$. Since $k - c < k'$, tuple $(v, w, u, k - c, f)$ will be dequeued before (v, n', s', k', f). Suppose that transition t is form (u, w) to (x, y) in H. When the tuple $(v, w, u, k - c, f)$ is dequeued at line (2), we know from Lemma 3.1 that invoking $\texttt{Succ}(u, w, A_Q, G)$ at line (5), will, at line (6), enqueue all tuples representing successive transitions in H, including the tuple (v, y, x, k, f).

If $y = n$ and $x = s$, the proof is complete. If not, tuple (v, y, x, k, f) will be dequeued before (v, n', s', k', f) since $k < k'$ and $\texttt{queue}_R$ is a priority queue. Because the remaining transitions on r'' are of cost zero, it is easy to see that (v, n, s, k, f) will be added to $\texttt{queue}_R$ before any (v, n', s', k', f) is dequeued.

So for a minimum cost run of cost d from (s_0, v) to (s_f, n), where s_f is a final

state, we know that tuple (v, n, s_f, k, f) will also be *dequeued* from $\texttt{queue}_R$ before any tuple (v, n', s', d', f), $d' > d$, is dequeued (because $\texttt{queue}_R$ is a priority queue). The (v, n, s) triple is then added to $\texttt{visited}_R$ at line (4), enqueued as a 'final' tuple at line (8), and once again dequeued before any tuple at greater cost. This time the dequeued tuple results in the tuple (v, n, d) being added to $\texttt{answers}_R$ at line (11).

($\Rightarrow$) We show that if $(v, n, d) \in \texttt{answers}_R$, then the minimum cost of any run in H from (s_0, v) to (s_f, n), for some final state s_f, is d. The result then follows by Lemma 4.3. The proof is by contradiction.

Suppose that a triple $(v, n, d) \in \texttt{answers}_R$ but that the minimum cost of a run in H from (s_0, v) to (s_f, n) is $d' < d$. As (v, n, d) was added to $\texttt{answers}_R$ at line (11), we know, by line (7), that the triple (v, n, d') was not added to $\texttt{answers}_R$ prior to the tuple (v, n, s, d, f) being dequeued from $\texttt{queue}_R$ at line (2). There are only two possibilities which could give rise to this state.

The first possibility is that the tuple (v, m, s, d, f) was dequeued *before* the tuple (v, m, s, d', f) was enqueued in $\texttt{queue}_R$. However, as we have seen in ($\Leftarrow$) above, (v, m, s, d, f) cannot be dequeued before (v, m, s, d', f) is enqueued. Thus, we have a contradiction.

The second possibility is that, at line (5), the invocation of $\texttt{Succ}(s_i, v_i, A_Q, G)$, for some state s_i and some node v_i, did not return a transition $\xrightarrow{d''} (s_f, n)$, for some cost d'', where $d'' \leq d'$, and hence the tuple (v, m, s, d', f) was never enqueued at line (6). But we know, from Lemma 3.1, that $\texttt{Succ}$ returns all and only transitions that occur in H. Thus, we have a contradiction.

Therefore, either $(v, n, d) \notin \texttt{answers}_R$ or the minimum cost of any run in H from (s_0, v) to (s_f, n) is d.

Part (2): From Part (1), we have that the tuple (v, n, s, d, f) — representing the minimum cost run in H from (s_0, v) to (s, n) — is dequeued from $\texttt{queue}_R$ before any tuple (v, n', s', d', f), where $d < d'$, is dequeued.

Suppose we have two runs, r and r'; suppose run r is from (s_{r0}, v_r) to (s_{rf}, n_r) of minimum cost d_r and run r' from (s'_{r0}, v'_r) to (s'_{rf}, n'_r) of minimum cost d'_r, where s_{rf} and s'_{rf} are final states. It is then straightforward to see that if the triple (v_r, n_r, d_r) had been added as the ith item in $\texttt{answers}_R$ as a result of completely traversing run r, and the triple (v'_r, n'_r, d'_r) had been added as the jth item in $\texttt{answers}_R$ as a result of completely traversing run r', for some $i < j$, then $d_r \leq d'_r$.

□

4.3 Relaxation of single-conjunct queries

All the proofs in this section relate to the constructs introduced in Section 3.5.2.

The following proposition shows firstly the correctness of the construction and traversal of the product automaton, H, constructed as described in Section 3.5.2; and secondly that the relaxation distance from a semipath in a graph G to the matchings for a single-conjunct query Q is equal to the minimum cost of a run in H.

Proposition 4.2. *Let Q be a query comprising a single conjunct (X, R, Y). Let $M_Q^K = (S', \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \tau, S_0, S_f, \xi)$ be the relaxed automaton for query Q and ontology $K = \texttt{extRed}(K)$, where the ϵ-transitions have been removed from M_Q^K. Let $G = closure_K(G)$ be a graph and H be the product automaton of M_Q^K and G. Let θ be a (Q, G)-matching such that $\theta(X) = v_0$ and $\theta(Y) = v_n$. (1) There is a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G that r-conforms to $\theta(Q)$ if and only if there is a run $r = ((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}),\ l_n, c_n, (s_n, v_n))$ in H, where $s_0 \in S_0$ and $s_n \in S_f$. (2) The relaxation distance of $\theta(Q)$ is given by $c_1 + \cdots + c_n$ if and only if r is of minimum cost over all runs from (s_0, v_0) to (s_n, v_n) for any $s_0 \in S_0$ and $s_n \in S_f$.*

Proof. Part (1): ($\Rightarrow$) Let p be a semipath $(v_0, l_1, \ldots, l_n, v_n)$ in G that r-conforms to $\theta(Q)$. Hence there is a $q \in L(R)$, a triple form T_q of $(\theta(Q), q)$ and a triple form T_p of p such that $T_q \leq T_p$. Since $q \in L(R)$, we know that there is a run in M_R corresponding to T_q. We therefore need to show that there is a run in M_Q^K corresponding to T_p. The proof proceeds by induction on the number of direct relaxations required to yield T_p from T_q.

Basis: When the number of direct relaxations applied to T_q is zero, $T_q = P_0 = T_p$. Thus, there is a run for T_p in M_Q^K, which corresponds to a run in M_R.

Induction: For the inductive step, assume that there is an $n \geq 0$ such that, for all $m \leq n$, if m direct relaxations are used in a relaxation sequence which yields T_p from T_q, there is a corresponding run in M_Q^K.

Now assume that $n+1$ direct relaxations are required to produce T_p from T_q. Let such a sequence be given by $S = P_0 \preceq_R \cdots \preceq_R P_n \preceq_R P_{n+1}$, where $P_0 = T_q$ and $P_{n+1} = T_p$. From the induction hypothesis, we know that there is a run r_n in M_Q^K corresponding to the sequence of direct relaxations given by the sequence $S' = P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$.

We consider in turn each type of direct relaxation operation induced by the rules given in Figure 3.6 which can be used to produce P_{n+1} from P_n. We show that, corresponding to each such operation, is a transition in M_Q^K which, when traversed, gives rise to a run for T_p in M_Q^K. In all cases below, d denotes the cost of the transition.

Rule 2: Suppose that P_{n+1} is produced by applying rule 2 to the triple $(a, \mathtt{sp}, b) \in K$ and the triple pattern $f = (W_{m-1}, a, W_m)$ in P_n which results in f being replaced by (W_{m-1}, b, W_m) in P_{n+1}, where W_{m-1} and W_m are variables or constants. Suppose that f is matched by the transition (h, a, d, s) in r_n. By construction, M_Q^K has a transition $(h, b, d + c_{r2}, s)$ and hence replacing (h, a, d, s) by $(h, b, d + c_{r2}, s)$ in r_n yields a run for P_{n+1}.

On the other hand, if P_{n+1} is produced by applying rule 2 to (W_m, a, W_{m-1}) in P_n, where the matching transition in r_n is (h, a^-, d, s), then (W_m, b, W_{m-1}) is in P_{n+1} and there is a transition $(h, b^-, d + c_{r2}, s)$ in M_Q^K.

Rule 4: Suppose that P_{n+1} is produced by applying rule 4 to the triple $(c, \mathtt{sc}, c') \in K$ and the triple pattern $f = (W_m, \mathtt{type}, c)$ in P_n which results in f being replaced by $(W_m, \mathtt{type}, c')$ in P_{n+1}, where W_m is a variable. Suppose that f is matched by the transition $(h, \mathtt{type}, d, s)$ in r_n, where s is annotated with c. By the induction hypothesis and the fact that f is the last triple pattern in P_n, s must be a final state. By construction, M_Q^K has a transition $(h, \mathtt{type}, d + c_{r4}, s')$ where $s' \in S_f$ and is annotated with c', and hence replacing $(h, \mathtt{type}, d, s)$ by $(h, \mathtt{type}, d + c_{r4}, s')$ in r_n yields a run for P_{n+1}.

On the other hand, if P_{n+1} is produced by applying rule 4 to $f = (c, \mathtt{type}, W_m)$ in P_n, where W_m is a variable and where the matching transition in r_n is $(s_0, \mathtt{type}^-, d, h)$ (where, by the induction hypothesis and the fact that f must be the first triple pattern in P_n, we have that s_0 must be an initial state), then $(c', \mathtt{type}, W_m)$ is in P_{n+1} and there is a transition $(s_0', \mathtt{type}^-, d + c_{r4}, h)$, where s_0' is annotated with c', in M_Q^K.

Rule 5: Suppose that P_{n+1} is produced by applying rule 5 to the triple $(a, \mathtt{dom}, c) \in$

K and the triple pattern f in P_n. Because applying relaxation requires that the variables between pairs of triple patterns remain the same, it must be the case that f is of the form (W, a, z), where $z = \theta(Y)$ or $z = v_n$ (respectively $z = \theta(X)$ or $z = v_0$), and W is a variable. Hence, f must be matched by a transition t which leads to a final state (respectively starts from an initial state) in r_n; thus we have $t = (h, a, d, s)$ (respectively (s_0, a^-, d, h)) where $s \in S_f$ (respectively $s_0 \in S_0$). From the application of rule 5, we also have that f is replaced by $(W, \texttt{type}, c)$ in P_{n+1}. By construction, M_Q^K has a transition $(h, \texttt{type}, d + c_{r5}, s')$ (respectively $(s'_0, \texttt{type}^-, d + c_{r5}, h)$) where $s' \in S_f$ (respectively $s'_0 \in S_0$) and is annotated with c, and hence replacing (h, a, d, s) (respectively (s_0, a^-, d, h)) by $(h, \texttt{type}, d+c_{r5}, s')$ (respectively $(s'_0, \texttt{type}^-, d+c_{r5}, h)$) in r_n yields a run for P_{n+1}.

Rule 6: Suppose that P_{n+1} is produced by applying rule 6 to the triple $(a, \texttt{range}, c) \in K$ and the triple pattern f in P_n. Because applying relaxation requires that the variables between pairs of triple patterns remain the same, it must be the case that f is of the form (z, a, W), where $z = \theta(X)$ or $z = v_0$ (respectively $z = \theta(Y)$ or $z = v_n$), and W is a variable. Hence, f must be matched by a transition t which starts from an initial state (respectively leads to a final state) in r_n; thus we have $t = (s_0, a, d, h)$ (respectively (h, a^-, d, s)) where $s_0 \in S_0$ (respectively $s \in S_f$). From the application of rule 6, we also have that f is replaced by $(W, \texttt{type}, c)$ in P_{n+1}. By construction, M_Q^K has a transition $(s'_0, \texttt{type}^-, d + c_{r6}, h)$ (respectively $h, \texttt{type}, d + c_{r6}, s')$ where $s'_0 \in S_0$ (respectively $s' \in S_f$) and is annotated with c, and hence replacing (s_0, a, d, h) (respectively (h, a^-, d, s)) by $(s'_0, \texttt{type}^-, d + c_{r6}, h)$ (respectively $h, \texttt{type}, d + c_{r6}, s')$ in r_n yields a run for P_{n+1}.

Thus, we have shown that, in all cases, there is a run in M_Q^K corresponding to T_p. By the construction of H from M_Q^K and G, if we have the run $(s_0, l_1, c_1, s_1), \ldots, (s_{n-1}, l_n, c_n, s_n)$ in M_Q^K and the semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G, we have a run $r = ((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ in H, where $s_0 \in S_0$ and $s_n \in S_f$.

($\Leftarrow$) Let r be a run $((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ in H, where $s_0 \in S_0$ and $s_n \in S_f$. By the construction of H from M_Q^K and G, there must be a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G and a run $(s_0, l_1, c_1, s_1), \ldots, (s_{n-1}, l_n, c_n, s_n)$ in M_Q^K. Let T_p be a triple form of p.

We know that there is a run from $s_0 \in S_0$ to $s_n \in S_f$ in M_Q^K corresponding to

T_p. By the construction of M_Q^K, we also know that the transitions added to the transition relation τ correspond to the relaxation operations induced by the rules in Figure 3.6. Thus, each transition in any run in M_Q^K corresponds to either a transition in M_R or a relaxation of one of the transitions in M_R. By definition, we also know that every run in M_R corresponds to an acceptance of a sequence of labels $q \in L(R)$.

We therefore need to show that every run in M_Q^K corresponds to a sequence of direct relaxations of the triple form of some sequence of labels $q \in L(R)$; i.e. that a run in M_Q^K corresponds to T_p.

For the purposes of the proof, we assume that each transition in M_Q^K is assigned a *derivation number*. Each transition in M_R is assigned a derivation number of zero. Then, each application of rule 2, 4, 5 or 6 will result in a new transition t', derived from an existing transition t, where the derivation number of t' is the derivation number of t plus one. The *derivation length* of a run r is then the sum of the derivation numbers of the transitions comprising r. The proof proceeds by induction on the derivation length of a run.

Basis: For the base case, we consider runs having a derivation length of zero; i.e. runs only containing transitions with a derivation number of zero, which are present in M_R. Thus, $T_q = P_0 = T_p$ (no relaxation has occurred), and every run in M_R corresponds to T_q, as in Definition 3.10.

Induction: For the inductive step, assume that there is an $n \geq 0$ such that, for all $m \leq n$, if m is the derivation length of a run r in M_Q^K, then r corresponds to a sequence of m direct relaxations which yields T_p from T_q.

Now let r_{n+1} be a run in M_Q^K with derivation length $n+1$. We need to show how r_{n+1} corresponds to a triple form representing a sequence of $n+1$ direct relaxations, given by $T_q = P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n \preceq_R P_{n+1} = T_p$.

Since r_{n+1} has derivation length $n+1$, there must be a transition t in r_{n+1} with a non-zero derivation number λ. Below, we consider each rule that may have given rise to t in M_Q^K. In all cases, d indicates the cost of the transition.

Rule 2: Suppose that $t = (h, b, d, s)$. Because t was added using rule 2, there must be a transition $t' = (h, a, d - c_{r2}, s)$ in M_Q^K for some $(a, \texttt{sp}, b) \in K$, with a derivation number of $\lambda - 1$. Replacing t in r_{n+1} by t' gives rise to a run r_n with a derivation length of n. From the induction hypothesis, there is a sequence of n direct

relaxations $P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$ and a triple pattern $f = (W_{m-1}, a, W_m)$ in P_n matched by t'. Applying rule 2 to f will give rise to a triple form P_{n+1} corresponding to a sequence of $n+1$ direct relaxations having been applied to q.

By an analogous process where $t = (h, b^-, d, s)$, we can show that its matching triple pattern (W_m, b, W_{m-1}) is in P_{n+1}.

Rule 4: Suppose that $t = (h, \texttt{type}, d, s'_n)$ where $s'_n \in S_f$ and is annotated with c'. Because t was added using rule 4, there must be a transition $t' = (h, \texttt{type}, d - c_{r4}, s_n)$ in M_Q^K for some $(c, \texttt{sc}, c') \in K$, with a derivation number of $\lambda - 1$, and where $s_n \in S_f$ and is annotated with c. Replacing t in r_{n+1} by t' gives rise to a run r_n with a derivation length of n. From the induction hypothesis, there is a sequence of n direct relaxations $P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$ and a triple pattern $f = (W_m, \texttt{type}, c)$ in P_n matched by t'. Applying rule 4 to f will give rise to a triple form P_{n+1} corresponding to a sequence of $n+1$ direct relaxations having been applied to q.

By an analogous process where $t = (s'_0, \texttt{type}^-, d, h)$, and $s'_0 \in S_0$ and is annotated with c', we can show that its matching triple pattern $(c', \texttt{type}, W_m)$, where W_m is a constant, is in P_{n+1}.

Rule 5: Suppose that $t = (h, \texttt{type}, d, s'_n)$ (respectively $(s'_0, \texttt{type}^-, d, h)$) where $s'_n \in S_f$ (respectively $s'_0 \in S_0$). Because t was added using rule 5, there must be a transition $t' = (h, a, d - c_{r5}, s_n)$ (respectively $(s_0, a^-, d - c_{r5}, h)$) in M_Q^K for some $(a, \texttt{dom}, c) \in K$, with a derivation number of $\lambda - 1$, and where $s_n \in S_f$ (respectively $s_0 \in S_0$). Replacing t in r_{n+1} by t' gives rise to a run r_n with a derivation length of n. From the induction hypothesis, there is a sequence of n direct relaxations $P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$ and a triple pattern f in P_n matched by t'. Because applying relaxation requires that the variables between pairs of triple patterns remain the same, and from the fact that f is matched by t' which leads to a final state (respectively starts from an initial state), it must be the case that f is of the form (W, a, z), where $z = \theta(Y)$ or $z = v_n$ (respectively $z = \theta(X)$ or $z = v_0$), and W is a variable. Applying rule 5 to f will give rise to a triple form P_{n+1} corresponding to a sequence of $n+1$ direct relaxations having been applied to q.

Rule 6: Suppose that $t = (s'_0, \texttt{type}^-, d, h)$ (respectively $(h, \texttt{type}, d, s'_n)$) where $s'_0 \in S_0$ (respectively $s'_n \in S_f$). Because t was added using rule 6, there must be a transition $t' = (s_0, a, d - c_{r6}, h)$ (respectively $(h, a^-, d - c_{r6}, s_n)$) in M_Q^K for some $(a, \texttt{range}, c) \in K$, with a derivation number of $\lambda - 1$, and where $s_0 \in S_0$

(respectively $s_n \in S_f$). Replacing t in r_{n+1} by t' gives rise to a run r_n with a derivation length of n. From the induction hypothesis, there is a sequence of n direct relaxations $P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$ and a triple pattern f in P_n matched by t'. Because applying relaxation requires that the variables between pairs of triple patterns remain the same, and from the fact that f is matched by t' which starts from an initial state (respectively leads to a final state), it must be the case that f is of the form (z, a, W), where $z = \theta(X)$ or $z = v_0$ (respectively $z = \theta(Y)$ or $z = v_n$), and W is a variable. Applying rule 6 to f will give rise to a triple form P_{n+1} corresponding to a sequence of $n+1$ direct relaxations having been applied to q.

Part (2): ($\Rightarrow$) Let θ be a matching mapping X to v_0 and Y to v_n in G, r be a run $((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ in H, where $s_0 \in S_0$ and $s_n \in S_f$, and let the relaxation distance of $\theta(Q)$ be given by $c_1 + \cdots + c_n$. By definition, we know that the relaxation distance of $\theta(Q)$ is the minimum relaxation distance to $\theta(Q)$ from any semipath p that r-conforms to $\theta(Q)$. By the construction of H from M_Q^K and G, there is a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G and a run $(s_0, l_1, c_1, s_1), \ldots, (s_{n-1}, l_n, c_n, s_n)$ in M_Q^K. From Part (1), we know that p r-conforms to $\theta(Q)$. Thus, we have that $c_1 + \cdots + c_n$ is the minimum relaxation distance to $\theta(Q)$ from p.

Also by definition, we have that $c_1 + \cdots + c_n$ is the minimum relaxation distance from p to $(\theta(Q), q)$ for any sequence of labels $q \in L(R)$ and that, for any such q, $c_1 + \cdots + c_n$ is the minimum cost of any sequence of direct relaxation operations which yields p from q. Thus, there can be no run of cost less than $c_1 + \cdots + c_n$ for p in H as this would contradict the fact that p is of relaxation distance $c_1 + \cdots + c_n$ from $\theta(Q)$. Hence, r is a minimum cost run over all runs from (s_0, v_0) to (s_n, v_n).

($\Leftarrow$) We assume that r is a minimum cost run in H over all runs from some (s_0, v_0) to some (s_n, v_n), where $s_0 \in S_0$ and $s_n \in S_f$. This means, by construction, that the minimum cost of any run is $c_1 + \cdots + c_n$. Also by the construction of H, there is a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G. But Part (1) shows us that p r-conforms to $\theta(Q)$. Thus, the relaxation distance of $\theta(Q)$ is $c_1 + \cdots + c_n$.

□

For the rest of this section, we consider the complexity of returning relaxed answers. Proposition 4.3 shows the upper bound for the size of the relaxed automaton,

M_Q^K, and Proposition 4.4 shows that the relaxed answer of a single-conjunct query Q on the closure of a graph G can be computed in time that is polynomial in the size of Q, G and the ontology K. We subsequently show that the data and query complexity is polynomial in the size of G and the regular expression of Q respectively, and that the space complexity is dominated by the space required by the product graph.

Proposition 4.3. *Let Q be a query comprising a single conjunct (X, R, Y), $M_Q^K = (S', \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}, \tau, S_0, S_f, \xi)$ be the relaxed automaton for Q and ontology $K = \mathtt{extRed}(K)$, where $K = (V_K, E_K)$. M_Q^K has at most $2|R|(|V_K| + 1)$ states and $2|R|^2(|E_K||V_K| + 8|E_K| + 8)$ transitions.*

Proof. From [2] and [32], we have that $M_R = (S, \Sigma \cup \{\mathtt{type}\}, \delta, s_0, S_f, \xi)$ contains at most $2|R|$ states and $4|R|$ transitions. The subsequent removal of ϵ-transitions as described in [32] may result in at most $16|R|^2$ transitions. We recall that M_Q^K initially consists of all states S in M_R and all these transitions.

We know that each node in the set V_K is either a class node or a property node in K. From the construction of M_Q^K — in particular, by the application of rules 4, 5 and 6 — we can see that, for each class node in V_K, at most *one* new state, corresponding to the class node, is added for any existing state s, provided that $s \in S_0$ or $s \in S_f$. Hence, we can see that no more than $|V_K|$ new states may be added for each of the original states in S, which results in at most $2|R||V_K|$ new states in total. Thus, M_Q^K has at most $2|R|(|V_K| + 1)$ states.

Since there are at most $|E_K|$ edges in K with label $\mathtt{sp}$, rule 2 adds at most $16|R|^2|E_K|$ transitions to M_Q^K. Rules 4, 5 and 6 can collectively be applied no more than $|E_K|$ times. Each application results, in the worst case, in $|R|$ transitions being added for each of the $2|R||V_K|$ new, cloned states, giving rise to at most $2|R|^2|V_K||E_K|$ transitions. Thus, overall M_Q^K has at most $2|R|^2(|E_K||V_K|+8|E_K|+8)$ transitions.

□

Proposition 4.4. *Let $K = (V_K, E_K)$ be an ontology such that $k = \mathtt{extRed}(K)$, $G = (V_G, E_G, \Sigma)$ be a graph such that $G = closure_K(G)$, and Q be a single-conjunct query using regular expression R over alphabet $\Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}$. The*

relaxed answer of Q on G can be found in time $O(|R|^2|V_K|^2|V_G|(|R||E_K||E_G| + |V_G|\log(|R||V_K||V_G|)))$.

Proof. Let M_Q^K be the relaxed automaton constructed from R and K, and H be the product automaton constructed from M_Q^K and G. Proposition 4.2 shows that traversing H yields all relaxed answers to Q. Proposition 4.3 tells us that M_Q^K has at most $2|R|(|V_K| + 1)$ states and $2|R|^2(|E_K||V_K| + 8|E_K| + 8)$ transitions. Therefore H has at most $2|R||V_G|(|V_K| + 1)$ nodes and $2|R|^2|E_G|(|E_K||V_K| + 8|E_K| + 8)$ edges. If we assume that H is sparse (which is highly likely), then running Dijkstra's algorithm on each node of a graph with node set N and edge set A can be done in time $O(|N||A| + |N|^2 \log |N|)$. So, for graph H, the combined time complexity is $O(|R|^3|E_K||V_K|^2|V_G||E_G| + |R|^2|V_K|^2|V_G|^2 \log(|R||V_K||V_G|))$ which is equal to $O(|R|^2|V_K|^2|V_G|(|R||E_K||E_G| + |V_G|\log(|R||V_K||V_G|)))$.

□

As a corollary, it is easy to see that the data complexity is $O(|V_K|^2|V_G|(|E_K| |E_G| + |V_G|\log(|V_K||V_G|)))$ and the query complexity is $O(|R|^3)$. The space complexity is dominated by the space requirements of H given in the proof above.

Further, we observe that the data complexity results are consistent with the results in our empirical evaluation, which is discussed in detail in our chapter on query performance analysis, in Section 7.1.4.

4.4 Concluding remarks

In this chapter, we provided proofs of correctness and complexity for the constructs and algorithms introduced in Chapter 3.

In the next chapter, we describe a prototype system, called *ApproxRelax*, which is an implementation of query approximation and query relaxation, and present a qualitative case study demonstrating the application of query approximation and query relaxation in the domain of lifelong learning.

CHAPTER 5

The *ApproxRelax* System and a Case Study

In the previous two chapters, we presented the theoretical background comprising the data model and query language underlying this research, and provided detailed descriptions and full proofs of correctness for the constructs and algorithms required to evaluate approximated and relaxed queries.

In this chapter, we discuss our first implementation of approximated and relaxed querying in the form of a prototype system called *ApproxRelax*, which we have presented previously in [113]. *ApproxRelax* is a precursor to our final implementation, called *Omega*, and full details of the latter are described in Chapter 6. Here, we restrict our focus to the user-facing features of *ApproxRelax*, and present a qualitative case study showing how *ApproxRelax* overcomes problems in a previous system in the domain of lifelong learning, entitled *L4All*, by providing more flexible querying capabilities, resulting in answers of greater relevance being returned to the user.

The rest of this chapter is structured as follows: Section 5.1 presents the *L4All* system, which aims to support lifelong learners in planning and reflecting on their learning. As part of its functionality, it allows users to search over information relating to the educational and work experiences of other members of their lifelong learning network. In Section 5.2 we describe *ApproxRelax* by initially providing some contextual example data and queries, and then proceeding on to describing,

from a user-interface perspective, how exact, approximated and relaxed queries may be posed and evaluated, and how answers are displayed to the user. We compare the *ApproxRelax* and *L4All* systems in Section 5.3, and present the conclusions of an evaluation undertaken with two lifelong learning practitioners. We end the chapter with some concluding remarks in Section 5.4.

5.1 Case study: Lifelong Learning

Assisting and understanding the requirements of lifelong learners has resulted in research into the role played by online support systems for providing careers guidance [25] and learner-oriented models of the delivery of learning capabilities [80, 81].

The *L4All* ("LifeLong Learning in London for All") system aims to support lifelong learners in exploring learning opportunities and in planning and reflecting on their learning [26, 27] by providing a full record encompassing their entire adult learning career, as opposed to focussing solely on a single educational period.

The *L4All* system allows users to create and maintain a chronological record of their learning, work and personal episodes — their *timelines*. Users' timelines are stored in the form of RDF/S, through the Jena framework. The system's interface provides screens for the user to enter their personal details, to create and maintain their timeline, and to search over the timelines of other users, based on a variety of search criteria. This sharing of timelines exposes future learning and work possibilities that may otherwise not have been considered, positioning successful learners as role models to inspire confidence and a sense of opportunity.

There are some 20 types of episode supported by the system, each belonging to one of four categories: Educational, Occupational, Personal or Other. These classifications are drawn from standard U.K. occupational and educational taxonomies. In particular, all Educational episodes are classified by a subject from the Labour Force Survey Subject of Degree (SBJ) classification and a qualification level from the National Qualifications Framework (NQF). All work and voluntary Occupational episodes are classified by an industry sector from the Standard Industrial Classification (SIC) and an occupation/position from the Standard Occupational Classification (SOC). We refer the reader to the Labour Force Survey User Guide

for details of these standards[1].

A key aim of the *L4All* system is to allow learners to search over the timeline data, and to identify possible choices for their own future learning and professional development by seeing what others with a similar background have gone on to do. In particular, [128, 130] describe a facility that is provided by the system for searching for "People like me". This facility allows the user to specify which parts of their own timeline should be matched with other users' timelines, by selecting which types of episodes should be matched, as well as the similarity metric to be applied (one of Jaccard Similarity, Dice Similarity, Euclidean Distance, and Needleman-Wunsch Distance).

In order for the system to be able apply these similarity metrics, the users' timelines are encoded as token-based strings. In particular, each episode is encoded as a single token comprising a 2-letter unique identifier denoting the category of the episode, followed by up to two 4-digit codes classifying the episode according to the four levels of the taxonomies relevant for this type of episode (which may be 0, 1 or 2 taxonomies). The information about episodes' start and end dates is ignored and only the relative position of episodes is captured. Filters are applied to the string of tokens to remove those types of episode that should not be considered in the current search, and for limiting the depth of their classification to be considered in the matching process. We refer the reader to [130] for more details of the timeline encoding and for a detailed comparison of the different similarity metrics considered for incorporation within the system.

Once the user's definition of "People like me" has been specified, the system returns a list of all the candidate timelines, ranked by their normalised similarity. The user can then select one of these timelines to visualise in detail. Episodes within the selected timeline are visible, and the user can click on any of these to expose its details and to explore it further.

An evaluation study conducted on the "People like me" facility identified the need for a more contextualised usage of timeline similarity matching, which explicitly identifies possible future learning and professional possibilities for the user (see [128, 129]). Follow-on work explored a more contextualised usage of timeline similarity

[1] `http://www.ons.gov.uk/ons/guide-method/method-quality/specific/labour-market/labour-market-statistics/volume-3-2015.pdf`

matching which uses just one similarity metric (hence removing this element of choice, and potential difficulty, for the user) and which explicitly shows the episodes of the selected timeline that have no match within the user's timeline and thus represent episodes the user may be inspired to explore further for their own learning and career development [129].

This new facility was termed "What Next" and it uses just the Needleman-Wunsch similarity metric. Using this metric, the "What Next" facility considers the distance between two strings of tokens x and y to be the minimum cost of transforming x to y by a series of insert or delete operations (substitution operations are not considered). The system builds a cost matrix incrementally by constructing a cost value for each pair of tokens x_i and y_j from each string. A summary of the relevant timelines as found by the system is presented to the user in the form of a list, ordered by their similarity to the user's timeline with respect to the specified parameters and summarised by the name and age of their owners and a short description.

Magoulas, Poulovassilis and van Labeke [129] report on the results of two evaluation sessions that were held with mature learners at Birkbeck, University of London and at the College of North East London assessing the *L4All* system. Although the learners found the system very useful, three issues were identified, centred around the way in which users specify their search queries and with the ranking of the search results:

- The top-ranked timelines in the list returned by "What Next" will be timelines that are most similar to the user's own timeline. These timelines may in practice offer few suggestions of episodes for the user's future development.

- The level of detail that should be used in episode classification for the purpose of applying the similarity metric: selection of different classification levels by the user will give rise to different similarity values, and therefore different possible timeline alignments.

- Using the distance matrix computed by the Needleman-Wunsch algorithm to generate the episode alignments may generate several possible alignments between two timelines. Determining the 'best' one in a given context is not easy, as subjective factors relating to the user's own definition of 'relevance'

have to be taken into account. The current implementation in *L4All* always makes the same default choice of alignment, whereby the "common ground" of matching episodes in the two timelines is selected to be as late as possible within the user's timeline.

The above problems arise because the "What Next" facility is rather *rigid*: it uses the whole of the user's timeline; it offers just one similarity metric over the timeline data; it allows just a single level of detail to be applied to the classifications of the selected categories of episode for the similarity matching; and the similarity matching is applied to all episodes of these categories in the user's timeline. Thus, there is limited flexibility for users to formulate their precise requirements for the timeline search and to explore alternative formulations of selected parts of their query.

These observations motivated the development of the *ApproxRelax* prototype, that implements our query approximation and query relaxation techniques and supports more flexible user querying of timeline data.

5.2 The *ApproxRelax* system

The *ApproxRelax* prototype system is a web application that supports flexible querying — through the provision of query approximation and query relaxation — over timeline data.

For example, Figures 5.1, 5.2 and 5.3 illustrate fragments of data and metadata relating to three users' timelines (Dan, Liz and Al). There are several types of episode, e.g. *UniversityEpisode* and *WorkEpisode*. Associated with each type of episode are several properties e.g. *qualif[ication]* and *job*. Episodes are ordered according to their start date — as indicated by edges labelled *next* (for simplicity, the episodes' start and end dates are not shown). If two episodes have the same start date, the one that ends earlier is considered to precede the other. If two episodes have identical start and end dates, an arbitrary one is chosen as being the earlier one. An edge labelled *prereq* from one episode to another is an annotation created by the timeline's owner indicating that they consider that undertaking an earlier episode was necessary in order for them to be able to proceed to or achieve a later episode.

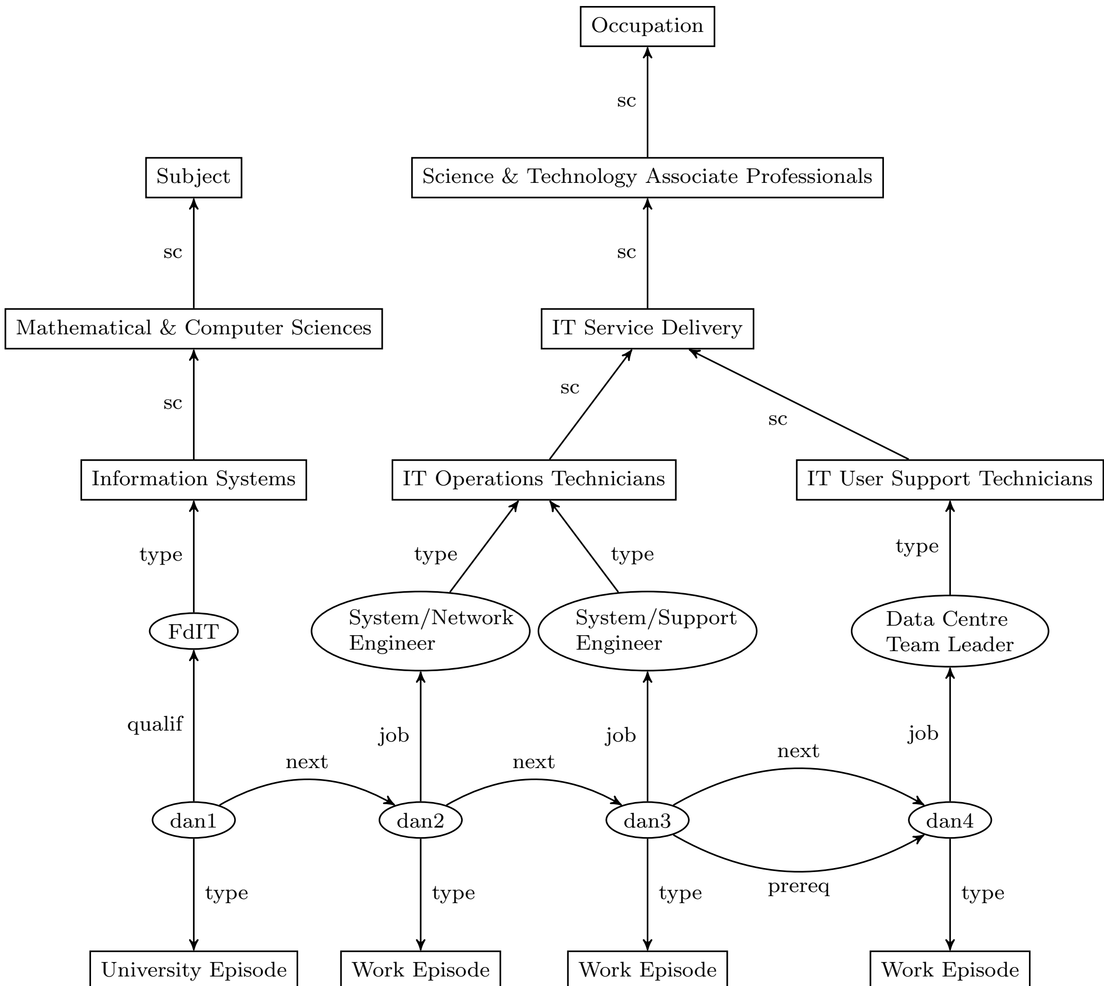

Figure 5.1: A fragment of Dan's timeline data and metadata.

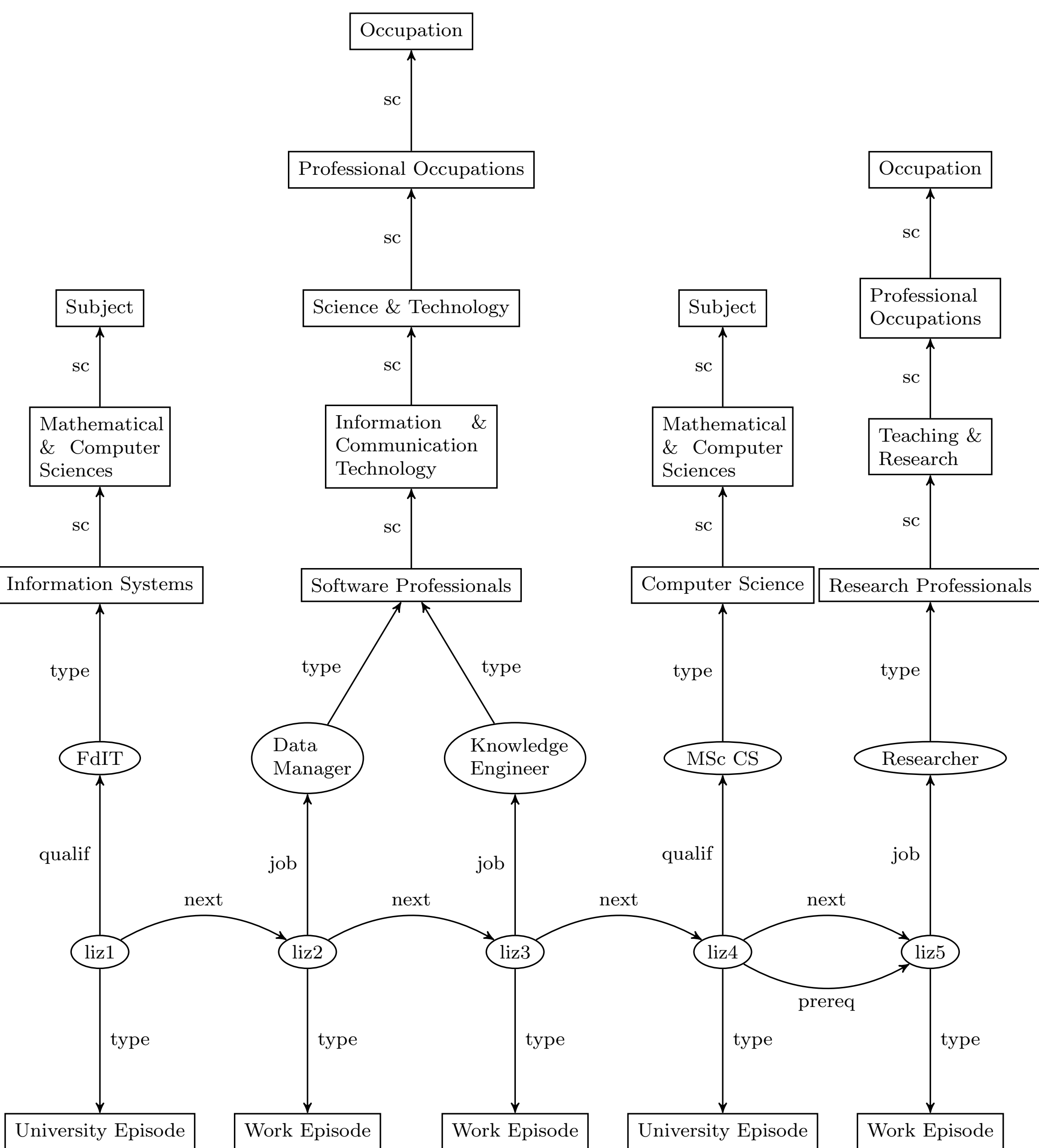

Figure 5.2: A fragment of Liz's timeline data and metadata.

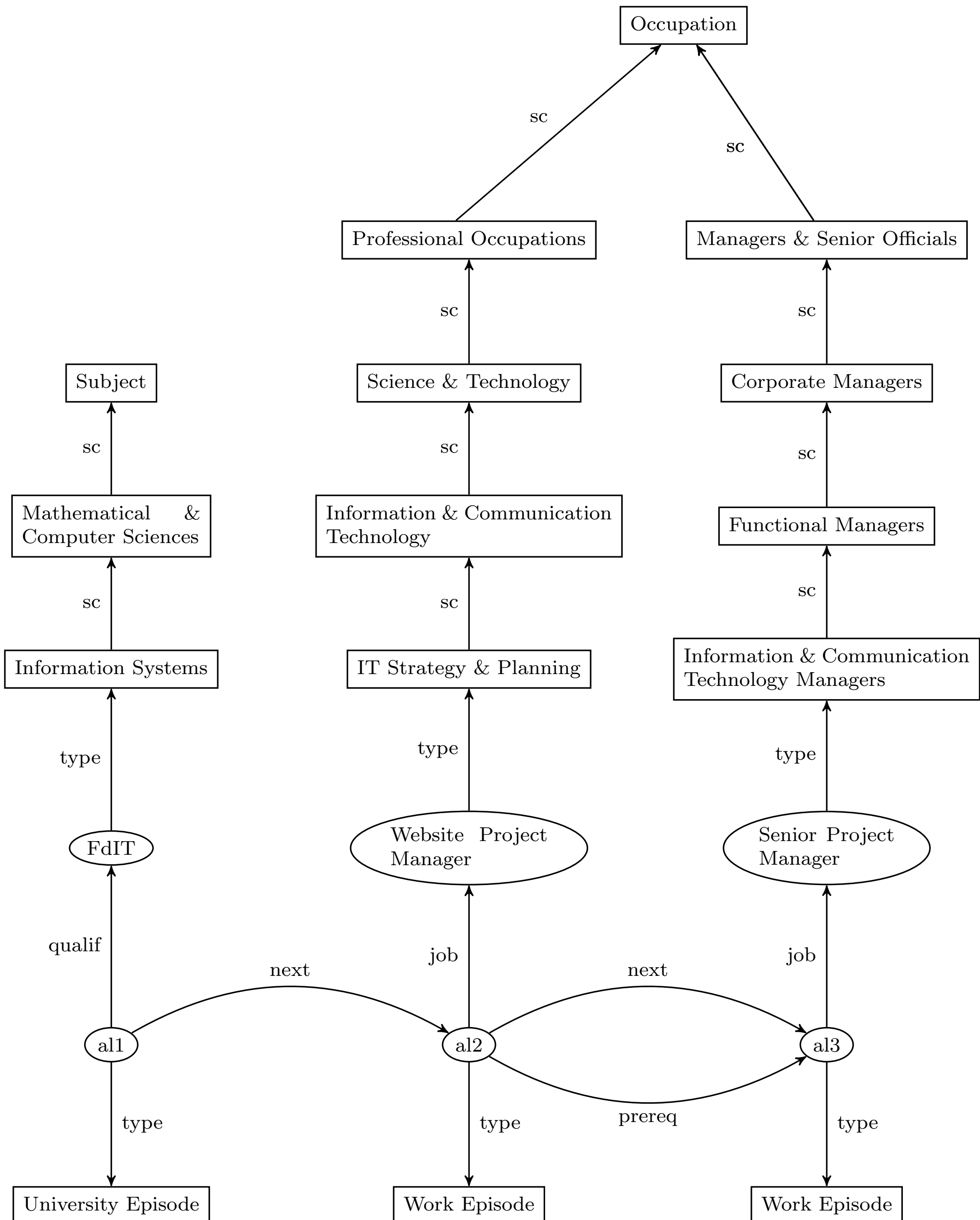

Figure 5.3: A fragment of Al's timeline data and metadata.

ApproxRelax supports query approximation by allowing edge labels to be inserted or deleted only (thus, there is no provision for the substitution of edge labels), each with the same configurable cost. Query relaxation is supported by allowing a class label to be replaced by that of a superclass or a property label by that of a superproperty, also each with the same configurable cost. For the remainder of this chapter, we assume that the system has been configured to only allow the insertion of an edge label, setting its cost to be 1, and the replacement of a class by its superclass, at a cost of 2.

The *ApproxRelax* prototype provides users with a graphical user interface through which they can formulate their queries. In order to facilitate ease of use, the more complex parts of the user interface are explained to the user by means of 'tooltips' and hover-over text (indicated by boxes containing a '?').

We now illustrate how *ApproxRelax* might be used by a fictitious student, whose profile corresponds to that of the users we envision using the system. Suppose Gaby is studying on the Foundation Degree in Information Technology (FdIT) at Birkbeck, University of London and she wishes to find out what possible future career choices there may be for her by seeing what other people with qualifications in Information Systems have gone on to do in their careers. Gaby opens up her browser and proceeds to the screen shown in Figure 5.4. This screen allows the user to start formulating a query by creating a query template for matching educational episodes or occupational episodes.

Gaby clicks the 'Create an educational episode' image and is presented with the screen shown in Figure 5.5. From the 'Type' drop-down menu she is able to make a choice from the Educational episode types, and she selects 'University Episode'. From the 'Subject' drop-down menu she is able to make a choice from different subject areas (as sourced from the SBJ taxonomy mentioned in Section 5.1). She selects 'Information Systems' and ticks the 'Fetch similar or related subjects?' checkbox.

As she has not yet finished constructing her query, she clicks the 'Next' button. At this point the system generates internally these query conjuncts:

```
(?A,type,UniversityEpisode)                    [C1]
RELAX(?A,qualif.type,InformationSystems)       [C2]
```

Conjunct `C1` is generated from Gaby's selection of 'University Episode' in the 'Type'

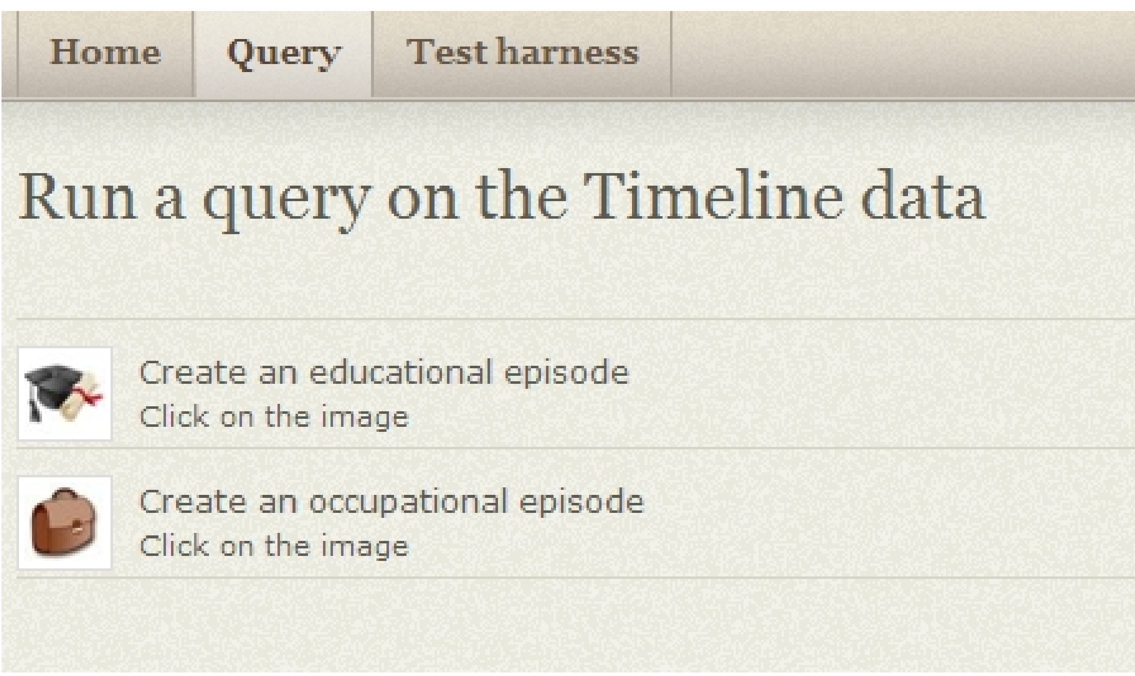

Figure 5.4: *ApproxRelax* query set-up.

drop-down. The fact that Gaby designated this episode as an educational episode and selected 'Information Systems' from the 'Subject' drop-down gives rise to conjunct `C2`. Because she ticked the 'Fetch similar or related subjects?' checkbox, this conjunct additionally has the 'RELAX' keyword applied to it by the system.

Having clicked 'Next', Gaby is presented again with the screen in Figure 5.4. Gaby now clicks the 'Create an occupational episode' image and is presented with the screen shown in Figure 5.6. As this is not the first episode of the query, there is a 'Link from previous episode' drop-down, which allows the user to specify the way in which the previously specified episode is related to the one currently being specified. The possible choices here (in the current prototype) are *next*, *next+*, *prereq* and *prereq+*, which are displayed in the drop-down using more user-friendly descriptions: 'next episode', 'next or subsequent episode', 'direct prerequisite', and 'direct or indirect prerequisite'.

Gaby selects the 'next episode' option and ticks the 'Flexible matching of the link between this episode and the previous one?' checkbox. From the 'Type' drop-down menu she is able to make a choice from the Occupational episode types, and she

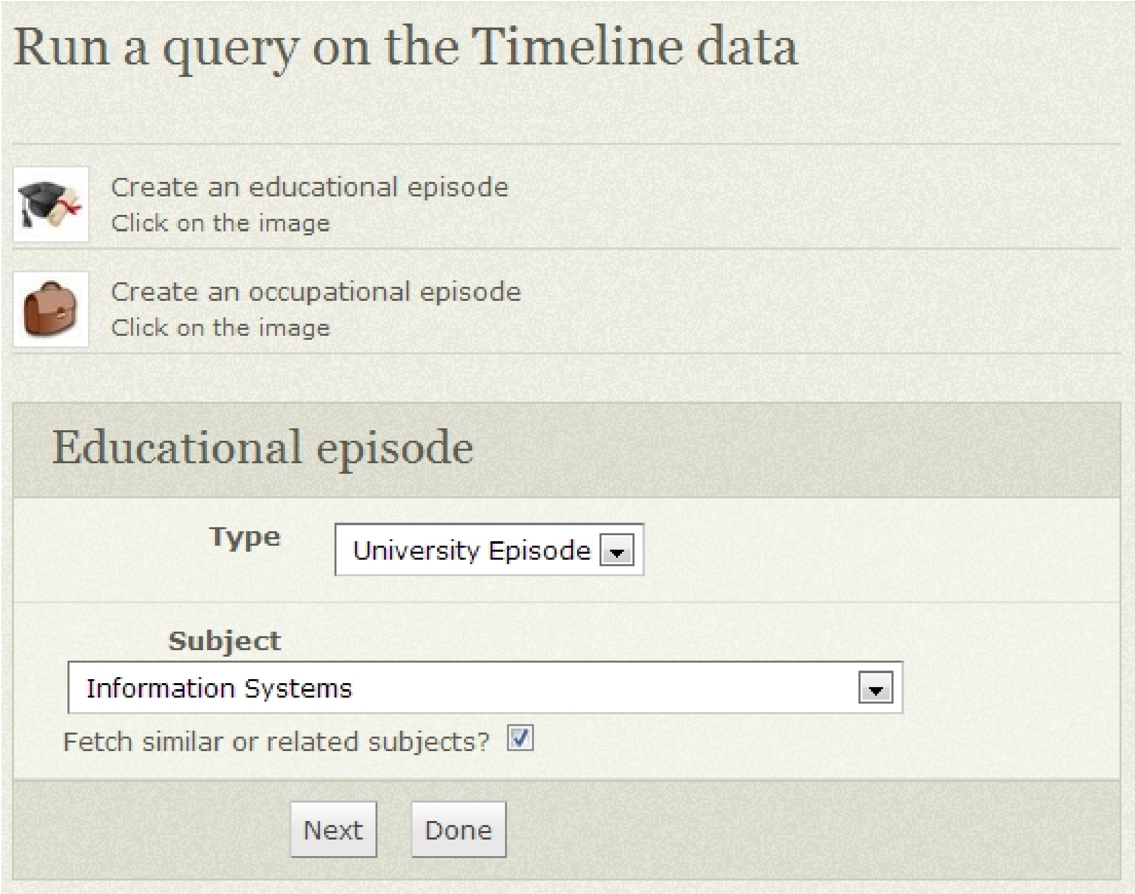

Figure 5.5: Constructing an Educational episode query template.

selects ‘Work Episode’. From the ‘Job’ drop-down menu she is able to make a choice from different jobs (as sourced from the SOC taxonomy mentioned in Section 5.1). Gaby selects ‘Software Professionals’ and ticks the ‘Fetch similar or related occupations?’ checkbox. She has now finished constructing her query and clicks the ‘Done’ button. At this point the system generates internally the following query conjuncts:

```
APPROX(?A,next,?B)                        [C3]
(?B,type,WorkEpisode)                     [C4]
RELAX(?B,job.type,SoftwareProfessionals) [C5]
```

Conjunct `C3` links the query episode set up previously (denoted by `?A`) to this second episode (denoted by `?B`). It contains the selection made by Gaby in the ‘Link from previous episode’ drop-down. Additionally, as Gaby has ticked the ‘Flexible matching ... ’ checkbox, `C3` has the ‘APPROX’ keyword applied to it. Conjunct `C4` is generated from Gaby’s selection of ‘Work Episode’ in the ‘Type’ drop-down. The fact that Gaby designated this episode as an occupational episode, and selected ‘Software Professionals’ from the ‘Job’ drop-down gives rise to conjunct `C5`. Since Gaby ticked the ‘Fetch similar or related occupations?’ checkbox, `C5` additionally has the ‘RELAX’ keyword applied it. The system provides the facility for the user to view the details of previously-constructed query templates, as may be seen on the right-hand side of Figure 5.6; more about this facility is detailed in the next paragraph.

The next screen that Gaby is presented with is shown in Figure 5.7. It allows the user to view at a glance the episode query templates making up their query (this is identical to the facility seen on the right-hand side of Figure 5.6) and allows the user to view previously-constructed query templates whilst in the process of creating their query. The type of each episode (educational or occupational) is immediately clear, as denoted by the image. In Figure 5.7, the second image has been clicked (its number is highlighted in red) and the information pertaining to Gaby’s second query template is displayed, namely, that the link from the previous episode is via the ‘next episode’ link and has been approximated, that it is a ‘Work Episode’ and that the job is ‘Software Professionals’, which has been relaxed. Gaby can now click on the ‘cog’ image (which has the relevant tooltip) to execute her query.

Gaby is now presented with the screen shown in Figure 5.8, which displays

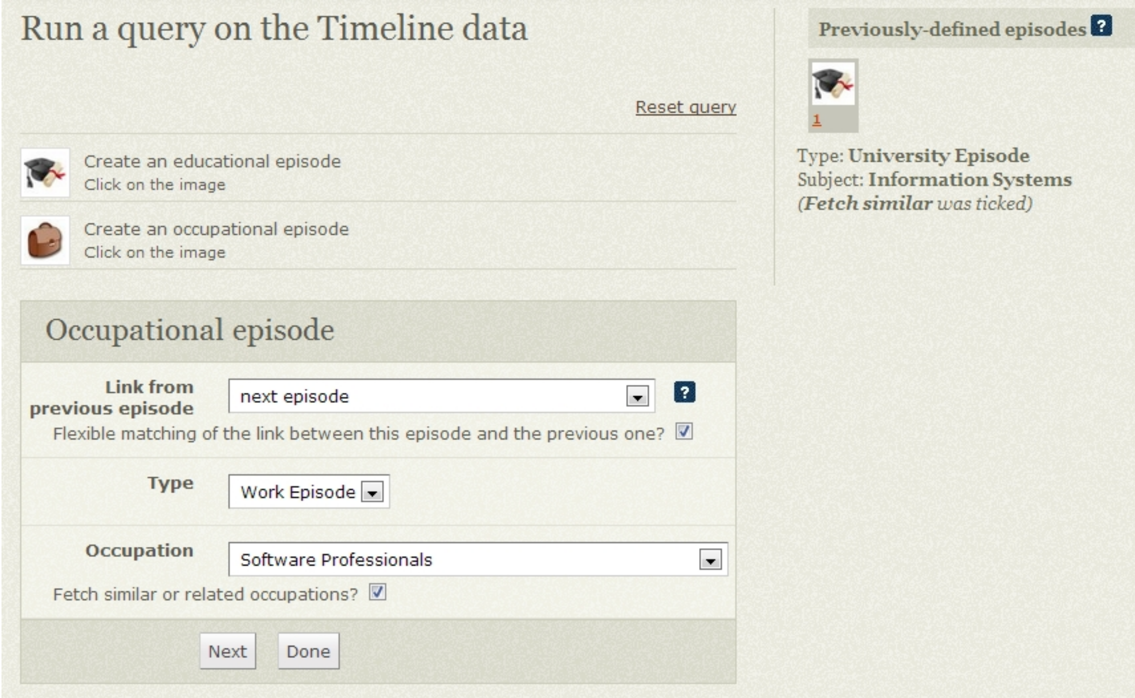

Figure 5.6: Constructing an Occupational episode query template.

query results ranked in order of increasing distance from the non-approximated, non-relaxed version of her query.

The query results are derived incrementally and one screenful at a time is displayed to the user, on their request. For each result, an avatar representing the timeline's owner is displayed, as well as their name, the episode in their timeline which matches the last episode query template of the user's query, the distance at which this result has been retrieved, and an automatically generated summary of the timeline's owner and contents of their timeline. The latter description is displayed to give the user an overview of the matching timeline so that they can decide whether they wish to explore it in more detail.

At present, this is as far as the *ApproxRelax* prototype goes in terms of displaying query results. Missing from the current *ApproxRelax* prototype are abilities for querying additional classifications according to the NQF (for Educational episodes) and SIC (for Occupational episodes) taxonomies mentioned in Section 5.1, and for formulating query templates for Personal and Other episode types; this could be

Figure 5.7: Viewing episode query templates.

Results

Liz
Distance is 0; Liz Episode 2: 'UK Data Manager'

Worked in the research industry as an extracts analyst; obtained the Birkbeck Foundation Degree in IT; worked as a UK data manager and knowledge engineer; obtained the Birkbeck MSc in Computer Science; currently works as a researcher

Liz
Distance is 1; Liz Episode 3: 'Knowledge Engineer'

Worked in the research industry as an extracts analyst; obtained the Birkbeck Foundation Degree in IT; worked as a UK data manager and knowledge engineer; obtained the Birkbeck MSc in Computer Science; currently works as a researcher

Al
Distance is 2; Al Episode 2: 'Website Project Manager'

Obtained A-levels in the physical sciences; obtained a Diploma in Biochemistry; worked as a website manager; obtained the Birkbeck Foundation Degree in IT; currently works as a senior project manager

Liz
Distance is 8; Liz Episode 5: 'Researcher'

Worked in the research industry as an extracts analyst; obtained the Birkbeck Foundation Degree in IT; worked as a UK data manager and knowledge engineer; obtained the Birkbeck MSc in Computer Science; currently works as a researcher

Dan
Distance is 8; Dan Episode 2: 'System/Network Engineer'

Obtained A-levels in the physical sciences; worked as a museum curator; obtained a diploma in Computer Science whilst working as a courier; obtained the Birkbeck Foundation Degree in IT; worked as a system/network engineer; currently works as a support team leader for a data centre

Figure 5.8: Viewing the query results.

part of future work.

The query answers underlying the GUI view of Figure 5.8 are shown in Table 5.1, where the answers produced for each individual conjunct are shown in the first five columns. We recall that the query is as follows:

```
(?A,?B) <-
  (?A,type,UniversityEpisode),                 [C1]
  RELAX(?A,qualif.type,InformationSystems),    [C2]
  APPROX(?A,next,?B),                          [C3]
  (?B,type,WorkEpisode),                       [C4]
  RELAX(?B,job.type,SoftwareProfessionals)     [C5]
```

In the second, third and fifth columns of Table 5.1, the edit or relaxation distance of the answers is shown as the value of the attribute 'D'. The final column shows the overall query answers, in order of non-decreasing total distance. The conjunct answer tuples that contribute to the final answer tuples are italicised and are subscripted with the ordering of the final answer tuple (i.e. subscript i denotes the ith final answer tuple).

5.3 Comparison with *L4All*'s "What Next"

A fundamental difference between *L4All*'s "What Next" facilities and the *ApproxRelax* prototype is that with *ApproxRelax* users can pose search queries that are different from their own timeline, e.g. where some episodes in their timeline are not included in the search query, or if included they need not be approximated, or where there are episodes in the search query not related to their own timeline. The problem of the top-ranked timelines in *L4All* being very similar to the user's own timeline is avoided by not requiring that *all* of a user's timeline is matched against the timeline data.

Relaxation in *ApproxRelax* is also more flexible than in *L4All*: in *L4All* the information about each episode is encoded as a single token and the same depth of classification is applied to all types of episodes for similarity matching, whereas in *ApproxRelax* each episode query template results in several query conjuncts each of which can be individually approximated or relaxed.

?A	?A, D	?A, ?B, D	?B	?B, D	?A, ?B, D
$dan1_{579}$	$dan1, 0_{579}$	$dan1, dan2, 0_5$	$dan2_5$	$liz2, 0_1$	$liz1, liz2, 0$
$liz1_{126}$	$liz1, 0_{126}$	dan2, dan3, 0	$dan3_7$	$liz3, 0_2$	$liz1, liz3, 1$
$liz4_4$	$al1, 0_{38}$	dan3, dan4,0	$dan4_9$	$al2, 2_3$	$al1, al2, 2$
$al1_{38}$	$liz4, 2_4$	$liz1, liz2, 0_1$	$liz2_1$	$liz5, 6_{46}$	$liz4, liz5, 8$
		liz2, liz3 ,0	$liz3_2$	$al3, 8_8$	$dan1, dan2, 8$
		liz3, liz4, 0	$liz5_{46}$	$dan2, 8_5$	$liz1, liz5, 9$
		$liz4, liz5, 0_4$	$al2_3$	$dan3, 8_7$	$dan1, dan3, 9$
		$al1, al2, 0_3$	$al3_8$	$dan4, 8_9$	$al1, al3, 9$
		al2, al3, 0			$dan1, dan4, 10$
		$dan1, dan3, 1_7$			
		dan2, dan4, 1			
		$liz1, liz3, 1_2$			
		liz2, liz4, 1			
		liz3, liz5, 1			
		$al1, al3, 1_8$			
		$dan1, dan4, 2_9$			
		liz1, liz4, 2			
		liz2, liz5, 2			
		$liz1, liz5, 3_6$			

Table 5.1: Evaluation of the query

As demonstrated in the example of the previous section, each episode in *ApproxRelax* is encoded within a query by a conjunct for the type of the episode and up to two conjuncts for episode classification. A consequence of this finer level of representation is that each classification can be relaxed independently, and that answers resulting from fewer relaxations will automatically be ranked higher by the system. For example, if the depth of episode classification is set to 2 in *L4All*, then all classification taxonomies are relaxed to that level and all answers are considered to be equally relevant to the user (modulo the similarity matching). This corresponds to answers in *ApproxRelax* of distance up to 4 times the cost of applying the relaxation operations (configured in the evaluation to be 2) from the original query. In contrast, *ApproxRelax* will return answers ranked by their distance, so that answers resulting from fewer relaxations will be ranked higher than those using more.

An evaluation of *ApproxRelax* compared with *L4All* was undertaken with two lifelong learning practitioners (see [113] for details). These two practitioners reported that they found it "much more useful" to be able to explicitly set up a search query in *ApproxRelax* rather than using the built-in similarity matching of *L4All* based on the user's whole timeline. They also found it helpful that *ApproxRelax* allows users to specify what kind of episode they are looking for, for inspiration; for example, as in conjunct `C5` above.

Returning to the other issues identified at the end of Section 5.1, the problem of the user having to decide on the level of classification for episode comparisons is avoided in *ApproxRelax* because relaxation of episode types is performed automatically by the system, with timelines containing episodes matching the user's query in more detail being ranked higher than those matching at higher levels of classification (everything else being equal). Thus, there is no need for the user to have detailed knowledge of the classification hierarchy and to make a choice of which level should be applied for episode comparisons.

The problem of finding the 'best' alignment will remain difficult. However, because each episode query template is represented by several query conjuncts in *ApproxRelax*, rather than as a single token in *L4All*, similarity is more finely measured. Furthermore, the user can specify in *ApproxRelax* whether or not they would like particular episodes to be matched exactly.

5.4 Concluding remarks

Facilitating the collaborative formulation of learning goals and career aspirations has the potential to enhance learners' engagement with the lifelong learning process [25, 80, 81]. The *L4All* system offers similarity matching over learners' timelines in order to identify possible choices for future learning and professional development. Here, we have explored how supporting query approximation and query relaxation can provide greater flexibility in users' querying of heterogeneous timeline data.

We have described a prototype system encompassing query approximation and query relaxation called *ApproxRelax* which provides users with a graphical facility for incrementally constructing search queries over learners' timelines. We have described how the system can be used to construct conjunctive regular path queries over timeline data and metadata, and to allow approximation or relaxation to be applied to selected parts of the user's query, showing how the system is able to return results in ranked order of their distance from the original query.

In a small evaluation (see [113]), *ApproxRelax* returned more results deemed relevant by two lifelong learning practitioners than *L4All*. As a result, there may be an opportunity for further work in this area, such as the extension of the *ApproxRelax* system by undertaking engineering enhancements to render it usable in a multi-user environment under heavy transactional loads, as well as the addition of features such as the ability for users to create timelines, query additional classifications according to the NQF (for Educational episodes) and SIC (for Occupational episodes), and to visualise the timelines returned.

In the next chapter, we discuss our final implementation of query approximation and relaxation, in a generic application setting, describing the system architecture, data structures and query processing algorithms.

CHAPTER 6

The *Omega* System

In the previous chapter, we described our initial implementation of approximated and relaxed querying within the *ApproxRelax* prototype system, restricting our focus to the user-facing features of *ApproxRelax*. We also presented a qualitative case study showing how *ApproxRelax* overcame problems in a previous system in the domain of lifelong learning.

In this chapter we present *Omega*, an implementation of approximation and relaxation of conjunctive regular path queries. We present our system architecture in Section 6.1, followed in Section 6.2 with an introduction to the collections library used in *Omega*. In Section 6.3 we provide an overview of Sparksee, which we use as our data store, along with the main API methods used in our implementation. This is followed with a description of how we create our data graphs in Section 6.4. Section 6.5 describes the initialisation of a query conjunct, encompassing both the construction of the associated automaton and the initialisation of the query conjunct's data structures. This is followed in Section 6.6 by a description of the algorithms used in the evaluation of a query conjunct. In Section 6.7 we discuss approaches followed by other implementations of regular path queries, and end the chapter with some concluding remarks in Section 6.8.

6.1 System architecture

Figure 6.1 illustrates the architecture of the *Omega* system. Sparksee[1] (formerly known as DEX) is used as the data store. The development was undertaken using the Microsoft .NET framework. The system is comprised of four components: (i) the *console* layer, in which queries are submitted, and which displays the results; (ii) the *system* layer in which query plans are constructed and executed; (iii) the Sparksee API (C#) which provides an interface for invoking the required data access methods to the data store; and (iv) the data store itself.

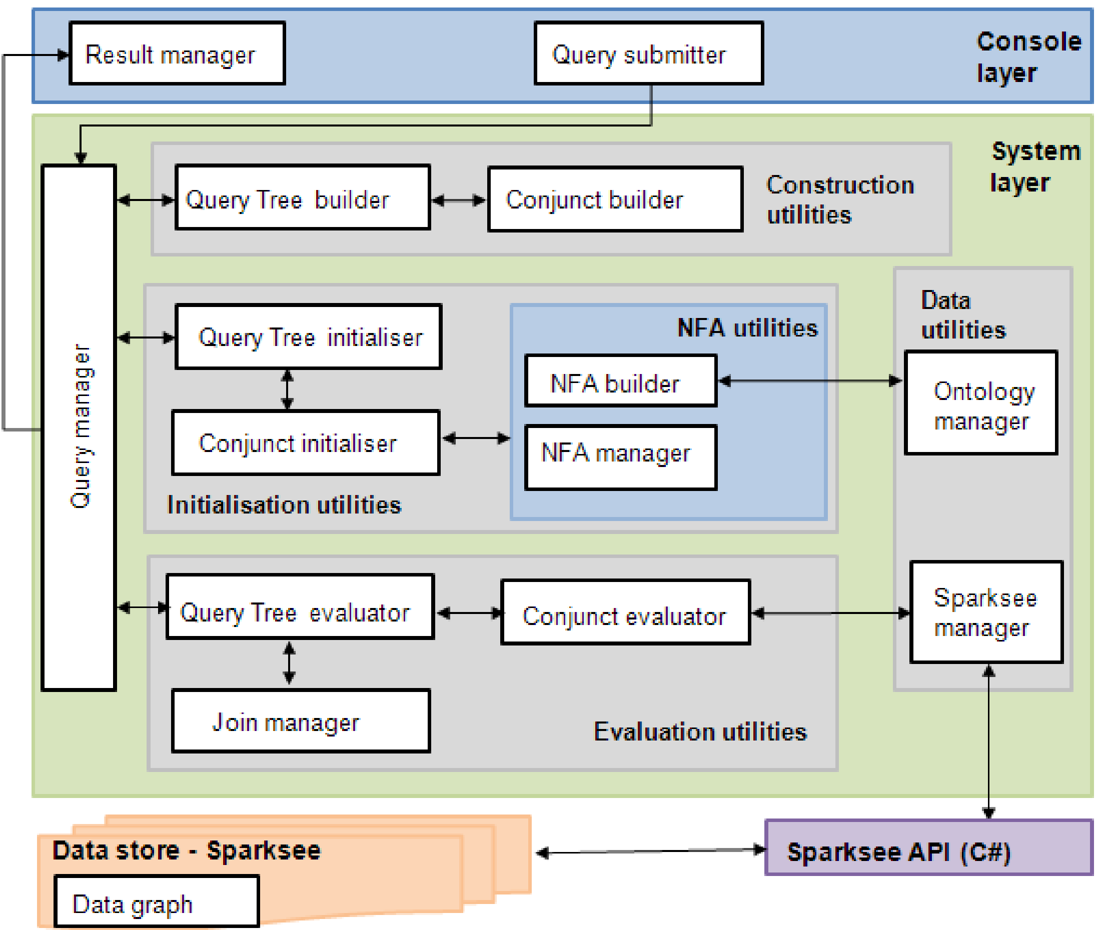

Figure 6.1: System architecture.

Query evaluation commences when *Query submitter* invokes *Query manager*,

[1]http://sparsity-technologies.com

passing it a query that is to be evaluated. *Query manager* invokes *Query Tree builder* to construct the query tree, comprising inner nodes representing join operators and leaf nodes representing individual query conjuncts. The query tree is constructed using standard methods [127].

Query Tree builder calls *Conjunct builder* to construct each leaf node of the query tree. *Query manager* next passes the query tree to *Query Tree initialiser*, which traverses the query tree in a top-down manner, beginning at the root. Whenever *Query Tree initialiser* encounters a leaf node in the query tree, it invokes *Conjunct initialiser* on that conjunct. This in turn invokes *NFA builder* to construct the automaton, M_R, corresponding to the conjunct's regular expression. If the conjunct is approximated or relaxed, then *NFA manager* is invoked to produce an approximate or relaxed automaton (A_Q or M_Q^K respectively), with the relevant edit or relaxation operators applied; this is discussed further in Section 6.5.1. For the construction of a relaxed automaton, *NFA manager* interacts with *Ontology manager*, which stores the extended reduction of the ontology[2].

Query manager then invokes *Query Tree evaluator*. *Query Tree evaluator* traverses the query tree. If the current query tree node is a leaf, the ranked answers for the query conjunct are computed by invoking *Conjunct evaluator*. This module constructs the weighted product automaton, H, of the conjunct's automaton with the closure of the data graph, G. APPROX queries in *Omega* will use the closure of G as the latter is materialised.

The construction of H is incremental, with *Conjunct evaluator* invoking *Sparksee manager* to retrieve only those nodes and edges of G that are required in order to compute the next batch of k results (for some predefined value of k, default 100).

If the current query tree node is a join, *Query Tree evaluator* works in conjunction with *Join manager* to perform a ranked join of the answers returned thus far by its two child nodes. The join algorithm used is that described in [67], itself adapted from [68], and we provide here a brief overview. The query tree supports an *Iterator* interface, implementing the `Open` procedure and the `GetNext` function. The evaluation of the query commences by invoking the query tree's `Open` procedure, which initialises the internal structures needed for each inner node and each leaf node.

[2]The primary motivation for basing our implementation on an NFA-oriented approach is to be able to execute uniformly both APPROX and RELAX queries. Additionally, this aligns the implementation closely with our theoretical NFA-based approach presented in Chapters 3 and 4.

Evaluation of the query commences by the invocation of `GetNext` on the root node. `GetNext` cascades further down the tree until a conjunct node is reached. `GetNext` on a conjunct node computes the answers for the conjunct in order of non-decreasing distance. The query tree is thus traversed depth-first, with `GetNext` invoked down the tree, and results returned on the way up. During this process, the answers from each conjunct undergo a natural join operation with the answers of their sibling conjunct node upon invocation of their parent node's `GetNext` function. For each pair of tuples that are joined, their individual distance values are added to obtain the distance value of the resulting tuple. The results from this join operation are then pipelined upwards, providing input for the next level's join operation and so on, until the root of the tree is reached. Once the root of the query tree has been reached, the processing terminates and the list of answers now holds the next k results, ranked by increasing distance. *Query manager* passes this list to *Result manager* which displays the results in ranked order.

6.2 The C5 Generic Collection library

The importance of using optimised data structures with regard to our required access patterns is crucial in terms of performance. The early version of *Omega* used data structures that came as part of the .NET framework (version 4.0). According to the language specification, most of the native .NET data structures re-allocate items in an array by copying them into a new, larger array, which leads to a performance penalty when executing our algorithms; these are detailed later on in this chapter.

The *Omega* implementation makes extensive use of data structures provided by the **C5 Generic Collection** library[3]. This library was written by researchers at the IT University of Copenhagen, and provides optimised collection-based data structures for C#. The comprehensive manual lists the time complexity for every data structure and method. Below, we detail the structures used in *Omega* and their time complexity:

- `HashSet`: This collection is a set of items (of some type T) using a hash table with linear chaining. Both the expected time for lookups and expected

[3] `http://www.itu.dk/research/c5/`

amortised time for updates is $O(1)$. Adding an item returns a boolean value indicating whether or not the insertion was successful; attempting to add a duplicate value returns *false*; we make use of this feature in Section 6.6.1.

- `HashedLinkedList`: This collection corresponds to an ordinary linked list but additionally maintains a hash table in order to optimise item lookups within the list (thus, lookups by item value are $O(1)$). Items are of some type T, and are added to the end of the list in $O(1)$ time and removed from the end in $O(1)$ time. Additionally, this collection does not allow duplicate values to be stored.

- `HashDictionary`: This collection is a hash table of typed (key,value) pairs; the key is of some type T, and the value is of some type T', where T and T' may or may not be the same type. Accessing an entry by key, entry insertion and deleting an entry by key all take expected time $O(1)$.

6.3 The Sparksee Data Model and API

For our data store, we use Sparksee[4], which supports a labelled directed attributed graph data model. The two main structures used in our implementation are *nodes* and *edges*, each of which has a pre-created *type* — this is a label, of *string* data type — and a unique object identifier, *oid*, of *long* data type, which is used to identify uniquely the node or edge. Edge types may be defined to be *directed* or *undirected* when they are first created. Associated with each node and edge are zero or more attributes, which are key-value pairs; the value may be of any primitive data type.

The main Sparksee API functions used in *Omega* are as follows:

- `Neighbors`: takes as arguments a node n and edge type t, and returns the set of nodes connected to n via an edge of type t. The direction of the edge may optionally also be specified, so that only outgoing or incoming neighbouring nodes of n are returned.

- `Heads`: takes a set of edges E, and returns the set of nodes which are the target of an edge in E.

[4]`http://sparsity-technologies.com`

- `Tails`: is analogous to `Heads`, except that nodes which are sources of edges in E are returned.
- `TailsAndHeads`: returns the union of `Heads` and `Tails`.

To store the data, Sparksee uses a combination of inverted indices and bitmap vectors [96]. Inverted indices are used either to provide a fast access path to the type of the node or edge given by the *oid*, or to the nodes connected to the edge represented by the *oid*, or the attribute value on the node or edge represented by the *oid*. The latter are then linked to bitmap vectors which are used to store *oids* representing respectively in each case, either all the nodes and edges in the graph of the given type, or all the edges connected to the node, or all the nodes and edges in the graph having the attribute value. The bitmap vectors are compressed by grouping the bits into 32-bit clusters and then storing only those clusters with at least one bit set, and, additionally, compressing long sequences of zeroes. The maps — the inverted indices — use a B+ tree implementation.

To improve the performance of the `Neighbors` function, an indexing option may be set when creating an edge type t, resulting in the creation of an index entry whenever an edge of type t is created between any two nodes. Although there is a small performance penalty incurred when creating such edges, this is more than compensated for when executing the `Neighbors` function, in our experience, as we are optimising for applications where reads greatly dominate writes.

Additionally, both node- and edge-level attributes may also be configured to be indexed when they are created (the index stores all *oids* associated with the attribute's value); this improves performance when retrieving a node or an edge by an associated attribute.

Full details regarding Sparksee may be found in [96, 97] and the User Manual.[5]

6.4 Creating data graphs in Omega

As it is mandatory in Sparksee for each node to have a *type*, and as our graph data model does not assume that nodes are typed, we create all of our nodes to be of the same Sparksee type, 'node'. All of our nodes have one Sparksee attribute, of *string*

[5] `http://www.sparsity-technologies.com/downloads/UserManual.pdf`

data type, representing the node label (which is unique in the data graph G). This attribute is created with indexing enabled.

We create multiple Sparksee edge types, all of which are defined to be *directed* edge types with indexing enabled. Specifically, for each edge in G having label $l \in \Sigma$, two Sparksee edges are created: (i) one having type l, and (ii) one having type 'edge' with an associated indexed *string*-valued attribute corresponding to l.

We introduce the generic 'edge' type to counter a limitation of the `Neighbors` function, which requires the type of the edge to be provided as an argument, in order to allow us easily to retrieve multiple types of edges simultaneously. For instance, in Figure 5.2 there is a 'next' edge from the node 'liz2' to the node 'liz3', and thus both a '`next`' edge type and the generic 'edge' type with an attribute 'next' would be created between these nodes.

On the other hand, for each edge in G having the `type` label, only one Sparksee edge is created, whose type is '`type`'. As the purpose of the `type` label is very specific (expressing class membership) in contrast to all the labels in Σ, this is done to distinguish the former from the latter, enabling us to test the effects of, for example, excluding the `type` label from approximation operations. In cases which require the retrieval of *all* types of edges adjacent to a node, we retrieve all 'edge' edges, followed by all `type` edges.

6.5 Conjunct initialisation

The initialisation of a query conjunct (X, R, Y) is comprised of two main elements: the construction of the associated automaton (one of M_R, A_Q or M_Q^K), and the initialisation of its data structures prior to the evaluation of the conjunct. We discuss each in detail here.

6.5.1 Construction of the automaton

We remind the reader that we described the automata, M_R, A_Q and M_Q^K, in detail in Chapter 3. For all conjuncts, an automaton (NFA) M_R is first constructed from regular expression R using standard techniques. Then, if the conjunct is prefixed by APPROX or RELAX in the query, additional transitions and states are added,

along with the removal of ϵ-transitions, to form A_Q or M_Q^K respectively. As the automaton is weighted, the removal of ϵ-transitions may result in final states having an additional, positive weight. For state s, we denote this weight by $weight(s)$. We note that in this thesis, *weight* is synonymous with *cost*.

The NFA is represented as a set of transitions (s, a, c, t), where s is the 'from' state, t is the 'to' state, a is the label, and c is the cost. To implement the automaton, we use the `HashDictionary` data structure introduced in Section 6.2, where the key is an *integer* representing a 'from' state s, and the value is a `HashedLinkedList` of tuples representing the transitions outgoing from s. Each tuple comprises an *integer* representing the 'to' state t, a *string* representing the label a, and an *integer* representing the cost c. The tuples in a `HashedLinkedList` are stored in order of ascending cost, and there are no duplicate transitions; i.e. transitions with the same 'to' state and label.

If the conjunct is in APPROX form and either the *insertion* or *substitution* edit operation has been applied, this will result in many transitions in the NFA for each such operation, one transition for each label in $\Sigma \cup \Sigma^- \cup \{\texttt{type,type-}\}$. Thus, in order to render our automaton more compact, we represent the insertion or substitution of all the labels as a single transition, represented by the 'wildcard' label $*$.

If X (respectively, Y) is a constant c, we annotate the initial (resp. final) state with c; otherwise we annotate the initial (resp. final) state with the wildcard symbol matching any constant.

6.5.2 Initialisation

Pseudocode for the initialisation of a conjunct first appeared in [67]. However, in practice, its performance when applied to larger graphs proved to be inefficient, owing to having to enqueue every node in a graph G. We therefore present here an amended version (the `Open` procedure), which incorporates several modifications to address the problem of needing to enqueue every node in G; we discuss the modifications under the paragraphs *Cases 1 - 3* below.

After constructing the appropriate automaton, the `Open` procedure evaluates the conjunct by traversing the automaton and the data graph G simultaneously.

The main data structure used in the evaluation of the query conjunct is a global dictionary $\texttt{D}_R$. For implementing $\texttt{D}_R$, we use the `HashDictionary` introduced in Section 6.2. Each key is a pair (d, f) of values, where d represents an integer distance and f represents a boolean flag denoting the final or non-final tuples at that distance. The value associated with each key, implemented using a `HashedLinkedList`, comprises tuples of the form (v, n, s, d, f), where d is the distance associated with visiting node n in state s having started from node v, and f is a flag denoting whether the tuple is 'final' or 'non-final', with the latter being the initial value for f[6]. More details regarding the usage of 'final' and 'non-final' tuples are discussed in Section 6.6.1, but, briefly, a 'final' tuple is one corresponding to a completed run in the product automaton H (owing to having reached a final state in the automaton), and thus will, when dequeued, be an answer, whereas a 'non-final' tuple is still in the process of being evaluated.

Tuples are always added to, and removed from, the head of the linked list. From Section 6.2, adding any tuple to $\texttt{D}_R$ takes $O(1)$ and removing it takes $O(n)$, where n is the number of keys in $\texttt{D}_R$ (at most twice the number of distances in $\texttt{D}_R$), as all the keys need to be scanned to determine the current minimum distance.

We maintain a global list $\texttt{answers}_R$ containing tuples of the form (v, n, d), where d is the smallest approximation distance of this answer tuple to the query and ordered by non-decreasing value of d. This list is used to avoid returning again (v, n, d') for any $d' \geq d$. A global set $\texttt{visited}_R$ is also maintained, storing tuples of the form (v, n, s) representing the fact that node n of G was visited in state s having started the traversal from node v. Both $\texttt{answers}_R$ and $\texttt{visited}_R$ are initialised to the empty list.

We distinguish between three cases in the `Open` procedure:

(*Case 1*) If the conjunct is of the form $(C, R, ?Y)$ where C is a constant, we begin the traversal at the node in G having the attribute value C.

(*Case 2*) If the conjunct is of the form $(?X, R, C)$, the conjunct is transformed to $(C, R^-, ?X)$. In R^- each symbol in R is inverted. This can be done by constructing initially the NFA for R, and then reversing it to recognise R^- as follows (see [145]): (i) add a new state s, and, for each final state f, add a new ϵ-transition from f to

[6]The distance and the flag are stored in both the key and the tuple associated with the key as this is useful for implementation-specific reasons.

Procedure Open

Input: query conjunct (X, R, Y)

(1) construct NFA M_R for R; initial state is s_0
(2) transform M_R into A_Q (APPROX) or M_Q^K (RELAX) if necessary
(3) $\texttt{visited}_R \leftarrow \emptyset$
(4) $\texttt{answers}_R \leftarrow \emptyset$
(5) **if** *conjunct is of the form* $(C, R, ?X)$ **then**
  /* Let n be the node in G corresponding to C */
(6) **if** *RELAX is being applied and* C *is a class node* **then**
(7)   **foreach** *node* $m \in \texttt{GetAncestors}(n)$ **do**
(8) $add(\texttt{D}_R, (m, m, s_0, 0, false))$
(9) **else**
(10) $add(\texttt{D}_R, (n, n, s_0, 0, false))$
(11) **else**
  /* the conjunct is of the form $(?X, R, ?Y)$ */
(12) **if** s_0 *is final* **then**
(13)   **if** $weight(s_0) = 0$ **then**
(14)   **foreach** *node* $n \in G$ **do**
(15) $add(\texttt{D}_R, (n, n, s_0, 0, true))$
(16)   **else**
(17)   **foreach** $n \in \texttt{GetAllNodesByLabel}(s_0)$ **do**
(18) $add(\texttt{D}_R, (n, n, s_0, 0, false))$
(19) **else**
(20)   **foreach** $n \in \texttt{GetAllStartNodesByLabel}(s_0)$ **do**
(21) $add(\texttt{D}_R, (n, n, s_0, 0, false))$

s; (ii) convert each final state f into a non-final state; (ii) change the initial state s_0 to a final state; (iv) convert s into a final state; and (v) reverse the direction of all transitions and invert all the labels. This reversal can be accomplished in linear time starting from the NFA for R [145]. Thereafter, any additional states and transitions due to the application of APPROX or RELAX are added. Thus, *Case 2* reverts to *Case 1*.

If the RELAX operator has been applied to the conjunct and C is a class node, we also add to $\mathtt{D}_R$ every node returned by the function `GetAncestors` (line 7). This function returns all superclasses of C in order of increasing specificity so that they are added to the value component (the `HashedLinkedList` containing the tuples) of $\mathtt{D}_R$ in that order. We want to process more specific classes first, given that nodes representing more general classes will have larger degree (owing to transitive closure) and will lead to answers of greater cost; indeed, the higher-degree nodes may never even get to be added to $\mathtt{D}_R$ if the lower-degree nodes yield all the answers needed. For example, using the second conjunct (`C2`) from the example given in Section 5.2, `RELAX(?A,qualif.type,InformationSystems)`, and referring to Dan's timeline graph in Figure 5.1 in Chapter 5, `GetAncestors` returns the nodes representing 'Subject', 'Mathematical & Computer Sciences' and 'Information Systems'.

(*Case 3*) For a conjunct of the form $(?X, R, ?Y)$, lines 12 to 21 are invoked in order to limit the number of nodes in G added to $\mathtt{D}_R$. If the initial state s_0 of the NFA is not also a final state, the objective is to limit the nodes added to $\mathtt{D}_R$ to those from which there will be matching edges. The function `GetAllStartNodesByLabel` (line 20), returning a set of nodes in G, takes as input a list of all labels on transitions whose 'from' state is the initial state s_0. Each label in the list is processed as follows (the list of labels is in order of increasing cost):

- The directionality of the label is determined — i.e. whether it is an incoming or an outgoing edge, or whether both incoming *and* outgoing edges are required (as for the *-labelled transitions, introduced above).
- The set of object identifiers (oids) for the nodes having the relevant edge and directionality (as determined above) are retrieved using the Sparksee functions `Heads`, `Tails` and `TailsAndHeads` (introduced in Section 6.3) which provide access methods by edge type and direction.

- Sparksee set operations are used to maintain a distinct set of nodes, which means that the same node is not re-added to $\mathtt{D}_R$ at a higher cost (this can occur with the '*' label).

Each node returned by `GetAllStartNodesByLabel` is subsequently added to $\mathtt{D}_R$ in line 21. We note that as the nodes are returned — and enqueued in $\mathtt{D}_R$ — in order of increasing cost, the nodes reached by transitions of lower cost will be evaluated before those reached by transitions of higher cost.

If the initial state s_0 of the NFA is also a final state, the $weight(s_0)$ variable referred to in line 13 denotes the value of the final weight introduced in Section 6.5.1. If $weight(s_0) = 0$, then all nodes are answers at distance 0, which means that all the nodes need to be enqueued (line 14). If, on the other hand, $weight(s_0) > 0$, then the function `GetAllNodesByLabel` (line 17) is invoked instead. This is identical to `GetAllStartNodesByLabel`, except that it additionally returns all the remaining nodes in the graph G. This is because these nodes may contribute to the list of answers at a cost equivalent to the deletion of all edge labels in the path given by R.

We have implemented the above two functions and the function for retrieving all nodes in G (line 14) as coroutines in conjunction with the `GetNext` function (discussed in Section 6.6.1), incrementally obtaining nodes in batches (the default is 100 nodes at a time). We found that, as a result, the execution time of some queries was reduced by half, since nodes not required to answer the user's query are not added to $\mathtt{D}_R$.

6.6 Query conjunct evaluation

The two functions concerned with the evaluation of a single query conjunct are `GetNext` and `Succ`, which have previously been presented in [67, 115] and in Chapter 3. We present here updated versions of both these functions, taking into account optimisations, as well as presenting for the first time the detail of the `NextStates` function. Additionally, we describe our physical implementation of these functions.

6.6.1 The GetNext function

The function `GetNext` returns the next query answer, in order of non-decreasing distance from the original query Q, by repeatedly removing the first tuple (v, n, s, d, f) from the distance d list of D_R until D_R is empty (lines 1 and 2). If the removed tuple is final (f is $true$) and the answer (v, n, d') has not been generated before for some d', the triple (v, n, d) is returned (line 6) after being added to $\texttt{answers}_R$ (line 5). If the tuple is not final, we add (v, n, s) to $\texttt{visited}_R$ (line 9), and add $(v, m, s', d + d', false)$ to D_R (line 11) for each transition $\xrightarrow{d'} (s', m)$ returned by the function $\texttt{Succ}(s, n)$ (invoked in line 10) such that $(v, m, s') \notin \texttt{visited}_R$. If s is a final state, its annotation matches n, and the answer (v, n, d') has not been generated before for some d', then we add the weight of s to d, mark the tuple as final, and add the tuple to D_R (line 13).

We now list some implementation aspects of `GetNext`.

- In lines 14 to 16, we incrementally add the next batch of initial nodes by utilising a coroutine for $(?X, R, ?Y)$ conjuncts, as introduced in Section 6.5.2. If D_R no longer contains any tuples at distance 0, we retrieve and add the next batch of nodes from lines 15, 18 or 21 in the `Open` procedure. We refer to these nodes as initial nodes on line 14 in `GetNext`.

- For the list $\texttt{visited}_R$, we use a `HashSet` structure, introduced in Section 6.2, thus taking advantage of the $O(1)$ lookup time. Lines 8 and 9 in practice are executed as a single step, and the logic in lines 10 to 13 is only executed if the item was added. This means that we never re-process a previously-processed (v, n, s) triple; this situation may arise when (v, n, s) triples of monotonically-increasing distances are created and added at lines 11 and 13 (we therefore never process 'duplicate' tuples at a higher distance). When we used a structure with bag rather than set semantics in earlier versions of *Omega*, some queries — in particular, those consisting of more complex paths and/or processing large numbers of nodes with many matching edges resulting in many intermediate tuples — failed to terminate owing to millions of duplicate tuples being placed on D_R.

- For the dictionary D_R, we need a fast way to dequeue tuples. We recall that

dequeuing a tuple takes $O(n)$, where n is the number of keys (this is at most twice the total number of distances currently in $\mathtt{D}_R$) in the dictionary, as we scan the keys of $\mathtt{D}_R$ to find the current minimum distance. We note that by keeping track of the current minimum distance, the complexity could be reduced from $O(n)$ to $O(1)$. However, this is not necessary as, in practice, n is small (never exceeding single digits in our experiments). We introduce the notion of final and non-final tuples in order always to prioritise the dequeueing of the 'final' tuples (rather than the 'non-final' ones) at the minimum distance (if any exist), so that the answers may be returned earlier. Including this refinement improved performance, and also ensured that some queries, which had previously failed by running out of memory, completed.

- The list of answers ($\mathtt{answers}_R$), which both stores the answers and in which we perform lookups to ensure answers are not duplicated, uses a `HashDictionary` structure. The key of this structure is the string representation of the answer — not including the cost — and the triple itself, containing the distance, is the value.

- During the execution of a query, we only ever retrieve, process and store the object identifier of a node (this is an *oid*, which is a *long* data type) in memory, as opposed to its *string* name.

- In order to consume less memory, we use *strings* rather than objects to store all the tuples — i.e. $(v, n, s, d, \text{'}final')$, (v, n, d) and (v, n, s) — decomposing and recomposing the constituent components of any tuple when required. When objects were used, some of the queries which resulted in many intermediate tuples being found and enqueued either ran slowly or ran out of memory owing to objects consuming more memory and extra logic being required for checking equality of objects.

6.6.2 The `NextStates` function

The function `NextStates`(s) returns the set of states in the automaton reachable from state s, along with the associated input a and cost c for each state in the

Function GetNext(X, R, Y)

Input: query conjunct (X, R, Y)
Output: triple (v, n, d), where v and n are instantiations of X and Y

```
(1)  while nonempty(D_R) do
(2)      (v, n, s, d, final) ← remove(D_R)
(3)      if final then
(4)          if ∄d'.(v, n, d') ∈ answers_R then
(5)              append (v, n, d) to answers_R
(6)              return (v, n, d)
(7)      else
(8)          if (v, n, s) ∉ visited_R then
(9)              add (v, n, s) to visited_R
(10)             foreach -d'-> (s', m) ∈ Succ(s, n) s.t. (v, m, s') ∉ visited_R do
(11)                 add(D_R, (v, m, s', d + d', false))
(12)             if s is a final state and its annotation matches n and
                   ∄d'.(v, n, d') ∈ answers_R then
(13)                 add(D_R, (v, n, s, d + weight(s), true))
(14)     if no distance 0 tuples in D_R and more initial nodes available then
(15)         foreach initial node n' do
(16)             add(D_R, (n', n', s_0, 0, false));
(17) return null
```

set. In `NextStates`, we recall from Section 6.5.1 that the automaton is encoded as a dictionary (using `HashDictionary`) containing the *from* state as the key, and a hashed linked list as the value (using `HashedLinkedList`). The variable `successors` is the hashed linked list (using `HashedLinkedList`) retrieved from the dictionary where the key is the state s. We then add (line 4) the *to* state, label a and cost c, all of which are obtained from the tuples within the linked list, to the `nextstates` variable. The final operation, represented by the function `OrderStates` in line 5, is to order `nextstates` first by ascending *cost*, and then by *label* in descending order (so that the '*' label is last in its cost group). This means the costliest labels are processed last by `Succ`.

In an earlier version of the system, ϵ-transitions were not removed from the automaton. However, this resulted in `NextStates` needing repeatedly to follow ϵ-transitions until a non-ϵ-transition was found, whilst summing up the costs of the

intermediate ϵ-transitions. However, we found that the fastest performance was achieved by removing ϵ-transitions [32], which is done as part of the creation of the automaton.

Function NextStates(s)

Input: state s of NFA

Output: set of states in NFA that can be reached from state s, along with the input a and cost c of reading a

```
(1) nextstates ← ∅
(2) successors ← NFA(s)
(3) foreach (s', a, c) ∈ successors do
(4)     add (a, s', c) to nextstates
(5) return OrderStates(nextstates)
```

6.6.3 The `Succ` function

The `Succ` function was presented in [115] and in Chapter 3; briefly, this took as input a node (s, n) of the weighted product automaton H and returned a set of transitions $\stackrel{a,d}{\rightarrow} (p, m)$, such that there was an edge in H from (s, n) to (p, m) with label a and cost d, where the function `NextStates`(s) (discussed in Section 6.6.2) returned the set of states that could be reached from state s, along with the associated input a and cost c for each state in the set.

In our updated `Succ` function, we explicitly only request those edges for node n in G whose label corresponds to one of those returned by `NextStates`(s), thereby using the transitions in the automaton to guide the selection of neighbouring nodes in G. As it is possible for `NextStates` to return identical labels consecutively (for example, label a at cost 0 for states s_i and s_j), we also store the results of the `NeighboursByEdge` function in U, so that an identical call to `NeighboursByEdge` can be avoided.

The function `NeighboursByEdge` takes as input the *oid* of a node n from G and a *label*, and returns a list of neighbouring node *oids*. If the label is not '*', we invoke the Sparksee function `Neighbors` to retrieve all neighbouring nodes for n connected by an edge labelled with *label*, taking directionality into account. If, on the other hand, the label is the '*' label (we recall these are added to the automaton if either

the *insertion* or *substitution* edit operation was included in the APPROX operator), we first retrieve, using the Sparksee function `Neighbors`, all the edges labelled ‘edge’ for node n. Thus, in one call, we obtain all the non-`type` neighbouring nodes. We then subsequently invoke `Neighbors` to retrieve all the edges labelled `type`. We do this for both directions, in both cases. In each case, we iterate over the neighbouring nodes, adding their *oid* to W (lines 6 and 7).

Function Succ(s, n)

Input: state s of NFA and node n of G
Output: set of transitions from (s, n) in H

```
(1) W ← ∅; U ← ∅
(2) prevlabel ← null
(3) foreach (currlabel, successor, cost) ∈ NextStates(s) do
(4)     if currlabel ≠ prevlabel then
(5)         U ← NeighboursByEdge(n, currlabel)
(6)     foreach node m ∈ U do
(7)         add the transition --cost--> (successor, m) to W
(8)     prevlabel ← currlabel
(9) return W
```

6.7 Other implementations

We now discuss the approaches taken by other implementations of regular path queries, and contrast these with our work.

To our knowledge, the earliest implementation of regular path queries is DataGuides, presented by Goldman and Widom [43] and based on Lorel [1] by Abiteboul *et al.*, which uses an automaton-based approach. The NFA representing the graph is converted to a DFA which is then minimised and subsequently used as an index. The size of this index may become prohibitively large and, to mitigate this, Goldman and Widom [44] present heuristics by which the index size may be reduced by merging paths either with matching last labels or containing more than one instance of the same label.

Zauner *et al.* [141] present RPL, a regular path language allowing conditional predicates to be expressed over the nodes, edges, or nodes and edges appearing on

paths within RDF data. The implementation, like ours, uses an automaton-based approach.

The work undertaken by Koschmieder and Leser [82] resembles our work in terms of the evaluation of regular path queries over large graph-structured data. The authors introduce a technique taking advantage of labels that occur infrequently in a graph. They present an algorithm which, when given a regular path query containing one or more instances of one of these infrequent labels, breaks down the original query into a sequence of sub-queries so that each sub-query either begins or ends with one of the infrequent labels. This has the effect of accelerating the query execution as a whole by exploiting the notion of high selectivity. The authors compare their method — which uses a bi-directional search utilising indexes storing the number of occurrences of each edge label in the data graph — to the automata-based methods, and present a comprehensive description of their implementation, with results showing that their method is faster and benefits from being parallelizable. In comparison with our work, their system only caters for exact queries, and relies upon the presence of rare labels. However, a direction of future research could be to incorporate into *Omega* the notion of query rewriting based on any rare labels in a graph.

Cedeño and Candan [19] present a framework, R^2DF, allowing weighted RDF data to be queried in a cost-aware manner, and returning results ranked according to cost; the motivation for this is to be able to query data in which factors such as trust or validity play an integral part. This is accomplished by (i) the introduction of a ranked RDF specification allowing triples to be augmented with a weight (or cost); and (ii) a proposed extension to the SPARQL specification, SPARankQL, which provides novel predicates expressing flexible paths between nodes and the capacity to define ranked queries (in which the weights are used). Additionally, AR2Q, a query processing engine supporting SPARankQL is introduced, which is constructed over the ARQ implementation incorporated within the Jena framework. This work bears some similarity with ours in that SPARankQL allows a flexible path to be expressed. However, whereas our work allows the path to be expressed by a regular expression which may then be mutated by a series of edit operations, SPARankQL can only be used either to express no restrictions on paths from a node (so that all edges from the node are matched) or to express restrictions on specified label/s of

the path along with weights specifying an upper and lower bound.

Another implementation approach is described by Dey *at al.* [28], in which (exact) RPQs are converted into either Datalog queries or recursive SQL. The implementation is able to imbue the answers with additional provenance-related information.

An implementation exemplifying the reachability indexing approach for exact CRPQs is described by Gubichev *et al.* [51].

At the end of Chapter 7, we compare the performance of *Omega* with the implementations discussed in this section.

6.8 Concluding remarks

In this chapter, we presented in detail our *Omega* system, describing the system architecture, how data graphs are created, the internal data structures used, and how a query conjunct is initialised and thence evaluated, as well as describing in full the algorithms used. We also discussed alternative implementations of regular path queries, comparing these with our approach.

In the next chapter, we present performance results and analyses on two sets of queries evaluated on two data graphs using *Omega*, and, additionally, provide a performance comparison between *Omega* and the implementations discussed in this section, along with optimisations designed to improve further the performance of some of the poorly-performing queries.

CHAPTER 7

Query Performance Analysis

In this chapter, we present performance results from two evaluations using the *Omega* system described in the previous chapter, each of which is concerned with a different domain: (i) lifelong learners' data derived from the *L4All* system (henceforth referred to as L4All data); and (ii) YAGO data, derived from DBPedia and WordNet. These two datasets are used owing to the contrast between them in data characteristics: the L4All data is sparse in that there are comparatively few connections relative to its size, whereas YAGO is more densely-linked, yielding more heterogeneous and semantically rich data, especially when combined with its ontology. Moreover, the two ontologies also exhibit contrasting characteristics. We additionally describe an optimisation approach designed to improve the performance of poorly-performing approximation queries.

In Sections 7.1 and 7.2, we present the L4All evaluation and YAGO evaluation, respectively. In each section, we describe in detail the data graph, the associated ontology and the queries used, concluding by presenting and discussing the experimental results. We continue by providing a performance comparison in Section 7.3 between *Omega* and the implementations discussed in Section 6.7.

In Section 7.4, we describe by means of an empirical evaluation how the availability of simple statistics relating to paths in the data graph can be used to improve

the run-times of some of the poorly-performing queries from the performance evaluations of the two datasets, before ending this chapter with some concluding remarks in Section 7.5.

For both evaluations, the focus is on the performance of single-conjunct queries. Multi-conjunct queries are not considered as we assume that a standard rank-join method [68] is used for these, as detailed in [67]. However, detailed performance evaluations of multi-conjunct queries will form part of future work.

All experiments were run on an Intel Core i7-950 (3.07-3.65GHz) with 6GB memory, running Windows 7 (64 bit).

7.1 The L4All evaluation

Our first evaluation uses data from the *L4All* system, introduced in Chapter 5. We remind the reader that the *L4All* system aimed to support lifelong learners in exploring learning opportunities and in planning and reflecting on their learning. The system allows users to create and maintain a chronological record — a timeline — of their learning and work episodes. Each episode is (i) linked to a category by an edge labelled `type`, (ii) linked to other episodes by either a `next` or a `prereq` edge (indicating whether the later episode simply followed chronologically or whether the the earlier episode was necessary in order for them to be able to proceed to or achieve the later episode) and (iii) either linked to an occupational or an educational event, respectively by means of a `job` or `qualif` edge, which in turn is classified in terms of Educational Qualification Level or Industry Sector.

Table 7.1 shows the class hierarchies used in the ontology accompanying the L4All data; the *depth* is the length of the longest path from the root to the leaf nodes, and the *average fan-out* is the average number of children of each non-leaf class. There is only one property hierarchy: the super-property `isEpisodeLink` has `next` and `prereq` as subproperties. All the non-`type` properties in the data are shown in Table 7.2.

Class hierarchy	**Depth**	**Average fan-out**
Episode	2	2.67
Subject	2	8
Occupation	4	4.08
Education Qualification Level	2	3.89
Industry Sector	1	21

Table 7.1: Characteristics of the class hierarchies in the L4All data graphs.

Property	**Domain**	**Range**
next	Episode	Episode
prereq	Episode	Episode
qualif	Episode	SBJ
level	SBJ	NQF
job	Episode	SOC
sector	SOC	SIC

Table 7.2: Properties in the L4All data graphs other than '`type`'.

7.1.1 Data

Our initial L4All data comprised five detailed timelines from real users. Each of these timelines consisted of a mixture of educational and occupational episodes, and varied in terms of the number of episodes contained within them, as well as the classification of each episode.

We then scaled this data graph up by creating synthetic versions of the five real timelines in order to obtain four data graphs of increasing size, called **L1**, **L2**, **L3** and **L4**, containing, respectively, 143, 1,201, 5,221, and 11,416 timelines. Table 7.3 shows the characteristics of each data graph. All of the data, including the closure of the graph, is materialised to disk. Included are all the nodes and edges of the graphs, along with the edges induced by applying the transitive closure of the `type` edges, as detailed in Chapter 3. Materialising the closure of the graph results in a 25 to 30 percent increase in the size of the data stored to disk. Both query approximation and query relaxation are applied to the closure of the data graph. The non-`type` properties shown in Table 7.2 also have defined domains and ranges, but as these are not used in the our performance study, we do not discuss them further. A future direction of research is to investigate an in-memory computation of the closure of the data graph, thereby only storing the data graph itself on disk.

The synthetic timelines were generated by duplicating a real timeline and using the ontology to alter the classification of each episode to be a 'sibling' class of its original class, for as many sibling classes as are present. Each duplicated timeline remained identical to the original in terms of the number of episodes, whether the type of the episode was educational or occupational, and the manner in which episodes were linked to each other. Thus, as the closure of the data graph increases in size, the degree of the class nodes (i.e. the nodes with incoming `type` edges) increases linearly. We see in Figure 7.1 that, as the size of the closure of the data graph increases, the number of edges increases linearly with the number of nodes.

7.1.2 Queries

We executed 12 single-conjunct queries on the L4All data sets in order to evaluate the performance of our APPROX and RELAX operators. Each query was first run in 'exact' mode — i.e. neither APPROX nor RELAX is used — followed by versions

	L1	L2	L3	L4
Nodes	2,691	15,188	68,544	240,519
Edges (data only)	10,724	63,979	300,523	1,009,051
Edges (closure)	19,856	118,088	558,972	1,861,959
Size (data only)	2.37 MB	11.1 MB	49.53 MB	169.9 MB
Size (closure)	2.99 MB	14.18 MB	64.51 MB	220.8 MB
Percentage increase (closure)	26.2%	27.8%	30.2%	29.96%

Table 7.3: Characteristics of the L4All data graphs.

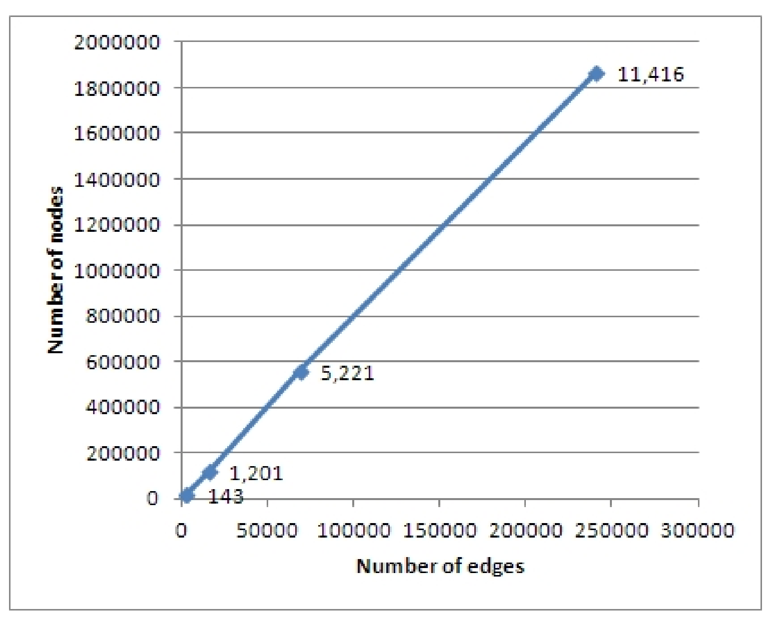

Figure 7.1: The L4All data graph sizes (using the closure of the data graph).

of the same query containing either the APPROX or the RELAX operator. We therefore ran 36 queries in total. We exclude queries 4, 5, 6 and 7 from further discussion as running the exact form of these queries on even the smallest data graph returned well over 100 results (for three of these, over 3,800 results were found). Owing to the large number of exact results being returned for these queries, neither the APPROX nor the RELAX operator was applied to them[1]. Table 7.4 lists the remaining 8 queries that we focus on here. Queries 1, 2 and 3 are derived from the *L4All* case study detailed in Chapter 5, while queries 8 - 12 were additional queries designed to stress-test our implementation.

7.1.3 Baseline experimental results

We used a cost of 1 for each approximation operation (insertion, substitution and deletion). For RELAX, we applied rules 2 and 4 from Figure 3.6, also at a cost of 1. We ran each query five times, discarding the first run as the cache warm-up. After initialisation, each exact query was run to completion, in which *all* results are obtained. On the other hand, each APPROX and RELAX run comprises the following sequence: initialisation; obtain results 1–10 ('batch 1'); obtain results 11–20 ('batch 2'); ...; obtain results 91–100 (batch 10)[2]. For exact queries, the average time to return all answers was taken across runs 2 to 5. For APPROX and RELAX queries, we took the average of each of the 10 batches across runs 2 to 5 to obtain an average for each batch. We then computed the average over all batches. Some of these queries yielded fewer than 100 results.

We show the number of results obtained per data graph for each query in Table 7.5. For APPROX and RELAX queries yielding non-exact answers, we also show in Table 7.5 the distances of the non-exact answers, as well as the number of the answers at each non-zero distance in brackets (with the number of exact answers comprising the difference). The letters '**E**', '**A**' and '**R**' after the name of the data graph in each row refer to the exact, approximated and relaxed versions of the query denoted in the corresponding column. For example, query Q9/APPROX on

[1] As there is no cardinality estimation in *Omega*, it is not possible to predict the number of exact results without executing the query.

[2] We envisage APPROX and RELAX queries being used to retrieve the top k results in an incremental manner, hence our selection of '100' as the maximum number of answers to retrieve.

Query	Details
Q1	(Work Episode,type^{-},?X) This query finds all episodes of type Work Episode.
Q2	(Information Systems,type^{-}.qualif^{-},?X) This query finds all episodes relating to qualifications of type Information Systems.
Q3	(Software Professionals,type^{-}.job^{-},?X) This query finds all episodes relating to jobs of type Software Professionals.
Q8	(Mathematical and Computer Sciences,type.prereq+,?X) The intention of this query is to find all episodes of which educational episodes of type Mathematical and Computer Sciences are a prerequisite. However, this will not return any results as no such path exists (because the type label should be type^{-} and there should also be a label qualif^{-} following it in the regular expression; thus, this query tests the case for when there is an error in R to exemplify the value of our APPROX and RELAX operators.
Q9	(Alumni 4 Episode 1_1,prereq*.next+.prereq,?X) This query finds episodes that are linked to Alumni 4 Episode 1_1 through 0 or more prereq edges, followed by one or more next edges, followed by one prereq edge. This query tests the effects of having a lengthier R, starting from a node of small degree, and where the user wishes to begin traversing from a particular episode, which we assume would be a common goal.
Q10	(Librarians,type^{-},?X) The intention of this query is to find all episodes relating to jobs of type Librarians. However, the label **job**$^{-}$ is missing from the regular expression. This query tests the effects of having a missing label in R, and once again exemplifies the use of our APPROX and RELAX operators.
Q11	(Librarians,type^{-}.job^{-}.next,?X) All episodes that follow episodes representing jobs of type Librarians are returned as answers. This query exemplifies the use of the RELAX operator, as there are relatively few episodes relating to jobs of type Librarians.
Q12	(BTEC Introductory Diploma and Certificate,level^{-}.qualif^{-}.prereq,?X) The intention of this query is to find episodes for which having a BTEC Introductory Diploma and Certificate qualification is a prerequisite. However, this will not return any results as all episodes denoting the qualification only have chronological successors (and thus are not prerequisites for any episode), exemplifying the use of the APPROX operator; the query also demonstrates the use of the RELAX operator, as there are relatively few episodes relating to qualifications of type BTEC Introductory Diploma and Certificate.

Table 7.4: The L4All query set: Q1 - Q3 and Q8 - Q12.

data graph **L2** (looking at row '**L2-A**' and column '**Q9**') returns 1 exact answer (100-(32+67)), 32 answers at distance 1 and 67 answers at distance 2.

7.1.4 Analysis

Figures 7.2, 7.3 and 7.4, all of which use a logarithmic scale, show the average execution times for the exact, APPROX and RELAX versions, respectively, of queries 3, 8, 9, 10, 11 and 12 over the data graphs **L1**–**L4**. Queries 1 and 2 showed similar performance to query 3. For completeness, we show in Table 7.6 the total initialisation time and the total execution time (i.e. the sum of batches 1-10 for the APPROX and RELAX versions) for these queries; the total initialisation time and total execution time is computed as an average across runs 2 to 5.

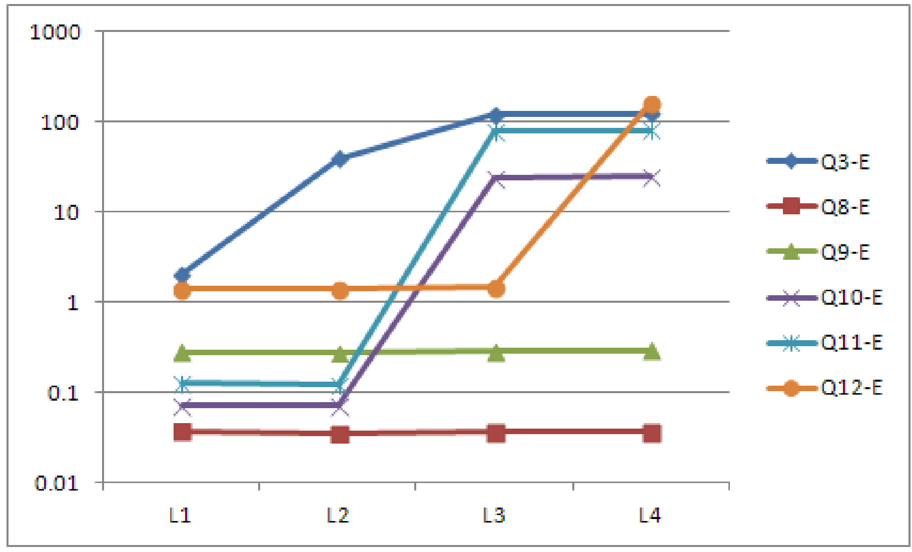

Figure 7.2: Execution time (ms) – exact L4All queries.

For the exact queries shown in Figure 7.2, we see that queries 8 and 9 take constant time for all the data graphs since at most a single answer is returned. The sharp increase in execution time from **L2** to **L3** for queries 10 and 11 is caused by the large increase in the number of answers (from 1 and 2 answers to 1,024 and 2,048 answers, respectively). Query 3 shows a more gradual increase in execution time from **L1** to **L2** and **L3**, corresponding to the increasing number of answers returned (returning 58, 1,090 and 3,104 answers, respectively). Query 12 shows a steep increase owing to the manner in which the synthetic timelines were generated,

	Q1	**Q2**	**Q3**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**
L1: E	873	136	58	0	1	1	2	0
L1: A	100	100	100 **1** *(42)*	100 **2** *(100)*	100 **1** *(32)* **2** *(67)*	100 **1** *(7)* **2** *(92)*	100 **1** *(12)* **2** *(86)*	100 **1** *(100)*
L1: R	100	100	100 **1** *(42)*	0	12 **1** *(11)*	100 **1** *(20)* **2** *(20)* **3** *(59)*	100 **1** *(40)* **2** *(40)* **3** *(18)*	59 **1** *(59)*
L2: E	5,052	425	1,090	0	1	1	2	0
L2: A	100	100	100	100 **2** *(100)*	100 **1** *(32)* **2** *(67)*	100 **1** *(7)* **2** *(92)*	100 **1** *(12)* **2** *(86)*	100 **1** *(100)*
L2: R	100	100	100	0	12 **1** *(11)*	100 **1** *(20)* **2** *(20)* **3** *(59)*	100 **1** *(40)* **2** *(40)* **3** *(18)*	59 **1** *(59)*
L3: E	21,168	1,964	3,104	0	1	1,024	2,048	0
L3: A	100	100	100	100 **2** *(100)*	100 **1** *(32)* **2** *(67)*	100	100	100 **1** *(100)*
L3: R	100	100	100	0	12 **1** *(11)*	100	100	59 **1** *(59)*
L4: E	83,298	5,272	3,104	0	1	1,024	2,048	0
L4: A	100	100	100	100 **2** *(100)*	100 **1** *(32)* **2** *(67)*	100	100	100 **1** *(100)*
L4: R	100	100	100	0	12 **1** *(11)*	100	100	100 **1** *(100)*

Table 7.5: Results for each query and L4All data graph.

	Q3	Q8	Q9	Q10	Q11	Q12
L1: E	*0.09*	*0.13*	*0.16*	*0.08*	*0.11*	*0.11*
	2.08	0.04	0.28	0.07	0.13	1.41
L1: A	*0.12*	*0.18*	*0.24*	*0.09*	*0.15*	*0.15*
	7.29	27.78	361.25	11.21	34.94	12.59
L1: R	*0.97*	*0.14*	*0.18*	*0.96*	*1.00*	*0.11*
	6.37	0.10	1.53	4.47	6.95	3.83
L2: E	*0.10*	*0.13*	*0.16*	*0.09*	*0.10*	*0.11*
	39.39	0.04	0.28	0.07	0.12	1.41
L2: A	*0.12*	*0.18*	*0.24*	*0.10*	*0.17*	*0.15*
	16.40	362.82	2,132.76	62.44	252.78	12.70
L2: R	*0.97*	*0.14*	*0.18*	*0.96*	*0.99*	*0.12*
	5.94	0.10	1.42	18.78	20.98	3.81
L3: E	*0.10*	*0.13*	*0.16*	*0.09*	*0.11*	*0.11*
	121.42	0.04	0.29	24.52	79.39	1.47
L3: A	*0.13*	*0.19*	*0.24*	*0.10*	*0.15*	*0.15*
	30.90	1,092.57	13,138.05	9.92	18.47	12.68
L3: R	*0.97*	*0.14*	*0.18*	*0.97*	*0.99*	*0.12*
	8.43	0.10	1.43	4.24	6.36	3.85
L4: E	*0.10*	*0.14*	*0.17*	*0.09*	*0.11*	*0.11*
	122.94	0.04	0.29	24.80	80.37	162.57
L4: A	*0.13*	*0.19*	*0.26*	*0.10*	*0.16*	*0.18*
	30.74	3,320.73	50,668.37	10.08	18.79	977.61
L4: R	*1.00*	*0.15*	*0.19*	*0.99*	*1.02*	*0.12*
	8.41	0.10	1.48	4.26	6.50	254.72

Table 7.6: Total initialisation and execution times (ms) for each query run over the L4All data graphs (initialisation time in italics).

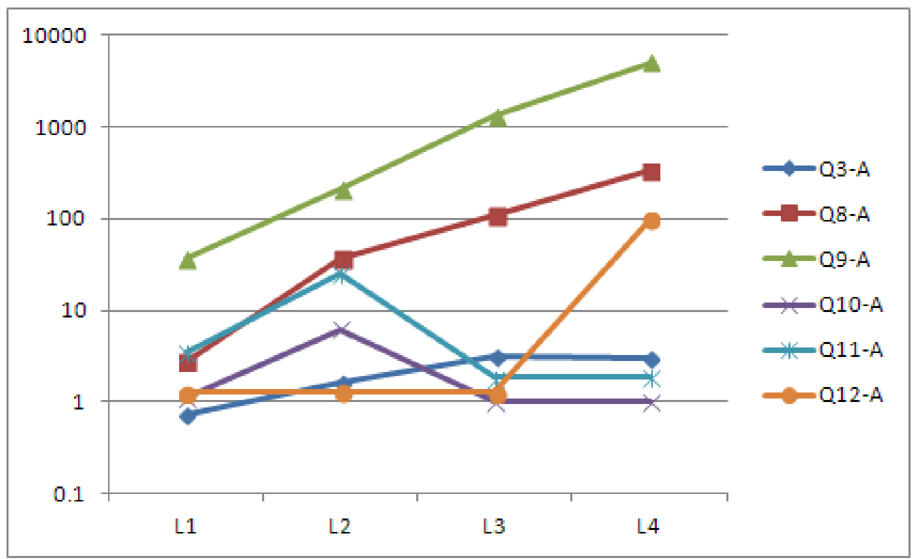

Figure 7.3: Execution time (ms) – APPROX L4All queries.

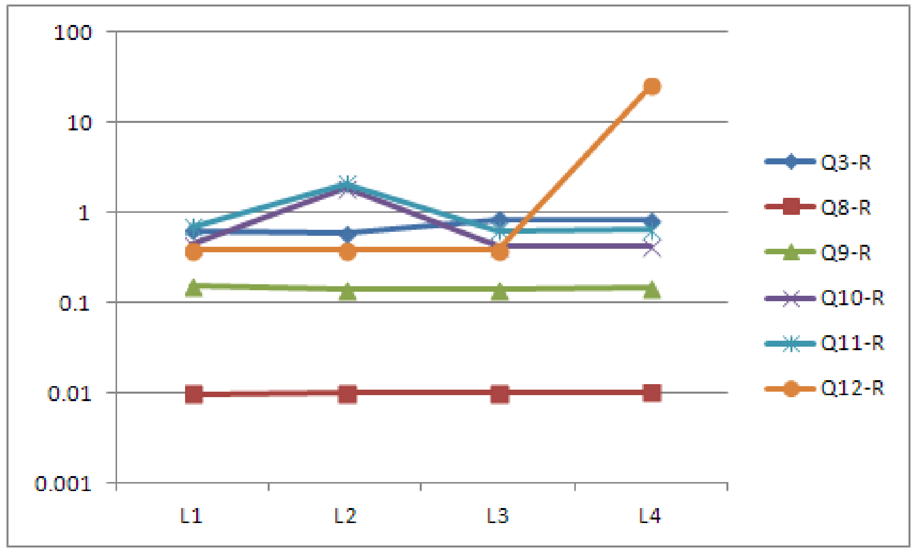

Figure 7.4: Execution time (ms) – RELAX L4All queries.

whereby some nodes in the larger data graphs had far more edges than in the smaller data graphs leading to more nodes being traversed and tuples being processed.

For the APPROX queries shown in Figure 7.3, queries 10 and 11 show a decrease in the time taken for **L3** and **L4** compared with **L2** which is caused by the fast processing of sufficient exact results for the larger two data graphs; query 3 shows a very minor increase of a few milliseconds in execution time. However, the APPROX

versions of queries 8, 9 and 12 exhibit a sharp increase in the time taken to retrieve the top 100 results. This is caused by a large number of intermediate results being generated (due to the `Succ` function returning a large number of transitions which are then converted into tuples in function `GetNext` and added to $\mathtt{D}_R$). We see that in some cases, such as query 9, the query becomes an order of magnitude slower when moving from one data graph to the next, even though the increase in the size of each successive data graph is at most a factor of 6. This is caused by changes in the connectivity patterns within the data, where the connections between nodes get progressively more dense.

The RELAX queries 3, 8, 9, 10 and 11 shown in Figure 7.4 all exhibit a fairly constant execution time across the data graphs. Query 12 shows an increase from **L3** to **L4** for similar reasons to its APPROX version.

We note that these results are consistent with the data complexity results shown in Proposition 4.1 for APPROX queries, and in Proposition 4.4 for RELAX queries, which are detailed in Chapter 4, in that we can observe a sharp increase in the execution time of some of the queries as the size of the data graph grows.

7.2 The YAGO evaluation

For our second evaluation, we used data from YAGO, the well-known semantic knowledge base [75]. We selected this data graph on account of the presence of differing connectivity patterns when compared with the L4All data — which is more 'linear' in structure owing to the timelines — in order to provide a contrasting basis on which to evaluate the performance of our queries. Additionally, the associated ontology differs from the L4All one in terms of breadth and depth.

7.2.1 Data

We downloaded the simpler taxonomy and core data facts from the YAGO website (the SIMPLETAX and CORE portions) and imported these into our system.[3]

The resulting data graph consists of 3,110,056 nodes and 14,406,857 (17,043,938) edges and consumes 1.57 GB (1.76 GB) of disk space, where the figures in brackets

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

denote the value for the closure of the graph (an increase of just over 12%). As for L4All, the closure of the data graph is materialised to disk.

There is only one classification hierarchy in the YAGO data graph. The *depth* (the length of the longest path from the root to the leaf nodes) is **2**, and the *average fan-out* (the average number of children of each non-leaf class) is **933.43**.

Including the `type` property, YAGO uses **38** properties. There are two property hierarchies, containing 2 and 6 subproperties respectively. The properties also have domains and ranges defined, but since these are not used in the processing of the queries, we do not discuss them further.

7.2.2 Queries

We executed 9 queries on the YAGO data in order to evaluate the performance of our operators, APPROX and RELAX, just as we did for the L4All data. The exact, APPROX and RELAX versions therefore give rise to 27 queries, for which we calculated the timings as described in Section 7.1, with the edit and relaxation costs the same as those used for the L4All evaluation. We exclude queries 7 and 8 from further discussion as running the exact form of these queries returned well over 100 results. Owing to the large number of results, neither the APPROX nor the RELAX operator was applied to them. Table 7.7 shows the remaining seven queries.

7.2.3 Baseline experimental results

The number of results obtained for the queries over the YAGO data graph are shown in Table 7.8. For each query, the exact version was run to completion, and the APPROX and RELAX versions were run until the top 100 answers were retrieved. The '?' indicates instances where the system ran out of memory and hence failed without returning any answers. For APPROX and RELAX queries yielding non-exact answers, we additionally show the distances of the answers followed by the number of answers in brackets.

Query	Details
Q1	`(Halle,_Saxony-Anhalt,bornIn^-.marriedTo.hasChild,?X)` All people who are the children of people married to someone born in `Halle, Saxony-Anhalt` are returned as answers. This query motivates the use of the APPROX and RELAX, as extra answers are returned.
Q2	`(Li_Peng,hasChild.graduatedFrom.graduatedFrom^-.hasWonPrize,?X)` The prizes won by people who graduated from the same institution as one of `Li Peng`'s children are returned as answers. This query motivates the use of APPROX as extra answers are returned, and also uses a more complex R.
Q3	`(wordnet_ziggurat,type^-.locatedIn^-,?X)` All entities located in a structure of type `wordnet_ziggurat` are returned as answers. This query motivates the use of APPROX and RELAX, as extra answers are returned; in particular, RELAX as applied to the `wordnet_ziggurat` subclass is demonstrated as being very useful owing to `wordnet_ziggurat` having a very low *type* degree (3).
Q4	`(?X,directed.married.married+.playsFor,?Y)` The purpose of this query is to find all pairs of people having some relationship governed by a path expressing at least two *marriages*, preceded by a notion of *directing*, and, finally, of *playing for* some sports team. There is an error in R (something which is *directed* cannot itself be *married* to someone) and this motivates the use of the APPROX operator. Furthermore, R itself describes a long path. Also, as this query contains no constants, this query additionally tests the effects of having to cater for the initialisation of a significant portion of the nodes in G (there are 41,811 *directed* edges).
Q5	`(?X,isConnectedTo.wasBornIn,?Y)` The purpose of this query is to find all entity-people pairs where the entity is *connected* to some location in which the person was *born*. There is thus an error in R, which therefore motivates the use for APPROX and RELAX. As this query contains no constants, this query also tests the effects of having to cater for the initialisation of a significant portion of the nodes in G (there are 33,834 $isConnectedTo$ edges).
Q6	`(?X,imports.exports^-,?Y)` This query finds all country pairs, where the first country *imports* something that the second country *exports*. As this query contains no constants, this query tests the effects of having to cater for the initialisation of a portion of the nodes in G; this time, fewer than in previous queries, as there are only 391 *imports* edges.
Q9	`(United Kingdom,(livesIn^-.hasCurrency)|(locatedIn^-.graduatedFrom),?X)` The query has an error in R: the $hasCurrency$ label is incorrect in the context of the query (as there is nothing that *lives* in the `United Kingdom` and has a *currency*); and the $graduatedFrom$ label's direction is incorrect (as there is nothing *located* in the `United Kingdom` which has also *graduated* from some institution). This query once again exemplifies the use of our APPROX and RELAX operators, with R consisting of a disjunction.

Table 7.7: The YAGO query set: Q1 - Q6 and Q9.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q9**
E	13	2	0	0	0	3,459	0
A	100 **1** *(87)*	100 **1** *(98)*	100 **1** *(5)* **2** *(95)*	?	?	100	100 **1** *(100)*
R	15 **1** *(2)*	2	100 **1** *(100)*	0	100 **1** *(100)*	100	100 **1** *(100)*

Table 7.8: The number of results per query for the YAGO data graph.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q9**
E	*0.12* 1.06	*0.13* 0.39	*0.10* 0.09	*50.20* 656.06	*38.37* 445.89	*0.55* 102.20	*0.16* 94.40
A	*0.19* 13,284.15	*0.23* 25,614.40	*0.15* 1,965.80	*?* ?	*?* ?	*0.58* 758.13	*0.21* 1,012.32
R	*0.12* 2.06	*0.14* 0.65	*2.96* 310.19	*49.66* 8,240.47	*38.66* 683.41	*0.55* 3.66	*0.16* 142.24

Table 7.9: Total initialisation and execution times (ms) for each query run over the YAGO data graph (initialisation time in italics).

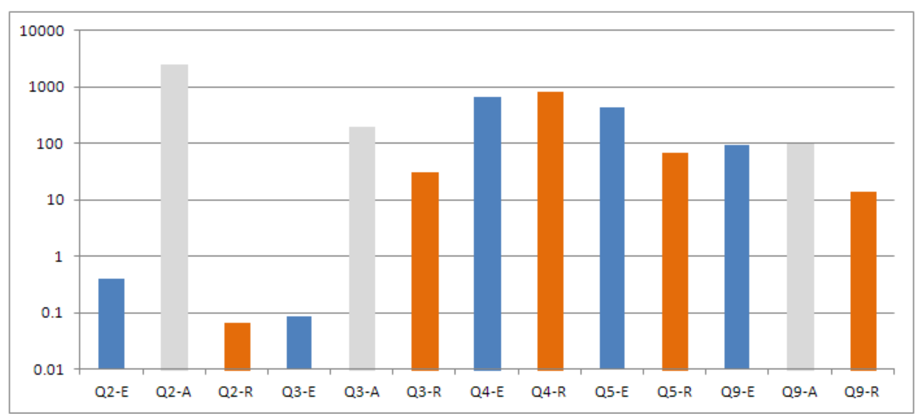

Figure 7.5: Execution times (ms), YAGO data graph.

7.2.4 Analysis

Figure 7.5 shows the average execution times for queries 2, 3, 4, 5 and 9. Query 1 showed a similar performance to query 2; query 6 is similar to queries 4 and 5 in terms of query structure, but its APPROX form terminated, unlike these. For completeness, we show in Table 7.9 the total initialisation time and the total execution time (i.e. the sum of batches 1-10 for the APPROX and RELAX versions) for queries 1, 2, 3, 4, 5, 6 and 9; the total initialisation time and total execution time is computed as an average across runs 2 to 5. We observe that the longer initialisation time for queries 4 and 5 – both of the form $(?X, R, ?Y)$ – are due to the fact that large numbers of nodes need to be added to $\mathtt{D}_R$ in the `Open` procedure. Moreover, the APPROX versions of queries 4 and 5 failed to terminate; this is elaborated upon later.

For the exact queries shown in Figure 7.5, queries 2 and 3 execute quickly. Queries 4 and 5 take longer to execute because their conjuncts are of the form $(?X, R, ?Y)$. Hence processing is initiated from a large number of nodes (41,811 and 33,834 respectively), and further traversal leads to large numbers of intermediate results (owing to the `Succ` function returning a large number of transitions which are then converted into tuples in function `GetNext` and added to $\mathtt{D}_R$). Query 9 takes longer to execute owing to its more complex regular expression (an alternation consisting of two concatenated labels).

Referring to Figure 7.5, APPROX queries 2 and 3 exhibit poor performance owing to a large number of intermediate results, while the APPROX form of query

9 takes the same time as the exact version. APPROX queries 4 and 5 failed to terminate as the system ran out of memory; this, too, is due to a large number of intermediate results. However, applying the APPROX — or RELAX — operator to conjuncts of the form $(?X, R, ?Y)$ resulting in the approximation (relaxation) of the path between *any* two nodes in the data graph may be of limited practical use. Rather, we anticipate a conjunct of this form only being used as part of a multi-conjunct query, preceded and succeeded by conjuncts containing a constant, just as in queries executed as part of the qualitative evaluation described in Chapter 5. In these cases, standard techniques such as 'sideways information passing' may be used [117].

Regarding Figure 7.5, RELAX queries 2, 3 and 9 performed competitively, returning more results for the latter two than their exact counterparts. The time taken for RELAX query 4 to execute was the same as for the exact version (with no extra results). RELAX query 5 returned results and executed faster than the exact version; this is due to 100 results being found (by the application of rules 2 and 4 from Figure 3.6) and execution terminating sooner.

7.3 Performance comparison

We now compare the performance of *Omega* to that of other systems whose implementation approaches were presented in Section 6.7, with the observation that the comparison is not strictly like-for-like, owing to the other systems having been benchmarked on different datasets and different hardware. We note that the performance of all the exact queries on the *Omega* system is competitive with the expected behaviour of native NFA-based approaches to regular path query evaluation [82], as discussed below.

The implementation of the DataGuides system [43] by Goldman and Widom — using an automaton-based approach — is no longer supported, and Koschmieder and Leser [82] have reported that it compared unfavourably with their implementation.

Zauner *et al.* [141] present RPL, a regular path language for RDF data, and provide an automaton-based implementation. However, according to Koschmieder and Leser [82], who compared their system against RPL, RPL was only able to process efficiently tiny graphs. Koschmieder and Leser [82] report that queries executed by

RPL against graphs both smaller and more sparse than our **L2** graph required 16 GB of main memory and returned results in a few seconds, and that all graphs larger than this were not able to be processed.

The search-based processing system of Koschmieder and Leser [82] breaks down a query so that each resulting sub-query either begins with or ends with a rare label. They conducted a performance evaluation using two real-world graphs, consisting of 50K nodes (340K edges) and 80K nodes (1 million edges), which are slightly smaller, respectively, than the sizes of our **L3** and **L4** graphs (with their second graph being more dense than ours). Experiments were also conducted on a series of artificially-created graphs, all of which were sparse compared with ours (the maximum node to edge ratio was 1:4, in contrast to ours, which has a ratio of 1:7 on average), and none of which was as large as our **YAGO** graph (the sizes of these ranged from 1K to 1 million nodes). Our queries perform very favourably against the results in [82], in most cases exhibiting faster execution times; their queries took between 500 and 800 milliseconds to execute over the two real-world graphs.

We previously described the TALE method for approximate subgraph matching [126] in Section 2.5. The paper describes experimental results using graphs smaller than our **L1** graph, our smallest graph.

The R^2DF framework, presented by Cedeño and Candan [19], allows weighted RDF data to be queried in a cost-aware manner. The authors conduct a performance study over AR2Q using two graphs that have, respectively, 9K nodes (24K edges) and 10K nodes (25K edges), both of which are smaller and more sparse than our **L2** graph.

The implementation described by Dey *at al.* [28], in which RPQs are translated into either Datalog queries or recursive SQL, uses graphs ranging between 1.1K and 430K nodes, with the number of edges being approximately twice the number of nodes. Our queries perform favourably against the results in [28], in which queries using regular expressions of the form $(a|b)+$ took a few seconds to execute.

The reachability indexing approach underpinning the implementation presented by Gubichev *et al.* [51] for evaluating CRPQs uses an extended version of the YAGO dataset, YAGO2S [62], containing 100 million triples. Their queries — using single-hop paths and the closure operator — performed favourably, taking a few milliseconds to 300 milliseconds to execute.

7.4 Approximated Regular Path Query Optimisation

In the previous sections, we presented performance results conducted on two data graphs. Although we showed a reasonable baseline performance of exact queries when compared with that of other state-of-the-art implementations, some approximated queries — in particular — performed poorly. We describe in this section an optimisation approach designed to improve the performance of such queries.

In Section 7.4.1 we introduce *path indexes*, the concept underpinning our optimisation approach. This is followed in Section 7.4.2 by a description of our optimisation approach, detailing the queries selected and what statistics are required. We discuss the experimental results in Section 7.4.3, and conclude this section with directions for future work in Section 7.4.4.

7.4.1 Path indexes

Recent work [41, 108, 124] has shown that the presence of *path indexes* greatly improves the performance of path-based queries, including RPQs.

Assume a path p in a data graph $G = (V_G, E_G, \Sigma \cup \texttt{type})$ is given by the sequence of edge labels $(l_1, l_2, \ldots, , l_n)$ where $n \geq 0$ and for each l_i, drawn from $(\Sigma \cup \texttt{type})$, either $l_i \in E_G$ or $l_i^- \in E_G$. A *path index* is a data structure in G which returns all pairs of nodes (v, n), where $v, n \in V_G$, such that p is a (semi)path from v to n in G. The work in [41, 108] shows how regular expressions can be compiled into *path queries*, i.e. sequences of edge labels, so that RPQs may be able to be evaluated efficiently through the use of path indexes. In particular, the authors describe how the Kleene operator '*' in an RPQ can be replaced by bounded recursion through the use of a lower and an upper bound, so that query plans comprise just a union of path queries.

Sumrall [124] presents a simple path-oriented index designed and implemented using a B^+ tree. Empirical evaluation with respect to the property graph database Neo4j[4] shows that the performance of path queries improves by several orders of magnitude when using the index.

[4] `http://neo4j.com/`

7.4.2 Approach and methodology

Taking the research on path indexes as an inspiration, we aim to establish whether using simple statistics that estimate the number of answers to RPQs arising from query approximation in our framework would be a promising direction to explore. In particular, we focus on RPQs of the form $(?Y) \leftarrow (C, R, ?Y)$, where C is a constant.

We can use the techniques of [41, 108] to determine the number of pairs of nodes in a graph matching the triple pattern $(?X, R, ?Y)$, the number of distinct bindings for the variable $?X$, and also the actual values for the bindings of $?X$[5]. For any triple pattern $(C, R, ?Y)$, we compute all the rewritten versions of the regular expression R at each distance greater than zero required to retrieve at least 100 answers[6] for the query. Then, the statistics we use in our optimisation approach described below are, for each rewritten query $(?X, ?Y) \leftarrow (?X, R', ?Y)$ of the original query: the total number of answers returned; the unique bindings for $?X$; and the average number of answers per binding. Moreover, only the rewritten queries that contain C in their answer set are considered.

The worst-performing queries arising from our performance analysis, described in Sections 7.1 and 7.2, are the APPROX queries. Thus, we do not consider RELAX queries for the purposes of this evaluation, but note that similar techniques may be applied to these queries.

We selected the worst-performing terminating APPROX query from both the largest L4All data graph, **L4**, and the YAGO data graph to use in our optimisation evaluation.

The query from the L4All data graph is given by Q_1 below (we note that this is $Q9$ in Section 7.1), which took on average 4,930ms per batch to return 100 answers in 10 batches consisting of 10 answers each. Executing the query returned 1 exact answer, 32 answers at distance 1 and 67 answers at distance 2.

```
(?X) <- (Alumni 4 Episode 1_1,prereq*.next+.prereq,?X)
```

[5]We do not at this time consider the cost of determining these numbers and we leave as future work the investigation of the trade-offs between the costs and benefits of the query optimisation process itself.

[6]Following the same process used in Sections 7.1 and 7.2, we select '100' to be the maximum number of answers to retrieve (we envisage APPROX and RELAX queries being used to retrieve the top k results in an incremental manner).

The query from the YAGO data graph is given by Q_2 below (we note that this is $Q2$ in Section 7.2), which took on average 2,508ms per batch to return 100 answers in 10 batches consisting of 10 answers each. Executing the query returned 2 exact answers and 98 answers at distance 1.

```
(?X) <- (Li_Peng,hasChild.graduatedFrom.graduatedFrom-.hasWonPrize,?X)
```

We recall that the closure of the **L4** L4All data graph consists of 240,519 nodes and 1,861,959 edges, and that the closure of the YAGO data graph consists of 3,110,056 nodes and 17,043,938 edges.

We now describe which statistics are computed for our experiments here[7]:

- For both Q_1 and Q_2, we compute all the rewritings of the regular expression R at each distance greater than zero required to retrieve at least 100 results for the query. To illustrate this process, we focus on Q_1, so that $R = prereq * .next + .prereq$, G is the L4All data graph, and the set of labels $L = \{prereq, next, qualif, level, job, sector, \texttt{type}\}$.

  We compute all the rewritings of R at distance 1 by applying either one insertion, one deletion or one substitution operation to R using a label drawn from L, where each operation has a cost of 1. All these 'distance 1' rewritings of R are given by the set $R_1(Q_1)$ for Q_1. For example, $R_1(Q_1)$ contains, among others, '$prereq * .next + .prereq.qualif$' (insertion of $qualif$), '$prereq * .next+$' (deletion of the second instance of $prereq$) and '$prereq * .next + .next$' (substitution of the second instance of $prereq$ by $next$).

  As the evaluation of Q_1 gave answers at distance 2 in the top 100 results, the same process as described above is used to compute all the rewritings of R at distance 2. In this case, a single insertion, deletion or substitution is applied to each item in $R_1(Q_1)$, and the result is added to the set $R_2(Q_1)$. Owing to Q_2 not having answers at distance 2 in the top 100 results, $R_2(Q_2)$ does not need to be constructed for this query.

- For each item r in $R_1(Q_1)$, $R_1(Q_2)$ and $R_2(Q_1)$, we do the following:

  We execute the *exact* version of the query using r, and with distinct variables as the source and target nodes. For example, if $r = prereq * .next+$, then the

[7] We recall that in reality these would form part of the graph database statistics.

query `(?A,?B) <- (?A,prereq*.next+,?B)` is executed. This gives the set of answers in G conforming to each r; we denote by $A(r)$ this set of answers. This also gives us the set of unique bindings for $?A$ which we denote by $N(r)$. The average number of answers for each binding for $?A$ is given by $n_{avg}(r) = |A(r)|/|N(r)|$.

Finally, if the constant C of the original query is not contained in $N(r)$, then r is removed from $R_1(Q_1)$, $R_1(Q_2)$ and $R_2(Q_1)$, yielding, respectively, $R_{1,C}(Q_1)$, $R_{1,C}(Q_2)$ and $R_{2,C}(Q_1)$. Clearly, these items are not going to yield answers to our queries.

7.4.3 Experimental results

For Q_1 (respectively Q_2), we rewrote the original query by substituting the original regular expression R in the query by each item in $R_{1,C}(Q_1)$ (respectively $R_{1,C}(Q_2)$), resulting in $|R_{1,C}(Q_1)|$ (respectively $|R_{1,C}(Q_2)|$) new queries, each perturbed by one edit operation from the original R. For Q_1, the same process was repeated for each item in $R_{2,C}(Q_1)$.

Thus, $|R_{1,C}(Q_1)|+|R_{2,C}(Q_1)|$ new queries associated with Q_1 and $|R_{1,C}(Q_2)|$ new queries associated with Q_2 were generated, with the list of queries in each case given by $W(Q_1)$ and $W(Q_2)$, respectively. These lists are ordered by increasing distance — i.e. all queries using the items in $R_{1,C}(Q_1)$ are ordered before those generated using the items in $R_{2,C}(Q_1)$ — and within that in order of decreasing $n_{avg}(r)$ value. This ordering is used so that the queries most likely to return the most answers are executed first, thus reaching the 'top-k' number (100, in this case) as soon as possible.

We subsequently followed a very similar methodology to the one used for our performance study in Sections 7.1 and 7.2. All experiments were run on an Intel Core i7-950 (3.07-3.65GHz) with 6GB memory, running Windows 7 (64 bit).

Each query q in $W(Q_1)$ and $W(Q_2)$ was executed five times as an *exact* query, with the first run being discarded as the cache warm-up. All five runs were executed in 10 batches of 10 to simulate running APPROX to retrieve the top 100 results after initialisation; i.e. obtain results 1–10 ('batch 1'); obtain results 11–20 ('batch 2'); . . . ; obtain results 91–100 (batch 10). We executed each q in the order in which it

	Q_1	Q_2
Original time	50,668ms	25,614ms
t_{Q_i}	29ms	10.4ms
$\lvert R_1(Q_i)\rvert$	124	679
$\lvert R_{1,C}(Q_i)\rvert$	22	31
$\lvert R_2(Q_i)\rvert$	8,075	N/A
$\lvert R_{2,C}(Q_i)\rvert$	257	N/A
No. q ($\lvert R_{1,C}(Q_i)\rvert$)	22	20
No. q ($\lvert R_{2,C}(Q_i)\rvert$)	1	N/A

Table 7.10: Results of using the statistics to simulate the running of Q_1 and Q_2 as approximated queries.

appears in $W(Q_1)$ and $W(Q_2)$, and halted processing as soon as at least 100 results were returned. The average of each of the 10 batches was taken across runs 2 to 5 to obtain an average for each batch. These 10 average batch times were then added to yield the total time taken for each query q to execute, which is denoted by q_t. For Q_1 and Q_2, all the q_t values are summed, and this total is then added to the total time taken for the exact, original version of the query (i.e. with the original R) to execute. This gives us the total time taken for Q_1 and Q_2 to execute, which we'll denote by t_{Q_1} and t_{Q_2}, respectively.

The results are shown in Table 7.10. The '**Original time**' column shows the total time taken for the the original, approximated version of Q_1 and Q_2 to execute, which is computed in the same way as q_t described above. The '**No.** q ($|R_{iC}(Q_i)|$)' columns show the number of queries at each distance that needed to be run in order to return at least 100 answers for Q_1 and Q_2.

We can see that the execution time for each of Q_1 and Q_2 has improved by three orders of magnitude. This is caused by the `Succ` function returning fewer transitions — compared to running the APPROX version of the query — leading to correspondingly fewer tuples being enqueued on $\texttt{D}_R$.

7.4.4 Further work

For this preliminary investigation into the use of simple statistics as a means to optimise approximated queries, we selected the worst-performing (terminating) query from both the largest L4All data graph and the YAGO data graph. However, this is by no means an exhaustive evaluation, and future work includes conducting a more extensive study into the execution performance of the remaining approximated queries and all of the relaxed versions of the queries. Leading on from this, more evaluations can be conducted using a wider range of approximated and relaxed queries over different data graphs.

Another direction which requires further investigation is the performance of CRPQs rather than just single-conjunct queries that may be approximated or relaxed.

Future work could include making use of disk-resident data structures for $\texttt{queue}_R$ to guarantee the termination of APPROX queries with large intermediate results. Another promising direction is to use labels that are rare in the graph to split the processing of a regular expression into smaller fragments such that either their first or last label is a 'rare' label, as described in [82] (but not for approximated/relaxed queries). Most recently, the Waveguide system [137] uses automata as part of a cost-based optimiser for SPARQL regular path queries. This cost-based optimisation approach may be applicable also to the approximated/relaxed evaluation of CRPQs, and this would be an interesting area of further investigation.

7.5 Concluding remarks

In this chapter, we presented performance results conducted on two data graphs, each sourced from a different domain and exhibiting dissimilar characteristics. A set of queries was evaluated on each data graph, and the results of each were discussed. We established that our baseline performance of exact queries is comparable to that of other state-of-the-art system implementations, and that, in many cases, our approximated and relaxed queries executed quickly. However, this was not the case for some of the approximated queries, in particular. To address this, we presented optimisations designed to improve further the performance of such queries, showing that this was a promising and viable direction for future work.

In the next chapter, we describe how it is possible to merge the APPROX and RELAX operators into a single operator, called FLEX, which applies simultaneously both approximation and relaxation to a query conjunct.

CHAPTER 8

The FLEX Operator

In Chapter 7, we presented performance results conducted on two data graphs, and described an optimisation approach designed to improve further the performance of some of the poorly-performing approximation queries.

In this chapter, we consider the merging of APPROX and RELAX operators into one integrated FLEX operator that applies both approximation and relaxation to a query conjunct.

In Section 8.1, we show there are answers returned by CRPQs using FLEX semantics which cannot be returned by any CRPQ using APPROX/RELAX semantics. We discuss how the evaluation of single-conjunct RPQs that have FLEX applied to them can be undertaken using a combination of the techniques used for approximation and relaxation of single-conjunct queries. The evaluation is again accomplished in time that is polynomial in the size of the query, the data graph and the ontology graph, with answers being returned in ranked order.

In Section 8.2, we discuss the characteristics of multi-conjunct RPQs in which conjuncts can have FLEX applied to them, considering query evaluation, complexity, and expressiveness, and we conclude the chapter with Section 8.3.

8.1 Evaluation of single-conjunct FLEX queries

We now combine the use of both query approximation and query relaxation within one integrated FLEX operator that applies both techniques at the same time. This aims to allow greater ease of querying for users, in that they do not need to be aware of the ontology structure and to identify explicitly which parts of their overall query are amenable to relaxation and which to approximation. As we will see below, it also allows answers to be returned that cannot be obtained by applying only APPROX or RELAX to individual query conjuncts.

Definition 8.1. Let $K = \texttt{extRed}(K)$ be an ontology and Σ be an alphabet of edge labels. A *flex* operation is either an edit operation on a symbol in $\Sigma \cup \Sigma^-$ or a direct relaxation using K.[1]

□

Example 8.1. (Drawn from [114].) Referring to the data graph shown in Figure 3.1 and the ontology shown in Figure 3.2 in Section 3.1, the user may pose the following query which uses the new FLEX operator to apply both approximation and relaxation simultaneously to both conjuncts:

$$Y \leftarrow FLEX(\text{'FL56'}, fn_1.ppn_1.pn_1^-, Y), FLEX(Y, n_1.type, N_1)$$

By replacing fn_1 by fn_1^- and inserting ie_1 after pn_1^- (at a cost of $c_s + c_i$), the result e_1 is returned. By replacing fn_1 by fn_1^-, relaxing pn_1^- to pn^-, replacing n_1 by n_2, and relaxing N_1 to N, (at an overall cost of $2c_s + c_{r2} + c_{r4}$), the result p_2 is returned; the paths in G matched by the two conjuncts are, respectively, ('FL56', fn_1^-, f_1, ppn_1, '6789', pn_2^-, p_2) and ($p_2, n_2, n_2, type, N_2$). We note that result p_2 could not have been returned by applying only APPROX or RELAX to the two conjuncts — it requires the FLEX operator in order to be returned.

□

Definition 8.2. (We refer the reader back to Definitions 3.3 - 3.5 for the definitions

[1]Edit operations on labels in {`type`, `type`$^-$} are not allowed for FLEX because, as we will see below, allowing such edits would require multiple rounds of approximation and relaxation to be applied to yield a final automaton, rather than a simple two-step process that we describe below. An investigation into reinstating `type` and `type`$^-$ for consideration of this aspect of FLEX is the subject of future work in this area.

of a semipath and a triple form, and Section 3.5.1 for the definitions of $\texttt{extRed}(K)$ and $closure_K(G)$.) Let $K = \texttt{extRed}(K)$ be an ontology, $G = closure_K(G)$ a graph, p a semipath in G, Q a query with single conjunct (X, R, Y), θ a (Q, G)-matching, $q \in L(R)$, T_q a triple form for $(\theta(Q), q)$, and T_p a triple form for p. We write $T_q \preceq T_p$, if T_q can be transformed to T_p (up to variable renaming) by a sequence of flex operations. The *cost* of the sequence of flex operations is the sum of the costs of each operation. The *distance* from p to $(\theta(Q), q)$ is the minimum cost of any sequence of flex operations which yields T_p from T_q. The cost of the empty sequence of flex operations (so T_q is already a triple form of p) is zero. The *distance* from p to $\theta(Q)$ is the minimum distance from p to $(\theta(Q), q)$ for any string $q \in L(R)$.

□

Definition 8.3. Given ontology $K = extRed(K)$, graph $G = closure_K(G)$, single-conjunct query Q to which FLEX has been applied, and (Q, G)-matching θ, the *distance* of $\theta(Q)$, denoted $dist(\theta, Q)$, is the minimum distance to $\theta(Q)$ from any semipath p in G. The *answer* of Q on G is a list of pairs $(\theta(vars), dist(\theta, Q))$, where θ is a (Q, G)-matching, ranked in order of non-decreasing distance. The *top-k answer* of Q on G is a list containing the first k tuples in the answer of Q on G.

□

Example 8.2. (Drawn from [114].) Consider the conjunct $(\text{'FL56'}, fn_1.ppn_1.pn_1^-, Y)$ of the query from Example 8.1. There is only one sequence q of labels denoted by the regular expression of the conjunct, so a triple form of q is:

$$(\text{'FL56'}, fn_1, X_1), (X_1, ppn_1, X_2), (X_2, pn_1^-, Y)$$

Replacing fn_1 by fn_1^- and inserting ie_1 after pn_1^- gives rise to

$$(\text{'FL56'}, fn_1^-, X_1), (X_1, ppn_1, X_2), (X_2, pn_1^-, X_3), (X_3, ie_1, Y)$$

with cost $c_s + c_i$. This will match the semipath p from 'FL56' to e_1 in the graph of Figure 3.1, thereby instantiating Y to e_1. Since p cannot be matched by q with fewer flex operations, the distance from p to q is $c_s + c_i$. For example, relaxing pn_1^- to pn^- will also instantiate Y to e_1, but at a cost of $c_s + c_i + c_{r2}$.

□

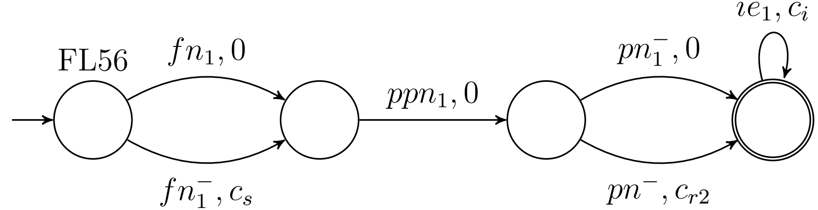

Figure 8.1: Automaton for conjunct (‘FL56’, $fn_1.ppn_1.pn_1^-, Y$).

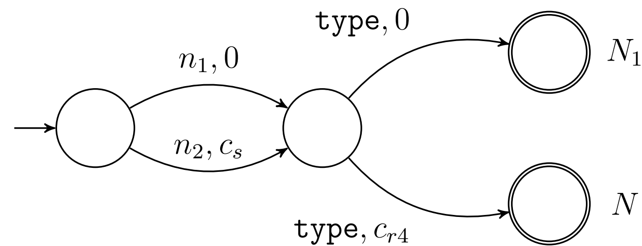

Figure 8.2: Automaton for conjunct ($Y, n_1.\texttt{type}, N_1$).

The answer of a single-conjunct query Q to which FLEX has been applied on a graph G can be computed by using an automaton A_Q^K constructed from the approximate automaton A_Q and ontology K which captures both approximation and relaxation with respect to K, as follows:

Step 1: We construct a weighted automaton M_R from R (in which all weights are zero), and then the approximate automaton A_Q, using essentially the same process as described in Section 3.4.1 (we describe the distinction after the following example).

Step 2: We construct the relaxed automaton A_Q^K from A_Q, applying relaxation using rules 2, 4, 5 and 6 from Figure 3.6, following the same process as described in Section 3.5.2.

Example 8.3. (Drawn from [114].) Figure 8.1 shows the automaton corresponding to the conjunct (‘FL56’, $fn_1.ppn_1.pn_1^-.Y$) of the query from Example 8.1 (only those transitions that contribute to finding answers in the data graph are shown).

The transition with cost c_s results from replacing fn_1 with fn_1^- and the transition with cost c_i results from inserting ie_1. The transition with cost c_{r2} results from applying rule (2) from Figure 3.6 to the transition for pn_1^- and the triple (pn_1, $\texttt{sp}$, pn) in K.

The automaton for conjunct $(Y, n_1.\texttt{type}, N_1)$ is shown in Figure 8.2. In this case, the type transition with cost c_{r4} has been added as a result of applying rule (4) to the other type transition using triple $(N_1, \texttt{sc}, N)$ from K, which results in a cloned final state annotated with N.

□

In this section, we focus on single-conjunct RPQs and consider CRPQs in Section 8.2. The answer to a single-conjunct RPQ Q is obtained by constructing and traversing the weighted product automaton, H, of A_Q^K with the closure of the data graph $G = (V_G, E_G, \Sigma)$, viewing each node in V_G as both an initial and final state: the process is the same as described in Section 3.4.1. To evaluate Q on G, if X is a node $v \in V_G$, a shortest path traversal of H is undertaken starting from each state (s_0, v) such that $s_0 \in S_0$. If X is a variable, these shortest path traversals are undertaken for each $v \in V_G$. In each case, the answers to Q on G are given by the bindings for Y found from the final states reached during the traversal of H.

As stated earlier, the automaton A_Q^K is constructed by applying relaxation to the approximate automaton A_Q. However, in contrast to Definition 3.11 in Section 3.4.1, the edit operations used in the construction of A_Q are confined to those on labels in $\Sigma \cup \Sigma^-$, i.e., we do not allow edits to the labels in $\{\texttt{type}, \texttt{type}^-\}$. Henceforth in this chapter, we make this assumption about the approximate automaton A_Q. Allowing edits to the labels in $\{\texttt{type}, \texttt{type}^-\}$ would require multiple rounds of approximation and relaxation to be applied to yield a final automaton, rather than the simple two-step process described above. Determining an upper bound for the number of rounds of approximation/relaxation that would be needed for the construction of the automaton to reach a fixed point is an open problem. This is illustrated in the following example.

Example 8.4. Suppose we have the query Q with single conjunct $(?X, e.\texttt{type}, c)$, the labels $a, b, e \in \Sigma$, and the triples $(c, \texttt{sc}, c') \in K$ and $(a, \texttt{sp}, b) \in K$. The two transitions with cost zero in Figure 8.3 arise from the original query Q. Approximation would add various transitions not shown in Figure 8.3. Then in the relaxation phase, we apply rule 4(i) which creates a final state s_f', annotated with c', and a transition $(s_1, \texttt{type}, c_{r4}, s_f')$. However, the presence of $(s_1, \texttt{type}, c_{r4}, s_f')$ means we are able to apply more edit operations to the automaton to produce, say, the transition

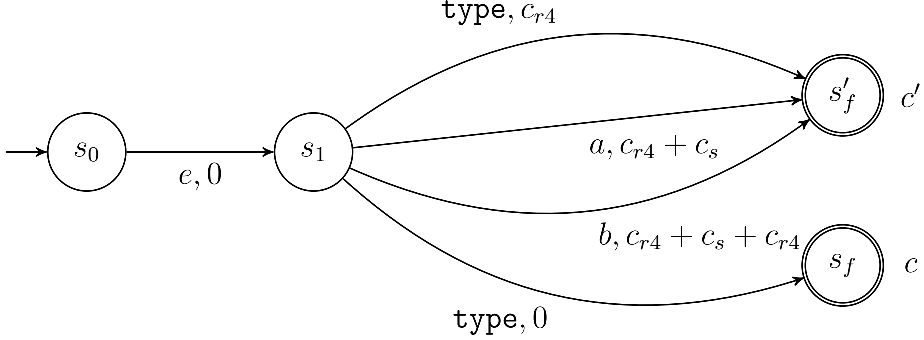

Figure 8.3: Automaton for $(X, e.\texttt{type}, \text{`c'})$.

$(s_1, a, c_{r4} + c_s, s'_f)$ by replacing `type` with a. But, by once again applying relaxation operations to the automaton, a new transition $(s_1, b, c_{r4} + c_s + c_{r2}, s'_f)$ induced by rule 2 would be created. The resulting automaton is shown in Figure 8.3 (only those transitions and states explicitly mentioned are shown).

Thus, for the purposes of the FLEX operator, we do not allow edit operations on `type` or `type`$^-$ labels.

□

In Theorem 8.1 below, we show that using automaton A_Q^K is sufficient to find all sequences of labels generated by flex operations at distance k from a given query; i.e. that applying a second approximation step after the relaxation step does not (i) yield any additional answers, and (ii) yield any answers previously obtained at cost k, at some cost $j < k$.

We first have the following lemma that shows that for semipath p, sequence of labels $q \in L(R)$, and (Q, G)-matching θ such that the distance from p to $(\theta(Q), q)$ is k, there is a cost-k sequence of flex operations yielding triple form T_p from triple form T_q in which all edit operations precede all relaxation operations.

Lemma 8.1. *Let Q be a query comprising a single conjunct (X, R, Y), $K = \texttt{extRed}(K)$ be an ontology, $G = (V_G, E_G, \Sigma)$ be the closure of a data graph with respect to K, p be a semipath $(v_0, l_1, \ldots, l_n, v_n)$ in G, θ be a (Q, G)-matching such that $\theta(X) = v_0$ and $\theta(Y) = v_n$, and $q \in L(R)$.*

Let T_p be a triple form for p and T_q a triple form for $(\theta(Q), q)$ such that the distance from p to $(\theta(Q), q)$ is k. There is a sequence of flex operations of cost k,

yielding T_p from T_q, in which all the edit operations (applied to symbols in $\Sigma \cup \Sigma^-$) precede all the relaxation operations.

Proof. Let the sequence of flex operations of cost k, yielding T_p from T_q, consist of n flex operations, where $n \leq k$. Let this sequence, $flex_n$, be given by $T_q = P_0 \preceq P_1 \preceq \cdots \preceq P_n = T_p$, in which the edit and relaxation operations have been applied in an arbitrary order. We need to show that there is an alternative sequence of flex operations, $flex'_n$, also of cost k, yielding T_p from T_q, and comprising the same operations as those in $flex_n$, but where all edit operations (considering only symbols in $\Sigma \cup \Sigma^-$) have been applied prior to all relaxation operations. That is, $flex'_n$ is given by $T_q = P_0 \preceq_A \cdots \preceq_A P_\lambda \preceq_R \cdots \preceq_R P_n = T_p$. The proof proceeds by induction on the number of flex operations applied to T_q.

Basis: For the base case, we assume no flex operations have been applied; hence, $T_q = P_0 = T_p$ and all edits precede all relaxations.

Induction: For the inductive step, suppose that there is an $n \geq 0$ such that, for all $m \leq n$, any sequence of m flex operations of cost k yielding T_p from T_q can be rewritten as a sequence (also yielding T_p from T_q at cost k) in which all edit operations precede all relaxation operations.

Now consider sequence $flex_{n+1}$ of $n + 1$ flex operations in which the last is an edit operation. We show that this edit operation can be moved before all the relaxations. If there are no relaxations in the sequence, this is trivial, so assume there is at least one relaxation. By the induction hypothesis, the sequence up to P_n can be rewritten so that relaxations follow edits. Hence $flex_{n+1}$ can be rewritten as ψ given by $T_q = P_0 \preceq \cdots \preceq_R P_n \preceq_A P_{n+1} = T_p$ where $n + 1$ flex operations have been applied to the sequence and the $n + 1^{th}$ operation is an edit operation, denoted by op_E.

First suppose op_E is applied to a triple pattern present in T_q. Clearly, op_E can be applied to P_0 and hence can precede relaxations.

Next suppose op_E is applied to a triple pattern resulting from an edit operation. As this operation precedes all relaxations, op_E can follow it directly and hence it too can precede all relaxations.

In the case where op_E is an insertion operation, it is straightforward to see that the result also follows, as insertion is not dependent on the presence of any triple pattern and so can be placed before all relaxations.

Thus, we now need to consider the cases in which op_E is a *substitution* or *deletion* operation applied to a triple pattern t of triple form P_n, such that t was the result of having applied some relaxation operation op_R to some triple pattern t' in P_{n-1}. We show below that in fact these are not possible, given that the distance from p to $(\theta(Q), q)$ is k.

We note that t may only be in one of the following forms, depending on which relaxation operation was applied: (i) (W_{m-1}, a, W_m), where W_{m-1} and W_m are variables or constants, (ii) $(W, \texttt{type}, c)$, or (iii) $(c, \texttt{type}, W)$, where W is a variable and c a constant. As edit operations are not applied to the `type` label, we only need consider what happens when applying op_E to P_n if t is of the form (W_{m-1}, a, W_m). By definition, such a t could only have been produced as a result of applying the relaxation operation induced by Rule 2 to t' in P_{n-1}; thus, op_R may only ever be a Rule 2 relaxation operation.

Suppose that $t' = (W_{m-1}, b, W_m)$ and $t = (W_{m-1}, a, W_m)$, where there is a triple $(b, \texttt{sp}, a) \in K$. Let the cost of the sequence ψ be $k = C + c_{r2} + c_e$, where c_{r2} denotes the cost of op_R, c_e denotes the cost of op_E (and is thus either c_s or c_d) and C is the cost of the remaining operators used in ψ. We now consider the following possibilities for op_E applied to t:

- *Substitution:* Suppose op_E substitutes t in P_n by (W_{m-1}, e, W_m) in P_{n+1}, for some $e \in \Sigma$. However, we could replace op_R by a substitution of b by e in t' in order to obtain (W_{m-1}, e, W_m) and hence T_p at a cost of $C + c_s < k$. This contradicts the assumption that the distance from p to $(\theta(Q), q)$ is k; hence, op_E cannot be substitution.

- *Deletion:* Suppose P_n is given by

$$\cdots, (W_{m-2}, g, W_{m-1}), t = (W_{m-1}, a, W_m), (W_m, f, W_{m+1}), \cdots$$

  and op_E deletes t to obtain P_{n+1}, given by

$$\cdots, (W_{m-2}, g, W_{m-1}), (W_{m-1}, f, W_{m+1}), \cdots$$

  However, we could replace the application of op_R on t' (in P_{n-1}) by deleting t' instead, in order to obtain a sequence yielding T_p at cost $C + c_d < k$. This

contradicts the assumption that the distance from p to $(\theta(Q), q)$ is k; hence, op_E cannot be deletion.

Thus, we have shown that, for all allowable operations op_E, the sequence ψ can be rewritten to a sequence of the same cost in which all edit operations precede all relaxation operations.

□

Theorem 8.1. *Let Q be a query comprising a single conjunct (X, R, Y) and $K =$ `extRed`(K) be an ontology. Let A_Q^K be the automaton constructed for Q as described above, where the ϵ-transitions have been removed. Let $G = (V_G, E_G, \Sigma)$ be the closure of a data graph with respect to K, H be the product automaton of G and A_Q^K, p be a semipath $(v_0, l_1, \ldots, l_n, v_n)$ in G, and θ be a (Q, G)-matching such that $\theta(X) = v_0$ and $\theta(Y) = v_n$. The distance from p to $\theta(Q)$ is k if and only if k is the minimum cost of a run for the sequence of labels comprising p from (s_0, v_0) to (s_n, v_n) in H, where s_0 is an initial state and s_n a final state in A_Q^K.*

Proof. ($\Rightarrow$) By Lemma 8.1, we know that if the distance from p to $(\theta(Q), q)$ is k, for any $q \in L(R)$, then k is the minimum cost of any sequence of flex operations yielding the triple form T_p from the triple form T_q, where the flex operations have been applied in an analogous manner to the construction of A_Q^K.

The result then follows from the construction of A_Q^K, and by Lemma 4.3 in Section 4.1 (as A_Q^K contains A_Q as a subautomaton) and Proposition 4.2 in Section 4.3 (as A_Q^K contains M_Q^K as a subautomaton).

($\Leftarrow$) Let $r = ((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ be a minimum cost run of cost k in H for the sequence of labels $l_1, \ldots, l_n$ of p, where s_0 is an initial state and s_n a final state in A_Q^K. From the construction of H from A_Q^K and G, there must be a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G and a minimum cost run $(s_0, l_1, c_1, s_1), \ldots, (s_{n-1}, l_n, c_n, s_n)$ of cost k in A_Q^K, corresponding to T_p, a triple form of p. From the construction of A_Q^K, we also know that the transitions added to those transitions originally present in M_R correspond to flex operations. By definition, we also know that every run in M_R corresponds to an acceptance of a sequence of labels $q \in L(R)$. Let the triple form of such a q be T_q.

But, by Lemma 4.3 (for edit operations) and Proposition 4.2 (for relaxation operations), it follows that the minimum cost of any sequence of flex operations

yielding T_p from T_q is k. The result then follows straightforwardly from Lemma 8.1. □

The following proposition shows that if FLEX has been applied to a single-conjunct query Q, the answers on the closure of a graph G can be computed in time that is polynomial in the size of Q, G and the ontology K.

Proposition 8.1. *Let* $K = \texttt{extRed}(K)$ *be an ontology, where* $K = (V_K, E_K)$, $G = (V_G, E_G, \Sigma)$ *be the closure of a data graph with respect to* K*, and* Q *be a single-conjunct query using regular expression* R *over alphabet* $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$*. If FLEX has been applied to* Q*, the answer of* Q *on* G *can be found in time* $O(|R|^3|V_G||V_K||E_G||E_K|(|V_K| + |\Sigma|) + |R|^2|V_G|^2|V_K|^2(log|R||V_G||V_K|))$.

Proof. Let A_Q^K be the automaton constructed from Q and K which captures both approximation and relaxation with respect to K, and H be the product graph constructed from A_Q^K and G. Lemma 4.3 and Proposition 4.2 show that traversing H correctly yields all answers to Q. Lemma 4.4 tells us that A_Q has at most $2|R|$ states and $4|R|^2|\Sigma|$ transitions.

By the construction of A_Q^K from A_Q, the application of rules 4, 5 and 6 results in at most one new state for each class node in V_K being added for any existing state s, where $s \in S_0$ or $s \in S_f$. Hence, we can see that no more than $|V_K|$ new states may be added for each of the original states in A_Q, resulting in at most $2|R||V_K|$ new states in total. Thus, A_Q^K has at most $2|R|(|V_K| + 1)$ states.

Since there are at most $|E_K|$ edges in K with label sp, rule 2 adds at most $4|R|^2|\Sigma||E_K|$ transitions to A_Q^K. Rules 4, 5 and 6 can collectively be applied no more than $|E_K|$ times. Each application results, in the worst case, in $|R|$ transitions being added for each of the $2|R||V_K|$ new, cloned states, giving rise to at most $2|R|^2|V_K||E_K|$ transitions. Thus, overall A_Q^K has at most $2|R|^2(|E_K||V_K| + 2|\Sigma| + |\Sigma||E_K|)$ transitions.

Therefore H has at most $2|R||V_G|(|V_K|+1)$ nodes and $2|R|^2|E_G|(|E_K||V_K|+2|\Sigma|+|\Sigma||E_K|)$ edges. If we assume that H is sparse (which is highly likely), then running Dijkstra's algorithm on each node of a graph with node set N and edge set A can be done in time $O(|N||A| + |N|^2 \log |N|)$. So, for graph H, the combined time complexity is $O(|R|^3|V_G||V_K||E_G|(|V_K||E_K|+|\Sigma|+|\Sigma||E_K|)+|R|^2|V_G|^2|V_K|^2 \log(|R||V_G||V_K|))$ which simplifies to $O(|R|^3|V_G||V_K||E_G||E_K|(|V_K| + |\Sigma|) + |R|^2|V_G|^2|V_K|^2(log|R||V_G|$

$|V_K|))$.

□

As a corollary, it is easy to see that the data complexity is $O(|V_G||V_K||E_G||E_K|(|V_K| + |\Sigma|) + |V_G|^2|V_K|^2(\log |V_G||V_K|))$ and the query complexity is $O(|R|^3)$. The space complexity is dominated by the space requirements of H given in the proof above.

The above query evaluation can also be accomplished "on-demand" by incrementally constructing the edges of H as required, thus avoiding precomputation and materialisation of the entire graph H. The incremental evaluation process is the same as described in Chapter 3 for the cases of approximation and relaxation, considered separately.

It is easy to show that if the ontology K is empty and there are no `type` edges in the data graph G, then FLEX semantics reduce to approximate matching of CRPQs, as in Section 3.4.1. Similarly, if only ontology-based relaxation is permitted, and the queries over G are limited to be simple conjunctive queries, then this reduces to the query processing semantics with ontology relaxation presented in [66].

8.2 Multi-conjunct FLEX queries and comparison with APPROX/RELAX

A general FLEX query Q is of the form

$$(Z_1, \ldots, Z_m) \leftarrow (X_1, R_1, Y_1), \ldots, (X_n, R_n, Y_n)$$

where any conjuncts may in addition have the FLEX operator applied to them. Let θ be a (Q, G)-matching. If conjuncts $i_1, \ldots, i_j$, $j \leq n$, have FLEX applied to them, then the *distance* from θ to Q, $dist(\theta, Q)$, is defined as

$$dist(\theta, (X_{i_1}, R_{i_1}, Y_{i_1})) + \cdots + dist(\theta, (X_{i_j}, R_{i_j}, Y_{i_j}))$$

The definitions of *minimum-distance* matching, *answer* of Q on G, and *top-k answer* of Q on G are as in Section 3.6 for multi-conjunct APPROX/RELAX queries.

The evaluation of a multi-conjunct query FLEX Q can be undertaken incrementally in the same way as described in Section 3.6 for multi-conjunct APPROX/RELAX queries, joining the answers arising from the incremental evaluation of each of its conjuncts using a rank-join algorithm.

Considering the complexity of FLEX queries compared to APPROX/RELAX queries, Proposition 4.1, Proposition 4.4 and Proposition 8.1 state the relative complexities of evaluating APPROX, RELAX and FLEX single-conjunct RPQs, from which it can be observed that FLEX has higher complexity than both APPROX and RELAX. This is to be expected as the automaton A_Q^K used to evaluate a FLEX single-conjunct RPQ contains all the states and transitions that would appear in the approximate automaton derived directly from M_R for evaluating APPROX (limited to labels in $\Sigma \cup \Sigma^-$) as well as all the states and transitions in the relaxed automaton derived from M_R, for evaluating RELAX.

Considering the expressiveness of FLEX queries compared to APPROX / RELAX queries, it is easy to see that given any graph G, ontology K and CRPQ Q over G that has APPROX or RELAX applied to any of its conjuncts (with APPROX limited to labels in $\Sigma \cup \Sigma^-$) then any answer that is returned at distance k would also be returned, at the same or a lower distance, by a query that has the same conjuncts as Q but with FLEX in place of any occurrence of APPROX or RELAX. Again, this is because any automaton A_Q^K contains all the states and transitions that would appear in the approximate automaton derived from M_R for evaluating APPROX (limited to labels in $\Sigma \cup \Sigma^-$) as well as all the states and transitions that appear in the relaxed automaton derived from M_R. An answer that is returned using FLEX semantics may be at a lower distance than the same answer returned using APPROX/RELAX semantics if an APPROX were replaced by a FLEX in the query and if c_{r2} were less than c_s, because in this case a property relaxation (if applicable) would be less costly than substitution.

Conversely, there exist CRPQs that return answers using FLEX semantics which cannot be returned by any query under APPROX/RELAX semantics, as noted in Example 8.1.

As a final remark, we would argue that the availability of FLEX does not render APPROX and RELAX redundant. Firstly, FLEX does not apply edit operations to labels in $\{\texttt{type}, \texttt{type}^-\}$, whereas APPROX does. Secondly, the user may only

want to consider in some given setting the application of (syntactic) edits — and hence use APPROX, or the application of (semantic) relaxations — and hence use RELAX.

8.3 Concluding remarks

In this chapter, we described how APPROX and RELAX operators can be merged into a single, integrated FLEX operator, showing how answers may be returned therefrom which cannot be returned by the application of APPROX or RELAX semantics alone. We described also how single-conjunct queries with FLEX operators can be evaluated, and discussed the FLEX operator in the context of multi-conjunct CRPQs. Throughout, we provided the proofs for the correctness and complexity of the constructs and algorithms.

We conclude this thesis in the next chapter.

CHAPTER 9

Conclusions and future work

In this thesis we considered approximation and relaxation of conjunctive regular path queries (CRPQs), showing how these may be used and combined to support users in flexible querying of complex graph-structured data. Users can specify approximations and relaxations using the APPROX and RELAX operators, respectively, to be applied to selected conjuncts of their original query, and are able to configure the relative costs of these. Furthermore, using the FLEX operator, users are able to specify that both approximation and relaxation should be applied simultaneously to a query conjunct. For all three operators — APPROX, RELAX and FLEX — query answers are returned incrementally, in polynomial time, ranked in order of increasing distance from the user's original query.

We briefly summarise the thesis in Section 9.1, and then follow this by a discussion of the contributions of this work in Section 9.2. We conclude with a discussion of possible areas of future work in this area in Section 9.3.

9.1 Thesis summary

In Chapter 2, we reviewed related work on graph data models and query languages in general, keyword-based querying, query relaxation and approximation and subgraph

matching.

We provided the background underpinning our research in Chapter 3, in which we introduced the graph data model and the query language (CRPQs). We provided a formal definition of CRPQs and exact matching of single-conjunct RPQs, and gave formal definitions of approximate matching and relaxation of such queries. We concluded with a discussion of the evaluation of multi-conjunct RPQs, each of whose conjuncts may be approximated or relaxed, and the complexity of query answering.

In Chapter 4, we provided formal correctness proofs for the constructs and algorithms introduced in Chapter 3. We established that both the approximate and relaxed answer to a single-conjunct RPQ can be computed in time that is polynomial in the size of the query, the data graph and the ontology graph (in the case of a relaxed query), with answers being returned in ranked order of their 'distance' from the original query.

We presented the *ApproxRelax* prototype system in Chapter 5, restricting our focus to the user-facing features of the system. We subsequently presented a qualitative case study showing how *ApproxRelax* overcame problems in a previous system in the same application domain of lifelong learning.

In Chapter 6, we described the implementation details of the *Omega* system, our final implementation of query approximation and query relaxation, superseding the *ApproxRelax* system. We discussed the system architecture, data structures, data model, API and query evaluation algorithms, and concluded with a review of related implementations in the area of regular path queries.

We presented a performance study conducted with two contrasting real-world data graphs in Chapter 7, establishing that our baseline performance of exact queries is comparable to that of state-of-the-art systems. We additionally showed that, in most cases, our APPROX and RELAX queries exhibit reasonable performance, and discussed causal factors for the poorly-performing queries. We subsequently illustrated by means of an empirical evaluation how the availability of simple statistics relating to paths in the data graph can be used to improve the run-times of a selection of poorly-performing approximated queries.

We concluded the research undertaken in this thesis with Chapter 8, in which we discussed the application of both approximation and relaxation to an individual query conjunct using a single query operator, FLEX, which allows additional answers

to be returned that cannot be obtained by applying only approximation or relaxation to a query conjunct. We described how single-conjunct RPQs that have FLEX applied to them can be evaluated, providing formal proofs of correctness.

9.2 Contributions

This thesis makes the following contributions:

- Extending the work in [115], each edit and relaxation operation is now able to have a different associated cost by modifying the construction of the data structures required for the evaluation of the queries. We also provided a uniform framework for handling both query approximation and query relaxation for CRPQs.

- We specified detailed algorithms for the initialisation and incremental evaluation of approximated and relaxed query conjuncts.

- We provided full proofs of correctness for the constructs and algorithms necessary to evaluate approximated and relaxed CRPQs. We showed formally how both the approximate answer and relaxed answer to a single-conjunct RPQ can be computed in time that is polynomial in the size of the query, the data graph and the ontology graph (in the case of relaxation), with answers being returned in ranked order of their 'distance' from the original query.

- We presented a prototype system called *ApproxRelax* and, focusing on its user-facing features, we presented a qualitative case study in the domain of lifelong learning. We showed how *ApproxRelax* improves on an earlier system by supporting query approximation and query relaxation that provide greater flexibility when querying heterogeneous timeline data, resulting in answers of greater relevance being returned to the user. *ApproxRelax* implements, for the first time, ontology-based relaxation of regular path queries, as well as combined support for approximation and relaxation of CRPQs.

- We provided a description of the implementation details of the *Omega* system, our final implementation of approximation and relaxation of CRPQs. We

discussed the system architecture and low-level data structures, describing how data graphs are created in *Omega*, and we presented the query evaluation algorithms, along with physical optimisations, both in terms of the interaction with the graph store, Sparksee, and our query processing layer within *Omega*.

- We presented a comprehensive performance study of the *Omega* system conducted with two large contrasting real-world data graphs, establishing that our baseline performance of exact queries is comparable to that of state-of-the-art system implementations. The benefits of our APPROX and RELAX operators have been shown in terms of additional answers being returned for queries returning few or no answers for the exact version. Many of the APPROX and RELAX queries executed quickly, but some either failed to terminate or did not complete within a reasonable amount of time. We discussed the reasons for this in each case.

- Building on this performance study, we established that simple graph statistics may be used for optimising the evaluation of approximated queries.

- We introduced and motivated an additional operator combining both query approximation and query relaxation into a single operator, FLEX. This allows easier querying of complex heterogeneous datasets for users as they do not have to be aware of the ontology structure and do not have to identify explicitly which parts of their overall query may be amenable to relaxation. It may also allow more query results to be returned, as there exist CRPQs that return answers using FLEX semantics which cannot be returned by any CRPQ using APPROX/RELAX semantics. We showed how the evaluation of CRPQs that have FLEX applied to them can be undertaken using a combination of the techniques used for approximation and relaxation, and provided theoretical proofs of correctness and complexity of the constructs used.

9.3 Directions for future work

There are several directions for future work:

- It would be interesting to investigate the use of disk-bound data structures

to augment our current in-memory implementation approach, with the aim of improving query performance and guaranteeing the termination of APPROX queries with large intermediate results.

- An extended investigation of our preliminary work using graph statistics as described in Chapter 7 is another area of future work. More extensive empirical testing of the optimisation techniques presented in that chapter is required, along with a greater corpus of queries and data.

- Other promising directions are query rewriting [16, 42], taking advantage of rare labels as described in [82], and further research into optimisation techniques for query evaluation, drawing for example from recent work in [18, 41, 73, 92, 133, 137].

- The FLEX operator is not yet supported by the *Omega* system, and extending the implementation and the performance studies to include this operator is an area of further work.

- Deeper investigation of the relationships between FLEX, APPROX and RELAX is required, for example to determine if there are characteristics of a multi-conjunct CRPQ query, data graph or ontology that mean that FLEX will always return more results than any combination of APPROX or RELAX; and to investigate further integration of the automaton-based query evaluation approaches for APPROX, RELAX and FLEX that we have presented here, into a higher-level abstract framework of which APPROX, RELAX and FLEX are specific instances.

- The area of distributed graph data processing is increasing in importance, with the aim of handling larger volumes of complex, irregular, graph-structured data than can be handled on a single server and to achieve horizontal scalability, for example, in systems such as Giraph[1], Pegasus [71], Pregel [94], GPS [120] Horton [121] and Trinity.RDF [142]. Using distributed graph query processing techniques to enable flexible querying of larger volumes of complex, irregular graph-structured data is an important area of future work.

[1] `https://github.com/apache/giraph`

- The implementation and optimisation aspects of our flexible querying framework may be applicable to other frameworks or future research, in the sense that our results and methods may turn out to be of broader relevance. Examples are computational frameworks in which automata play an integral part, and frameworks in which a 'product graph' (i.e. the product of two different graph-based structures) is constructed and traversed.

- Finally, further work is required into developing a comprehensive system providing users with the flexible query processing capabilities proposed in this thesis. For example, a graphical front-end could allow users to pose queries using forms, keywords, or even natural language, which the system would then translate into CRPQs. The user would select which approximation and relaxation operations, from the full range of operations supported by our framework, should be applied by the system to parts of their queries in order to approximate or to relax them. The user could select which parts of the ontology should be used for the relaxation operations (rather than the whole ontology) and which labels for the edit operations (rather than the full set of edge labels in the data graph). For the substitution operation, it would be straightforward to extend our framework to support finer-grained ranking of the substitution of one property label by another, e.g. through the application of lexical or semantic similarity measures on property labels (rather than assuming the same cost for all substitutions). Finally, to help users interpret answers to their queries, the system could provide a trace of the successive edits/relaxations applied by the system to the original query, and the answers arising from each modified query. Showing the sequence of changes by which the original query was approximated or relaxed could help the user decide whether the answers being returned are relevant to them.

Bibliography

[1] S. Abiteboul, D. Quass, J. Mchugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1:68–88, 1997.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.*, 7(2):57–73, 2009.

[4] B. Amann and M. Scholl. Gram: A graph data model and query languages. In *Proc. ACM Conference on Hypertext*, pages 201–211, 1992.

[5] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible structure and full-text querying for XML. In *SIGMOD Conference*, pages 83–94, 2004.

[6] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. V. d. Bussche. Concepts for graph-oriented object manipulation. In *Proc. 3rd Int. Conf. on Extending Database Technology: Advances in Database Technology*, pages 21–38, 1992.

[7] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[8] K. Anyanwu, A. Maduko, and A. P. Sheth. SPARQ2L: Towards support for subgraph extraction queries in RDF databases. In *Proc. 16th Int. Conf. on World Wide Web*, pages 797–806, 2007.

[9] M. Arenas and J. Pérez. Federation and Navigation in SPARQL 1.1. In *Reasoning Web*, volume 7487, pages 78–111, 2012.

[10] T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proc. 3rd Int. Semantic Web User Interaction Workshop*, 2006.

[11] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - A Crystallization Point for the Web of Data. *Web Semantics*, 7(3):154–165, 2009.

[12] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Recommendation, World Wide Web Consortium (W3C), 1998. See `http://www.w3.org/TR/1998/REC-xml-19980210`.

[13] P. Buche, J. Dibie-Barthélemy, and H. Chebil. Flexible SPARQL querying of web data tables driven by an ontology. In *Proc. 8th Int. Conf. on Flexible Query Answering Systems*, pages 345–357, 2009.

[14] P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, Mar. 2000.

[15] G. Buratti and D. Montesi. Ranking for approximated XQuery Full-Text queries. In *Proc. 25th British National Conf. on Databases*, pages 165–176, 2008.

[16] A. Calì, R. Frosini, A. Poulovassilis, and P. T. Wood. Flexible querying for SPARQL. In *On the Move to Meaningful Internet Systems*, pages 473–490, 2014.

[17] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Proc. 7th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 176–185, 2000.

[18] R. Castillo, C. Rothe, and U. Leser. RDFMatView: Indexing RDF data using materialized SPARQL queries. In *Proc. 6th Int. Workshop on Scalable Semantic Web Knowledge Base Systems*, 2010.

[19] J. P. Cedeño and K. S. Candan. R2DF framework for ranked path queries over weighted RDF graphs. In *Proc. of the Int. Conf. on Web Intelligence, Mining and Semantics*, pages 40:1–40:12, 2011.

[20] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: a semantic search engine for XML. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 45–56, 2003.

[21] M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in GraphLog. In *Proc. 2nd Annual ACM Conference on Hypertext*, pages 269–292, 1989.

[22] M. P. Consens and A. O. Mendelzon. Low-complexity Aggregation in GraphLog and Datalog. *Theoretical Computer Science*, 116(1):95–116, 1993.

[23] I. Cruz, A. Mendelzon, and P. Wood. G+: Recursive Queries without Recursion. In *Proc. 2nd Expert Database Systems Conference*, pages 645–666, 1989.

[24] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *Proc. ACM SIGMOD*, pages 323–330, 1987.

[25] L. Cych. 'Social Networks' in Emerging Technologies for Learning, Coventry. *Becta*, pages 32–40, 2006.

[26] S. de Freitas, I. Harrison, G. Magoulas, A. Mee, F. Mohamad, M. Oliver, G. Papamarkos, and A. Poulovassilis. The development of a system for supporting the lifelong learner. *British Journal of Educational Technology*, 37(6):867–880, 2006.

[27] S. de Freitas, I. Harrison, G. Magoulas, G. Papamarkos, A. Poulovassilis, N. van Labeke, A. Mee, and M. Oliver. L4All: a web-service based system for lifelong learners. In *The Learning Grid Handbook: Concepts, Technologies and Applications, Volume 2: The Future of Learning*. IOS Press, 2008.

[28] S. Dey, V. Cuevas-Vicenttín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher. On Implementing Provenance-aware Regular Path Queries with Relational Query Engines. In *Proc. 16th Int. Conf. on Extending Database Technology*, pages 214–223, 2013.

[29] P. Dolog, H. Stuckenschmidt, and H. Wache. Robust query processing for personalized information access on the semantic web. In *Proc. 7th Int. Conf. on Flexible Query Answering Systems*, pages 343–355, 2006.

[30] P. Dolog, H. Stuckenschmidt, H. Wache, and J. Diederich. Relaxing RDF queries based on user and domain preferences. *J. Intell. Inf. Syst.*, 33(3):239–260, 2009.

[31] X. Dong and A. Halevy. Indexing dataspaces. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 43–54, New York, NY, USA, 2007.

[32] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[33] S. Elbassuoni, M. Ramanath, R. Schenkel, M. Sydow, and G. Weikum. Language-model-based ranking for queries on RDF-graphs. In *Proc. 18th ACM Conf. on Information and Knowledge Management*, pages 977–986, 2009.

[34] S. Elbassuoni, M. Ramanath, R. Schenkel, and G. Weikum. Searching RDF graphs with SPARQL and keywords. *IEEE Data Eng. Bull.*, 33(1):16–24, 2010.

[35] S. Elbassuoni, M. Ramanath, and G. Weikum. Query relaxation for entity-relationship search. In *Proc. 8th Extended Semantic Web Conference on The Semanic Web*, pages 62–76, 2011.

[36] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *Proc. 27th International Conference on Data Engineering*, pages 39–50, 2011.

[37] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *Proc. VLDB Endow.*, 6(13):1510–1521, Aug. 2013.

[38] C. Fellbaum, editor. *WordNet: an electronic lexical database.* MIT Press, 1998.

[39] M. Fernández, D. Florescu, A. Levy, and D. Suciu. Declarative specification of web sites with STRUDEL. *The VLDB Journal*, 9(1):38–55, Mar. 2000.

[40] J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. In *Proc. ACM SIGMOD*, pages 415–428, 2009.

[41] G. H. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *19th Int. Conf. on Extending Database Technology*, pages 636–639, 2016.

[42] R. Frosini, A. Calì, A. Poulovassilis, and P. T. Wood. Flexible query processing for SPARQL. *Semantic Web Journal, In press*, 2016.

[43] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. 23rd Int. Conf. on Very Large Data Bases*, pages 436–445, 1997.

[44] R. Goldman and J. Widom. Approximate DataGuides. In *Proc. of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, pages 436–445, 1999.

[45] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 43(3):431–498, May 2001.

[46] G. Grahne and A. Thomo. Approximate reasoning in semi-structured databases. In *Proc. 8th Int. Workshop on Knowledge Representation meets Databases*, 2001.

[47] G. Grahne and A. Thomo. Regular path queries under approximate semantics. *Ann. Math. Artif. Intell.*, 46(1-2):165–190, 2006.

[48] G. Grahne, A. Thomo, and W. W. Wadge. Preferentially annotated regular path queries. In *Proc. 11th Int. Conf. on Database Theory*, pages 314–328, 2007.

[49] M. Graves. A graph-theoretic data model for genome mapping databases, 1995.

[50] M. Graves, E. Bergeman, and C. Lawrence. Querying a genome database using graphs. In *Proc. 3rd Int. Conf. on Bioinformatics and Genome Research*, 1994.

[51] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling Kleene: Fast property paths in RDF-3X. In *Proc. 1st Int. Workshop on Graph Data Management Experiences and Systems*, pages 14:1–14:7, 2013.

[52] C. Gutierrez, C. Hurtado, A. O. Mendelzon, and J. Perez. Foundations of semantic web databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011.

[53] R. H. Güting. GraphDB: Modeling and querying graphs in databases. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 297–308, 1994.

[54] M. Gyssens, J. Paredaens, J. V. den Bussche, and D. van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994.

[55] O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL queries over the web of linked data. In *Proc. 8th International Semantic Web Conference*, volume 5823, pages 293–309, Chantilly, Virginia, 2009.

[56] P. Hayes and P. F. Patel-Schneider, editors. *RDF 1.1 Semantics*, W3C Recommendation, February 2014.

[57] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. ACM SIGMOD Int. Conf. on Management of data*, pages 405–418, 2008.

[58] T. Heath, M. Hausenblas, C. Bizer, and R. Cyganiak. How to publish linked data on the web (tutorial). In *Proc. 7th Int. Semantic Web Conference*, 2008.

[59] J. Hidders. *A Graph-based Update Language for Object-Oriented Data Models*. PhD thesis, Eindhoven University of Technology, Eindhoven, the Netherlands, 2001. PhD-thesis.

[60] J. Hidders. Typing graph-manipulation operations. In *Proc. 9th Int. Conf. on Database Theory*, pages 394–409, 2002.

[61] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. Research Report MPI-I-2010-5-007, Max-Planck-Institut für Informatik, 2010.

[62] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artificial Intelligence*, 194:28–61, 2013.

[63] A. Hogan, M. Mellotte, G. Powell, and D. Stampouli. Towards fuzzy query-relaxation for RDF. In *Proc. 9th Int. Conf. on The Semantic Web*, pages 687–702, 2012.

[64] H. Huang and C. Liu. Query relaxation for star queries on RDF. In *Proc. 11th Int. Conf. on Web Information Systems Engineering*, pages 376–389, 2010.

[65] H. Huang, C. Liu, and X. Zhou. Computing relaxed answers on RDF databases. In *Proc. 9th Int. Conf. on Web Information Systems Engineering*, pages 163–175, 2008.

[66] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Query relaxation in RDF. *Journal on Data Semantics*, X:31–61, 2008.

[67] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Ranking approximate answers to semantic web queries. In *Proc. 6th European Semantic Web Conference*, pages 263–277, 2009.

[68] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, 2004.

[69] B. Iordanov. HyperGraphDB: A generalized graph database. In *Proc. 11th Int. Conf. on Web-age Information Management*, volume 6185, pages 25–36, 2010.

[70] R. Jin, N. Ruan, S. Dey, and J. Y. Xu. SCARAB: scaling reachability computation on large graphs. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 169–180, 2012.

[71] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *Proc. 9th Int. Conf. on Data Mining*, pages 229–238, 2009.

[72] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proc. 20th ACM Symposium on Principles of Database Systems*, pages 40–51, 2001.

[73] Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. SPARQL query optimization on top of DHTs. In *Proc. 9th Int. Semantic Web Conference*, pages 418–435, 2010.

[74] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. STAR: Steiner-Tree approximation in relationship graphs. In *Proc. 25th Int. Conf. on Data Engineering*, pages 868–879, 2009.

[75] G. Kasneci, M. Ramanath, F. Suchanek, and G. Weikum. The YAGO-NAGA approach to knowledge discovery. *SIGMOD Rec.*, 37(4):41–47, Mar. 2009.

[76] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and ranking knowledge. In *Proc. 24th Int. Conf. on Data Engineering*, pages 953–962, 2008.

[77] C. Kiefer, A. Bernstein, and M. Stocker. The fundamentals of iSPARQL: A virtual triple approach for similarity-based semantic web tasks. In *Proc. 6th Int. Semantic Web Conference*, pages 295–309, 2007.

[78] G. Klyne and J. J. Carroll. Resource description framework (RDF): Concepts and abstract syntax, 2004.

[79] K. Kochut and M. Janik. SPARQLeR: Extended SPARQL for semantic association discovery. In *Proc. 4th European Semantic Web Conference*, pages 145–159, 2007.

[80] R. Koper, B. Giesbers, P. van Rosmalen, P. B. Sloep, J. van Bruggen, C. Tattersall, H. Vogten, and F. Brouns. A design model for lifelong learning networks. *Interactive Learning Environments*, (1-2):71–92, 2005.

[81] R. Koper and C. Tattersall. New directions for lifelong learning using network technologies. *British Journal of Educational Technology*, 35(6):689 –700, 2004.

[82] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Proc. of the 24th Int. Conf. on Scientific and Statistical Database Management*, pages 177–194, 2012.

[83] A. Langegger, W. Wöß, and M. Blöchl. A semantic web middleware for virtual data integration on the web. In *Proc. 5th European Semantic Web Conference*, 2008.

[84] J. Larriba-Pey, N. Martínez-Bazan, and D. Domínguez-Sal. Introduction to Graph Databases. In *Proc. 10th International Summer School: Reasoning on the Web in the Big Data Era*, pages 171–194, 2014.

[85] U. Leser. A query language for biological networks. *Bioinformatics*, 21(2):33–39, Jan. 2005.

[86] U. Leser and S. Trißl. Graph Management in the Life Sciences. In *Encyclopedia of Database Systems*, pages 1266–1271. 2009.

[87] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 695–704, New York, NY, USA, 2008. ACM.

[88] M. Levene and G. Loizou. A graph-based data model and its ramifications. *IEEE Trans. on Knowl. and Data Eng.*, 7(5):809–823, Oct. 1995.

[89] M. Levene and A. Poulovassilis. The Hypernode Model and Its Associated Query Language. In *Proc. 5th Jerusalem Conference on Information Technology*, pages 520–530, 1990.

[90] M. Levene and A. Poulovassilis. An object-oriented data model formalised through hypergraphs. *Data Knowl. Eng.*, 6(3):205–224, May 1991.

[91] C. Liu, J. Li, J. X. Yu, and R. Zhou. Adaptive relaxation for querying heterogeneous XML data sources. *Information Systems*, 35(6):688–707, 2010.

[92] A. Loizou and P. T. Groth. On the formulation of performant SPARQL queries. *CoRR*, abs/1304.0567, 2013.

[93] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.*, 39(1):4:1–4:46, Jan. 2014.

[94] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. 2010 Int. Conf. on Management of data ACM SIGMOD*, pages 135–146, 2010.

[95] F. Mandreoli, R. Martoglia, G. Villani, and W. Penzo. Flexible query answering on graph-modeled data. In *12th Int. Conf. on Extending Database Technology*, pages 216–227, 2009.

[96] N. Martínez-Bazan, M. A. Águila Lorente, V. Muntés-Mulero, D. Dominguez-Sal, S. Gómez-Villamor, and J.-L. Larriba-Pey. Efficient graph management based on bitmap indices. In *Proc. 16th Int. Database Engineering*, pages 110–119, 2012.

[97] N. Martínez-Bazan and D. Dominguez-Sal. Using semijoin programs to solve traversal queries in graph databases. In *Proc. Workshop on GRAph Data Management Experiences and Systems*, pages 6:1–6:6, 2014.

[98] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey. DEX: High-Performance Exploration on Large Graphs for Information Retrieval. In *Proc. 16th ACM Conference on Information and Knowledge Management*, pages 573–582, 2007.

[99] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. In *Proc. of the 15th Int. Conf. on Very Large Data Bases*, pages 185–193, 1989.

[100] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[101] X. Meng, Z. M. Ma, and L. Yan. Providing flexible queries over web databases. In *Proc. 12th Int. Conf. on Knowledge-Based and Intelligent Information & Engineering Systems*, pages 601–606, 2008.

[102] M. Mochol, A. Jentzsch, and H. Wache. Suitable employees wanted? Find them with semantic techniques. Paper presented at the Workshop on Making Semantics Work For Business, 2007.

[103] S. Pandit, D. H. Chau, S. Wang, and C. Faloutsos. Netprobe: A fast and scalable system for fraud detection in online auction networks. In *Proc. 16th Int. Conf. on World Wide Web*, pages 201–210, 2007.

[104] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proc. 11th Int. Conf. on Data Engineering*, pages 251–260, 1995.

[105] J. Paredaens, P. Peelman, and L. Tanca. G-log: A graph-based query language. *IEEE Trans. on Knowl. and Data Eng.*, 7(3):436–453, June 1995.

[106] P. Pareja-Tobes, R. Tobes, M. Manrique, E. Pareja, and E. Pareja-Tobes. Bio4j: a high-performance cloud-enabled graph-based data platform. 2015.

[107] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. In *Proc. 7th Int. Semantic Web Conference*, pages 66–81, 2008.

[108] J. Peters. Regular path query evaluation using path indexes. In *Master's thesis, Eindhoven University of Technology*, 2015.

[109] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *Proc. 9th Int. Conf. on The Semantic Web*, pages 406–421, 2012.

[110] A. Poulovassilis. Database research challenges and opportunities of big graph data. In *Proceedings of the 29th British National Conference on Big Data*, pages 29–32, 2013.

[111] A. Poulovassilis and S. G. Hild. Hyperlog: A graph-based system for database browsing, querying, and update. *IEEE Trans. Knowl. Data Eng.*, 13(2):316–333, 2001.

[112] A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Trans. Inf. Syst.*, 12(1):35–68, Jan. 1994.

[113] A. Poulovassilis, P. Selmer, and P. T. Wood. Flexible Querying of Lifelong Learner Metadata. *IEEE Transactions on Learning Technologies*, 5(2):117–129, 2012.

[114] A. Poulovassilis, P. Selmer, and P. T. Wood. Approximation and Relaxation of Semantic Web Path Queries. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 40:1–21, 2016.

[115] A. Poulovassilis and P. T. Wood. Combining approximation and relaxation in semantic web path queries. In *Proc. 9th Int. Semantic Web Conf.*, pages 631–646, 2010.

[116] J. Pound, I. F. Ilyas, and G. E. Weddell. QUICK: Expressive and flexible search over knowledge bases and text collections. *PVLDB*, 3(2):1573–1576, 2010.

[117] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* 3rd edition, 2003.

[118] B. R. K. Reddy and P. S. Kumar. Efficient approximate SPARQL querying of web of linked data. In *Proc. 6th Int. Workshop on Uncertainty Reasoning for the Semantic Web*, volume 654, pages 37–48, 2010.

[119] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases.* O'Reilly Media, Inc., 2013.

[120] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *Scientific and Statistical Database Management*. Stanford InfoLab, July 2013.

[121] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. In *Proc. 28th Int. Conf. on Data Engineering*, pages 1289–1292, 2012.

[122] A. Seaborne and S. Harris(Editors). SPARQL 1.1 Query Language, W3C Recommendation 21 march 2013.

[123] A. Singhal. 'Introducing the Knowledge Graph: things, not strings', Official Google Blog, 2012. Available at `https://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html`.

[124] J. M. Sumrall. Path indexing for efficient path query processing in graph databases. In *Master's thesis, Eindhoven University of Technology*, 2015.

[125] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for TopX search. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 625–636, 2005.

[126] Y. Tian and J. M. Patel. TALE: A tool for approximate large graph matching. In *Proc. 24th Int. Conf. on Data Engineering*, pages 963–972, 2008.

[127] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. 1990.

[128] N. van Labeke, G. D. Magoulas, and A. Poulovassilis. Searching for "People like me" in a Lifelong Learning System. In *Proc. 4th European Conf. on Technology Enhanced Learning*, pages 106–111, 2009.

[129] N. van Labeke, G. D. Magoulas, and A. Poulovassilis. Personalised search over lifelong learner's timelines using string similarity measures. Technical Report BBKCS-11-01, Birkbeck, 2011. Available at `http://www.dcs.bbk.ac.uk/research/techreps/2011/bbkcs-11-01.pdf`.

[130] N. van Labeke, A. Poulovassilis, and G. D. Magoulas. Using similarity metrics for matching lifelong learners. In *Proc. 9th Int. Conf. on Intelligent Tutoring Systems*, pages 142–151, 2008.

[131] R. Varadarajan, V. Hristidis, L. Raschid, M. Vidal, L. D. Ibáñez, and H. Rodríguez-Drumond. Flexible and efficient querying and ranking on hyperlinked data sources. In *Proc. 12th Int. Conf. on Extending Database Technology*, pages 553–564, 2009.

[132] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.

[133] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin. An efficient graph indexing method. In *Proc. 28th Int. Conf. on Data Engineering*, pages 210–221, 2012.

[134] G. Weikum, G. Kasneci, M. Ramanath, and F. M. Suchanek. Database and information-retrieval methods for knowledge discovery. *Commun. ACM*, 52(4), 2009.

[135] P. T. Wood. Query languages for graph databases. *SIGMOD Rec.*, 41(1):50–60, Apr. 2012.

[136] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *Proc. SIGMOD Conference*, pages 537–538, 2005.

[137] N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In *Proc. 2016 Int. Conf. on Management of Data, ACM SIGMOD*, pages 1875–1889, 2016.

[138] S. Yang, Y. Wu, H. Sun, and X. Yan. Schemaless and structureless graph querying. *Proc. VLDB Endow.*, 7(7):565–576, Mar. 2014.

[139] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: scalable reachability index for large graphs. *Proc. VLDB Endow.*, 3(1-2):276–284, Sept. 2010.

[140] C. Yu and H. V. Jagadish. Querying complex structured databases. In *Proc. 33rd Int. Conf. on Very Large Data Bases*, pages 1010–1021, 2007.

[141] H. Zauner, B. Linse, T. Furche, and F. Bry. A RPL through RDF: expressive navigation in RDF graphs. In *Proc. 4th Int. Conf. on Web Reasoning and Rule Systems*, pages 251–257, 2010.

[142] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *Proc. VLDB Endow.*, 6(4):265–276, 2013.

[143] S. Zhang, J. Yang, and W. Jin. SAPPER: Subgraph indexing and approximate matching in large graphs. *Proc. of the VLDB Endowment*, 3(1):1185–1194, 2010.

[144] X. Zhou, J. Gaugaz, W.-T. Balke, and W. Nejdl. Query relaxation using malleable schemas. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 545–556, 2007.

[145] D. D. Zhu and K. I. Ko. *Problem Solving in Automata, Languages, and Complexity*. 2004.

[146] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. In *Proc. Int. Conf. on Very Large Data Bases*, pages 482–493, 2011.