Entity Comparison in Knowledge Graphs

Alina Petrova

Exeter College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Michaelmas 2019

To Grandpa, and

to Zhuzhuka

Aknowledgements

Grad school is a journey, and I would like to thank everybody who was part of my DPhil journey, who helped and supported me, who showed me kindness and shared laughs, tears and drinks. So, in the order of appearance, I would like to thank:

· Prof. Evgeny Kharlamov, for encouraging me to apply to the University of Oxford;

· my friends in Dresden and St. Petersburg who supported me during some difficult times before grad school and helped me to actually make it to Oxford;

· Prof. Ian Horrocks and Prof. Bernardo Cuenca Grau, for believing in me, for giving me the once-in-a-lifetime opportunity to work in their group, for teaching me and sharing ideas, and for their constant academic support;

· the Department of Computer Science and DeepMind, for generously funding my studies at Oxford; without their scholarship my DPhil would have never been possible;

· Nadya, for being an amazing friend and for discovering Oxford together;

· Pan and Alessandro, for being the terrific 311 squad;

· the fabulous KRR group, Yavor, Andrew, Babis, Robert, David, Giorgio, Federico, Temitope, Mark, Ana, Przemek and Stefano, for being an absolute joy in and out of the office;

· Egor, for stepping in, for always caring and for being a most amazing supervisor one can only wish for;

· OxWoCS, for creating an environment of like-minded, driven people to which you feel you truly belong;

· Julia, Oana, Bushra, Klaudia, Aysha, Ruslan, and all my wonderful friends from the department, for sharing every step of the way and living through the ups and downs of grad school together;

· Lucy and Ayumi, for being the best housemates I could dream of, and for making Oxford finally feel like home;

· George and Rui, for all the warmth, the drinks and the sarcastic comments, and for being there when it was the darkest;

· the AI Gaming team, Toby, Rachel, Stephen, Paul, Sarah and Aaron, for giving me the opportunity to work with you, it's been a pleasure;

· Lukas and Harley, for our thrilling journey in the land of entrepreneurship, for being my friends and my teachers, for all the late night discussions, trips, hopes and grilling work;

· Ryan Gosling, for being the subject of one particularly effective meme that would tell me every day he wants to watch me write my thesis.

Finally, I would like to thank my family for never ever for a second doubting that I can do it; my best friends Lubasha and Vicha, for your love that I feel even when I am miles away and for always being there for me.

And lastly, I would like to thank Zhenya, for being my soulmate, my rock and my accomplice, and for making my dreams come true. This journey would never be the same without you.

Abstract

Knowledge graphs (KGs) represent information in a simple, interlinked format and are used in a variety of applications. Besides the standard reasoning tasks such as query answering, there has been an increasing need for new types of analysis. One such fundamental analysis task is entity comparison, i.e., determining what two entities have in common and how they are different. The importance of such a task can be seen in various applications such as product comparisons, and explainable recommender systems. This thesis studies the problem of entity comparison over knowledge graphs and presents a novel framework that models comparisons via similarity and difference queries. It is the first declarative comparison framework for linked data that treats both similarities and differences as first-class citizens.

We first define the syntax and semantics of various types of comparison queries, motivating each type with examples and establishing important relations between different query types. Next, we study the complexity of the two fundamental decision problems over queries, namely the existence and verification problems, for basic comparison queries and for most specific similarity queries (MSSQs). We consider two underlying fragments of SPARQL, with and without arithmetic comparisons, and we show how they affect the complexity of the above problems, and which types of data they suit most. We then study practical aspects of computing the MSSQ. In particular, we establish an important uniqueness result for MSSQs and propose an efficient approximation algorithm. We implement an algorithm for computing acyclic similarity queries and evaluate its performance on large-scale KGs; our empirical results demonstrate the practical feasibility of our algorithm.

Publications

Alina Petrova. *Comparing entities in RDF graphs.* In: Proceedings of the VLDB 2017 PhD Workshop co-located with the 43rd International Conference on Very Large Data Bases (VLDB 2017). CEUR-WS.org, 2017.

Alina Petrova, Evgeny Sherkhonov, Bernardo Cuenca Grau and Ian Horrocks. *Entity Comparison in RDF Graphs.* In: Proceedings of the 16th International Semantic Web Conference (ISWC 2017), pages 526–541. Springer, 2017.

Alina Petrova, Egor V. Kostylev, Bernardo Cuenca Grau and Ian Horrocks. *Query-Based Entity Comparison in Knowledge Graphs Revisited.* In: Proceedings of the 18th International Semantic Web Conference (ISWC 2019), pages 558–575. Springer, 2019.

Alina Petrova, Egor V. Kostylev, Bernardo Cuenca Grau and Ian Horrocks. *Towards Explainable Entity Matching via Comparison Queries.* In: Proceedings of the 14th International Workshop on Ontology Matching co-located with the 18th International Semantic Web Conference (ISWC 2019). CEUR-WS.org, 2019.

Contents

List of Figures

List of Tables

List of Abbreviations

KG	knowledge graph
SQ	similarity query
ESQ	exact similarity query
DQ	difference query
MSSQ	most specific similarity query
MGDQ	most general difference query
RA	relational algebra
DL	description logic
CQ	conjunctive query
CQAC	conjunctive query with arithmetic comparisons
AFC	arithmetic filter condition
QRE	query reverse engineering
lcs	least common subsumer
TFG	Twitter follower graph

Chapter 1

Introduction

1.1 Knowledge Graphs

Knowledge graphs (KGs) represent information in a simple, interlinked format and are used in a variety of applications, such as search, question answering, product comparisons, and explainable recommender systems. Knowledge graphs originated from several important ideas [106]. On the one hand, KGs stem from earlier attempts to represent human knowledge graphically in the form of semantic networks, frames and conceptual graphs [67, 81, 99, 110], in which concepts were linked to each other via some semantic relations. On the other hand, KGs draw on the area of Semantic Web. The idea of linked data and Semantic Web belongs to Tim Berners-Lee [18,19] and consists in enriching the World Wide Web with standardized, machine-readable metadata about web pages. Collections of metadata about Web resources, and later about other types of entities became independent data sources, e.g., KGs and knowledge bases like DBpedia [68] and YAGO [114]. While there exists certain ambiguity in the nomenclature, the terms 'knowledge graph', and 'knowledge base' are often used interchangeably, and since many KGs are published in the RDF format [37], these are also called 'RDF graphs'. The term 'knowledge graph' was first coined

in 1980s [59, 86], and it was popularized by Google in 2012 when introducing their new semantic data resource that enabled searching for "things, not strings" [108]. The term has since been widely adopted to define graph-shaped data that describes entities, their properties and relations between various entities [40,65,90]. Nowadays there exist different types of knowledge graphs.

- The most common type is general-domain KGs, e.g., Freebase [22], YAGO [114], DBpedia [68], Microsoft Concept Graph [61], and ConceptNet [83,111]. Such KGs are usually crowd- and open-source, and consist of millions of entries. Proprietary KGs like Google Knowledge Graph [108] and Microsoft's Bing KG [85] are primarily used for search, question answering and recommendations as well as for artificial personal assistants [21].

- There also exist domain-specific KGs, e.g., in the areas of banking, healthcare, commerce and retail [85]. Academic KGs, e.g., Microsoft Academic Knowledge Graph [43] and DBLP [72], are also being created.

- Large companies such as Bloomberg, Amazon, LinkedIn or eBay create in-house KGs in order to store enterprise data and use it for internal operations as well as in client-oriented products [40, 85].

- Finally, there exist comprehensive, web-scale KGs that are automatically populated via web crawling and scraping. They contain data not only about physical entities, but also about articles, events, discussions, social media post, etc. The largest such KG to date is Diffbot Knowledge Graph.[1]

Large-scale knowledge graphs are increasingly being used in applications, and there is a growing need for tools that can effectively support users in analysis and exploration. The canonical reasoning task over RDF data exploited in applications is

[1] `https://www.diffbot.com/knowledge-graph/`

query answering, where SPARQL is the standard query language developed for that purpose [52]. There is, however, an increasing need in many applications for non-standard analysis tasks that do not directly correspond to SPARQL query answering. One such important task is entity comparison — to describe in an informative way the similarities and differences between two given entities as outlined in a knowledge graph. This is in stark contrast to the computation of a similarity measure, where the output is a number indicating how similar the given entities are likely to be rather than a human-readable explanation.

1.2 Comparison over Knowledge Graphs

Entity comparison is used routinely across multiple domains and applications, from online shopping to food and nutrition comparison widgets, to Facebook's 'see what you have in common' pages. Existing tools typically focus on a constrained application domain (e.g., used cars) and provide a side-by-side comparison of the given entities based on a fixed set of relevant attributes (e.g., price, engine size, or colour). We are, however, interested in the generic entity comparison support in knowledge graphs, in which case it is no longer possible to fix a relevant set of attributes or relationships upfront.

Let us consider two example use cases. In the first one, a startup company is developing a toolkit for analysing widely-used biomedical RDF repositories, such as Bio2RDF [16]. The tool being developed should provide a drug comparison functionality; in particular, when given two drugs described in an RDF graph from the repository, such as Ibuprofen and Metamizole, the tool should be able to automatically report that "both drugs are analgesics and can reduce fever; however, Metamizole can also act as a spasm reliever, whereas Ibuprofen has an anti-inflammatory function". The second use case concerns the development of an analysis tool on

top of IMDB data; such tool should allow users to compare arbitrary aspects of movie-making, such as directors, producers, actors and so on. For example, when comparing Quentin Tarantino to Martin Scorsese, the tool should report that they are similar in that they are both male directors who won both an Oscar and a Golden Globe and who have also acted in their own movies; in turn, they are different in that Tarantino won the Palme d'Or at the Cannes Film Festival, while Scorsese won an Emmy award, to which Tarantino was only nominated.

Entity comparison is conventionally seen in the Information Retrieval community as a type of exploratory search [76, 127]. It is an important task which is implemented in a wide range of tools and web portals, in domains as diverse as hotels,[2] cars,[3] universities,[4] online shopping,[5] or social networks.[6] Existing entity comparison tools typically perform a side-by-side comparison of items based on a fixed (often hard-coded) template of features to compare, e.g., price, size, memory capacity and other technical parameters in the case of tablets (see Figure 1). Relying on a fixed set of features is a reasonable solution for tabular, domain specific data whose structure is relatively rigid and stable. It is even appropriate in the context of graph data, provided that a limited set of relevant features can be specified beforehand; for instance, Facebook Friendship pages[7] allow for the comparison of two Facebook users by displaying their shared information based on a limited set of features specific to social networks (e.g., "likes", mutual friends, relationship status). However, the approach is hard to be directly transferred to Linked Data, namely to loosely structured knowledge graphs. Hence, a more flexible approach to entity comparison over knowledge graphs is needed. Up to now, such approaches

[2] http://www.flightnetwork.com/pages/hotel-comparison-tool/

[3] http://www.cars.com/go/compare/modelCompare.jsp

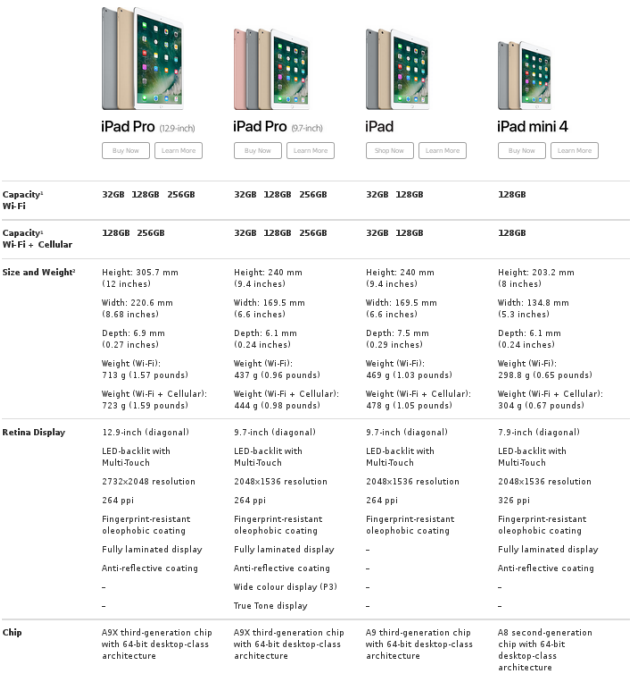
[4] http://colleges.startclass.com/

[5] http://www.intel.co.uk/content/www/uk/en/products/compare-products.html

[6] https://www.facebook.com

[7] https://web.archive.org/web/20101030105622/http://blog.facebook.com/blog.php?post=443390892130 (original announcement)

have mainly been based on the structure of the graph, e.g., finding a path that connects the two entities (see Chapter 3 for a discussion of related work). In this thesis we study the problem of entity comparison in knowledge graphs, and propose a novel declarative framework that treats comparison as a non-standard reasoning task and enables automatic and informative comparison of entities. In the framework, comparison is modelled via SPARQL queries. Given two entities in an RDF graph, we differentiate between *similarity queries* that capture patterns common for both entities in that RDF graph, and *difference queries* that reflect patterns in data that fit only one of the two entities. Additionally, we study the *most specific similarity queries* that correspond to the most comprehensive and informative patterns describing both input entities, and we look at how such queries can be computed.



	iPad Pro (12.9-inch)	iPad Pro (9.7-inch)	iPad	iPad mini 4
	Buy Now / Learn More	Buy Now / Learn More	Shop Now / Learn More	Buy Now / Learn More
Capacity[1] Wi-Fi	32GB 128GB 256GB	32GB 128GB 256GB	32GB 128GB	128GB
Capacity[1] Wi-Fi + Cellular	128GB 256GB	32GB 128GB 256GB	32GB 128GB	128GB
Size and Weight[2]	Height: 305.7 mm (12 inches)	Height: 240 mm (9.4 inches)	Height: 240 mm (9.4 inches)	Height: 203.2 mm (8 inches)
	Width: 220.6 mm (8.68 inches)	Width: 169.5 mm (6.6 inches)	Width: 169.5 mm (6.6 inches)	Width: 134.8 mm (5.3 inches)
	Depth: 6.9 mm (0.27 inches)	Depth: 6.1 mm (0.24 inches)	Depth: 7.5 mm (0.29 inches)	Depth: 6.1 mm (0.24 inches)
	Weight (Wi-Fi): 713 g (1.57 pounds)	Weight (Wi-Fi): 437 g (0.96 pounds)	Weight (Wi-Fi): 469 g (1.03 pounds)	Weight (Wi-Fi): 298.8 g (0.65 pounds)
	Weight (Wi-Fi + Cellular): 723 g (1.59 pounds)	Weight (Wi-Fi + Cellular): 444 g (0.98 pounds)	Weight (Wi-Fi + Cellular): 478 g (1.05 pounds)	Weight (Wi-Fi + Cellular): 304 g (0.67 pounds)
Retina Display	12.9-inch (diagonal)	9.7-inch (diagonal)	9.7-inch (diagonal)	7.9-inch (diagonal)
	LED-backlit with Multi-Touch	LED-backlit with Multi-Touch	LED-backlit with Multi-Touch	LED-backlit with Multi-Touch
	2732×2048 resolution	2048×1536 resolution	2048×1536 resolution	2048×1536 resolution
	264 ppi	264 ppi	264 ppi	326 ppi
	Fingerprint-resistant oleophobic coating	Fingerprint-resistant oleophobic coating	Fingerprint-resistant oleophobic coating	Fingerprint-resistant oleophobic coating
	Fully laminated display	Fully laminated display	–	Fully laminated display
	Anti-reflective coating	Anti-reflective coating	–	Anti-reflective coating
	–	Wide colour display (P3)	–	–
	–	True Tone display	–	–
Chip	A9X third-generation chip with 64-bit desktop-class architecture	A9X third-generation chip with 64-bit desktop-class architecture	A9 third-generation chip with 64-bit desktop-class architecture	A8 second-generation chip with 64-bit desktop-class architecture

Figure 1: A comparison of different iPad models[8]

[8] A screenshot taken from `http://www.argos.co.uk/static/ArgosPromo3/includeName/apple-ipad-comparison.htm` on September 5th, 2017.

1.3 Contributions and Thesis Structure

This thesis presents a novel entity comparison framework for knowledge graphs. Given a knowledge graph and two entities from it, the framework generates various comparison queries that highlight in which respect the two entities are similar and how they are different. It is the first declarative comparison framework for linked data that treats both similarities and differences as first-class citizens. The results of our work have been previously published in [93–96]. The thesis is organised as follows. Part I outlines the theoretical background and a body of previous works used in this thesis.

- Chapter 2 provides necessary preliminaries for the entity comparison framework, covering formalisms and languages the framework is based on, namely the RDF data model, the SPARQL query language, and the language of conjunctive queries (CQs), as well as foundations of computational complexity and graph theory. Additionally, in Section 2.2.5 we formally introduce the query language of the framework, giving its syntax, semantics and key notions.

- Chapter 3 presents a survey of the existing works on explanations and comparisons over knowledge graphs, paying particular attention to relatedness-based explanations, as well as gives a thorough overview of the areas that are related to entity comparison, namely query reverse engineering (QRE) and query subsumption. We recapitulate the classical definition of the QRE decision problem together with its common variations, and cite complexity results of QRE for several fragments of SPARQL and CQs.

Part II introduces the entity comparison framework and establishes key complexity results.

- Chapter 4 presents different types of comparison queries, namely, similarity, exact similarity, most specific similarity, difference, and most general difference

queries, motivating each type of queries with examples and with a practical rationale. We introduce two main decision problems relevant for computing comparison queries: the problem of verifying that a given query is of a particular type, and the problem of checking whether a query of particular type exists for the given data input. Finally, we study these problems for similarity, exact similarity and difference queries, and discuss their complexity. We differentiate between queries with and without the arithmetic comparisons, arguing that each type is more suitable for particular input data, and provide complexity results for both types of queries.

- Chapter 5 focuses on the most specific similarity queries, or MSSQs. We argue the MSSQs represent one type of the most informative explanations for the given entities in an RDF graph, and show that MSSQs hold the uniqueness property for a pair of entities and a dataset. We present a polynomial-time algorithm for computing MSSQs, COMPUTE_MSSQ, which is able to generate queries both with and without the arithmetic comparisons, and study the complexity of existence and verification problems for MSSQs, illustrating how the presence of arithmetic comparisons affects the complexity of the latter.

Part III focuses on the practical aspects of computing MSSQs over real-world data.

- Chapter 6 further improves on the algorithm from Chapter 5, addressing scalability issues of the latter. We propose an approximation algorithm for computing MSSQs, COMPUTE_APPROX_MSSQ, which is guaranteed to compute a similarity query that is not necessarily an MSSQ for the given input, but has a high empirical approximation ratio with respect to the MSSQ. The algorithm uses two novel data structures, pair trees and similarity trees, that are defined and studied in the chapter. Finally, we discuss the notion of acyclicity with regards to queries over knowledge graphs, and we show that the approximation

algorithm always outputs acyclic similarity queries defined in the chapter.

- Chapter 7 discusses the implementation details of the COMPUTE_MSSQ and COMPUTE_APPROX_MSSQ algorithms, and reports on the evaluation experiments that demonstrated the algorithm's scalability over three large-scale, real-world datasets, namely LUBM, YAGO and a subset of the Twitter follower graph. In addition, we compare the two algorithms in an indirect manner, arguing that COMPUTE_APPROX_MSSQ reaches an approximation ratio of over 90% on real-world data, making the algorithm suitable for practical settings.

Finally, Part IV concludes the thesis by outlining future directions of research in the area of declarative entity comparisons over knowledge graphs.

- Chapter 8 discusses the entity comparison framework and recapitulates the contribution made in this thesis, as well as suggests further theoretical extensions of the framework and highlights practical problems of user-oriented entity comparison.

Part I

Preliminaries

Chapter 2

Foundations

In this chapter we outline the theoretical background this thesis is based on, and give the necessary preliminaries on computational complexity, on graph theory and homomorphisms, and on several query languages and formalisms.

2.1 Computational Complexity

In this thesis we study the complexity of decision problems related to entity comparison over RDF graphs. While we assume some basic notions from the theory of computational complexity whose definitions are beyond the scope of this work, e.g., decision problems, (non-)deterministic Turing machines and boolean circuits, we next restate several complexity classes that are relevant for query comparison [11, 47, 88, 109].

Complexity Classes A complexity class is a set of decision problems of a certain resource-bound complexity, i.e., of problems that require a certain amount of resources — time or space. The amount of time or space required is measured with respect to the size of the input. For example, if a decision problem takes as input an object of size n, we say that the problem can be solved in polynomial time, if it

can be solved in the amount of time that is polynomial to n, i.e., in n^k for some k. In what follows we define the complexity classes used in the original proofs of this thesis, and we further assume the standard definitions of the LOGSPACE, LOGCFL, EXPTIME, CONEXPTIME and PSPACE complexity classes that are mentioned in related work.

- The class AC^0 contains problems that can be recognized by a boolean circuit of constant depth and polynomial number of unlimited fan-in AND, OR and NOT gates.

- The classes PTIME and NP contain decision problems that can be solved on a deterministic and nondeterministic, respectively, Turing machine in polynomial time. The PTIME problems are called *tractable*, or solvable in practice.

- The class CONP contains decision problems whose complements are in class NP.

- The class DP contains decision problems that are intersections of a problem in NP and a problem in CONP.

We can establish the following relationships between the classes mentioned above: $AC^0 \subset$ PTIME $\subseteq$ NP $\cap$ CONP. Whether the inclusions of PTIME into NP and CONP are strict or not, and whether NP = CONP remain open problems; the inclusions are commonly believed to be strict.

Polynomial Hierarchy Polynomial hierarchy is the hierarchy of complexity classes that extend PTIME, NP and CONP with *oracle calls* — calls to a black-box Turing machine which can solve decision problems in a certain complexity class in one step; the output of an oracle can be further used in the main computation [113]. In particular, the classes Σ_2^P and Π_2^P contain decision problems that can be solved on an NP and a CONP machine with an NP oracle, respectively. It holds that

NP $\subseteq \Sigma_2^P$ and coNP $\subseteq \Pi_2^P$, although it is not known whether the inclusions are strict.

Upper and Lower Bounds A (decision) problem P is said to be $\mathcal{C}$-hard for a complexity class $\mathcal{C}$ if any other problem P' in $\mathcal{C}$ can be *reduced* to P — if an instance of P' can be transformed into an instance of P and solved using the algorithm or the Turing machine for P. For example, the "prototypical" hard problems for the classes of NP and coNP are the boolean satisfiability problem and its complement, respectively [33, 71]. The classical hard problem for the class of Π_2^P is $\forall\exists$SAT — the satisfiability problem for quantified boolean formulas of the form $\forall\bar{x}\exists\bar{y}\varphi$, where $\bar{x}$ and $\bar{y}$ are sequences of variables and φ is a quantifier-free boolean formula [113]. Hardness results for $\mathcal{C} \in \{\text{NP}, \text{coNP}, \Pi_2^P\}$ are achieved using *polynomial reductions* — reductions that transform one instance into the other one in polynomial time, and call the algorithm for solving the harder problem. Finally, a problem is $\mathcal{C}$-complete, if it is $\mathcal{C}$-hard and it belongs to the class $\mathcal{C}$. When proving that a problem is $\mathcal{C}$-complete, we often refer to the membership in $\mathcal{C}$ as the upper (complexity) bound and to it being $\mathcal{C}$-hard as the lower bound.

2.2 Formalisms and Query Languages

In this section we introduce the formalisms and query languages that are used in this thesis, namely the RDF data model, the SPARQL query language, and the language of conjunctive queries as well as the fragment of SPARQL used in the entity comparison framework. We define the key decision problems for query languages that are relevant for the current work, cover the syntax and semantics and discuss the complexity results for the decision problems for every language.

2.2.1 Decision Problems over Queries

In what follows we assume the standard definitions of a query and a database [1]. The three fundamental problems over query languages are *query evaluation*, *query containment* and *query equivalence* problems [1, 64, 122]. We start with query evaluation. Without going into formal details yet, assume that when a query Q is evaluated over a dataset D (a database or an RDF graph), a set of *answers* that match the query in that dataset, denoted $Q(D)$, is returned. Then the query evaluation problem consists in finding $Q(D)$ for the given Q and D, and it is the main problem in query processing. It can be formulated as a decision problem as follows:

Query Evaluation

Input: Query Q, dataset D, potential answer a

Question: Does $a \in Q(D)$ hold?

When discussing the complexity of a decision problem, e.g., of the query evaluation problem, one may consider data, query or combined complexity [122]. *Data complexity* is measured with respect to the size of the data, while the query is fixed. *Query complexity* is measured with respect to the size of the query, while the data source is fixed. Lastly, *combined complexity* is measured with respect to the size of all inputs of the problem, i.e., the data and the query. As in real world applications it is very common that the size of the query is orders of magnitude smaller than the size of the data, tractable data complexity plays a crucial role in the choice of a query language. For most query languages data complexity is lower than combined or query complexity. For example, query evaluation for Relational Algebra is PSPACE-complete in combined complexity and is in AC^0 in data complexity [122].

We now move the decision problems that involve two queries, namely containment and query equivalence. Containment and equivalence are two notions that help establish relations between queries. They play an important part in query pro-

cessing, optimization and rewriting, since it is often necessary to transform a query to an equivalent or a related one. They are defined as follows:

Definition 1 (Query containment and equivalence)**.** *Let Q_1 and Q_2 be two queries. Then Q_1 is* contained *in Q_2, denoted $Q_1 \subseteq Q_2$, if $Q_1(D) \subseteq Q_2(D)$ for any dataset D. Analogously, Q_1 is* equivalent *to Q_2, denoted $Q_1 \equiv Q_2$, if $Q_1(D) = Q_2(D)$ holds for any dataset D.*

Using the notions defined above we can now formulate two fundamental decision problems over queries:

Query Containment

Input: Queries Q_1 and Q_2

Question: Does $Q_1 \subseteq Q_2$ hold?

Query Equivalence

Input: Queries Q_1 and Q_2

Question: Does $Q_1 \equiv Q_2$ hold?

Query containment and query equivalence problems are closely related to each other. In fact, given two queries Q_1 and Q_2, it holds that

$$Q_1 \equiv Q_2 \text{ if and only if } Q_1 \subseteq Q_2 \text{ and } Q_2 \subseteq Q_1.$$

Moreover, for languages that allow conjunction, e.g., conjunctive queries, it also holds that

$$Q_1 \subseteq Q_2 \text{ if and only if } Q_1 \wedge Q_2 \equiv Q_1,$$

Since query containment can be solved using query equivalence, and vice versa, the complexity of the two problems is the same for a given language. For example,

both problems are undecidable in Relational Algebra [117] and are NP-complete for conjunctive queries [25, 102].

2.2.2 RDF Data Model

The Resource Description Framework (*RDF*) is a widely adopted data model for representing information about Web resources [37]. The information is kept in a form of machine-readable statements about resources — *triples* — and the resources are referenced using Internationalized Resource Identifiers (*IRIs*) and *blank nodes*. IRIs act as unique string identifiers for resources, whereas blank nodes reference anonymous Web resources and can be seen as existentially quantified variables. Triples may also contain *literals* — data values like numbers, dates, text strings, etc. We assume that literals include all integers. We refer to IRIs, literals and blank nodes collectively as *elements* of an RDF dataset, and we refer to IRIs and literals collectively as *entities*.

A triple is a tuple of the form $(s, p, o) \in (\mathbf{U} \cup \mathbf{B}) \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L} \cup \mathbf{B})$, where $\mathbf{U}$, $\mathbf{L}$ and $\mathbf{B}$ be pairwise disjoint, countably infinite sets of IRIs, literals and blank nodes, respectively. In our framework the first and the third positions in a triple are called *vertex positions*. The elements of a triple are conventionally named the *subject*, the *predicate* and the *object*. Each triple expresses a statement that the subject resource is in some relation to the object resource, and this relation is specified by the predicate resource. For example, a triple

$$t_{ex} = (\texttt{http://en.wikipedia.org/wiki/Quentin\_Tarantino},\ \texttt{http://xmlns.com/foaf/0.1/age},\ \texttt{54})$$

expresses a statement that Quentin Tarantino is 54 years old, the subject and the predicate of the triple being IRIs and the object being a literal. IRIs have a specific

syntax [20]. Since their aim is to serve as unique identifiers, some of the IRIs are lengthy and they share a common substring called a *namespace IRI*. In order to improve readability of RDF documents, namespace IRIs are often abbreviated as *prefixes*; a namespace-IRI-to-prefix pairing is given at the beginning of an RDF document, and the prefix is used throughout the rest of the document. For example, if we write the triple t_{ex} using Turtle [15], a common file format for expressing RDF data, we may use the following prefixes:

```
@prefix wiki: <http://en.wikipedia.org/wiki/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
wiki:Quentin_Tarantino foaf:age 54 .
```

In this thesis we are going to use short IRIs, without the namespace part, e.g., *Quentin_Tarantino* or *age*.

An (*RDF*) *graph* is a finite set of triples [92]. Indeed, an RDF dataset can be represented as a directed labelled graph in which each triple is depicted as an edge, with the subject (resp. the object) labelling the start (resp. end) vertices, and the predicate labelling the edge itself. For example, the triple t_{ex} given above can be graphically represented as in Figure 2.

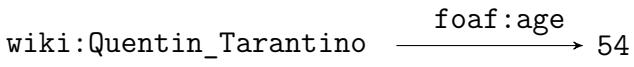


Figure 2: Example RDF triple t_{ex} in graphical representation

2.2.3 The SPARQL Query Language

SPARQL is the standard language for querying RDF graphs [52]. While SPARQL provides a rich set of operators, its core component is a *basic graph pattern* — a non-empty finite set of *triple patterns* of the form $(\mathbf{U} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{L} \cup \mathbf{V})$, where $\mathbf{U}$,

L and **V** are pairwise disjoint, countably infinite sets of IRIs, literals and variables, respectively. A *term* is an element from $\mathbf{U} \cup \mathbf{L} \cup \mathbf{V}$. By $\mathsf{term}(exp)$ and $\mathsf{var}(exp)$ we denote a set of all terms and variables, respectively, occurring in some SPARQL expression *exp*. Below we give a brief description of syntax and semantics for the core fragment of SPARQL and outline its most important complexity results, and we specify the subset of the core SPARQL that is going to be used in our framework in the next subsection.

Syntax of Core SPARQL

We are now ready to define the full syntax of SPARQL, and we start with the type of expressions called *built-in conditions*. Common approaches [8, 91, 92], restrict SPARQL built-in conditions to boolean combinations of equalities of the form:

1. if $?X, ?Y \in \mathbf{V}$ and $c \in \mathbf{U} \cup \mathbf{L}$, then $bound(?X)$, $?X = c$ and $?X = ?Y$ are built-in conditions;
2. if R_1 and R_2 are built-in conditions, then $(\neg R_1), (R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in conditions.

We consider another type of built-in conditions called *arithmetic comparisons* — expressions of the form $(?X \lhd n)$, where $?X$ is a variable in **V**, n is an integer, and $\lhd$ is a comparison symbol in $\{<, \leq, >, \geq\}$. A (*arithmetic*) *filter condition* is a finite (possibly empty) set of arithmetic comparisons.

Built-in conditions can be part of *graph patterns*.

Definition 2 (Graph pattern [91])**.** *The syntax of SPARQL* graph patterns *is defined recursively:*

- *A basic graph pattern is a pattern.*
- *If* P_1 *and* P_2 *are graph patterns, then* $(P_1$ AND $P_2)$, $(P_1$ OPT $P_2)$ *and* $(P_1$ UNION $P_2)$ *are graph patterns.*

- *If P is a graph pattern and R is a built-in condition, then* (P FILTER R) *is a graph pattern.*

For example, a graph pattern ($?X$, *isa*, *director*) AND ($?X$, *age*, $?Y$) FILTER ($?Y = 54$) OPT ($?X$, *nationality*, *American*) intuitively gives a description of an entity that is a director whose age is 54, and whose nationality is American, in case the information about the nationality if available in the RDF data.

Lastly, we consider SPARQL queries. Any SPARQL pattern can be viewed as a query, and it returns a set of answers over all variables that appear in the pattern. For example, given an RDF graph G, a pattern $P = (?X, marriedTo, ?Y)$ AND ($?X$, $worksIn$, $?Z$) returns all possible answers in G over variables $?X$, $?Y$, $?Z$. However, in SPARQL one can also use four specific *query forms*, which are manifested by four respective operators SELECT, CONSTRUCT, DESCRIBE and ASK [92]. For example, the following is a SELECT query:

$$\text{SELECT } W \text{ WHERE } P,$$

where P is a graph pattern and W is a finite set of variables. A pattern without any specified query form is equivalent to a SELECT query where W contains all variables in P, denoted SELECT $*$ WHERE P.

SPARQL queries in which the WHERE clause consist only of triple patterns, i.e., queries in which P is a basic graph pattern, correspond to Select-Project-Join queries in Relational Algebra [25], and therefore to Conjunctive Queries [57] (see Section 2.2.4). Such queries are sometimes called the conjunctive fragment of SPARQL. Full SPARQL is equivalent in expressive power to Relational Algebra [3], however, it is tailored to query RDF data, since in its nature SPARQL is a graph-matching query language [9].

Semantics of SPARQL

The semantics of SPARQL expressions is defined using valuation mappings, or valuations. A *valuation* of a finite set of variables $?\bar{X}$ from $\mathbf{V}$ is a mapping from $?\bar{X}$ to $\mathbf{U} \cup \mathbf{L} \cup \mathbf{B}$. The domain of a valuation μ, denoted $dom(\mu)$, is a set of variables on which μ is defined. Two valuations μ_1 and μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, if $\mu_1(?X) = \mu_2(?X)$ for all variables $?X \in dom(\mu_1) \cap dom(\mu_2)$ [91]. A set of valuations is commonly denoted by Ω. Four operations over sets of valuations we are going to use next are join, union, difference, and left outer join:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1,\ \mu_2 \in \Omega_2,\ \mu_1 \sim \mu_2\},$$

$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},$$

$$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2\ \mu \nsim \mu'\},$$

$$\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2).$$

Now we can define the semantics of a graph pattern, following its recursive nature.

Definition 3 (Graph pattern evaluation)**.** *Given an RDF graph G, an* evaluation *of a graph pattern, denoted $[\cdot]_G$, is a function from graph patterns to sets of valuations such that:*

- $[t]_G = \{\mu \mid dom(\mu) = var(t)$ *and* $\mu(t) \in G\}$*, where t is a triple pattern;*
- $[P_1$ AND $P_2]_G = [P_1]_G \bowtie [P_2]_G$*;*
- $[P_1$ OPT $P_2]_G = [P_1]_G \mathbin{⟕} [P_2]_G$*;*
- $[P_1$ UNION $P_2]_G = [P_1]_G \cup [P_2]_G$*;*
- $[P_1$ FILTER $R]_G = \{\mu \in [P]_G \mid \mu \models R\}$*, where R, R_1, R_2 are filter conditions, and $\mu \models R$ holds if:*

- R *is* $bound(?X)$ *and* $?X \in dom(\mu)$;
- R *is* $?X = c$, $?X \in dom(\mu)$ *and* $\mu(?X) = c$;
- R *is* $?X = ?Y$, $?X \in dom(\mu)$, $?Y \in dom(\mu)$ *and* $\mu(?X) = \mu(?Y)$;
- R *is* $\neg R_1$ *and* $\mu \not\models R_1$;
- R *is* $(R_1 \vee R_2)$ *and* $(\mu \models R_1$ *or* $\mu \models R_2)$;
- R *is* $(R_1 \wedge R_2)$, $\mu \models R_1$ *and* $\mu \models R_2$;
- R *is* $(?Y \lhd n)$ *and* $(\mu(?Y) \lhd n)$, $\lhd$ *being the comparison symbol.*

Complexity of SPARQL queries

The query evaluation problem, defined in Section 2.2.1, can be instantiated for SPARQL as follows [92]: given an RDF graph G, a graph pattern P and a mapping μ, does $\mu \in [P]_G$ hold? The complexity of answering the evaluation problem depends on the specific fragment of SPARQL and its expressiveness. The fragments are encoded as SP[Λ] or S-SP[Λ], where S- indicates that the SELECT queries are allowed. The symbol $\Lambda \subseteq \{A, F, O, U\}$ denotes a set of the operators present in this fragment, the letters corresponding to AND, FILTER, OPT and UNION, respectively. For example, SP[AO] is a fragment of SPARQL, in which graph patterns are constructed only using AND and OPT operators.

Table 1 lists the combined complexity results for various fragments of SPARQL. In particular, it states that the evaluation problem for the core fragment of SPARQL is PSPACE-complete in the combined complexity. However, the SPARQL core fragment (and all its subfragments) is in LOGSPACE in the data complexity, when the pattern P is fixed, which is an important result since the size of the input graph is usually considerably bigger than that of P. Notice also that all fragments that allow for the OPT operator are PSPACE-complete.

SPARQL fragments	Complexity class
SP[A], SP[F], SP[U], SP[AF], SP[FU]; s-SP[F], s-SP[U], s-SP[FU]	PTIME
SP[AU], SP[AFU]; s-SP[A], s-SP[AF], s-SP[AU], s-SP[AFU]	NP-complete
any fragment with OPT, including SP[O], SP[AO], SP[AOF], SP[AOU] and the core fragments SP[AOFU], s-SP[AOFU]	PSPACE-complete

Table 1: Combined complexity of the query evaluation problem for different fragments of SPARQL [13, 92, 103]

2.2.4 Conjunctive Queries

Conjunctive queries (CQs) are a query language that is a fragment of first-order logic that has a fairly simple syntax yet high expressive power and some desirable computational properties. CQs are one of the simplest and most common types of queries that can be expressed over a relational database or an RDF graph [46]. They correspond to Select-Project-Join queries in Relational Algebra [25] and to the conjunctive fragment of SPARQL. A conjunctive query is a function-free first-order logic formula that is an existentially quantified conjunction of atomic formulae, also called atoms: $Q(x_1, \ldots, x_k) = \exists\, x_{k+1}, \ldots, x_m . A_1 \wedge \ldots \wedge A_n$, where x_j are variables and $A_i = R_i(c_1, \ldots, c_l)$ are atoms, with R_i a relation and each c_j either a variable or a constant. A CQ is called boolean, if $k = 0$; it is called *monadic*, if $k = 1$.

Let D be some data source instance (e.g., an RDF graph or a database). The semantics of a CQ $Q(x_1, \ldots, x_k)$ is defined using a valuation — a total mapping from the variables of Q, denoted $\mathsf{var}(Q)$, into the domain of D, denoted $dom(D)$. An tuple t consisting of elements from D is an answer to Q over D if there exists a valuation μ of $\mathsf{var}(Q)$ such that $\mu(x_1, \ldots, x_k) = t$, and for every atom A_i from Q it

holds that $\mu(A_i) \in D$.

Query containment and query evaluation problems are NP-complete for CQs (with respect to combined complexity in case of query evaluation) [25, 102]. Moreover, data complexity of the query evaluation problem is in AC^0 for CQs [122], i.e., in LOGSPACE. The aforementioned results hold both for queries asked over relational databases and over RDF graphs, since RDF data can be viewed as a relational database table with three columns: subject, predicate and object.

Unlike for Relational Algebra, query containment for CQs can be reduced not only to query equivalence, but also to query evaluation [25]. This link is established using the Homomorphism problem, a fundamental algorithmic problem that asks for the existence of a homomorphism between two database instances (homomorphisms are formally defined in Section 2.3), and leads to an important result, the Homomorphism Theorem [25], that says that following are equivalent:

- $Q_1 \subseteq Q_2$, and
- there exists a homomorphism from the canonical instance of Q_2 to the canonical instance of Q_1,

where the *canonical instance* of a query Q is a database, in which variables and constants from Q are elements and atoms are facts. It follows directly that in order to prove query subsumption $Q_1 \subseteq Q_2$, it is sufficient to show query homomorphism from Q_2 to Q_1, as defined in Section 2.3.

Some CQs are redundant in structure: they contain atoms that can be removed from the query so that the updated query is equivalent to the original one. The process of removing such atoms is called *query minimization*. There exists a well-known procedure for minimizing a CQ Q: choose a query atom at random, remove it from Q, obtaining a new query Q'; then check whether $Q' \subseteq Q$ holds. If yes, repeat the process, using Q'; if not, try removing another atom from Q. Query

minimization is an important step in many applications, as it makes sense first to minimize the query and then to process it further. However, since the problem of deciding whether a CQ Q' is a minimal equivalent of a CQ Q is NP-complete, there is no polynomial-time algorithm for computing a minimized CQ (unless P = NP); the same holds for the queries used in our framework (see Section 2.2.5).

CQs with Arithmetic Comparisons

CQs with arithmetic comparisons (CQACs) is a larger query language that extends CQs with comparison atoms. CQACs are equivalent to Select-Project-Join queries in Relational Algebra that also include equality and inequality conditions, and they are defined as follows [63]:

Definition 4. *A CQ with arithmetic comparisons is a query Q of the form $Q(\bar{x}) = \exists\, \bar{y}.(C_1 \wedge \ldots \wedge C_m) \bigwedge (L_1 \wedge \ldots \wedge L_n)$, where:*

- *$\bar{x}$ is a set of free (answer) variables,*
- *$\bar{y}$ is a set of existentially quantified variables,*
- *C_i is an atom,*
- *L_i is an arithmetic comparison of the form $v\,\theta\,c$, where $v \in \bar{x} \cup \bar{y}$ is a variable, c is an integer, and $\theta \in \{=, <, \leq, >, \geq\}$.*

The semantics of CQACs is defined similarly to that of CQs: a tuple t is an answer to Q under a valuation μ if additionally $\mu(v)\ \theta\ c$ holds for every arithmetic comparison L_i.

Query evaluation is NP-complete for CQACs. We are, however, particularly interested in the problem of containment for CQACs, since we use the containment complexity result in our work (see Chapter 5). Containment of CQs with arithmetic comparisons is Π_2^P-complete [63,121]. Such high complexity stems from the fact that,

unlike for CQs, the homomorphism property does not hold for CQACs: instead of checking the existence of only one homomorphism, one needs to check the existence of all possible homomorphisms with respect to the query terms ordering. However, we can extend the definition of the homomorphism between queries so that it becomes a *sufficient* condition for query containment: there exists a homomorphism h from Q to Q' if additionally the set of arithmetic comparisons from Q' entails the set of arithmetic comparisons from Q under h (see Section 2.3). Then $Q' \subseteq Q$ holds if such a homomorphism exists [87]. Note that the converse proposition does not hold.

Some specific fragments of CQACs have lower complexity of the containment problem. If we restrict θ to be either in $\{=, <, \leq\}$ or in $\{=, >, \geq\}$, we get the languages of *left semi-interval queries* or *right semi-interval queries*, respectively, which are fragments of the language of CQACs and for which the homomorphism property holds, making the containment problem for these fragments NP-complete. If we keep the full set of θ, but restrict the structure of CQs to be acyclic, in particular, if we impose Berge-acyclicity on CQs, the containment problem is in coNP [107].

Acyclic CQs

There exists a type of conjunctive queries with a specific structural property — *acyclicity*. *Acyclic queries*, while restricted in form, have favourable computational properties, thus they are well researched and heavily used in practice [17,42,49,130]. In particular, while query evaluation and containment are NP-complete for arbitrary CQs [25], they are LOGCFL-complete for acyclic CQs, therefore they allow for efficient parallelisable algorithms [49].

There exist several definitions of acyclicity, Berge-acyclicity being the most restrictive. Let Q be a CQ, with $\mathsf{var}(Q)$ all variables in Q, and let E_Q be the set that contains, for each atom in Q, the set of variables appearing in this atom.

Definition 5 (Incidence graph)**.** *An* incidence graph $Inc(Q)$ *is an undirected bipar-*

tite graph where $\mathsf{var}(Q) \cup E_Q$ *is the set of its vertices, and* $\{V, e\}$ *is an edge if and only if a variable* $V \in \mathsf{var}(Q)$ *appears in a set* e *from* E_Q.

An incidence graph may contain *cycles* — a tuple of distinct vertices $(v_1, \ldots, v_k)$ from $Inc(Q)$, with $k \geq 2$, such that for every pair of vertices v_i, v_{i+1} there exists an edge in $\{v_i, v_{i+1}\}$ in $Inc(Q)$ for $1 \leq 1 < k$, and there exists an edge $\{v_1, v_k\}$ in $Inc(Q)$. A CQ is *Berge-acyclic* if its incidence graph is acyclic. For example, the incidence graph corresponding to the following CQ Q_a looks as depicted in Figure 3:

$$Q_a(X) = \exists Y, Z.(X, isa, Y) \wedge (Y, name, alice) \wedge (Y, friend, Z) \wedge (Y, colleague, Z).$$

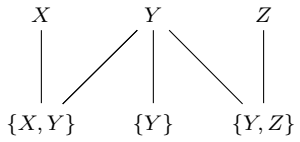


Figure 3: Incidence graph for an example CQ Q_a

2.2.5 Queries Used in the Entity Comparison Framework

In this thesis we are going to use a very specific fragment of core SPARQL. In particular, we restrict built-in conditions to arithmetic filter conditions, and we use the subset of SPARQL patterns that include:

- basic graph patterns, and optionally
- expressions of the form (P FILTER R), where P is a basic graph pattern, and R is an arithmetic filter condition.

Note that we restrict the filter conditions to only arithmetic comparisons between variables and constants. This is justified by the fact that all other comparisons, such as general inequalities between variables and IRIs, have very little meaning in the

context of entity comparisons, and moreover may flood comparison queries hiding the essential parts.

A basic graph pattern P is *connected* if for every two triple patterns $p, p' \in P$ there is a sequence of triple patterns $p_1, \ldots, p_m$ in P such that $p_1 = p$, $p_m = p'$, and for all $1 \leq i < m$ there exists a term t appearing in a vertex position in both p_i and p_{i+1}. We say that a query containing a basic graph pattern is *connected* if so is its basic graph pattern. We concentrate on (SPARQL) *queries* of the following form:

Definition 6. *A* query *is an expression of the form*

$$\textsc{Select}\ ?X\ \textsc{Where}\ P\ \textsc{Filter}\ C$$

where P is a connected basic graph pattern, $?X \in \mathsf{var}(P)$ is the answer variable *of the query and C is an* arithmetic filter condition*, or AFC, satisfying $\mathsf{var}(C) \subseteq \mathsf{var}(P)$.*

We are now able to redefine the semantics of our queries using valuations in a simpler way: valuations applied to basic graph patterns and arithmetic comparisons are defined as before. An element e from $\mathbf{U} \cup \mathbf{L} \cup \mathbf{B}$ is an *answer* to a query Q over a graph G if there exists a valuation ν of $\mathsf{var}(P)$ so that $\nu(?X) = e$, $\nu(P) \subseteq G$ and $\nu(?Y) \lhd n$ holds for each comparison $(?Y \lhd n)$ in C, with $?Y \in \mathsf{var}(C)$ and $\mathsf{var}(C) \subseteq \mathsf{var}(P)$; such ν is called the *satisfying valuation*. We denote by $[Q]_G$ the set of all answers to Q over G.

We are now able to instantiate the notions of containment and equivalence from Section 2.2.1 for our queries using the notation in the framework. Note that query containment is sometimes called query subsumption, and the two terms are used interchangeably in this thesis. A query Q_1 is *subsumed* by a query Q_2, written $Q_1 \subseteq Q_2$, if $[Q_1]_G \subseteq [Q_2]_G$ for every graph G. Query Q_1 is *strictly subsumed* by query Q_2, denoted by $Q_1 \subset Q_2$, if $Q_1 \subseteq Q_2$ and $Q_2 \not\subseteq Q_1$. Finally, Q_1 and Q_2 are *equivalent*, denoted by $Q_1 \equiv Q_2$, if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. Subsumption and

equivalence allow us to compare queries relative to their specificity, so we sometimes say that Q_1 is *(strictly) more specific* than Q_2 if Q_1 is (strictly) subsumed by Q_2 (see Chapter 4).

Our queries correspond to connected monadic conjunctive queries with arithmetic comparisons [63] (see Section 2.2.4), restricted to signatures over a single ternary relation and using no comparisons between variables. If the filter condition C is empty, we use the simplified query notation SELECT $?X$ WHERE P; such queries without arithmetic comparisons correspond to monadic conjunctive queries (see Section 2.2.4). From these language correspondences we can immediately conclude that query subsumption in our framework is NP-complete for queries with an empty arithmetic filter condition [25], and it is Π_2^P-complete if both queries contain non-empty arithmetic filter conditions [63, 121] (see Section 2.2.4). As for query evaluation, it is NP-complete for queries both with and without the arithmetic filter conditions [25].

2.3 Graph Theory and Homomorphisms

In this work we use formalisms and query languages, in which objects (RDF graphs, queries) and their derivative structures (incidence graphs, pair trees) can be represented as graphs. In this section we present the key notions from graph theory that are used in this thesis [23, 39, 126]. A graph is a tuple (V, E) where V is a non-empty set of vertices, and E is a set of edges between pairs of vertices. A graph is *undirected* if edges link two vertices symmetrically:

$$E \subseteq \{\{u, v\} \mid u, v \in V\};$$

a graph is *directed* if edges link two vertices asymmetrically:

$$E \subseteq \{(u, v) \mid u, v \in V\}.$$

A subgraph of a graph G is itself a graph whose vertex set and edge set are subsets of those of G. Both vertices and edges in a graph can be labelled.

The two fundamental relasionships between two graphs that we use in this thesis are isomorphism and homomorphism. Given two undirected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$:[1]

- a *homomorphism* between G and H is a mapping $f : V_G \to V_H$ such that for every edge $\{u, v\}$ in E_G there exists an edge $\{f(u), f(v)\}$ in E_H;
- an *isomorphism* between G and H is a bijection $f : V_G \to V_H$ such that an edge $\{u, v\}$ in E_G exists if and only if an edge $\{f(u), f(v)\}$ exists in E_H.

Isomorphisms and homomorphisms are sometimes called edge-preserving and structure-preserving maps, an isomorphism being a type of a homomorphism. Deciding whether there exists a homomorphism between two graphs is an NP-complete problem [55]. The problem of deciding whether there exists an isomorphism between two graphs is in NP, however, it is not known whether it is NP-hard or it can be solved in polynomial time.

The notion of a homomorphism can be straightforwardly adapted to queries. Next we are going to define query homomorphism for the query language used in our framework (see Section 2.2.5).

Definition 7. *Let* $Q =$ Select $?X$ Where P Filter C *and* $Q' =$ Select $?X'$ Where P' Filter C' *be two queries, with* P, P' *basic graph patterns and* C, C' *arithmetic filter conditions. A* homomorphism h *from* Q *to* Q' *is a function from* $\mathsf{term}(Q)$ *to* $\mathsf{term}(Q')$ *such that*

[1] The definitions for directed graphs are analogous.

(a) entities are mapped to themselves,

(b) the answer variable $?X$ *is mapped to the answer variable* $?X'$*, i.e.,* $h(?X) = ?X'$*,*

(c) for every triple pattern $(t_s, t_p, t_o) \in P$ *there exists a triple pattern* $(h(t_s), h(t_p), h(t_o)) \in P'$*, and*

(d) the set of arithmetic comparisons C' *entails the set of arithmetic comparisons* C *under* h*:* $C' \models h(C)$*.*

If both queries do not contain arithmetic filter conditions, the condition *(d)* can be ignored. The definition can be straightforwardly adapted to CQs and CQACs.

Some graphs are homomorphic to their own subgraphs. The *core* of a graph is its smallest subgraph which is also a homomorphic image [56]. The notion of a core can be adapted to queries, e.g., to conjunctive queries: a core of a query is its smallest subquery for which there exists a homomorphism from the original query. Since taking the core of a query is the same as query minimization, the two terms are used interchangeably in this thesis. For CQs and conjunctive SPARQL queries, as well as for the queries used in our framework (see Section 2.2.5) the core of the query is unique (i.e., all minimal queries are isomorphic).

Chapter 3

Related Work

In this chapter I give a state of the art overview of the areas that are relevant for entity comparison, such as query reverse engineering, computing least common subsumers, and relatedness-based explanations over graph data. In addition, in Section 3.2 I give an overview of the latest research on logic-based similarity explanations over RDF data.

3.1 Query Reverse Engineering

Computing comparison queries from two entities in an RDF graph can be viewed as a special case of *query reverse engineering*. The problem of reverse engineering a query given some examples originated in late 1970s and was first introduced for the domain of relational databases [134]. Later it was extensively researched with respect to different query formats: regular languages [4] [5] [6], XML queries [28] [112], relational database queries [118] [119] [132], graph database queries [24] and SPARQL queries [8].

Formally, query reverse engineering (QRE) is a problem of reconstructing a query in some query language, given a database (or a dataset) and a set of examples, such that the query returns the examples when evaluated over the database [119].

Coen_movies

movieId	title	year
1	"The Big Lebowski"	1998
2	"A Serious Man"	2009
3	"Hail, Caesar!"	2016

Lead_actors

id	actor
1	Jeff Bridges
2	Michael Stuhlbarg
3	George Clooney

Figure 4: Movie database D

Examples:

Jeff Bridges	"The Big Lebowski"
George Clooney	"Hail, Caesar!"

Figure 5: User examples

The corresponding decision problem is the problem of whether there exist such a query, given input data. For instance, given a database D from Figure 4 and a set of user examples from Figure 5, the following SQL query fits the examples with respect to D:

```
SELECT actor, title
FROM Coen_movies, Lead_actors
WHERE Coen_movies.movieId = Lead_actor.id.
```
(3.1)

The problem has several different formulations, depending on the type of the input and the conditions imposed on the output query:

- there may be only positive examples (e.g., the result table), or there may be both positive and negative examples — answers that should not be in the answer set of the query;
- the result table may consists of complete tuples [119] (e.g., table A), or it may be a set of partial tuples (e.g., table B), or it may consist of keywords that do not necessarily directly match values in the dataset, in which case QRE transforms into the problem of keyword-based search [89] [131] (e.g., table C);

A

person	profession
Q. Tarantino	director
U. Thurman	actress
L. Bender	producer

B

name	company	position
Andrew Ng	Baidu	
Peter Norvig		
	CEO	Facebook

C

Hitzler Rudolph Semantic Web
Abiteboul databases
Russell artificial intelligence

- the result table may consist only of positive examples, or it may contain additional tuples.

With multiple formulations of the QRE problem come different approaches and different namings: query reverse engineering [119], query by example [134], query by output [118], definability problem [7], instance equivalent query [118], etc.

Reverse Engineering CQs Query reverse engineering has been well-studied for conjunctive queries over relational databases [14,115,128]. Barceló et al. [14] consider two separate problems. Given a database instance D and n-ary relations S^+ and S^- of positive and negative examples, respectively, decide whether there exists a query Q such that:

- either Q precisely defines positive examples: $Q(D) = S^+$ (*definability problem*), or

- evaluation of Q contains positive, but not negative examples: $S^+ \subseteq Q(D)$ but $Q(D) \cap S^- = \emptyset$ (*query-by-example problem*).

Both problems are proven to be coNExpTime-complete [115, 128]. ten Cate et al. [115] further restricts the definability problem, showing that the same complexity result holds already for unary queries over a single binary relation.

Reverse Engineering CQs with Ontologies Gutierrez et al. [51] studied the query-by-example (QBE) and definability problems of CQs and union of CQs (UCQs) with the presence of Horn-$\mathcal{ALC}$ and Horn-$\mathcal{ALCI}$ knowledge bases. QBE is 2-ExpTime-complete for Horn-$\mathcal{ALCI}$ and both CQs and UCQs. In case of Horn-$\mathcal{ALC}$ ontologies, QBE is coNExpTime-complete for CQs and ExpTime-complete for UCQs. The same results hold for the definability problem.

Reverse Engineering SPARQL Queries While QRE is fairly well studied for the area of databases, there has been only one attempt to adapt QRE to RDF data [8, 38]. Examples in the RDF setting come in the form of variable mappings, e.g., a set of mappings $\Omega = \{\{?X \rightarrow Bruce\_Willis, ?Y \rightarrow Uma\_Thurman\}, \{?X \rightarrow Jeff\_Bridges, ?Y \rightarrow Julianne\_Moore\}\}$ contains two example mappings that provide information about two variables each (valuation mappings are defined formally in Section 2.2.3). The complexity of QRE with partial positive examples in SPARQL is currently known for the well-designed fragments of SPARQL that contain $\{\mathsf{AND}\}$, $\{\mathsf{AND}, \mathsf{OPT}\}$ and $\{\mathsf{AND}, \mathsf{OPT}, \mathsf{FILTER}\}$ operators; it is in PTime, Σ_2^P-complete and NP-complete, respectively [8].

Reverse Engineering SPJ-Queries The work of Weiss and Cohen [125] studies the problem of reverse engineering select-project-join (SPJ) relational algebra (RA) queries from positive and negative examples. Given a class of queries L (i.e., a certain fragment of RA), a database D and sets of positive and negative examples E^+ and E^-, decide whether there exists a query $Q \in L$ such that:

- $schema(Q) = schema(E^+) = schema(E^-)$,

- $E^+ \subseteq Q(D)$,[1]
- $E^- \cap Q(D) = \emptyset$.

The work studies the QRE problem and the respective computational problem (referred to as the *learning* problem) for various fragments of SPJ-queries, considering as parameters the size of the query (i.e., unbounded or bounded by a threshold), the size of the database schema and the number of input examples. The parameter values are treated either as constants or as variables, which leads to two types of complexity results presented in the paper.

The paper aligns perfectly with our work on comparison queries since positive and negative examples, in particular when $|E^+| = 2$, can be viewed as inputs to similarity and difference queries, respectively. However, it studies a different query language, i.e., a different fragment of RA: Weiss and Cohen consider a fragment of RA that allows for projection (π), selection (σ) and natural join ($\bowtie$). The selection condition is a conjunction of comparison statements, the allowed operators being $=, \neq, \leq, \geq$. We are going to call this set of operators $O^{SPJ} = \{\pi, \sigma, \bowtie\}$. We consider a fragment that is equivalent to conjunctive queries (CQs) without inequalities, which in RA (in the named perspective) corresponds to selection that only allows for equalities, projection and natural join, as well as renaming (ρ) [1]. The two query languages are incomparable: on the one hand, the language L_1 in [125] allows for inequalities in selection condition and is not restricted to monadic queries; on the other hand, the language L_2 that we consider contains renaming and generalized selection. However, if we restrict L_1 to monadic queries and selections without inequalities, this fragment would be a strict subset of L_2. Consider an RDF graph G from Figure 6 and two

[1]Note that this decision problem (referred to as *satisfiability* problem) differs from the classical *definability* problem, where the condition is more strict: $E^+ = Q(D)$.

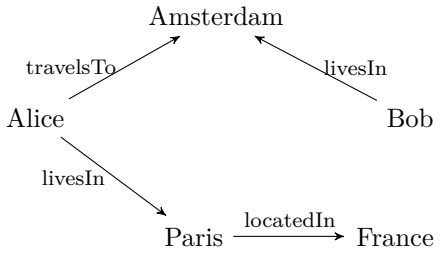


Figure 6: Example RDF graph G

queries:

$$Q_1 = \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, livesIn, Paris), (?X, travelsTo, Amsterdam)\},$$

$$Q_2 = \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, livesIn, ?Y), (?Y, locatedIn, France)\}.$$

Following the approach of Cyganiak [36], we can represent G as a *graph relation* (let us call it $Triple$) with three attributes s, p and o (for subject, predicate and object):

Triple

s	p	o
Alice	livesIn	Paris
Alice	travelsTo	Amsterdam
Bob	livesIn	Amsterdam
Paris	locatedIn	France

If we allow generalized selection, i.e., arbitrary logical connectives in the selection condition (instead of just the conjunction), the first query Q_1 can be rewritten into RA using only O^{SPJ} as follows:[2]

$$\pi_s(\sigma_{(p='livesIn' \wedge o='Paris') \vee (p='travelsTo' \wedge o='Amsterdam')}(Triple)).$$

[2]Note that if we restrict the selection condition to only consist of conjunctions of (in-)equalities, Q_1 cannot be rewritten using O^{SPJ}.

The second query Q_2 has a *path-like* structure, different from the structure of Q_1, and it cannot be rewritten using only O^{SPJ} either, since it requires renaming:

$$\pi_s(\sigma_{p='livesIn'}(\rho_{a/o}(Triple)) \bowtie \sigma_{p='locatedIn' \wedge o='France'}(\rho_{a/s,\, p'/p}(Triple))).$$

To sum up, there has been attempts to reverse engineer queries with arithmetic comparisons and inequalities. However, they all differ from the problem we suggest to study. Furthermore, reverse engineering of CQACs has not been studied yet, either in general form, or in case of monadic queries.

3.2 Least Common Subsumers

Another problem that is highly relevant for this work is the problem of computing *least common subsumers* (lcs) between concepts in Description Logics (DLs) [12]. The notion of lcs was first introduced in 1992 by Cohen et al. [29]. Formally, a lcs for two concept C_1 and C_2 is a concept C that:

(1) subsumes both input concepts: $C_1 \sqsubseteq C$ and $C_2 \sqsubseteq C$, and

(2) for any other concept D that subsumes C_1 and C_2 the following holds: $C \sqsubseteq D$,

where the subsumption of DL concepts is defined in the classical way [12]. Finding a lcs is one of the non-classical reasoning tasks in DLs. In particular, it is used while building a knowledge base bottom up, which can be viewed as learning (an ontology) from examples [120].

We could cast the problem of comparing two KG entities, and in particular the problem of computing similarity and most specific similarity queries for two entities in an RDF graph, as the problem of finding the least (i.e., most specific modulo subsumption) DL concept that contains both entities as instances. An important difference with respect to our setting is that DL concepts in logics such

as $\mathcal{EL}$ and $\mathcal{ALC}$ can only capture conjunctive queries that are both constant-free and tree-shaped. In this sense, our query language is more expressive, as it allows for arbitrarily-shaped connected CQs. The additional expressivity turns out to be critical: while a least DL concept may not exist (e.g., if the input graph has cycles then the least concept could be infinite), we will show in Section 5.1 that a most specific similarity query is always finite and can be computed in polynomial time.

We are not going to delve into the related work on lcs. However, in what follows we will give an overview of several recent works which adopt the notion of lcs, but define it over RDF data rather than DL concepts.

Common Subsumers in RDF Colucci et al. [30] view the task of computing a lcs for two RDF nodes as a means of clustering web resources that presumably have similar information content. While clustering of RDF resources has been mostly done using statistical distance-based approaches, the authors seek to infer clusters using the formal semantics of RDF. Taking into account the "hypothetically unlimited" size of the web resources dataset, the authors focus on the problem of finding a common subsumer (cs) rather than a least common subsumer.

One peculiarity of the work is that instead of operating over some arbitrary fixed RDF graphs, the authors consider the whole Semantic Web, sometimes referred to as "the Web" or "the Web of Data". Hence, in order to specify which fragment of the potentially unbound Web is involved in the computation of common subsumers, the notion of a *rooted graph*, or *r-graph* is introduced: an r-graph is a pair $\langle r, T_r \rangle$, where r is a IRI or a blank node and T_r is a subset of all triples with r as a subject. T_r is assumed to be the subset of triples relevant to r with respect to some boolean characteristic function σ_{T_r}. A trivial example of σ_{T_r} marks as relevant only triples that belong to a specific RDF dataset, e.g., DBpedia. Intuitively, T_r is some relevant subset of the neighbourhood of the node r (of depth 1) in an RDF graph. However,

the neighbourhood excludes the "incoming links".

The notion of common subsumers is defined inductively over rooted graphs:

Definition 8. *Let $\langle a, T_a\rangle$, $\langle b, T_b\rangle$ be two r-graphs, and x, w, y be blank nodes. The* common subsumer *(cs) is defined as follows:*

- *if $\langle a, T_a\rangle = \langle b, T_b\rangle$, then the cs is $\langle a, T_a\rangle$;*
- *if $T_a = \emptyset$ or $T_b = \emptyset$, then the cs is $\langle x, \emptyset\rangle$;*
- *otherwise the cs for $\langle a, T_a\rangle$ and $\langle b, T_b\rangle$ is an r-graph $\langle x, T\rangle$ such that:*
  - *T_a, $T_b \subseteq T$,*
  - *$\exists t_1 = (a, p, c)$, $T \models t_1$ and $\langle w, T\rangle$ is the cs for $\langle p, T_p\rangle = \langle q, T_q\rangle$, and*
  - *$\exists t_2 = (b, q, d)$, $T \models t_2$ and $\langle y, T\rangle$ is the cs for $\langle c, T_c\rangle = \langle d, T_d\rangle$.*

The tuple $\langle x, T\rangle$ (where x is a blank node) can be interpreted as a unary conjunctive query over an RDF graph presented as a graph relation *Triple* [36], with x being the answer variable.

The properties of idempotency, commutativity and associativity are shown for common subsumers defined over RDF graphs, and an anytime algorithm that computes a cs for two RDF resources is presented [30]. The algorithm is very similar to our algorithm of computing MSSQs: it computes a cs recursively, starting from two input nodes and at each iteration trying exhaustively to compute a cs for all possible pairs of triples for two input nodes. The authors extend their approach to computing a lcs rather than a cs in [31].

Learning Commonalities in RDF Another relevant work by El Hassad et al. [41] attempts to compare two Web resources using RDF data and the notion of *least general generalization*, or lgg. An idea of inductively learning a concept from a set of instances has been studied since the 1970s [82], and a framework for learning lggs

was proposed by Plotkin [97]. El Hassad applies the framework of Plotkin so that finding commonalities between resource descriptions "amounts to computing a *least general generalization* (lgg) of such descriptions" [41].

Given two RDF entities a and b, or rather graph neighbourhoods of a and b treated as two separated RDF graphs G_a and G_b, the lgg of G_a and G_b is defined via RDF(S) entailment rules [53]. Formally, given a set R of RDF(S) entailment rules, the following notions can be defined:[3]

- A *saturation* of an RDF graph G wrt R is another RDF graph G^∞ obtained from G by adding all implicit triples that can be derived from G using R. A saturation is always finite and unique up to blank node renaming.

- An RDF graph G *entails* an RDF graph G' wrt R, denoted $G \models_R G'$, iff there exists a homomorphism h from blank nodes of G' to values (IRIs, literals and blank nodes) of G such that $h(G') \subseteq G^\infty$.

- An *lgg* of RDF graphs $G_1, \ldots, G_n$ is an RDF graph G such that for $1 \leq i \leq n$

  · $G \models_R G_i$, and
  · for all G' such that $G' \models_R G_i$: $G \models_R G'$.

An lgg of RDF graphs *always exists and is unique up to entailment*, i.e., semantically unique, although it may have multiple syntactic forms due to redundant triples. It is shown that computing an lgg of n RDF graphs can be done iteratively, irrespective of the order of the graphs: $lgg(G_1, G_2, G_3) = lgg(G_1, lgg(G_2, G_3))$. Hence, the paper focuses on the functional problem of computing an lgg for *two* input graphs.

The computation of lgg is done using the notion of a *cover graph*: given RDF graphs G_1 and G_2, their cover graph G is an RDF graph constructed as follows: $(s_1, p, o_1) \in G_1$ and $(s_2, p, o_2) \in G_2$ iff $(s_3, p, o_3) \in G$, where

[3] any subset of RDF(S) entailment rules defined by the RDF Standard [53]

- $s_3 = s_1$ if $s_1 = s_2$ and s_1 is not a blank node, otherwise s_3 is a fresh blank node, and
- $o_3 = o_1$ if $o_1 = o_2$ and o_1 is not a blank node, otherwise o_3 is a fresh blank node.

Intuitively, a cover graph is a generalization of its input graphs since each of its triple (s, p, o) is a most general unifier of a pair of triples from input graphs. Formally, an lgg of two RDF graphs G_1 and G_2 wrt some R is the cover graph of their saturations wrt R.

El Hassad criticizes Colucci et al. [31, 32] for having a limited approach to the problem and in turn uses RDF(S) entailment when finding commonalities between two nodes in an RDF graph. The commonalities are also expressed in a form of an RDF graph rather than a higher-level graph pattern, as compared to our approach. To the best of our knowledge, while utilizing RDF(S) entailment rules, this work is the first step taken so far towards integrating ontological knowledge into computing similarities for RDF resources. However, our approach can easily accommodate RDF(S) statements by first materializing the graph and then applying the same algorithms for computing comparison queries. Furthermore, the work of [41] does not consider differences between the resources.

3.3 Explanations over RDF Graphs

There is a growing interest in techniques for discovering and explaining relationships between entities in an RDF graph [26, 54, 69]. These approaches are typically based on computing paths in the input graph connecting the input entities. Such paths are first computed via standard graph traversal algorithms, and then ranked according to certain structural and statistical measures [26]. We note that the problem of finding connections between entities is orthogonal to that of computing similarity

and difference queries for them. The two approaches are intended to capture different relations between nodes: the former explore possible paths that link the two nodes together, while the latter seek to find commonalities in the neighbourhoods of the input nodes.

Figure 7 illustrates an example of a connectedness explanation that does not correspond to any similarity query: while there is a path from *Alice* to *Bob* that connects these two target nodes, it cannot be captured by any query in our framework containing both nodes in the answer set.

$$\text{Alice} \xrightarrow{\text{follows}} \text{Jack} \xrightarrow{\text{follows}} \text{John} \xrightarrow{\text{follows}} \text{Bob}$$

Figure 7: An example of a relation between target nodes that is explained by connectedness, but not similarity

Conversely, Figure 8 is an example of a situation when the two nodes disconnected in a graph are nonetheless similar, which can be described by a similarity query $Q = \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, follows, ?Y)\}$.

$$\text{Alice} \xrightarrow{\text{follows}} \text{Jack} \qquad \text{John} \xleftarrow{\text{follows}} \text{Bob}$$

Figure 8: An example of a relation between target nodes that is explained by similarity, but not connectedness.

Furthermore, as already argued, the natural adaptations of such techniques to our setting do not enable entity comparison at a sufficiently high level of abstraction.

3.4 Other Relevant Approaches

Finally, it is worth mentioning that there has been a lot of work on *similarity measures* for computing a numeric score that estimates how similar two entities in a graph are [27, 58, 133]; this has applications, for instance, in discovering entities that are similar to a given one (i.e., those with the highest similarity score). Please

note that we are considering a very different problem since our focus is on *describing* similarities and differences in a declarative way. Declarative comparison, and in particular declarative similarities have considerable advantages over numeric similarity scores. Firstly, describing similarities declaratively, e.g., as queries, leads to overall explainability and interpretability of the system that uses these similarities. Declarative comparison is also very useful for catching errors as some of the similarities might be drawn mistakenly, and these errors are hard to debug and understand from numeric scores only. Lastly, declarative similarities and differences give a whole new perspective on the data both to the end users and to the data engineers, as they answer the question *why* two entities are (dis-)similar, which can lead to further data exploration insights.

One last area of research that is relevant for entity comparison and that we are going to cover in this thesis is *link discovery*, also known as entity linking or entity matching [48, 77]. Given two data sources S and T, e.g., DBpedia[4] and LinkedGeoData,[5] and a relation R, e.g., dbo:actor, link discovery is the problem of finding all pairs of entities $(s, t) \in S \times T$ such that $R(s, t)$ holds [84]; the resulting pairs are called links or mappings. When these mappings are computed, it is quite common to utilize some numeric similarity measures, either to pre-select candidate entity pairs or to actually compute the pairs [2, 44].

Entity comparison and link discovery are two orthogonal problems, yet there exists a certain synergy between them. Both problems investigate the relationships between entities in the data. Link discovery focuses on creating mappings across different data sources, while the default setting of the entity comparison framework involves a single data source. However, it is also straightforward to use two or more knowledge graphs in our framework: one would take the first entity from one KGs,

[4] `http://dbpedia.org`
[5] `http://linkedgeodata.org`

the other entity from the second KGs, and run entity comparison over the union of the two data sources. This way we would be able to compare entities across several KGs, with the aim to either investigate common and dissimilar patterns, or to find very similar entities that could refer to the same thing, which can be a step in the link discovery pipeline [96]. A use case of entity comparison complementing entity linkage is presented in Section 8.2.2.

Entity comparison lies at the intersection of several diverse areas of research and development. From the theoretic perspective, it draws from query reverse engineering, database theory and graph mining, while from the practical perspective it converges with information exploration, relationship explanation and discovery, and similarity measures. Finally, declarative entity comparison can contribute to other data processing tasks and applications, from explainable machine learning and natural language processing, to link discovery, to recommendations, since it offers a unique, comprehensible view of the data that other approaches miss. We will look at the advantages entity comparison can offer to other frameworks in Section 8.2.2.

Part II

Entity Comparison

Chapter 4

Entity Comparison Framework

In this chapter, we present our entity comparison framework and its main definitions, discuss properties of the main types of comparison queries and motivate our choice of modeling and query formalisms. As a running example, we are going to use a small RDF graph about the movie industry G_{mov} depicted in Figure 9, which is a subset of the YAGO graph [114]. In our example, we would like to compare Quentin Tarantino and Martin Scorsese. The graph G_{mov} contains relevant information about the two entities *Q_Tarantino* and *M_Scorsese* and is suitable for entity comparison.

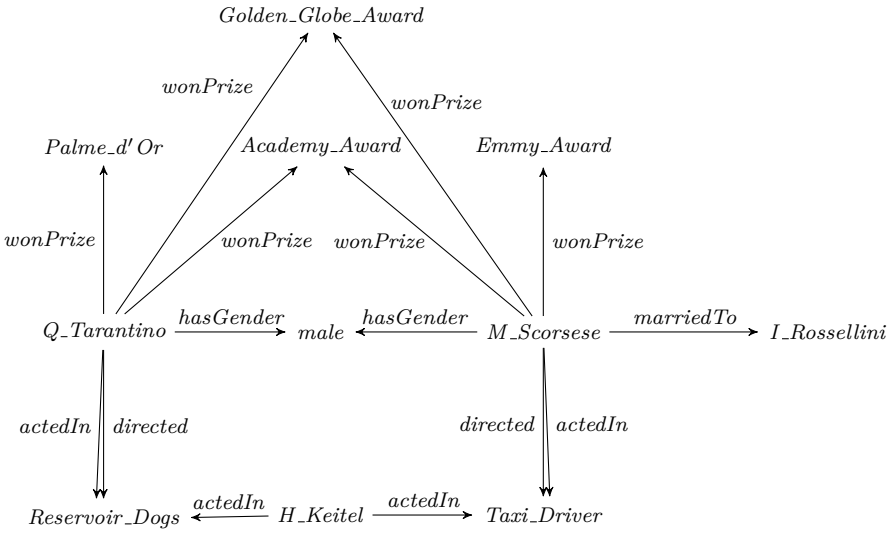


Figure 9: Example RDF graph G_{mov}

How can we formalise and automate entity comparison? We argue that in order to compare two entities, the user would like to find similarities and differences between these entities, that is, what information is common for both entities and what information discriminates one of them from the other. By inspecting G_{mov} we can observe, for instance, that Tarantino and Scorsese are similar in that both of them are male, they both won an Academy Award and a Golden Globe Award, and they both acted in some of their own movies. In turn, they are different in that Tarantino directed *Reservoir Dogs*, whereas Scorsese directed *Taxi Driver*; furthermore, unlike Scorsese, Tarantino also won the Palme d'Or at the Cannes Film Festival, while Scorsese won an Emmy award, to which Tarantino was only nominated.

How can we define and automatically identify such similarities and differences? There has been significant recent work in the literature on discovering relationships between entities in an RDF graph [26, 54, 69]. Existing approaches describe such relationships by means of explicit paths in the graph, which are then grouped and ranked. Using such an approach, we could view a similarity between entities as paths originating in those entities and converging into the same node; for instance, we could justify as a similarity the fact that both Tarantino and Scorsese are male by two paths leading to the node for male and starting from the nodes for Scorsese and Tarantino, respectively. In turn, we could justify a difference through the absence of such paths; for instance, the node for Emmy Award is reachable from the node for Scorsese but not from that for Tarantino. An important limitation of existing approaches, however, is that they cannot capture comparison at a *higher level of abstraction*; for instance, we cannot justify by means of explicit converging paths in a graph the fact that both Scorsese and Tarantino participated in a film as both actors and directors, where the specific names of those films are irrelevant.

In our framework we propose to capture similarities, differences and other types of comparisons using *queries* rather than explicit paths, where the presence of vari-

ables allows us to represent information at a higher level of abstraction. In our framework we are going to use SPARQL queries of specific form, since SPARQL is the standard language for querying RDF data and can be straightforwardly adapted for our purposes. While SPARQL does not fully accommodate navigational queries, it is an excellent query language to capture graph patterns of various degree of abstraction [9], which makes it a natural choice for an underlying formalism that compares entities using arbitrary, rather than path-based patterns. As we have already mentioned in Section 2.2.5, we use the fragment of SPARQL that allows for the AND and SELECT operators, and optionally for FILTER operator over built-in conditions of the form $?Y \lhd n$, where $?Y$ is a variable, n is an integer, and $\lhd$ is one of the inequality symbols $<$, $\leq$, $>$ or $\geq$. As has been discussed in Chapter 2, such queries without the FILTER operator correspond to conjunctive queries, while queries with the FILTER operator correspond to conjunctive queries with inequalities (in both cases the corresponding CQs contain only atoms of arity 3). The choice of this query language is motivated both by the compatibility of the SPARQL syntax with RDF data and by known advantages of CQs, e.g., their favourable computational properties and rich expressive power. Once we introduce the main definitions of the framework, we are going to illustrate the rationale behind using the two versions of query languages, i.e., with and without AFCs. In general, arithmetic comparisons are necessary for some types of the input data, for which they yield much more informative results, while excluding AFCs has computational advantages for some types of comparison queries.

Finally, in Section 2.2.5 we impose two important syntactic restrictions on the queries in our framework, and we are now ready to motivate them. First, we require all queries to be monadic, i.e., containing precisely one SELECT variable. SELECT variables determine the structure of the query answers, and since we are interested in queries matching (or not matching) individual entities in an RDF graph, only one

answer variable per query is required. Secondly, we require the queries, or rather their basic graph patterns, to be connected. Two triple patterns are considered connected in our framework if they share a term appearing as subject or object in both triple patterns. This is done because we are comparing entities using the information about them in an RDF graph, which essentially means the triples that lie in the immediate "neighbourhood" of each entity in the graph. Whenever we evaluate a comparison query, we want all parts of the query to match data that is relevant to the entities under consideration, not random parts of the graph. For example, when using G_{mov} we may be interested in persons that won both an Oscar and an Emmy (a connected query), whereas a pattern than matches directors who won an Oscar, and all movies of Harvey Keitel (not a connected query) is of little use.

The remainder of this chapter is structured as follows: in Section 4.1 we introduce all main definitions of the entity comparison framework, and in Section 4.2 we study the complexity of basic types of comparison queries.

4.1 Main Definitions

We start by formalising similarities. Given two entities in a graph, we view a similarity as a query having both entities as answers.

Definition 9. *A query Q is a* similarity query *(SQ) for entities a and b in an RDF graph G if $\{a, b\} \subseteq [Q]_G$.*

For instance, the following queries Q_1–Q_3 are similarity queries for Tarantino

and Scorsese in our example graph G_{mov}:

$$Q_1 = \text{SELECT } ?X \text{ WHERE } \{(?X, wonPrize, Academy\_Award)\};$$

$$Q_2 = \text{SELECT } ?X \text{ WHERE } \{(?X, hasGender, male),$$
$$(?X, wonPrize, Academy\_Award)\};$$

$$Q_3 = \text{SELECT } ?X \text{ WHERE } \{(?X, directed, ?Y), (?X, actedIn, ?Y),$$
$$(H\_Keitel, actedIn, ?Y)\}.$$

These similarity queries can be interpreted as follows: Q_1 says that both Scorsese and Tarantino received an Academy award, whereas Q_2 additionally states that they are both male; in turn, Q_3 states that both are directors who acted in their own movies, in which Harvey Keitel was also part of the cast.

We next formalise the notion of a difference. Intuitively, given two entities in an RDF graph, a difference is a query having one of the entities as answer, but not the other.

Definition 10. *A query Q is a* difference query *(DQ) for an entity a relative to an entity b in an RDF graph G if $a \in [Q]_G$ but $b \notin [Q]_G$.*

For instance, the following queries Q_4–Q_5 are difference queries for Scorsese relative to Tarantino in G_{mov}. Note that unlike SQ, the notion of a difference query is asymmetric: a DQ for a relative to b cannot be a DQ for b relative to a. An immediate consequence is that the sets of DQs for each entity relative to the other one are non-overlapping.

$$Q_4 = \text{SELECT } ?X \text{ WHERE } \{(?X, marriedTo, ?Y)\};$$

$$Q_5 = \text{SELECT } ?X \text{ WHERE } \{(?X, wonPrize, Academy\_Award),$$
$$(?X, wonPrize, Emmy\_Award)\}.$$

As we can see from the aforementioned examples, there may be multiple (even infinitely many) similarity and difference queries for a given pair of entities. Some of them are, however, more informative than others. In the case of similarity queries, it is natural to expect more specific queries to be more informative; for instance, it is natural to prefer our example query Q_2 over Q_1 since it better differentiates Tarantino and Scorsese from other directors, by ruling out those who won an Emmy but are female. In contrast, in the case of difference queries it is natural to favour more general queries over more specific ones; for instance, Q_4 is more informative than the following query Q_6 since it conveys the information that Scorsese is married, but Tarantino is not (or at least not known to be):

$$Q_6 = \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, \mathit{marriedTo}, \mathit{I\_Rossellini})\}.$$

There exists a formal way of measuring the relative specificity/generality of queries – using *query subsumption*. We now define the notions of most informative similarity and difference queries formally, using subsumption.

Definition 11. *Query Q is a* most specific similarity query *(MSSQ) for a and b in G if Q is a SQ for a and b in G, and there is no SQ Q' for a and b in G such that $Q' \subset Q$.*

As an example, consider the following query, which is subsumed by the SQs Q_1–Q_3; it can be checked that it is an MSSQ for Scorsese and Tarantino in G_{mov}:

$$\begin{aligned} Q_7 = \textsc{Select}\ ?X\ \textsc{Where}\ \{&(?X, \mathit{hasGender}, \mathit{male}),\\ &(?X, \mathit{wonPrize}, \mathit{Academy\_Award}), (?X, \mathit{wonPrize}, \mathit{Golden\_Globe\_Award}),\\ &(?X, \mathit{actedIn}, ?Y), (?X, \mathit{directed}, ?Y), (\mathit{H\_Keitel}, \mathit{actedIn}, ?Y), G_{mov}\}. \end{aligned}$$

Note that the query Q_7 contains in its basic graph pattern P_7 triple patterns that

correspond to all triples from G_{mov}. For brevity, we denote them by G_{mov}.

We will show in Chapter 5 that while there can exist multiple MSSQs for the given entities and a graph that are syntactically different, they all are unique modulo query equivalence. Intuitively, given two similarity queries Q and Q' for the same pair of entities, their conjunction is also a similarity query that is more specific than both of them. Indeed, a query $Q_8 =$ SELECT $?X$ WHERE $P_7 \cup \{(?X, actedIn, ?Z)\}$ is also an MSSQ but it is equivalent to Q_7.

The notion of MSSQ relies on query subsumption, which is a data-independent relationship between queries. It would clearly also make sense to look for similarity queries that are as discriminating for input entities a and b as possible over the specific input graph G at hand—that is, those similarity queries that return only a and b as answers when evaluated over G.

Definition 12. *A query Q is an* exact similarity query (ESQ) *for entities a and b in a graph G if $\{a, b\} = [Q]_G$.*

For instance, queries Q_1–Q_2 are not only SQs, but also ESQs in the example graph G_{mov} because Tarantino and Scorsese are the only film directors represented in the graph. However, as already discussed, these queries are not MSSQs because they are not minimal with respect to subsumption. Furthermore, if we were to consider the whole of YAGO instead of our example excerpt, queries Q_1–Q_2 would certainly no longer be ESQs since YAGO contains many other film directors, including those that received the Academy awards. So, MSSQs and ESQs are incomparable in general. However, as we will see in Chapter 5, there is still a connection between the two types of queries.

Finally, we can define the most informative difference queries using the notion of query subsumption, analogous to the most informative similarity queries.

Definition 13. *Query Q is a* most general difference query *(MGDQ) for a relative*

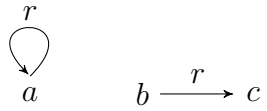


Figure 10: Example graph G

to b in G if Q is a DQ for a relative to b in G, and there is no DQ Q' for a relative to b in G such that $Q \subset Q'$.

Unlike MSSQs, most general difference queries do not have the uniqueness property: for two entities in an RDF graph there may exist multiple MGDQs. For example, both query Q_4 and the following query Q_9 are both MGDQs for Scorsese relative to Tarantino, incomparable with respect to subsumption:

$$Q_9(X) = \text{ SELECT } ?X \text{ WHERE } \{(?X, ?Y, \mathit{Emmy\_Award})\}.$$

Moreover, in some cases the number of MGDQs is even infinite, and the size of MGDQs is not bound by the size of the input RDF graph or any other parameter. Consider an example in Figure 10. There exist infinitely many MGDQs for a relative to b in G of the form SELECT $?X$ WHERE P, where the basic graph patterns are of the form:

$$\{(?X, ?R_1, a)\},$$
$$\{(?X, ?R_1, ?Y_1), (?Y_1, ?R_2, a)\},$$
$$\{(?X, ?R_1, ?Y_1), (?Y_1, ?R_2, ?Y_2), (?Y_2, ?R_3, a)\},$$
$$\{(?X, ?R_1, ?Y_1), (?Y_1, ?R_2, ?Y_2), (?Y_2, ?R_3, ?Y_3), (?Y_3, ?R_4, a)\},$$
$$\textit{etc.}$$

While there is no unique MGDQ for the given input, we cannot exhaustively list all MGDQs either. Moreover, as we will see in the next sections, difference queries

are computationally harder than similarity queries, and MGDQs are expected to be harder than MSSQs. Therefore, it is unlikely to get tractable results for MGDQs. In this thesis we are going to focus on one type of most informative comparison queries, namely on MSSQs.

So far we only used the examples without arithmetic filter conditions. However, many real-world datasets and resources contain rich numeric information: prices, sizes, technical parameters, etc. Consider a dataset in Table 2 with a typical side-by-side comparison of two models of iPhone; for brevity we present it in tabular form, but the dataset can straightforwardly be converted into the RDF format. It would be useful to be able to handle similarities and differences over all numeric values from the dataset.

	iPhone SE	iPhone 5s
Camera	12MP	8MP
Bluetooth	4.2	4.0
RAM	2Gb	1Gb
Battery	1642 mAh	1560 mAh

Table 2: Example comparison of two models of iPhone

Without arithmetic comparisons it is still possible to compute similarity and difference queries for iPhone SE and iPhone 5s, however such queries would be quite *uninformative*. For instance, an MSSQ would express the fact that for both smartphone models there is some information available about their cameras, bluetooth, RAM and battery:

$$Q_{sim} = \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, camera, ?Y), (?X, bluetooth, ?Z), (?X, ram, ?W), (?X, battery, ?V)\}.$$

Difference queries, on the other hand, state the exact parameter values for RAM,

battery, etc., thus simply reciting the information from the dataset in Table 2:

$$\begin{aligned} Q_{SE} = \ & \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, camera, 12MP), (?X, bluetooth, 4.2), \\ & (?X, ram, 2Gb), (?X, battery, 1642mAh)\}, \\ Q_{5s} = \ & \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, camera, 8MP), (?X, bluetooth, 4.0), \\ & (?X, ram, 1Gb), (?X, battery, 1560mAh)\}. \end{aligned}$$

Hence, without the AFCs the entity comparison framework generates rather uninformative similarity and difference queries which are arguably suboptimal in terms of readability, as compared to tabular representation. On the other hand, adding AFCs to the query language solves both the problem. A possible similarity query in the extended language would look as follows:

$$\begin{aligned} Q'_{sim} = \textsc{Select}\ ?X\ \textsc{Where}\ & \{(?X, camera, ?Y), (?X, bluetooth, ?Z), \\ & (?X, ram, ?W), (?X, battery, ?V)\} \\ \textsc{Filter}\ \{\ & ?Y \geq 8MP,\ ?Y \leq 12MP, \\ & ?Z \geq 4.0,\ ?Z \leq 4.2, \\ & ?W \geq 1Gb,\ ?W \leq 2Gb, \\ & ?V \geq 1560mAh,\ ?V \leq 1642mAh\ \}, \end{aligned}$$

and possible difference queries would be:

$$\begin{aligned} Q'_{SE} = \ & \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, camera, ?Y), (?X, ram, ?Z)\} \\ & \textsc{Filter}\ \{\ ?Y \geq 9MP,\ ?Z \geq 2Gb\}, \\ Q'_{5s} = \ & \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, camera, ?Y), (?X, ram, ?Z)\} \\ & \textsc{Filter}\ \{\ ?Y \leq 8MP,\ ?Z \leq 1Gb\}. \end{aligned}$$

We have now introduced the entity comparison framework and the five types of comparison queries: SQs, ESQs, DQs, MSSQs and MGDQs. We conceptually divide them into two groups: the *basic* comparison queries, consisting of SQs, ESQs, and DQs, and the *most informative* comparison queries, i.e., MSSQs and MGDQs. In the next section we are going to study the complexity of the basic comparison queries and establish some fundamental relationships between different types of queries.

4.2 Complexity of Basic Comparison Queries

When talking about the complexity of comparison queries, we are going to focus on two main decision problems EXISTS_$\mathcal{X}$ and VERIFY_$\mathcal{X}$, where $\mathcal{X}$ ranges over all basic and most informative comparison queries, i.e., SQs, DQs, ESQs, MSSQs and MGDQs. Each problem exists in two versions: for queries without arithmetic filter conditions,[1] and for queries with non-empty arithmetic filter conditions. While we always specify which version of the problem is being studied, it holds for both EXISTS_$\mathcal{X}$ and VERIFY_$\mathcal{X}$ that given a query type, the two versions of the problem yield the same results, with the exception of VERIFY_MSSQ (see Section 5.2).

EXISTS_$\mathcal{X}$

Input: Entities a and b in an RDF graph G

Question: Does there exist a query of type $\mathcal{X}$ for a, b and G?

VERIFY_$\mathcal{X}$

Input: Query Q, entities a and b in an RDF graph G

Question: Is Q a query of type $\mathcal{X}$ for a and b in G?

In this section we will study the complexity of the two problems for SQs, DQs and ESQs. We will see that adding or removing arithmetic comparisons will not affect

[1] or with empty ones

the complexity in case of basic queries. However, we will later observe that this is not the case for MSSQs.

4.2.1 Complexity of Similarity Queries

We start by studying similarity queries. It is not difficult to see that a similarity query exists, provided the input entities appear at the same position (i.e., subject, predicate, or object) in the input graph. For example, if both a and b appear as subjects in some triples in G, the query SELECT $?X$ WHERE $\{(X, Y, Z)\}$ is a similarity query for a and b in G. Checking that both entities appear in the same triple position in a graph can be done using a simple boolean circuit of depth 2. Therefore, the EXISTS_SQ problem is in AC^0 for both query languages, i.e., for queries with or without the arithmetic filter condition.

The complexity of the verification problem for SQs both with and without the arithmetic filter conditions can also be shown in a straightforward way. Verifying whether a given query Q is a similarity query for a and b in G is in fact a variant of the evaluation problem for conjunctive queries: given a data source G, a unary query Q and two answers a and b, decide whether $\{a, b\} \subseteq [Q]_G$. Query evaluation is known to be NP-complete for CQs and CQs with arithmetic comparisons (see Section 2.2.4). Therefore, VERIFY_SQ is NP-complete for similarity queries with and without the arithmetic filter condition.

4.2.2 Complexity of Difference Queries

We now turn our attention to difference queries, and we start by studying the EXISTS_DQ problem. In contrast to the case of similarity queries, it is unlikely that there exists a polynomial-time algorithm for computing a difference query. In fact, we show that the associated decision problem of checking whether a difference query

exists is coNP-complete. This result stems from a characterization of existence of a difference query in terms of (non-)existence of homomorphisms. A homomorphism h between two queries Q and Q' is a function from $\mathsf{term}(Q)$ to $\mathsf{term}(Q')$ such that IRIs and literals are mapped to themselves, answer variables of Q are mapped to answer variables of Q', and for every triple pattern $(t_s, t_p, t_o) \in Q$ there exists a triple pattern $(h(t_s), h(t_p), h(t_o)) \in Q'$ (see Section 2.3).

Complexity of Exists_DQ

We first look at Exists_DQ for queries without the arithmetic filter condition. We introduce a type of basic graph pattern that we call a *snapshot graph* of the RDF graph G and a distinguished entity e. Intuitively, for each triple in G there exists one or more corresponding triple patterns in the snapshot graph that preserve IRIs and literals, substitute blank nodes with variables, and either preserve or substitute the distinguished entity with a fresh variable. From such a snapshot graph we are going to construct two queries of particular shape, and we are going to show the connection between the existence of a difference query and the non-existence of a homomorphism between these queries. This connection will be the key step in proving the complexity of Exists_DQ. We start by formally defining the snapshot graph.

Definition 14. *A* snapshot graph *of G and e, denoted $\mathsf{Sn}_e(G)$, is largest basic graph pattern such that a triple pattern (t_s, t_p, t_o) is in $\mathsf{Sn}_e(G)$ if a triple (s, p, o) is in G, and for each term t_c, with $c \in \{s, p, o\}$, it holds that*

$$t_c = \begin{cases} c, & \text{if } c \in \mathbf{U} \cup \mathbf{L} \setminus \{e\}, \\ ?X_c, & \text{if } c \in \mathbf{B}, \\ ?X \text{ or } e, & \text{if } c = e. \end{cases}$$

Using Definition 14 we construct two snapshot queries $Q_a =$ Select $?X$ Where

$\mathsf{Sn}_a(G|_a)$ and $Q_b = \textsc{Select}\ ?X\ \textsc{Where}\ \mathsf{Sn}_b(G)$ for two entities a and b in G, with $G|_a$ being the connected component of G containing a. The following result holds for the snapshot queries.

Lemma 1. *A difference query for a relative to b in G exists if and only if there is no homomorphism from Q_a to Q_b.*

Proof. ($\Longleftarrow$). The following properties hold for Q_a: it is connected and $a \in [Q_a]_G$ (witnessed by the satisfying valuation $\nu = \{?X \to a\} \cup \{?X_{\_:c} \to \_:c \mid \_:c \text{ is a blank node in } G|_a\}$). We now prove that $b \notin [Q_a]_G$. Suppose $b \in [Q_a]_G$, then there exists a satisfying valuation μ over $\mathsf{var}(Q_a)$ s.t. $\mu(Q_a) \subseteq G$ and $\mu(?X) = b$. But then there exists a homomorphism h from Q_a to Q_b:

$$h(t) = \begin{cases} t, & \text{if } t \in \mathbf{U} \cup \mathbf{L} \cup \{?X\}, \\ \mu(t), & \text{if } t \in \mathsf{var}(Q_a) \setminus \{?X\} \text{ and } \mu(t) \notin \mathbf{B}, \\ ?X_{\mu(t)}, & \text{if } t \in \mathsf{var}(Q_a) \setminus \{?X\} \text{ and } \mu(t) \in \mathbf{B}, \end{cases}$$

which is a contradiction. Hence, $a \in [Q_a]_G$, while $b \notin [Q_a]_G$. Thus, Q_a is a difference query for a relative to b in G.

($\Longrightarrow$). Let Q be a difference query for a relative to b in G. By definition, $a \in [Q]_G$, while $b \notin [Q]_G$. It implies that there is a satisfying valuation μ over $\mathsf{var}(Q)$ for Q in G s. t. $\mu(Q) \subseteq G$ and $\mu(?X) = a$. Then there exists a homomorphism h from Q to Q_a:

$$h(t) = \begin{cases} t, & \text{if } t \in \mathbf{U} \cup \mathbf{L} \cup \{?X\}, \\ \mu(t), & \text{if } t \in \mathsf{var}(Q) \setminus \{?X\} \text{ and } \mu(t) \notin \mathbf{B}, \\ ?X_{\mu(t)}, & \text{if } t \in \mathsf{var}(Q) \setminus \{?X\} \text{ and } \mu(t) \in \mathbf{B}. \end{cases}$$

Now for the sake of contradiction, suppose there is a homomorphism h' from Q_a to

Q_b. Taking the composition of two homomorphisms, we get a new homomorphism $f = h' \circ h$ from Q to Q_b (since both h and h' map IRIs and literals to themselves, map the answer variable $?X$ to itself, and map variables to other terms in a structure-preserving manner). Let $f|_{\mathsf{var}(Q)}$ be the restriction of f to the variables in Q. By construction of the snapshot query Q_b it holds that $b \in [Q_b]_G$, witnessed by the satisfying valuation $\mu' = \{?X \rightarrow b\} \cup \{?X_{\_:c} \rightarrow \_:c \mid \_:c \text{ is a blank node in } G\}$. But then we can construct a valuation $\mu'' = \mu' \circ f|_{\mathsf{var}(Q)}$ of a query Q for which it holds that:

- $\mu''(?X) = \mu'(f(?X)) = \mu'(?X) = b$, and
- for every triple pattern T from Q: $\mu''(T) = \mu'(f(T)) = \mu'(T') = t$ for some triple pattern T' from Q_b and some triple $t \in G$.

Thus, μ'' is a satisfying valuation for Q in G mapping the answer variable to b, a contradiction to the fact that Q is a difference query for a relative to b in G. □

Since homomorphism checking is a well-known NP-complete problem [55], the following result follows.

Theorem 1. *The problem* EXISTS_DQ *of checking whether a difference query without the arithmetic filter condition exists for a relative to b in G is* coNP*-complete.*

Proof. It is known that checking existence of a homomorphism is in NP [55]. Together with Lemma 1 it implies that existence of a difference query can be checked in coNP. We show the lower bound by reduction of the homomorphism problem for graphs to the complement of our problem.

Construction:

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed unlabelled graphs which we can assume to be disjoint and connected. We treat elements in V_1 and V_2 as blank nodes, and we introduce two fresh blank nodes a and b. We then construct an

RDF graph G over the set of blank nodes $V_1 \cup V_2 \cup \{a, b\}$ and IRIs $\{e, e'\}$, where $\{e, e'\} \cap V_i = \emptyset, i = 1, 2$, as the following set:

$$G = \{(u, e, v) \mid (u, v) \in E_1 \cup E_2\} \cup \{(a, e', u) \mid u \in V_1\} \cup \{(b, e', v) \mid v \in V_2\}.$$

Let Q_a and Q_b be snapshot queries for a and b in G. It is easy to show that there exists a homomorphism from G_1 to G_2 if and only if there is a homomorphism from Q_a to Q_b. Lemma 1 implies that this is equivalent to non-existence of a difference query for a relative to b in G.

Correctness:

First notice that when we construct G, for every vertex $v \in V_1 \cup V_2$ we create a unique blank node $\_:v$ in G, and we add two more blank nodes a and b. Furthermore, when we construct the snapshots of G and subsequently the queries Q_a and Q_b, for every node in G we create a unique variable. Therefore, we can define two pairs of bijective functions f_a, f_a^{-1} and f_b, f_b^{-1} that unambiguously map original vertices into query variables and back.Finally, let there be two mappings $h : V_1 \to V_2$ and $g : \mathsf{term}(Q_a) \to \mathsf{term}(Q_b)$ such that:

$$g(t) = \begin{cases} ?X, & \text{if } t = ?X, \\ t & \text{if } t \in \{e, e'\}, \\ f_b(h(f_a^{-1}(t))), & \text{if } t \in \mathsf{var}(Q_a) \setminus \{?X\}. \end{cases}$$

All mappings are illustrated in Figure 11.

We claim that h is a homomorphism from G_1 to G_2 if and only if g is a homomorphism from Q_a to Q_b. Since g maps the answer variable to itself and maps IRIs to themselves, it remains to prove that $(s, p, o) \in Q_a$ implies $(g(s), g(p), g(o)) \in Q_b$ if and only if $(u, v) \in E_1$ implies $(h(u), h(v)) \in E_2$.

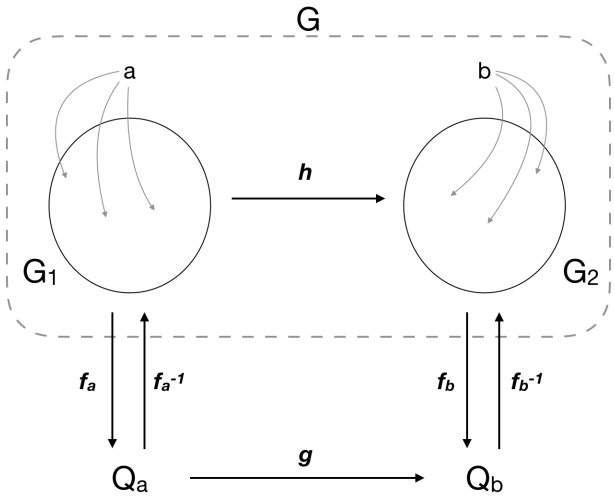


Figure 11: Schematic illustration of the mappings h, g, f_a, f_a^{-1}, f_b and f_b^{-1}

Suppose h is a homomorphism, and $(u, v) \in E_1$ implies $(u', v') \in E_2$ where $u' = h(u)$ and $v' = h(v)$. This holds if and only if $(u, e, v), (u', e, v') \in G$ with u, v, u' and v' blank nodes in G, by constructing of G. This in turn holds if and only if $(?X, e', ?X_u), (?X, e', ?X_v), (?X_u, e, ?X_v) \in Q_a$ and $(?X, e', ?X_{u'}), (?X, e', ?X_{v'}), (?X_{u'}, e, ?X_{v'}) \in Q_b$, by construction of the snapshot graphs of G and a or b. Using the bijective functions, the triple patterns can be rewritten as follows:

$$
\begin{aligned}
(?X_{u'}, e, ?X_{v'}) &= (f_b(u'), e, f_b(v')) = (f_b(h(u)), e, f_b(h(v))) = \\
&\quad (f_b(h(f_a^{-1}(?X_u))), e, f_b(h(f_a^{-1}(?X_v)))), \\
(?X, e', ?X_{u'}) &= (?X, e', f_b(u)) = (?X, e', f_b(h(u))) = (?X, e', f_b(h(f_a^{-1}(?X_u)))),
\end{aligned}
$$

and analogously for $(?X, e', ?X_{v'})$. This is equivalent to saying that for any triple pattern in Q_a of the form $(?X_u, e, ?X_v)$ (resp. $(?X, e, ?X_u)$) there exists a triple pattern in Q_b of the form $(g(?X_u), g(e), g(?X_v))$ (resp. $(g(X), g(e), g(X_w))$), since g is the identity function on $?X$, e and e'. This makes g a homomorphism from Q_a to Q_b.

□

So, EXISTS_DQ is NP-complete for difference queries without the arithmetic filter conditions (AFC). In what follows we will show that adding AFCs to the query language does not change the complexity of the existence problems, since a DQ without the AFC exists for a given input if and only if a DQ with the AFC exists for this input. One direction of this statement holds trivially since any query without AFC can be represented as a query with an empty AFC. It remains to prove the other direction of the statement.

Lemma 2. *If a DQ containing AFC exists for entity a relative to entity b in an RDF graph G, then a DQ without AFC exists for a relative to b in G.*

Proof. Let Q be a difference query for a relative to b in G that contains an arithmetic filter condition: $Q = \text{SELECT } ?X \text{ WHERE } P \text{ FILTER } C$. If $C = \emptyset$, the lemma holds trivially. Assume C is not empty: $C = \{(?Y_1 \lhd n_1), \ldots, (?Y_k \lhd n_k)\}$ for $k \geq 1$. Since Q is a DQ, it holds that $b \notin [Q]_G$, and therefore there is no valuation according to which b is the answer to Q: $\nexists \mu : \mathsf{var}(Q) \rightarrow \mathbf{U} \cup \mathbf{L} \cup \mathbf{B}$ such that $\mu(?X) = b,\ \mu(P) \subseteq G$, and $\mu(?Y_i) \lhd n_i$ for $1 < i \leq k$. However, since $a \in [Q]_G$ there must exist a valuation ν such that $\nu(?X) = a,\ \nu(P) \subseteq G$, and $\nu(?Y_i) \lhd n_i$ for $1 < i \leq k$. Let $C' \subseteq C$ be a set of all arithmetic comparisons from C containing some variable $?Y_i$ from C: $C' = \{(?Y_i \lhd n_i^1), \ldots, (?Y_i \lhd n_i^m)\}$ with $m \leq k$. C' is consistent with respect to G since there must exist an element $e \in \mathbf{U} \cup \mathbf{L} \cup \mathbf{B}$ such that $\nu(?Y_i) = e$ and $e \lhd n_i^j$ for $1 < j \leq m$. Let Q' be a new query of the form $Q' = \text{SELECT } ?X \text{ WHERE } P' \text{ FILTER } \{\}$, with the basic graph pattern P' being computed from P by replacing each occurrence of $?Y_i$ with the element e: $P' = P|_{?Y_i \rightarrow e}$. It holds that $a \in [Q]_G$, since there exists a valuation $\nu' = \nu|_{\mathsf{var}(Q')}$ such that $\nu'(P') \subseteq G$ and $\nu'(?X) = a$. It remains to prove that Q' is a DQ for a relative to b in G, i.e., that $b \notin [Q]_G$. Suppose $b \in [Q]_G$. Then there must exist a valuation μ' that witnesses b as an the answer to Q over G: $\mu'(?X) = b$ and $\mu'(P') \subseteq G$. But then $\mu' \cup \nu|_{?Y_i}$ would be a satisfying valuation for Q such that

$b \in [Q]_G$, which is a contradiction. Thus, Q' is a AFC-free difference query for a relative to b in G. □

The following two results immediately follow.

Corollary 1. *A DQ Q containing AFC exists for entity a relative to entity b in an RDF graph G if and only if a DQ Q' without AFC exists for a relative to b in G.*

Theorem 2. *The problem* Exists_DQ *of checking whether a difference query with the arithmetic filter condition exists for a relative to b in G is* coNP*-complete.*

Proof. The upper bound follows from the equi-existence of DQs in the two query languages. The lower bound can be shown by a trivial reduction of Exists_DQ for queries without AFC. □

Complexity of Verify_DQ

We now move to the Verify_DQ problem of verifying whether a given Q is a difference query for the given input, and we start with the upper bound.

Lemma 3. *The problem* Verify_DQ *of checking whether Q is a difference query for a relative to b in G is in* DP*.*

Proof. The membership in DP follows easily from the complexity of the existence problem for similarity queries. Let L_1 and L_2 be two languages such that $L_1 = \{(G, a, b, Q) \mid a \in [Q]_G\}$ and $L_2 = \{(G, a, b, Q) \mid b \notin [Q]_G\}$. Then the language $L = L_1 \cap L_2$ is exactly the language of difference queries for given inputs. Since L_1 is in NP (evaluation problem) and L_2 is in coNP (its complement) for queries with and without the arithmetic filter condition, L is in DP. □

For the proof of the lower bound we need to introduce two auxiliary decision problems over graph homomorphisms. We use the first problem as a stepping stone

to prove the complexity of the second problem, and we use the reduction of the second problem to show the lower bound of VERIFY_DQ. We start by formally defining the two intermediate decision problems.

Definition 15. *Given four undirected unlabelled graphs H_1, H_2, H_3 and H_4, let* HOM-NOHOM *be the problem of deciding whether H_1 is homomorphic to H_2, and H_3 is not homomorphic to H_4.*
Given three undirected labelled *graphs D, D_1 and D_2 with pairwise disjoint sets of node labels, let* LABEL-HOM-NOHOM *be the problem of deciding whether D is homomorphic to D_1, but not to D_2.*

The two aforementioned problems are modifications of the classical *graph homomorphism* problem (or the *H-colouring* problem) HOM: given two undirected, unlabelled graphs G and G', decide whether G is homomorphic to G'. Hell et al. [55] showed that HOM is NP-complete.

Homomorphism for unlabelled graphs is conventionally defined as a mapping h from a graph $G = (V, E)$ to a graph $G' = (V', E')$ such that for every edge $\{u, w\} \in E$ it holds that $\{h(u), h(w)\} \in E'$ (see Section 2.3). Homomorphism for labelled graphs is defined as a pair of functions (h_v, h_l), one for graph vertices and the other for vertex and edge labels: given two graphs $G = (V, E)$ and $G' = (V', E')$ and their labelling functions l and l', there is a homomorphism (h_v, h_l) from G to G', if for each $\{u, w\} \in E$ there exists an edge $\{h_v(u), h_v(w)\} \in E'$ such that $h_l(l(u)) = l'(h_v(u))$, $h_l(l(w)) = l'(h_v(w))$, and $h_l(l(\{u, w\})) = l(\{h_v(u), h_v(w)\})$.

We first prove hardness results for the two auxiliary problems.

Lemma 4. *The* HOM-NOHOM *problem is* DP*-hard.*

Proof. In order to prove hardness, we need to show that there is a polynomial-time reduction R of an arbitrary problem L in DP to HOM-NOHOM: $L \leq_p$ HOM-NOHOM.

Since $L \in$ DP, it holds that there exist two languages L_1, L_2 s.t. $L = L_1 \cap L_2$, $L_1 \in$ NP and $L_2 \in$ coNP. Since Hom is NP-complete, there exists a reduction $r_1 : L_1 \leq_p$ Hom such that $x \in L_1 \leftrightarrow (r_1(x) = \langle H_1, H_2 \rangle) \in$ Hom for some undirected unlabelled H_1, H_2. Similarly, Nohom is the complement of Hom, hence it is coNP-complete, hence there exists a reduction $r_2 : L_2 \leq_p$ Nohom such that $x \in L_2 \leftrightarrow (r_2(x) = \langle H_3, H_4 \rangle) \in$ Nohom for some H_3, H_4. Hence, $L \leq_p$ Hom-Nohom by the reduction $r \coloneqq (r_1, r_2)$. □

Lemma 5. *The* Label-Hom-Nohom *problem is* DP*-hard.*

Proof. The proof is by reduction of Hom-Nohom. Given $(H_1, H_2, H_3, H_4) \in$ Hom-Nohom, construct an input to Label-Hom-Nohom in polynomial time as follows:

- label each node in $H_1, \ldots, H_4$ with a unique node label;
- label each edge in H_1 and H_2 with an edge label R;
- label each edge in H_3 and H_4 with a edge label S;
- let D be the union of labelled graphs H_1 and H_3;
- let D_1 be the union of labelled graphs H_2 and H_3;
- let D_2 be the union of labelled graphs H_1 and H_4.

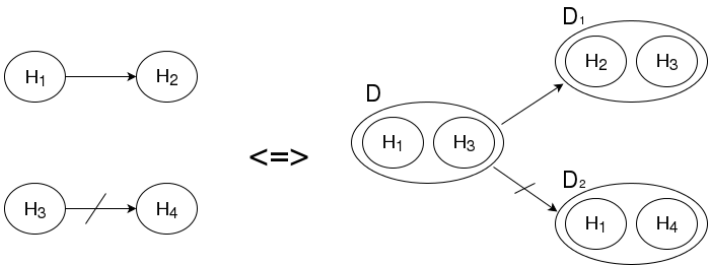


Figure 12: Schematic illustration of the reduction of Hom-Nohom

Parts of D and D_1 corresponding to H_3 are trivially homomorphic. Therefore, D is homomorphic to D_1 iff the part of D corresponding to H_1 is homomorphic to the part of D_1 corresponding to H_2, i.e., iff there is a homomorphism between

the original graphs H_1 and H_2. Parts of D and D_2 corresponding to H_1 are also trivially homomorphic. Then D is not homomorphic to D_2 iff the part of D corresponding to H_3 is not homomorphic to the part of D_2 corresponding to H_4, i.e., iff the original H_3 is not homomorphic to H_4. Note that other mappings are not possible, i.e., the part of D corresponding to H_1 cannot be homomorphic to the part of D_1 corresponding to H_3 since the two parts have different edge labelling. Therefore $(H_1, H_2, H_3, H_4) \in$ HOM-NOHOM iff $(D, D_1, D_2) \in$ LABEL-HOM-NOHOM, and hence LABEL-HOM-NOHOM is DP-hard. □

We are now ready to prove hardness of the verification problem for difference queries without the arithmetic filter conditions.

Lemma 6. *The problem* VERIFY_DQ *of checking whether Q is a difference query without AFC for a relative to b in G is* DP*-hard.*

Proof. The proof is by reduction of LABEL-HOM-NOHOM. The idea of the reduction is, given the input graphs D, D_1, D_2 and their labelling functions l, l_1, l_2, to first introduce two new nodes with labels a and b, then to convert D_1 and D_2 together into one RDF graph, then to covert D into a query and finally to check whether it is a difference query for a relative to b. In particular, we do the following steps.

1. Introduce a fresh node labelled a to D_1, connecting it to each node in the graph, except to itself, with an edge labelled r. Repeat the same step with D_2 and a fresh node labelled b.

2. Take the union of D_1 and D_2 and convert it into an RDF graph G by turning each edge $\{v, u\} \in E(D_i)$ into two triples $(l_i(v), l_i(\{v, u\}), l_i(u))$ and $(l_i(u), l_i(\{u, v\}), l_i(v))$, for $i \in \{1, 2\}$.

3. Introduce a fresh node labelled X to D, connecting it to each node in the graph, except to itself, with an edge r.

4. Generate a query $Q = \textsc{Select}\ ?X\ \textsc{Where}\ P\ \textsc{Filter}\ \{\}$ from the graph D by adding 2 triple patterns $(?X_{l(v)}, ?X_{l(\{v,u\})}, ?X_{l(u)})$ and $(?X_{l(u)}, ?X_{l(\{v,u\})}, ?X_{l(v)})$ to P for each edge $\{v, u\} \in E(D)$, where $?X_i$'s are variables. Node labelled X is translated into the variable $?X$. Note that the resulting query is connected, since the node labelled X is connected to every other node in the original graph D.

Then $(D, D_1, D_2) \in$ Label-Hom-Nohom iff Q is a difference query for a relative to b in G. Hence, the difference query checking problem is DP-hard. □

Theorem 3. *The problem* Verify_DQ *of checking whether Q is a difference query for a relative to b in G is* DP*-complete.*

Proof. Completeness for difference queries without arithmetic filter conditions follows directly from Lemmas 3 and 6. Lower bound of the verification problem for DQs with AFCs can be shown by trivial reduction of that for DQs without AFCs. Together with Lemma 3 this makes Verify_DQ DP-complete for all types of difference queries. □

To sum up, problems involving difference queries are inherently more difficult than those involving similarity queries: verifying a similarity query is an NP-complete task, while verifying a difference query is a DP-complete task, and checking whether a query exists can be trivially done in AC^0 for similarity queries, while the problem is coNP-complete for difference queries.

4.2.3 Complexity of Exact Similarity Queries

Lastly, we are going to study the complexity of exact similarity queries, again considering queries with and without the arithmetic filter conditions, and we start with the Exists_ESQ problem. Unlike difference queries, ESQs with and without AFCs

do not have the equi-existence property. Consider a simple social RDF graph G_{soc} consisting of three triples. It is not possible to construct an ESQ for *Anna* and *Bob* without using arithmetic comparisons; as a result, both Q_{10} are similarity queries, but only Q_{11} is an ESQ for the two entities in G_{soc}.

$$G_{soc} = \{(Anna, age, 26), \\ (Bob, age, 27), \\ (Carla, age, 32)\}$$

$$Q_{10} = \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, age, ?Y)\}$$

$$Q_{11} = \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, age, ?Y)\}\ \textsc{Filter}\ \{(?Y \leq 27)\}$$

The problem of deciding whether an ESQ exists for the given input is coNP-complete for queries with and without AFCs. In this chapter we are going to include the hardness result, leaving the membership proof to Chapter 5. Once we study the complexity of MSSQs, the proof of the upper bound of Exists_ESQ will follow naturally.

Theorem 4. *The problem* Exists_ESQ *of checking whether an exact similarity query exists for a and b in G is* coNP*-hard.*

Proof. The lower bound is obtained by reduction of the coNP-complete problem of checking whether there exists a difference comparison-free query Q for an entity a relative to an entity b in an RDF graph G. The reduction goes as follows: given G, a, and b as an instance to the difference existence problem, consider the graph $G' = G \cup \{(a, r, c), (b, r, c)\}$ for fresh entities r and c not occurring in G. Then there exists a difference query for a relative to b in G if and only if there exists an ESQ for a and a in G', i.e., an ESQ for the entity a and its copy in G'.

($\Longrightarrow$) Let $Q = \textsc{Select}\ ?X\ \textsc{Where}\ P\ \textsc{Filter}\ C$ be a difference query for a relative to b in G. Then there must exist a valuation μ over $\mathsf{var}(P)$ such that $\mu(?X) = a$, $\mu(P) \subseteq G$, and all arithmetic comparisons from C are satisfied by μ. Let $Q' = \textsc{Select}\ ?X\ \textsc{Where}\ P'\ \textsc{Filter}\ C$ be a query obtained from Q by adding one more triple pattern to P: $P' = P \cup \{(?X, r, c)\}$; note that $\mathsf{var}(P) = \mathsf{var}(P')$. Then

the valuation μ also satisfies Q' over G', since $\mu(P \cup \{(?X, r, c)\}) \subseteq G \cup \{(a, r, c)\}$. Moreover, any other valuation that satisfies Q' over G' must map $?X$ either to a or to b, by construction of Q' and G. But we know that no valuation over $\mathsf{var}(P)$ satisfying Q over G maps $?X$ to b, hence no valuation satisfying Q' over G' will either. Thus, a is the only answer to Q' over G', or alternatively Q' is an ESQ for a and a in G'.

($\Longleftarrow$) Let Q' = SELECT $?X$ WHERE P' FILTER C' be an ESQ for a and a in G'. Then there must exist a valuation μ satisfying Q' over G' with $\mu(?X) = a$, and no valuation satisfying Q' over G' that maps $?X$ to b. Let $V = \{?V_1 \ldots ?V_k\}$ be all the variables in $\mathsf{var}(Q')$ such that $\mu(?V_i) = r$ or $\mu(?V_i) = c$, for $1 \leq i \leq k$. Finally, let $P'' \subseteq P'$ be a set of all triple patterns in P' containing variables from V. Then $\mu(P'') \subseteq \{(a, r, c), (b, r, c)\}$ and $\mu(P' \setminus P'') \subseteq G' \setminus \{(a, r, c), (b, r, c)\}$. Since $G' \setminus \{(a, r, c), (b, r, c)\} = G$, then $\mu(P' \setminus P'') \subseteq G$. Note that $P' \setminus P''$ is not empty, otherwise Q' would have b as one of its answers over G'. Let Q = SELECT $?X$ WHERE P FILTER C be a new query obtained from Q' by removing from P' all triple patterns in P'' ($P = P' \setminus P''$) and by removing from C' all arithmetic comparisons containing variables in V. The valuation μ satisfies Q over G, hence a is an answer to Q over G. We claim that Q is a difference query for a relative to b in G. Suppose this is not the case, i.e., b is also an answer to Q over G. Then there must be a valuation μ' satisfying Q over G such that $\mu'(?X) = b$. Then let μ'' be another valuation obtained from μ and μ': $\mu'' = \mu' \cup \mu \mid_V$. By construction $\mu''(?X) = b$ and $\mu''(P') \subseteq G'$, which makes Q' not an ESQ. This is a contradiction, hence Q is a difference query for a relative to b in G.

The above proof work equally fine for both queries with empty and non-empty arithmetic filter conditions C and C', therefore the EXISTS_ESQ problem is coNP-hard both for comparison-free ESQs and ESQs with comparisons. □

We conclude this chapter with the complexity of the ESQ verification problem.

The lower bound is once again proven by reduction of the corresponding verification problem for DQs. In fact, difference and exact similarity queries are similar in nature: both types of queries imply that there exist certain valuations but not the other ones.

Theorem 5. *The problem* VERIFY_ESQ *of checking whether Q is an exact similarity query for a and b in G is* DP*-complete.*

Proof. To establish the upper bound, we use the query evaluation problem, which is in NP for queries with or without AFCs. Given two entities a and b, an RDF graph G and a query Q, consider a non-deterministic algorithm that checks in NP that both input entities are answers to the query on the input graph, and checks in coNP that there are no other answer. The algorithm then decides whether Q is an ESQ for a and b in G, hence the problem is in DP.

The lower bound is obtained by reduction of the verification problem for comparison-free difference queries, which is now proven to be DP-hard. The reduction is similar to the one presented in the proof of Theorem 4: let G, a, b and Q = SELECT $?X$ WHERE P FILTER $\{\}$ be an instance of the difference verification problem. Moreover, let r be a fresh entity and $\_{:}c$ and $\_{:}d$ be fresh blank nodes in G, and $?Y$ be a fresh variable in Q.

- In polynomial time check whether there exists a triple in G with the entity a as a subject. If yes, construct an RDF graph $G' = G \cup \{(a, r, \_{:}c), (b, r, \_{:}d)\}$ and a query Q' = SELECT $?X$ WHERE $P \cup \{(?X, r, ?Y)\}$ FILTER $\{\}$.
- Otherwise check whether there exists a triple in G with the entity a as an object. If yes, construct an RDF graph $G' = G \cup \{(\_{:}c, r, a), (\_{:}d, r, b)\}$ and a query Q' = SELECT $?X$ WHERE $P \cup \{(?Y, r, ?X)\}$ FILTER $\{\}$.
- Otherwise return the answer *no*, since there does not exist a query for which a is an answer in G.

Then Q is a difference query for a relative to b in G if and only if Q' is an ESQ for a and a in G'. Observe that this reduction establishes the lower bound for queries with and without the arithmetic filter condition.

($\Longrightarrow$) Without loss of generality assume that there exists a triple in G with the subject entity a. Let $Q = \textsc{Select}\ ?X\ \textsc{Where}\ P'\ \textsc{Filter}\ \{\}$ be a difference query for a relative to b in G, with $P' = P \cup \{(?X, r, ?Y)\}$. Then there must exist a valuation μ satisfying Q over G such that $\mu(?X) = a$ and $\mu(P) \subseteq G$. But then $\mu(P') = \mu(P \cup \{(?X, r, ?Y)\}) \subseteq G \cup \{(a, r, \_{:}c)\} \subseteq G'$. Therefore a is an answer to Q' over G'. Moreover, we know that there does not exist a valuation μ' satisfying Q over G such that $\mu'(?X) = b$ (premise). Therefore, if a valuation μ' maps $?X$ to b, then $\mu'(P) \nsubseteq G$ (since the filter condition is empty). Suppose there exists a satisfying valuation μ'' for Q' over G' such that $\mu''(?X) = b$. Then $\mu''(P') = \mu''(P \cup \{(?X, r, ?Y)\}) = \mu''(P) \cup \mu''(\{(?X, r, ?Y)\}) = \mu'(P) \cup \{(b, r, \_{:}d)\} \nsubseteq G \cup \{(b, r, \_{:}d)\} \subseteq G'$. Thus $\mu''(P') \nsubseteq G'$, and b cannot be an answer to Q' over G'. But since $\{(?X, r, ?Y)\}$ enforces any valuation to map $?X$ either to a or to b in G', a is the only answer to Q', and Q' is an ESQ for a and a in G'.

($\Longleftarrow$) Without loss of generality assume that there exists a triple (a, e_1, e_2) in G with the entity a as the subject and with an entity e_1 and an entity or a blank node e_2. Let $Q' = \textsc{Select}\ ?X\ \textsc{Where}\ P'\ \textsc{Filter}\ \{\}$ be an ESQ for a and a in G'. Then there must exist a valuation μ satisfying Q' over G' with $\mu(?X) = a$ and $\mu(P') \subseteq G'$.

Claim. *Let G, G', Q, Q', a and b be as defined above. If Q' is an ESQ for a and a in G, then there must exist a valuation μ satisfying Q' over G' such that $\mu(?X) = a$ and $\mu(P') \subseteq G'$, but also $\mu(P) \subseteq G$, with $P' = P \cup \{(?X, r, ?Y)\}$.*

Proof. Suppose this is not the case. Then for any valuation ν there must exist a triple pattern $t \in P$ such that $\nu(t) \subseteq G' \setminus G = \{(a, r, \_{:}c), (b, r, \_{:}d)\}$. Since r is an

	SQ	ESQ	DQ
EXISTS_$\mathcal{X}$	in AC^0	coNP-c.	coNP-c.
VERIFY_$\mathcal{X}$	NP-c.	DP-c.	DP-c.

Table 3: Complexity of VERIFY_$\mathcal{X}$ and EXISTS_$\mathcal{X}$ problems for the three basic types of queries in the framework, where c. denotes complete

entity not appearing in G, the triple pattern t can be of one of the following forms: $(a, ?R, ?V), (b, ?R, ?V), (?X, ?R, ?V)$ or $(?V', ?R, ?V)$. But then it can be mapped to another triple (a, e_1, e_2) from G, which is a contradiction. □

Let ν the valuation that satisfies Q' over G', maps $?X$ to a and $\nu(P) \subseteq G$. Then ν satisfies Q over G, and hence a is an answer to Q over G. Since Q' has only one answer a over G', there does not exist a valuation μ' such that it satisfies Q' over G' and maps $?X$ to b. It implies that if $\mu'(?X) = b$, then $\mu'(P') \nsubseteq G'$ (the filter condition of Q' is empty). Suppose that there exists a valuation μ'' satisfying Q over G: $\mu''(?X) = b$ and $\mu''(P) \subseteq G$. Then $\mu''(P) = \mu''(P' \setminus \{(?X, r, ?Y)\}) =$ (since r is not in G) $= \mu''(P') \setminus \mu''(\{(?X, r, ?Y)\}) = \mu''(P') \setminus \{(a, r, \_{:}c), (b, r, \_{:}d)\} \nsubseteq G' \setminus \{(a, r, \_{:}c), (b, r, \_{:}d)\} = G$. Hence, no valuation satisfying Q over G and mapping $?X$ to b exists, and b is not an answer to Q over G. Therefore, Q is a difference query for a relative to b in G. □

4.2.4 Overview of the Results

In this chapter we introduced the main notions of the entity comparison framework and illustrated them using a running example from the movie domain. We studied the complexity of two fundamental types of decision problems – existence and verification – for the three *basic* types of queries: similarity, exact similarity and difference queries. In case of basic comparison queries the complexity results are the same for queries with and without the arithmetic filter conditions, and they are summarised in Table 3. Moreover, we showed that multiple similarity and difference

queries for the same two entities in a graph can be partially ordered using the subsumption relation, and that such ordering produces two new types of queries that we consider to be the most informative ones. In the following two chapters we will focus on most specific similarity queries, and we will study their properties, their complexity (which this time will be different for the two query languages), and we will propose ways of efficiently computing them.

Chapter 5

Most Specific Similarity Queries

In this chapter we are going to study the first type of the most informative comparison queries — most specific similarity queries, or MSSQs. MSSQs are similarity queries for the two given entities in an RDF graph that are minimal with respect to subsumption. They represent the most comprehensive graph pattern that corresponds to both input entities, and thus finding MSSQs is one of the key problems of our entity comparison framework.

5.1 Computing MSSQs

In this section, we present an algorithm that computes an MSSQ, if one exists, and reports failure otherwise. The algorithm has an optional part that is responsible for dealing with arithmetic comparisons; without this part, the algorithm produces a comparison-free MSSQ, while adding this part allows the algorithm to compute an MSSQ with the arithmetic filter condition. We also show how a simple modification of this algorithm can be used for computing an ESQ, which we use to show the upper bound of the Exists_ESQ problem discussed in Chapter 4.

Our algorithm relies on the notion of the (tensor) product graph, which is com-

monly exploited in Graph Theory and in Databases (under the name of *direct product* [115]). Given two RDF graphs G_1 and G_2, the product $G_1 \times G_2$ is a graph whose vertex set is the cartesian product of the vertices of G_1 and G_2, and where two vertices in the product graph are connected by an edge if and only if their component elements are also related by an edge in the original graph. We next adapt the standard notion of product to RDF graphs.

Definition 16. *Given triples* $\tau_1 = (s_1, p_1, o_1)$ *and* $\tau_2 = (s_2, p_2, o_2)$*, the* product *of* τ_1 *and* τ_2*, denoted* $\tau_1 \times \tau_2$*, is the tuple of the form*

$$(\langle s_1, s_2\rangle, \langle p_1, p_2\rangle, \langle o_1, o_2\rangle).$$

The product graph $G_1 \times G_2$ *of two RDF graphs* G_1 *and* G_2 *is the set*

$$\{\tau_1 \times \tau_2 \mid \tau_1 \in G_1, \tau_2 \in G_2\}.$$

For example, the self-product $G_{mov} \times G_{mov}$ of our example graph G_{mov} introduced on page 44 contains tuples such as the following:

$$(\langle Q\_Tarantino, M\_Scorsese\rangle, \langle wonPrize, wonPrize\rangle, \langle Palme\_d'Or, Emmy\_Award\rangle),$$

which is the product of triples $(Q\_Tarantino, wonPrize, Palme\_d'Or)$ and $(M\_Scorsese, wonPrize, Emmy\_Award)$. Intuitively, given entities a, b and an RDF graph G, the connected subgraph of the product $G \times G$ of G with itself represents the "largest common pattern" in the neighbourhoods of a and b. Next we are going to use the notion of the product graph in the computation of an MSSQ.

Algorithm Compute_MSSQ (given in Algorithm 1) accepts as input a graph G, and entities a and b in G. First it computes the product graph $G \times G$ (line 1) and checks whether the node $\langle a, b\rangle$ occurs in some tuple in $G \times G$ (line 2). If it does

Algorithm 1: COMPUTE_MSSQ

Input: graph G, entities a and b in G
Output: MSSQ for a and b in G, or *fail*

1 compute $G \times G$;
2 **if** $\langle a, b \rangle$ *is not in a tuple in* $G \times G$ **then**
3 **return** *fail*;
4 compute the connected component $G_\times$ of $\langle a, b \rangle$ in $G \times G$;
5 **let** P be the pattern obtained from $G_\times$ by replacing each pair $\langle c_1, c_2 \rangle$ with either variable $?X_{c_1,c_2}$, if $c_1 \neq c_2$ or $c_1 \in \mathbf{B}$, or with c_1 otherwise;
6 **if** $a = b$ **then**
7 add to P all triple patterns obtained from triple patterns already in P by replacing at least one occurrence of a with $?X_{a,a}$;
8 create an empty set C;
9 *(optional)* **let** C be $\{(?X_{n_1,n_2} \leq \max(n_1, n_2)),\ (?X_{n_1,n_2} \geq \min(n_1, n_2)) \mid ?X_{n_1,n_2} \in \mathsf{var}(P);\ n_1, n_2 \in \mathbb{Z}\}$;
10 **return** SELECT $?X_{a,b}$ WHERE P FILTER C.

not, then the algorithm determines that a similarity query, and hence an MSSQ, for a and b in G does not exist, and reports failure (line 3). In contrast, if $\langle a, b \rangle$ occurs in the product graph $G \times G$, then the algorithm computes the connected component $G_\times$ of $\langle a, b \rangle$ in the product graph (line 4). Note that $G_\times$ is computed in a graph-like manner, i.e., two tuples are deemed connected if they share the first or the third "vertex" component. Then the algorithm constructs the output query based on it (line 10). Specifically, it computes the pattern P in the query by replacing each element of a product triple in $G \times G$ with either a constant or a variable uniquely associated with the element (lines 5–7). Finally, in case we are computing an MSSQ with arithmetic comparisons, the algorithm also performs an additional step of computing the filter condition C by adding suitable inequalities for those variables representing pairs of numeric literals in the product graph (optional line 9). Alternatively, if we are computing a comparison-free MSSQ, the value of C is set to an empty set (line 8).

Let us demonstrate how the COMPUTE_MSSQ algorithm works on a simple RDF graph G_{twi} from Figure 13 representing who follows who on a social network.

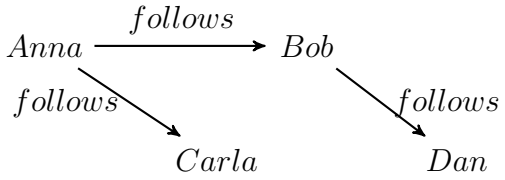


Figure 13: Example graph G_{twi}

Suppose COMPUTE_MSSQ takes as input G_{twi} and two entities *Anna* and *Bob*. First the algorithm computes the product graph of G_{twi} with itself which is

$$\{(\langle Anna, Bob\rangle, \langle follows, follows\rangle, \langle Carla, Dan\rangle),$$
$$(\langle Anna, Bob\rangle, \langle follows, follows\rangle, \langle Bob, Dan\rangle),$$
$$(\langle Anna, Anna\rangle, \langle follows, follows\rangle, \langle Bob, Carla\rangle),$$
$$(\langle Anna, Anna\rangle, \langle follows, follows\rangle, \langle Carla, Bob\rangle),$$
$$(\langle Bob, Anna\rangle, \langle follows, follows\rangle, \langle Dan, Carla\rangle),$$
$$(\langle Bob, Anna\rangle, \langle follows, follows\rangle, \langle Dan, Bob\rangle)\}.$$

The tuple $\langle Anna, Bob\rangle$ is clearly present in the product graph, hence the algorithm proceeds to the next step and computes the connected component of the product graph, consisting of two tuples $(\langle Anna, Bob\rangle, \langle follows, follows\rangle, \langle Carla, Dan\rangle)$ and $(\langle Anna, Bob\rangle, \langle follows, follows\rangle, \langle Bob, Dan\rangle)$, and then translates it into a graph pattern $P = \{(?X_{Anna,Bob}, follows, ?Y_{Carla,Dan}), (?X_{Anna,Bob}, follows, ?Y_{Bob,Dan})\}$. Since there are no numeric values in the input graph, the arithmetic filter condition C is assigned an empty set. Finally the output query SELECT $?X_{Anna,Bob}$ WHERE P FILTER C is constructed.

Clearly, the algorithm works in polynomial time. In particular, its most computationally expensive step is getting the product graph $G \times G$, which is quadratic in the size of G, hence computing an MSSQ both with and without the arithmetic filter conditions can be done in polinomial time. Correctness is established by the

following theorem.

Theorem 6. COMPUTE_MSSQ *is a quadratic time procedure that returns an MSSQ for its input entities and graph, if it exists, or fail otherwise.*

Proof. First, as already mentioned, all steps can be done in quadratic time. Second, recall that a similarity query of entities a and b in a graph G exists if and only if both a and b appear in the same position in triples in G, which happens precisely when $\langle a, b\rangle$ appears in a tuple in $G \times G$ by construction. So, if COMPUTE_MSSQ returns *fail* in line 3 then there is no MSSQ for a and b.

Third, let us prove that the output query Q is a similarity query for a and b in G such that any similarity query Q' for a, b and G is homomorphically embeddable into Q. We first show that $\{a, b\} \subseteq [Q]_G$. Define two valuations over $\mathsf{var}(Q)$, ν_1 and ν_2, as follows: for every variable $?X_{\langle c,c'\rangle}$ in Q it holds that $\nu_1(?X_{\langle c,c'\rangle}) = c$ and $\nu_2(?X_{\langle c,c'\rangle}) = c'$. We now show that G satisfies Q under both ν_1 and ν_2. Let $(?X_{\langle s_1,s_2\rangle}, ?X_{\langle p_1,p_2\rangle}, ?X_{\langle o_1,o_2\rangle})$ be in Q, then it follows by definition of Q that $(\langle s_1, s_2\rangle, \langle p_1, p_2\rangle, \langle o_1, o_2\rangle) \in G \times G$. By construction of $G \times G$ we know that both (s_1, p_1, o_1) and $(s_2, p_2, o_2) \in G$. We then obtain that by definition of ν_1 and ν_2: $(\nu_i(?X_{\langle s_1,s_2\rangle}), \nu_i(?X_{\langle p_1,p_2\rangle}), \nu_i(?X_{\langle o_1,o_2\rangle}))) \in G$, for $i = 1, 2$. Hence, ν_1 and ν_2 are satisfied for Q in G. We have $\nu_1(?X_{\langle a,b\rangle}) = a$ and $\nu_2(?X_{\langle a,b\rangle}) = b$. Therefore, $\{a, b\} \subseteq [Q]_G$.

Let Q' be an arbitrary similarity query for a and b. There are two satisfying valuations ν_1 and ν_2 over $\mathsf{var}(Q')$ for Q' in G that map the answer variable $?X$ of Q' to a and b respectively. We define $\nu(?Y) = \langle \nu_1(?Y), \nu_2(?Y)\rangle$ for $?Y$ a variable and $\nu(e) = \langle e, e\rangle$ for e an entity. Since Q' is connected and $\nu(?X) = \langle a, b\rangle$, the image of Q' under ν is a connected subgraph in $G \times G$, and thus is contained in $G_\times$. Since $G_\times$ and Q are isomorphic, ν can be considered as a homomorphism from Q' to Q. It is a well-known result that such a homomorphism guarantees that $Q \subseteq Q'$ [25]. Therefore, the algorithm returns a comparison-free MSSQ (with the filter condition

C being empty), if the optional step in line 9 is omitted. Finally, if the step in line 9 is to be performed, the algorithm additionally computes the filter condition C which contains arithmetic comparisons for all possible numeric variables in the pattern P, and these comparisons are constrained in the tightest way possible by the integer values. Hence, COMPUTE_MSSQ returns an MSSQ for the languages with and without the arithmetic comparisons. □

As we have discussed in Chapter 4, there can be multiple syntactically different MSSQs for a pair of entities in a graph. We next show, however, that MSSQs are unique modulo equivalence, i.e., if Q and Q' are MSSQs for a and b in RDF graph G, then $Q \equiv Q'$. Intuitively, this is the case because the conjunction of similarity queries is also a similarity query.

Proposition 1. *MSSQs are unique up to equivalence.*

Proof. Let a and b be entities in a graph G. Consider two arbitrary MSSQs $Q_i =$ SELECT $?X$ WHERE P_i FILTER C_i, $i \in \{1, 2\}$, for a and b in G. Then the query

$$Q \quad = \quad \text{SELECT } ?X \quad \text{WHERE } P_1 \cup P_2 \quad \text{FILTER } C_1 \cup C_2$$

is a similarity query, and it is more specific than both Q_1 and Q_2. Note that $P_1 \cup P_2$ is connected because P_1 and P_2 are both connected, and both mention $?X$. Therefore, Q_1, Q_2, and Q are all equivalent MSSQs. □

The uniqueness result for MSSQs helps us establish an important link between two types of comparison queries, namely MSSQs and ESQs.

Proposition 2. *If Q is an MSSQ for entities a and b in a graph G such that $[Q]_G \neq \{a, b\}$, then no ESQ for a and b in G exists.*

Proof. Let Q' be an ESQ for a and b in G—that is, Q' is a similarity query with

Algorithm 2: COMPUTE_ESQ

Input: graph G, entities a and b in G
Output: ESQ for a and b in G, or *fail*

```
1 if COMPUTE_MSSQ returns fail on a, b and G then
2     return fail;
3 let Q be an MSSQ for a and b in G computed by COMPUTE_MSSQ;
4 compute the set [Q]_G of all answers to Q over G;
5 if [Q]_G = {a, b} then
6     return Q;
7 else
8     return fail.
```

$[Q']_G = \{a, b\}$. So, Q' is a similarity query that is not subsumed by the MSSQ Q, which contradicts Proposition 1. □

Finally, we next observe that the COMPUTE_MSSQ algorithm can be easily modified to compute an ESQ, if one exists. Indeed, let algorithm COMPUTE_ESQ (given in Algorithm 2) be the same as COMPUTE_MSSQ except that it additionally evaluates the constructed query at the end, and returns the query only if the result is precisely a, b, and *fail* otherwise.

Theorem 7. COMPUTE_ESQ *is a procedure that returns an ESQ for its input entities and graph if it exists, or fail otherwise.*

Proof. If the algorithm returns a query Q, then Q is an ESQ for the input entities a and b in the input graph G since this is explicitly checked in the last step. Assume now that the algorithm returns *fail*; we argue that no ESQ exists. If it returns *fail* in line 3, then by the correctness of Algorithm 1 we can conclude that no similarity query (and hence no ESQ) exists for a and b in G. In turn, if the algorithm returns *fail* in the last step, we know that the constructed query Q is not an ESQ. Furthermore, by the correctness of Algorithm 1, we know that Q is an MSSQ for a and b in G, so, by Proposition 2, no ESQ exists. □

Using the COMPUTE_ESQ algorithm we can finally establish the upper bound of the EXISTS_SQ problem of deciding whether an ESQ exists for the given input, that was missing from Chapter 4. Note that the evaluation step in COMPUTE_ESQ does not work in (deterministic) polynomial time for queries neither with, no without arithmetic comparisons.

Proposition 3. *The problem* EXISTS_ESQ *of checking whether an exact similarity query exists for a and b in G is in* coNP.

Proof. The result follows from the COMPUTE_ESQ algorithm: first it computes, in polynomial time, a candidate query Q, and then universally guesses an entity different from a and b verifying that it is not an answer to Q. The last can be done in coNP by usual query evaluation algorithms. □

From Proposition 3 and Theorem 4 the next result directly follows.

Corollary 2. *The problem* EXISTS_ESQ *of checking whether an exact similarity query exists for a and b in G is* coNP*-complete.*

5.2 Complexity of MSSQs

In this section we are going to study the complexity of existence and verification problems (introduced in Chapter 4) for most specific similarity queries. The correctness of the COMPUTE_MSSQ algorithm established by Theorem 6 implies that an MSSQ is always guaranteed to exist whenever a similarity query exists for the given input. Furthermore, checking whether a similarity query exists for the given input can be efficiently done in AC^0. From this follows that the problem EXISTS_MSSQ of checking whether an MSSQ with or without the arithmetic filter condition exists for two input entities a and b in an RDF graph G is in AC^0.

We now move to the verification problem for MSSQs. Unlike in the case of basic comparison queries, i.e., SQs, DQs or ESQs, the complexity of verifying that a query is an MSSQ is different for queries with and without the arithmetic filter conditions. The difference in complexity stems from the internal containment check that needs to be done in order to ensure that the input query is indeed an MSSQ for the given input, i.e., that it is equivalent to the MSSQ computed be the COMPUTE_MSSQ algorithm. The containment problem is NP-complete for CQs [25], while it is Π_2^P-complete for CQs with arithmetic constraints [63, 121], and these theoretic results underpin the complexity of the VERIFY_MSSQ problem.

Theorem 8. *The problem* VERIFY_MSSQ *of checking whether a query Q without arithmetic comparisons is an MSSQ for a and b in G is* NP*-complete.*

Proof. The upper bound follows from the fact that the MSSQ for a given G, a, b is unique up to equivalence (see Proposition 1), hence it is sufficient to compute an MSSQ Q_{mssq} using the COMPUTE_MSSQ algorithm (in PTIME) and then to check whether the input Q is equivalent to Q_{mssq} (in NP).

The lower bound is proven by reduction of the VERIFY_SQ problem, which is NP-complete. Given the input G, a, b and Q = SELECT $?X$ WHERE P, let Q_{mssq} = SELECT $?X$ WHERE P_{mssq} be the MSSQ for a and b in G computed by the COMPUTE_MSSQ algorithm in polynomial time, omitting step 9, i.e., not adding any arithmetic comparisons to the query. Without loss of generality assume that all variables in Q_{mssq} excluding $?X$ are renamed apart from the variables of Q: $\mathsf{var}(Q) \cap \mathsf{var}(Q_{mssq}) = \{?X\}$. Furthermore, let Q' be the conjunction of Q and Q_{mssq}: Q' = SELECT $?X$ WHERE $P \cup P_{mssq}$. Then Q is a similarity query for a and b in G if and only if Q' is the MSSQ for a and b in G. The correctness of the reduction is proven as follows.

($\Longrightarrow$). Let Q and Q_{mssq} be an SQ and an MSSQ for a and b in G, respectively. It holds by construction of Q' that there exists a homomorphism $h : Q_{mssq} \to Q'$,

where h is the identity function. Then by the classical Homomorphism Theorem [25] it holds that $Q' \subseteq Q_{mssq}$. On the other hand, $Q_{mssq} \subseteq Q'$ holds since P_{mssq} is part of the basic graph pattern of Q'. But then $Q' \equiv Q_{mssq}$, which makes Q' an MSSQ for a and b in G.

($\Longleftarrow$). If Q' is an MSSQ for a and b in G, then it is trivially a similarity query for a and b in G. Therefore there exist two valuations ν_1 and ν_2 over $\mathsf{var}(Q')$ such that $\nu_1(?X) = a$, $\nu_2(?X) = b$, and $\nu_i(P \cup P_{mssq}) \subseteq G$ for $i \in \{1, 2\}$. Since without loss of generality we can assume that all variables in P and P_{mssq} are renamed apart, except for $?X$, each ν_i can be represented as a union of two valuations v'_i and v''_i ($v_i = v'_i \cup v''_i$ for $i \in \{1, 2\}$) such that:

$$\nu'_1 \cap \nu''_1 = \{?X \to a\},$$
$$\nu'_2 \cap \nu''_2 = \{?X \to b\},$$
$$\nu'_i \text{ is a valuation of } \mathsf{var}(P), \text{ and}$$
$$\nu''_i \text{ is a valuation of } \mathsf{var}(P_{mssq}), \text{ for } i \in \{1, 2\}.$$

And since valuations ν'_1 and ν'_2 satisfy the input RDF graph, i.e., $\nu'_i(P) \subseteq G$ for $i \in \{1, 2\}$, a and b are answers to Q, which makes Q a similarity query for a and b in G.

□

Verifying an MSSQ that contains arithmetic comparisons is a computationally harder problem that lies one level higher in the polynomial hierarchy.

Theorem 9. *The problem* VERIFY_MSSQ *of checking whether a query Q with arithmetic comparisons is an MSSQ for a and b in G is Π_2^P-complete.*

Proof. For the upper bound, we construct an algorithm for checking whether a query Q is an MSSQ for entities a and b in a graph G that proceeds in two steps. First, we apply Algorithm 1 to obtain (in polynomial time) an MSSQ Q_{mssq} for a and b

in G. By Proposition 1, Q is an MSSQ if and only if it is equivalent to Q_{mssq}. In the second step we check equivalence of Q and Q_{mssq}. Since the two queries can be seen as CQACs, the check is feasible in Π_2^P [63].

For the lower bound proof, we use the following characterisation for verifying if a query is an MSSQ. A query Q is an MSSQ for a and b in G if and only if $a, b \in [Q]_G$ and $Q \subseteq Q_{mssq}$. Note that the size of Q_{mssq} is polynomial in the size of G. The proof is by reduction of the $\forall\exists$3SAT problem, the prototypical Π_2^P-hard problem. While it initially seemed reasonable to us to use the reduction of the query containment problem for CQACs, since both query containment and VERIFY_MSSQ are "query-based" problems, the fundamental $\forall\exists$3SAT problem evetually proved to be more suitable for the task at hand.

Let $\varphi = \forall x_1, \ldots, x_m \exists y_1, \ldots y_k(\gamma_1 \wedge \ldots \wedge \gamma_n)$ be a quantified boolean formula where each γ_i is a 3-clause. Using the above characterisation it is enough to construct an RDF graph G and a query Q both of polynomial size in the size of φ such that φ is a valid formula if and only if $a, b \in [Q]_G$ and $Q \subseteq Q_{mssq}$. Construction of an RDF graph G and query Q is split into two parts: encoding of the universally quantified variables x_i and encoding of the satisfaction of each clause γ_j. We start with the first part.

For each $i, 1 \leq i \leq m$, graph G contains the subgraphs G_i^a and G_i^b defined as follows:

$$G_i^a = \{(a, e_i, l_i^a), (l_i^a, d_i, 4), (l_i^a, s_i, r_i^a), (a, e_i, r_i^a), (r_i^a, d_i, 5), (r_i^a, t_i, x_i^a)\},$$
$$G_i^b = \{(b, e_i, l_i^b), (l_i^b, d_i, -1), (l_i^b, s_i, r_i^b), (b, e_i, r_i^b), (r_i^b, d_i, 10), (r_i^b, t_i, x_i^b)\}.$$

Note that Q_{mssq} is then equivalent to the query SELECT $?Ans$ WHERE P_{mssq}

FILTER F_{mssq} such that P_{mssq} contains a subpattern

$$\begin{aligned} P_i^{mssq} = &(?Ans, e_i, ?L_i), (?L_i, d_i, ?U_i), \\ &(?L_i, s_i, ?R_i), (?Ans, e_i, ?R_i), (?R_i, d_i, ?V_i), (?R_i, t_i, ?X_i), \end{aligned}$$

and F_{mssq} contains comparisons

$$F_i^{mssq} = (?U_i \geq -1), (?U_i \leq 4), (?V_i \geq 5), (?V_i \leq 10).$$

Next, for the query $Q =$ SELECT $?Ans$ WHERE P FILTER F, P contains the pattern

$$\begin{aligned} P_i = &(?Ans, e_i, ?L_i), (?L_i, d_i, ?U_i), (?L_i, s_i, ?M_i), \\ &(?Ans, e_i, ?M_i), (?M_i, d_i, ?W_i), (?M_i, t_i, 1), (?M_i, s_i, ?R_i), \\ &(?Ans, e_i, ?R_i), (?R_i, d_i, ?V_i), (?R_i, t_i, 0), \end{aligned}$$

and F contains the comparisons

$$(?U_i \geq 2), (?U_i \leq 4), (?V_i \geq 7), (?V_i \leq 9), (?W_i \geq 3), (?W_i \leq 6).$$

The defined parts of graph G, Q_{mssq} and Q are better illustrated in Figure 14. The above construction is enough to encode the universally quantified variables x_i by connecting an arbitrary variable assignment $V : \{x_1, \ldots, x_m\} \to \{0, 1\}$ with a satisfying valuation ν of Q in G as follows:

- $V(x_i) = 0$ iff $\nu(?W_i) \leq 4$,
- $V(x_i) = 1$ iff $\nu(?W_i) \geq 5$.

This is well defined since for any RDF graph G' satisfying Q, a satisfying valuation of Q_{mssq} in G' maps the variable $?X_i$ in Q_{mssq} to exactly either 0 (when it maps $?U_i$

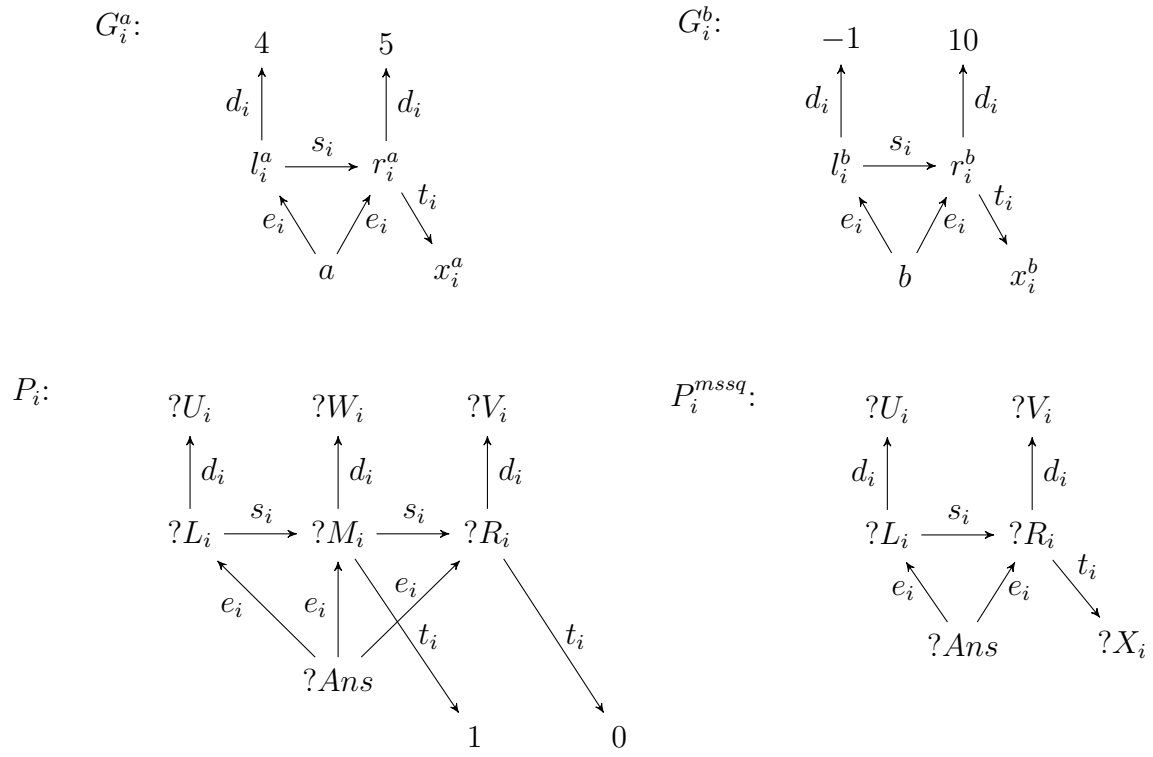


Figure 14: Encoding of the universally quantified variables

to the image of $?W_i$ in G') or 1 (when it maps $?V_i$ to the image of $?W_i$ in G').

Note that in order for a and b to be answers of Q over G, we need to add to G new sets of triples, called G'_a and G'_b, that are isomorphic to all P_i. In turn this requires patterns P and comparisons F of Q to contain new patterns and comparisons to ensure that in any model of Q, the subquery in Q_{mssq} obtained from the product of triples in G'_a and G'_b is satisfied as well. Crucially, such additions still satisfy the requirement that for any model G' of Q with valuation μ, and satisfying valuation ν of Q_{mssq} in G' either $\nu(?U_i) = \mu(?W_i)$ or $\nu(?V_i) = \mu(?W_i)$ holds.

Let us move to the encoding of satisfaction of each clause $\gamma_j, 1 \leq j \leq n$. We demonstrate it on the example clause $\gamma_j = (x_2 \vee \neg x_4 \vee y_3)$. We require the graph G

to contain the following subgraphs

$$G^a_{\gamma_j} = \{(a, p_j, g^a_j), (g^a_j, q_j, z^a_j), (z^a_j, first, x^a_2), (z^a_j, second, x^a_4), (z^a_j, third, y^a_3)\},$$
$$G^b_{\gamma_j} = \{(b, p_j, g^b_j), (g^b_j, q_j, z^b_j), (z^b_j, first, x^b_2), (z^b_j, second, x^b_4), (z^a_j, third, y^b_3)\}.$$

Because of these subgraphs, the pattern P_{mssq} of Q_{mssq} contains the subpattern

$$P_{\gamma_j} = (?Ans, p_j, ?G_j), (?G_j, q_j, ?Z_j), (?Z_j, first, ?X_2),$$
$$(?Z_j, second, ?X_4), (?Z_j, third, ?Y_3).$$

Next, we require the pattern P of Q to contain the pattern

$$P' = \{(?Ans, p_j, ?G_j), (?G_j, q_j, ?Z^1_j), \ldots, (?G_j, q_j, ?Z^7_j)\} \cup P'',$$

where P'' encodes all the combinations of variable assignments of x_2, x_4 and y_3 that make γ_j true:

$$P'' = \{(?Z^1_j, first, 0), (?Z^1_j, second, 0), (?Z^1_j, third, 0)\} \cup$$
$$\ldots$$
$$\cup \{(?Z^7_j, first, 1), (?Z^7_j, second, 1), (?Z^7_j, third, 1)\}.$$

The constructed patterns are depicted in Figure 15. Intuitively, the image of a satisfying valuation of variables $?X_2, ?X_4$ and $?Y_3$ in a model of Q encodes a satisfying variable assignment for x_2, x_4 and y_3 for γ_j. In order for a and b to be answers of Q over G, we additionally require G to contain the canonical RDF graph for pattern P'. Note that adding this set of triples will also imply adding the new patterns in P_{mssq} which however can be homomorphically mapped onto P, the patterns of Q.

We are ready to prove correctness of the reduction. By construction we have

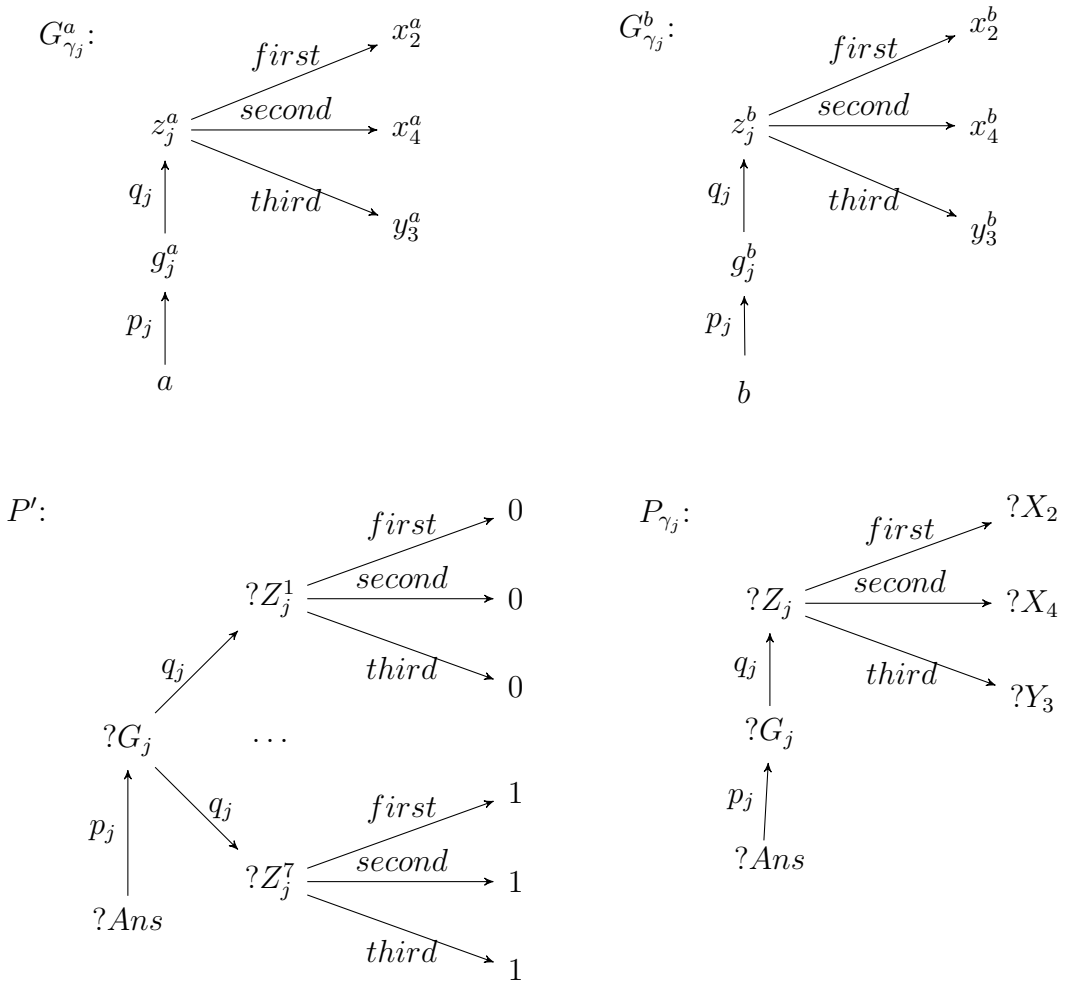


Figure 15: Encoding of individual clauses

that $a, b \in [Q]_G$. Thus, it is enough to show that φ is valid if and only if $Q \subseteq Q_{mssq}$.

($\Longrightarrow$). Let φ be valid. That is, for every variable valuation $V_x : \{x_1, \ldots, x_m\} \to \{0, 1\}$ there is a variable valuation $V_y : \{y_1, \ldots, y_k\} \to \{0, 1\}$ such that φ is assigned to true under $V_x \cup V_y$. Let G' be an RDF graph such that $c \in [Q]_{G'}$. We show that $c \in [Q_{mssq}]_{G'}$ by constructing a valuation μ of variables of Q_{mssq} in G' with $\mu(?Ans) = c$. Let ν be a valuation of variables of Q in G' with $\nu(?Ans) = c$. The valuation μ can be defined depending on the value of $\nu(?W_i)$. In particular, the patterns P_i^{mssq} are preserved under μ, $\mu(?Ans) = \nu(?Ans) = c$ and we have that $\mu(?X_i) = 0$ if $\nu(?W_i) \leq 4$ and $\mu(?X_i) = 1$ if $\nu(?W_i) \geq 5$. This defines a

variable assignment V_x with $V_x(x_i) = \mu(?X_i)$. Hence, there is a variable assignment $V_y : \{y_1, \ldots, y_k\} \to \{0, 1\}$ such that φ is true under $V_x \cup V_y$. We thus can extend μ by assuming $\mu(?Y_i) = V_y(y_i)$. Moreover, since every γ_j is true under $V_x \cup V_y$, every pattern P_{γ_j} can be mapped to the corresponding image of a pattern from P''. Thus, μ witnesses the fact $c \in [Q_{mssq}]_{G'}$.

($\Longleftarrow$). Let $Q \subseteq Q_{mssq}$, and $V_x : \{x_1, \ldots, x_m\} \to \{0, 1\}$ be an arbitrary variable assignment. We define the RDF graph G' to be canonical for Q such that the canonical image of $?W_i$ is defined according to V_x: $\nu(?W_i) = 4$ if $V_x(x_i) = 0$ and $\nu(?W_i) = 5$ if $V_x(x_i) = 1$. Since $c = \nu(?Ans) \in [Q]_{G'}$, it holds that $c \in [Q_{mssq}]_{G'}$. Let μ be a satisfying valuation of variables of Q_{mssq} in G'. Note that μ agrees with V_x, i.e., $\mu(?X_i) = V_x(x_i)$. Furthermore, μ defines a variable assignment $V_y : \{y_1, \ldots, y_k\} \to \{0, 1\}$ as follows: $V_y(y_i) = \mu(?Y_i)$. It can be seen from the construction that every pattern P_{γ_j} must map onto the image of P' in G' which implies that γ_j is assigned to true under $V_x \cup V_y$. Thus it shows that φ is valid. □

We have now studied most specific similarity queries that are the most informative SQs for two given entities in an RDF graph. While an MSSQ always exists as long as a similarity query exists for the given input, computing an MSSQ takes quadratic time in the size of the input graph, which is impractical for real-life RDF graphs. Therefore, in the next two chapters we will consider a possible approximation of MSSQs that is feasible to compute yet retains a high degree of specificity.

Part III

Practical Considerations

Chapter 6

Acyclic Similarity Queries

In the previous chapter we have introduced the COMPUTE_MSSQ algorithm that computes a most specific similarity query for the given input. However, despite running in polynomial time, the algorithm is impractical. Indeed, real-life RDF graphs tend to contain millions of triples, and the algorithm explicitly computes the product graph $G \times G$, which is of quadratic size in the size of G. For example, the size of the commonsense YAGO knowledge base is over 120 million facts (triples) [75, 114]; an RDF dump of the DBLP bibliographic record base contains over 1.2 million entries [73]; ConceptRDF, an RDF presentation of the ConceptNet knowledge base, consists of over 21 million records [83,111]. The COMPUTE_MSSQ algorithm is unsuitable for all of these knowledge bases because of their size. Not only is the algorithm unable to process these input data sources in reasonable time (see Chapter 7), but also the queries it outputs are proportionate in size to the size of the product graph generated by the algorithm, i.e., they are of size quadratic in the size of the input graph. Large MSSQs are incomprehensible, and therefore are not useful for practical entity comparison.

Hence, it makes sense to design an *approximation algorithm*, which, on the one hand, constructs reasonably specific similarity queries and, on the other hand, can

scale to large input graphs. In this chapter we devise one such algorithm. We propose a practical algorithm that computes an *acyclic* similarity query for two entities in a graph (if one exists). Although the query computed by the algorithm is not guaranteed to be an MSSQ, we will verify empirically in Chapter 7 that it is a reasonable approximation in practice. Finally, we discuss practical aspects of the algorithm, its restrictions on the input entities as well as its computational advantages on large, real-world datasets.

6.1 Acyclic Queries

We start by defining acyclic comparison queries. In Section 2.2.4 we have outlined different types of acyclic conjunctive queries. For our queries we are going to use the definition of acyclicity that is similar to Berge-acyclicity, but that uses a different underlying incidence structure, namely the *incidence multigraph*. Let $Q = \textsc{Select}\ ?X\ \textsc{Where}\ P\ \textsc{Filter}\ C$ be a query, with P the set of all triple patterns, and $\mathsf{var}(P) = \mathsf{var}(Q)$ the set of all variables occurring in P.

Definition 17 (Incidence multigraph)**.** *An* incidence multigraph *of a basic graph pattern P, denoted $Inc(P)$, is an undirected bipartite multigraph, in which $\mathsf{var}(P) \cup P$ is the set of vertices, and for every triple pattern p in P, for every occurrence of a variable $?V$ in p, there exists a distinct edge $\{?V, p\}$.*

For brevity we say that a multigraph is the incidence multigraph of a query Q, denoted $Inc(Q)$, if it is the incidence multigraph of the basic graph pattern of that query. For example, for a query

$$Q_{follows} = \textsc{Select}\ \ ?X\ \ \textsc{Where}\ \ P\ \ \textsc{Filter}\ \ \{\},$$
$$\text{with } P = \{(?X, follows, ?X), (?X, follows, ?Y), (?Y, follows, ?X)\}$$

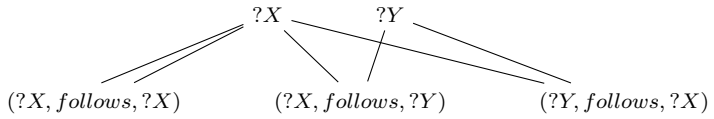


Figure 16: Incidence multigraph for the $Q_{follows}$ query

its corresponding incidence multigraph is depicted in Figure 16.

Cycles in an incidence multigraph are defined analogous to *Berge cycles* [17]: a tuple of vertices $(v_1, \ldots, v_n)$ from $Inc(Q)$ is a cycle, with $n \geq 2$, if for every pair of vertices v_i, v_{i+1} there exists a distinct edge $\{v_i, v_{i+1}\}$ in $Inc(Q)$, for $1 \leq i < n$, and if there exists a distinct edge $\{v_1, v_n\}$ in $Inc(Q)$. Note that unlike in Berge cycles, vertices in the tuple are not required to be distinct. An incidence multigraph is acyclic if and only if it contains no cycles. For example, the incidence multigraph depicted in Figure 16 is cyclic, since it contains multiple cycles, e.g., $(?X, (?X, follows, ?X), ?X, (?X, follows, ?Y), ?Y, (?Y, follows, ?X)$ or $(?X, (?X, follows, ?X))$.

We are now ready to define the notion of query acyclicity used in our framework.

Definition 18 (Acyclic query)**.** *A query is* acyclic *if and only if its incidence multigraph is acyclic.*

For example, $Q_{follows}$ is cyclic, while the following query is acyclic:

$$Q'_{follows} = \textsc{Select}\ \ ?X\ \ \textsc{Where}\ \ P\ \ \textsc{Filter}\ \ \{\},$$
$$\text{with } P = \{(?X, follows, ?Y), (?Y, follows, ?Z)\}.$$

Our notion of acyclicity is stricter than that of Berge-acyclicity (and therefore of other conventional definitions of acyclic queries, see Section 2.2.4), and it is motivated by the graph nature of RDF data. In a setting where queries are evaluated over RDF data, the definition of Berge-acyclicity (adapted to SPARQL syntax in

the straightforward way) can sometimes yield counter-intuitive results, when several triple patterns in a query contain the same set of variables, or when a variable appears multiple times in a triple pattern. Consider a simple RDF graph $G = \{(a, r, b), (b, r, a)\}$ and three basic graph patterns:

$$P_1 = \{(?X, r, ?Y), (?Y, r, ?X)\},$$
$$P_2 = \{(?X, ?R, ?Y), (?Y, r, ?X)\},$$
$$P_3 = \{(?X, ?R, ?Y), (?Y, ?R, ?X)\}.$$

The three queries $Q_i = \textsc{Select}\ ?X\ \textsc{Where}\ P_i$, for $1 \leq i \leq 3$, match the cyclic RDF data and have the same answer sets $\{a, b\}$ over G. However, queries Q_1 and Q_3 are Berge-acyclic, while Q_2 is not. All the three queries are not acyclic in our framework. Another query $\textsc{Select}\ ?X\ \textsc{Where}\ \{(X, R, X)\}$ is Berge-acyclic, although it matches any self-loop in the data; it is also acyclic in the comparison framework.

6.2 Computing Acyclic MSSQs

6.2.1 Auxiliary Definition

Our algorithm relies on the notion of a *similarity tree* for entities a and b in a graph G, which we define next. Roughly speaking, a similarity tree is a labelled directed tree, where each node is labelled with a pair of sets of entities (appearing in subject and object positions in G), with the first set in a pair corresponding to a and the second to b; the root node is labelled with the pair $(\{a\}, \{b\})$. Each edge in the tree is labelled with two sets of entities (appearing in the predicate position in triples from G) and a direction of triples. Furthermore, we require that the tree is consistent with the structure of G in that each edge in the tree is justified by corresponding

triples in G.

Definition 19. *A* pair tree *is a rooted labelled directed tree such that*

- *each node v is labelled with a pair (V_1, V_2), where each V_i is a non-empty set of entities satisfying either $V_1 \cap V_2 = \emptyset$ or $V_1 = V_2 = \{c\}$ for an entity c; if $V_1 = V_2 = \{c\}$, then v must be a leaf;*
- *each edge e is labelled with a tuple (E_1, E_2, dir), where each E_i is a set of entities satisfying either $E_1 \cap E_2 = \emptyset$ or $E_1 = E_2 = \{c\}$ for an entity c, and where $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$.*

An edge $e = (v, v')$ in a pair tree $\mathcal{T}$ is justified *in a graph G if the following properties hold for both $i = 1, 2$, where (V_1, V_2), (E_1, E_2, dir), and (V_1', V_2') are labels of v, e, and v', respectively:*

- *for each entity $c \in V_i$ there is a triple* justifying *e in G for c— that is, a triple (s, p, o) such that $p \in E_i$ and either $s = c$ and $o \in V_i'$ when dir is $\rightarrow$, or $s \in V_i'$ and $o = c$ otherwise.*

Pair tree $\mathcal{T}$ is a similarity tree *for entities a and b in a graph G if the root is labelled with $(\{a\}, \{b\})$ and all edges in $\mathcal{T}$ are justified in G.*

Consider Figure 17, where a graph G_{ex} and two pair trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are depicted (for brevity, $George$, $follows$, and $retweets$ in the trees abbreviate $(\{George\}, \{George\})$, $\{follows\}, \{follows\}$, and $\{retweets\}, \{retweets\}$, respectively). Note that the roots in both trees are labelled by $(\{Ana\}, \{Bob\})$. In $\mathcal{T}_1$ the edge between the root and the node labelled $(\{Claire\}, \{David, Ellen\})$ is justified: for both Ana and Bob there exists a triple in G that has this entity as the subject, $follows$ as the predicate, and $Claire$ and $David$ (or $Ellen$), respectively, as the object. However, neither of the other two edges in $\mathcal{T}_1$ is justified, because of the $\{David, Ellen\}$ component in the parent node label: there are no triples $(George, retweets, Ellen)$ and

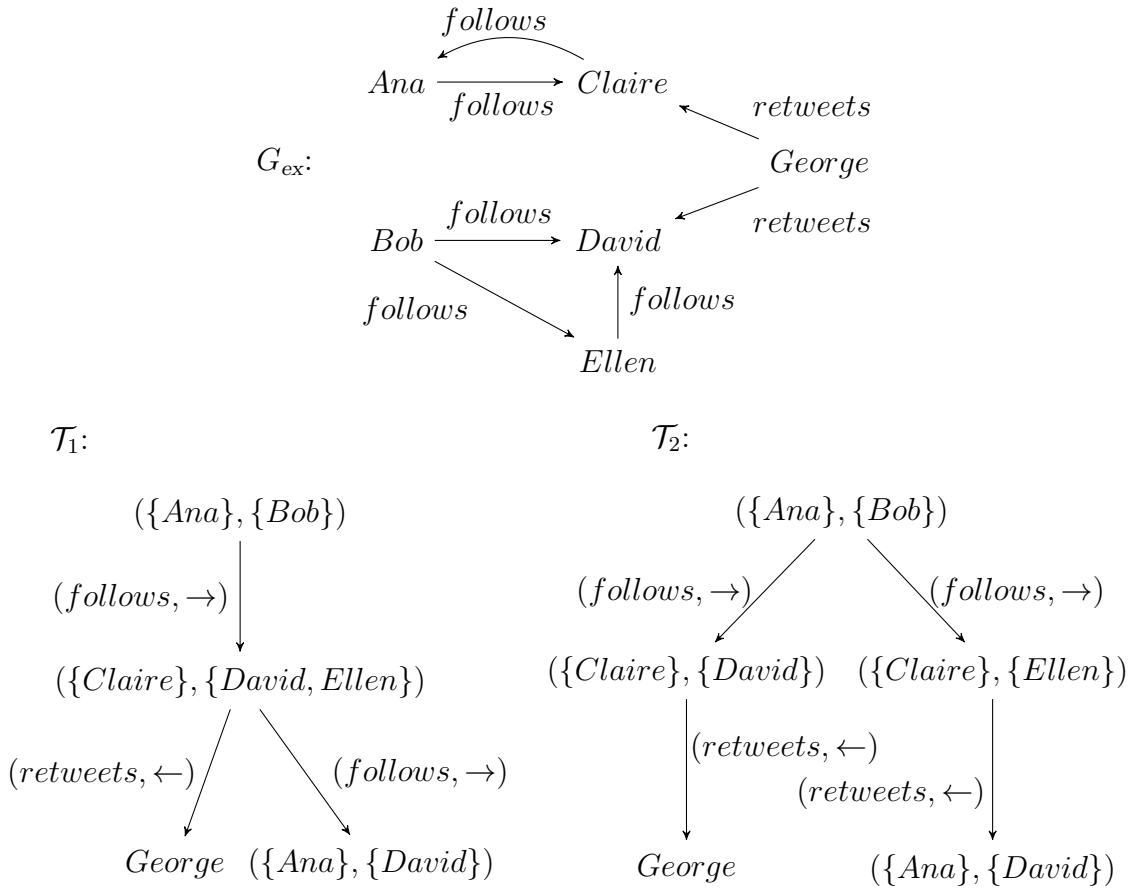


Figure 17: Example graph G_{ex} and two pair trees $\mathcal{T}_1$ and $\mathcal{T}_2$

$(David, follows, David)$ in G. In contrast, every edge in $\mathcal{T}_2$ is justified, and hence $\mathcal{T}_2$ is a similarity tree.

Similarity trees are relevant since they have corresponding similarity queries.

Definition 20. *Let $\mathcal{T}$ be a similarity tree for entities a, b in a graph G. For each node or edge u in $\mathcal{T}$ labelled with (L_1, L_2) or (L_1, L_2, dir), respectively, let t_u be*

- *a variable $?X$, if u is the root of the tree;*
- *the entity c, if $L_1 \cap L_2 = \{c\}$; or*
- *a fresh variable otherwise.*

The query corresponding *to $\mathcal{T}$ is* SELECT $?X$ WHERE P FILTER C *with*

- *P containing, for each edge $e = (v, v')$ in $\mathcal{T}$, the triple pattern $(t_v, t_e, t_{v'})$ or $(t_{v'}, t_e, t_v)$ if e is labelled with $\rightarrow$ or $\leftarrow$, respectively; and*
- *C containing, for each node v in $\mathcal{T}$ labelled (V_1, V_2) with each V_i consisting of only numeric values, the arithmetic comparisons $(t_v \geq min)$ and $(t_v \leq max)$, where min and max are the minimal and the maximal, respectively, values in $V_1 \cup V_2$.*

If we use the query formalism without arithmetic comparisons, we leave C empty. For example, the query corresponding to the similarity tree $\mathcal{T}_2$ from Figure 17 is

$$Q_{sim} = \textsc{Select}\ ?X\ \textsc{Where}\ \{(?X, follows, ?Y_1),\ (George, retweets, ?Y_1),\ (?X, follows, ?Y_2),\ (?Y_2, follows, ?Y_3)\}. \tag{6.1}$$

The following proposition establishes that the query corresponding to a similarity tree is indeed a similarity query having an acyclic structure.

Proposition 4. *The query corresponding to a similarity tree for entities a and b in a graph G is an acyclic similarity query for a and b in G.*

Proof. Let $\mathcal{T}$ be a similarity tree for a and b in G. The following holds for $\mathcal{T}$:

Claim. *For every node v in $\mathcal{T}$ labelled (V_1, V_2) and for every entity in its labels $c \in V_i$, for both $i = 1, 2$, it holds that in every child node v' connected to v with an edge $e = (v, v')$ (where (E_1, E_2, dir) and (V_1', V_2') are labels of e and v', respectively), there exist at least one entity $d \in V_i'$ and one entity $r \in E_i$ that form a justifying triple in G with c: (c, r, d) if dir is $\rightarrow$, or (d, r, c) if dir is $\leftarrow$.*

Proof. Since $\mathcal{T}$ is a similarity tree, it follows from Definition 19 that each edge $e = (v, v')$ in $\mathcal{T}$ is justified in G, i.e., for every entity in its labels $c \in V_i$ there exist at least one entity $d \in V_i'$ and one entity $r \in E_i$ such that (c, r, d) if dir is $\rightarrow$, or (d, r, c) if dir is $\leftarrow$. Let c be an entity in V_i, and let $v^1, \ldots, v^k$ be all child nodes of v.

Then it follows straightforwardly that there exist an entity in the i^{th} label of each child node and an entity in the i^{th} label of each edge connecting v with that child node, so that the entities form a triple justified in G. □

Now let us first traverse $\mathcal{T}$ from the root to the leaves and recursively associate each node and edge in $\mathcal{T}$ with a pair of entities such that the first entity is from the first component of the label of the node or edge and the second entity is from the second component, and the following holds:

- the root is associated with (a, b), and,
- for each edge $e = (v, v')$ with v associated with (c_a, c_b), e and v' are associated with pairs of entities (d_a, d_b), and (c'_a, c'_b), respectively, from the labels of e and v' such that the triples (c_a, d_a, c'_a) and (c_b, d_b, c'_b), if e is labelled by $\rightarrow$, or the triples (c'_a, d_a, c_a) and (c'_b, d_b, c_b) otherwise, justify c_a and c_b, respectively, in G (these triples exists by the claim above).

Let Q be the query corresponding to the similarity tree $\mathcal{T}$. Consider the valuations μ_a and μ_b that send $?X$ to a and b, respectively, and every other variable $?Y$ of Q to the entities c_a and c_b, respectively, in the pair associated to the node or edge u such that t_u is $?Y$ according to Definition 20. It is immediate to check that valuations μ_a and μ_b justify a and b as answers to Q, as required, satisfying both the basic graph pattern and the arithmetic filter condition of Q.

It remains to prove that Q is acyclic. We need the following auxiliary result.

Claim. *Let $(v_1, \ldots, v_n)$ be a cycle in an incidence multigraph $Inc(Q)$. By σ we define a subset of edges from $Inc(Q)$ that "support" the cycle: $\sigma = \{\{v_i, v_{i+1}\} | 1 \leq i < n\} \cup \{\{v_1, v_n\}\}$. Then each vertex v_i must be part of at least two distinct edges in σ.*

Proof. Each vertex v_i for $1 < i < n$ appear in edges $\{v_{i-1}, v_i\}$ and $\{v_i, v_{i+1}\}$. Vertex v_1 appear in edges $\{v_1, v_2\}$ and $\{v_1, v_n\}$. Vertex v_n appear in edges $\{v_{n-1}, v_n\}$ and $\{v_1, v_n\}$. □

Let λ be the translation function that assigns a term t_u to each node or edge u in $\mathcal{T}$, as described in Definition 20. The proof is by induction on the size of $\mathcal{T}$, i.e., on the number of edges in $\mathcal{T}$.

Base: Let $\mathcal{T}$ consist of one edge $e = (v, v')$. Then v is the root of $\mathcal{T}$, with $\lambda(v) = ?X$, and $\lambda(e), \lambda(v')$ are either entities or fresh variables. Then $Inc(Q)$ is trivially acyclic, hence a query Q corresponding to a similarity tree of size 1 is always acyclic.

Hypothesis: Q corresponding to a similarity tree of size k is always acyclic.

Step: Let $\mathcal{T}$ be of size $k + 1$, and let v be a leaf node in $\mathcal{T}$ with an incoming edge $e = (w, v)$. Without loss of generality e is labelled with $(L_1, L_2, \rightarrow)$. Furthermore, let $\mathcal{T}_k$ be a similarity tree obtained from $\mathcal{T}$ by removing v and e. Trivially, $\mathcal{T}_k$ is of size k. Moreover, let Q_k be a query corresponding to $\mathcal{T}_k$, with an incidence multigraph $Inc(Q_k)$. Finally, let $\mathcal{P}$ be the countably infinite set of all possible tuples formed of vertices from $Inc(Q_k)$. Since Q_k is acyclic under the induction hypothesis, no tuple in $\mathcal{P}$ is a cycle.

It holds by construction of Q that $P = P_k \cup p$, where $p = \{(\lambda(w), \lambda(e), \lambda(v))\}$, and P and P_k are basic graph patterns of Q and Q_k, respectively. The term $\lambda(w)$ must be a variable $?W$, and $\lambda(e), \lambda(v')$ are either entities or fresh variables, by definition of λ. Let us consider 4 cases:

- both $\lambda(e)$ and $\lambda(v')$ are entities

  Then $Inc(Q)$ is $Inc(Q_k)$ extended with a new vertex p and a new edge $\{?W, p\}$. $Inc(Q_k)$ is acyclic, and so is $Inc(Q)$: the new vertex appear only in one edge in $Inc(Q)$, and therefore it cannot be part of any cycle, as its σ would need to

have at least two edges with p. Since no tuple of vertices containing p can be a cycle, and all other tuples are the same as tuples in $\mathcal{P}$, Q is acyclic.

- $\lambda(e)$ is an entity and $\lambda(v)$ is a variable $?V$

  Then $Inc(Q)$ is $Inc(Q_k)$ extended with new vertices $?V$ and p, and new edges $\{?W, p\}$ and $\{?V, p\}$. A potential cycle cannot contain the vertex $?V$, since its σ would only have one edge with $?V$. But then it cannot contain p either, since without $?V$ the σ set would contain at most one edge with p. Therefore, all potential vertex tuples are the same as tuples in $\mathcal{P}$, so Q is acyclic.

- $\lambda(e)$ is a variable $?E$ and $\lambda(v)$ is an entity

  The case is similar to the previous one, except $Inc(Q_k)$ is extended with $?E$ and $\{?E, p\}$ instead of $?V$ and $\{?V, p\}$.

- $\lambda(e)$ is a variable $?E$ and $\lambda(v)$ is a variable $?V$

  Then $Inc(Q)$ is $Inc(Q_k)$ extended with new vertices $?V$, $?E$ and p, and new edges $\{?W, p\}$, $\{?V, p\}$ and $\{?E, p\}$. A potential cycle cannot contain the vertices $?V$ or $?E$, since $?V$ and $?E$ appear in precisely one edge each. Then in turn a cycle cannot contain p, since without $?V$ and $?E$ the σ set would contain at most one edge with p. Therefore, all potential vertex tuples are the same as tuples in $\mathcal{P}$, so Q is acyclic.

□

This concludes the introduction of the definitions necessary for computing approximated MSSQs, and we now move to the algorithm that computes one such query for the given input.

Algorithm 3: COMPUTE_APPROX_MSSQ

Input: graph G, entities a and b in G, depth dep
Output: acyclic similarity query for a and b in G

1 **let** $\mathcal{T}_0$ be a pair tree with a single root node v_0 labelled $(\{a\}, \{b\})$;
2 **let** $\mathcal{T}_{gen}$:= GENERATE_TREE($\mathcal{T}_0$, v_0, G, dep);
3 **let** $\mathcal{T}_{sim}$:= UNCOUPLE_NODES($\mathcal{T}_{gen}$, G);
4 **return** the query corresponding to $\mathcal{T}_{sim}$.

6.2.2 The Compute_Approx_MSSQ Algorithm

We are ready to present algorithm COMPUTE_APPROX_MSSQ (given in Algorithm 3), which computes an acyclic similarity query of a given depth dep (a natural number) for given entities a and b in a given graph G according to the three steps described next. In the first step (line 2), we create a preliminary pair tree $\mathcal{T}_{gen}$. For example, for the input graph G_{ex} from Figure 17, for the entities *Ana* and *Bob* in that graph and for depth 2 the pair tree $\mathcal{T}_{gen}$ is $\mathcal{T}_1$. As in this example, $\mathcal{T}_{gen}$ may not yet be a similarity tree. Hence, in the second step (line 3), we uncouple some of the nodes in $\mathcal{T}_{gen}$, making all edges in the tree justified, and thus creating a similarity tree $\mathcal{T}_{sim}$. For example, we uncouple the node from $\mathcal{T}_1$ labelled $(\{Claire\}, \{David, Ellen\})$ into two new nodes, labelled $(\{Claire\}, \{David\})$ and $(\{Claire\}, \{Ellen\})$, respectively. The former becomes the parent node for the node labelled *George*, while the latter becomes the parent node for the node labelled $(\{Ana\}, \{David\})$. As the result, in this example $\mathcal{T}_{sim}$ is $\mathcal{T}_2$. Finally (in step 4), we turn $\mathcal{T}_{sim}$ into an acyclic similarity query corresponding to this tree; for example we turn $\mathcal{T}_2$ into Q_{sim} from 6.1.

Let us look at each of the steps in more detail. In the first step (line 2), the algorithm constructs, by means of the recursive subroutine GENERATE_TREE (Algorithm 4), a pair tree $\mathcal{T}_{gen}$ of depth at most dep. In particular, in lines 1–2 of COMPUTE_APPROX_MSSQ a root labelled $(\{a\}, \{b\})$ is created and passed to the recursion. When a node v in $\mathcal{T}$ labelled (V_1, V_2) is received in a recursive call of GEN-

Algorithm 4: GENERATE_TREE

Input: pair tree $\mathcal{T}$, node v in $\mathcal{T}$ labelled (V_1, V_2), graph G, depth dep
Output: pair tree of depth dep

foreach $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$, c, and d with $(c_i, d, c)^{\mathsf{dir}} \in G$ for $c_i \in V_i$, $i = 1, 2$ **do**
  add to $\mathcal{T}$ an edge labelled $(\{d\}, \{d\}, \mathsf{dir})$
    from v to a new node labelled $(\{c\}, \{c\})$;
foreach $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ and c with $(c_i, d_i, c)^{\mathsf{dir}} \in G$ for $c_i \in V_i$, $i = 1, 2$ **do**
  let $E_i := \{d_i \mid (c_i, d_i, c)^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\}$ for each $i = 1, 2$;
  if $E_1 \setminus E_2 \neq \emptyset$ and $E_2 \setminus E_1 \neq \emptyset$ **then**
    add to $\mathcal{T}$ an edge labelled $(E_1 \setminus E_2,\ E_2 \setminus E_1,\ \mathsf{dir})$
      from v to a new node labelled $(\{c\}, \{c\})$;
foreach $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ and d with $(c_i, d, c_i')^{\mathsf{dir}} \in G$ for $c_i \in V_i$, $i = 1, 2$ **do**
  let $V_i' := \{c_i' \mid (c_i, d, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\}$ for each $i = 1, 2$;
  if $V_1' \setminus V_2' \neq \emptyset$ and $V_2' \setminus V_1' \neq \emptyset$ **then**
    add to $\mathcal{T}$ an edge labelled $(\{d\}, \{d\}, \mathsf{dir})$
      from v to a new node v' labelled $(V_1' \setminus V_2',\ V_2' \setminus V_1')$;
    if $\mathsf{dep} > 0$ **then let** $\mathcal{T} :=$ GENERATE_TREE$(\mathcal{T}, v', G, \mathsf{dep} - 1)$;
foreach $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ **do**
  let $E_i := \{d_i \mid (c_i, d_i, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\}$ for each $i = 1, 2$;
  let $V_i' := \{c_i' \mid (c_i, d_i, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i \text{ and } d_i \in E_i \setminus E_{3-i}\}$ for each $i = 1, 2$;
  if $E_1 \setminus E_2 \neq \emptyset$, $E_2 \setminus E_1 \neq \emptyset$, $V_1' \setminus V_2' \neq \emptyset$, and $V_2' \setminus V_1' \neq \emptyset$ **then**
    add to $\mathcal{T}$ an edge labelled $(E_1 \setminus E_2,\ E_2 \setminus E_1,\ \mathsf{dir})$
      from v to a new node v' labelled $(V_1' \setminus V_2',\ V_2' \setminus V_1')$;
    if $\mathsf{dep} > 0$ **then let** $\mathcal{T} :=$ GENERATE_TREE$(\mathcal{T}, v', G, \mathsf{dep} - 1)$;
return $\mathcal{T}$.

ERATE_TREE, the following extensions are performed, where $(s, p, o)^{\rightarrow}$ and $(s, p, o)^{\leftarrow}$ denote (s, p, o) and (o, p, s), respectively:

– first, for each direction $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ and each pair of entities c, d such that, for both $i = 1, 2$, there are triples $(c_i, d, c)^{\mathsf{dir}} \in G$ with $c_i \in V_i$, a new edge labelled $(\{d\}, \{d\}, \mathsf{dir})$ from v to a new node labelled $(\{c\}, \{c\})$ is added to $\mathcal{T}$;

– second, for each $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ and each entity c such that, for both $i = 1, 2$, there

exists $(c_i, d_i, c)^{\mathsf{dir}} \in G$ with $c_i \in V_i$ the sets

$$E_i = \{d_i \mid (c_i, d_i, c)^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\}$$

are considered; if the sets $E_1 \setminus E_2$ and $E_2 \setminus E_1$ (i.e., the sets of edge entities not covered in the previous case) are both non-empty, then an edge labelled $(E_1 \setminus E_2,\ E_2 \setminus E_1,\ \mathsf{dir})$ from v to a new node labelled $(\{c\}, \{c\})$ is added;

– third, for each $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ and each d such that, for both $i = 1, 2$, there are triples $(c_i, d, c_i')^{\mathsf{dir}} \in G$ with $c_i \in V_i$ the sets

$$V_i' = \{c_i' \mid (c_i, d, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\}$$

are considered; if $V_1' \setminus V_2'$ and $V_2' \setminus V_1'$ (i.e., the sets of not covered node entities) are non-empty, then an edge labelled $(\{d\}, \{d\}, \mathsf{dir})$ from v to a new node v' labelled $(V_1' \setminus V_2',\ V_2' \setminus V_1')$ is added; moreover, if the depth of v is non-zero, then GENERATE_TREE is recursively called for v';

– finally, for both $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ the sets

$$E_i = \{d_i \mid (c_i, d_i, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\} \text{ and}$$
$$V_i' = \{c_i' \mid (c_i, d_i, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i \text{ and } d_i \in E_i \setminus E_{3-i}\}$$

are considered for both $i = 1, 2$; if the sets $E_1 \setminus E_2$, $E_2 \setminus E_1$, $V_1' \setminus V_2'$, and $V_2' \setminus V_1'$ are all non-empty, then an edge labelled $(E_1 \setminus E_2,\ E_2 \setminus E_1,\ \mathsf{dir})$ from v to a new node v' labelled $(V_1' \setminus V_2',\ V_2' \setminus V_1')$ is added; moreover, if the depth of v is non-zero, then GENERATE_TREE is called for v'.

After all these extensions, $\mathcal{T}$ is returned to the previous level of recursion.

As mentioned above, the resulting $\mathcal{T}_{gen}$ is a pair tree; however, it may not be a

Algorithm 5: UNCOUPLE_NODES

Input: pair tree $\mathcal{T}$, graph G
Output: a pair tree

```
1  let dep be the depth of T;
2  for i := dep − 1 to 0 do
3      foreach node v labelled (V1, V2) of depth i do
4          foreach node v' labelled (V1', V2') with (v, v') labelled (E1, E2, dir) do
5              add to T a node v* and an edge (v*, v') labelled (V1*, V2*) and
                (E1*, E2*, dir), respectively, for maximal Vi* ⊆ Vi and Ei* ⊆ Ei,
                i = 1, 2, with (v*, v') justified by G;
6              if i > 0 then
7                  add to T an edge (u, v*) labelled as the incoming edge (u, v)
                    to v;
8          merge each set of v* with the same label;
9          remove v with adjusted edges from T;
10 return T.
```

similarity tree for a and b, since some edges may not be justified in G. So, in the second step (line 3) of COMPUTE_APPROX_MSSQ, pair tree $\mathcal{T}_{gen}$ is refined from the leaves upwards using subroutine UNCOUPLE_NODES, which ensures that each edge in the tree is suitably justified, and hence yields a similarity tree $\mathcal{T}_{sim}$ for a and b in G. In particular, this subroutine considers nodes of its input pair tree $\mathcal{T}$ from leaves to the root, and for each node v under consideration and each child v' of v—that is, a node with an edge $e = (v, v')$—the following is performed, where (V_1, V_2), (E_1, E_2, dir), and (V_1', V_2') are labels of v, e, and v', respectively:

– a node v^* and an edge (v^*, v') labelled (V_1^*, V_2^*) and $(E_1^*, E_2^*, \mathsf{dir})$, respectively, are added to $\mathcal{T}$, for maximal sets $V_i^* \subseteq V_i$ and $E_i^* \subseteq E_i$, $i = 1, 2$, with (v^*, v') justified by G;

– if v is not the root, then an edge (v_p, v^*) labelled as the incoming edge (v_p, v) to v is added to $\mathcal{T}$;

– when all children of v are processed, each group of children with the same label are merged to one, and v is removed.

Note that, by construction, the resulting $\mathcal{T}_{sim}$ is a pair tree as well; moreover, we will see that, contrary to $\mathcal{T}_{gen}$, it is a similarity tree.

Finally, in the last step (line 4), algorithm Compute_Approx_MSSQ constructs the query corresponding to the similarity tree according to Definition 20, which is guaranteed to be an acyclic similarity query by Proposition 4.

Overall, we arrive to the following correctness theorem.

Theorem 10. *For each positive integer* dep*,* Compute_Approx_MSSQ *computes an acyclic similarity query for entities* a *and* b *in a graph* G*.*

Proof. From Proposition 4 it follows that a query corresponding to a similarity tree for entities a and b in a graph G is an acyclic similarity query for a and b in G. It remains to prove that Compute_Approx_MSSQ computes a similarity tree at step 3, i.e., that (i) a pair tree $\mathcal{T}$ computed at step 3 has a root node labelled with $(\{a\}, \{b\})$, and (ii) each edge in $\mathcal{T}$ is justified in G.

In order to prove (i), let us look at the construction of $\mathcal{T}$. At step 1, an initial pair tree $\mathcal{T}_0$ with a root node labelled $(\{a\}, \{b\})$ is constructed. At step 2, $\mathcal{T}_0$ is populated with additional nodes by the recursive subroutine Generate_Tree. At each of all Generate_Tree new nodes may be added, but the already existing nodes and their labels are not modified. Thus, the pair tree $\mathcal{T}_{gen}$ obtained at step 2 has the root node labelled $(\{a\}, \{b\})$. Finally, at step 3 the $\mathcal{T}$ is obtained from the subroutine Uncouple_Nodes using $\mathcal{T}_{gen}$.

Claim. *If a pair tree* $\mathcal{T}'$ *is obtained from* Uncouple_Nodes *and another pair tree* $\mathcal{T}$*, then the root node in* $\mathcal{T}'$ *is labelled with the same pair of singletons* $(\{c_1\}, \{c_2\})$ *as the root node in* $\mathcal{T}$*.*

Proof. Let dep be the depth of $\mathcal{T}$. The subroutine Uncouple_Nodes iterates over nodes at various depth, starting with $\mathsf{dep} - 1$ and up to depth 0. The root node v of $\mathcal{T}$ is not modified by the subroutine up until the last iteration of the for-loop

in step 2, when nodes at depth 0, i.e., the root node itself, are considered. Then all child nodes of v are 'uncoupled' in steps 4–7, and for each child node a new parent node v^* labelled (V_1^*, V_2^*) is created; only the entities from $(\{c_1\}, \{c_2\})$ that are justified in G are left in (V_1^*, V_2^*). Let $e = (v, v')$ be an edge labelled (E_1, E_2, dir) between the root node v and its child node v' labelled (V_1', V_2'). It has been created in the GENERATE_TREE, in line 2, 7, 12 or 19. In either case, for both $i = 1, 2$, E_i and V_i' are populated from the triples $(c_i, p, o)^{\mathsf{dir}} \in G$, for some p and o. Therefore, $(V_1^*, V_2^*) = (\{c_1\}, \{c_2\})$ for all newly created nodes v^*, and therefore all of them are merged into a single node labelled $(\{c_1\}, \{c_2\})$ in line 8. This node becomes the new root of $\mathcal{T}$ in line 9. □

Hence, the root node of $\mathcal{T}$ is labelled $(\{a\}, \{b\})$.

In order to prove (ii), we also reverse-engineer the construction of $\mathcal{T}$. Let $e = (v, v')$ be an edge labelled (E_1, E_2, dir) between a node v labelled (V_1, V_2) and a node v' labelled (V_1', V_2') in $\mathcal{T}$. It has been first created in the GENERATE_TREE subroutine, in line 2, 7, 12 or 19, and then 'uncoupled' in the UNCOUPLE_NODES subroutine, in lines $3 - 9$. In GENERATE_TREE, an edge e is created if one of the conditions in lines 1, 4, 9 or 15 is met. Each condition requires that for both $i = 1, 2$ *there exist some* entity $c_i \in V_i$ such that a triple $(c_i, d_i, c_i')^{\mathsf{dir}}$ exists in G; d_i and c_i' are then added to E_i and V_i', respectively. Then each edge $e = (v, v')$ is updated in UNCOUPLE_NODES into a new edge $e^* = (v^*, v')$, where v^* and e^* are labelled with maximal $V_i^* \subseteq V_i$ and $E_i^* \subseteq E_i$ such that *for all* entities $c_i \in V_i^*$ there exists a triple $(c_i, d_i, c_i')^{\mathsf{dir}}$ in G with $d_i \in E_i^*$ and $c_i' \in V_i'$,for both $i = 1, 2$. Hence, each edge in $\mathcal{T}$ is justified by G. □

6.2.3 Algorithmic Properties

In this section we are briefly going to discuss some of the practical aspects of the COMPUTE_MSSQ and COMPUTE_APPROX_MSSQ algorithm, leaving the detailed

qualitative and quantitative analysis to Chapter 7.

Restriction on Input Entities

The theoretical framework and the exact algorithm Compute_MSSQ treat subjects, predicates, and objects in the same way; however, in practice we are mostly interested in comparing subject and object entities, e.g., persons, locations or artifacts. Predicates, on the other hand, are typically considered as relations that link subject and object entities together. Indeed, it is common in practical research on RDF to assume that entities in RDF graphs are split into disjoint sets of *node entities*, which can appear only in subject and object positions of triples, and *edge entities*, which can appear only in predicate positions. As we focus our interest on comparing node entities, our approximation algorithm assumes that the compared entities appear in the graph either both as subjects or both as objects at least once (and hence an MSSQ exists).

Depth of Approximation Queries

One crucial difference between Compute_MSSQ and Compute_Approx_MSSQ is that the former algorithm does not have the depth parameter dep. Intuitively, instead of comparing the neighbourhoods of two input entities in an RDF graph, Compute_MSSQ considers the whole graph and all patterns common for the two target entities that it contains. This, however, proves to be neither feasible, not practically useful: we would much rather be interested in the information that is directly related to the two entities in question, than in the information known about an entity that is related to an entity, that is in turn related to an entity, etc., that is related to the input entities.

In turn, the approximation algorithm Compute_Approx_MSSQ only compares two entities with respect to their immediate neighbourhoods of a certain depth

dep in the graph, i.e., it considers only the triples that can be reached in the graph by at most dep hops from the two entities, in the bread-first search manner [101]. This enables us to explicitly control how much of the input data we would like to consider in order to compare the two entities, and how deep the comparison process should go.

Formally, a sequence of triples (resp. triple patterns) $(p_1, \ldots, p_k)$ from an RDF graph (resp. from a query) is called *connected*, if for every pair p_i, p_{i+1}, $1 \leq i < k$, it holds that $e \in p_i \cap p_{i+1}$, with e appearing in vertex positions in both p_i and p_{i+1}. By N_e^d we denote a *neighbourhood* of an entity e of depth d in an RDF graph G, which includes all triples reachable from e by connected sequences of triples of up to length d:

$$N_e^d = \{t \mid t \in G \text{ and } \exists(t_1, \ldots, t_k) \text{ connected triple sequence in } G \\ \text{s.t. } e \in t_1,\ t = t_k \text{ and } k \leq d\}.$$

Moreover, a query Q is said to be *of depth* d, if for every triple pattern $p \in Q$ there exists a connected sequence of triple patterns $(p_1, \ldots, p_k)$ with $k \leq d$ such that $?X \in p_1$ and $p = p_k$. Then for the input a, b, G and dep the Compute_Approx_MSSQ algorithm computes queries of depth dep over neighbourhoods N_a^{dep} and N_b^{dep} of G.

Running Time

Finally, we briefly discuss the running time of the algorithm. One execution of the Generate_Tree subroutine runs in $\mathcal{O}(\rho \cdot |G|)$, where ρ is the number of different entities appearing in the predicate position in triples from G. Generate_Tree is recursively called at most $(2\rho)^{\mathsf{dep}-1}$ times, hence the full runtime of these calls is $\mathcal{O}(\rho^{\mathsf{dep}} \cdot |G|)$. Then the subroutine Uncouple_Nodes performs a check on $\mathcal{O}(\rho^{\mathsf{dep}} \cdot |G|)$ pair tree nodes, each check being in $\mathcal{O}(|G|)$. Hence, Compute_Approx_MSSQ

runs in $\mathcal{O}(\rho^{\mathsf{dep}} \cdot |G|^2)$ in the worst case. Note that ρ for a graph G is typically much smaller in practice than the number of triples in G (e.g., $\rho = 128$ for full YAGO), and the checks in UNCOUPLE_NODES are made for all triples in G containing the current entity, which usually constitute only a small fraction of G. This makes the algorithm suitable for real-case scenarios, which we will demonstrate in Chapter 7.

Chapter 7

Implementation and Evaluation

In this chapter we are going to discuss the implementation of the two algorithms, Compute_MSSQ and Compute_Approx_MSSQ, and the evaluation of the latter on synthetic and real-world datasets. We will illustrate why Compute_MSSQ is not suitable for practical use, and perform an indirect comparison of the two algorithms in a limited setting that allows us to estimate how close the outputs of the two algorithms in terms of query specificity are. Finally, we will discuss possible optimisations of Compute_Approx_MSSQ that would further improve its scalability.

7.1 Implementation

We have implemented both algorithms in Python.[1] While the implementation of Compute_MSSQ was necessary in order to (a) validate that the algorithm is computationally infeasible in real settings, and (b) perform comparative evaluation of the two algorithms (see Section 7.2.2), it is of little value as a standalone implementation. Therefore, in this section we are going to focus on the implementation of

[1]The source code of the implementation can be found at `https://github.com/chinmusique/entitycomparison`.

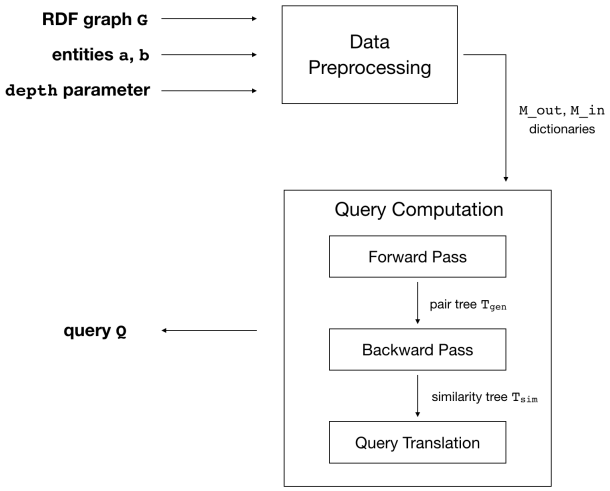


Figure 18: Architecture of the Compute_Approx_MSSQ implementation

Compute_Approx_MSSQ.

We have chosen Python since it is one of the best high-level programming languages for prototyping and implementing proof-of-concept ideas. If we were to develop an end-user entity comparison tool, we might have used a different programming language that allows for more performance optimisation, however, in the scope of this thesis we focus on the initial development of the comparison algorithms. The implementation can be used through the command line. It takes as input an RDF graph, two entities from it, and a predefined parameter *depth*, and computes the acyclic SQ that is returned to the user through the standard output. The workflow of the implementation can be divided into two stages: the preprocessing stage that handles the input graph and loads it into the memory, and the main stage for computing the approximated MSSQ. We next describe each stage in more detail, while the architecture of the Compute_Approx_MSSQ implementation can be found in Figure 18.

Data Preprocessing The implementation stores the input RDF graph in memory and does not use third-party triple stores. Instead, it reads the input data from one

of the selected formats (*.ttl*, *.tsv* or *.mtx*), splits it into triples, and loads it into two dictionaries that group triples by common subjects, predicates and objects.

The computation in COMPUTE_APPROX_MSSQ is based on generating the similarity tree, which in turn is based on grouping triples sharing the same subject and predicate, or the same object and predicate. To facilitate the retrieval of such triples for a pair of sets of labels, we transform the input graph G into two dictionaries, `M_out` and `M_in`, that are structured as follows: every element, i.e., an entity or a blank node, appearing as a subject (resp. object) in G is mapped to all elements that co-appear in the same triples with it as predicates; every such predicate element is in turn mapped to all elements co-appearing as objects (resp. subjects) in the same triples with it and the subject (resp. object) element. A schematic structure of `M_out` is depicted in Figure 19; the structure of `M_in` is analogous. For example, for the G_{soc} RDF graph the dictionaries will look as follows:

$$G_{soc} = \{(anna,\, likes,\, bob),\\ (anna,\, likes,\, carla),\\ (bob,\, likes,\, carla),\\ (anna,\, follows,\, carla)\}$$

$$\texttt{M\_out} = \{anna : \{likes : \{bob, carla\},\\ follows : \{carla\}\},\\ bob : \{likes : \{carla\}\}\}$$

$$\texttt{M\_in} = \{bob : \{likes : \{anna\}\},\\ carla : \{likes : \{anna, bob\},\\ follows : \{anna\}\}\}.$$

The process of generating the `M_out` and `M_in` dictionaries runs in time linear to the size of the input graph. In turn, generating child nodes from a given node labelled with (L_1, L_2) in a pair tree boils down to three steps:

- query `M_out` with every element in L_1 and collect all relevant predicate and vertex elements ($\mathcal{O}(|G|) \times \mathcal{O}(1) = \mathcal{O}(|G|)$);
- do the same for L_2;
- do the intersection of two sets of predicate elements ($\mathcal{O}(|G|)$).

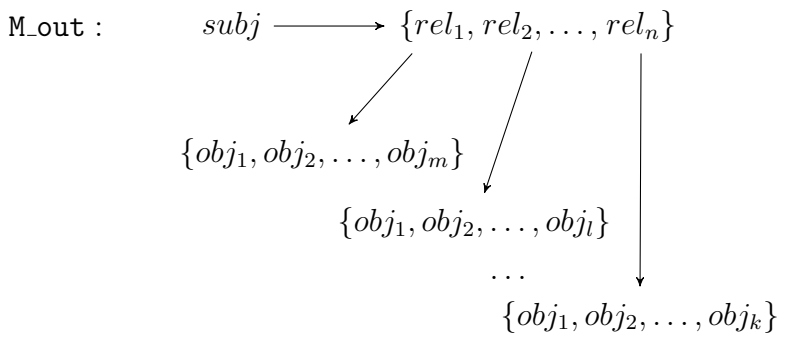


Figure 19: Schematic structure of the `M_out` dictionary

Hence, generating a new set of child nodes using the precomputed `M_out` and `M_in` has linear time complexity.

We deliberately chose to load the full RDF data dump in memory for the proof-of-concept implementation; as the performance analysis shows in Section 7.2.1, current implementation can efficiently process RDF graphs of up to 1 million triples. However, loading very large graphs of hundreds of millions of triples could be unfeasible. To this end, two workarounds could be suggested: either loading the data partially, e.g., only the triples that are in the neighbourhoods of the target nodes in the graph, or using a SPARQL endpoint.[2] In the latter case, the code should be modified so that the `M_out` and `M_in` dictionaries are incrementally computed in the course of constructing the initial pair tree by several calls to the endpoint.

Query Computation The main phase of the computation closely follows the pseudocode presented in Algorithm 3. An object of class `TreeNode` is created, and it represents the root node of a pair tree to be computed. Each `TreeNode` node has four properties: two sets of labels, a set of references to child nodes, and a reference to the parent node. The root node created initially has as labels singletons containing the two input entities, and contains no parent or child node references. Next comes the "forward pass" of the computation, in which a pair tree $\mathcal{T}_{gen}$ is

[2]e.g., DBpedia SPARQL endpoint: `http://dbpedia.org/sparql`

created. The actual implementation of the GENERATE_TREE procedure described in Algorithm 4 is done in two steps: firstly, a pair tree is computed in which each `TreeNode` node has property values for the labels and the child nodes; secondly, the pair tree is traversed one more time, and each non-root node is updated with the parent node information. The depth of $\mathcal{T}_{gen}$ is predefined by a numeric input parameter `depth`. After $\mathcal{T}_{gen}$ is computed, it is used as input in the "backward pass" of the computation, in which the pair tree is traversed from leaves to root, and some nodes are uncoupled as per the UNCOUPLE_NODES procedure described in Algorithm 5; the updated pair tree is a similarity tree $\mathcal{T}_{sim}$. The "backward pass" is the most computationally expensive step in the whole process of generating an acyclic SQ. Finally, $\mathcal{T}_{sim}$ is rewritten into a corresponding acyclic SQ Q by a recursive procedure that follows Definition 20. The tree is traversed from root to leaves, and each edge is translated into a triple, while each node or edge pair of labels is translated into a term. Additionally, for each node that is labelled only with numeric values and that cannot be translated into a numeric literal, a pair of arithmetic constraints is created.

Optimization Remarks To further improve the scalability of the algorithm, and in particular, to speed up the bottleneck step of the computation, i.e., the "backward pass", we could introduce two more input parameters, `max_label_size` and `max_child_nodes`, which would set the upper limit on the number of elements in each label and on the number of child nodes for each non-leaf node, respectively. The two parameters would be used during the generation of the pair tree $\mathcal{T}_{gen}$, and in case a node had too many elements in its label or too many child nodes, only the allowed number of randomly chosen such elements and nodes would be kept in the tree, while the rest would be ignored. Such optimization would establish a tradeoff between how specific the output queries are and how fast the computation is, while

preserving the correctness of the algorithm: the output queries would still be acyclic SQs for the given entities in a graph.

7.2 Evaluation

We evaluated the performance of our implementations and estimated to what extent the similarity queries computed by algorithm Compute_Approx_MSSQ approximate MSSQs computed by algorithm Compute_MSSQ in practical cases. We used the following three RDF graphs (datasets) in our experiments:

- the synthetic graph LUBM1 [50] consisting of $100,543$ triples over $26,437$ entities, out of which 17 appear in the predicate positions;
- a subset of the anonymised Twitter follower graph (TFG) [100] consisting of $713,319$ triples over $404,719$ entities, only one of which (i.e., entity *follows*) appears in the predicate positions; and
- a subset of YAGO graph [114] consisting of $1,069,072$ triples over $604,905$ entities, out of which 42 appear in the predicate positions.

The graphs are different in size and nature: YAGO has a rich set of property entities, while TFG uses only one; LUBM1 has a regular structure and resembles data typically encountered in databases, whereas YAGO is more heterogeneous. The summary of the datasets is presented in Table 4.[3]

All experiments were performed on a MacBook Air laptop with macOS 10.14, 1.6 GHz Intel Core i5 processor, and 16 GB 2133 MHz LPDDR3 memory.

[3]The three datasets used in our experiments are available for downloading at `https://zenodo.org/record/3685288`.

	number of entities	number of triples	number of predicate types
YAGO	604,905	1,069,072	42
TFG	404,719	713,319	1
LUBM1	26,437	100,543	17

Table 4: Overview of the YAGO, TWG and LUBM1 datasets.

7.2.1 Performance Analysis

We start by analysing the performance and scalability of the two algorithms. Due to the computational bottleneck of building a product graph of the input RDF graph (see Chapter 6), Compute_MSSQ timed out on all inputs. On the other hand, Compute_Approx_MSSQ yielded decent performance results on all three datasets, which we will report next.

We evaluated the runtime of our implementation of Compute_Approx_MSSQ for increasing values of the depth parameter. For this, we randomly selected 100 pairs of entities in each graph and, for each such pair, we ran the implemented algorithm for values of the depth parameter ranging from 1 to 4. For each graph and each depth value, we recorded the average, median and maximum runtime as well as the average number of triple patterns in a query among all the selected pairs of entities. We limited the maximum depth to 4, since queries beyond that depth are very difficult to comprehend due to their size and structure; indeed, psychologists established precise limitations in the human capacity to store and process information, where experiments show that most people would have trouble keeping in memory chains of related pieces of information longer than 4 [35].

Our results for LUBM1, TFG, and YAGO are summarised in Table 5. We can observe that our similarity queries can be computed efficiently with sub-second average running times in most cases; in contrast our implementation of exact Compute_MSSQ timed out in all cases. The average runtime becomes larger for depth 4 for larger datasets, such as TFG and YAGO; in case of YAGO the algo-

RDF graph	depth	runtime				size
		avg	median	max	t-outs	avg
LUBM1	1	0.000851	0.000346	0.006910	–	1.88
	2	0.002690	0.000971	0.036051	–	11.25
	3	0.072132	0.001389	2.101702	–	463.00
	4	0.348439	0.002058	8.558924	–	3235.02
TFG	1	0.000811	0.000356	0.045334	–	0.75
	2	0.001115	0.000373	0.045334	–	3.54
	3	0.058080	0.000415	3.540030	–	592.86
	4	67.203592	11.308518	352.100547	–	35904.21
YAGO	1	0.000918	0.000327	0.056005	–	0.73
	2	0.006476	0.000338	0.175918	–	7.81
	3	8.318439	0.000347	461.952534	–	149.63
	4	84.950921	0.640530	488.342738	3	1287.67

Table 5: Runtime (in seconds) and output query size (in number of triples) of Compute_Approx_MSSQ on the LUBM1, TFG, and YAGO graphs

rithm reached 3 timeouts for 500 seconds threshold. However, we can also observe that output queries tend to become very large (and hence difficult to interpret, verbalise, and comprehend) for depths greater than 3. Therefore, it is only practical to consider approximated MSSQs of depth up to 3 for the selected RDF graphs, for which our algorithm can always compute a similarity query.

7.2.2 Query Specificity Analysis

In this section we report the results of an experiment that aims to estimate how different the similarity queries computed using Compute_Approx_MSSQ are from the actual MSSQs. Unfortunately, our implementation of Compute_MSSQ timed out and hence failed to produce a query for all inputs in our datasets; thus, a direct comparison of the answers to the similarity queries produced by the algorithms is not feasible. To circumvent this limitation, we have designed an experiment consisting of the following steps for each of the LUBM1, TFG, and YAGO graphs:

1. we first created 40 random connected graphs, called *pattern graphs*, such that

each of them consists of 4 triples; among them 20 pattern graphs were acyclic, and 20 contained cycles;

2. for each pattern graph G, we created its copy G' with all entities renamed to fresh entities (except TFG, where the only property entity is used in both copies);

3. we then picked an entity a from each such G at random and the corresponding a' in the copy G', and ran both algorithms on $G \cup G'$ as a graph and a, a' as input entities; the approximation algorithm was run for depths 1 to 3;

4. finally, we evaluated the resulting queries on the considered graph (LUBM1, TFG, or YAGO) and compared the answers.

Intuitively, each pattern graph G represents a 'pattern' that may occur in the real data (and hence a pattern that will be reflected in the MSSQ). The approximation algorithm Compute_Approx_MSSQ constructs an acyclic query (i.e., a tree-like query in which variables in the predicate positions of triple patterns occur at most once), and hence the query returned by Compute_Approx_MSSQ on a graph $G \cup G'$ may not faithfully reflect the data pattern encoded by G. By evaluating the resulting queries in step 4 we are assessing (a) how faithfully the approximated query reflects the pattern, as well as (b) how common each pattern is in the graph, based on the total number of answers to the MSSQ.

Our results are summarised in Table 6. Firstly, acyclic patterns are much more common than patterns containing cycles in all three RDF graphs. This runs in accordance with the comprehensive study of queries over RDF data by Arias et al. [10], which concluded that most real-world SPARQL queries have a very simple structure. This validates the acyclicity condition of the Compute_Approx_MSSQ algorithm. Moreover, as can be seen from the average percentage of entities contained in query answer sets, similarity queries computed by Compute_Approx_MSSQ become

RDF graph		MSSQs		Approximations					
				dep = 1		dep = 2		dep = 3	
		avg	%	avg	%	avg	%	avg	%
LUBM1	A	7983.15	30.20	12157.45	45.97	10360.35	39.19	10332.05	39.08
	C	33.65	0.13	6697.00	25.33	2960.45	11.20	2522.35	9.54
TFG	A	156566.47	38.69	161958.50	40.02	161345.60	39.87	156566.47	38.69
	C	42838.20	10.58	83284.95	20.59	82541.10	20.39	78122.65	19.30
YAGO	A	147284.37	24.51	207236.80	34.26	175541.00	29.02	169331.26	27.99
	C	7175.25	1.19	83641.85	13.83	44372.90	7.34	41518.15	6.86

Table 6: Average number of answers (avg) and average percentage of all entities in answers (%) to MSSQs and the approximating queries, computed over acyclic (A) and cyclic (C) pattern graphs and evaluated on the LUBM1, TFG, and YAGO graphs

more specific and closer to MSSQs as the depth grows. Unsurprisingly, the approximating queries evaluated on the TFG graph are almost identical to MSSQs, since the graph contains a single relation. It should be noted that in general we cannot hope to have theoretical guarantees of a constant approximation ratio for any approximation algorithm that outputs an SQ that is not an MSSQ, since it is always possible to find an example where the approximating SQ has arbitrary many answers while the MSSQ has just two (i.e., the input entities) over some input graph. However the empirical approximation error demonstrated by our experiments consistently goes below 10% for both cyclic and acyclic pattern graphs for depth 3 on all datasets, as can be seen from the percentage for MSSQs and approximated queries of $\texttt{depth} = 3$. This makes Compute_Approx_MSSQ suitable for real-world applications of entity comparison.

If we are to put the results from Table 6 in information retrieval terms, then it makes more sense to talk about accuracy rather then precision and recall, since the recall value (i.e., the ratio of answers returned by an MSSQ that are also returned by thee approximating query) is guaranteed to be 1 for any experimental setting, since Compute_Approx_MSSQ always computes a similarity query. The accuracy

RDF graph		Average accuracy, %		
		dep = 1	dep = 2	dep = 3
LUBM1	A	84.21	91.01	91.12
	C	74.80	88.93	90.59
TFG	A	98.67	98.82	100.00
	C	90.01	90.19	91.28
YAGO	A	90.09	95.33	96.36
	C	87.36	93.85	94.32

Table 7: Average accuracy of answers returned by the approximating queries, computed over acyclic (A) and cyclic (C) pattern graphs and evaluated on the LUBM1, TFG, and YAGO graphs

rates for all three datasets are given in Table 7. As expected, accuracy grows with the increase in the depth value, and it is lower for cyclic patterns as opposed to patterns without cycles. The accuracy values consistently reach above 90%, and the average accuracy across all depth values and both cyclic and acyclic patterns amounts to 86.77%, 94.83%, 92.88% for LUBM1, TFG and YAGO, respectively.

7.3 Case Study

As a proof of concept of effective entity comparison, we ran the COMPUTE_APPROX_MSSQ algorithm on a fragment of DBpedia [68] that captures the information corresponding to Wikipedia *infoboxes* and analysed the types of similarity patterns we were able to compute. The analysis was initially published in [94].

Wikipedia infoboxes are tables with a fixed structure used in Wikipedia to present the key information about entities in a concise and structured way.[4] Infoboxes are located on the right-hand-side of Wikipedia pages that belong to certain categories, such as people, organisations or geographical locations. For these categories entity comparison in Wikipedia could be implemented by directly comparing the entities' infoboxes; such a tool would provide functionality analogous to that in

[4] `https://en.wikipedia.org/wiki/Help:Infobox`

Born	William Bradley Pitt December 18, 1963 (age 53) Shawnee, Oklahoma, U.S.
Occupation	Actor • producer
Years active	1987–present
Works	Filmography
Home town	Springfield, Missouri
Spouse(s)	Jennifer Aniston (m. 2000; div. 2005) Angelina Jolie (m. 2014; separated 2016)
Children	6
Relatives	Douglas Pitt (brother)

Born	Thomas Cruise Mapother IV July 3, 1962 (age 54) Syracuse, New York, U.S.
Occupation	Actor, producer
Years active	1981–present
Spouse(s)	Mimi Rogers (m. 1987; div. 1990) Nicole Kidman (m. 1990; div. 2001) Katie Holmes (m. 2006; div. 2012)
Children	3
Relatives	William Mapother (cousin)
Website	tomcruise.com

Figure 20: Wikipedia infoboxes for actors Brad Pitt (left) and Tom Cruise (right)

existing comparison tools in Web portals (see Chapter 1), in the sense that the features to compare would be considered fixed. However, since data from all infoboxes forms an interconnected graph, the comparison can go beyond two given infoboxes, by recursively comparing entities mentioned in the infoboxes, and thus producing similarity queries beyond depth 1.

Figure 20 displays side by side the infoboxes for two actors Brad Pitt and Tom Cruise.[5] We can observe similarities such as their occupations and country of birth, or the fact that they have both been married and have children. Both infoboxes are fairly well populated, therefore the two corresponding entities form a suitable input for the Compute_Approx_MSSQ algorithm.

We ran our algorithm on Brad Pitt and Tom Cruise and the aforementioned fragment of DBpedia. We observed that the computed MSSQ provided much richer information than what can be obtained by direct inspection of the infoboxes. Since the resulting MSSQ is rather large, we concentrate on its subqueries, i.e., fragments of its basic graph pattern and arithmetic filter condition, of special interest. First, we notice that we generated all the similarities that could be obtained by manual

[5]Screenshots were made on July 24th, 2017.

inspection of the infoboxes. In particular, with $?X$ being the answer variable, we automatically found that both Brad Pitt and Tom Cruise:

- are both actors and producers, as witnessed by the subquery

$$\{(?X, occupation, actor), (?X, occupation, producer)\};$$

- were born in the U.S., as witnessed by

$$\{(?X, birthplace, ?Y_1), (?Y_1, country, us)\};$$

- were married, have at least three kids, and have relatives, as witnessed by

$$\{(?X, children, ?Y_2), (?X, spouse, ?Y_3), (?X, relatives, ?Y_4)\}, \text{ with } (?Y_2 \geq 3).$$

However, the computed MSSQ also contains plenty of additional useful information. For instance, both Pitt and Cruise:

- were married to U.S. actresses, as witnessed by

$$\{(?X, spouse, ?Y_3), (?Y_3, nationality, us), (?Y_3, occupation, actress)\};$$

- were married to actresses who were also married to musicians:

$$\{(?X, spouse, ?Y_3), (?Y_3, occupation, actress), (?Y_3, spouse, ?Y_6), (?Y_6, occupation, musician)\}.$$

To sum up, even using only DBpedia data capturing Wikipedia infoboxes, we are able to significantly enhance the explicit contents of fairly comprehensive infoboxes

and exploit the graph nature of the data to discover "deeper-level" similarities between the entities of interest. We envision that our approach could even be more useful if the whole of DBpedia had been considered, especially in the case where the infoboxes corresponding to the entities of interest are rather minimalist and hence do not provide sufficiently many features to compare, as well as for categories of entities which do not currently have infobox information.

Part IV

Discussion

Chapter 8

Conclusion

In this chapter we are going to give a brief overview of the research presented in this thesis and to summarise main results that have been achieved. We will also discuss directions of future research in the area of declarative entity comparison, as well as a number of potential practical applications of the entity comparison framework.

8.1 Summary

In this thesis we studied the problem of comparing entities using knowledge graphs, and we presented the first declarative, query-based entity comparison framework over knowledge graphs [93, 94]. The framework exploits the information from a KG that is directly or indirectly related to the target entities, and automatically computes meaningful, informative comparisons that have certain guaranteed properties. It is the first framework that approaches comparisons over KGs in a declarative way, and handles differences in an independent way rather than as a by-product or the absence of similarities.

In particular, in this thesis we modeled comparisons as SPARQL queries, and we introduced basic comparison queries — similarity, exact similarity and difference

queries — as well as proposed the notion of the most informative comparison queries. Query informativeness in the framework is measured using subsumption, and most informative similarity and difference queries are defined as the most specific and most general such queries with respect to subsumption [94].

We considered two SPARQL fragments that underlie the framework, namely conjunctive SPARQL queries with or without arithmetic comparisons. Arithmetic comparisons are particularly useful when comparison is done over KGs with rich numeric information: price, size, age and other quantitative parameters. We specifically fixed the form of comparisons to be between a variable and a numeric value, excluding comparisons between variables, since the latter is of little use in the context of entity comparison: it would yield a lot of incomparable cases when, e.g., an age value is compared with a price value. We discussed the advantages and disadvantages of the two query languages, and showed the tradeoff between their complexity and expressive power.

We introduced two fundamental decision problems for queries, namely the existence and verification problems, following the approach by Arenas et al. [8]. The existence decision problem Exists_$\mathcal{X}$ checks whether, given two entities and an RDF graph, there exists a comparison query of type $\mathcal{X}$ that satisfies the input data. The verification problem Verify_$\mathcal{X}$ checks whether, given two entities, an RDF graph and a query, the query is a comparison query of type $\mathcal{X}$ for the given input. We studied the computational complexity of both problems for similarity queries (SQs), exact similarity queries (ESQs), difference queries (DQs) and most specific similarity queries (MSSQs). The complexity is not affected by the presence of arithmetic comparisons, except for the case of verifying MSSQs, which is NP-complete for the queries without the arithmetic filter condition (AFC), and is Π_2^P-complete for the queries with AFC [95]. Similarity queries appeared to be the easiest type of comparison queries from the computational perspective: verifying an SQ is an

NP-complete task, while checking for the existence of an SQ can be efficiently done in AC^0. Exact similarity queries (ESQs) and difference queries (DQs) are similar in nature as they both require two checks, one for the presence of an answer in the answer set and one for the absence of an answer; in fact the complexity of ESQs is shown through the reduction of the corresponding problems for DQs. The existence of both type of queries is a coNP-complete task, while verifying both ESQs and DQs is a DP-complete task. Finally, an MSSQ exists if and only if an SQ exists for the given input, hence the existence problem for MSSQs is in AC^0. The full overview of the complexity results for the existence and verification problems can be found in Table 8.

	SQ	ESQ	DQ	MSSQ
VERIFY_$\mathcal{X}$	NP-c.	DP-c.	DP-c.	NP-c. / Π_2^P-c.
EXISTS_$\mathcal{X}$	in AC^0	coNP-c.	coNP-c.	in AC^0

Table 8: Complexity of VERIFY_$\mathcal{X}$ and EXISTS_$\mathcal{X}$ problems for SQs, ESQs, DQs and MSSQs

We studied procedures for computing different types of similarity queries: a simple AC^0-time check for SQs, the COMPUTE_ESQ algorithm for ESQ, and the COMPUTE_MSSQ algorithm for MSSQs. Furthermore, we demonstrated that if there exists an SQ for two entities in an RDF graph, there must exist a unique MSSQ (modulo query equivalence) for the given input. Lastly, in order to make the computation of MSSQs scalable, we came up with an approximation condition for computing MSSQs and proposed the COMPUTE_APPROX_MSSQ algorithm that can scale up to RDF graphs consisting of millions of triples. We evaluated COMPUTE_APPROX_MSSQ on two real-world datasets YAGO [114] and Twitter follower graph [100], and on the LUBM1 synthetic dataset [50], and demonstrated that COMPUTE_APPROX_MSSQ provides both an efficient and accurate way of computing MSSQs, reaching the empirical approximation ratio of over 90%. Finally, we illustrated the usefulness and expressivity of the framework on numerous examples and

a use case involving Wikipedia infoboxes.

8.2 Future Work

8.2.1 Directions of Research

Declarative entity comparison over knowledge graphs is a novel area of research with numerous potential applications, and not surprisingly it lends itself to multiple additional directions of further research. While this thesis lies foundations for query-based entity comparison, the following research directions are worth investigating from the theoretical perspective.

- One immediate step in extending the framework is to study most general difference queries (MGDQs), their properties and computational complexity. We have demonstrated that there could exist infinitely many MGDQs for two given entities in a knowledge graph. Whether or not the existence and verification problems are decidable for MGDQs remains an open problem. Additionally, one might want to study acyclic MGDQs; the complexity results for them are likely to be decidable. Finally, in order to have a comprehensive entity comparison implementation, it is important to devise efficient algorithms for computing full or approximated MSSQs.

- Another possible extension for the framework is adding the ability to compare non-numeric literals, primarily text strings, but also dates, times, geographic coordinates, etc. Dates and times may be treated similarly to numeric values, i.e., they can be compared using the "earlier than" and "later than" relations that are semantically close to proper numeric comparison operators $\leq$, $<$, $\geq$ and $>$. Text strings, on the other hand, require different comparison methods, e.g., textual similarity methods used in the areas of Natural Language Process-

ing and Information Retrieval [27,34,58,79,80]. For example, given two triples ($book1$, $title$, $'Artifical\ Intelligence\ Basics'$) and ($book2$, $title$, $'Foundation\ of\ AI'$), a useful similarity query would state the two books are on Artificial Intelligence. Geographic coordinates can be compared using distance measures, so that the smaller the distance between two coordinates is, the more similar the two coordinates are.

- Additionally, one might want to study the data-dependent versions of the most informative comparison queries. Our framework uses the notion of data-agnostic query subsumption in the definition of most informative queries: MSSQs and MGDQs are defined as SQs and DQs, respectively, that are minimal and maximal, respectively, with respect to query subsumption. Such modeling, as opposed to the one that uses data-dependent query subsumption, was chosen due to the fact that (web-based) knowledge graphs are inherently incomplete. If we were to use modified definitions of MSSQ and MDGQ in which queries are required to be maximal or minimal with respect to subsumption over the given RDF graph, such MSSQs and MGDQs may no longer be the most informative queries, once the data in the graph is partially modified, added or lost. However, data-dependent MSSQs and MGDQs are nevertheless of considerable interest as they correspond to the patterns that best describe or discriminate the two entities using a particular dataset. Therefore, another potential direction of research is to study the complexity and theoretical properties of such queries.

- Furthermore, the proposed entity comparative framework is not exhaustive in terms of possible types of comparative queries. One could think of additional query types that are relevant for practical entity comparison. One such type is difference queries modulo similarity proposed in [94]: let Q' be a similarity

query for a and b in G, then, we say that Q is a difference query modulo Q' if Q is a difference query for a relative to b, and it holds that $Q \subseteq Q'$. Such DQs are interesting since are relevant to an identified similarity, in the sense that they distinguish the entities based on an aspect that they have in common. For example, given two triples $(Ana, studies, computer\_science)$ and $(Bob, studies, biology)$, one possible similarity is that both persons are students, and differences relevant to this similarity are that Ana studies computer science, while Bob studies biology. Together these SQ and DQ queries offer a relevant, informative comparison of the two entities.

- Finally, to the best of our knowledge, this thesis presents the first declarative entity comparison framework for KGs. However, there have been numerous attempts to come up with similarity measures, including the structural ones that take into account the graph structure of the data, that would probabilistically estimate how similar or dissimilar two entities (objects, text strings, images, etc.) are and produce a numeric score [60, 62, 70, 78, 116, 133]. While such score-based similarities are orthogonal to the way we approach comparison, it would nonetheless be interesting to consider a hybrid framework in which, e.g., certain numeric predictions are integrated into the declarative, query-based comparison. For example, one could statistically predict which nodes should be kept while pruning the similarity tree (see Section 7.1) so as to reduce the size of the output query. And the other way around, one could use the entity comparison framework inside a larger statistical similarity model, e.g., by using comparative queries as features in a machine learning model, or use comparison queries as declarative explanations as to why a pair of entities produced a certain similarity score.

From the practical, user-oriented perspective the entity comparison framework

poses several interesting challenges. In order to create a comprehensive and convenient entity comparison tool, the following things need to be taken into account.

- The most important question is how to present comparison queries to the user and make then readable, concise and reasonably sized. In particular, the following questions may arise.

  · A practical implementation of the tool would effectively address the problem of large-sized comparison queries, in particular MSSQs, and how they can be presented to a user in an easy-to-read manner. One possible solution would be to split the output MSSQs into comprehensible subqueries Such subqueries can then be sorted and ranked according to their informativeness, e.g., using the ranking techniques from the Information Retrieval community [98, 104].

  · Another open problem is output representation. One could use the formal query notation, e.g., the SPARQL syntax or the syntax of CQ. Another solution would involve partially or fully verbalizing MSSQs into natural language explanations. For example, a query $Q = \textsc{Select}\ ?X\ \textsc{Filter}\ \{(?X, \mathit{livesIn}, \mathit{London}), (?X, friendsWith, ?Y), (?Y, worksAt, Oracle)\}$ could be transformed into a natural language explanation "Both input entities live in London and are friends with someone who works at Oracle".

- Finally, when addressing all the aforementioned problems, one should think about how query usefulness and informativeness can be measured and evaluated in practice.

```
Emma_Watson:                        E_Watson:                        Emily_Watson:

Emma_Watson nationality British     E_Watson actedIn Ballet_Shoes    Emily_Watson nationality British
Emma_Watson actedIn Harry_Potter    E_Watson actedIn Little_Women    Emily_Watson actedIn Little_Women1
Emma_Watson actedIn Ballet_Shoes    Little_Women year 2019           Little_Women1 year 2017
```

Figure 21: A fragment of data involving three concepts to be matched.

8.2.2 Potential Applications

Apart from a standalone entity comparison tool which could be used for data exploration and visualisation, declarative entity comparison can be used as an auxiliary module in other tools and applications.

- Entity comparison can be used in ontology and KG matching, in particular for debugging and validation purposes [96]. One way to facilitate entity matching across datasets is to provide human-readable explanations that highlight what the two entities have in common, as well as what differentiates the two entities. For instance, the COMPUTE_APPROX_MSSQ algorithm and the underlying similarity tree can be adapted for such purposes. Suppose there are three entities, Emma_Watson, Emily_Watson and E_Watson, that need to be either matched or disambiguated, and a data fragment given in Figure 21. Then the similarity trees produced by COMPUTE_APPROX_MSSQ over Emma_Watson and E_Watson, and over Emily_Watson and E_Watson will look as depicted in Figure 22. Each path in a similarity tree can be treated as a separate simi-

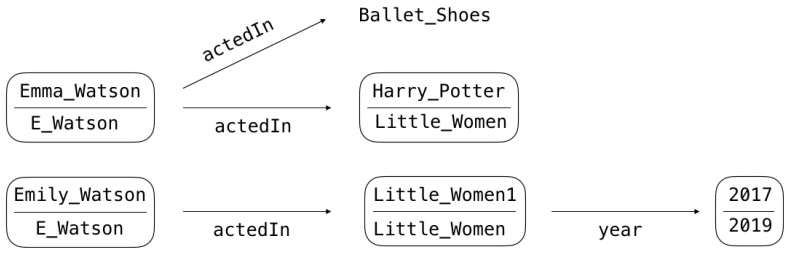


Figure 22: Similarity trees rooted in two pairs of entities.

larity query, in which each edge is encoded as a triple pattern, and each label (L_1, L_2) is encoded as either a fresh variable (if $L_1 \neq L_2$) or an entity l (if $L_1 =$

$L_2 = \{l\}$). For example, a query Q_1 = SELECT $?X$ FILTER $\{(?X, actedIn,$ $Ballet\_Shoes)\}$ is a similarity query for Emma_Watson and E_Watson, while a query Q_2 = SELECT $?X$ FILTER $\{(?X, actedIn, ?Y), (?Y, year, ?Z)\}$ is a similarity query for Emily_Watson and E_Watson. Moreover, each query branch and sub-branch involving non-entity labels can also be treated as a difference query, if instead of variables we take entities from one of the label sets. For example, Q_3 = SELECT $?X$ FILTER $\{(?X, actedIn, Little\_Women), (Little\_Women,$ $year, 2019)\}$ is a difference query for E_Watson relative to Emily_Watson. Both types of queries can assist in explaining why the two entities should or should not be merged: while Q_1 gives a good enough reason to match Emma_Watson and E_Watson into one entity, Q_2 is not specific enough to match the other pair, and in fact Q_3 can act as an indicator that the two movies named Little_Women are indeed two different movies, and Emily_Watson and E_Watson are two difference persons.

- As we have briefly mentioned in Section 8.2.1, comparison queries can provide additional, alternative explanations as to why two entities, e.g., items in an online shop or movies in a recommender system, are considered to be statistically similar. The explainability of complex machine learning-based systems is an open problem and an active area of research, and semantic technologies, in particular knowledge graphs, lend themselves for the task [45, 105, 123]. It has been partially implemented in social networks like Facebook or LinkedIn, which suggest users their potential acquaintances motivating the suggestions by something the two users have in common, e.g., friends, a university, a workplace or a hometown. Using the full expressive power of formalisms like SPARQL or CQs would considerably enrich such explanations and add transparency to any system. However, integrating declarative explanations into machine learning-based systems does not stop at social networks. Quite

the contrary, explainable predictions, suggestions and generated content can greatly benefit, respectively, financial forecasting, e-commerce, as well as dialog systems, news summarisation and artificial personal assistants, to name a few. Explainable results are especially important in the domains like biomedicine, jurisprudence, insurance or public policies, where the decisions of an artificial system affect people's lives and can be critical. This is why hybrid systems that combine traditional statistical methods with symbolic representations and that offer higher degrees of explainability, interpretability and trust are the very likely future of AI [66, 74, 124, 129], and entity comparison offers one way of integrating explanations into statistical reasoning.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: The logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] M. Al Hasan and M. J. Zaki. A survey of link prediction in social networks. In *Social network data analytics*, pages 243–275. Springer, 2011.

[3] R. Angles and C. Gutierrez. The expressive power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference (ISWC)*, pages 114–129. Springer, 2008.

[4] D. Angluin. Inductive inference of formal languages from positive data. *Information and control*, 45(2):117–135, 1980.

[5] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[6] D. Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.

[7] T. Antonopoulos, F. Neven, and F. Servais. Definability problems for graph query languages. In *Proceedings of the 16th International Conference on Database Theory (ICDT)*, pages 141–152. ACM, 2013.

[8] M. Arenas, G. I. Diaz, and E. V. Kostylev. Reverse engineering SPARQL queries. In *Proceedings of the 25th International World Wide Web Conference (WWW)*, pages 239–249, 2016.

[9] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *Proceedings of the 13th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 305–316. ACM, 2011.

[10] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. *Online proceedings of the 1st International Workshop on Usage*

Analysis and the Web of Data (USEWOD) co-located with the 20th International World Wide Web Conference (WWW), abs/1103.5043, 2011.

[11] S. Arora and B. Barak. *Computational complexity: A modern approach*. Cambridge University Press, 2009.

[12] F. Baader and A.-Y. Turhan. On the problem of computing small representations of least common subsumers. In *Proceedings of the 25th Annual German Conference on Artificial Intelligence (KI)*, pages 99–113, 2002.

[13] P. Barceló, R. Pichler, and S. Skritek. Efficient evaluation and approximation of well-designed pattern trees. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 131–144. ACM, 2015.

[14] P. Barceló and M. Romero. The complexity of reverse engineering problems for conjunctive queries. In *Proceedings of the 20th International Conference on Database Theory (ICDT)*, pages 7:1–7:17, 2017.

[15] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. RDF 1.1 turtle – terse RDF triple languageturtle. *World Wide Web Consortium. https://www.w3.org/TR/turtle/ (last visited Sep 12, 2019)*, 2014.

[16] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2RDF: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5):706–716, 2008.

[17] C. Berge. *Graphs and hypergraphs*. Amsterdam, The Netherlands: North-Holland Pub. Co., 1973.

[18] T. Berners-Lee. Linked data. Design Issues. World Wide Web Consortium. https://www.w3.org/DesignIssues/LinkedData.html (last visited Oct 27, 2019), 2006.

[19] T. Berners-Lee and M. Fischetti. *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. DIANE Publishing Company, 2001.

[20] T. Berners-Lee, L. Masinter, and R. T. Fielding. Uniform resource identifier (URI): Generic syntax, January 2005. *STD 66, RFC 3986. https://www.rfc-editor.org/info/rfc3986 (last visited Sept 12, 2019)*, 2005.

[21] A. Bernstein, J. Hendler, and N. Noy. A new look at the semantic web. *Communications of the ACM*, 59(9):1–5, 2016.

[22] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1247–1250. ACM, 2008.

[23] B. Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 2013.

[24] A. Bonifati, R. Ciucanu, and A. Lemay. Learning path queries on graph databases. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT)*, pages 109–120, 2015.

[25] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of computing (STOC)*, pages 77–90. ACM, 1977.

[26] G. Cheng, Y. Zhang, and Y. Qu. Explass: exploring associations between entities via top-K ontological patterns and facets. In *Proceedings of the 13th International Semantic Web Conference (ISWC)*, pages 422–437, 2014.

[27] S.-S. Choi, S.-H. Cha, and C. C. Tappert. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, 8(1):43–48, 2010.

[28] S. Cohen and Y. Y. Weiss. Learning tree patterns from example graphs. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 31. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2015.

[29] W. W. Cohen, A. Borgida, H. Hirsh, et al. Computing least common subsumers in description logics. In *Proceedings of the 10th National Conference on Artificial intelligence (AAAI)*, volume 1992, pages 754–760, 1992.

[30] S. Colucci, F. M. Donini, and E. Di Sciascio. Common subsumbers in RDF. In *Congress of the Italian Association for Artificial Intelligence*, pages 348–359. Springer, 2013.

[31] S. Colucci, F. M. Donini, S. Giannini, and E. Di Sciascio. Defining and computing least common subsumers in RDF. *Journal of Web Semantics*, 39:62–80, 2016.

[32] S. Colucci, S. Giannini, F. M. Donini, and E. Di Sciascio. Finding commonalities in Linked Open Data. In *Proceedings of the 29th Italian Conference on Computational Logic (CILC)*, pages 324–329, 2014.

[33] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of computing (STOC)*, pages 151–158. ACM, 1971.

[34] C. Corley and R. Mihalcea. Measuring the semantic similarity of texts. In *Proceedings of the ACL 2005 Workshop on Empirical Modeling of Semantic Equivalence and Entailment*, pages 13–18. Association for Computational Linguistics, 2005.

[35] N. Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and brain sciences*, 24(1):87–114, 2001.

[36] R. Cyganiak. A relational algebra for SPARQL. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, 35, 2005.

[37] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 concepts and abstract syntax. W3C proposed recommendation, 25 February 2014. *World Wide Web Consortium. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (last visited Oct 1, 2016)*, 2014.

[38] G. Diaz, M. Arenas, and M. Benedikt. SPARQLByE: Querying RDF data by example. *Proceedings of the VLDB Endowment*, 9(13), 2016.

[39] R. Diestel. *Graph Theory*, volume 173. Graduate texts in mathematics, 2012.

[40] L. Ehrlinger and W. Wöß. Towards a definition of knowledge graphs. *Joint proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems (SEMANTiCS) and the 1st International Workshop on Semantic Change and Evolving Semantics (SuCCESS) co-located with the 12th International Conference on Semantic Systems (SEMANTiCS)*, 48, 2016.

[41] S. El Hassad, F. Goasdoué, and H. Jaudoin. Learning commonalities in SPARQL. In *Proceedings of the 16th International Semantic Web Conference (ISWC)*, pages 278–295, 2017.

[42] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM (JACM)*, 30(3):514–550, 1983.

[43] M. Färber. The Microsoft Academic Knowledge Graph: A Linked Data source with 8 billion triples of scholarly data. In *Proceedings of the 18th International Semantic Web Conference (ISWC)*, pages 113–129. Springer, 2019.

[44] A. Ferrara, A. Nikolov, J. Noessner, and F. Scharffe. Evaluation of instance matching tools: The experience of OAEI. *Journal of Web Semantics*, 21:49–60, 2013.

[45] M. H. Gad-Elrab, D. Stepanova, J. Urbani, and G. Weikum. ExFaKT: A framework for explaining facts over knowledge graphs and text. In *Proceedings of the 12th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 87–95. ACM, 2019.

[46] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book (2nd Edition)*. Pearson, 2 edition, 6 2008.

[47] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 29. W. H. Freeman and Company, New York, 2002.

[48] L. Getoor and C. P. Diehl. Link mining: A survey. *ACM SIGKDD Explorations Newsletter*, 7(2):3–12, 2005.

[49] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM (JACM)*, 48(3):431–498, 2001.

[50] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[51] V. Gutiérrez-Basulto, J. C. Jung, and L. Sabellek. Reverse engineering queries in ontology-enriched systems: The case of expressive horn description logic ontologies. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1847–1853, 2018.

[52] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C proposed recommendation, 21 March 2013. *World Wide Web Consortium. https://www.w3.org/TR/sparql11-query/ (last visited Oct 1, 2016)*, 2013.

[53] P. J. Hayes and P. F. Patel-Schneider. RDF 1.1 semantics. *World Wide Web Consortium. https://www.w3.org/TR/rdf11-mt/ (last visited Oct 26, 2019)*, 2014.

[54] P. Heim, S. Hellmann, J. Lehmann, S. Lohmann, and T. Stegemann. RelFinder: Revealing relationships in RDF knowledge bases. In *Proceedings of the 4th International Conference on Semantic and Digital Media Technologies (SAMT)*, pages 182–187, 2009.

[55] P. Hell and J. Nešetřil. On the complexity of H-coloring. *Journal of Combinatorial Theory, Series B*, 48(1):92–110, 1990.

[56] P. Hell and J. Nešetřil. The core of a graph. *Discrete Mathematics*, 109(1):117–126, 1992.

[57] P. Hitzler, M. Krotzsch, and S. Rudolph. *Foundations of semantic web technologies*. CRC Press, 2009.

[58] A. Huang. Similarity measures for text document clustering. In *Proceedings of the 6th New Zealand Computer Science Research Student Conference (NZCSRSC2008)*, pages 49–56, 2008.

[59] P. James. Knowledge graphs. *Linguistic Instruments in Knowledge Engineering*, pages 97–117, 1992.

[60] G. Jeh and J. Widom. SimRank: a measure of structural-context similarity. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 538–543. ACM, 2002.

[61] L. Ji, Y. Wang, B. Shi, D. Zhang, Z. Wang, and J. Yan. Microsoft Concept Graph: Mining semantic concepts for short text understanding. *Data Intelligence*, 1(3):238–270, 2019.

[62] S. Joshi, N. Agrawal, R. Krishnapuram, and S. Negi. A bag of paths model for measuring structural similarity in web documents. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 577–582. ACM, 2003.

[63] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM (JACM)*, 35(1):146–160, 1988.

[64] P. G. Kolaitis. Lecture notes in Principles of database systems. UC Santa Cruz. https://courses.soe.ucsc.edu/courses/cmps277/Fall11/01 (last visited Aug 10, 2018), 2011.

[65] M. Kroetsch and G. Weikum. Special issue on knowledge graphs. *Journal of Web Semantics*, 4:2018, 2015.

[66] F. Lecue. On the role of knowledge graphs in explainable AI. *Semantic Web*, 11(1):41–51, 2020.

[67] F. Lehmann. *Semantic networks in artificial intelligence*. Elsevier Science Inc., 1992.

[68] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia — a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web Journal*, 6(2):167–195, 2014.

[69] J. Lehmann, J. Schüppel, and S. Auer. Discovering unknown connections — the DBpedia relationship finder. *Proceedings of the 1st Conference on Social Semantic Web (CSSW)*, 113:99–110, 2007.

[70] E. A. Leicht, P. Holme, and M. E. Newman. Vertex similarity in networks. *Physical Review E*, 73(2):026120, 2006.

[71] L. A. Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[72] M. Ley. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 1–10. Springer, 2002.

[73] M. Ley. DBLP: some lessons learned. *Proceedings of the VLDB Endowment*, 2(2):1493–1500, 2009.

[74] W. Li, G. Qi, and Q. Ji. Hybrid reasoning in knowledge graphs: Combing symbolic reasoning and statistical reasoning. *Semantic Web*, 11(1):53–62, 2020.

[75] F. Mahdisoltani, J. Biega, and F. M. Suchanek. YAGO3: A knowledge base from multilingual Wikipedias. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[76] G. Marchionini. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.

[77] V. Martínez, F. Berzal, and J.-C. Cubero. A survey of link prediction in complex networks. *ACM Computing Surveys (CSUR)*, 49(4):1–33, 2016.

[78] L. Meng, R. Huang, and J. Gu. A review of semantic similarity measures in Wordnet. *International Journal of Hybrid Information Technology*, 6(1):1–12, 2013.

[79] D. Metzler, S. Dumais, and C. Meek. Similarity measures for short segments of text. In *Proceedings of the 29th European Conference on Information Retrieval (ECIR)*, January 2007.

[80] R. Mihalcea, C. Corley, and C. Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, volume 6, pages 775–780, 2006.

[81] M. Minsky. *A framework for representing knowledge.* Massachusetts Institute of Technology, 1974.

[82] T. Mitchell. *Machine learning.* McGraw-Hill Education, 1997.

[83] E. Najmi, Z. Malik, K. Hashmi, and A. Rezgui. ConceptRDF: An RDF presentation of ConceptNet knowledge base. In *Proceedings of the 7th International Conference on Information and Communication Systems (ICICS)*, pages 145–150. IEEE, 2016.

[84] M. Nentwig, M. Hartung, A.-C. Ngonga Ngomo, and E. Rahm. A survey of current link discovery frameworks. *Semantic Web*, 8(3):419–436, 2017.

[85] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, and J. Taylor. Industry-scale knowledge graphs: Lessons and challenges. *Queue*, 17(2):20, 2019.

[86] S. Nurdiati and C. Hoede. 25 years development of knowledge graph theory: The results and the challenge. *Memorandum*, 1876, 2008.

[87] W. Nutt. Lecture notes in Ontology and database systems: Foundations of database systems. Free University of Bozen-Bolzano. http://www.inf.unibz.it/~nutt/Teaching/ODBS1314/ODBSSlides/3-conjQueries.pdf (last visited Oct 22, 2018), 2013.

[88] C. H. Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[89] J. Park and S.-g. Lee. Keyword search in relational databases. *Knowledge and Information Systems*, 26(2):175–193, 2011.

[90] H. Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508, 2017.

[91] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, pages 30–43. Springer, 2006.

[92] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

[93] A. Petrova. Comparing entities in RDF graphs. In *Proceedings of the VLDB 2017 PhD Workshop co-located with the 43rd International Conference on Very Large Data Bases (VLDB)*, 2017.

[94] A. Petrova, E. Sherkhonov, B. Cuenca Grau, and I. Horrocks. Entity comparison in RDF graphs. In *Proceedings of the 16th International Semantic Web Conference (ISWC)*, pages 526–541, 2017.

[95] A. Petrova, E. V. Kostylev, B. Cuenca Grau, and I. Horrocks. Query-based entity comparison in knowledge graphs revisited. In *Proceedings of the 18th International Semantic Web Conference (ISWC)*, pages 558–575, 2017.

[96] A. Petrova, E. V. Kostylev, B. Cuenca Grau, and I. Horrocks. Towards explainable entity matching via comparison queries. In *Proceedings of the 14th International Workshop on Ontology Matching co-located with the 18th International Semantic Web Conference (ISWC)*, 2019.

[97] G. D. Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1):153–163, 1970.

[98] J. Ramos. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the 1st instructional Conference on Machine Learning (iCML)*, volume 242, pages 133–142, 2003.

[99] R. H. Richens. Preprogramming for mechanical translation. *Mechanical Translation*, 3(1):20–25, 1956.

[100] R. A. Rossi and D. F. Gleich. A dynamical system for PageRank with time-dependent teleportation. *Internet Mathematics*, 10(1-2), 2014.

[101] S. J. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Malaysia; Pearson Education Limited, 2016.

[102] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)*, 27(4):633–655, 1980.

[103] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory (ICDT)*, pages 4–33. ACM, 2010.

[104] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to information retrieval*. Cambridge University Press, 2008.

[105] A. Seeliger, M. Pfaff, and H. Krcmar. Semantic web technologies for explainable machine learning models: A literature review. *Joint proceedings of the 6th International Workshop on*

Dataset PROFILing and Search (PROFILES) and the 1st Workshop on Semantic Explainability (SEMEX) co-located with the 18th International Semantic Web Conference (ISWC), pages 30–45, 2019.

[106] J. F. Sequeda and C. Gutierrez. A brief history of Knowledge Graph's main ideas: A tutorial. A tutorial at the 18th International Semantic Web Conference (ISWC). http://knowledgegraph.today/paper.html (last visited Oct 27, 2019), 2019.

[107] E. Sherkhonov and M. Marx. Containment of acyclic conjunctive queries with negated atoms or arithmetic comparisons. *Information Processing Letters*, 120:30–39, 2017.

[108] A. Singhal. Introducing the Knowledge Graph: Things, not strings. Official Google blog. https://www.blog.google/products/search/introducing-knowledge-graph-things-not/ (last visited Oct 27, 2019), 2012.

[109] M. Sipser et al. *Introduction to the theory of computation*, volume 2. Thomson Course Technology Boston, 2006.

[110] J. F. Sowa. *Semantic networks*. Citeseer, 1987.

[111] R. Speer, J. Chin, and C. Havasi. ConceptNet 5.5: An open multilingual graph of general knowledge. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, pages 4444–4451, 2017.

[112] S. Staworko and P. Wieczorek. Learning twig and path queries. In *Proceedings of the 15th International Conference on Database Theory (ICDT)*, pages 140–154. ACM, 2012.

[113] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[114] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A large ontology from Wikipedia and Wordnet. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3):203–217, 2008.

[115] B. ten Cate and V. Dalmau. The product homomorphism problem and applications. In *Proceedings of the 18th International Conference on Database Theory (ICDT)*, pages 161–176, 2015.

[116] K. Todorov and P. Geibel. Ontology mapping via structural and instance-based similarity measures. In *Proceedings of the 7th International Semantic Web Conference (ISWC)*, page 224, 2008.

[117] B. Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. *Doklady AN SSR*, 70(4):569–572, 1950.

[118] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 535–548. ACM, 2009.

[119] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query reverse engineering. *The VLDB Journal—The International Journal on Very Large Data Bases*, 23(5):721–746, 2014.

[120] A.-Y. Turhan. *Reasoning Services for the Maintenance and Flexible Access to Description Logic Ontologies*. TU Dresden, 2014.

[121] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 331–345. ACM, 1992.

[122] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of the 14th Annual ACM Symposium on Theory of computing (STOC)*, pages 137–146. ACM, 1982.

[123] N. Voskarides, E. Meij, M. Tsagkias, M. De Rijke, and W. Weerkamp. Learning to explain entity relationships in knowledge graphs. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL-IJCNLP)*, page 11, 2015.

[124] X. Wang, D. Wang, C. Xu, X. He, Y. Cao, and T.-S. Chua. Explainable reasoning over knowledge graphs for recommendation. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, volume 33, pages 5329–5336, 2019.

[125] Y. Y. Weiss and S. Cohen. Reverse engineering SPJ-queries from examples. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 151–166, 2017.

[126] D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ, 1996.

[127] R. W. White and R. A. Roth. Exploratory search: Beyond the query-response paradigm. *Synthesis lectures on information concepts, retrieval, and services*, 1(1):1–98, 2009.

[128] R. Willard. Testing expressibility is hard. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 9–23. Springer, 2010.

[129] Y. Xian, Z. Fu, S. Muthukrishnan, G. De Melo, and Y. Zhang. Reinforcement knowledge graph reasoning for explainable recommendation. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 285–294, 2019.

[130] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB)*, volume 81, pages 82–94, 1981.

[131] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Engineering Bulletin*, 33(1):67–78, 2010.

[132] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 809–820, 2013.

[133] P. Zhao, J. Han, and Y. Sun. P-Rank: a comprehensive structural similarity measure over information networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, pages 553–562. ACM, 2009.

[134] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.