Harvesting commonsense and hidden knowledge from web services

Julien Romero

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2020IPPAT032

Thèse de doctorat

TELECOM Paris

IP PARIS

Harvesting Commonsense and Hidden Knowledge From Web Services

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626
École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat: Computing, Data and Artificial Intelligence

Thèse présentée et soutenue à Palaiseau, le 5 Octobre 2020, par

Julien Romero

Composition du Jury :

Pierre Senellart Professor, École Normale Supérieure	Président
Tova Milo Professor, Tel Aviv University	Rapporteur
Katja Hose Professor, Aalborg University	Rapporteur
Michael Benedikt Professor, University of Oxford	Examinateur
Andrea Calì Professor, University of London, Birkbeck College	Examinateur
Meghyn Bienvenu Full-Time CNRS Researcher, University of Bordeaux (LaBRI)	Examinateur
Fabian Suchanek Professor, Télécom Paris	Directeur de thèse
Nicoleta Preda Associate Professor, University of Versailles	Co-directeur de thèse
Antoine Amarilli Associate Professor, Télécom Paris	Invité

Harvesting Commonsense and Hidden Knowledge From Web Services

Julien Romero

5 Octobre 2020

À mon grand-père

Abstract

In this thesis, we harvest knowledge of two different types from online resources. The first one is commonsense knowledge, i.e. intuitive knowledge shared by most people like "the sky is blue". We extract salient statements from query logs and question-answering sites by carefully designing question patterns. Next, we validate our statements by querying other web sources such as Wikipedia, Google Books, or image tags from Flickr. We aggregate these signals to create a final score for each statement. We obtain a knowledge base, *Quasimodo*, which, compared to its competitors, has better precision and captures more salient facts.

The other kind of knowledge we investigate is hidden knowledge, i.e. knowledge not directly given by a data provider. More concretely, some Web services allow accessing the data only through predefined access functions. To answer a user query, we have to combine different such access functions, i.e. we have to rewrite the query in terms of the functions. We study two different scenarios: In the first scenario, the access functions have the shape of a path, the knowledge base respects constraints called "Unary Inclusion Dependencies", and the query is atomic. We show that the problem is decidable in polynomial time, and we provide an algorithm with theoretical evidence. In the second scenario, we remove the constraints and create a new class of relevant plans called "smart plans". We show that it is decidable to find these plans, and we provide an algorithm.

Remerciements

On ne prend jamais assez de temps pour remercier les gens qui nous sont chers et qui nous aident à aller de l'avant. Un simple merci me paraît trop ordinaire, mais nul autre mot ne semble assez fort. Toutes ces personnes donnent un sens à nos actions et à nos choix: elles mériteraient autant que moi de figurer sur la première page de cette thèse.

Bien sûr, il me serait impossible de ne pas parler de mes deux directeurs de thèse, Nicoleta et Fabian, sans qui aucun mot figurant ici n'aurait été possible. Venir faire ma thèse avec eux a complètement changé ma vie de bien des manières. Ils m'ont permis de m'épanouir librement et je sens qu'avec eux, je suis allé bien plus loin qu'un simple doctorat.

Je remercie chaleureusement les rapporteuses Tova Milo et Katja Hose, ainsi que tous les membres du jury, Pierre Senellart, Michael Benedikt, Andrea Calì et Meghyn Bienvenu.

Un été de ma thèse s'est déroulé au Max Planck Institute for Informatics à Saarbrucken, et je remercie Gehard Weikum de m'avoir accueilli dans son équipe, ainsi que Simon Razniewski et Koninika Pal qui m'ont grandement aidé dans mes travaux de recherche.

Sans ma famille, jamais je ne serais arrivé jusqu'à la thèse. Ma mère, Catherine, a tout sacrifié pour moi, malgré les problèmes que nous avons rencontrés. Tout ce que je sais, c'est grâce à elle. Elle m'a fait découvrir la musique et les arts, elle a passé d'innombrables heures à m'accompagner dans mon éducation, elle a supporté mes études, ... Jamais je ne pourrais assez la remercier.

Mon arrière-grand-mère a aussi beaucoup fait pour moi. Aujourd'hui encore, je me souviens que c'est elle qui m'a appris à lire. J'ai passé de nombreuses heures devant sa bibliothèque à éplucher chaque livre et devant sa télé à regarder *C'est pas sorcier* et *les gendarmes de Saint-Tropez*.

J'ai beaucoup passé de temps avec mes grands-parents, Firmin et Monique, surtout durant les étés au bord de la mer. Ils m'ont toujours aidé dans tout ce que j'ai entrepris et j'espère que de là où il est, mon grand-père est fier de moi.

Il y a aussi Thomas et Mélanie, mon frère et ma sœur qui ont traversé avec moi les tourments de l'enfance, mon beau-père, Jacques, qui s'est montré très aimant envers ma famille et enfin, je n'oublie pas ma tante Laurence et mon oncle Yann qui ont toujours cherché à aiguiser ma curiosité.

Finalement, en même temps que j'ai commencé cette thèse, j'ai agrandi ma famille un peu plus. La plus grande de mes découvertes s'appelle Oana, et j'espère bien passer le reste de ma vie avec elle. Elle a toujours été à mes côtés au cours de ces trois dernières années, et m'a aidé à surmonter toutes les difficultés. Je sens

qu'à ses côtés rien ne peut m'arriver.

Je pense aussi à tous les proches amis que j'ai pu avoir et avec qui j'ai passé de très bons moments. Je ne pourrai tous les nommer ici, mais je citerai Didier et tous les projets un peu fous de nous avons pu avoir, Jérémie et Daniel, qui ont partagé ma chambre en prépa et Victor, qui ne m'a pas quitté depuis la maternelle.

La recherche ne se fait pas tout seul dans une chambre. DIG m'a accueilli pendant toute ma thèse, et je tiens à remercier tous ses membres: Albert, Armand, Arnaud, Camille, Etienne, Favia, Jacob, Jean-Benoît, Jean-Louis, Jonathan, Julien, Lihu, Louis, Marc (merci de m'avoir aidé dans tant de projets), Marie, Maroua, Mauro, Mikaël, Miy-oung, Mostafa, Nathan, Ned, Nicolas, Pierre-Alexandre, Quentin, Quentin, Samed, Talel, Thomas (le chef), Thomas (le grand frère doctorant), Thomas (le frère jumeau doctorant, qui a répondu à bien nombre de mes questions) et Ziad. Je remercie particulièrement Antoine pour son aide précieuse sur bien des sujets. Merci aussi à tous les enseignants de Télécom et à tous mes élèves qui m'ont fait aimer transmettre mes connaissances.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Information is not knowledge.
>
> *Albert Einstein*

1.1 Motivation

1.1.1 Information Jungle

Last century, the Human race entered the *Information age* [29]. This historical period started with the invention of the transistor and the first computers. However, it is the emergence of the Internet in the 70s, and more particularly of the World Wide Web [17] in 1991, that pushed Mankind into the Information Jungle.

In January 2000, the Web consisted of around 10 million unique domain names. In January 2010, this number reached 100 million and in January 2020, 1.3 billion (https://news.netcraft.com/archives/category/web-server-survey/). This exponential growth follows the development of computational power and storage capacities. The Internet Live Stats (https://www.internetlivestats.com/) provides insights that show how important the Internet has become. Currently, there are:

- more than 4.5 billion people connected to the Internet
- nearly 3 million emails sent per second (67% of them are spam)
- around 85,000 videos viewed on YouTube per second
- more than 80,000 queries on Google per second
- around 9,000 tweets per second
- about 97,000 GB of Internet Traffic per second

Still, these numbers are only the visible part of the iceberg. Many Internet applications are not indexed by search engines and form what we call the Deep Web [16]. Among these applications, we find, for example, webmails, banking applications,

web sites with restricted access (like Netflix, Facebook or some newspapers) or applications with a private domain name (such as an IP or a hostname not indexed in traditional DNS).

There are two kinds of users on the Internet: humans and machines. Machines exchange structured information between them or provide services: these are what we call the Web Services. People use the Internet for two primary goals which are entertainment and finding information. In the first case, they usually spend time on social media platforms (more than 3.8 billion people are on social media like Facebook, Twitter or Reddit) or watch videos (on YouTube or Netflix for example). In the second case, they generally use search engines to filter the complexity of the Web.

However, even when using search engines, it is not very easy for people to turn the amount of information into knowledge. Some websites are specialised in knowledge-gathering. The most prominent example is Wikipedia, which contains articles about six million subjects. Nevertheless, outside this "safe zone", knowledge is lost into an ocean of conflicting and complicated information. There are several reasons for this problem.

First, specialised knowledge is rare. Even when requested explicitly on question-answering forums, people can sometimes provide incorrect answers to questions. Some websites gather experts and allow the community to grade solutions to prevent potential mistakes. This is the case for technical questions in the Stack Exchange Network and more general questions on question answering forums (like Quora or Reddit). However, it is quite common to observe wrong content, especially on unfashionable subjects where there are few experts.

Second, people may willingly put wrong information on the Internet. In the domain of news, this is what we call *Fake News* (word of the year in 2017 for the Collins dictionary). There are multiple goals behind such behaviour. Some people just want to propagate their ideas. However, there are also cases where the attractiveness motivates the diffusion: false and polemic news generate more visits than correct ones. The former journalist Paul Horner [118] became famous for creating many such websites, publishing only fake news to make money through advertisement. Some parody web sites are specialised in funny news (like the Gorafi [71] in France). The sad consequence is that some politicians diffuse such comic news as if they were true [64].

The problem of fake news also exists in mainstream media [86]. In many cases, the authors of the articles do not care about the truth: The goal is to produce content that gets the interest of readers. The philosopher Harry Frankfurt [66] called it "Bullshit" and distinguished it from lying because the truth is not the central point. Sébastien Dieguez [42] applied this concept to our current situation and explained how "Bullshit" was established as an institution on a global scale.

1.1.2 Knowledge Bases Structure Information

The Internet illustrates perfectly the dichotomy between information and knowledge. On the one hand, information takes many forms, is noisy, can be either true or false and is abundant. On the other hand, knowledge is structured, precise, focuses on

truth and has limits. However, the two work in synergy and complement each other.

For example, a priori knowledge helps to make sense of information as many things are implicit, incomplete and potentially incorrect. For example, let us consider the BBC headline: "New York begins reopening after lockdown". To understand it, we need additional knowledge:

- New York was locked down due to Covid-19
- New York is a major city in the U.S.
- Covid-19 is a pandemic that killed many people
- Covid-19 is disappearing
- The reopening of New York means that people can go out of their house to work and shops
- People need to work and go shopping
- The lockdown was a difficult period
- ...

With this knowledge, most humans can reason about the headline and draw conclusions. If we want a computer to do the same, we have to provide this background knowledge in a computer-readable form through a knowledge base, i.e. a technology that gathers "knowledge".

1.1.3 A Brief History of Knowledge Bases

Before we continue, it is essential to identify two kinds of knowledge. The first one is encyclopedic knowledge, which we learn at school. For example, when studying geography, we learn that the capital of France is Paris. The other type is commonsense knowledge, which we learn intuitively throughout our life. For example, we all know that if we drop an object, it falls. Yet, nobody taught it to us. This last kind of knowledge is tough to get for a computer as people universally share it, never mention it but use it all the time. Sometimes, the separation between encyclopedic and commonsense knowledge is blurry, especially when they are mixed in a cultural context. For example, every French person knows that a corkscrew opens a bottle of wine. However, in some countries where people do not drink alcohol, this might be known by much fewer people.

Knowledge bases emerged with the creation of Artificial Intelligence in the 80s. Interestingly, the first main systems focused on *commonsense knowledge*, which seemed to be the key to break human intelligence. The first notable work is Cyc [79]. This long-term project began in 1984 under the direction of Douglas Lenat at the Microelectronics and Computer Technology Corporation. The goal was to gather as much knowledge as possible, which other expert systems can then use. Cyc relies heavily on specialised human labour, and the currently open version OpenCyc4.0 contains two million facts. Besides, Cyc comes with an inference engine that allows reasoning over the knowledge base.

WordNet [87] is another notable early work. This knowledge base was created in 1985 by the Cognitive Science Laboratory of Princeton University. It focuses on lexical and semantic properties of words and the relations between them. Like Cyc, it was handcrafted by specialised humans.

With the rise of the Web at the end of the 90s, more scalable ways of gathering knowledge appeared. In 1999 at the MIT, the Open Mind Common Sense project [117] began and allowed people from all around the world to contribute to gathering commonsense knowledge. During the first year of the project, it was able to collect more than one million facts from 15,000 contributors. Later, ConceptNet [119] emerged from Open Mind Common Sense by structuring the available data in a more usable form. Besides, the current versions of ConceptNet leverage additional sources like DBpedia, Wiktionary, WordNet and OpenCyc.

At the beginning of the millennium, the first automatically-generated knowledge bases appeared. In 2004, KnowItAll [59] from the University of Washington used search engines and carefully designed queries to collect knowledge and was able to produce 55 thousand facts in four days. However, the most notable knowledge bases emerged thanks to the Wikipedia project, which started the era of extensive encyclopedic knowledge bases.

Wikipedia was created in 2001 by Jimmy Wales and Larry Sanger and is now the largest encyclopedia ever created. It contains 50 million pages, written in 285 different languages by 40 million people all around the world. Between the years 2005 and 2008, the number of contributions exploded and thus, many automated systems tried to leverage this valuable information.

In 2007, the three projects DBpedia (FU Berlin, U Mannheim, U Leipzig, [6]), Freebase (Metaweb, [130]) and Yago (Telecom Paris, Max Planck Institute, [123]) emerged. They are all based on the same principle: automatically extract and structure facts from diverse sources, the main one being Wikipedia. The principal goal of these projects is to gather *encyclopedic knowledge*. Thanks to the automated methods, the number of facts exploded: 9.5 billion for DBpedia, 2.4 billion for Freebase, and 2 billion for Yago. In 2012, La Sapienza released BabelNet [90] which merges Wikipedia with Wordnet to create a multilingual lexicalised semantic network and knowledge base.

In 2010, the Never-Ending Language Learning [28] (NELL) system wanted to expand the automatic extraction to the entire Web. Starting from a seed knowledge base, the project automatically collected new facts from websites. Although the noisiness of the data did not allow a precision as high as Wikipedia-based systems, NELL was able to accumulate over 50 million candidate beliefs on the Web, among which nearly three million are considered highly accurate.

In 2012, a significant project derived from Wikipedia and with similar principles appeared: Wikidata [131]. It is partially generated automatically by robots and manually by contributors, similarly to Wikipedia. This knowledge base currently contains around 1 billion facts. Due to the rise of popularity and quality of Wikidata, Yago decided in its fourth version to stop extracting facts from Wikipedia and to use only Wikidata [128].

Finally, knowledge bases also attracted the interest of industrial groups. Although their content is quite opaque, they are worth mentioning. Google created

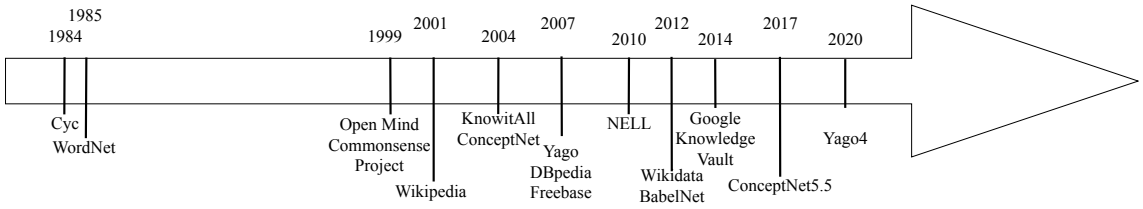

Figure 1.1: Timeline of the history of knowledge bases

the Knowledge Vault [44] partially constructed from Freebase, which they acquired in 2010. Other famous companies also have their knowledge bases: Microsoft's Satori, Amazon's Evi, LinkedIn's Knowledge Graph, and the IBM Watson KB [62].

Most of the main open-access knowledge bases are interlinked and form the Web of Linked Open Data [19]. In Figure 1.1, we summarise the main knowledge bases.

1.1.4 Applications

As knowledge bases contain crisp information about the real world, they have many applications, mainly centred around interaction with human beings.

Search Engines. Most search engines have a knowledge base running in the background, which helps them to interpret and answer queries. It is a crucial task to understand what a query is about when a user submits a query to a search engine. However, human language is highly ambiguous, and so, the task of guessing the subject of a query can be difficult. For example, when a user asks for "Paris train station", it is primordial to interpret Paris as the capital of France, and not as Paris Hilton or as the Greek Hero. In many cases, knowledge bases provide enough clues: the city of Paris is the only subject connected to a train station. Once the search engine knows what the topic of the query is, it can print additional information about it. Thus, when typing *Georges Brassens* on Google or Bing, an infobox provides information such as his job title, his size, his most famous songs, etc. Google uses the closed-access *Knowledge Vault* [44], which is partially powered by Freebase [130]. Bing has a similar knowledge base, which can be accessed partially by the Bing Entity Search API (https://blogs.bing.com/search-quality-insights/2018-02/bing-entity-search-api-now-generally-available).

Question Answering. Another area for which knowledge bases seem perfectly designed is question answering, in particular for questions about facts. When performing Question Answering over Knowledge bases [41], the key idea is to analyse the question to extract the subject and the kind of information required. Then, one can create a tailored request to a knowledge base to retrieve potential answers. Then, a classifier can weight all the answers before providing a final solution. Most search engines can answer questions, and other systems like Wolfram Alpha (https://www.wolframalpha.com/) also provide such functionality. One of the most popular software is IBM's Watson [62]: In 2011, it won the TV show Jeopardy! against human champions.

Chatbot. Chatbots are conversational automatic programs. They appear in many applications, ranging from simple question answering to restaurant recommendations. For example, Facebook created an API to design chatbots

(https://developers.facebook.com/docs/messenger-platform/), which help a company to answer customers questions. SNCF's Ouibot (https://www.oui.sncf/bot) allows travellers to find a train in France. More advanced dialogue systems exist, including Apple's Siri, Amazon's Alexa, Google Assistant, or Microsoft's Cortana. All of these systems rely on some knowledge base, used to represent products, train schedules, restaurants, etc.

Language Comprehension. When humans talk, they make many assumptions about what the receiver knows. A computer has to use a knowledge base of some sort to understand each statement. For example, the sentence "The president of France is currently buying masks" makes total sense if we know that: The president of France is Emmanuel Macron, masks prevent a virus from spreading, people use masks, there is currently a pandemic, etc. Thus, knowledge bases can be used to understand daily news, social media and scholarly publications. The software AmbiverseNLU (https://github.com/ambiverse-nlu/ambiverse-nlu) implements a tool for natural language understanding. In particular, it can link groups of words in a sentence to entities in a knowledge base.

Fact Checking. This category is a particular kind of language comprehension tasks that involves reasoning to verify whether a fact is true or not. It is becoming more and more critical due to the multiplication of fake news on the Web. This task relies on high-quality knowledge about the considered domain [34] from which one can deduce whether a fact is true or not.

Argumentation Mining. Argumentation mining is a task that involves the automatic detection of arguments in texts, the decomposition of argument components, and the finding of relations between arguments and their components. Argumentation mining is a complex task, which requires a deep understanding of a subject, and knowledge bases help to represent and reason about an argument. Recently, the IBM Debater Project (https://www.research.ibm.com/artificial-intelligence/project-debater/) proved that it can have a real debate with another human being. One of its key components is its knowledge base, which allows it to understand arguments and humans dilemmas.

Visual Comprehension. Knowledge bases can also be used for tasks that do not involve text. In the case of visual comprehension, having a priori information about the subjects in the pictures can help better guessing what the items are and what they are doing. For example, in an image containing both a giraffe and a mountain, which have the same pixel size, knowing that giraffes are smaller than mountains help us understand that the mountain is far away.

Explainable IA. The emergence of Deep Learning and complex black-box systems during the past few years raised the question of explaining the decisions of an algorithm. In 2016, the General Data Protection Regulation (GDPR) gave a legal dimension to the quest of relevant explanations. In [5], the authors stress the fact that knowledge is a crucial component. Indeed, some systems [60] leverage knowledge bases to explain the choice of a neural network system.

1.1.5 Accessing Knowledge Bases Through Web Services

As we saw, knowledge bases help many applications thanks to high-quality data. To provide such knowledge without giving away entirely, some data providers offer Web services. A web service is a service on the web that can be automatically queried by a computer. In 2020, the website programmableweb indexes more than 23.000 such Web services. The provider's goal is to get as much profit as possible out of their resources. So, the access is often limited, and intensive users have to pay a subscription fee.

Web service users, on the other hand, have two goals: get an exact and complete answer to a question (also called a query) and pay a minimal cost. To do so, they might have to combine access methods from one or several web services. For example, combining a food web service with a medical one might help to find healthy diets.

Trying to answer to a query thanks to partial access functions (also called views) is a well-known problem called *query rewriting over views*. When the rewriting gives guarantees that it yields exactly the results of the initial query, we talk about an *equivalent rewriting*.

1.2 Contributions

In this thesis, we will tackle two main challenges in the knowledge base community. The first one is about the automatic extraction of commonsense knowledge, and the second is about the querying of knowledge bases. These two subjects relate to the two components of a knowledge base, which are the ABox and the TBox. The ABox is the assertion component and contains statements about the world. The commonsense knowledge extraction creates facts which are stored in the ABox. The TBox is the terminology component and allows representing constraints on our knowledge base. When data providers give limited access to their knowledge base, we can try to exploit TBox constraints to reason and get access to additional information.

1.2.1 Harvesting Commonsense Knowledge Automatically

As we saw in Section 1.1.3, knowledge bases are either handcrafted (as in WordNet) or automatically extracted (as in Yago). In the second case, the algorithms generally rely on structured texts such as articles on Wikipedia. As a consequence, these knowledge bases store mainly encyclopedic knowledge.

In comparison, *commonsense knowledge* is complicated to harvest automatically for several reasons. The main one is that people rarely write it down because they suppose the readers already know it. Thus, information extraction algorithms struggle to find commonsense knowledge and to differentiate it from contextual knowledge, i.e. knowledge true in the context of a document. However, commonsense knowledge is crucial to understand humans and their interactions. Besides, it also deals with relevant negative statements, which are missing in most current systems.

Our research question is the following: How can we build a commonsense knowledge base automatically by querying online resources?

In Chapter 3, we introduce a general method to extract commonsense knowledge from unusual sources. We analyse the questions asked in query logs and question answering forums and show that they convey salient information about entities. From this observation, we gather many questions which we plug into a pipeline and turn them into statements. The resulting knowledge base is open-source, contains more than two million facts and is called *Quasimodo*. Our results were published at CIKM 2019 [111]:

> Romero, J., Razniewski, S., Pal, K., Z. Pan, J., Sakhadeo, A., & Weikum, G. (2019, November).
> Commonsense properties from query logs and question answering forums.
> Full Paper at the International Conference on Information and Knowledge Management (CIKM).

It was followed by a demo paper published at CIKM 2020 [110]:

> Romero, J. & Razniewski, S.
> Inside Quasimodo: Exploring the Construction and Usage of Commonsense Knowledge.
> Demo Paper at the Conference on Information and Knowledge Management (CIKM)

1.2.2 Decidable, Polynomial and Equivalent Query Rewriting

As we saw in Section 1.1.5, many knowledge bases can be accessed only through Web services. In some cases, we need to combine access methods from one or several Web services to get the answer to a query, a process called query rewriting. In this way, we access what we call *hidden* knowledge.

Query rewriting is a challenging problem for many reasons. First, finding if there exists a query rewriting is not *decidable* for some categories of query rewriting. This means that the algorithms can run forever without knowing when to stop looking for a query rewriting. Second, the algorithms to solve the problem are in general *exponential*. So, even in the decidable cases, the computation time can be very long. Finally, when we look for *equivalent* rewritings, we need to reason about integrity constraints. This task comes with challenges of its own, also related to decidability and computation time.

Querying knowledge bases hidden behind Web Services is an old topic. The first purpose of requesters is to get as much value as possible from their requests. So, they want to have guarantees that the returned results answer their queries. However, this often implies reasoning over access methods and the structure of the knowledge base. This reasoning is usually very complicated. In general, when we have constraints, it relies on the *chase* algorithm. This algorithm often takes very long to execute and is not guaranteed to terminate in all cases. So, other methods emerged, but give partial results or restrict themselves to particular scenarios.

Our research questions are the following: When we have access to integrity constraints, can we find a practical scenario where equivalent query rewriting is decidable and polynomial? If we do not have integrity constraints, can we create a practical class of query rewriting that is decidable?

We tackle these questions in several chapters.

In Chapter 5, we study the problem of finding equivalent query rewritings on views with binding patterns. We isolate a particular setting in which it is tractable to find the existence of such a rewriting. In this setting, we use what we call *path views* and unary inclusion dependencies. Then, we provide an algorithm that can find out in polynomial time if there exists an equivalent query rewriting and, if it is the case, enumerate all of them. The algorithm uses notions of formal language theory: We express the resulting plans as the intersection between a context-free grammar and a regular expression. Our work was published at ESWC 2020 [108]:

> Romero, J., Preda, N., Amarilli, A., & Suchanek, F. (2020, May).
> Equivalent Rewritings on Path Views with Binding Patterns.
> Full Paper at the European Semantic Web Conference (ESWC)

It was followed by a demo paper at CIKM 2020 [107]:

> Romero, J., Preda, N., Amarilli, A., & Suchanek, F.
> Computing and Illustrating Query Rewritings on Path Views with Binding Patterns.
> Demo Paper at the Conference on Information and Knowledge Management (CIKM)

In Chapter 6, we continue in the same direction as Chapter 5 and analyse what happens when we drop the integrity constraints. In this situation, equivalent rewritings are of little interest, and we define a new class of plan called the *smart plans*. We describe what their properties are and how we can generate them in finite time.

> Romero, J., Preda, N., & Suchanek, F.
> Query Rewriting On Path Views Without Integrity Constraints
> Workshop paper at Datamod 2020

In Chapter 7, we introduce *Pyformlang*, an open-source python library for formal language manipulation. We originally designed it to support the algorithm introduced in Chapter 5. Indeed, existing libraries did not support the operations we required. Now, *Pyformlang* contains most of the textbook operations on regular expressions, finite state machines, context-free grammar and push-down automata. Besides, it implements some functionalities of finite transducer and operations on indexed grammars which were never implemented in a public library.

> Romero, J.
> Pyformlang: An Educational Library for Formal Language Manipulation
> Full paper at SIGCSE 2021

Chapter 2

Preliminaries

> The only true wisdom is in knowing you know nothing.
>
> *Socrates*

Before we enter into the main contributions of this thesis, we first introduce the fundamentals and recurring concepts that will later appear. The first one is the concept of a "knowledge base", which we describe in Section 2.1. The second one is the concept of "Web services", which we introduce in Section 2.2.

2.1 Knowledge Bases

Knowledge bases are composed of two main components. The first one is called the ABox and contains assertions about the world. The way these assertions are stored depends on the knowledge model. In Section 2.1.1, we introduce the knowledge representation used by most existing systems. The second component of a knowledge base is the TBox and contains conceptual information we can use to reason on the ABox. In Section 2.1.2, we study one possible way to represent the TBox using schemas.

2.1.1 Knowledge Representation

Knowledge bases store various kinds of data that represent knowledge about a large variety of topics. We will now see how to represent them. The problem of knowledge representation is quite complicated, and, in practice, we can only represent simple information. In what follows, we are going to talk about entity-centric knowledge bases [124]. They represent:

- Entities, which are elements of the world (like *Georges Brassens*, *The Eiffel Tower* or *Inkscape* [129])
- Semantic classes and class membership of entities (*GeorgesBrassens is a singer*, *Inkscape is a software*)

- Relationships between entities (*GeorgesBrassens won the award GrandPrixDePoésie*)
- Time indications about relationship (*Inkscape is a member of OpenInventionNetwork Since March 2013*).

Some knowledge bases have additional information such as a modality or a polarity for a relationship, but it is uncommon. Let us now dive into the components of knowledge bases.

Entities

The cornerstone of knowledge bases is the notion of an entity. Its definition is very general and we can apply it to many things, but some systems restrict it to fit their needs. We here give the definition from the Cambridge Dictionary.

Definition 2.1.1 (Entity)**.** *An entity is something that exists apart from other things, having its own independent existence.*

All of the following are entities:

- People, like *Georges Brassens*
- Animals, like *elephant*
- Events, like *The Paris Commune*
- Words, like *take*
- Concepts, like *Freedom*
- Fictional Characters, like *Madame Bovary*
- Feelings, like *fear*
- Situations, like *Being a doctor*
- Numbers, like *5,271,009*

When developers design a knowledge base, they generally choose to restrict themselves to a subset of the infinitely many possible entities. WordNet focuses on words, whereas Wikidata focuses on encyclopedic entities.

An entity can have different names. For example, we could call *Georges Brassens* simply *Georges*, or *The singer of Brave Margot*, or even *The guy with a moustache*. We call these denominations *labels*. The notion of label in knowledge bases differs slightly from the common usage and we give here the definition from the RDF specification [135].

Definition 2.1.2 (Label)**.** *A label is a human-readable version of an entity's name.*

Note that an entity can have more than one label. For example, Alain Damasio is the pen name of the French writer Alain Raymond.

An entity that has a label is called a *named entity*. This category of entities is the core of many knowledge bases such as Wikidata or Yago. *Georges Brassens* is a named entity, but also *Harry Potter*, *The Tower Bridge*, and *Nintendo Switch*. We notice that there exist entities without a name. For example, the computer or the paper on which you read this thesis is very unlikely to have one.

From our previous example, we see that *Georges Brassens* has more than one label. When we can refer to an entity with several labels, we call them *synonymous*. When a label refers to several entities, we call it *polysemous*. However, knowledge bases have to represent unique entities. Therefore, they pick a discriminating name for each entity. We call it an *identifier*. We adapt here the definition from the Cambridge dictionary.

Definition 2.1.3 (Identifier)**.** *An identifier is a set of numbers, letters, or symbols that is used to represent a unique entity in a computer program.*

In general, the identifier is a number or a string of characters, but other kinds of objects are sometimes used such as dates. An identifier is unique as it should refer to a single entity. The format can vary a lot: It can be cryptic as in Wikidata (Q41777585, Q91970384) or quite explicit like in Yago (yago:Georges_Brassens).

Another notion of knowledge bases is the notion of literals. In a computer program, a literal is associated to a type and we give here the definition of [121].

Definition 2.1.4 (Typed Literal)**.** *Typed literals are of the form "v"^^u, where v is a Unicode string, called the lexical form of the typed literal, and u is a URI reference of a data type.*

A URI (Uniform Resource Identifier) is a string that identifies a unique resource. We generally use typed literals to represent a string of characters or numbers. For example, the full name of *Georges Brassens* is the literal "Georges Charles Brassens"^^xs:string, where xs:string is the XML string data type. His year of birth is the literal "1921"^^xs:gYear, where xs:gYear is the XML date data type. Note that for this example, we could have modelled the year as an entity (in particular for recent history).

Relations

Until this point, we have focused on creating the base elements of a knowledge base: Entities. However, the entities are only the actors, we still need to represent the actions, i.e. the links between them. To model the interactions, we introduce the concept of a *relation*, similar to the one used in mathematics and defined in [1].

Definition 2.1.5 (Relation)**.** *A n-ary relation (or relationship) over a set of entities $\mathcal{E}$ is a subset of the Cartesian product $\mathcal{E}^n$.*

When possible, relations are given a name that makes sense in the real-world such as *parent*. Although relations can be anything, knowledge bases model real-world relationships. So, they mainly contain relations whose elements are considered true

in the real-world or in one of its representations (like in a novel). Besides, we extend the notions of labels and identifiers to relations. These two can be different. For example, in Wikidata, the relation to link a capital to its country is labelled *capital of* and has the identifier *P1376*.

As an example of a relation, let us consider the 3-ary relation *parents*. In the real world, it represents the link between a child and her parents. Thus, it contains triples like ⟨ *Georges Brassens*, *Elvira Dagrosa*, *Jean-Louis Brassens* ⟩, ⟨ *Jacques Brel*, *Élisabeth "Lisette" Lambertine*, *Romain Brel* ⟩ and ⟨ *Léo Ferré*, *Marie Scotto*, *Joseph Ferré* ⟩.

Given an n-ary relation R and one of its elements $\langle e_1, \ldots, e_n \rangle$, it is common to write $R(e_1, \ldots, e_n)$ instead of $\langle e_1, \ldots, e_n \rangle \in R$. We will adopt this notation in what follows. There are other standard notions used talk about relations:

- The number n is the *arity* of the relation R
- $R(e_1, \ldots, e_n)$ is a *statement*, *fact* or *record*
- $\langle e_1, \ldots, e_n \rangle$ is a tuple of R
- $e_1, \ldots, e_n$ are the arguments of the fact

In what follows, we will consider sub-classes of relations obtained by fixing the arity. In particular, we are interested in 1-ary and 2-ary relations.

Classes

The most degenerate type of relation is a class. It represents something about an entity without referring to anything else. In [8], a concept (or class) "is interpreted as a set of individuals", which can also be described as:

Definition 2.1.6 (Class)**.** *A class (or concept, type) is a 1-ary relation. An element of a class is an instance.*

Knowledge bases generally use classes to represent groups of entities that share similar properties. Thus, we could have the class of *singers*, *poets*, *fictional characters*, *cities* or *historical events*. *Georges Brassens* would be an instance of the class *singers* and of the class *poets*, *Harry Potter* of the *fictional characters*, *Paris* of *cities* and *Paris Commune* of *historical events*.

The choice of the classes depends on the designers of a knowledge bases. Even if they represent most of the time real groups of similar entities, the question of what is a relevant class or not is quite open. For example, it is quite natural to have a class for singers. However, the existence of a class of singers with a moustache (which contains *Georges Brassens*) is more questionable in a knowledge base (though it exists in the real-world). It depends on what is the purpose of the knowledge base. For example, WordNet has very precise classes, close to a scientific description of the world.

The distinction between what should be considered a class and what should be an instance is also a crucial point in the design of a knowledge base. For example, should we consider that *iPhone* is an instance of the class *smartphone brands* or

that it is a class of objects containing the *iPhone 1* (which could also be a class)? We could also have both situations in the same knowledge base.

In order to avoid choosing between being a class or being an entity, most knowledge bases choose to consider a class (and more generally any relation) an entity. Wikidata made this choice: everything is an entity. In that case, we often refer to entities that are neither classes nor literals as *common entities*.

Binary Relations

Knowledge bases rarely use the general concept of n-ary relations. Instead, they restrict themselves to what we call binary relations. In [8], they call them *roles*, that "are interpreted as sets of pairs of individuals".

Definition 2.1.7 (Binary Relation)**.** *A binary relation (or role) is a 2-ary relation.*

These simplified relationships have many practical advantages.

- Many real-world relations are binary relations: *mother*, *father*, *capital of*, *located in*, *wrote*, *composed*, *died in*, etc.
- We can turn n-ary relations into binary ones (see below)
- We can have partial information about an event or a situation. For example, if we had the relation *parents* (in its 3-ary version presented before), it would not be possible to represent the fact that we know the mother of a person but not their father. So, it might be a better idea to split *parents* into two relations: *mother* and *father*.
- We can add more easily facts about an event. For example, if we decide to add the birth date to the relation *parents*, the modification is quite heavy as it requires to modify all parents relations. It is easier to create a new binary relation *birthdate*.
- It makes knowledge bases easier to manipulate by many applications: Many inference systems and rule mining systems use binary relations, we can represent a knowledge base by an heterogeneous graph, the query systems are easier to write, we can inverse relations, etc.

Binary relations also come with standard terminology. For a statement $R(x, y)$:

- x is the *subject*
- R is the *predicate* or *property*
- y is the *object*
- if R has only literals as objects, and maps a subject to at most one object, it is an *attribute* (e.g. the relations *birthyear* and *title* are attributes)
- R^- is the *inverse relation* of R and contains all tuples $\langle y, x\rangle$ such that $\langle x, y\rangle \in R$ (e.g. the relation *sang* has *issungby* as inverse)

- if R is a relation over two classes $A \times B$, we call A the *domain* and B the *range*

There exists a trick to turn any n-ary relation ($n > 2$) into several binary relations. Let us consider an n-ary relation R and a statement $R(x_1, \ldots, x_n)$. For example, we could have the relation *parents* and the fact *parents*(*Georges Brassens*, *Elvira Dagrosa*, *Jean-Louis Brassens*). We suppose that one of the argument is a key. The key is a position i independent of a statement such that the i^{th} argument of R is always associated to a single element of R. For the relation *parents*, the key is the first argument: a child has only two parents. Let us assume that the key is the first argument. Then, we can decompose R into $n-1$ relations: R_2, ..., R_n, which are $R_2(x_1, x_2)$, ..., $R_n(x_1, x_n)$. In our example, we would create the relations $parent_1$(*Georges Brassens*, *Elvira Dagrosa*) and $parent_2$(*Georges Brassens*, *Jean-Louis Brassens*). We notice that $parent_1$ is in fact the relation *mother* and $parent_2$ the relation $father$.

In case there is no key, it is still possible to apply the trick. Let us consider the 3-ary relation *sang_in* which links a singer and a song to a place where the singer performed the song. For example, *sang_in*(*Georges Brassens*, *Les Copains d'abord*, *Bobino*). Obviously, this relation does not have a key. Then, we create an artificial entity e per statement that we call the *event entity* and that we connect to n relations R_1, ..., R_n. We then add to our set of facts the new facts $R_1(e, x_1)$, ..., $R_n(e, x_n)$. For our example, we can create the event entity *BrassensCopainsBobino* that represents this performance, and the facts $sang\_in_1$(*BrassensCopainsBobino*, *Georges Brassens*), $sang\_in_2$(*BrassensCopainsBobino*, *Les Copains d'abord*) and $sang\_in_3$(*BrassensCopainsBobino*, *Bobino*). In this case, we see that it is complicated to interpret the newly created entity and relations. The entity could have the label "the fact that Georges Brassens sang Les Copains d'abord at Bobino", but this looks very artificial.

Class membership can also be represented with binary relations. Knowledge bases generally introduce a new relation such as *hasProperty*, *hasClass* or *type* or reuse the RDF *rdf:type* relation that links an entity to one of its classes. For example, if we have that *Georges Brassens* is in the class of singers, then we could create the binary fact *type(Georges Brassens, singer)*.

A Standard For Representing Knowledge: RDF

As we mentioned above, most knowledge bases use only binary relations to encode their content. We explained that we could transform affiliation to a class and n-ary relations into binary relations. When we are in this case, we generally call the statements *triples*. Indeed, we often write statements as *(subject, predicate, object)*, where *subject* is the subject of the relation, *predicate* is the relation and *object* the object of the relation.

In this situation, knowledge bases generally use the RDF (Resource Description Framework) standard to store their data. The W3C (World Wide Web Consortium) first issued this standard in 1996 to represent and describe web resources. However, it quickly found a central place in the knowledge representation community.

RDF is an abstract model centred around the triple representation of data as we described above. There is no specified format to save information. For example, we could use XML (Extensible Markup Language), N-triples (a white space separates the components of a fact), Turtle (a more compact version of N-triples) or JSON (JavaScript Object Notation).

RDF comes with a set of predefined relations and classes such as *rdf:Property* (the class of all properties) or *rdf:type* (the instance-of relation). Unique URIs are used to represent resources (subjects, predicates and objects). These URIs often link to online resources. In Wikidata, the URI of an entity is its URL on the website, like https://www.wikidata.org/wiki/Q41777585. Schema.org provides a global vocabulary for online resources. For example, the URL https://schema.org/name represent the *name* relationship.

Several standards extend RDF to provide additional functionality. For example, we have RDFs (RDF Schema) and OWL (Web Ontology Language). They generally come with new resource descriptions such as *rdfs:subClassOf* (which represents the sub-class relationship) or *owl:thing* (the class of all OWL individuals).

To query data in RDF format, we generally use SPARQL (SPARQL Protocol and RDF Query Language). It is very similar to SQL (the language to query databases) except that it is designed to work with triple stores with entities and classes. Most notorious knowledge bases generally come with a SPARQL endpoint. For example, for Wikidata, it is https://query.wikidata.org/ and for YAGO4, it is https://yago-knowledge.org/sparql.

2.1.2 Reasoning Over Knowledge Bases

We are now going to focus on the second kind of information a knowledge base contains. This will allow us to reason about the statements by adding constraints respected in the knowledge base. The central notion here is the *schema* of a knowledge base.

The definition can vary depending on the authors. In [1], they define a *database schema* as the set of all relations. Here, we prefer the definition of [15], which provides additional integrity constraints we will later use.

Definition 2.1.8 (Schema)**.** *A schema is a triple* $(\mathcal{R}, \mathcal{E}, \mathcal{IC})$ *where:*

- $\mathcal{R}$ *is a set of n-ary relations*
- $\mathcal{E}$ *is a set of entities (or constants)*
- $\mathcal{IC}$ *is a set of integrity constraints, generally described in first-order logic on* $\mathcal{R}$ *and* $\mathcal{E}$*.*

Example 2.1.9. *Let us consider a schema for a knowledge base about artists. We could define:*

- *the set of relations as* $\mathcal{R} = \{$*sang, painted, painter, singer, playInstrument, guitarist, musician, artist* $\}$
- *the set of constants could be* $\mathcal{E} = \{$ *Georges Brassens, Jacques Brel, Edgar Degas, Andrés Segovia, Guitar* $\}$

- *the set of integrity of constraints, which contains:* $\forall x, singer(x) \Rightarrow \exists y, sang(x, y)$ *and* $\forall x, guitarist(x) \Rightarrow playInstrument(x, Guitar)$.

2.1.3 Defining Knowledge Bases

We now have all the components to define what is a knowledge base. As we discussed at the beginning of this section, a knowledge base has two parts: the part that represents knowledge is called the *ABox*, and the logical part is called the *TBox*. We follow the definitions of [121].

Definition 2.1.10 (Abox)**.** *The ABox (A for assertion) is a set of statements that represent knowledge.*

Definition 2.1.11 (TBox)**.** *The TBox (T for terminology) is a schema used for reasoning about knowledge.*

The database community uses the word *schema*, whereas the word *TBox* is from the knowledge base community.

Definition 2.1.12 (Knowledge Base)**.** *A knowledge base is a couple composed of an ABox and a TBox.*

2.2 Web Services

2.2.1 Motivation and Definition

Web services give access to specialised knowledge on the Web for other machines. For example, in the music world, MusicBrainz (musicbrainz.org/) gives access to information about music through an online endpoint. This *Web service* follows the *REST* architecture (see Section 2.2.2 for more details about this architecture) and exposes parametric URLs to access information. For example, to get the list of all the artists called Elvis is the US, one has to query the URL musicbrainz.org/ws/2/artist/?query=artist:elvis%20AND%20type:person%20AND%20country:US. We give in Figure 2.1 the beginning of the XML returned by MusicBrainz.

The definition of a Web service is very general:

Definition 2.2.1 (Web Service)**.** *We call a Web service any service on the World Wide Web that can be automatically queried and that provides machine-readable outputs.*

The main point of this definition is that a Web service needs to be usable by other machines. In some case, it might be impossible to access it easily for humans. Besides, the returned format is first aimed for automated processing. So, a human might have difficulties reading it (compared to plain text) but a machine can parse it and turn it into an internal representation. Typically, Web services use formats like XML or JSON, but also, sometimes, directly compressed data streams.

```
▼<metadata xmlns="http://musicbrainz.org/ns/mmd-2.0#" xmlns:ns2="http://musicbrainz.org/ns/ext#-2.0" created="2020-06-03T15:40:15.001Z">
  ▼<artist-list count="12" offset="0">
    ▼<artist id="01809552-4f87-45b0-afff-2c6f0730a3be" type="Person" type-id="b6e035f4-3ce9-331c-97df-83397230b0df" ns2:score="100">
       <name>Elvis Presley</name>
       <sort-name>Presley, Elvis</sort-name>
       <gender>male</gender>
       <country>US</country>
      ▼<area id="489ce91b-6658-3307-9877-795b68554c98" type="Country" type-id="06dd0ae4-8c74-30bb-b43d-95dcedf961de">
         <name>United States</name>
         <sort-name>United States</sort-name>
        ▼<life-span>
           <ended>false</ended>
         </life-span>
       </area>
      ▼<begin-area id="157e3c0e-1b68-403d-a515-6e37cb67947b" type="City" type-id="6fd8f29a-3d0a-32fc-980d-ea697b69da78">
         <name>Tupelo</name>
         <sort-name>Tupelo</sort-name>
        ▼<life-span>
           <ended>false</ended>
         </life-span>
       </begin-area>
      ▼<end-area id="c2d96f61-75a4-4375-aed5-6aacb0b6326a" type="City" type-id="6fd8f29a-3d0a-32fc-980d-ea697b69da78">
         <name>Memphis</name>
         <sort-name>Memphis</sort-name>
        ▼<life-span>
           <ended>false</ended>
         </life-span>
       </end-area>
       <disambiguation>King of Rock and Roll</disambiguation>
      ▼<ipi-list>
         <ipi>00056021705</ipi>
         <ipi>00056021803</ipi>
         <ipi>00058226279</ipi>
       </ipi-list>
      ▼<isni-list>
         <isni>0000000121241960</isni>
```


Figure 2.1: Example of XML returned by a Web service

2.2.2 Web Service Architectures

We review here the main architectures that are used to design a Web service.

REST Architecture

The REST (Representational state transfer) architecture is a client-server architecture created in 2000 by Roy Fielding [63]. Its main goal is to permit access and manipulation of textual Web resources. Machines access these resources by a query to a specific URI (Uniform Resource Identifier). The most common protocol used is HTTP in which URIs are URLs. In particular, RESTful Web services use HTTP. Thus, we use HTTP methods such as GET, POST, PUT, DELETE to access, modify, create and delete resources.

The main property of the REST architecture is that it is *stateless*. This means that there are no sessions for the users and, therefore, the requesters must send all relevant information with each query. This property has many advantages. It makes RESTful systems:

- Fast, as there is no global state to manage. Speed is a key factor for many applications.
- Reliable, as the components of the systems can be separated more easily, thus preventing the propagation of failures.
- Flexible and scalable, as the system and data can be modified without impacting the entire system.

However, REST also comes with drawbacks:

- Being stateless may be a disadvantage for some applications.

- REST uses HTTP, so a server cannot send a notification to a client, which is often required.
- REST is limited to HTTP request methods such as GET, POST or PUT.
- From a security point of view, REST only uses SSL/TLS, which is not flexible in the way we can encrypt the information. As a comparison, *WS-Sec** allows partially encrypted data, which makes content-based routing easier.

In addition to the stateless property, RESTful APIs must respect other design rules: the client and the server must be separated, caching information should be provided (to know whether or not a resource is prone to change), the systems must use abstraction layers (the access to data should be a black box), and the interface should be uniform.

The formats used in standard RESTful API are JSON, XML, HTML or more specific formats, depending on the application.

SOAP-based Architecture

SOAP (Simple Object Access Protocol) [21] was introduced in 1998 and is a high-level messaging protocol for Web services. This protocol extends XML to encode the interactions between the service provider and the user. SOAP is built on three principles:

- extensibility: components can be added to the main implementation.
- neutrality: the protocol is high level and is independent of the underlying layers, which can use protocols such as HTTP, TCP or UDP.
- independence: it works the same way on all machines, all programming languages or all operating systems.

However, SOAP is relatively slow compared to lower-level systems, mainly due to the additional workload created by the XML protocol. So, nowadays, most Web services have a REST interface, since it is faster and easier to use.

In this thesis, we suppose we have Web services implemented with the REST architecture.

Chapter 3

Quasimodo: A Commonsense Knowledge Base

> Common sense is not so common.
>
> *Voltaire, A Pocket Philosophical Dictionary*

In the first part of this thesis, we focus on the extraction of commonsense knowledge from online resources. We build a knowledge base we call Quasimodo. We mainly focus on the ABox.

This work was published in CIKM 2019 [111]:

> Romero, J., Razniewski, S., Pal, K., Z. Pan, J., Sakhadeo, A., & Weikum, G. (2019, November). Commonsense properties from query logs and question answering forums. In Proceedings of the 28th ACM International Conference on Information and Knowledge Management (pp. 1411-1420).

****

3.1 Introduction

3.1.1 Motivation and Goal

Commonsense knowledge (CSK for short) is an old theme in AI, already envisioned by McCarthy in the 1960s [85] and later pursued by AI pioneers like Feigenbaum [61] and Lenat [79]. The goal is to equip machines with knowledge of properties of everyday objects (e.g., bananas are yellow, edible and sweet), typical human behaviour and emotions (e.g., children like bananas, children learn at school, death causes sadness) and general plausibility invariants (e.g., a classroom of children should also have a teacher).

In recent years, research on the automatic acquisition of such knowledge has been revived, driven by the pressing need for human-like AI systems with robust and explainable behaviour. Essential use cases of CSK include the interpretation of user intents in search-engine queries, question answering, versatile chatbots, language comprehension, visual content understanding, and more.

Examples: A keyword query such as "Jordan weather forecast" is ambiguous, but CSK should tell the search engine that this refers to the country and not to a basketball player or machine learning professor. A chatbot should know that racist jokes are considered tasteless and would offend its users; so CSK could have avoided the 2016 PR disaster of the Tay chatbot (www.cnbc.com/2018/03/17/facebook-and-youtube-should-learn-from-microsoft-tay-racist-chatbot.html). In an image of a meeting at an IT company where one person wears a suit, and another person is in jeans and t-shirt, the former is likely a manager and the latter an engineer. Last but not least, a "deep fake" video where Donald Trump rides on the back of a tiger could be easily uncovered by knowing that tigers are wild and dangerous and, if at all, only circus artists would do this.

The goal of this chapter is to advance the automatic acquisition of salient commonsense properties from online content of the Internet. For knowledge representation, we focus on simple assertions in the form of subject-predicate-object (SPO) triples such as `children like bananas` or `classroom include teacher`. Complex assertions, such as Datalog clauses, and logical reasoning over these are outside our scope.

A significant difficulty that prior work has struggled with is the sparseness and bias of possible input sources. Commonsense properties are so mundane that they are rarely expressed in explicit terms (e.g., countries or regions have weather; people do not). Therefore, conventional sources for information extraction like Wikipedia are relatively useless for CSK. Moreover, online contents, like social media (Twitter, Reddit, Quora etc.), fan communities (Wikia etc.) and books or movies, are often heavily biased and do not reflect typical real-life situations. For example, existing CSK repositories contain odd triples such as `banana located_in monkey's_hand`, `engineer has_property conservative`, `child make choice`.

3.1.2 State of the Art and Limitations

Popular knowledge bases like DBpedia, Wikidata or Yago have a strong focus on encyclopedic knowledge about individual entities like (prominent) people, places etc., and do not cover commonsense properties of general concepts. The notable exception is the inclusion of SPO triples for the `(sub-)type` (aka. `isa`) predicate, for example, `banana type fruit`. Such triples are ample especially in Yago (derived from Wikipedia categories and imported from WordNet). Our focus is on additional properties beyond `type`, which are absent in all of the above knowledge bases.

The most notable projects on constructing commonsense knowledge bases are Cyc [79], ConceptNet [119], WebChild [127] and Mosaic TupleKB [35]. Each of these has specific strengths and limitations. The seminal Cyc project solely relied on human experts for codifying logical assertions, with inherent limitations in scope and scale. Currently, only a small sample of Cyc is accessible: OpenCyc.

ConceptNet used crowdsourcing (mainly originating from the Open Mind Common Sense project) for scalability and better coverage, but is limited to only a few different predicates like `has_property`, `located_in`, `used_for`, `capable_of`, `has_part` and `type`. Moreover, the crowdsourced inputs often take noisy, verbose or uninformative forms (e.g., `banana type bunch`, `banana type herb`, `banana has_property good_to_eat`). Besides, ConceptNet includes more traditional sources, such as a subset of DBpedia and OpenCyc or synonyms and antonyms from Wikitionary.

WebChild tapped into book n-grams and image tags to overcome the bias in many Web sources. It has a wider variety of 20 predicates and is much larger, but contains a massive tail of noisy and dubious triples – due to its focus on possible properties rather than typical ones (e.g., engineers are conservative, cool, qualified, hard, vital etc.). Besides, WebChild is only able to generate facts which can be expressed in a window of five words, which is very limiting. The idea of Webchild is to use a label propagation algorithm seeded by WordNet over a graph constructed from the co-occurrence of words in n-grams. Finally, Webchild tries to perform entity disambiguation by linking its subjects to WordNet synsets. However, it creates too much noise to be used in practice.

TupleKB is built by carefully generating search-engine queries on specific domains and performing various stages of information extraction and cleaning on the query results. Despite its clustering-based cleaning steps, it contains substantial noise and is limited in scope by the way the queries are formulated.

The work in this chapter aims to overcome the bottlenecks of these prior projects while preserving their positive characteristics. In particular, we aim to achieve high coverage, like WebChild, with high precision (i.e., a fraction of valid triples), like ConceptNet. Besides, we strive to acquire properties for a wide range of predicates – more diverse and refined than ConceptNet and WebChild, but without the noise that TupleKB has acquired.

3.1.3 Approach and Challenges

This chapter puts forward *Quasimodo*, a framework and tool for scalable automatic acquisition of commonsense properties. The name stands for "Query Logs and QA Forums for Salient Commonsense Definitions". Quasimodo is the main character in Victor Hugo's novel "The Hunchback of Notre Dame" who epitomises human preconception and also exhibits unexpected traits. Quasimodo is designed to tap into non-standard sources where questions rather than statements provide cues about commonsense properties. This leads to noisy candidates for populating a commonsense knowledge base (CSKB). To eliminate false positives, we have devised a subsequent cleaning stage, where corroboration signals are obtained from a variety of sources and combined by learning a regression model. This way, Quasimodo reconciles extensive coverage with high precision. In doing this, it focuses on salient properties which typically occur for familiar concepts, while eliminating possible but atypical and uninformative output. This counters the reporting bias - frequent mentioning of sensational but unusual and unrealistic properties (e.g., pink elephants in Walt Disney's Dumbo).

The new sources that we tap into for gathering candidate assertions are search-

engine query logs and question answering forums like Reddit, Quora etc. Query logs are unavailable outside industrial labs but can be sampled by creatively using search-engine interfaces. To this end, Quasimodo generates queries judiciously and collects auto-completion suggestions.

The subsequent corroboration stage harnesses statistics from search-engine answer snippets, Wikipedia editions, Google Books and image tags employing a learned regression model. This step is geared to eliminate noisy, atypical, and uninformative properties.

A subsequent ranking step further enhances the knowledge quality in terms of typicality and saliency. Finally, to counter noisy language diversity, reduce semantic redundancy, and canonicalise the resulting commonsense triples to a large extent, Quasimodo includes a novel way of clustering the triples that result from the fusion step. This is based on a tri-factorisation model for matrix decomposition.

Our approach faces two major challenges:

1. coping with the heavy reporting bias in cues from query logs, potentially leading to atypical and odd properties,

2. coping with the noise, language diversity, and semantic redundancy in the output of information extraction methods.

The work shows how these challenges can be (mostly) overcome. Experiments demonstrate the practical viability of Quasimodo and its improvements over prior works.

3.1.4 Contributions

This work makes the following original contributions:

1. a complete methodology and tool for multi-source acquisition of typical and salient commonsense properties with principled methods for corroboration, ranking and refined grouping,

2. novel ways of tapping into non-standard input sources like query logs and QA forums,

3. a high-quality knowledge base of ca. 4.4 million salient properties for ca. 103 thousand concepts, publicly available as a research resource (https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/commonsense/quasimodo/),

4. an experimental evaluation and comparison to ConceptNet, WebChild, and TupleKB which shows major gains in coverage and quality, and

5. experiments on extrinsic tasks like language games (Taboo word guessing) and question answering.

Our code is available on Github (https://github.com/Aunsiels/CSK).

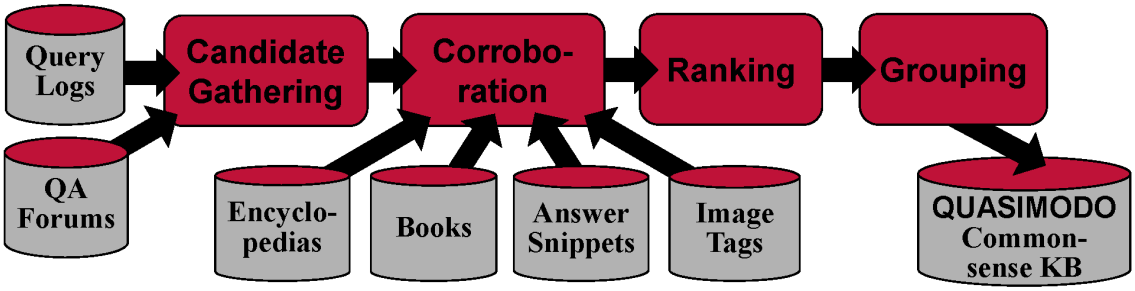

Figure 3.1: Quasimodo system overview

3.2 Related Work

3.2.1 Commonsense Knowledge Bases (CSKB's)

The most notable projects on building sizeable commonsense knowledge bases are the following.

Cyc. The Cyc project, started in 1984 by Douglas Lenat, was the first significant effort towards collecting and formalizing general world knowledge [79]. Knowledge engineers manually compiled knowledge in the form of grounded assertions and logical rules. Parts of Cyc were released to the public as OpenCyc in 2002, but these parts mostly focus on concept taxonomies, that is, the `(sub-)type` predicate.

ConceptNet. Crowdsourcing has been used to construct ConceptNet, a triple-based semantic network of commonsense assertions about general objects [119, 120]. ConceptNet contains ca. 1.3 million assertions for ca. 850,000 subjects (counting only English assertions and semantic relations, i.e., discounting relations like *synonym* or *derivedFrom*). The focus is on a small number of broad-coverage predicates, namely, `type, locationOf, usedFor, capableOf, hasPart`. ConceptNet is one of the highest-quality and most widely used CSK resources.

WebChild. WebChild has been automatically constructed from book n-grams (and, to a smaller degree, image tags) by a pipeline of information extraction, statistical learning and constraint reasoning methods [53, 127]. WebChild contains ca. 13 million assertions, and covers 20 distinct predicates such as `hasSize, hasShape, physicalPartOf, memberOf`, etc. It is the biggest of the publicly available commonsense knowledge bases, with the largest slice being on part-whole knowledge [54]. However, a large mass of WebChild's contents is in the long tail of possible but not necessarily typical and salient properties. So it comes with a substantial amount of noise and non-salient contents.

Mosaic TupleKB. The Mosaic project at AI2 aims to collect commonsense knowledge in various forms, from grounded triples to procedural knowledge with first-order logic. TupleKB, released as part of this ongoing project, is a collection of triples for the science domain, compiled by generating domain-specific queries and extracting assertions from the resulting web pages. A subsequent cleaning step, based on integer linear programming, clusters triples into groups. TupleKB contains ca. 280,000 triples for ca. 30,000 subjects.

Wikidata. This collaboratively built knowledge base is mostly geared to organise encyclopedic facts about individual entities like people, places, organisations etc.

[57,131]. It contains more than 400 million assertions for more than 50 million items. This includes some world knowledge about general concepts, like `type` triples, but this coverage is very limited. For instance, Wikidata neither knows that birds can fly nor that elephants have trunks.

3.2.2 Use Cases of CSK

Commonsense knowledge and reasoning are instrumental in a variety of applications in natural language processing, computer vision, and AI in general. These include question answering, especially for general world comprehension [138] and science questions [47]. Sometimes, these use cases also involve additional reasoning (e.g., [55]), where CSK contributes, too. Another NLP application is dialogue systems and chatbots (e.g., [58]), where CSK adds plausibility priors to language generation.

For visual content understanding, such as object detection or caption generation for images and videos, CSK can contribute as an informed prior about spatial co-location derived, for example, from image tags, and about human activities and associated emotions (e.g., [56, 136, 137]). In such settings, CSK is an additional input to supervised deep-learning methods.

3.2.3 Information Extraction from Query Logs

Prior works have tapped into query logs for goals like query recommendation (e.g., [49]) and extracting semantic relationships between search terms, like synonymy and hypernymy/hyponymy (e.g., [10, 52, 94, 134]). The latter can be seen as gathering triples for CSK, but its sole focus is on the `(sub-)type` predicate – so the coverage of the predicate space is restricted to class/type taxonomies. Moreover, these projects were carried out on full query logs within industrial labs of search-engine companies. In contrast, Quasimodo addresses a much wider space of predicates and operates with an original way of sampling query-log-derived signals via auto-completion suggestions. To the best of our knowledge, no prior work has aimed to harness auto-completion for CSK acquisition (cf. [24]).

The methodologically closest work to ours is [95]. Like us, that work used interrogative patterns (e.g. "Why do ...") to mine query logs – with full access to the search-engine company's logs. Unlike us, subjects, typically classes/types such as "cars" or "actors", were merely associated with salient phrases from the log rather than extracting complete triples. One can think of this as organizing CSK in SP pairs where P is a textual phrase that comprises both predicate and object but cannot separate these two. Moreover, [95] restricted itself to the extraction stage and used simple scoring from query frequencies, whereas we go further by leveraging multi-source signals in the corroboration stage and refining the SPO assertions into semantic groups.

3.3 System Overview

Quasimodo is designed to cope with the high noise and potentially strong bias in online contents. It taps into query logs via auto-completion suggestions as a non-

standard input source. However, frequent queries – which are the ones that are visible through auto-completion – are often about sensational and untypical issues.

Therefore, Quasimodo combine a recall-oriented candidate gathering phase with two subsequent phases for cleaning, refining, and ranking assertions. Figure 3.1 gives a pictorial overview of the system architecture.

3.3.1 Candidate Gathering

In this phase, we extract candidate triples from some of the world's largest sources of the "wisdom of crowds", namely, search-engine query logs and question answering forums such as Reddit or Quora. While the latter can be directly accessed via search APIs, query logs are unavailable outside of industrial labs. Therefore, we creatively probe and sample this guarded resource using generating queries and observing auto-completion suggestions by the search engine. The resulting suggestions are typically among the statistically frequent queries.

As auto-completion works only for short inputs of a few words, we generate queries that are centred on candidate subjects, the S argument in the SPO triples that we aim to harvest.

Technical details are given in Section 3.4.

3.3.2 Corroboration

This phase is precision-oriented, aiming to eliminate false positives from the candidate gathering. We consider candidates as invalid for three possible reasons: 1) they do not make sense (e.g., programmers eat python); 2) they are not typical properties for the instances of the S concept (e.g., programmers drink espresso); 3) they are not salient in the sense that they are immediately associated with the S concept by most humans (e.g., programmers visit restaurants). To statistically check to which degree these aspects are satisfied, Quasimodo harnesses corroboration signals in a multi-source scoring step. This includes conventional sources like Wikipedia articles and books, which were used in prior works already, but also non-standard sources like image tags and answer snippets from search-engine queries.

Technical details are given in Section 3.5.

3.3.3 Ranking

To identify typical and salient triples, we devised a probabilistic ranking model with the corroboration scores as an input signal. This stage is described in Section 3.6.

3.3.4 Grouping

For this phase, we have devised a clustering method based on the model of tri-factorisation for matrix decomposition [43]. The output consists of groups of SO pairs and P phrases linked to each other. So we semantically organise and refine both the concept arguments (S and O) in a commonsense triple and the way the predicate (P) is expressed in language. Ideally, this would canonicalise all three components, in analogy to what prior works have achieved for entity-centric encyclopedic knowledge

bases (e.g., [67, 125]). However, commonsense assertions are rarely as crisp as facts about individual entities, and often carry subtle variation and linguistic diversity (e.g., `live in` and `roam in` for animals being near-synonymous but not quite the same). Our clustering method also brings out refinements of predicates. This is in contrast to prior work on CSK which has mostly restricted itself to a small number of coarse-grained predicates like `partOf, usedFor, locatedAt`, etc. Technical details are given in Section 3.7.

3.4 Candidate Gathering

The key idea for this phase is to utilise questions as a source of human commonsense. For example, the question *"Why do dogs bark?"* implicitly conveys the user's knowledge that dogs bark. Questions of this kind are posed in QA forums, such as Reddit or Quora, but their frequency and coverage in these sources alone are not sufficient for building a comprehensive knowledge base. Therefore, we additionally tap into query logs from search engines, sampled through observing auto-completion suggestions. Although most queries merely consist of a few keywords, there is a substantial fraction of user requests in interrogative form [132].

3.4.1 Data Sources

Quasimodo exploits two data sources: (i) QA forums, which return questions in user posts through their search APIs, and (ii) query logs from major search engines, which are sampled by generating query prefixes and observing their auto-completions.

QA forums. We use four different QA forums: Quora, Yahoo! Answers (https://answers.yahoo.com and https://webscope.sandbox.yahoo.com), Answers.com, and Reddit. The first three are online communities for general-purpose QA across many topics, and Reddit is a large discussion forum with a wide variety of topical categories.

Search engine logs Search engine logs are rich collections of questions. While logs themselves are not available outside of industrial labs, search engines allow us to glimpse at some of their underlying statistics by auto-completion suggestions. Figure 3.2 shows an example of this useful asset. Quasimodo utilises Google and Bing, which typically return 5 to 10 suggestions for a given query prefix. In order to obtain more results, we recursively probe the search engine with increasingly longer prefixes that cover all letters of the alphabet, until the number of auto-completion suggestions drops below 5. For example, the query prefix *"why do cats"* is expanded into *"why do cats a"*, *"why do cats b"*, and so on.

We intentionally restrict ourselves to query prefixes in an interrogative form, as these are best suited to convey commonsense knowledge. In contrast, simple keyword queries are often auto-completed with references to prominent entities (celebrities, sports teams, product names, etc.), given the dominance of such queries in the overall Internet (e.g., the query prefix "cat" is expanded into "cat musical"). These very frequent queries are not useful for CSK acquisition.

In total, we collected ca 20 million questions from autocompletion.

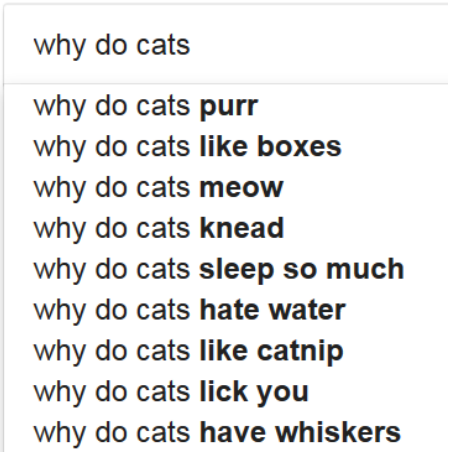

Figure 3.2: A glimpse into a search-engine query log

Pattern	Frequency
why is	40%
why are	15%
how is	11%
why do	9%
why does	7%
how does	6%
how do	3%
why isn't	3%
how are	3%
how can	1%
why	1%
why aren't	1%
why can't	1%
why don't	<1%
why doesn't	<1%
why can	<1%

Table 3.1: Question patterns for candidate gathering

3.4.2 Question Patterns

We performed a quantitative analysis of frequent question words and patterns on Reddit. As a result, we decided to pursue two question words, *Why* and *How*, in combination with the verbs *is, do, are, does, can, can't*, resulting in 16 patterns in total. Their relative frequency in the question set that we gathered is shown in Table 3.1. For forums, we performed title searches centred around these patterns. For search engines, we appended subjects of interest to the patterns for query generation (e.g., "Why do cats") for cats as subject. The subjects were chosen from the common nouns extracted from WordNet [87].

	Question	Statement
(1)	why *is* voltmeter not connected in series	voltmeter *is* not connected in series
(2)	why *are* chimpanzees endangered	chimpanzees *are* endangered
(3)	why *do* men have nipples	men have nipples
(4)	why *are* elephant seals mammals	elephant seals *are* mammals
(5)	why *is* becoming a nurse in france hard	becoming a nurse in france *is* hard

Table 3.2: Examples of questions and statements

3.4.3 From Questions to Assertions

Open information extraction (Open IE) [84], based on patterns, has so far focused on assertive patterns applied to assertive sentences. In contrast, we deal with interrogative inputs, facing new challenges. We address these issues by rule-based rewritings. As we need to cope with colloquial or even ungrammatical language as inputs, we do not rely on dependency parsing but merely use part-of-speech tags for rewriting rules. Primarily, rules remove the interrogative words and re-order subject and verb to form an assertive sentence. However, additional rules are needed to cast the sentence into a naturally phrased statement that OpenIE can deal with. Most notably, auxiliary verbs like "do" need to be removed, and prepositions need to be put in their proper places, as they may appear at different positions in interrogative vs assertive sentences. Table 3.2 shows some example transformations, highlighting the modified parts.

After transforming questions into statements, we employ the Stanford OpenIE tool [48] and OpenIE5.0 [32, 92, 112, 113] to extract triples from assertive sentences. We leave the choice of the predicates to these tools. When several triples with the same S and O are extracted from the same sentence, we retain only the one with the longest P phrase.

The resulting extractions are still noisy, but this is taken care of by the subsequent stages of corroboration, ranking and grouping, to construct a high-quality CSKB.

3.4.4 Output Normalisation

The triples produced by OpenIE exhibit various idiosyncrasies. For cleaning them, we apply the following normalisation steps:

1. Replacement of capital letters with lower case characters.
2. Replacement of plural subjects with singular forms.
3. Replacement of verb inflections by their infinitive (e.g., *are eating* $\rightarrow$ *eat*).
4. Removal of modalities (always, sometimes, occasionally, ...) in the predicate and object. These are kept as modality qualifiers.
5. Removal of negation, put into a dedicated qualifier as negative evidence.

6. Replacement of generic predicates like *are*, *is* with more specific ones like *hasColor*, *hasBodyPart*, depending on the object.
7. Removal of adverbs and phatic expressions (e.g., "so", "also", "very much" etc.).

We completely remove triples containing any of the following:

1. subjects outside the initial seed set,
2. personal pronouns ("my", "we"), and
3. a shortlist of odd objects (e.g., xbox, youtube, quote) that frequently occur in search results but do not indicate commonsense properties.

The output of this phase are tuples of the form *(Subject, Predicate, Object, Modality, Negativity, Source, Score)*, for instance, *(lion, hunts, zebra, often, positive, Google, 0.4)*.

3.4.5 Generation of New Subjects

After the normalisation phase, we only keep subjects that were in a predefined list. This filtering allows us to consider only non-esoteric subjects as potential subjects are infinite and often, do not generalise to a more global context. For example, we removed *dancing nun*, *video help* and *Twitter user type*. However, we also removed subjects that could be interesting to consider. These ignored subjects that were not present in ConceptNet nor WordNet are very diverse. We give here some examples of potentially useful subjects that we manually extracted and classified for this list:

- Fictional characters such as *John Snow* or *Harry Potter*
- Notorious people such as *Donald Trump* or *Hillary Clinton*
- Actions such as *eating asparagus* (thus, we can extend ConceptNet actions) or *drinking tea*
- Objects such as *energy drink* or *mechanical keyboards*
- Concepts such as *gas prices* or *social media*
- etc.

These subjects are of primary interest as people often talk about them. So, we extract them using the following procedure:

- During the subject removal step, count for each ignored subject how often it appears.
- Filter ignored subjects that appear less than a certain threshold (10 appearances in our experiments)
- Filter subjects that contain personal words (such as *my*, *your* and *its*)
- Filter subjects that start by a determinant (such as *an elephant ear*)

Finally, we get a list of new subjects that the pipeline is going to use during the next iteration.

3.5 Corroboration

The output of the candidate gathering phase is bound to be noisy and contains many false positives. Therefore, we scrutinise the candidate triples by obtaining corroboration signals from a variety of additional sources. Quasimodo queries sources to test the occurrence and obtain the frequency of SPO triples. These statistics are fed into a logistic-regression classifier that decides on whether a triple is accepted or not.

The goal of this stage is to validate whether candidate triples are plausible, i.e., asserting them is justified based on several corroboration inputs.

3.5.1 Wikipedia and Simple Wikipedia

For each SPO candidate, we probe the article about S and compute the frequency of co-occurring P and O within a window of n successive words (where $n = 5$ in our experiments). If no evidence can be found, we give a score of zero.

3.5.2 Answer Snippets From Search Engine

We generate Google queries using the S and O arguments of a triple as keywords, and analyse the top-100 answer snippets. The frequency of snippets containing all of S, P and O is viewed as an indicator of the triple's validity. As search engines put tight constraints on the number of allowed queries per day, we can obtain this signal only for a limited subset of candidate assertions. We prioritise the candidates for which the other sources (Wikipedia, etc.) yield high evidence.

3.5.3 Google Books

We create queries to the Google Books API by first forming disjunctions of surface forms for each of S, P and O, and then combining these into conjunctions. For instance, for the candidate triple (lion, live in, savanna), the query is "lion OR lions live OR lives in savanna OR savannas". As we can use the API only with a limited budget of queries per day, we prioritised candidate triples with high evidence from other sources (Wikipedia, etc.).

3.5.4 Image Tags From OpenImages and Flickr

OpenImages is composed of ca. 20.000 classes used to annotate images. Human-verified tags exist for ca. 5.5 million images. Quasimodo checks for co-occurrences of S and O as tags for the same image and computes the frequency of such co-occurrences. For Flickr, we use its API to obtain clusters of co-occurring tags. Individual tags are not available through the API. We test for the joint occurrence of S and O in the same tag cluster.

Source	Fraction
Google Auto-complete	84%
Answers.com	9%
Reddit	7%
Quora	2%
Yahoo! Answers, Bing Auto-complete	< 1%
CoreNLP Extraction	80%
OpenIE5 Extraction	27%
Custom Extraction	9%

Table 3.3: Proportions of candidate triples by sources

3.5.5 Captions From Google's Conceptual Captions Dataset

Google's Conceptual Captions dataset (https://ai.google.com/research/ConceptualCaptions) is composed of around three millions image descriptions. Using a method similar to the one used for Wikipedia, we check for a given fact SPO the concurrence of S with P and O.

3.5.6 What Questions

During the candidate gathering presented in Section 3.4, we focused on *why* and *how* questions. One of their main advantages is that they contain the subject, the predicate, and the object. As a comparison, *what* questions generally contain only two of these three elephants. For example, *what does an elephant eat* only contains the subject (elephant) and the predicate (eat) of a triple. Still, *what* questions can be used to enforce the relationship between two components of a triple. Thus, we use it as an additional signal.

Table 3.3 gives the fractions of candidate triples for which each of the sources contributes to scoring signals.

3.5.7 Classifier Training and Application

We manually annotated a sample of 700 candidate triples obtained in the candidate gathering phase (In comparison, TupleKB required crowd annotations for 70,000 triples). These are used to train a logistic regression, which gives us a precision of 61%.

Features. The features of our model are the signals from the gathering phase and the corroboration phase. Besides, we consider the following additional features:

- Number of sentences from which the statement was extracted
- The number of modalities associated with the statement
- The sources that do not generate the statement
- The patterns used to generate a statement

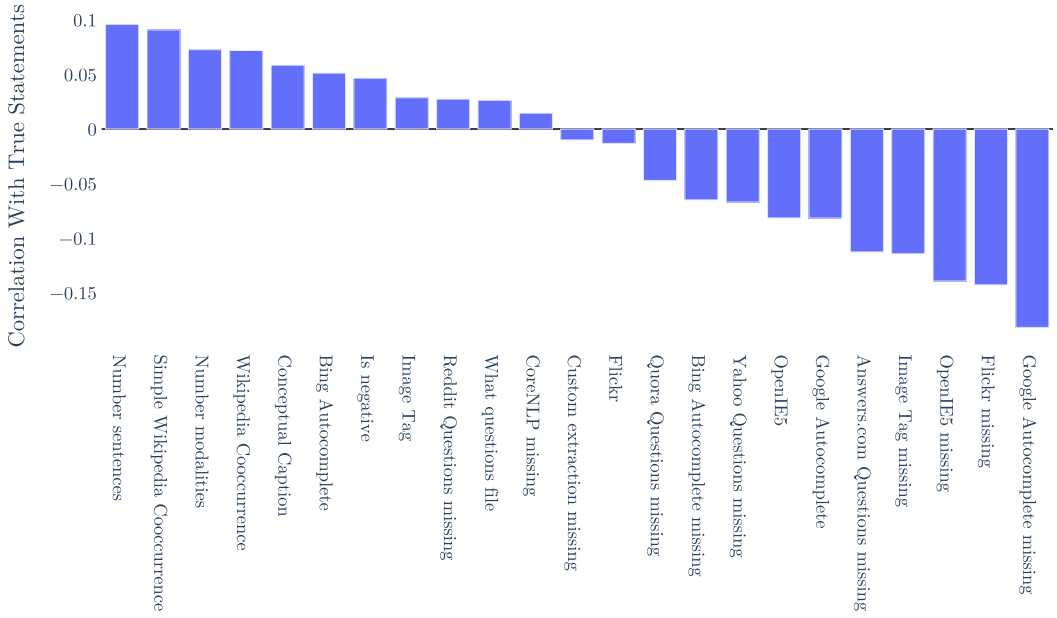

Figure 3.3: Correlation of features with statements labelled as true

In Figure 3.3, we display the correlation of some features with the label. As expected, when a feature is missing, it has a negative impact on a statement. It is important to notice that Reddit generally generates noisy statements, and thus harms the final score. Besides, we can see a major difference between OpenIE and CoreNLP. In Figure 3.3, we noticed that CoreNLP generates more facts than OpenIE5. However, here we can see that OpenIE5 facts are globally more precise. Google auto-completion has a negative correlation. This is what we expected as the score for this feature is the rank of the suggestion.

Classifier Comparison.

When we choose a classifier, we are not only interested in its precision, but also in its ability to rank salient features. We evaluate the ranking given by three classifiers (logistic regression, naive Bayes and AdaBoost) by their recall they obtain on the tasks presented in Section 3.8.2 and Section 3.8.3. The results are presented in Table 3.4. All in all, the logistic regression outperforms the other classifiers. AdaBoost beats it in two settings by a small margin. However, the logistic regression has two advantages over AdaBoost: it is faster, and it gives scores that cover the entire range between 0 to 1. Thus, it can represent extreme cases, where there are plenty of evidence. In comparison, AdaBoost scores are generally close to the decision boundary, 0.5.

	Taboo Guessing Game				MTurks Recall			
	Top 5		Top 10		Top 5		Top 10	
	P and O	O only	P and O	O only	strict	relaxed	strict	relaxed
Logistic Regression	**8.16%**	**7.83%**	**11.8%**	**11.4%**	6.98%	**8.69%**	9.82%	**13.6%**
Naive Bayes	5.43%	5.26%	9.70%	9.28%	4.95%	7.66%	7.48%	12.2%
AdaBoost	7.37%	7.11%	10.5%	10.2%	**7.03%**	8.28%	**10.0%**	12.8%

Table 3.4: Comparison of classifiers rankings

3.6 Ranking

3.6.1 The Plausibility-Typicality-Saliency Approach

At first, we decided to deduce three metrics out of the score given by the corroboration phase.

We refer to the scores resulting from the corroboration stage as plausibility scores π. These plausibility scores are essentially combinations of frequency signals. Frequency is an important criterion for ranking CSK, yet CSK has other vital dimensions.

In this section we propose two probabilistic interpretations of the scores π, referred to as τ ("typicality") and σ ("saliency"). Intuitively, τ enables the ranking of triples by their informativeness for their subjects. Conversely, σ enables the ranking of triples by the informativeness of their p, o part.

To formalise this, we first define the probability of a triple *spo*.

$$\mathbf{P}[s,p,o] = \frac{\pi(spo)}{\Sigma_{x \in KB} \ \pi(x)}.$$

Then, we compute τ and σ as:

$$\tau(s,p,o) = \mathbf{P}[p,o \mid s] = \frac{\mathbf{P}[s,p,o]}{\mathbf{P}[s]}.$$

$$\sigma(s,p,o) = \mathbf{P}[s \mid p,o] = \frac{\mathbf{P}[s,p,o]}{\mathbf{P}[p,o]}$$

In each case, the marginals are:

$$\mathbf{P}[p,o] = \Sigma_{s \in subjects} \ \mathbf{P}[s,p,o] \tag{3.6.1}$$

$$\mathbf{P}[s] = \Sigma_{p,o \in (predicates,objects)} \ \mathbf{P}[s,p,o] \tag{3.6.2}$$

At the end of this stage, each triple *spo* is annotated with three scores: an internal plausibility score π, and two conditional probability scores τ and σ, which we subsequently use for ranking.

The M-Turks experiments presented in Section 3.8.2 are based on these three metrics. However, afterwards, we observed limitations in the saliency and typicality score that pushed us to remove them from the newest versions of Quasimodo.

First, the plausibility score is, in practice, close to what we expect from the saliency score. Ranking the facts by plausibility gives us the most salient properties first.

Second, due to the wide variety of predicates and objects, a slight modification of one of them can have a tremendous effect on all scores. For example, let us consider the facts *(elephants, are afraid of, mice)* and *(elephants, are dreadfully afraid of, mice)*. The first one has a very high corroboration score as it appears a lot in many sources. However, the second one is much rarer, and so its corroboration score is lower. In the end, we get a massive gap in scores between the two facts, whereas they carry the same information.

Finally, the scores given here are probabilities. This means that they can be very low and difficult to interpret as so. In particular, the saliency makes sense only with a given predicate-object and the typicality only with a given subject. One way to overcome the small scores problem would be to normalise them in such a way that the maximum score is 1. However, the problem of global comparability remains.

3.6.2 The Smoothed Plausibility-Typicality-Saliency Approach

One way to deal with the problem of the diversity of the predicate-objects would be to "smooth" the scores by averaging the scores of similar facts.

Let us suppose we are given a distance function d_{spo} that gives us the distance between two facts. For example, this distance can be a Jaccard distance or a word2vec cosine distance. We could now adapt our definitions of $\mathbf{P}[s, p, o]$ in the following way:

$$\mathbf{P}_{smooth}[s, p, o] \propto \sum_{d_{spo}(spo, spo') < \lambda} \mathbf{P}[s, p, o] * sim_{spo}(spo, spo')$$

where sim_{spo} is a similarity measure based on d_{spo} (here $1 - d_{spo}$).

In this expression, we sum over all facts that are close enough to the considered fact (the maximum distance is symbolised by the constant λ). This condition could also be replaced by a condition over the number of similar facts considered by defining λ as $max(\{\lambda' \mid \text{size}(\{spo' \mid d_{spo}(spo, spo') < \lambda'\}) < C\})$ (where C is the maximum number of facts to consider).

In the same way, one could define $\mathbf{P}_{smooth}[p, o]$ from a distance function d_{po} that gives us the distance between two predicate-object pairs:

$$\mathbf{P}_{smooth}[p, o] \propto \sum_{d_{po}(po, po') < \lambda} \mathbf{P}[p, o] * sim_{po}(po, po')$$

As subjects allow less variety than the predicates and the objects, we did not consider smoothing to be useful on them. Finally, one can deduce the smoothed plausibility, typicality and saliency:

$$\pi_{smooth}(s, p, o) = \mathbf{P}_{smooth}[s, p, o]$$

$$\tau_{smooth}(s, p, o) = \mathbf{P}_{smooth}[p, o \mid s] = \frac{\mathbf{P}_{smooth}[s, p, o]}{\mathbf{P}[s]}.$$

$$\sigma_{smooth}(s,p,o) = \mathbf{P}_{smooth}[s \mid p,o] = \frac{\mathbf{P}_{smooth}[s,p,o]}{\mathbf{P}_{smooth}[p,o]}$$

Although this approach has the advantage of dealing with the variety of predicate-objects pairs, some problems remain: The first is that the definitions of distance functions are not trivial. Indeed, this task is very close to the one of semantic textual similarity, which is addressed through datasets such as SentEval [33] or the Quora Question Pairs (https://www.kaggle.com/c/quora-question-pairs), and is known to be quite complicated.

Second, computation problems can arise. Depending on the distance function and the number of triples generated, computing all similarities can be very expensive.

Third, the problem of global comparison of saliency and typicality is still present.

All in all, we observed that the smoothing improve the ranking for the saliency and the typicality. However, the problems we mentioned are still present at it would be future work to investigate scoring methods further.

3.7 Grouping

The corroboration stage of Quasimodo aimed to remove overly generic and overly specific assertions, but still yields diverse statements of different granularities with a fair amount of semantic redundancy. For instance, *hamsters are cute*, *hamsters are cute pets*, and *hamsters are cute pets for children* are all valid assertions, but more or less reflect the same commonsense property. Such variations occur with both O and P arguments, but less so with the subjects S as these are pre-selected seeds in the candidate gathering stage.

To capture such redundancies while preserving different granularities and aspects, Quasimodo groups assertions into near-equivalence classes. At the top level, Quasimodo provides groups as entry points and then supports a meta-predicate *refines* for more detailed exploration and use-cases that need the full set of diversely phrased assertions.

3.7.1 Soft Co-Clustering

Our goal is to identify diverse formulations for both predicates P and subject-object pairs SO. The prior work on TupleKB has used ILP-based clustering to canonicalise predicates. However, this enforces hard grouping such that a phrase belongs to exactly one cluster. With our rich data, predicates such as "chase" or "attack" can refer to very different meanings, though: predators chasing and attacking their prey, or students chasing a deadline and attacking a problem. Analogously, S and O arguments also have ambiguous surface forms that would map to different word senses.

WebChild [53] has attempted to solve this issue by comprehensive word sense disambiguation (see [89] for a survey), but this is an additional complexity that eventually resulted in many errors. Therefore, we aim for the more relaxed and – in our findings – more appropriate objective of computing *soft clusters* where the same phrase can belong to different groups (to different degrees). As the interpretation of

P phrases depends on the context of their S and O arguments, we cast this grouping task into a *co-clustering* problem where SO pairs and P phrases are jointly clustered.

3.7.2 Tri-Factorisation of SO-P Matrix

Our method for soft co-clustering of SO pairs and P phrases is non-negative matrix tri-factorisation; see [43] for mathematical foundations. We aim to compute clusters for SO pairs and clusters for P phrases and align them with each other when meaningful. For example, the SO pairs *student problem* and *researcher problem* could be grouped together and coupled with a P cluster containing *attack* and a second cluster containing *solve*.

This goal alone would suggest a standard form of factorizing a matrix with SO pairs as rows and P phrases as columns. However, the number of clusters for SO pairs and for P phrases may be very different (because of different degrees of diversity in real-world commonsense), and decomposing the matrix into two low-rank factors with the same dimensionality would not capture this sufficiently well. Hence our approach is tri-factorisation where the number of (soft) clusters for SO pairs and for P phrases can be different.

We denote the set of SO pairs and P phrases, as observed in the SPO triples after corroboration, as an $m \times n$ matrix $M_{m\times n}$, where element M_{ij} denotes the corroboration score of the triple with SO_i and P_j.

We factorise M as follows:

$$M_{m\times n} = U_{m\times k} \times W_{k\times l} \times V^T_{l\times n}$$

where the low-rank dimensionalities k and l are hyper-parameters standing for the number of target SO clusters and target P clusters and the middle matrix W reflects the alignments between the two kinds of clusters. The optimisation objective in this tri-factorisation is to minimise the data loss in terms of the Frobenius norm, with non-negative U, W, V and orthonormal U, V:

$$\begin{aligned} \textit{Minimise} \quad & \|M - U_{m\times k} \times W_{k\times l} \times V^T_{l\times n}\|_F \\ s.t. \quad & U^T U = I, \quad V^T V = I \\ & U, V, W \geq 0 \end{aligned} \tag{3.7.1}$$

We can interpret $U_{i\mu}$ as a probability of the membership of the i^{th} SO pair in the μ^{th} SO cluster. Similarly, $V_{j\nu}$ represents the probability of cluster membership of the j^{th} P phrase to the ν^{th} P cluster. The coupling of SO clusters to P clusters is given by the $W_{k\times l}$ matrix, where the μ^{th} SO cluster is linked to the ν^{th} P cluster if $W_{\mu\nu} > 0$.

Each SO pair and P phrase have a certain probability of belonging to an SO and P cluster, respectively. Hence, using a thresholding method, we assign SO_i to the μ^{th} cluster if $U_{i\mu} > \theta$ and P_j to the ν^{th} cluster if $V_{j\nu} > \theta$, in order to arrive at crisper clusters. In our experiments, we set the thresholds as follows: for the λ^{th} SO cluster, we set $\theta_\lambda = \delta \cdot max_i U_{i\lambda}$, and for the λ^{th} P cluster, we set $\theta_\lambda = \delta \cdot max_i V_{i\lambda}$.

Domains	#SPO	k	l	ρ	# SO/SO cluster avg.	# SO/SO cluster max	# P/P cluster avg.	# P/P cluster max	# P clusters/P avg.
Animals	201942	3500	2000	0.10	38.46	383	2.8	24	1.5
Persons	218924	5000	2000	0.10	11.7	235	4.7	67	1.5
Medicine	91184	3000	1800	0.15	45.17	171	2.91	31	1.3
Sport	30794	1500	400	0.15	13.3	73	3.8	15	1.14
macro-avg. (over	all 49 domains)	1457.8	603.7	0.12	33.97	123.0	3.5	24.8	1.24

Table 3.5: Statistics for SO clusters and P clusters for vertical domains Animals and Occupations

P clusters	SO clusters
make noise at, be loud at, make noises at, croak in, croak at, quack at	fox-night, frog-night, rat-night, mouse-night, swan-night, goose-night, chicken-night, sheep-night, donkey-night, duck-night, crow-night
misbehave in, talk in, sleep in, be bored in, act out in, be prepared for, be quiet in, skip, speak in	student-class, student-classes, student-lectures
diagnose, check for	doctor-leukemia, doctor-reflexes, doctor-asthma, doctor-diabetes, doctor-pain, doctor-adhd

Table 3.6: Anecdotal examples of coupled SO clusters and P clusters from vertical domains Animals and Occupations

By varying the common thresholding parameter δ, we tune the cluster assignments of SO pairs and P phrases based on the empirical perplexity of the resulting clusters.

This way, we found an empirically best value of $\delta = 0.1$.

The factor matrices in this decomposition should intuitively be sparse, as each SO pair would be associated with only a few P clusters and vice versa. To reward sparsity, L_1 regularisation is usually considered for enhancing the objective function. However, the L_1 norm makes the objective non-differentiable, and there is no analytic solution for the tri-factorisation model. Like most machine-learning problems, we rely on stochastic gradient descent (SGD) to approximately solve the optimisation in Equation 3.7.1.

For this reason, we do not use $L1$ regularisation. Our SGD-based solver initialises the factor matrices with a low density of non-zero values, determined by a hyper-parameter ρ for the ratio of non-zero matrix elements. The overall objective function then is the combination of data loss and sparseness:

$$\textit{Maximise} \quad \frac{\textit{fraction of zero elements } (W)}{\textit{data loss by Equation } 3.7.1}$$

All hyper-parameters – the factor ranks k and l, and the sparseness ratio ρ – are tuned by performing a grid search.

3.8 Experimental Evaluation

3.8.1 Implementation

Seeds. As seeds for subjects, we use a combination of concepts from ConceptNet, combined with nouns extracted from WordNet, resulting in a total of around 120,000 subjects.

Candidate Gathering. In this phase Quasimodo collected ca. 1.4 million questions from Quora (from the Kaggle "Quora Question Pair Challenge" https://www.kaggle.com/c/quora-question-pairs), 600,000 questions from Yahoo! Answers, 2.5 million questions from Answers.com (via its sitemap), and 3.5 million questions from a Reddit dump (with a choice of suitable subreddits). From auto-completion suggestions, we obtained ca. 20 million questions from Google and 200,000 questions from Bing.

After applying the rewriting of questions into statements and running OpenIE, we obtained 11 million candidate triples; the subsequent normalisation further reduced this pool to ca. 4.4 million triples.

Corroboration. The logistic regression model assigned a mean score of 0.23, with a standard deviation of 0.11. For high recall, we do not apply a threshold in this phase but utilise the scores for ranking in our evaluations.

Grouping. We performed this step on the top-50% triples, ordered by corroboration scores, amounting to ca. 1 million assertions. For efficient computation, we sliced this data into 49 *basic domains* based on the WordNet domain hierarchy [50]. To this end, we mapped the noun sense of each assertion subject to WordNet and assigned all triples for the subject to the respective domain (e.g., animals, plants, earth, etc.). The five largest domains are *earth, chemistry, animal, biology, and person*, containing on average 3.9k subjects and 198k assertions. We performed co-clustering on each of these slices, where hyperparameters were tuned by grid search.

Table 3.5 gives hyperparameter values and cluster-specific statistics of the co-clustering for three domains: number of assertions (#SPO); the co-clustering hyperparameters SO clusters (k), P clusters (l) and sparseness ratio (ρ); the average number of elements per cluster for both SO and P clusters; and the average number of P-clusters per predicate. Additionally, we provide macro-averaged statistics for all 49 domains. Table 3.6 shows anecdotal examples of co-clusters for illustration.

Quasimodo CSKB. The resulting knowledge base contains ca. 4.4 million assertions for 103,000 subjects. Quasimodo is accessible online (https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/ research/yago-naga/ commonsense/quasimodo/).

Run-Time. One of the expensive components of Quasimodo is the probing of Google auto-completions. This was carried out within the allowed query limits over a long time. Bing auto-completion was accessed through the Azure API. Another expensive component is co-clustering of all 49 domains, which takes total 142 hours in an Intel Xeon(R)(2 cores@3.20GHz) server (average 3.14 hours/ slice). All other components of Quasimodo run within a few hours at most. We use some caching mechanisms to make the pipeline faster to run again.

About Caching. We cached several components of our pipeline that do not need to be recomputed every time. First, the result of the autocompletion is cached as we

	Full KB				
	#S	#P	#P≥10	#SPO	#SPO/S
ConceptNet-full@en	842,532	39	39	1,334,425	1.6
ConceptNet-CSK@en	41,331	19	19	214,606	5.2
TupleKB	28,078	1,605	1,009	282,594	10.1
WebChild	55,036	20	20	13,323,132	242.1
Quasimodo	103,218	102,801	7,534	4,369,850	42.3

Table 3.7: Statistics for different full KBs

KB	animals		occupations	
	#S	#SPO	#S	#SPO
ConceptNet-full@en	50	2,678	50	1,906
ConceptNet-CSK@en	50	1,841	50	1,495
TupleKB	49	16,052	38	5,321
WebChild	50	27,223	50	26,257
Quasimodo	50	67,111	50	28,747

Table 3.8: Statistics for two slices on animals and occupations on different KBs

suppose it does not change. In future work, we could investigate the potential temporal changes of the autocompletion, in particular when new concepts emerge (like the coronavirus). Next, we cached the transformation of questions into statements. This step can be lengthy because we deal with more than ten million questions. The OpenIE step is also saved as it does not change through time (except if the version of the software changes) and is very costly. During the corroboration phase, the calls to APIs such as Wikipedia and Google Books are saved. Finally, we save intermediate states of the pipeline, so we can run it from a particular step (like ranking, for example, if we only update the scoring model).

3.8.2 Intrinsic Evaluation

We evaluate four aspects of Quasimodo: 1) size of the resulting CSKB, 2) quality, 3) recall, and 4) cluster coherence.

Size. We compare KBs in Table 3.7 by the number of subjects (#S), the number of predicates (#P), predicates occurring at least ten times (#P≥10), and the number of triples (#SPO). For Quasimodo we exclude all triples with *isA / type* predicate denoting subclass-of or instance-of relations, as these are well covered in traditional knowledge resources like WordNet, Wikidata and Yago. We also compare in Table 3.8, on two vertical domains: assertions for the 50 most popular animals and 50 most popular occupations, as determined by frequencies from Wiktionary.

For ConceptNet, we report numbers for the full data including *isA / type* and *related* and other linguistic triples (e.g., on etymology) imported from DBpedia, WordNet and Wiktionary (ConceptNet-full), and for the proper CSK core where these relations are removed (ConceptNet-CSK).

Table 3.7 and Table 3.8 convey that Quasimodo has more abundant knowledge

per subject than all other resources except for WebChild. The advantage over the manually created ConceptNet becomes particularly evident when looking at the two vertical domains, where ConceptNet-CSK contains less than 10% of the assertions that Quasimodo knows.

Quality. On the version of Quasimodo from our original paper, we asked MTurk crowd workers to evaluate the quality of CSK assertions along three dimensions: 1) meaningfulness, 2) typicality, 3) saliency. Meaningfulness denotes if a triple is conveys meaning at all, or is absurd; typicality denotes if most instances of the S concept have the PO property; saliency captures if humans would spontaneously associate PO with the given S as one of the most essential traits of S.

For each evaluated triple, we obtained two judgments for the three aspects, each graded on a scale from 1 (lowest) to 5 (highest). A total of 275 crowd workers completed the evaluation, with mean variance 0.70 on their ratings from 1 to 5 indicating good inter-annotator agreement.

We sampled triples from the different CSKBs under two settings: In *comparative sampling*, we sampled triples for the same 100 subjects (50 popular occupations and 50 popular animals) across all KBs. For subject and each KB we considered the top-5-ranked triples as a pool, and uniformly sampled 100 assertions for which we obtain crowd judgement. For Quasimodo, as the rankings by typicality τ and by saliency σ differ, this sampling treated Quasimodo-τ and Quasimodo-σ as distinct CSKBs. This setting provides a side-by-side comparison of triples for the same subjects.

In *horizontal sampling*, we sampled each KB separately; so they could differ on the evaluated subjects. We considered the top 5 triples of all subjects present in each KB as a pool and picked samples from each KB uniformly at random. This evaluation mode gave us insights into the average quality of each KB. Note that it gives KBs that have fewer long-tail subjects an advantage, as triples for long-tail subjects usually receive lower human scores. Again, we considered Quasimodo rankings by τ and σ as distinct CSKBs.

The results of these evaluations are shown in Figure 3.4 and Figure 3.5. With comparative sampling, Quasimodo-τ significantly outperforms both WebChild and TupleKB, and nearly reaches the quality of the human-generated ConceptNet. In horizontal sampling mode, Quasimodo-τ outperforms WebChild along all dimensions and outperforms TupleKB in all dimensions but saliency. This is remarkable given that Quasimodo in our original paper was three times bigger than ConceptNet, and is therefore penalised with horizontal sampling by its much larger number of long-tail subjects. In both evaluations, Quasimodo-τ significantly outperforms Quasimodo-σ in terms of meaningfulness and typicality. Regarding saliency, the results are mixed, suggesting that further research on ranking models would be beneficial.

Recall. To compare the recall (coverage) of the different CSKBs, we asked crowd workers at MTurk to make statements about 50 occupations and 50 animals as subjects. We asked to provide short but general sentences, as spontaneous as possible to focus on typical and salient properties. Together with these instructions, we gave three examples for elephants (e.g., "elephants are grey", "elephants live in Africa") and three examples for nurses. For each subject, crowd workers had four text fields to complete, which were pre-filled with "[subject] ...". Each task was handed out six times; so in total, we obtained 2,400 simple sentences on 100 subjects.

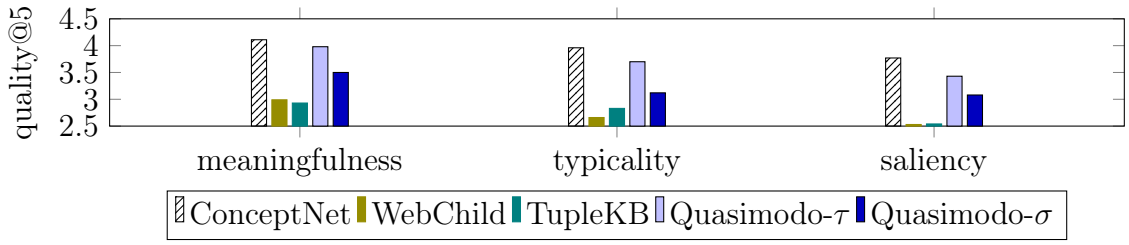

Figure 3.4: Quality for comparative sampling

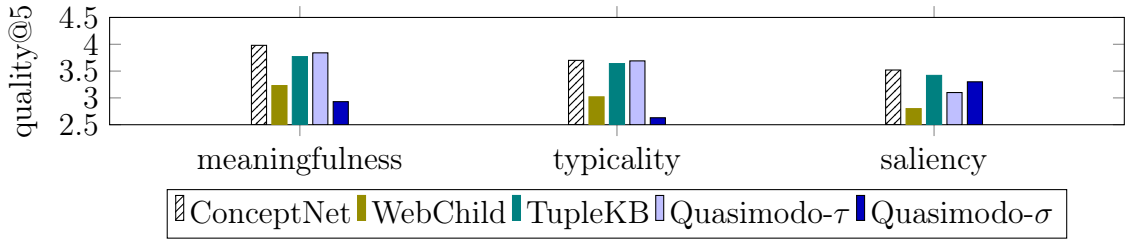

Figure 3.5: Quality for horizontal sampling

We computed CSKB recall w.r.t. these crowd statements in two modes. In the *strict* mode, we checked for each sentence if the KB contains a triple (for the same subject) where both predicate and object are contained in the sentence and, if so, computed the word-level token overlap between PO and the sentence. In the *relaxed* setting, we checked separately if the KB contains an S-triple whose predicate appears in the sentence, and if it contains an S-triple whose object appears in the sentence. The results are shown in Figure 3.6. In terms of this coverage measure, Quasimodo outperforms the other CSKBs by a large margin, in both strict and relaxed modes and also when limiting ourselves to the top-5 highest-ranked triples per subject.

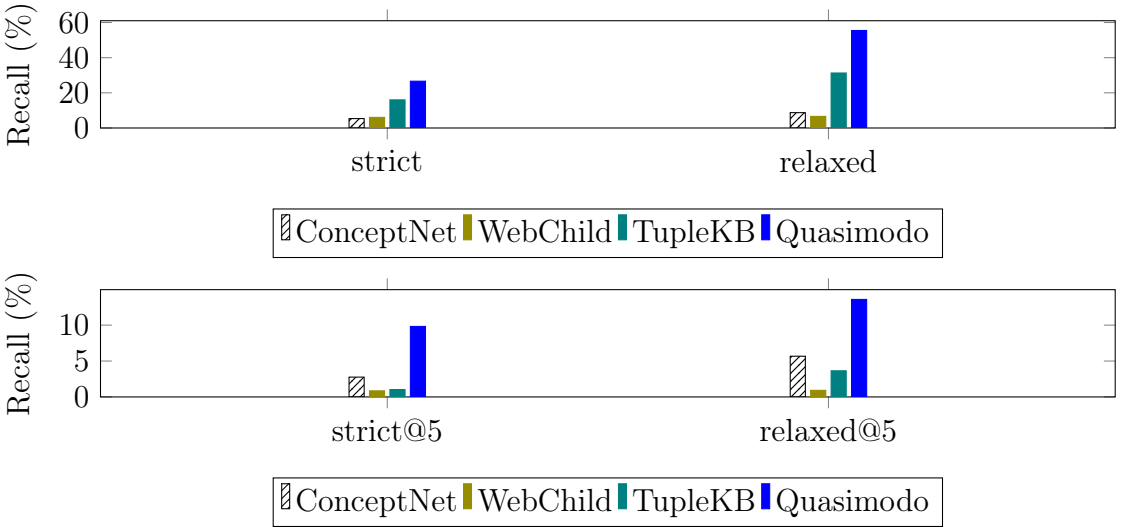

Figure 3.6: Recall evaluation

Quasimodo	ConceptNet	WebChild	TupleKB
(hasPhysicalPart, trunk)	(AtLocation, africa)	(quality, rare)	(has-part, brain)
(hasPhysicalPart, ear)	(HasProperty, cute)	(trait, playful)	(drink, water)
(live in, zoo)	(CapableOf, remember water source)	(size, large)	(prefer, vegetation)
(love, water)	(HasProperty, very big)	(state, numerous)	(eat, apple)
(be in, circus)	(CapableOf, lift logs from ground)	(quality, available)	(open, mouth)

Quasimodo	ConceptNet	WebChild	TupleKB
(help, people)	(HasA, private life)	(emotion, euphoric)	(complete, procedure)
(stand long for, surgery)	(CapableOf, attempt to cure patients)	(quality, good)	(conduct, examination)
(learn about, medicine)	(AtLocation, golf course)	(trait, private)	(get, results)
(cure, people)	(CapableOf, subject patient to long waits)	(atlocation, hospital)	(has-part, adult body)
(can reanimate, people)	(AtLocation, examination room)	(hasproperty, aggressive)	(treat, problem)

Table 3.9: Anecdotal examples (PO) for S *elephant* (top) and S *doctor* (bottom)

KB	Elementary NDMC	Middle NDMC	CommonsenseQA2	Trivia	Examveda	All
#Questions (Train/Test)	623/541	604/679	9741/1221	1228/452	1228/765	10974/3659
Random	25.5	23.7	21.0	25.9	25.4	22.0
word2vec	26.2	28.3	27.8	27.4	25.6	27.2
Quasimodo	**38.4**	**34.8**	26.1	**28.1**	**32.6**	**31.3**
ConceptNet	28.5	26.4	**29.9** (source)	24.4	27.3	27.5
TupleKB	34.8	25.5	25.3	22.2	27.4	27.5
WebChild	26.2	25.1	25.2	25.9	27.1	24.1

Table 3.10: Accuracy of answer selection in question answering

Cluster Coherence. We evaluate cluster coherence using an intruder task. For a random set of clusters that contain at least three P phrases, we show annotators sampled SO pairs from the cluster and samples of P phrases from the aligned cluster interspersed with an additional random intruder predicate drawn from the entire CSKB. For example, we show the SO pairs *spider-web, mole-tunnel, rabbit-hole*, along with the P phrases *build, sing, inhabit, live in*, where *sing* is the intruder to be found. We sampled 175 instances from two vertical slices, Animals and Persons, and used crowdsourcing (MTurk) to collect a total of 525 judgments on these 175 instances for the intruder detection task. We obtained an intruder detection accuracy of 64% for clusters in the Animals domain, and 54% in Persons domain (compared with 25% for a random baseline). This is supporting evidence that our co-clustering method yields reasonably coherent groups.

Anecdotal Examples. Table 3.9 provides a comparison of randomly chosen assertions for two subjects in each of the KBs: *(*elephant) (top) and *doctor* (bottom). WebChild assertions are quite vague, while TupleKB assertions are reasonable but not always salient. ConceptNet, constructed by human crowdsourcing, features high-quality assertions, but sometimes gives rather exotic properties. In contrast, the samples for Quasimodo are both typical and salient.

3.8.3 Extrinsic Evaluation

Answer Selection for QA. In this use case, we show that CSK helps in selecting answers for multiple-choice questions. We use five datasets: (i+ii) elementary school and middle school science questions from the AllenAI science challenge [91],

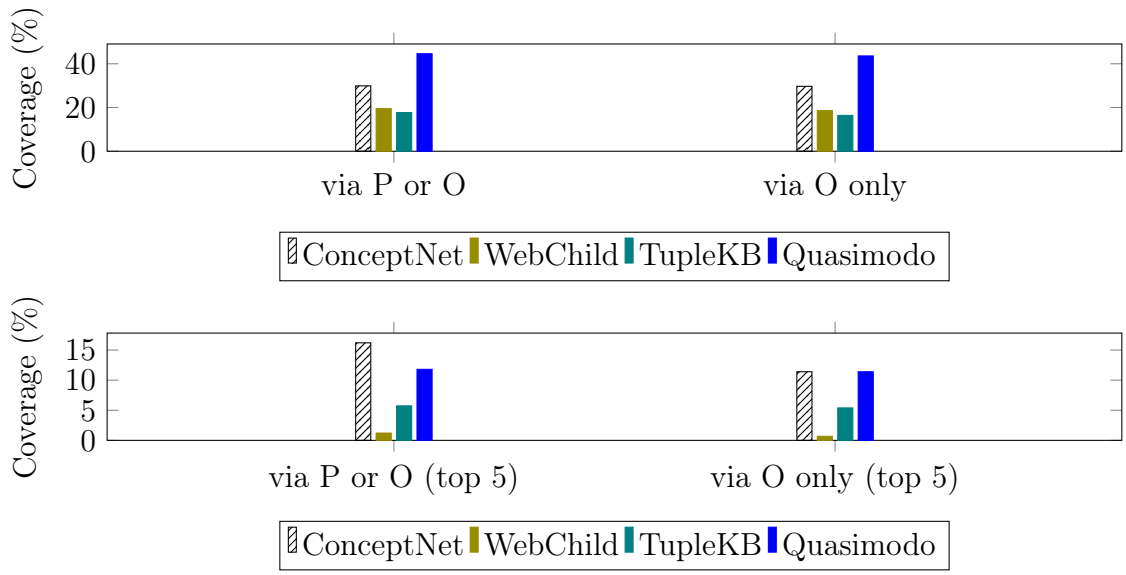

Figure 3.7: Coverage for word guessing game

(iii) commonsense questions generated from ConceptNet [126], (iv) reading comprehension questions from the TriviaQA dataset [51], and (v) exam questions from the Indian exam training platform Examveda [93]. The baseline are given by AllenAI, in aristo-mini (https://github.com/allenai/aristo-mini).

To assess the contribution of CSKBs, for each question-answer pair we build a pair of contexts $(ctx_{question}, ctx_{answer})$ as follows: for each group of words *grp* in the question (resp. the answer), we add all the triples of the form *(grp, p, o)* or *(s, p, grp)* in a KB. Then, we create features based on $(ctx_{question}, ctx_{answer})$ such as the number of SPO, SP, SO, PO, S, P, O overlaps, the size of the intersection of a context with the opposition original sentence, the number of useless words in the original sentence and the number of words in the original sentences. From these features, we train an AdaBoost classifier. This is a basic strategy for multiple-choice QA and could be improved in many ways. However, it is sufficient to bring out the value of CSK and the differences between the CSKBs.

We compare four CSKBs against each other and against a word2vec baseline which computes the embeddings similarity between questions and answers. The results are shown in Table 3.10. Quasimodo significantly outperforms the other CSKBs on four of the five datasets (ConceptNet performs better on CommonsenseQA dataset as it was used for the generation).

Word Guessing Game. Taboo is a popular word guessing game in which a player describes a concept without using five taboo words, usually the most reliable cues. The other player needs to guess the concept. We used a set of 578 taboo cards from the website playtaboo.com to evaluate the coverage of the different CSKBs.

Given a concept to be guessed, we compute the fraction of Taboo words that a KB associates with the concept, appearing in the O or P argument of the triples for the concept. This is a measure of a CSKB's potential ability to perform in this game (i.e., not playing the game itself). The resulting coverage is shown in

Table 3.7. Quasimodo outperforms all other KBs by this measure. TupleKB, the closest competitor on the science questions in the multiple-choice QA use case, has substantially lower coverage, indicating its limited knowledge beyond the (school-level) science domain. We notice that ConceptNet outperforms Quasimodo on this task when we consider only the top 5 statements per subject. This shows that even if ConceptNet contains fewer facts than Quasimodo, they are still salient.

3.9 Conclusion

This work presented Quasimodo, a methodology for acquiring high-quality commonsense assertions, by harnessing non-standard input sources, like query logs and QA forums, in a novel way.

As our experiments demonstrate, the Quasimodo knowledge base improves the prior state of the art, by achieving much better coverage of typical and salient commonsense properties (as determined by an MTurk study) while having similar quality in terms of precision. Extrinsic use cases further illustrate the advantages of Quasimodo.

Quasimodo data is available online (https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/commonsense/quasimodo/) and our code is available on Github (https://github.com/Aunsiels/CSK).

Chapter 4

Inside Quasimodo

> La grimace était son visage. Ou plutôt toute sa personne était une grimace.
>
> *Victor Hugo*

In Chapter 3, we constructed Quasimodo, a commonsense knowledge base. To better understand the data generated by Quasimodo and its internal components, we present in this chapter a companion Web portal. We also go one step further by showing applications of Quasimodo data.

This work was originally published as a demo paper at CIKM 2020 [110]:

> Romero, J. & Razniewski, S. Inside Quasimodo: Exploring the Construction and Usage of Commonsense Knowledge. In Proceedings of the 29th ACM International Conference on Information and Knowledge Management

4.1 Introduction

In the previous chapter, we introduced the commonsense knowledge base *Quasimodo*. The system leverages human curiosity expressed in query logs and question answering forums to extract commonsense knowledge, and significantly outperformed existing resources like ConceptNet in terms of coverage. In this chapter, we introduce a companion Web portal, which enables a comprehensive exploration of Quasimodo's content and its construction. In particular, our contributions are:

1. Development of a scalable architecture for knowledge base visualisation;
2. Visualisation of the extraction pipeline of Quasimodo;
3. Implementation of several applications on the Quasimodo data.

First, we review in Section 4.2 the state of the art of the methods used to display knowledge graphs for the main public knowledge bases. Then, we introduce the companion Web portal of Quasimodo in Section 4.3, and present the demonstration

experience in Section 4.4. Our Web portal is composed of several parts: an explorer, a visualisation for the extraction pipeline, a SPARQL endpoint and applications such as question answering or games like *Play Taboo!* and *Codenames*.

4.2 Previous Work

We present here some existing system for knowledge base visualisation. Most of them allow exploring the raw content and provide a SPARQL endpoint—however, very few focus on applications.

The most typical way to display a knowledge base is to print it as a table composed of three columns: Subject, predicate, object. ConceptNet [119], WebChild [127], TupleKB [35], Atomic [114], Comet [20] and Quasimodo [31] provide a CSV file in that style. Often, it is convenient to group the statements by subject on a separated page, and thus to omit the first column. We also regularly observe that systems tend to group the statements by predicates.

There exist more exotic ways to display a knowledge base. Yago [96] chooses to give a glimpse to the relations attached to a given subject through a star-shaped graph. Their companion Web portal displays an SVG of this graph. The graph structure is very natural when we deal with knowledge bases. Some third-party websites such as Geneawiki (https://tools.wmflabs.org/magnus-toolserver/ts2/geneawiki/) or the Wikidata Graph Builder (https://angryloki.github.io/wikidata-graph-builder/) use this representation to display relations between entities. However, due to the size of the graphs, it becomes tough for a human to find relevant information.

In our system, we choose to use a table representation where we added additional columns such as the polarity of a statement, an attached modality or a score.

Many systems also provide a simple search interface. More interestingly, some give access to a SPARQL endpoint to write complex queries. Finally, the websites offer an easy way to download data in different formats. We provide all these functionalities.

Third-party Web portals generally provide the applications associated with a knowledge base. For example, Inventaire (https://inventaire.io) uses Wikidata to extract information about books and The Art Browser (https://openartbrowser.org) uses Wikidata to display art information. Wikidata groups various projects related to their website (https://www.wikidata.org/wiki/Wikidata:Tools), and in particular about visualisation. In this chapter, we include several applications near the data visualisation, which can make the user more familiar with the knowledge base.

4.3 Quasimodo Web Portal Architecture

Our Web portal is accessible at https://quasimodo.r2.enst.fr. We use Nginx to manage connections, obtain HTTPS accesses and perform reverse proxy to the internal components.

To make our system scalable and reusable, we decomposed it into Docker containers. Docker containers are light and independent packages that contain everything

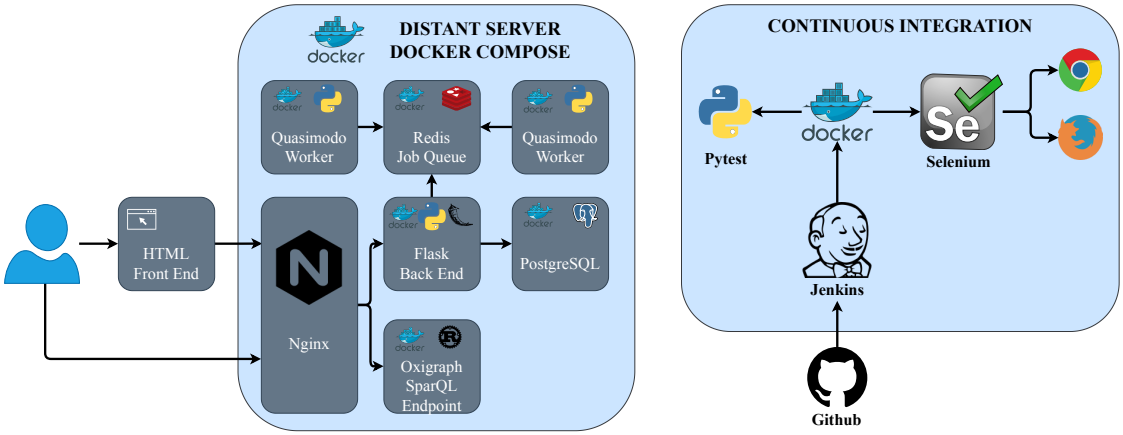

Figure 4.1: The Web portal architecture

to run an application. A developer can use them as building blocks for more complex applications deployed on a single or several computers. Our application runs on a single machine, and so a docker-compose file is used to link the entire system. Our system can be deployed on several computers as it is compatible with solutions such as Kubernetes that automatically scale each container according to its needs.

Let us enumerate the different containers present in the application. First, we wrote the core of the Web portal (we call it the back-end) in Python using Flask. Flask is a lightweight micro Web framework which comes with numerous additional packages. The back-end module orchestrates all the actions. In particular, it is linked to a container encapsulating a PostgreSQL database which stores Quasimodo. It is also linked to a container running a Redis database and which is used as a Job Queue (see Section 4.4.2). An arbitrary number of asynchronous workers on separated containers can connect to this job queue and execute tasks. Finally, we also have a particular container hosting a SPARQL endpoint: Oxigraph.

A user accesses our companion Web portal through a front end which uses Bootstrap4 and simple HTML, CSS and Javascript. The SPARQL endpoint is also accessible independently of the Web portal. We summarise the general architecture of the Web portal in Fig. 4.1.

We tested most of the components of our system using the Pytest library and Selenium. Selenium is a framework to emulate a browser such as Chrome or Firefox. The code is freely accessible on Github (https://github.com/Aunsiels/demo_quasimodo) where all the containers and the docker-compose file are also available. Finally, we used Jenkins to run a pipeline of tests ensuring the validity of each component. This pipeline gets executed every time the git repository receives a push. A Docker container encapsulates the pipeline which runs the tests.

The Web portal, the asynchronous workers and the SPARQL endpoint run on a single virtual machine which has access to 8 Virtual CPU of 2.6GHz and 16GB of RAM.

Subject	Predicate	Object
elephant	live in	africa
elephant	reject	baby
elephant	have	large ears
elephant	have	good memory
elephant	has_body_part	hair
elephant	be in	africa

Figure 4.2: Top Quasimodo statements for elephants

4.4 Demonstration Experience

4.4.1 Exploring and Searching Commonsense Knowledge

The data generated at the end of Chapter 3 is stored using a relational database (PostgreSQL) which provides a fast way to retrieve information. We use a single table to store all statements, with columns for subject, predicate, object, modality, polarity, example sentences and metrics (scores).

We provide a simple visualisation for the statements in Quasimodo as a table containing columns for the subject, predicate, object, modality, polarity (is it a positive or a negative statement) and scores. In Figure 4.2, we give a glimpse at this table. Different from KBs with fixed predicates, like Wikidata or ConceptNet, we organise the open predicate space by sorting statements by scores. Besides, we added the possibility to give positive or negative feedback about a statement, which could be used to refine supervised models. We also implemented a page per statement to display all information about it, and in particular the sentences that generated the statement and which sources they were derived from.

To traverse its content efficiently, a search function is available, which allows filtering the statements by subject, predicate, object and polarity. The search returns the number of matching statements and a table displaying them as explained above. In Figure 4.2, we show the top statements for the subject “elephant”.

4.4.2 Extraction Pipeline Visualisation

Quasimodo introduces an extraction pipeline to extract and process commonsense knowledge from various sources. In Chapter 3, we presented this extraction pipeline. Each module in this workflow is itself composed of several sub-modules for performing specialised tasks. The reader can find a list of these components in Chapter 3 and in the code provided with it (https://github.com/Aunsiels/CSK). This extraction pipeline, however, is very dense, and it can be difficult to understand the effect of each part. The present Web portal therefore gives further insights into the entire

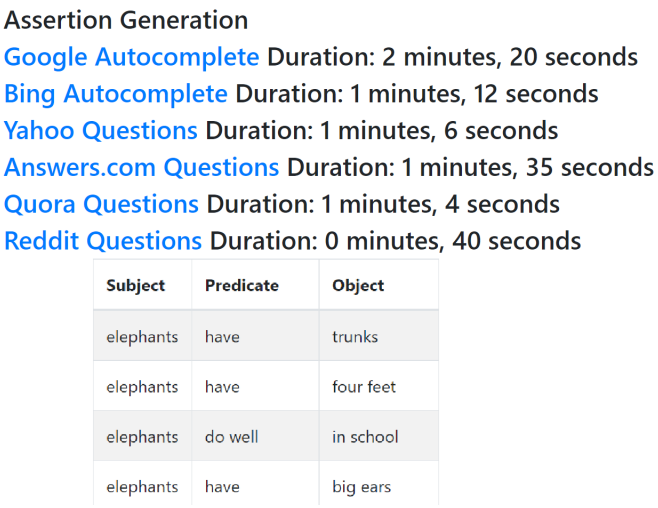

Figure 4.3: Top-level view of the extraction pipeline visualisation

process. In particular, we offer the possibility to run the extraction pipeline for a given subject. As most extraction sources must answer in a limited time, we launch the extraction pipeline in an asynchronous task using Redis Queues. Redis is an in-memory NoSQL database storing key-value couples. When we want to get the extraction pipeline information for a subject, we push a new job on a queue. Then, idle workers specialised in extraction pipeline execution come and read the pending task and start the extraction pipeline. Once they are done, they write back the details of the execution in the queue. They record these details every time a module or sub-module is executed, providing insights even if the extraction pipeline is still running.

The workers are based on the code given with Quasimodo and are encapsulated inside Docker containers. So, they can easily be duplicated, even on separated machines.

The back-end of the Web portal has access to the status of the jobs and the currently available information. We display the details of the extraction pipeline on a Web page showing the different stages of the extraction pipeline and the statements which are generated, modified or deleted. Besides, we also print the time spent in each module and sub-modules. The information of the extraction pipeline can be very dense, even for a single subject. So, the user has to choose in the interface a particular sub-module to display. We display the results as a table for the statements which were created, modified or deleted by the considered step. Figure 4.3 shows the beginning of the extraction pipeline of the subject *lawyer*.

We intentionally omitted the scoring phase as it must access all generated statements, for all subjects. Executing the extraction pipeline for a given subject takes approximately 30 minutes on our machine.

4.4.3 SPARQL Endpoint

We offer a SPARQL UI and endpoint https://quasimodo.r2.enst.fr/sparql. The UI is available on the Web portal, and the endpoint is callable by any program.

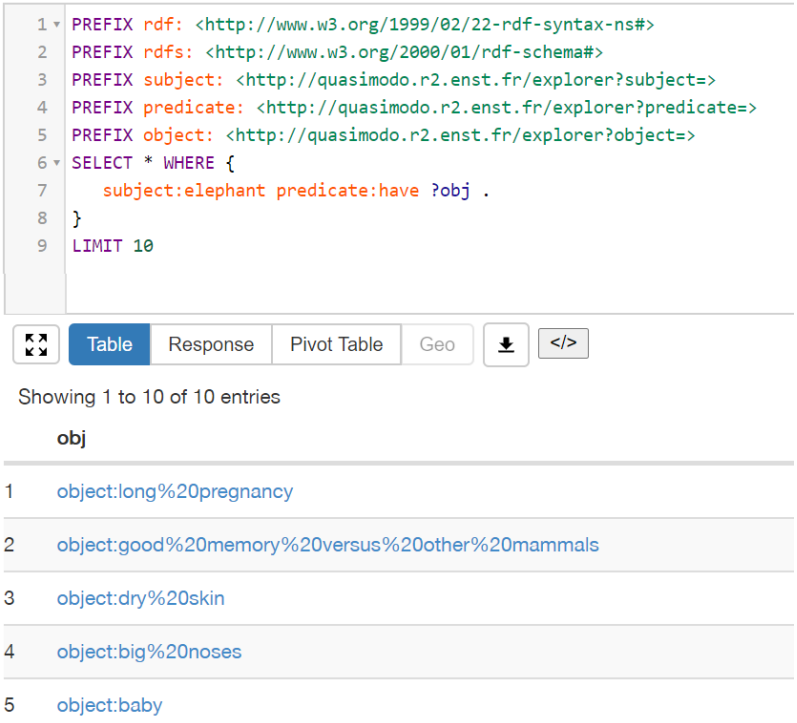

Figure 4.4: A sample SPARQL query

In Figure 4.4, we show a possible query. Nginx orientates the query to detect whether we are accessing the SPARQL interface or the main Web portal. Indeed, we put these two components on two separated containers. We dockerised Oxigraph (https://github.com/Tpt/oxigraph), a SPARQL endpoint written in Rust which is also used by Yago. It has the advantage to be very easy to use and very fast. We transformed our data into N-triples (https://www.w3.org/TR/n-triples/), at the cost of losing information such as the modality, the polarity and the scores. This might seem redundant with the data stored in PostgreSQL, but it helps optimise SPARQL queries. As no data is written while the application is running, this duplication is acceptable. Besides, PostgreSQL contains more information about the statements that are not compatible with SPARQL.

4.4.4 Play Taboo

Taboo is a game in which a player must make other players guess a word without using a list of forbidden words. In this demo, we provide an interface to play Taboo with Quasimodo. When a user starts a new game, the Web portal sends him a card. Then they must use a chat interface to give clue words to Quasimodo. Every time the user presses the *Make a Guess* button, the system tries to a find a relevant word. This process continues until Quasimodo finds the hidden word. Figure 4.5 shows a game example.

The algorithm used in the back end is simple. First, the database is filtered using the words given by the user. Then, we group the results by subjects, and we aggregate the scores using a sum or a max function, for example. We finally return

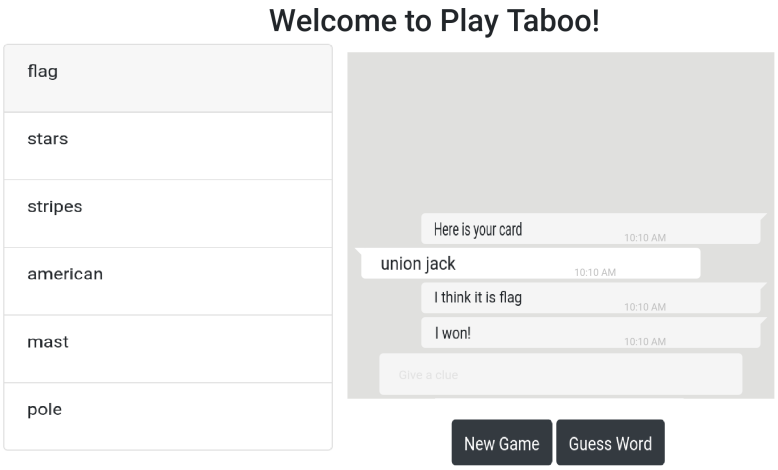

Figure 4.5: Play Taboo! interface

the best subject, under the condition that we never tried it before.

In addition to this game, we also provide the functionality to generate Taboo cards for any subject. We perform this generation by taking the most relevant objects associated with a subject by combining the scores.

4.4.5 Codenames

Codenames is a game designed by Vlaada Chvátil. It opposes two teams that must find their special agent before the other team. The agents are hidden behind codenames, which are simple words. Each team has a spymaster which must give a clue to the rest of the team (the operatives) to help us reveal the spies. For example, a spymaster can say *blue, 2* to help its companion guess the words *sky* and *sea*. The choice of the word must be made very carefully not to discover the agents of the other team.

In 2019, the Foundation of Digital Games conference hosted a competition: *The Codenames AI competition* (https://sites.google.com/view/the-codenames-ai-competition). However, we were not able to find the results of this competition. The presenters suggested that people should use word embeddings. This solution can be powerful when we make AI play against each other. However, the clues can become incomprehensible for humans.

Instead, we propose a solution based on Quasimodo to generate clues. We consider that the words to guess are subjects. Then, we take the object associated with the more subjects that has a score above a certain threshold and does not appear for the wrong subjects.

In the demonstration scenario, the user plays the role of the operative. He receives clues and must click on the potential agents. He plays against a bot which simply guesses one right word per turn. Vlaada Chvátil suggests this strategy for games with two players. We show in Figure 4.6 a game example.

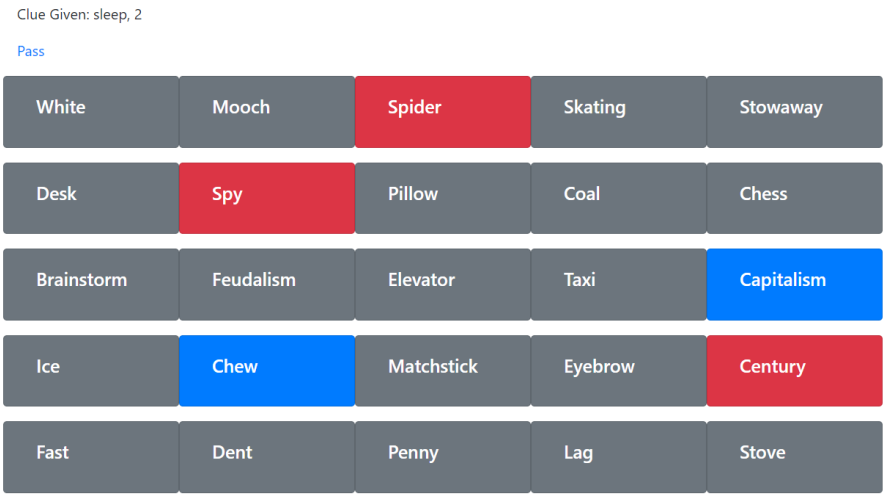

Figure 4.6: Codenames interface

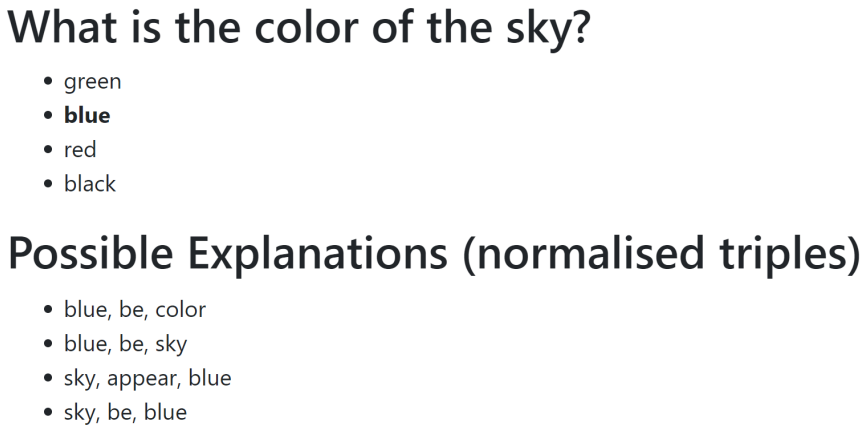

Figure 4.7: Question-answering interface

4.4.6 Multiple-Choice Question Answering

As in the original paper, we added the possibility to perform question answering using only Quasimodo, i.e. we do not have an underlying language model. We used the same algorithm: Given a question, an answer and a knowledge base, we generate a set of features based on the connections between the words in the knowledge graph. Then, using the same training data as in chapter 3, we train a linear classifier to predict a score for each answer. We reused the code provided with Quasimodo and added an interface to ask a question and give possible answers. Then, the system provides a score for each answer and displays them. In this demo, we also include the possibility to visualise the triples that might explain the answer. To generate these triples, we consider that we can use only one triple to answer the questing. Then we take the probability given by the logistic regression trained previously to rank the statements. We show in Figure 4.7 an example of a question, the answer given and the explanations.

4.5 Conclusion

With this demonstration, we give insights into the commonsense knowledge base Quasimodo. The user can access in a user-friendly way the raw data of Quasimodo. Besides, more details are given about the generation of the statements, making the entire process completely transparent. Finally, we showcased applications such as Taboo to prove the value of the database. Quasimodo is a scalable system to mine commonsense knowledge based on a general and adaptive extraction pipeline. We hope that this work will provide a better understanding of the system and the data so researchers can keep building on top of it, either on the application side or on the system side by proposing new extensions.

Chapter 5

Equivalent Query Rewritings

> The more constraints one imposes,
> the more one frees one's self.
>
> *Igor Stravinsky*

In Chapter 3 and Chapter 4, we focused on the ABox of a knowledge base. In particular, we saw how to create it in the case of commonsense knowledge, and we briefly showed some direct applications. In what follows, we turn our attention towards the TBox component of a knowledge base. We will study equivalent query rewriting using views with binding patterns.

A view with a binding pattern is a parameterised query on a database. Such views are used, e.g., to model Web services. To answer a query on such views, the views have to be orchestrated together in execution plans. We study a particular scenario where the views have the shape of a path, the queries are atomic, and we have integrity constraints in the form of unary inclusion dependencies. For this scenario, we show how queries can be rewritten into equivalent execution plans, which are guaranteed to deliver the same results as the query on all databases. More precisely, we provide a correct and complete algorithm to find these plans. Finally, we show that our method can be used to answer queries on real-world Web services.

This work was published in ESWC 2020:

> Romero, J., Preda, N., Amarilli, A., & Suchanek, F. (2020, May). Equivalent Rewritings on Path Views with Binding Patterns. In European Semantic Web Conference (pp. 446-462). Springer, Cham.

and comes with a demo paper, published at CIKM 2020 [107]:

> Romero, J., Preda, N., Amarilli, A., & Suchanek, F. Computing and Illustrating Query Rewritings on Path Views with Binding Patterns, In Proceedings of the 29th ACM International Conference on Information and Knowledge Management

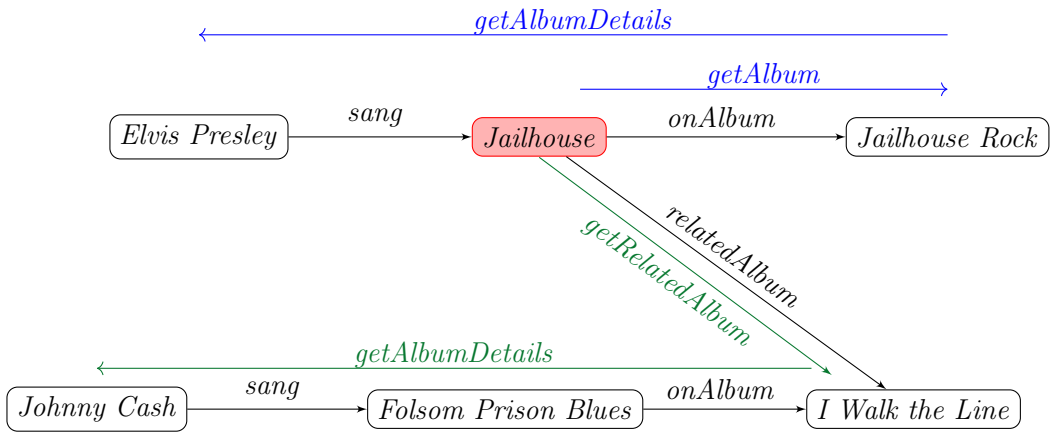

Figure 5.1: An equivalent execution plan (blue) and a maximal contained rewriting (green) executed on a database (black)

5.1 Introduction

In this chapter, we study views with binding patterns [102]. Intuitively, these can be seen as functions that, given input values, return output values from a database. For example, a function on a music database could take as input a musician, and return the songs by the musician stored in the database.

Several databases on the Web can be accessed only through such functions. They are usually presented as a form or as a Web service. For a REST Web service, a client calls a function by accessing a parameterised URL, and it responds by sending back the results in an XML or JSON file. The advantage of such an interface is that it offers a simple way of accessing the data without downloading it. Furthermore, the functions allow the data provider to choose which data to expose, and under which conditions. For example, the data provider can allow only queries about a given entity, or limit the number of calls per minute. According to programmableweb.com, there are over 20,000 Web services of this form – including LibraryThing, Amazon, TMDb, Musicbrainz, and Lastfm.

If we want to answer a user query on a database with such functions, we have to *compose* them. For example, consider a database about music – as shown in Figure 5.1 in black. Assume that the user wants to find the musician of the song *Jailhouse*. One way to answer this query is to call a function *getAlbum*, which returns the album of the song. Then we can call *getAlbumDetails*, which takes as input the album, and returns all songs on the album and their musicians. If we consider among these results only those with the song *Jailhouse*, we obtain the musician *Elvis Presley* (Figure 5.1, top, in blue). We will later see that, under certain conditions, this plan is guaranteed to return exactly all answers to the query on all databases: it is an *equivalent rewriting* of the query. This plan is in contrast to other possible plans, such as calling *getRelatedAlbum* and *getAlbumDetails* (Figure 5.1, bottom, in green). This plan does not return the exact set of query results. It is a *maximally contained rewriting*, another form of rewriting, which we will discuss in the related work.

Equivalent rewritings are of primordial interest to the user because they allow obtaining exactly the answers to the query – no matter what the database contains.

Equivalent rewritings are also of interest to the data provider: For example, in the interest of usability, the provider may want to make sure that equivalent plans can answer all queries of importance. However, finding equivalent rewritings is inherently non-trivial. As observed in [13, 15], the problem is undecidable in general. Indeed, plans can recursively call the same function. Thus, there is, a priori, no bound on the length of an execution plan. Hence, if there is no plan, an algorithm may try forever to find one – which indeed happens in practice.

In this chapter, we focus on path functions (i.e., functions that form a sequence of relations) and atomic queries. For this scenario, we can give a correct and complete algorithm that decides in PTIME whether a query has an equivalent rewriting or not. If it has one, we can give a grammar that enumerates all of them. Finally, we show that our method can be used to answer queries on real-world Web services. After reviewing related work in Section 5.2 and preliminaries in Section 5.3, we present our problem statement in Section 5.4, our algorithm in Section 5.5 and some theoretical background in Section 5.6, concluding with experiments in Section 5.7. Most of the proofs are given in the Appendix B.1.

5.2 Related Work

Formally, we aim at computing *equivalent rewritings* over views with binding patterns [102] in the presence of inclusion dependencies. Our approach relates to the following other works.

5.2.1 Views With Binding Patterns

We will formally introduce the preliminaries for our work in Section 5.3. To first discuss the related work, we briefly recall here the definitions of [102] of views with binding patterns. We assume that we have a set of predicates about which the queries are posed. For example, in Figure 5.1, we have the predicate *sang(x, y)*. A view (also called query template) is composed of a *head* and a *body*. We can divide the head into three components:

- A predicate denoting the view. For example, *getAlbum*
- Arguments for the predicate. For example, the arguments of *getAlbum* are the song and the album.
- A binding pattern indicating which arguments must be bound and which arguments can be free. For *getAlbum*, the song is bound, whereas the album is free.

Then, in our case, the body is a conjunction of predicates that produces a result to the query. Every variable in the head predicate must also appear in one of the body atoms.

In our notation, we underline the variables which must be bound and call them *input*. The other variables are called *output* variables. For example, in Figure 5.1, we have the view $getAlbumDetails(\underline{a}, s, m) \leftarrow onAlbum^{-}(a, s), sang^{-}(s, m)$ that,

given a binding for a (the album), returns a binding for a song and a singer. $getAlbumDetails(\underline{a}, s, m)$ is the head and $onAlbum^{-}(a, s), sang^{-}(s, m)$ is the body.

5.2.2 Equivalent Rewritings

Checking if a query is *determined* by views [70], or finding possible equivalent rewritings of a query over views, is a task that has been intensively studied for query optimisation [15, 65], under various classes of constraints. [45] shows the problem is undecidable in general for datalog queries. In our work, we are specifically interested in computing equivalent rewritings over views with binding patterns, i.e., restrictions on how the views can be accessed. This question has also been studied, in particular with the approach by Benedikt et al. [13] based on logical interpolation, for very general classes of constraints. In our setting, we focus on path views and unary inclusion dependencies on binary relations. This restricted (but practically relevant) language of functions and constraints had not been investigated. We show that, in this context, the problem is solvable in PTIME. What is more, we provide a self-contained, effective algorithm for computing plans, for which we provide an implementation. We compare experimentally against the PDQ implementation by Benedikt et al. [14] in Section 5.7.

5.2.3 Maximally Contained Rewritings

When datasources are incomplete, equivalent rewritings might fail to return query answers. To adress this issue, another line of work has studied how to rewrite queries against data sources in a way that is not equivalent but maximises the number of query answers [73]. Unlike equivalent rewritings, there is no guarantee that all answers are returned. However, due to the incompleteness of databases or the internal structure of a particular database, maximally contained rewritings might return more results than equivalent rewritings. For views with binding patterns, a first solution was proposed in [45, 46]. This work transforms views with binding patterns into a Datalog program with inverse rules.

Section 5.3.4 gives a formal definition of query containment. [116] shows that the problem of knowing whether a query q contains another query q' is undecidable with datalog queries. However, in the case where q' is not recursive, it becomes decidable. In [73] maximally-contained rewritings are defined as follows:

Definition 5.2.1 (Maximally Contained Rewriting)**.** *Let q be a query, $\mathcal{F}$ be a set of views and $\mathcal{L}$ a query language. The query π is a maximally contained rewriting of q using $\mathcal{F}$ w.r.t $\mathcal{L}$ if π is contained in q and there exists no rewriting π' such that $\pi \subseteq \pi' \subseteq q$ and π' is not equivalent to π.*

[45] shows that, in the case the rewriting language is Datalog, the queries are Datalog programs and views that are conjunctive queries, even if the equivalent rewriting problem is undecidable, it is still possible to find maximally-contained rewritings in polynomial time. In general, these rewritings must be recursive, so checking whether a maximally contained rewriting is an equivalent rewriting is undecidable.

The problem has also been studied for different query languages or under various constraints [25, 27, 39, 88]. We remark that by definition, the approach requires the generation of relevant but not-so-smart call compositions. These call compositions make sure that no answers are lost. Earlier work by some of the present authors proposed to prioritise promising function calls [97] or to complete the set of functions with new functions [98]. In our case, however, we are concerned with identifying only those function compositions that are guaranteed to deliver answers.

5.2.4 Web Service Orchestration

Web Service Orchestration or Composition [37, 78] has a broad definition: it represents the class of systems that add value above one or several Web services. The idea is to obtain additional information by combining smartly different sources. For example, let us imagine that we want to find all French singers who play the guitar, and that we have access to two Web services. The first one contains information about a musician but does not give access to nationalities. However, we have a second one which links people to their nationality. So, by combining the two Web services, we can first obtain all singers who are guitarists and then filter to get the French ones.

In our setting, however, we are concerned about with a single Web service and a single schema.

5.2.5 Federated Databases

Some works [100, 115] have studied *federated databases*, where each source can be queried with any query from a predefined language. By contrast, our sources only publish a set of preset parameterised queries, and the abstraction for a Web service is a view with a binding pattern, hence, a predefined query with input parameters. Therefore, our setting is different from theirs, as we cannot send arbitrary queries to the data sources: we can only call these predefined functions.

5.2.6 Web Services

There are different types of Web services, and many of them are not (or cannot be) modeled as views with binding patterns. AJAX Web services use JavaScript to allow a Web page to contact the server. Other Web services are used to execute complex business processes [38] according to protocols or choreographies, often described in BPEL [122]. The Web Services Description Language (WSDL) describes SOAP Web services. The Web Services Modeling Ontology (WSMO) [133], in the Web Ontology Language for Services (OWL-S) [83], or in Description Logics (DL) [103] can describe more complex services. These descriptions allow for Artificial Intelligence reasoning about Web services in terms of their behavior by explicitly declaring their preconditions and effects. Some works derive or enrich such descriptions automatically [22, 30, 99] in order to facilitate Web service discovery.

In our work, we only study Web services that are querying interfaces to databases. These can be modeled as views with binding patterns and are typically implemented

in the Representational State Transfer (REST) architecture, which does not provide a formal or semantic description of the functions.

5.3 Preliminaries

5.3.1 Global Schema

We assume a set $\mathcal{C}$ of constants and a set $\mathcal{R}$ of relation names.

We assume that all relations are binary, i.e., any n-ary relations have been encoded as binary relations by introducing additional constants (see https://www.w3.org/TR/swbp-n-aryRelations/).

A *fact* $r(a, b)$ is formed using a relation name $r \in \mathcal{R}$ and two constants $a, b \in \mathcal{C}$. A *database instance* I, or simply *instance*, is a set of facts. For $r \in \mathcal{R}$, we will use r^- as a relation name to mean the inverse of r, i.e., $r^-(b, a)$ stands for $r(a, b)$. More precisely, we see the inverse relations r^- for $r \in \mathcal{R}$ as being relation names in $\mathcal{R}$, and we assume that, for any instance I, the facts of I involving the relation name r^- are always precisely the facts $r^-(b, a)$ such that $r(a, b)$ is in I.

5.3.2 Inclusion Dependencies

A *unary inclusion dependency* for two relations r, s, which we write $r \rightsquigarrow s$, is the following constraint:

$$\forall x, y : r(x, y) \Rightarrow \exists z : s(x, z)$$

Note that one of the two relations or both may be inverses. In the following, we will assume a fixed set $\mathcal{UID}$ of unary inclusion dependencies, and we will only consider instances that satisfy these inclusion dependencies. We assume that $\mathcal{UID}$ is closed under implication, i.e., if $r \rightsquigarrow s$ and $s \rightsquigarrow t$ are two inclusion dependencies in $\mathcal{UID}$, then so is $r \rightsquigarrow t$.

5.3.3 Queries

An *atom* $r(\alpha, \beta)$ is formed with a relation name $r \in \mathcal{R}$ and α and β being either constants or variables. A *query* takes the form

$$q(\alpha_1, ..., \alpha_m) \leftarrow B_1, ..., B_n$$

where $\alpha_1, ...\alpha_m$ are variables, each of which must appear in at least one of the body atoms $B_1, ...B_n$. We assume that queries are *connected*, i.e., each body atom must be transitively linked to every other body atom by shared variables.

An *embedding* for a query q on a database instance I is a substitution σ for the variables of the body atoms so that $\forall B \in \{B_1, ..., B_n\} : \sigma(B) \in I$. A *result* of a query is an embedding projected to the variables of the head atom. We write $q(\alpha_1, ..., \alpha_m)(I)$ for the results of the query on I. An *atomic query* is a query that takes the form $q(x) \leftarrow r(a, x)$, where a is a constant and x is a variable.

5.3.4 Query Containment

[1] defines the notion of containment as follows:

Definition 5.3.1 (Query Containment)**.** *For two queries q_1, q_2 over the same schema $\mathcal{S}$, q_1 is contained in q_2, denoted $q_1 \subseteq q_2$, if for each database $\mathcal{I}$ satisfying $\mathcal{S}$, $q_1(\mathcal{I}) \subseteq q_2(\mathcal{I})$. Moreover, q_1 is equivalent to q_2, denoted $q_1 \equiv q_2$, iff $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$.*

5.3.5 Functions

We model functions as views with binding patterns [102], namely:

$$f(\underline{x}, y_1, ..., y_m) \leftarrow B_1, ..., B_n$$

Here, f is the function name, x is the *input variable* (which we underline), $y_1, ..., y_m$ are the *output variables*, and any other variables of the body atoms are *existential variables*. In this work, we are concerned with *path functions*, where the body atoms are ordered in a sequence $r_1(\underline{x}, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, x_n)$, the first variable of the first atom is the input of the plan, the second variable of each atom is the first variable of its successor, and the output variables are ordered in the same way as the atoms.

Example 5.3.2. *Consider again our example in Figure 5.1. There are 3 relations names in the database: onAlbum, sang, and relAlbum. The relation relAlbum links a song to a related album. The functions are:*

$$\begin{aligned} getAlbum(\underline{s}, a) &\leftarrow onAlbum(\underline{s}, a) \\ getAlbumDetails(\underline{a}, s, m) &\leftarrow onAlbum^{-}(\underline{a}, s), sang^{-}(s, m) \\ getRelatedAlbum(\underline{s}, a) &\leftarrow relAlbum(\underline{s}, a) \end{aligned}$$

The first function takes as input a song s, and returns as output the album a of the song. The second function takes as input an album a and returns the songs s with their musicians m. The last function returns the related albums of a song.

5.3.6 Execution Plans

Our goal in this work is to study when we can evaluate an atomic query on an instance using a set of path functions, which we will do using *plans*. Formally, a *plan* is a finite sequence $\pi_a(x) = c_1, \ldots, c_n$ of *function calls*, where a is a constant, x is the output variable. Each function call c_i is of the form $f(\underline{\alpha}, \beta_1, \ldots, \beta_n)$, where f is a function name, where the input α is either a constant or a variable occurring in some call in $c_1, \ldots, c_{i-1}$, and where the outputs $\beta_1, \ldots, \beta_n$ are either variables or constants. A *filter* in a plan is the use of a constant in one of the outputs β_i of a function call; if the plan has none, then we call it *unfiltered*. The *semantics* of the plan is the query:

$$q(x) \leftarrow \phi(c_1), \ldots, \phi(c_n)$$

where each $\phi(c_i)$ is the body of the query defining the function f of the call c_i in which we have substituted the constants and variables used in c_i, where we have

used fresh existential variables across the different $\phi(c_i)$, and where x is the output variable of the plan.

To *evaluate* a plan on an instance means running the query above. Given an execution plan π_a and a database I, we call $\pi_a(I)$ the answers of the plan on I. In practice, evaluating the plan means calling the functions in the order given by the plan. If a call fails, it can potentially remove one or all answers of the plan. More precisely, for a given instance I, the results $b \in \pi_a(I)$ are precisely the elements b to which we can bind the output variable when matching the semantics of the plan on I. For example, let us consider a function $f(\underline{x}, y) = r(x, y)$ and a plan $\pi_a(x) = f(a, x), f(b, y)$. This plan returns the answer a' on the instance $I = \{r(a, a'), r(b, b')\}$, and returns no answer on $I' = \{r(a, a')\}$.

Example 5.3.3. *The following is an execution plan for Example 5.3.2:*

$$\pi_{Jailhouse}(m) = getAlbum(\underline{Jailhouse}, a), getAlbumDetails(\underline{a}, Jailhouse, m)$$

The first element is a function call to getAlbum with the constant Jailhouse as input, and the variable a as output. The variable a then serves as input in the second function call to getAlbumDetails. The plan is shown in Figure 5.1 on page 58 with an example instance. This plan defines the query:

$$onAlbum(Jailhouse, a), onAlbum^{-}(a, Jailhouse), sang^{-}(Jailhouse, m)$$

For our example instance, we have the embedding:

$$\sigma = \{a = JailhouseRock, m = ElvisPresley\}.$$

5.3.7 Atomic Query Rewriting

Our goal is to determine when a given atomic query $q(x)$ can be evaluated as a plan $\pi_a(x)$. Formally, we say that $\pi_a(x)$ is a *rewriting* (or an *equivalent plan*) of the query $q(x)$ if, for any database instance I satisfying the inclusion dependencies $\mathcal{UID}$, the result of the plan π_a is equal to the result of the query q on I.

5.4 Problem Statement and Main Results

The goal of this work is to determine when a query admits a rewriting under the inclusion dependencies. If so, we compute a rewriting. In this section, we present our main high-level results for this task. We then describe in the next section (Section 5.5) the algorithm that we use to achieve these results, and show in Section 5.7 our experimental results on an implementation of this algorithm.

Remember that we study *atomic* queries, e.g., $q(x) \leftarrow r(a, x)$, that we study plans on a set $\mathcal{F}$ of path functions, and that we assume that the data satisfy integrity constraints given as a set $\mathcal{UID}$ of *unary inclusion dependencies*. In this section, we first introduce the notion of *non-redundant plans*, which are a specific class of plans that we study throughout the chapter; and we then state our results about finding rewritings that are non-redundant plans.

5.4.1 Non-Redundant Plans

Our goal in this section is to restrict to a well-behaved subset of plans that are *non-redundant*. Intuitively, a *redundant plan* is a plan that contains function calls that are not useful to get the output of the plan. For example, if we add the function call $getAlbum(m, a')$ to the plan in Example 5.3.3, then this is a redundant call that does not change the result of $\pi_{Jailhouse}$. We also call *redundant* the calls that are used to remove some of the answers, e.g., for the function $f(\underline{x}, y) = r(x, y)$ and the plan $\pi_a(x) = f(a, x), f(b, y)$ presented before, the second call is redundant because it does not contribute to the output (but can filter out some results). Formally:

Definition 5.4.1 (Redundant plan)**.** *An execution plan $\pi_a(x)$ is* redundant *if*

1. *it has no call using the constant a as input; or*
2. *it contains a call where none of the outputs is an output of the plan or an input to another call and the input of this call is not the output of the plan; or*
3. *there exists strictly more than one call that takes the output variable as input.*

If the plan does not satisfy these conditions, it is non-redundant*.*

Non-redundant plans can easily be reformulated to have a more convenient shape: the first call uses the input value as its input, and each subsequent call uses as its input a variable that was an output of the previous call. Formally:

Property 5.4.2. *The function calls of any non-redundant plan $\pi_a(x)$ can be organised in a sequence $c_0, c_1, \ldots, c_k$ such that the input of c_0 is the constant a, every other call c_i takes as input an output variable of the previous call c_{i-1}, and the output of the plan is in the call c_k.*

Proof. By definition of a non-redundant plan, there is an atom using the constant a as input. Let us call this atom c_0. Let us then define the sequence $c_0, c_1, \ldots, c_i$, and let us assume that at some stage we are stuck, i.e., we have chosen a call c_i such that none of the output variables of c_i are used as input to another call. If the output of the plan is not in c_i, then c_i witnesses that the plan is redundant. Otherwise, the output of the plan is in c_i. If we did not have $i = k$, then any of the calls not in $c_0, c_1, \ldots, c_i$ witness that the plan is redundant. So we have $i = k$, and we have defined the sequence $c_0, c_1, \ldots, c_k$ as required. □

Non-redundant plans seem less potent than redundant plans, because they cannot, e.g., filter the outputs of a call based on whether some other call is successful. However, as it turns out, we can restrict our study to non-redundant plans without loss of generality, which we do in the remainder of the chapter.

Property 5.4.3. *For any redundant plan $\pi_a(x)$ that is a rewriting to an atomic query $q(x) \leftarrow r(a, x)$, a subset of its calls forms a non-redundant plan, which is also equivalent to $q(x)$.*

To show this property, we first need to introduce the notion of well-filtering plans.

Well-Filtering Plans

In what follows, we introduce *well-filtering plans*, which are used both to show that we can always restrict to non-redundant plans (Property 5.4.3, showed in the next sub-section) and for the correctness proof of our algorithm. We then show a result (Lemma 5.4.6) showing that we can always restrict our study to well-filtering plans.

Let us first recall the notion of the *chase* [1]. The chase of an instance I by a set $\mathcal{UID}$ of unary inclusion dependencies (UIDs) is a (generally infinite) instance obtained by iteratively solving the violations of $\mathcal{UID}$ on I by adding new facts. In particular, if I already satisfies $\mathcal{UID}$, then the chase of I by $\mathcal{UID}$ is equal to I itself. The chase is known to be a *canonical database* in the sense that it satisfies precisely the queries that are true on all completions of I to make it satisfy $\mathcal{UID}$. We omit the formal definition of the chase and refer the reader to [1] for details about this construction. We note the following property, which can be achieved whenever $\mathcal{UID}$ is closed under UID implication, and when we do the so-called *restricted* chase which only solves the UID violations that are not already solved:

Property 5.4.4. *Let f be a single fact, and let I be the instance obtained by applying the chase on f. Then for each element c of I_0, for each relation $r \in \mathcal{R}$, there is at most one fact of I_0 where c appears in the first position of a fact for relation r.*

Remember now that that plans can use *filters*, which allow us to only consider the results of a function call where some variable is assigned to a specific constant. In this section, we show that, for any plan π_a, the only filters required are on the constant a. Further, we show that they can be applied to a single well-chosen atom.

Definition 5.4.5 (Well-Filtering Plan)**.** *Let $q(x) \leftarrow r(a, x)$ be an atomic query. An execution plan $\pi_a(x)$ is said to be* well-filtering for $q(x)$ *if all filters of the plan are on the constant a used as input to the first call and the semantics of π_a contains at least an atom $r(a, x)$ or $r^-(x, a)$, where x is the output variable.*

We can then show :

Lemma 5.4.6. *Given an atomic query $q(a, x) \leftarrow r(a, x)$ and a set of inclusion dependencies $\mathcal{UID}$, any equivalent rewriting of q must be well-filtering.*

Proof. We first prove the second part. We proceed by contradiction. Assume that there is a non-redundant plan $\pi_a(x)$ which is an equivalent rewriting of $q(a, x)$ and which contains a constant $b \neq a$. By Property 5.4.2, the constant b is not used as the input to a call (this is only the case of a, in atom c_0), so b must be used as an output filter in π_a.

Now, consider the database $I = \{r(a, a')\}$, and let I^* be the result of applying the chase by $\mathcal{UID}$ to I. The result of the query q on I^* is a', and I^* satisfies $\mathcal{UID}$ by definition, however b does not appear in I^* so π_a does not return anything on I^* (its semantics cannot have a match), a contradiction.

We now prove the first part of the lemma. We use the form of Property 5.4.2. If we separate the body atoms where a is an argument from those where both arguments are variables, we can write: $q'(a, x) \leftarrow A(a, x_1, x_2, \dots x_n), B(x_1, x_2, \dots x_n)$ where $A(a, x_1, x_2, \dots x_n) \leftarrow r_1(a, x_1), \dots r_n(a, x_n)$ (if we have an atom $r_i(x, a)$ we

transform it into $r_i^-(a,x)$) and a does not appear as argument in any of the body atoms of $B(x_1, x_2, \dots x_n)$. By contradiction, assume that we have $r_i \neq r$ for all $1 \leq i \leq n$.

Let I_0 be the database containing the single fact $r(a,b)$ and consider the database I_0^* obtained by chasing the fact $r(a,b)$ by the inclusion dependencies in $\mathcal{UID}$, creating a new value to instantiate every missing fact. Let $I_1^* = I_0^* \cup \{r(a_1,b_1)\} \cup \{r_i(a_1,c_i) \mid r_i(a,c_i) \in I_0^* \wedge r_i \neq r\} \cup \{r_i(b_1,c_i) \mid r_i(b,c_i) \in I_0^* \wedge r_i \neq r^-\}$. By construction, I_1^* satisfies $\mathcal{UID}$. Also, we have that $\forall r_i \neq r$, $r_i(a,c_i) \in I_1^* \Leftrightarrow r_i(a_1,c_i) \in I_1^*$. Hence, we have that $A(a,x_1,x_2,\dots x_n)(I_1^*) = A(a_1,x_1,x_2,\dots x_n)(I_1^*)$. Then, given that $B(x_1,x_2,\dots x_n)$ does not contain a nor a_1, we have that $q'(a_1,x)(I_1^*) = q'(a,x)(I_1^*)$. From the hypothesis we also have that $q'(a,x)(I_1^*) = q(a,x)(I_1^*)$ and $q'(a_1,x)(I_1^*) = q(a_1,x)(I_1^*)$. This implies that $q(a_1,x)(I_1^*) = q(a,x)(I_1^*)$. Contradiction. □

Proof that we Can Restrict to Non-Redundant Plans (Property 5.4.3)

We can now prove the property claiming that it suffices to study non-redundant plans. Recall its statement:

Property 5.4.3. *For any redundant plan $\pi_a(x)$ that is a rewriting to an atomic query $q(x) \leftarrow r(a,x)$, a subset of its calls forms a non-redundant plan, which is also equivalent to $q(x)$.*

Proof. In what follows, we write $q(a,x)$ instead of $q(x)$ to clarify the inner constant. Let $\pi_a(x)$ be an equivalent plan. From Lemma 5.4.6, we have that its semantics contains a body atom $r(a,x)$ or $r^-(x,a)$. Hence, there is a call c such that $r(a,x)$ or $r^-(x,a)$ appear in its semantics. From the definition of plans, and similarly to the proof of Property 5.4.2, there is a chain of calls $c_1, c_2, \dots c_k$ such that c_1 takes a constant as input, $c_k = c$, and for every two consecutive calls c_i and c_{i+1}, with $i \in \{1, \dots k-1\}$, there is a variable α such that α is an output variable for c_i and an output variable for c_{i+1}. From Lemma 5.4.6, we have that for all the calls that take a constant as input, the constant is a. Hence, the input of c_1 is a. Let $\pi'_a(x)$ be the plan consisting of the calls $c_1, c_2, \dots c_k = c$. Note that c ensures that $r(a,x)$ or $r^-(x,a)$ appear in the semantics of $\pi'_a(x)$.

We first notice that by construction $\pi'_a(x)$ is non-redundant. Now, if we consider the semantics of a plan as a set of body atoms, the semantics of $\pi'_a(x)$ is contained in the semantics of $\pi_a(x)$. Hence, we have $\forall I, \pi_a(x)(I) \subseteq \pi'_a(x)(I)$. As $\pi_a(x)$ is equivalent to $q(x) \leftarrow r(a,x)$, $\forall I$, we have $\pi_a(x)(I) = q(x)(I)$. As $\pi'_a(x)$ contains $r(a,x)$, $\pi'_a(x)(I) \subseteq q(x)(I)$. So, $\forall I, q(x)(I) = \pi_a(x)(I) \subseteq \pi'_a(x)(I) \subseteq q(x)(I)$. Hence, all the inclusions are equalities, and indeed $\pi'_a(x)$ is also equivalent to the query under $\mathcal{UID}$. This concludes the proof. □

5.4.2 Result Statements

Our main theoretical contribution is the following theorem:

Theorem 5.4.7. *There is an algorithm which, given an atomic query $q(x) \leftarrow r(a,x)$, a set $\mathcal{F}$ of path function definitions, and a set $\mathcal{UID}$ of UIDs, decides in*

polynomial time if there exists an equivalent rewriting of q. If so, the algorithm enumerates all the non-redundant plans that are equivalent rewritings of q.

In other words, we can efficiently decide if equivalent rewritings exist, and when they do, the algorithm can compute them. Note that, in this case, the generation of an equivalent rewriting is *not* guaranteed to be in polynomial time, as the equivalent plans are not guaranteed to be of polynomial size. Also, observe that this result gives a *characterisation* of the equivalent non-redundant plans, in the sense that *all* such plans are of the form that our algorithm produces. Of course, as the set of equivalent non-redundant plans is generally infinite, our algorithm cannot actually write down all such plans, but it provides any such plan after a finite time. The underlying characterisation of equivalent non-redundant plans is performed via a context-free grammar describing possible paths of a specific form, which we will introduce in the next section.

Our methods can also solve a different problem: given the query, path view definitions, unary inclusion dependencies, and given a candidate non-redundant plan, decide if the plan is correct, i.e., if it is an equivalent rewriting of the query. The previous result does not provide a solution as it produces all non-redundant equivalent plans in some arbitrary order. However, we can show using similar methods that this task can also be decided in polynomial time:

Proposition 5.4.8. *Given a set of unary inclusion dependencies, a set of path functions, an atomic query $q(x) \leftarrow r(a, x)$ and a non-redundant execution plan π_a, one can determine in PTIME if π_a is an equivalent rewriting of q.*

That proposition concludes the statement of our main theoretical contributions. We describe in the next section the algorithm used to show our main theorem (Theorem 5.4.7) and used for our experiments in Section 5.7. Section 5.6 gives more technical details and the appendix contains the proofs for our theorems.

5.5 Algorithm

We now present the algorithm used to show Theorem 5.4.7. The presentation explains at a high level how the algorithm can be implemented, as we did for the experiments in Section 5.7. However, technical details are given in Section 5.6 and formal proofs are located in the appendix.

Our algorithm is based on a characterisation of the non-redundant equivalent rewritings as the intersection between a context-free grammar and a regular expression (the result of which is itself a context-free language). The context-free grammar encodes the UID constraints and generates a language of words that intuitively describe forward-backward paths that are guaranteed to exist under the UIDs. As for the regular expression, it encodes the path functions and expresses the legal execution plans. Then, the intersection gets all non-redundant execution plans that satisfy the UIDs. We first detail the construction of the grammar, and then of the regular expression.

5.5.1 Defining the Context-Free Grammar of Forward-Backward Paths

Our context-free grammar intuitively describes a language of forward-backward paths, which intuitively describe the sequences of relations that an equivalent plan can take to walk away from the input value on an instance, and then walk back to that value, as in our example on Figure 5.1, to finally use the relation that consists of the query answer: in our example, the plan is *getAlbum(Jailhouse,a), getAlbumDetails(a,Jailhouse,m)*. The grammar then describes all such back-and-forth paths from the input value that are guaranteed to exist thanks to the unary inclusion dependencies that we assumed in $\mathcal{UID}$. Intuitively, it describes such paths in the *chase* by $\mathcal{UID}$ of an answer fact. We now define this grammar, noting that the definition is independent of the functions in $\mathcal{F}$:

Definition 5.5.1 (Grammar of forward-backward paths)**.** *Given a set of relations $\mathcal{R}$, given an atomic query $q(a,x) \leftarrow r(a,x)$ with $r \in \mathcal{R}$, and given a set of unary inclusion dependencies $\mathcal{UID}$, the* grammar of forward-backward paths *is a context-free grammar $\mathcal{G}_q$, whose language is written $\mathcal{L}_q$, with the non-terminal symbols $S \cup \{L_{r_i}, B_{r_i} \mid r_i \in \mathcal{R}\}$, the terminals $\{r_i \mid r_i \in \mathcal{R}\}$, the start symbol S, and the following productions:*

$$S \to B_r r \tag{5.5.1}$$
$$S \to B_r r B_{r^-} r^- \tag{5.5.2}$$
$$\forall r_i, r_j \in \mathcal{R} \text{ s.t. } r_i \rightsquigarrow r_j \text{ in } \mathcal{UID} : B_{r_i} \to B_{r_i} L_{r_j} \tag{5.5.3}$$
$$\forall r_i \in \mathcal{R} : B_{r_i} \to \epsilon \tag{5.5.4}$$
$$\forall r_i \in \mathcal{R} : L_{r_i} \to r_i B_{r_i^-} r_i^- \tag{5.5.5}$$

The words of this grammar describe the sequence of relations of paths starting at the input value and ending by the query relation r, which are guaranteed to exist thanks to the unary inclusion dependencies $\mathcal{UID}$. In this grammar, the B_{r_i}s represent the paths that "loop" to the position where they started, at which we have an outgoing r_i-fact. These loops are either empty (Rule 5.5.4), are concatenations of loops which may involve facts implied by $\mathcal{UID}$ (Rule 5.5.3), or may involve the outgoing r_i fact and come back in the reverse direction using r_i^- after a loop at a position with an outgoing r_i^--fact (Rule 5.5.5).

5.5.2 Defining the Regular Expression of Possible Plans

While the grammar of forward-backward paths describes possible paths that are guaranteed to exist thanks to $\mathcal{UID}$, it does not reflect the set $\mathcal{F}$ of available functions. This is why we intersect it with a regular expression that we will construct from $\mathcal{F}$, to describe the possible sequences of calls that we can perform following the description of non-redundant plans given in Property 5.4.2.

The intuitive definition of the regular expression is simple: we can take any sequence of relations, which is the semantics of a function in $\mathcal{F}$, and concatenate such sequences to form the sequence of relations corresponding to what the plan

retrieves. However, there are several complications. First, for every call, the output variable that we use may not be the last one in the path, so performing the call intuitively corresponds to a prefix of its semantics: we work around this by adding some backward relations to focus on the right prefix when the output variable is not the last one. Second, the last call must end with the relation r used in the query, and the variable that precedes the output variable of the whole plan must not be existential (otherwise, we will not be able to filter on the correct results). Third, some plans consisting of one single call must be handled separately. Last, the definition includes other technicalities that relate to our choice of so-called *minimal filtering plans* in the correctness proofs that we give in the appendix. Here is the formal definition:

Definition 5.5.2 (Regular expression of possible plans)**.** *Given a set of functions $\mathcal{F}$ and an atomic query $q(x) \leftarrow r(a, x)$, for each function $f : r_1(x_0, x_1), ...r_n(x_{n-1}, x_n)$ of $\mathcal{F}$ and input or output variable x_i, define:*

$$w_{f,i} = \begin{cases} r_1 \dots r_i & \text{if } i = n \\ r_1 \dots r_n r_n^- ... r_{i+1}^- & \text{if } 0 \leq i < n \end{cases}$$

For $f \in \mathcal{F}$ and $0 \leq i < n$, we say that a $w_{f,i}$ is final *when:*

- *the last letter of $w_{f,i}$ is r^-, or it is r and we have $i > 0$;*
- *writing the body of f as above, the variable x_{i+1} is an output variable;*
- *for $i < n - 1$, if x_{i+2} is an output variable, we require that f does not contain the atoms: $r(x_i, x_{i+1}).r^-(x_{i+1}, x_{i+2})$.*

The regular expression of possible plans is then $P_r = W_0|(W^*W')$, where:

- *W is the disjunction over all the $w_{f,i}$ above with $0 < i \leq n$.*
- *W' is the disjunction over the final $w_{f,i}$ above with $0 < i < n$.*
- *W_0 is the disjunction over the final $w_{f,i}$ above with $i = 0$.*

The intuition of this definition is that we want to get as output x_{i+1} and to filter on x_i when the last atom is r. Otherwise, we output x_{i+1} and filter on x_i. Besides, in Section 5.6.1, we will consider execution plans such that the position of the filter is unambiguous. When a plan ends with a function call containing the atoms $r(x_i, x_{i+1}).r^-(x_{i+1}, x_{i+2})$ with x_{i+1} as output of the plan, we prefer to filter on the latest possible variable, x_{i+2}.

5.5.3 Defining the Algorithm

We can now present our algorithm to decide the existence of equivalent rewritings and enumerate all non-redundant equivalent execution plans when they exist, which is what we use to show Theorem 5.4.7:
Input: a set of path functions $\mathcal{F}$, a set of relations $\mathcal{R}$, a set of $\mathcal{UID}$ of UIDs, and an atomic query $q(x) \leftarrow r(a, x)$.
Output: a (possibly infinite) list of rewritings.

1. Construct the grammar $\mathcal{G}_q$ of forward-backward paths (Definition 5.5.1).

2. Construct the regular expression P_r of possible plans (Definition 5.5.2).

3. Intersect P_r and $\mathcal{G}_q$ to create a grammar $\mathcal{G}$

4. Determine if the language of $\mathcal{G}$ is empty:

   If no, then no equivalent rewritings exist and stop;

   If yes, then continue

5. For each word w in the language of $\mathcal{G}$:

   - For each execution plan $\pi_a(x)$ that can be built from w (intuitively decomposing w using P_r, see below for details):
     - For each subset S of output variables of $\pi_a(x)$: If adding a filter to a on the outputs in S gives an equivalent plan, then output the plan (see below for how to decide this)

We now make more precise the last steps of our algorithm:

- Building all possible execution plans $\pi_a(x)$ from a word w of $\mathcal{G}$: this is specifically done by taking all preimages of w by the path transformation, which is done as shown in Property 5.6.9. Note that these are all minimal filtering plans by definition.

- Checking subsets of variables on which to add filters: for each minimal filtering plan, we remove its filter, and then consider all possible subsets of output variables where a filter could be added, so as to obtain a well-filtering plan which is equivalent to the minimal filtering plan that we started with. (As we started with a minimal filtering plan, we know that at least some subset of output variables will give a well-filtering plan, namely, the subset of size 0 of 1 that had filters in the original minimal filtering plan.) The correctness of this step is because we know by Lemma 5.4.6 that non-redundant equivalent plans must be well-filtering, and because we can determine using Theorem 5.6.2 if adding filters to a set of output variables yields a plan which is still an equivalent rewriting.

Our algorithm thus decides the existence of an equivalent rewriting by computing the intersection of a context-free language and a regular language and checking if its language is empty. As this problem can be solved in PTIME, the complexity of our entire algorithm is polynomial in the size of its input. The correctness proof of our algorithm (which establishes Theorem 5.4.7), and the variant required to show Proposition 5.4.8, are given in the appendix.

5.5.4 Example

Let us describe our algorithm for the Example 5.3.2. Our relations are $onAlbum$, $sang$, $relAlbum$ and all the opposite relations. We consider that we have one non-trivial UID: $\mathcal{UID} = \{sang^- \rightsquigarrow onAlbum, sang^- \rightsquigarrow sang^-, sang \rightsquigarrow sang, onAlbum^- \rightsquigarrow onAlbum^-, onAlbum \rightsquigarrow onAlbum, relAlbum \rightsquigarrow relAlbum, relAlbum \rightsquigarrow relAlbum\}$. The query we want to answer is $q(x) \leftarrow sang^-(Jailhouse, x)$.

First, we need to construct the grammar $\mathcal{G}_q$. The terminals are $sang$, $sang^-$, $onAlbum$, $onAlbum^-$, $relAlbum$ and $relAlbum^-$. It contains the following rules:

- $S \rightarrow B_{sang^-} sang^-$
- $S \rightarrow B_{sang^-} sang^- B_{sang} sang$
- $B_{sang^-} \rightarrow B_{sang^-} L_{onAlbum}$ from $sang^- \rightsquigarrow onAlbum$
- $B_{sang^-} \rightarrow B_{sang^-} L_{sang^-}$ from $sang^- \rightsquigarrow sang^-$
- $B_{sang} \rightarrow B_{sang} L_{sang}$ from $sang \rightsquigarrow sang$
- $B_{onAlbum^-} \rightarrow B_{onAlbum^-} L_{onAlbum^-}$ from $onAlbum^- \rightsquigarrow onAlbum^-$
- $B_{onAlbum} \rightarrow B_{onAlbum} L_{onAlbum}$ from $onAlbum \rightsquigarrow onAlbum$
- $B_{relAlbum^-} \rightarrow B_{relAlbum^-} L_{relAlbum^-}$ from $relAlbum^- \rightsquigarrow relAlbum^-$
- $B_{relAlbum} \rightarrow B_{relAlbum} L_{relAlbum}$ from $relAlbum \rightsquigarrow relAlbum$
- $B_{sang^-} \rightarrow \epsilon$, $B_{sang} \rightarrow \epsilon$, $B_{onAlbum^-} \rightarrow \epsilon$, $B_{onAlbum} \rightarrow \epsilon$, $B_{relAlbum^-} \rightarrow \epsilon$, and $B_{relAlbum} \rightarrow \epsilon$
- $L_{sang^-} \rightarrow sang^- B_{sang} sang$, $L_{sang} \rightarrow sang B_{sang^-} sang^-$, $L_{onAlbum^-} \rightarrow onAlbum^- B_{onAlbum} onAlbum$, $L_{onAlbum} \rightarrow onAlbum B_{onAlbum^-} onAlbum^-$, $L_{relAlbum^-} \rightarrow relAlbum^- B_{relAlbum} relAlbum$, and $L_{relAlbum} \rightarrow relAlbum\ B_{relAlbum^-}\ relAlbum^-$

Next, we construct the regular expression P_r of possible plans. To do so, we need to compute the $w_{f,i}$:

- $w_{getAlbum,0} = onAlbum.onAlbum^-$
- $w_{getAlbum,1} = onAlbum$
- $w_{getAlbumDetails,0} = onAlbum^-.sang^-.sang.onAlbum$
- $w_{getAlbumDetails,1} = onAlbum^-.sang^-.sang$
- $w_{getAlbumDetails,2} = onAlbum^-.sang^-$, which is final.
- $w_{getRelatedAlbum,0} = relAlbum.relAlbum^-$
- $w_{getRelatedAlbum,1} = relAlbum$

Then the regular expression is $(w_{getAlbum,1} \mid w_{getAlbumDetails,1} \mid w_{getAlbumDetails,2} \mid w_{getRelatedAlbum,1})^*.w_{getAlbumDetails,2}$.

We are not going to perform the intersection here as it quickly creates a lot of rules. However, we notice that this intersection is not empty as both $\mathcal{G}_q$ and P_r contain the word: $onAlbum.onAlbum^-.sang^-$. This word matches the execution plan $\pi_{Jailhouse}(x) = getAlbum(Jailhouse, y)getAlbumDetails(y, Jailhouse, x)$.

5.6 Capturing Languages

In this section, we give more formal details on our approach, towards a proof of Theorem 5.4.7 and Proposition 5.4.8. We will show that we can restrict ourselves to a class of execution plans called *minimal filtering plans* which limit the possible filters in an execution plan. Finally, we will define the notion of *capturing language* and show that the language $\mathcal{L}_q$ defined in Section 5.5 is capturing (Theorem 5.6.11); and define the notion of a language *faithfully representing plans* and show that the language of the regular expression P_r faithfully represents plans (Theorem 5.6.13). This section gives a high-level overview and states the theorem; the Appendix B.1 contains proofs for the present section; and the Appendix B.2 contains the proofs of the claims made in Sections 5.4 and 5.5.

5.6.1 Minimal Filtering Plans

Remember the definition of well-filtering plans (Definition 5.4.5). We now simplify even more the filters that should be applied to an equivalent plan, to limit ourselves to a single filter, by introducing *minimal filtering plans*.

Definition 5.6.1 (Minimal Filtering Plan)**.** *Given a well-filtering plan $\pi_a(x)$ for an atomic query $q(a, x) \leftarrow r(a, x)$, let the minimal filtering plan associated to $\pi_a(x)$ be the plan $\pi'_a(x)$ that results from removing all filters from $\pi_a(x)$ and doing the following:*

- *We take the greatest possible call c_i of the plan, and the greatest possible output variable x_j of call c_i, such that adding a filter on a to variable x_j of call c_i yields a well-filtering plan, and define $\pi'_a(x)$ in this way.*
- *If this fails, i.e., there is no possible choice of c_i and x_j, then we leave $\pi_a(x)$ as-is, i.e., $\pi'_a(x) = \pi_a(x)$.*

Note that, in this definition, we assume that the atoms in the semantics of each function follow the order in the definition of the path function. Also, note that the minimal filtering plan $\pi'_a(x)$ associated to a well-filtering plan is always itself well-filtering. This fact is evident if the first case in the definition applies, and in the second case, given that $\pi_a(x)$ was itself well-filtering, the only possible situation is when the first atom of the first call of $\pi_a(x)$ was an atom of the form $r(a, x)$, with a being the input element: otherwise $\pi_a(x)$ would not have been well-filtering. So, in this case, $\pi'_a(x)$ is well-filtering. Besides, note that, when the well-filtering plan π_a is non-redundant, then this is also the case of the minimal filtering plan π_a^{min}

because the one filter that we may add is necessarily at an output position of the last call.

Finally, note that a well-filtering plan is not always equivalent to the minimal filtering plan, as removing the additional filters can add some results. However, one can easily check if it is the case or not. This theorem is proven in Appendix B.1.1.

Theorem 5.6.2. *Given a query $q(x) \leftarrow r(a, x)$, a well-filtering plan π_a, the associated minimal filtering plan π_a^{min} and unary inclusion dependencies $\mathcal{UID}$:*

- *If π_a^{min} is not equivalent to q, then neither is π_a.*
- *If π_a^{min} is equivalent to q, then we can determine in polynomial time if π_a is equivalent to π_a^{min}*

This theorem implies that, when the query has a rewriting as a well-filtering plan, then the corresponding minimal filtering plan is also a rewriting:

Corollary 5.6.3. *Given unary inclusion dependencies, if a well-filtering plan is a rewriting for an atomic query q, then it is equivalent to the associated minimal filtering plan.*

Proof. This is the contrapositive of the first point of the theorem: if π_a is equivalent to q, then so in π_a^{min}, hence π_a and π_a^{min} are then equivalent. □

For that reason, to study equivalent rewritings, we will focus our attention on minimal filtering plans: Theorem 5.6.3 can identify other well-filtering plans that are rewritings, and we know by Lemma 5.4.6 that plans that are not well-filtering cannot be rewritings.

5.6.2 Path Transformations

We now show how to encode minimal filtering plans as words over an alphabet whose letters are the relation names in $\mathcal{R}$. The key is to rewrite the plan so that its semantics is a path query.

Here is the formal notion of a *path query*:

Definition 5.6.4. *A **path query** is a query of the form*

$$q_a(x_i) \leftarrow r_1(a, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, x_n)$$

*where a is a constant, x_i is the output variable, each x_j except x_i is either a variable or the constant a, and $1 \leq i \leq n$. The sequence of relations $r_1...r_n$ is called the **skeleton** of the query.*

We formalise as follows the transformation that transforms plans into path queries. We restrict it to non-redundant minimal filtering plans to limit the number of filters that we have to deal with:

Definition 5.6.5 (Path Transformation)**.** *Let $\pi_a(x)$ be a non-redundant minimal filtering execution plan and $\mathcal{R}$ a set of relations. We define the path transformation of $\pi_a(x)$, written $\mathcal{P}'(\pi_a)$, the transformation that maps the plan π_a to a path query $\mathcal{P}'(\pi_a)$ obtained by applying the following steps:*

1. *Consider the sequence of calls $c_0, c_1, ..., c_k$ as defined in Property 5.4.2, removing the one filter to element a if it exists.*

2. *For each function call $c_i(\underline{y_1}, y_{i_1}..., y_{i_j}, ...y_{i_n}) = r_1(y_1, y_2), ..., r_k(y_k, y_{k+1}), ..., r_m(y_m, y_{m+1})$ in π_a with $1 < i_1 < ... < i_n < m+1$, such that y_{i_j} is the output used as input by the next call or is the output of the plan, we call the* sub-semantics associated to c_i *the query: $r_1...r_m.r_m^-...r_{i_j}^-(y_1, ..., y_{i_j-1}, y'_{i_j}, ..., y'_m, y_{m+1}, ..., y_{i_j})$, where $y'_{i_j}, ..., y'_m$ are new variables. We do nothing if $i_j = m+1$.*

3. *Concatenate the sub-semantics associated to the calls in the order of the sequence of calls. We call this new query the* path semantics.

4. *There are two cases:*

   - *If the semantics of π_a contains the atom $r(a, x)$ (either thanks to a filter to the constant a on an output variable or thanks to the first atom of the first call with a being the input variable), then this atom must have been part of the semantics of the last call (in both cases). The sub-semantics of the last call is therefore of the form $\ldots, r(x_a, x'), r_2(x', x_2), \ldots, r_n(x_{n-1}, x_n), r_n^-(x_n, x_{n-1}), \ldots, r_2^-(x_2, x)$, in which x_a was the variable initially filtered to a (or was the input to the plan, in which case it is still the constant a) and we append the atom $r^-(x, a)$ with a filter on a, where x is the output of the path semantics.*
   - *Otherwise, the semantics of π_a contains an atom $r^-(x, a)$, then again it must be part of the last call whose sub-semantics looks like $\ldots, r^-(x', x'_2), r_2(x'_2, x'_3), ..., r_n(x'_{n-1}, x_n), r_n^-(x_n, x_{n-1}), ..., r(x_1, x)$, in which x'_2 was the variable initially filtered to a, and we replace the last variable x_1 by a, with x being the output of the path semantics.*

We add additional atoms in the last point to ensure that the filter on the last output variable is always on the last atom of the query. Notice that the second point relates to the words introduced in Definition 5.5.2.

The point of the above definition is that, once we have rewritten a plan to a path query, we can easily see the path query as a word in $\mathcal{R}^*$ by looking at the skeleton. Formally:

Definition 5.6.6 (Full Path Transformation)**.** *Given a non-redundant minimal filtering execution plan, writing $\mathcal{R}$ for the set of relations of the signature, we denote by $\mathcal{P}(\pi_a)$ the word over $\mathcal{R}$ obtained by keeping the skeleton the path query $\mathcal{P}'(\pi_a)$ and we call it the* full path transformation.

Note that this loses information about the filters, but this is not essential.

Example 5.6.7. *Let us consider the two following path functions:*

$$f_1(x, y) = s(x, y), t(y, z)$$
$$f_2(x, y, z) = s^-(x, y), r(y, z), u(z, z')$$

The considered atomic query is $q(x) \leftarrow r(a, x)$. *We are given the following non-redundant minimal filtering execution plan:*

$$\pi_a(x) = f_1(a, y), f_2(y, a, x)$$

We are going to apply the path transformation to π_a. *Following the different steps, we have:*

1. *The functions calls without filters are:*

$$c_0(a, y) = s(a, y), t(y, z)$$
$$c_1(y, z, x) = s^-(y, z), r(z, x), u(x, z_1)$$

2. *The sub-semantics associated to each function call are:*
   - *For* $c_0 : s(a, y'), t(y', z), t^-(z, y)$
   - *For* $c_1 : s^-(y, z), r(z, x'), u(x', z_1), u^-(z_1, x)$
3. *The path semantics obtained after the concatenation is:*

$$s(a, y'), t(y', z), t^-(z, y), s^-(y, z), r(z, x'), u(x', z_1), u^-(z_1, x)$$

4. *The semantics of* π_a *contained* $r(a, x)$, *so add the atom* $r^-(x, a)$ *to the path semantics.*

At the end of the path transformation, we get

$$\mathcal{P}'(\pi_a) = s(a, y'), t(y', z), t^-(z, y), s^-(y, z), r(z, x'), u(x', z_1), u^-(z_1, x), r^-(x, a)$$

and:

$$\mathcal{P}(\pi_a) = s, t, t^-, s^-, r, u, u^-, r^-$$

This transformation is not a bijection, meaning that possibly multiple plans can generate the same word:

Example 5.6.8. *Consider three path functions:*

- $f_1(x, y) = s(x, y), t(y, z)$,
- $f_2(x, y, z) = s^-(x, y)r(y, z)$,
- $f_3(x, y, z) = s(x, x_0), t(x_0, x_1), t^-(x_1, x_2), s^-(x_2, x_3), r(x_3, y), r^-(y, z)$,

The execution plan $\pi_a^1(x) = f_1(a, y), f_2(y, a, x)$ *then has the same image by the path transformation than the execution plan* $\pi_a^2(x) = f_3(a, a, x)$.

However, it is possible to efficiently reverse the path transformation whenever an inverse exists. We show this in Appendix B.1.2.

Property 5.6.9. *Given a word* w *in* $\mathcal{R}^*$, *a query* $q(x) \leftarrow r(a, x)$ *and a set of path functions, it is possible to know in polynomial time if there exists a non-redundant minimal filtering execution plan* π_a *such that* $\mathcal{P}(\pi_a) = w$. *Moreover, if such a* π_a *exists, we can compute one in polynomial time, and we can also enumerate all of them (there are only finitely many of them).*

5.6.3 Capturing Language

The path transformation gives us a representation of a plan in $\mathcal{R}^*$. In this section, we introduce our main result to characterise *minimal filtering plans*, which are atomic equivalent rewritings based on languages defined on $\mathcal{R}^*$. First, thanks to the path transformation, we introduce the notion of *capturing language*, which allows us to capture equivalent rewritings using a language defined on $\mathcal{R}^*$.

Definition 5.6.10 (Capturing Language)**.** *Let $q(x) \leftarrow r(a, x)$ be an atomic query. The language Λ_q over $\mathcal{R}^*$ is said to be a* capturing language *for the query q (or we say that Λ_q* captures *q) if for all non-redundant minimal filtering execution plans $\pi_a(x)$, we have the following equivalence: π_a is an equivalent rewriting of q iff we have $\mathcal{P}(\pi_a) \in \Lambda_q$.*

Note that the definition of capturing language does not forbid the existence of words $w \in \Lambda_q$ that are not in the preimage of $\mathcal{P}$, i.e., words for which there does not exist a plan π_a such that $P(\pi_a) = w$. We will later explain how to find a language that is a subset of the image of the transformation $\mathcal{P}$, i.e., a language which *faithfully represents plans*.

Our main technical result, which is used to prove Theorem 5.4.7, is that we have a context-free grammar whose language captures q: specifically, the grammar $\mathcal{G}_q$ (Definition 5.5.1):

Theorem 5.6.11. *Given a set of unary inclusion dependencies, a set of path functions, and an atomic query q, the language $\mathcal{L}_q$ captures q.*

5.6.4 Faithfully Representing Plans

We now move on to the second ingredient that we need for our proofs: we need a language which *faithfully represents plans*:

Definition 5.6.12. *We say that a language $\mathcal{K}$* faithfully represents plans *(relative to a set $\mathcal{F}$ of path functions and an atomic query $q(x) \leftarrow r(a, x)$) if it is a language over $\mathcal{R}$ with the following property: for every word w over $\mathcal{R}$, we have that w is in $\mathcal{K}$ iff there exists a minimal filtering non-redundant plan π_a such that $\mathcal{P}(\pi_a) = w$.*

We now show the following about the language of our regular expression P_r of possible plans as defined in Definition 5.5.2.

Theorem 5.6.13. *Let $\mathcal{F}$ be a set of path functions, let $q(x) \leftarrow r(a, x)$ be an atomic query, and define the regular expression P_r as in Definition 5.5.2. Then the language of P_r faithfully represents plans.*

Theorems 5.6.11 and 5.6.13 will allow us to deduce Theorem 5.4.7 and Proposition 5.4.8 from Section 5.4, as explained in Appendix B.2.

5.7 Experiments

We have given an algorithm that, given an atomic query and a set of path functions, generates all equivalent plans for the query (Section 5.5). We now compare our approach experimentally to two other methods, Susie [98], and PDQ [14], on both synthetic datasets and real functions from Web services.

5.7.1 Setup

We found only two systems that can be used to rewrite a query into an equivalent execution plan: Susie [98] and PDQ (Proof-Driven Querying) [14]. We benchmark them against our implementation. All algorithms must answer the same task: given an atomic query and a set of path functions, produce an equivalent rewriting, or claim that there is no such rewriting.

We first describe the Susie approach. Susie takes as input a query and a set of Web service functions and extracts the answers to the query both from the functions and from Web documents. Its rewriting approach is rather simple, and we have reimplemented it in Python. However, the Susie approach is not complete for our task: she may fail to return an equivalent rewriting even when one exists. What is more, as Susie is not looking for equivalent plans and makes different assumptions from ours, the plan that she returns may not be equivalent rewritings (in which case there may be a different plan which is an equivalent rewriting, or no equivalent rewriting at all).

Second, we describe PDQ. The PDQ system is an approach to generating query plans over semantically interconnected data sources with diverse access interfaces. We use the official Java release of the system. PDQ runs the chase algorithm [1] to create a canonical database, and, at the same time, tries to find a plan in that canonical database. If a plan exists, PDQ will eventually find it; and whenever PDQ claims that there is no equivalent plan, then indeed no equivalent plan exists. However, in some cases, the chase algorithm used by PDQ may not terminate as our constraints and views are too general [26]. In this case, it is impossible to know whether the query has a rewriting or not. We use PDQ by first running the chase with a timeout, and re-running the chase multiple times in case of timeouts while increasing the search depth in the chase, up to a maximal depth. The exponential nature of PDQ's algorithm means that already very small depths (around 20) can make the method run for hours on a single query.

Our method is implemented in Python and follows the algorithm presented in the previous section. For the manipulation of formal languages, we used pyformlang (https://pyformlang.readthedocs.io). Our implementation is available online (https://github.com/Aunsiels/query_rewriting). All experiments were run on a laptop with Linux, 1 CPU with 4 cores at 2.5GHz, and 16 GB RAM.

5.7.2 Synthetic Functions

In our first experiments, we consider a set of artificial relations $\mathcal{R} = \{r_1, ..., r_n\}$, and randomly generate path functions up to length 4. Then we tried to find a equivalent plan for each query of the form $r(c, x)$ for $r \in \mathcal{R}$. The set $\mathcal{UID}$ consists of all pairs

of relations $r \rightsquigarrow s$ for which there is a function in whose body r^- and s appear in two successive atoms. We made this choice because functions without these UIDs are useless in most cases.

For each experiment that we perform, we generate 200 random instances of the problem, run each system on these instances, and average the results of each method. Because of the large number of runs, we had to put a time limit of 2 minutes per chase for PDQ and a maximum depth of 16 (so the maximum total time with PDQ for each query is 32 minutes). In practice, PDQ does not strictly abide by the time limit, and its running time can be twice longer. We report, for each experiment, the following numbers:

- Ours: The proportion of instances for which our approach found an equivalent plan. As our approach is proved to be correct, this is the true proportion of instances for which an equivalent plan exists.
- Susie: The proportion of instances for which Susie returned a plan which is actually an equivalent rewriting (we check this with our approach).
- PDQ: The proportion of instances for which PDQ returned an equivalent plan (without timing out): these plans are always equivalent rewritings.
- Susie Requires Assumption: The proportion of instances for which Susie returned a plan, but the returned plan is not an equivalent rewriting (i.e., it is only correct under the additional assumptions made by Susie).
- PDQ Timeout: The proportion of instances for which PDQ timed out (so we cannot conclude whether a plan exists or not).

In all cases, the two competing approaches (Susie and PDQ) cannot be better than our approach, as we always find an equivalent rewriting when one exists, whereas Susie may fail to find one (or return a non-equivalent one), and PDQ may timeout. The two other statistics (Susie Requires Assumption, and PDQ Timeout) denote cases where our competitors fail, which cannot be compared to the performance of our method.

In our first experiment, we limited the number of functions to 15, with 20% of existential variables, and varied the number n of relations. Both Susie and our algorithm run in less than 1 minute in each setting for each query, whereas PDQ may timeout. Figure 5.2 shows which percentage of the queries can be answered. As expected, when the number of relations increases, the rate of answered queries decreases as it becomes harder to combine functions. Our approach can always answer strictly more queries than Susie and PDQ.

In our next experiment, we fixed the number of relations to 7, the probability of existential variables to 20%, and varied the number of functions. Figure 5.3 shows the results. As we increase the number of functions, we increase the number of possible function combinations. Therefore, the percentage of answered queries increases both for our approach and for our competitors. However, our approach answers about twice as many queries as Susie and PDQ.

In our last experiment, we fixed the number of relations to 7, the number of functions to 15, and we varied the probability of having an existential variable.

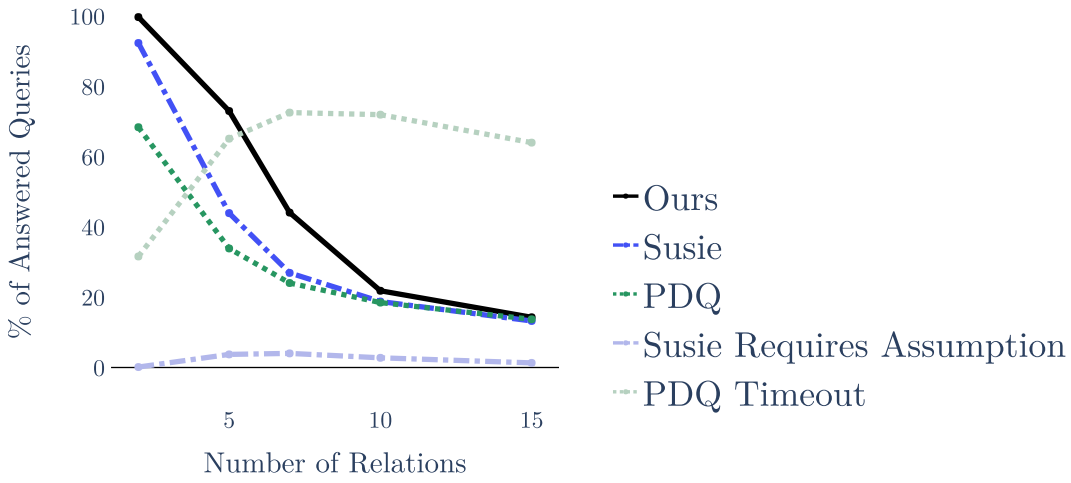

Figure 5.2: Percentage of answered queries with varying number of relations

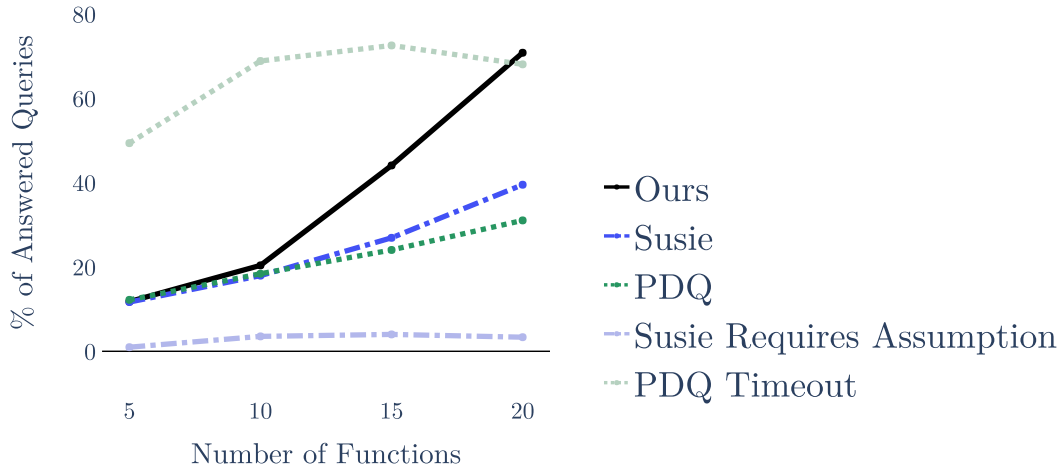

Figure 5.3: Percentage of answered queries with varying number of functions

Figure 5.4 shows the results. As we increase the probability of existential variables, the number of possible plans decreases because fewer outputs are available to call other functions. However, the impact is not as marked as before, because we have to impose at least one output variable per function, which, for small functions, results in few existential variables. As Susie and PDQ use these short functions in general, changing the probability did not impact them too much. Still, our approach can answer about twice as many queries as Susie and PDQ.

5.7.3 Real-World Web Services

We consider the functions of Abe Books (https://www.abebooks.fr/), ISBNDB (http://isbndb.com/), LibraryThing (http://www.librarything.com/), and MusicBrainz (http://musicbrainz.org/), all used in [98], and Movie DB (https://www.themoviedb.org) to replace the (now defunct) Internet Video Archive used in [98]. We add to these functions some other functions built by the Susie

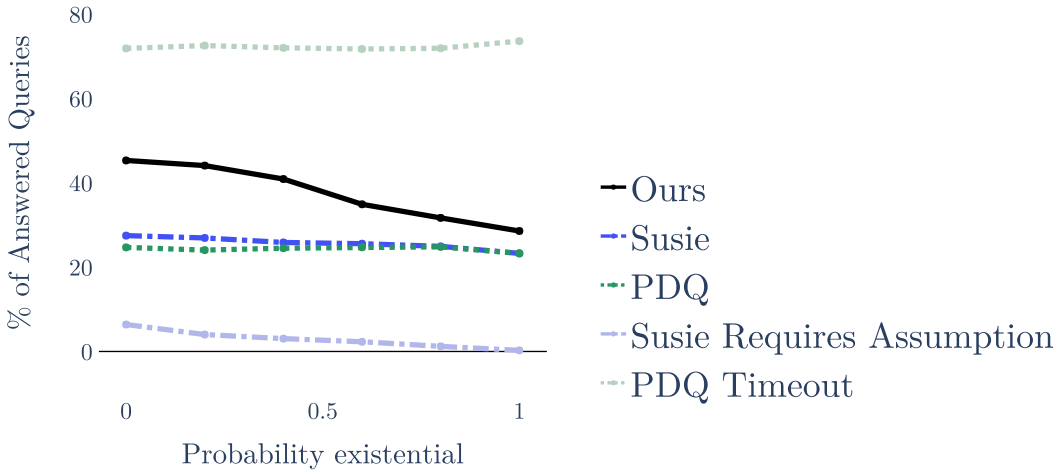

Figure 5.4: Percentage of answered queries with varying number of existential variables

approach. We group these Web services into three categories: Books, Movies, and Music, on which we run experiments separately. For each category, we manually map all services into the same schema and generate the UIDs as in Section 5.7.2. Our dataset is available online (see URL above).

The left part of Table 5.1 shows the number of functions and the number of relations for each Web service. Table 5.2 gives examples of functions. Some of them are recursive. For example, the first function in the table allows querying for the collaborators of an artist, which are again artists. This allows for the type of infinite plans that we discussed in the introduction, and that makes query rewriting difficult.

For each Web service, we considered all queries of the form $r(c, x)$ and $r^-(c, x)$, where r is a relation used in a function definition. We ran the Susie algorithm, PDQ, and our algorithm for each of these queries. The runtime is always less than 1 minute for each query for our approach and Susie but can timeout for PDQ. The time limit is set to 30 minutes for each chase, and the maximum depth is set to 16. Table 5.1 shows the results, similarly to Section 5.7.2. As in this case, all plans returned by Susie happened to be equivalent plans, we do not include the "Susie Requires Assumption" statistic (it is 0%). Our approach can always answer more queries than Susie and PDQ, and we see that with more complicated problems (like Music), PDQ tends to timeout more often.

In terms of the results that we obtain, some queries can be answered by rather short execution plans. Table 5.3 shows a few examples. However, our results show that many queries do not have an equivalent plan. In the Music domain, for example,

Web Service	Functions	Relations	Susie	PDQ (timeout)	Ours
Movies	2	8	13%	**25**% (0%)	**25**%
Books	13	28	57%	64% (7%)	**68**%
Music	24	64	22%	22% (25%)	**33**%

Table 5.1: Web services and results

it is not possible to answer *produced*(c, x) (i.e., to know which albums a producer produced), *hasChild*$^-$*(c,x)* (to know the parents of a person), and *rated*$^-(c, x)$ (i.e., to know which tracks have a given rating). This illustrates that the services maintain control over the data, and do not allow arbitrary requests.

5.8 Visualisation Demo

We have implemented a demo for our system. In our demo, the user can specify a set of functions and a query. The demo will then draw the functions, search for an execution plan that answers the query by composing the functions, and animate it to the user. The functions and the query are specified in textual form – simply by giving their sequence of relations, as in:

- *getAlbum : onAlbum*
- *getAlbumDetails : onAlbum*$^-$*, singer*
- *query : singer*

Our demo is shown in Figure 5.5 (left). The query is shown as a red edge. The input constant to the query is represented by a generic placeholder "IN". A click on the "Save&Run" button runs our algorithm, and proposes an execution plan if one exists. This plan is then animated by moving the graphical function representations into their place, so that we can see that the plan is equivalent to the query. The figure shows the second function moving into its place. The user can play with different function definitions, and also rerun the animation.
It is also possible to load function definitions of real Web services ("Load file" button). We provide function definitions for Abe Books (https://www.abebooks.fr/), ISBNDB (http://isbndb.com/), LibraryThing (http://www.librarything.com/), and MusicBrainz (http://musicbrainz.org/) from Chapter 5. Figure 5.5 (right) shows the final plan after the user has loaded music-related functions and asked the query *released(x,y)*. Note how the plan consists of several forward-backward paths. The stars in the function definitions (and the corresponding empty arrow heads in their graphical representation) indicate variables that appear in the body of the function, but are not an output variable.

Technically, our demo is an HTML page with JavaScript that runs in a Web browser at http://dangie.r2.enst.fr/. Once the user clicks on "Save&Run", the function definitions are passed to a Python implementation of our algorithm from [108]

GetCollaboratorsByID(artistId, collab, collabId) ←
hasId$^-$(artistId,artist), isMemberOf(artist,collab), hasId(collab,collabId)
GetBookAuthorAndPrizeByTitle(title, author, prize) ←
isTitled$^-$(title, book), wrote$^-$(book,author), hasWonPrize(author,prize)
GetMovieDirectorByTitle(title, director) ←
isTitled$^-$(title,movie), directed$^-$(movie,director)

Table 5.2: Examples of real functions

Query	Execution Plan
released	GetArtistInfoByName, GetReleasesByArtistID, GetArtistInfoByName, GetTracksByArtistID, GetTrackInfoByName, GetReleaseInfoByName
published	GetPublisherAuthors, GetBooksByAuthorName
actedIn	GetMoviesByActorName, GetMovieInfoByName

Table 5.3: Example plans

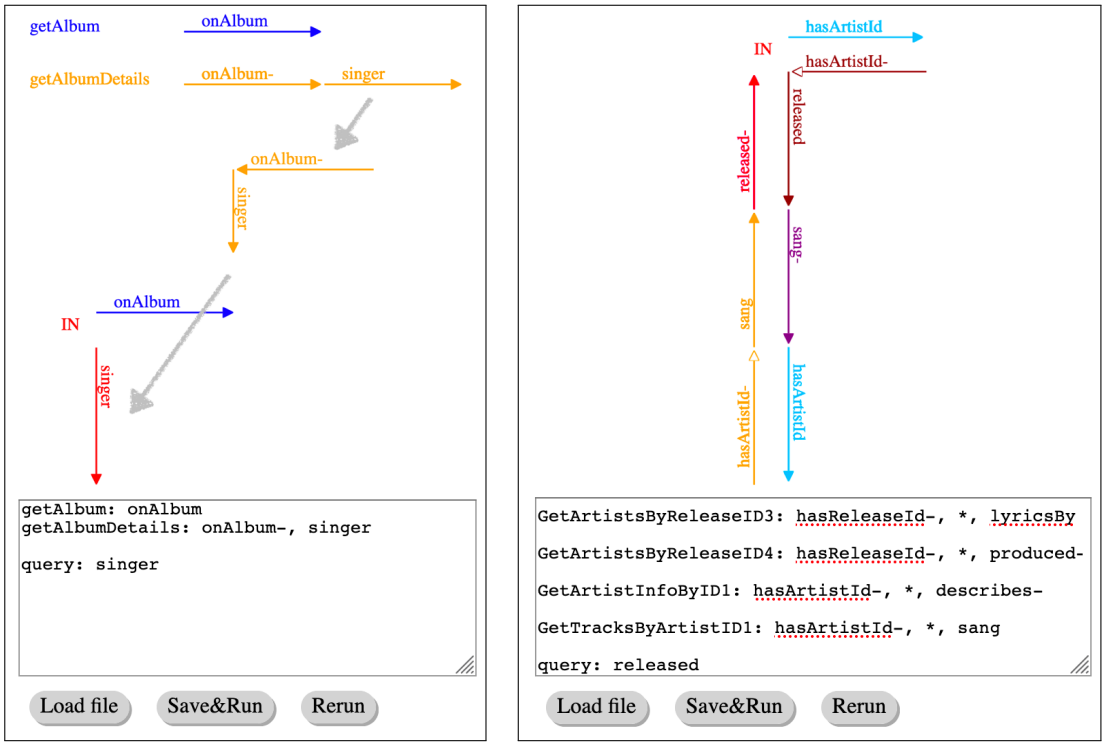

Figure 5.5: Screenshots of our demo. **Left:** Our toy example, with the functions on top and the plan being constructed below. The gray arrows indicate the animation. **Right:** A plan generated for real Web service functions

to compute the plan. The page then displays the plan as SVG, which is animated with `animation` tags.

5.9 Conclusion

In this chapter, we have addressed the problem of finding equivalent execution plans for Web service functions. We have characterised these plans for atomic queries and path functions, and we have given a correct and complete method to find them. Our experiments have demonstrated that our approach can be applied to real-world Web services and that its completeness entails that we always find plans for more queries than our competitors. All experimental data, as well as all code, is available at the URL given in Section 5.7. We hope that our work can help Web service providers to design their functions, and users to query the services more efficiently. For future work, we aim to broaden our results to non-path functions. We also

intend to investigate connections between our theoretical results and the methods by Benedikt et al. [13], in particular possible links between our techniques and those used to answer regular path queries under logical constraints [18].

Chapter 6

Query Rewriting Without Integrity Constraints

> Le petit Poucet croyait retrouver aisément son chemin, par le moyen de son pain qu'il avait semé partout où il avait passé; mais il fut bien surpris lorsqu'il ne put en retrouver une seule miette: les oiseaux étaient venus, qui avaient tout mangé.
>
> *Charles Perrault*

In Chapter 5, we studied query rewritings under path views with integrity constraints. However, we do not necessarily have integrity constraints over our data. So, in this chapter, we restrict the TBox to two components: the relations and the entities. Equivalent rewritings are easy to find in this case. However, as we will see, they are of little use in practice. Therefore, we propose a new class of plans, the *smart plans*, and characterise them.

This chapter comes from [109]:

> Romero, J., Preda, N., & Suchanek, F.
> Query Rewriting On Path Views Without Integrity Constraints
> Workshop paper at Datamod 2020

****

6.1 Introduction

In this chapter, we consider the example illustrated in Figure 6.1. In this example, if we want to find the job title of Anna, we first have to find her company (by calling *getCompany*), and then her job title (by calling *getHierarchy* on her company, and filtering the results about Anna). Chapter 5 and much of the literature concentrates on finding *equivalent rewritings*, i.e., execution plans that deliver the same result

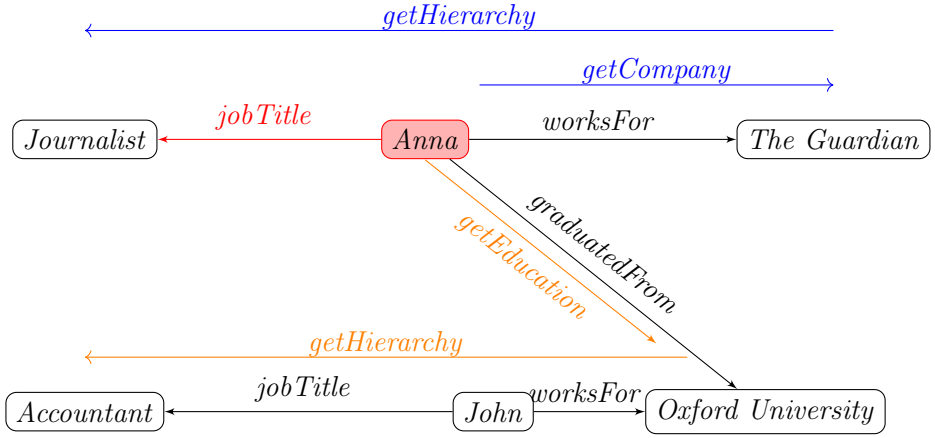

Figure 6.1: An equivalent execution plan (blue) and a maximal contained rewriting (orange) executed on a database (black)

as the original query on all databases. Unfortunately, our example plan is not an equivalent rewriting: it will deliver no results on databases where (for whatever reasons) Anna has a job title but no employer. The plan is equivalent to the query only if an integrity constraint stipulates that every person with a job title must have an employer.

Such constraints are hard to come by in real life, because they may not hold (a person can have a job title but no employer; a person may have a birth date but no death date; some countries do not have a capital like the Republic of Nauru). Even if they hold in real life, they may not hold in the database due to the incompleteness of the data. Hence, they are also difficult to mine automatically. In the absence of constraints, however, an atomic query has an equivalent rewriting only if there is a function that was defined precisely for that query. One solution is to resort to *maximally contained rewritings*. Intuitively speaking, these are execution plans that try to find all calls that could potentially lead to an answer. In our example, the plan *getAlmaMater*, *getHierarchy* is included in the maximally contained rewriting: It asks for the university where Anna graduated, and for their job positions. If Anna happens to work at the university where she graduated, this plan will answer the query.

This plan appears somehow less reasonable that our first plan because it works only for people who work at their alma mater. However, both plans are equal concerning their formal guarantees: none of them can guarantee to deliver the answers to the query. This is a conundrum: Unless we have information about the data distribution or more schema information, we have no formal means to give the first plan higher chances of success than the second plan – although the first plan is intuitively much better.

In this chapter, we propose a solution to this conundrum: We can show that the first plan (*getCompany*, *getHierarchy*) is "smart", in a sense that we formally define. We can give guarantees about the results of smart plans in the absence of integrity constraints. We also give an algorithm that can enumerate all smart plans for a given atomic query and path-shaped functions (as in Figure 6.1). We show that under a condition that we call the *Optional Edge Semantics* our algorithm is

complete and correct, i.e., it will exhaustively enumerate all such smart plans. We apply our method to real Web services and show that smart plans work in practice and deliver more query results than competing approaches.

This chapter is structured as follows: Section 6.2 introduces some additional preliminaries, and Section 6.3 gives a definition of smart plans. Section 6.4 provides a method to characterise smart plans, and Section 6.5 gives an algorithm that can generate smart plans. We provide extensive experiments on synthetic and real Web services to show the viability of our method in Section 6.6.

6.2 Preliminaries

We use the terminology of Chapter 5. We slightly modify the notation of an execution plan as it is more convenient in this chapter. But first, we give example functions:

Example 6.2.1. *Consider our example in Figure 6.1. There are 3 relation names in the database: worksFor, jobTitle, and graduatedFrom. The functions are:*

$$\begin{aligned} \mathit{getCompany}(\underline{x}, y) &\leftarrow \mathit{worksFor}(\underline{x}, y) \\ \mathit{getHierarchy}(\underline{y}, x, z) &\leftarrow \mathit{worksFor}^{-}(\underline{y}, x), \mathit{jobTitle}(x, z) \\ \mathit{getEducation}(\underline{x}, y) &\leftarrow \mathit{graduatedFrom}(\underline{x}, y) \end{aligned}$$

The first function takes as input a person x, and returns as output the organisation y. The second function takes as input an organisation y and returns the employees x with their job title y. The last function returns the university y from where a given person x graduated.

Calling a function for a given value of the input variable means finding the result of the query given by the body of the function on a database instance.

Plans. A *plan* takes the form

$$\pi_a(x) = c_1, \ldots, c_n, \gamma_1 = \delta_1, \ldots, \gamma_m = \delta_m$$

Here, a is a constant and x is the output variable. Each c_i is a *function call* of the form $f(\underline{\alpha}, \beta_1, \ldots, \beta_n)$, where f is a function name, the input α is either a constant or a variable occurring in some call in $c_1, \ldots, c_{i-1}$, and the outputs $\beta_1, \ldots, \beta_n$ are variables. Each $\gamma_j = \delta_j$ is called a *filter*, where γ_j is an output variable of any call, and δ_j is either a variable that appears in some call or a constant. If the plan has no filters, then we call it *unfiltered*. The *semantics* of the plan is the query:

$$q(x) \leftarrow \phi(c_1), \ldots, \phi(c_n), \gamma_1 = \delta_1, \ldots, \gamma_m = \delta_m$$

where each $\phi(c_i)$ is the body of the query defining the function f of the call c_i in which we have substituted the constants and variables used in c_i. We have used fresh existential variables across the different $\phi(c_i)$, where x is the output variable of the plan, and where $\cdot = \cdot$ is an atom that holds in any database instance if and only if its two arguments are identical.

To *evaluate* a plan on an instance means running the query above. In practice, this boils down to calling the functions in the order given by the plan. Given an execution plan π_a and a database I, we call $\pi_a(I)$ the answers of the plan on I.

Example 6.2.2. *The following is an execution plan for Example 6.2.1:*

$$\pi_{Anna}(y) = getCompany(\underline{Anna}, x), \\ getHierarchy(\underline{x}, z, y), z = Anna$$

The first element is a function call to getCompany with the name of the person (Anna) as input, and the variable x as output. The variable x then serves as input in the second function call to getHierarchy. Figure 6.1 shows the plan with an example instance. This plan defines the query:

$$worksFor(Anna, x), worksFor^{-}(x, z), \\ jobTitle(z, y), z = Anna$$

In our example instance, we have the embedding:

$$\sigma = \{x \to The\ Guardian, y \to Journalist, z \to Anna\}.$$

An execution plan $\pi_a(x)$ is *redundant* if it has no call using the constant a as input, or if it contains a call where none of the outputs is an output of the plan or an input to another call.

An *equivalent rewriting* of an atomic query $q(x) \leftarrow r(a, x)$ is an execution plan that has the same results as q on all database instances. For our query language, a *maximally contained rewriting* for the query q is a plan whose semantics contains the atom $r(y, x)$ for some variable y.

6.3 Defining Smart Plans

6.3.1 Introductory Observations

Given an atomic query, and given a set of path functions, our goal is to find an execution plan that answers the query. In our example from Figure 6.1, our goal is to find an execution plan of the three available functions *getHierarchy*, *getCompany*, *getEducation* in order to answer the query $q(x) \leftarrow jobTitle(Anna, x)$. The plan we aim at is:

$$\pi_{Anna}(z) = getCompany(\underline{Anna}, x), getHierarchy(\underline{x}, y, z), y = Anna \tag{6.3.1}$$

We have already seen that, in the absence of integrity constraints, it may be impossible to find an equivalent rewriting for a query on a given set of functions. In our example from Figure 6.1, there exists no equivalent rewriting. In particular, the plan 6.3.1 is not an equivalent rewriting, because it will deliver the same results as the query only on databases where Anna works at a company.

The standard solution is to resort to maximally contained rewritings [73]. The plan above is part of the maximally contained rewriting, but so is the following plan:

$$\pi_{Oxford}(y) = getHierarchy(\underline{OxfordUniversity}, x, y), x = Anna \tag{6.3.2}$$

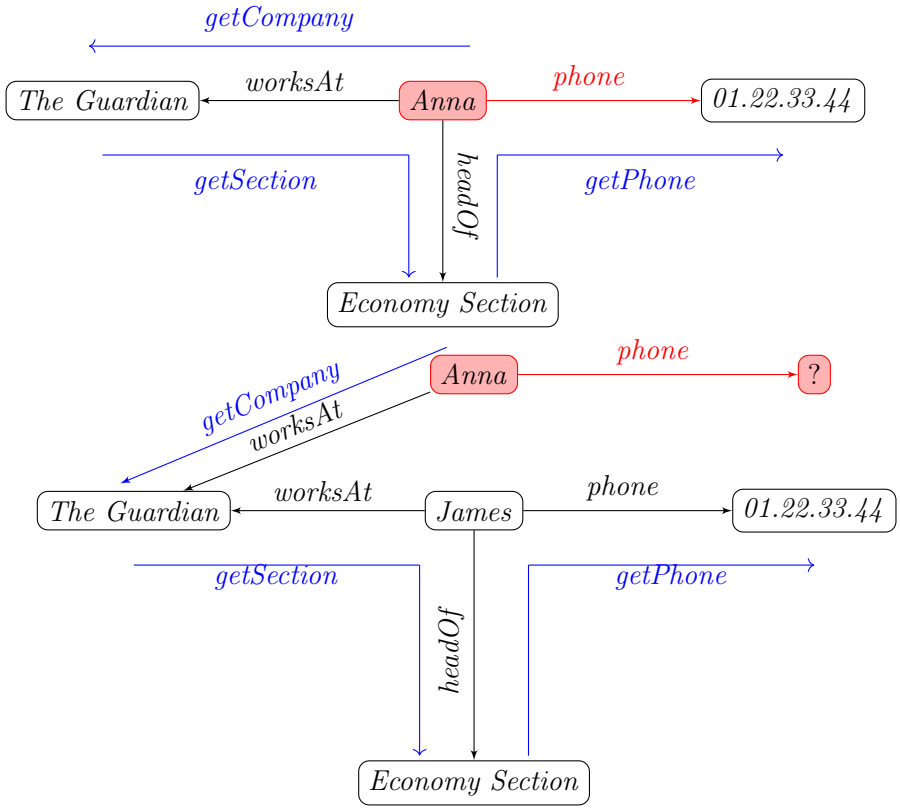

Figure 6.2: A non-smart execution plan for the query *phone(Anna,x)*. Top: a database where the plan answers the query. Bottom: a database where the unfiltered plan has results, but the filtered plan does not answer the query

This plan will answer the query only if Anna happens to work at Oxford University – which is a very strong hypothesis. The following plan is also part of the maximally contained rewriting:

$$\begin{aligned}\pi_{Anna}(t) = & getFriends(\underline{Anna}, x), \\ & getFriends(\underline{x}, y), getCompany(\underline{y}, z), \\ & getHierarchy(\underline{z}, p, t), p = Anna \end{aligned} \tag{6.3.3}$$

This plan asks for the persons y who are friends of the friends of Anna and then obtains their positions in their respective companies. This plan will work if Anna happens to have friends at the same company – which is as counter-intuitive as the previous plan. More worryingly, one can prolong the plan infinitely, or until the transitive closure is computed, by more calls to the function *getFriends*. As shown in [45], unions and recursive plans must be considered in maximally contained rewritings, and there are no bounds on the size of the recursion. Hence, it seems that only the first execution plan 6.3.1 is "smart". The reason appears to be that if the database contains an answer to the query, and if the plan delivers a result, then the plan will answer the query. Nevertheless, this definition will not work because it would also make the plan 6.3.2 smart.

6.3.2 Smart Plan Definition

We propose the following definition of smart plans:

Definition 6.3.1 (Smart Plan)**.** *Given an atomic query q and a set of functions, a plan π is* smart *if the following holds on all database instances I: If the filter-free version of π has a result on I, then π delivers exactly the answers to the query.*

It is easy to see that this definition separates the wheat from the chaff: Plan 6.3.1 is smart in this sense, whereas Plans 6.3.2 and 6.3.3 are not. We also introduce weakly smart plans:

Definition 6.3.2 (Weakly Smart Plan)**.** *Given an atomic query q and a set of functions, a plan π is* weakly smart *if the following holds on all database instances I where q has at least one result: If the filter-free version of π has a result on I, then π delivers a super-set of the answers to the query.*

Every smart plan is also a weakly smart plan. Some queries will admit only weakly smart plans and no smart plans, mainly because the variable that one has to filter on is not an output variable. Nevertheless, weakly smart plans can be useful: For example, if a data provider wants to hide private information, they do not want to leak it in any way, not even among other results. Thus, they will want to design their functions in such a way that no weakly smart plan exists for this specific query.

6.3.3 Comparison with Susie

The plans generated in Susie [98] are smart according to our definition. However, they only represent a subset of the smart plans. For instance, consider again the query *holdsPosition(Anna, x)*, and assume that, in addition to the function *getHierarchy*, we have the following two functions:

$$\begin{aligned} \mathit{getProfessionalAddress}(\underline{x}, y, z) &\leftarrow \mathit{worksFor}(\underline{x}, y), \mathit{locatedIn}(y, z) \\ \mathit{getEntityAtAddress}(\underline{x}, y) &\leftarrow \mathit{locatedIn}(\underline{x}, y) \end{aligned}$$

Then the following plan is smart (see Figure 6.3):

$$\begin{aligned} \pi_{Anna}(x) = &\mathit{getProfessionalAddress}(Anna, y, z), \\ &\mathit{getEntityAtAddress}(\underline{z}, y'), \mathit{getHierarchy}(\underline{y'}, t, x) \end{aligned}$$

However, it will not be discovered by the algorithm in [98].

6.3.4 Comparison with Equivalent Rewritings

We have already seen that there exists no equivalent rewriting of our query in our example from Figure 6.1, because there are no constraints. Now we could assume constraints. If we add, e.g., inclusion dependencies between all relations, then the work of Chapter 5 would indeed find Plan 6.3.1. However, it would also find non-smart plans. As an example, consider the plan in Figure 6.2. It tries to find the phone number of Anna. The plan calls the functions *getCompany* (which delivers

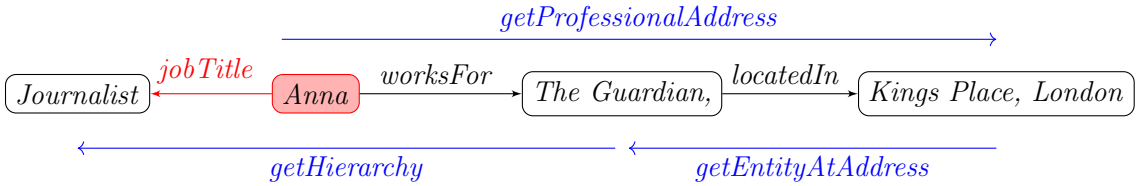

Figure 6.3: A smart plan for the query $jobTitle(Anna, ?x)$, which Susie will not find

The Guardian), and then *getSections* (which delivers the sections with their heads), and finally *getPhone*, which delivers the head of the section with their phone number. This plan is not smart: If Anna is not the head of a section, the unfiltered plan will return the phone number of someone else. Hence, the filtered plan will not answer the query, even though the unfiltered plan returned a result.

6.3.5 Sub-Smart Definition

In the definitions introduced in Section 6.3.2, we focused on plans which are able to return *all* results of a query. However, the user may be only interested in getting only one answer to his query. For example, if we are looking for the email of a professor, it is enough to find only one of them to contact him. Thus, we introduce the notion of *sub-smart plans*, which is a variation of the smart plans.

Definition 6.3.3 (Sub-Smart Plan)**.** *Given an atomic query q and a set of functions, a plan π is* sub-smart *if the following holds on all database instances I: If the filter-free version of π has a result on I, then π delivers a non-empty subset of the answers to the query.*

Like for smart plans, we also introduce a weak version of the previous definition:

Definition 6.3.4 (Weakly Sub-Smart Plan)**.** *Given an atomic query q and a set of functions, a plan π is* weakly sub-smart *if the following holds on all database instances I where q has at least one result: If the filter-free version of π has a result on I, then π contains a non-empty subset of the answers to the query.*

We note that if a plan is smart (resp. weakly smart) then it is also sub-smart (resp. weakly sub-smart).

6.4 Characterizing Smart Plans

6.4.1 Web Service Functions

We now turn to recognising smart plans. As previously stated, our approach can find smart plans only under a certain condition. This condition has to do with the way Web services work. Assume that for a given person, a function returns the employer and the address of the working place:

$$getCompanyInfo(\underline{x}, y, z) \leftarrow worksAt(\underline{x}, y), locatedIn(y, z)$$

Now assume that, for some person, the address of the employer is not in the database. In that case, the call will not fail. Rather, it will return only the employer y, and return a null-value for the address z. It is as if the atom *locatedIn(y, z)* were optional. In Chapter 5, we supposed that the function must have a binding for all its variables (output and existential) to yield a result. Here, with *getLifeDates*, the function would not have given any answer for *Elvis*. To model this phenomenon, we introduce the notion of *sub-functions*:

Definition 6.4.1 (Sub-Function)**.** *Given a path function* $f(\underline{x_0}, x_{i_1}, ..., x_{i_m}) \leftarrow r_1(\underline{x_0}, x_1), r_2(x_1, x_2), \ldots r_n(x_{n-1}, x_n)$, $0 < i_1 < \ldots < i_m \leq n$, *the sub-function associated to an output variable* x_{i_k} *is the function* $f_k(\underline{x_0}, x_{i_1}, ..., x_{i_k}) \leftarrow r_1(\underline{x_0}, x_1), \ldots r_{i_k}(x_{i_k-1}, x_{i_k})$.

Example 6.4.2. *The sub-functions of the function getCompanyInfo are* $f_1(\underline{x}, y) \leftarrow worksAt(\underline{x}, y)$, *which is associated to* y, *and* $f_2(\underline{x}, y, z) \leftarrow worksAt(\underline{x}, y), locatedIn(y, z)$, *which is associated to* z.

We can now express the Optional Edge Semantics:

Definition 6.4.3 (Optional Edge Semantics)**.** *We say that we are under the optional edge semantics if, for any path function* f, *a sub-function of* f *has exactly the same binding for its output variables as* f.

The optional edge semantics mirrors the way real Web services work. Its main difference to the standard semantics is that it is not possible to use a function to filter out query results. For example, it is not possible to use the function *getCompanyInfo* to retrieve only those people who work at a company with a known address. The function will retrieve companies with addresses and companies without addresses, and we can find out the companies without addresses only by skimming through the results after the call. This contrasts with the standard semantics of parametrised queries (as used, e.g., in [97, 98, 108]), which do not return a result if any of their variables cannot be bound.

This has a very practical consequence: As we shall see, smart plans under the optional edge semantics have a very particular shape.

6.4.2 Why We Can Restrict to Path Queries

Under the optional edge semantics, we can query for all partial results. Now if we have all sub-functions of a given function f, then f itself is no longer necessary (except if it is itself a sub-function). We call this substitution by sub-functions the sub-function transformation:

Definition 6.4.4 (Sub-Function Transformation)**.** *Let* $\mathcal{F}$ *be a set of path functions and* $\mathcal{F}_{sub}$ *the set of sub-functions of* $\mathcal{F}$. *Let* π_a *be a non-redundant execution plan over* $\mathcal{F}$. *Then, we define the Sub-Function Transformation of* π_a, *written* $\mathcal{P}_{sub}(\pi_a)$, *as the non-redundant execution plan over the sub-functions* $\mathcal{F}_{sub}$ *as follows:*

- *The output is the same than* π_a

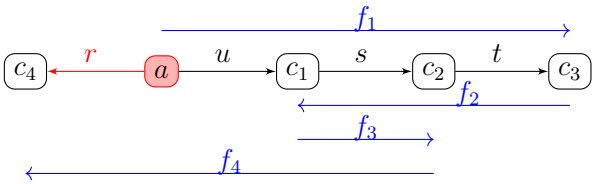

Figure 6.4: A bounded plan

- *Each function call c in π_a associated to a path function f is replaced by the smallest sub-function of f which contains the output variables which are either the output of the plan, used by other calls or involved in filters.*

This transformation preserves smartness.

Property 6.4.5. *Let $q(x) \leftarrow r(a, x)$ be an atomic query and $\mathcal{F}$ be a set of path functions. Let π_a be a non-redundant execution plan composed of the sequence of calls $c_1, ..., c_n$. Then, under the optional edge semantics, π_a is smart (resp. weakly smart, sub-smart, weakly sub-smart) iff its sub-function transformation $\mathcal{P}_{sub}(\pi_a)$ is smart (resp. weakly smart, sub-smart, weakly sub-smart).*

Proof. This property follows directly from the optional edge semantics: The outputs of the sub-function are precisely the same as the outputs of the full function for the same variables. So, we can apply all the filters and have the same effect. □

The property tells us that under the *optional edge semantics*, we can replace all path functions by one of their sub-functions.

Finally, in the case of constraint-free plans, we have that the sub-function transformation creates a path query, which will be easier to manipulate.

Property 6.4.6. *Let π_a be a constraint-free non-redundant execution plan. Then the semantics of the sub-function transformation $\mathcal{P}_{sub}(\pi_a)$ is a path query where the last variable of the last atom is the output atom.*

Proof. This property follows directly the fact that in a constraint-free plan, we require only one output per function. Therefore, function calls can be chained, and the last variable of the last atom is the output of the plan (we require nothing else after that). □

We can also deduce from this property that once we have transformed a constraint-free execution plan (exploited in weakly smart plans) to use only sub-functions, we can write the semantics of the plan unambiguously as a skeleton. In particular, we could consider that each sub-function has only one output.

In the end, we can see that it is safe to consider only execution plans whose semantics is a path query in the case of constraint-free plans. We shall see that this is also the case for minimal-filtering non-redundant execution plans.

6.4.3 Preliminary Definitions

Our main intuition is that smart plans under the optional edge semantics walk forward until a turning point. From then on, they "walk back" to the input constant and query (see again Figure 6.1). As a more complex example, consider the atomic query $q(x) \leftarrow r(a, x)$ and the database shown in Figure 6.4. The plan f_1, f_2, f_3, f_4 is shown in blue. As we can see, the plan walks "forward" and then "backward" again. Intuitively, the "forward path" makes sure that certain facts exist in the database (if the facts do not exist, the plan delivers no answer, and is thus trivially smart). If these facts exist, then all functions on the "backward path" are guaranteed to deliver results. Thus, if a has an r-relation, the plan is guaranteed to deliver its object. Let us now make this intuition more formal.

We first observe (and prove in Property 6.4.6) that the semantics of any filter-free execution plan can be written as a path query. The path query of Figure 6.4 is

$$q(a, x) \leftarrow u(a, y_1), s(y_1, y_2), t(y_2, y_3), t^-(y_3, y_2), s^-(y_2, y_1), \\ s(y_1, y_2), s^-(y_2, y_1), u^-(y_1, y_0), r(y_0, x)$$

Now any filter-free path query can be written unambiguously as the sequence of its relations – the *skeleton*. In the example, the skeleton is

$$u.s.t.t^-.s^-.s.s^-.u^-.r$$

In particular, the skeleton of an atomic query $q(x) \leftarrow r(a, x)$ is just r. Given a skeleton $r_1 r_2 ... r_n$, we write $r_1 ... r_n(a)$ for the set of all answers of the query when a is given as input. For path functions, we write the name of the function as a shorthand for the skeleton of the semantics of the function. For example, in Figure 6.4, we have $f_1(a) = \{c_3\}$, and $f_1 f_2 f_3 f_4(a) = \{c_4\}$. We now introduce two notions to formalise the "forward and backward" movement:

Definition 6.4.7 (Forward and Backward Step)**.** *Given a sequence of relations $r_0 ... r_n$ and a position $0 \leq i \leq n$, a* forward step *consists of the relation r_i, together with the updated position $i + 1$. Given position $1 \leq i \leq n + 1$, a* backward step *consists of the relation r_{i-1}^-, together with the updated position $i - 1$.*

Definition 6.4.8 (Walk)**.** *A* walk *to a position k ($0 \leq k \leq n$) through a sequence of relations $r_0 ... r_n$ consists of a sequence of steps (forward or backward) in $r_0 ... r_n$, so that the first step starts at position $n + 1$, every step starts at the updated position of the previous step, and the last step leads to the updated position k.*

If we do not mention k, we consider that $k = 0$, i.e., we cross the sequence of relations entirely.

Example 6.4.9. *In Figure 6.4, a possible walk through r^-ust is $t^-s^-ss^-u^-r$. This walk goes from c_3 to c_2 to c_1, then to c_2, and back through c_1, c, c_4 (as indicated by the blue arrows).*

We can now formalise the notion of the forward and backward path:

Definition 6.4.10 (Bounded plan)**.** *A bounded path for a set of relations $\mathcal{R}$ and a query $q(x) \leftarrow r(a, x)$ is a path query P, followed by a walk through r^-P. A bounded plan for a set of path functions $\mathcal{F}$ is a non-redundant execution plan whose semantics are a bounded path. We call P the forward path and the walk though r^-P the backward path.*

Example 6.4.11. *In Figure 6.4, $f_1f_2f_3f_4$ is a bounded path, where the forward path is f_1, and the backward path $f_2f_3f_4$ is a walk through r^-f_1.*

6.4.4 Characterising Weakly Smart Plans

Our notion of bounded plans is based purely on the notion of skeletons, and does not make use of filters. This is not a problem, because weakly smart plans do not need filters (i.e., if we remove the filters from a weakly smart plan, it will still be a weakly smart plan). Furthermore, we showed in Section 6.4.2 that we can restrict ourselves to execution plans whose semantics is a path query. This allows for the following theorems:

Theorem 6.4.12 (Correctness)**.** *Let $q(x) \leftarrow r(a, x)$ be an atomic query, F a set of path functions and F_{sub} the set of sub-functions of F. Let π_a be a non-redundant bounded execution plan over the F_{sub} such that its semantics is a path query. Then π_a is weakly smart.*

Proof. As π_a is a constraint-free non-redundant execution plan over the F_{sub}, its sub-function transformation can be written as a path query (see Property 6.4.6). We now consider we have this form.

Let π_a be a bounded plan for a query $q(x) \leftarrow r(a, x)$. Assume a database $\mathcal{I}$ such that $q(\mathcal{I}) \neq \emptyset$ and $\pi_a(\mathcal{I}) \neq \emptyset$ (such a database exists). Choose any constant $c_0 \in q(\mathcal{I})$. We have to show that $c_0 \in \pi_a(\mathcal{I})$. Since π_a is bounded, its consequences can be split into a forward path $F = r_1...r_m$ and a following backward path $B = r'_1...r'_n$ with $r'_n = r$ (by definition of a walk). Since $\pi_a(\mathcal{I}) \neq \emptyset$ it follows that $F(a) \neq \emptyset$. Hence, $\mathcal{I}$ must contain $r_1(a, c_2)...r_m(c_m, c_{m+1})$ for some constants $c_1, ..., c_{m+1}$. Since $r(a, c_0) \in \mathcal{I}$, the database $\mathcal{I}$ must contain $r^-(c_0, a)r_1(a, c_2)...r_m(c_m, c_{m+1})$. $B = r'_1...r'_n$ is a walk through r^-F. Let us prove by induction that $c_i \in Fr'_1...r'_j(a)$ if r'_j was generated by a step that leads to position i. To simplify the proof, we call $c_1 = a$.

The walk starts at position $i = m + 1$ with a backward step, leading to position $i = m$. Hence, $r'_1 = r^-_m$. Thus, we have $c_m \in Fr'_1(a)$. Now assume that we have arrived at some position $i \in [1, m]$, and that $c_i \in Fr'_1...r'_j(a)$, where r'_j was generated by the previous step. If the next step is a forward step, then $r_{j+1} = r_i$, and the updated position is $i+1$. Hence, $c_{i+1} \in Fr'_1...r'_{j+1}(a)$. If the next step is a backward step, then $r_{j+1} = r^-_{i-1}$, and the updated position is $i-1$. Hence, $c_{i-1} \in Fr'_1...r'_{j+1}(a)$. It follows then that $c_0 \in FB(a)$ as the walk ends at position 0. □

Theorem 6.4.13 (Completeness)**.** *Let $q(x) \leftarrow r(a, x)$ be an atomic query, F a set of path functions and F_{sub} the set of sub-functions of F. Let π_a be a weakly smart plan over F_{sub} such that its semantics is a path query. Then π_a is bounded.*

Proof. As π_a is a constraint-free non-redundant execution plan over the F_{sub}, its sub-function transformation can be written as a path query (see Property 6.4.6). We now consider we have this form.

Let π_a be a weak smart plan for a query $q(x) \leftarrow r(a, x)$, with consequences $r_1(x_1, x_2)... \ r_n(x_n, x_{n+1}), \ r_{n+1}(x_{n+1}, x_{n+2})$. Without loss of generality, we suppose that $r_1 \neq r$ (the proof can be adapted to work also in this case). Consider the database $\mathcal{I}$

$$\mathcal{I} = \{r^-(c_0, a), r_1(a, c_2), ..., r_n(c_n, c_{n+1}), r_{n+1}(c_{n+1}, c_{n+2})\}$$

Here, the c_i are fresh constants. For convenience, we write $a = c_1$. On this database, $\pi_a(\mathcal{I}) \supseteq \{c_{n+2}\} \neq \emptyset$ and $q(\mathcal{I}) = \{c_0\}$. Since π_a is weakly smart, we must have $c_0 \in \pi_a(\mathcal{I})$. Let σ be a binding for the variables of π_a that produces this result c_0, i.e., $\sigma(x_1) = \sigma(x_{n+1}) = a$ (as a is the only entity linked to c_0). We notice that we must have $r_{n+1} = r$ as it is the only relation which leads to c_0. Let us define

$$m = (max \ \ \{m' \mid \exists l : \sigma(x_l) = c_{m'}\}) - 1$$

Let us call $r^-r_1...r_m$ the *forward path*, and $r_{m+1}...r_nr_{n+1}$ the *backward path* (with $r_{n+1} = r$). We have to show that the backward path is a walk in the forward path.

Let us show by induction that $r_{m+1}...r_j$ can be generated by $j-m$ steps (with $j \in [m+1, n+1]$), so that the updated position after the last step is i, and $\sigma(x_{j+1}) = c_i$. We first note that, due to the path shape of $\mathcal{I}$, $\sigma(x_j) = c_i$ for any i, j always implies that $\sigma(x_{j+1}) \in \{c_{i-1}, c_{i+1}\}$. Let us start with $j = m + 1$ and position $i = m + 1$. Since $\sigma(x_{j+1})$ cannot be c_{m+1} (by definition of m), we must have $\sigma(x_{j+1}) = c_m$. Since $\sigma(x_j) = c_{m+1}$ and $\sigma(x_{j+1}) = c_m$, we must have $r_{m+1} = r_m^-$. Thus, r_{m+1} was generated by a backward step, and we call $i - 1$ the updated position. Now let us assume that we have generated $r_{m+1}...r_j$ (with $j \in [m + 2, n + 1]$) by some forward and backward steps that have led to a position $i \in [0, m]$, and let us assume that $\sigma(x_{j+1}) = c_i$. Consider the case where $\sigma(x_{j+2}) = c_{i+1}$. Then we have $r_{j+1} = r_i$. Thus, we made a forward step, and the updated position is $i + 1$. Now consider the case where $\sigma(x_{j+2}) = c_{i-1}$. Then we have $r_{j+1} = r_{i-1}^-$. Thus, we made a backward step, and the updated position is $i - 1$. Since we have $r_{n+1} = r$ and $\sigma(x_{n+2}) = c_0$, the walk must end at position 0. The claim follows when we reach $j = n + 1$. □

We have thus found a way to recognise weakly smart plans without executing them. Extending this characterisation from weakly smart plans to fully smart plans consists mainly of adding a filter. Section 6.5.5 shows how the adaptation is done in practice and we now give more technical details.

6.4.5 Characterising Smart Plans

When we want to show that a plan is smart, we must first show that its constraint-free version is weakly smart. Thus, the results of Section 6.4.4 can be applied to determine the shape of a smart plan. What remains to find are the constrains (or filters) that we must apply to the execution plan in order to make it smart.

As in Chapter 5, we are going to exploit the notion of well-filtering plan (Definition 5.4.5 and minimal well-filtering plan (Definition 5.6.1). We have a similar result as in Chapter 5:

Lemma 6.4.14. *Given an atomic query $q(a, x) \leftarrow r(a, x)$ and a set of path functions $\mathcal{F}$, any smart plan constructed from sub-functions of $\mathcal{F}$ must be well-filtering.*

Proof. First, we prove that there cannot be a filter on a constant b different from the input constant a. Indeed, let us consider a smart plan π_a and its constraint free version π'_a. The semantics of π'_a is $r_1(a, x_1)...r_n(x_{n-1}, x_n)$. We define the database $\mathcal{I}$ as:

$$\mathcal{I} = r(a, c_0), r_1(a, c_1)...r_n(c_{n-1}, c_n)$$

where $c_1, ..., c_n$ are fresh constants different from a and b. On $\mathcal{I}$, we have $q(\mathcal{I}) \neq \emptyset$ and $\pi'_a(\mathcal{I}) \neq \emptyset$. So, we must have $\pi_a(\mathcal{I}) = q(\mathcal{I})$. However, if π_a contained a filter using a constant b, we would have $\pi_a(\mathcal{I}) = \emptyset$ (as b is not in $\mathcal{I}$). This is impossible.

Now, we want to show that the semantics of $\pi_a(x)$ contains at least an atom $r(a, x)$ or $r^-(x, a)$. We still consider a smart plan π_a and its constraint free version π'_a. The semantics of π'_a is $r_1(a, x_1)...r_n(x_{n-1}, x_n)$. We define the database $\mathcal{I}$ as:

$$\begin{aligned}\mathcal{I} =& r(a, c_0), r_1(a, c_1), r_1(c_1, c_1), ..., r_n(c_1, c_1), r(c_1, c_1),\\ & r_1(c_1, c_1), ..., r_n(c_1, c_2), r(c_1, c_2),\\ & r_1(c_2, c_2), ..., r_n(c_2, c_2), r(c_2, c_2), r_1(c_0, c_1), ..., r_n(c_0, c_1), r(c_0, c_1),\\ & r_1(c_0, c_2), ..., r_n(c_0, c_2), r(c_0, c_2)\end{aligned}$$

Let us write the semantics of $\pi_a(x)$ as $r_1(a, y_1)...r_n(y_{n-1}, y_n)$ where each y_i is either an existential variable, the output variable x or the constant a. We call $\mathcal{B}(y_i)$ the set of possible bindings for y_i. We notice that for all $i \in [1, n]$, y_i is a filter to the constant a or $\mathcal{B}(y_i) \neq \{a\}$. This can be seen by considering all transitions for all possible sets of bindings. We consider the atoms containing the output variable x. They are either one or two such atoms. If there is one, it is $r_k(y_{k-1}, x)$ for some $k \in [1, n]$. If y_{k-1} is not a filter to a, as we know $\mathcal{B}(y_k) \neq \{a\}$, we have that $\{c_1, c_2\} \subseteq \{x\}$ due to the structure of $\mathcal{I}$. This is a contradiction so y_{k-1} must be a filter to a and $r_k = r$. The same reasoning applies when there are two atoms containing x, except that we have one of them which is either $r(a, x)$ or $r^-(x, a)$. □

We have the equivalent of Property 6.4.6 for minimal filtering plans:

Property 6.4.15. *Let $q(x) \leftarrow r(a, x)$ be an atomic query. Let π_a be a minimal filtering execution plan associated to q. Then the semantics of the sub-function transformation $\mathcal{P}(\pi_a)$ is a path query where the last atom is either $r(a, x)$ or $r^-(x, a)$.*

Proof. Let us decompose π_a into a sequence of calls $c_0, ..., c_n$. The only call that contains a filter must be the last one (as x only appears in one call). So, we require only one output for the calls $c_0, ..., c_{n-1}$ and thus the semantics of there calls is a path query. The last function call is a path function, so the semantics of π_a is a path query. We need to show the last atom is either $r(a, x)$ or $r^-(x, a)$. By definition of a minimal filtering plan, the semantics of the plan must contain either $r(a, x)$ or $r(x, a)$ and at most one filter. If the filter is before x, then the last atom is $r(a, x)$ and the sub-function can stop there as there are no other filter nor output variable. If the filter is after x, we must have $r^-(x, a)$ as the last atom and the sub-function can also stop there. □

We also have the equivalent of Theorem 5.6.2 in Chapter 5.

Theorem 6.4.16. *Given a query $q(x) \leftarrow r(a, x)$, a well-filtering plan π_a and the associated minimal filtering plan π_a^{min}:*

- *If π_a^{min} is not smart, then neither is π_a.*
- *If π_a^{min} is smart, then we can determine in polynomial time if π_a is also smart.*

Proof. Let us write π'_a the constraint free version of π_a which is also the one of π_a^{min}. We first prove the first point. If π_a^{min} is not smart, it means that there exists a database $\mathcal{I}$ such that $q(\mathcal{I}) \neq \emptyset$ and $\pi'_a(\mathcal{I}) \neq \emptyset$ but $\pi_a^{min} \neq q(\mathcal{I})$. As $r(a, x)$ (or $r^-(x, a)$) is in π_a and π_a^{min} and π_a might contain more filters than π_a^{min}, we have $\pi_a(\mathcal{I}) \subseteq \pi_a^{min}(\mathcal{I}) \subset q(\mathcal{I})$ and thus π_a can be smart.

For the second point, we consider that π'_a has the form described in Section 6.4.4, after the transformation of Property 6.4.15: $F.B$ where F is a forward path and B is a walk through F. Then the correct filters at filter on a variable which is at position 1 during the backward walk. To see that, we can consider the database $\mathcal{I}$:

$$\mathcal{I} = r(a, c_0), r_1(a, c_1)...r_n(c_{n-1}, c_n)$$

where the query was $q(x) \leftarrow r(a, x)$ and the semantics of the execution plan was $r_1(a, x_1) \dots r_n(x_{n-1}, x_n)$. If we follow a proof similar to the one in Section 6.4.4, we find that we can keep the result only iff we filter at position 1 during the backward walk. □

This theorem tells us that it is enough to find minimal filtering smart plans.

Given a plan π_a, let us consider that we construct the plan π'_a as described in Property 6.4.15 by using only sub-function calls associated with the output variables. If π'_a is smart, it means that its constraint-free version is weakly smart and thus we follow the description in Section 6.4.4: a forward path F followed by a walk in r^-F ending at position 0. Then, from this construction, one can deduce the minimal-filtering smart plans. First, we create plans ending by $r(a, x)$ by adding a filter on the last atom of π'_a. Second, we create plans ending by $r^-(x, a)$ by adding a new atom $r^-(x, a)$ to the semantics of the plan.

Thus, if we have an algorithm to find weakly smart plans, it can easily be extended to generate smart plans as we will see in Section 6.5.5.

6.4.6 Characterising Weakly Sub-Smart Plans

In this section, we are interested in characterising weakly sub-smart plans. We already know that all weakly smart plans are also weakly sub-smart, so Section 6.4.4 gives a partial characterisation. We will see that we can complete it by introducing *loosely bounded plans*. These new plans look like reversed bounded plans: We first cross the query, and then perform a forward path followed by a walk. The idea is that some results might be filtered out due to the forward path, but, in the end, if we get a result, some answers had to survive. Indeed, when we reach the end of the forward path, we obtain entities that are linked to the input though the forward path and the query. However, these linking path might not contain all the answers

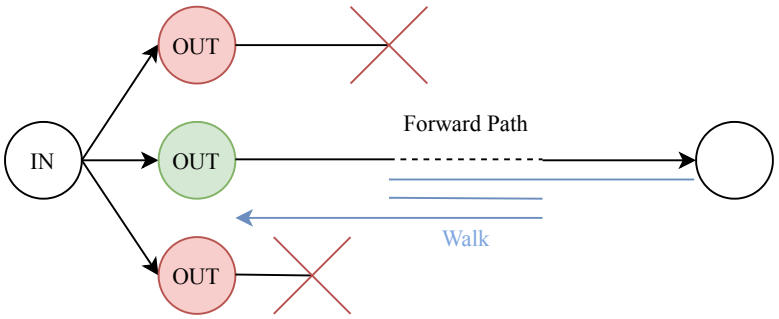

Figure 6.5: The forward path may filter solutions

to the query. In Figure 6.5, we illustrate such a situation where some results are lost because the forward path filters out solutions.

Definition 6.4.17 (Loosely Bounded Plan)**.** *A loosely bounded path for a set of relations $\mathcal{R}$ and a query $q(x) \leftarrow r(a, x)$ is either a bounded path or a path query of the form $r.P.B$, where $r.P$ is a path query, and B is a walk through $r^-.P$ to the position 1.*

A loosely bounded plan for a set of functions $\mathcal{F}$ is an non-redundant execution plan whose consequences are a loosely bounded path.

The difference to bounded plans (Definition 6.4.10) is thus that the query can also appear at the beginning of the path. As we did for smart plans, we are now going to show the correctness and the completeness of the loosely bounded plans.

Theorem 6.4.18 (Correctness)**.** *Let $q(x) \leftarrow r(a, x)$ be an atomic query, F a set of path functions and F_{sub} the set of sub-functions of F. Let π_a be a non-redundant loosely bounded execution plan over the F_{sub}. Then π_a is weakly sub-smart.*

Proof. Let us first prove that every loosely bounded plan is weakly sub-smart. If the loosely bounded plan is a bounded plan (Definition 6.4.10), then it is a weakly smart plan (Theorem 6.4.12). Hence, it is also a weakly sub-smart plan. Let us now consider the plans of the form $\pi_a = r.P.B$, where $r.P$ is a path query, and B is a walk through $r^-.P$ to the position 1. Take any database $\mathcal{I}$ such that $q(I) \neq \emptyset$ and $\pi_a(I) \neq \emptyset$. Let $\mathcal{C} = q(I)$. Let use write $\pi_a(x) = r.P.B(a, x) = r(a, x_1).r_1(x_1, x_2)...r_n(x_n, x_{n+1}).B(x_{n+1}, x)$. Let σ be a binding of the variables of π_a in $\mathcal{I}$. We know such binding exists as $\pi_a(I) \neq \emptyset$. Let $c = \sigma(x_1)$. We know that $c \in \mathcal{C}$ as the first relation is r. Using the same arguments as in Theorem 6.4.12, we can show that $c \in P.B(c)$. Thus, $c \in r.P.B(a)$ and so the plan is weakly sub-smart. □

Theorem 6.4.19 (Completeness)**.** *Let $q(x) \leftarrow r(a, x)$ be an atomic query, F a set of path functions and F_{sub} the set of sub-functions of F. Let π_a be a weakly sub-smart plan over the F_{sub}. Then π_a is loosely bounded.*

Proof. Let π_a be a weakly sub-smart plan for a query $q(x) \leftarrow r(a, x)$, with consequences $r_1(x_1, x_2)...\ r_n(x_n, x_{n+1})$. For convenience, we write $a = c_1$. Consider the database $\mathcal{I}$:

$$\mathcal{I} = \{r^-(c_0, c_1), r_1(c_1, c_2), ..., r_n(c_n, c_{n+1})\}$$

Here, the c_i are fresh constants. Let us first assume that $r_1 \neq r$. Then, $\pi_a(\mathcal{I}) \supseteq \{c_{n+1}\} \neq \emptyset$ and $q(\mathcal{I}) = \{c_0\}$. Since π_a is weakly sub-smart, we must have $c_0 \in \pi_a(\mathcal{I})$. Using the same argument as in Theorem 6.4.13, π_a must be a forward path P, followed by a walk B in r^-P. Hence, π_a is a bounded plan.

Let us now suppose that $r_1 = r$. We now have $q(\mathcal{I}) = \{c_0, c_2\}$. Since π_a is weakly sub-smart, we must have $c_0 \in \pi_a(\mathcal{I})$ or $c_2 \in \pi_a(\mathcal{I})$ (or both). Let us consider the first case. Using the same argument as before, π_a must consist of a forward path P followed by a walk B in $r^-.P$. That is, π_a is a bounded plan. In that case, $\pi_a(\mathcal{I}) = \{c_0, c_2\}$, and π_a is actually a weakly smart plan.

Now consider the case where $c_2 \in \pi_a(\mathcal{I})$ and $c_0 \notin \pi_a(\mathcal{I})$. We can then proceed the same way we did in Theorem 6.4.13. We take the same notations, with a binding such that $\sigma(x_{n+2}) = c_2$. We call $r_1...r_m = r.r_2...r_m$ the *forward path*. The induction then proceeds the same way. The only difference is that we have to stop at position 1 as we must end on c_2. □

We see here that weakly smart plans and weakly sub-smart plan are different. Let us consider the following example to better understand why it is the case.

Example 6.4.20. *We continue with the music example. We have access to two path functions: getAlbumsOfSinger(singer, album) = sing(singer, song), onAlbum(song, album) which gets all the albums on which a singer sings and getSongsOnAlbum(album, song) = onAlbum$^-$(album, song) which gives us the songs on an album. Our query is $q_{Pomme}(x)$ = sing(Pomme, x). The plan $\pi_{Pomme}(x)$ = getAlbumsOfSinger(Pomme, album), getSongsOnAlbum(album, x) is weakly sub-smart but not weakly smart. Indeed, if we consider the database $\mathcal{I}$ = {sing(Pomme, Pauline), sing(Pomme, Itsumo Nando Demo), onAlbum(Pauline, À peu près)}, then the plans return Pauline but not Itsumo Nando Demo, which was never published on an album.*

6.4.7 Characterising Sub-Smart Plans

We proceed like in Section 6.4.5: to show that a plan is sub-smart, we have to consider its constraint-free version and show it is weakly sub-smart. Then, again, what remains is to add the appropriate filters. As in Section 6.4.5 we have the lemma:

Lemma 6.4.21. *Given an atomic query $q(a, x) \leftarrow r(a, x)$ and a set of path functions $\mathcal{F}$, any sub-smart plan constructed from sub-functions of $\mathcal{F}$ must be well-filtering.*

and the theorem:

Theorem 6.4.22. *Given a query $q(x) \leftarrow r(a, x)$, a well-filtering plan π_a and the associated minimal filtering plan π_a^{min}:*

- *If π_a^{min} is not sub-smart, then neither is π_a.*
- *If π_a^{min} is sub-smart, then we can determine in polynomial time if π_a is also sub-smart.*

The proofs are similar to the ones in Section 6.4.5.

Now, let us construct sub-smart plans. We start by generating weakly sub-smart plans using Theorem 6.4.19: we know the sub-function transformation of a constraint-free weakly sub-smart plan is loosely bounded. The case where the plan is bounded was treated in Section 6.4.5: It is when we have a smart plan (which is also sub-smart). Let us now consider the case where the semantics of the plan is a path query of the form $r.P.B$, where $r.P$ is a path query, and B is a walk through $r^-.P$ to the position 1. Once we have this plan, the only way to create a minimal well-filtering sub-smart plan is to append an atom $r^-(x, a)$ to the semantics. This way, the filter is at the position 0 in the walk.

In the cases where the semantics of the plan ends with $r(a, x)$, we have a smart plan. Indeed, such a situation happens when we finish the walk by going from a position 0 to a position 1. We obtain a skeleton $r.F.B$ where B is a walk through $r^-.F$ to the position 1. In the case we end by going from a position 0 to 1, it is similar to have a forward path $F' = r.F$ and a backward path $B' = B$ which is a walk through r^-F' to a position 0. In such a case, we have a smart plan which we already know how to characterise.

Thus, if we have an algorithm to generate weakly sub-smart plans, we can adapt it to create sub-smart plans.

6.5 Generating Smart Plans

We have shown that weakly smart plans are precisely the bounded plans. We will now turn to generating such plans. We will later show how to adapt these plans to generate not just weakly smart plans, but also smart plans. Let us first introduce the notion of minimal plans.

6.5.1 Minimal Smart Plans

In line with Chapter 5, we will not generate redundant plans. These contain more function calls, and cannot deliver more results than non-redundant plans. More precisely, we will focus on *minimal plans*:

Definition 6.5.1 (Minimal Smart Plan)**.** *Let $\pi_a(x)$ be a non-redundant execution plan organised in a sequence $c_0, c_1, \ldots, c_k$ of calls, such that the input of c_0 is the constant a, every other call c_i takes as input an output variable of the previous call c_{i-1}, and the output of the plan is in the call c_k. π_a is a* minimal (weakly) smart plan *if it is a (weakly) smart plan and there exists no other (weakly) smart plan $\pi'_a(x)$ composed of a sub-sequence $c_{i_1}, ..., c_{i_n}$ (with $0 \leq i_1 < ... < i_n \leq k$).*

Example 6.5.2. *Let us consider the two functions $f_1(x, y) = r(x, y)$ and $f_2(y, z) = r^-(y, t).r(t, z)$. For the query $q(x) \leftarrow r(a, x)$, the plan $\pi_a(x) = f_1(a, y), f_2(y, x)$ is obviously weakly smart. It is also non-redundant. However, it is not minimal. This is because $\pi'_a(x) = f_1(a, x)$ is also weakly smart, and is composed of a sub-sequence of calls of π_a.*

In general, it is not useful to consider non-minimal plans because they are just longer but cannot yield more results. On the contrary, a non-minimal plan can have fewer results than its minimal version, because the additional calls can filter out results. The notion of minimality would make sense also in the case of equivalent rewritings. However, in that case, the notion would impact just the number of function calls and not the results of the plan, since equivalent rewritings deliver the same results by definition. In the case of smart plans, as we will see, the notion of minimality allows us to consider only a finite number of execution plans and thus to have an algorithm that terminates. The following property confirms the intuition that if we have a non-minimal execution plan, then we can turn it into a minimal one:

Property 6.5.3. *Let q be an atomic query and π_a a non-redundant weakly smart plan. Then, either π_a is minimal, or one can extract a minimal weakly smart plan from π_a.*

This means that even if we can generate only the minimal weakly smart plans (and not all weakly smart plans), we will be able to generate a plan if a weakly smart plan exists at all.

6.5.2 Susie

Our first approach to generate bounded plans is inspired by Susie [98]. We call the resulting plans *Susie plans*.

Definition 6.5.4 (Susie Plan)**.** *A Susie plan for a query $q(x) \leftarrow r(a, x)$ is a plan whose semantics is of the form $F.F^-.r$, where F is a path query, F^- is the reverse of F (with all relations inverted), $F^-.r$ is generated by a single function call and the last atom is $r(a, x)$.*

Example 6.2.2 (shown in Figure 6.1) is a simple Susie plan. We have:

Theorem 6.5.5 (Correctness)**.** *Every Susie plan is a smart plan.*

Proof. Given a Susie plan of the form $\pi = F.F^-.r$, F^- is a walk through F. Hence, $F^-.r$ is a walk through $r^-.F$. Hence, the constraint-free version of π is a bounded plan (Definition 6.4.10). Then, Theorem 6.4.12 tells us that π is a weakly smart plan. In addition, it ends with $r(a, x)$, so it is smart. □

Theorem 6.5.6 (Limited Completeness)**.** *Given a set of path functions so that (1) no function contains existential variables and (2) no function contains a loop (i.e., two consecutive relations of the form $r.r^-$), the Susie plans include all minimal filtering smart plans.*

Proof. We first note that the plan cannot end by $r^-(x, a)$, because then it would mean that the two last atoms would be $r(y, x).r^-(x, a)$. As no function can contain two consecutive atoms $r.r^-$, the last function call is $r^-(x, a)$ and it is useless as we could have created the plan that ends with $r(a, x)$ by putting the filter on the previous atom. Now consider any minimal smart plan $P = f_1 ... f_n$ with forward path F and backward path B for a query q. The last function call (f_n) will have

the form $r_1...r_mq$. Thus, F must start with $r_m^-...r_1^-$ as f_n does not contain loops. Consider the first functions $f_1...f_k$ of P that cover $r_m^-...r_1^-$. f_k must be of the form $r_i^-...r_1^-r_1'...r_l'$ for some $i \in [1,m]$ and some $l \geq 0$. Assume that $l > 0$. Since there are no existential variables, we also have a function f_k' of the form $r_i^-...r_1^-$. Then, the plan $f_1...f_{k-1}f_k'f_n$ is smart. Hence, P was not minimal. Therefore, let us assume that $l = 0$. Then, P takes the form $P = f_1...f_kf_{k+1}...f_{n-1}f_n$. If $k < n-1$, then we can remove $f_{k+1}...f_{n-1}$ from the plan, and still obtain a smart plan. Hence, P was not minimal. Therefore, P must have the form $P = f_1...f_kf_n$. Since $f_1...f_n$ has the form $r_m^-...r_1^-$, and f_n has the form $r_1...r_mq$, P is a Susie plan. □

Intuitively, Theorem 6.5.6 means that Susie plans are sufficient for all cases where the functions do not contain existential variables. If there are existential variables, then there can be minimal smart plans (as introduced in Chapter 5) that Susie will not find. An example is shown in Figure 6.3, if we assume that the middle variable of *getProfessionalAddress* existential.

The Susie plans for a query q on a set of functions $\mathcal{F}$ can be generated very easily: It suffices to consider each function $f = r_1....r_n$ that ends in $r_n = q$, and to consider all path function $f_1...f_m$ that have the semantics $r_{n-1}^-...r_1^-$. This algorithm runs in $O(|\mathcal{F}|^{k+1})$, where k is the maximal length of a function, and is typically small.

6.5.3 Bounding the Weakly Smart Plans

We would like to generate not just the Susie plans, but all minimal smart plans. This is possible only if their number is finite. This is indeed the case, as we shall see next: Consider a weakly smart plan for a query q. Theorem 6.4.13 tells us that the consequences of the plan are of the form $r_0...r_k...r_nq$, where $r_0...r_k$ is the forward path, and $r_{k+1}...r_n$ is a walk in the forward path. Take some function call $f = r_i...r_j$ of the plan. If $j \leq k$, we say that f *starts* at position i, that it *ends* at position j, and that it *crosses* positions i to j. For example, in Figure 6.4, the forward path is $F = rst$. The function f_1 starts at position 0, ends at position 3, and crosses the positions 0 to 3. If $i \geq k$, then r_i was generated by a step in the backward path. We say that f starts at the position before that step, and ends at the updated position after the step that produced r_j. In Figure 6.4, f_2 starts at position 3, ends at position 1, and crosses 3 to 1; f_3 starts at position 1, ends at position 2, and crosses 1 to 2; f_4 starts at 2, ends at -1, and crosses 2 to -1. Our main insight is the following:

Theorem 6.5.7 (No Duplicate Ends)**.** *Given a set of relations $\mathcal{R}$, a query $q(x) \leftarrow r(a,x), r \in \mathcal{R}$, and a set of path function definitions $\mathcal{F}$, let π be a minimal weakly smart plan for q. There can be no two function calls in π that end at the same position, and there can be no two function calls that start at the same position.*

Proof. Assume that there is a weakly smart plan $P = f_1...f_ng_1...g_mh_1...h_k$ such that f_n and g_m end at the same position. Then $f_1...f_nh_1...h_k$ is also a weakly smart plan. Hence, P is not minimal. Now assume that there is a weak smart plan $P = f_1...f_ng_1...g_mh_1...h_k$ such that g_1 and h_1 start at the same position. Then again, $f_1...f_nh_1...h_k$ is also a weakly smart plan and hence P is not minimal. □

This theorem is easy to understand: If two functions were ending or starting at the same position, the plan would not be minimal as we could remove the function calls between them. To turn this theorem into a bound on the number of possible plans, we need some terminology first. Let us consider the positions on the forward path one by one. Each position is crossed by several functions. Let us call the triple of a function $f = r_1...r_n$, a position $i \in [1, n]$, and a direction (forward or backward) a *positioned function*. For example, in Figure 6.4, we can say that position 2 in the forward path is crossed by the positioned function $\langle f_1, 3, \mathit{forward}\rangle$. Let us call the set of positioned functions at a given position in the forward path a *state*. For example, in Figure 6.4 at position 2, we have the state $\{\langle f_1, 3, \mathit{forward}\rangle, \langle f_2, 2, \mathit{backward}\rangle, \langle f_3, 2, \mathit{forward}\rangle, \langle f_4, 1, \mathit{backward}\rangle\}$.

We first observe that a state cannot contain the same positioned function more than once. If it contained the same positioned function twice, then the plan would not be minimal. Furthermore, Theorem 6.5.7 tell us that there can be no two functions that both end or both start in a state. Finally, a plan cannot contain the same state twice, because otherwise, it would not be minimal. This leads to the following bound on the number of plans:

Theorem 6.5.8 (Bound on Plans)**.** *Given a set of relations $\mathcal{R}$, a query $q(x) \leftarrow r(a, x), r \in \mathcal{R}$, and a set of path function definitions $\mathcal{F}$, there can be no more than $M!$ minimal weakly smart plans, where $M = |\mathcal{F}|^{2k}$ and k is the maximal number of atoms in a function.*

Proof. Theorem 6.5.7 tells us that there can be no two positions in the forward path where two function calls end. It means that, at each position, there can be no more than $2 \times k$ crossing function calls. Therefore, there can be only $M = |\mathcal{F}|^{2k}$ different states overall. No minimal plan can contain the same state twice (because otherwise, it would be redundant). Hence, there can be at most $M!$ minimal weak smart plans. Indeed, in the worst case, all plans are of length M, because if a plan of length $< M$ is weakly smart, then all plans that contain it are redundant. □

This bound is very pessimistic: In practice, the succession of states in very constrained and thus, the complete exploration is quite fast, as we will show in Section 6.6.

6.5.4 Generating the Weakly Smart Plans

Theorem 6.5.8 allows us to devise an algorithm that enumerates all minimal weakly smart plans. For simplicity, let us first assume that no function definition contains a loop, i.e., no function contains two consecutive relations of the form rr^-. This means that a function cannot be both on a forward and backward direction. We will later see how to remove this assumption. Algorithm 1 takes as input a query q and a set of function definitions $\mathcal{F}$. It first checks whether the query can be answered trivially by a single function (Line 1). If that is the case, the plan is printed (Line 2). Then, the algorithm sets out to find more complex plans. To avoid exploring states twice, it keeps a history of the explored states in a stack H (Line 3). The algorithm finds all non-trivial functions f that could be used to answer q. These are the functions whose short notation ends in q (Line 4). For each of these functions, the algorithm

considers all possible functions f' that could start the plan (Line 5). For this, f' has to be *consistent* with f, i.e., the functions have to share the same relations. The pair of f and f' constitute the first state of the plan. Our algorithm then starts a depth-first search from that first state (Line 6). For this purpose, it calls the function *search* with the current state, the state history, and the set of functions. In the current state, a marker (a star) designates the forward path function.

Algorithm 1: FindMinimalWeakSmartPlans

Data: Query $q(a) \leftarrow r(a, x)$, set of path function definitions and all their sub-functions $\mathcal{F}$
Result: Prints minimal weakly smart plans

```
1 if ∃f = r ∈ F then
2     print(f)
3 H ← Stack()
4 foreach f = r_1...r_n.r ∈ F do
5     foreach f' ∈ F consistent with r_n^-...r_1^- do
6         search({⟨f, n, backward⟩, ⟨f', 1, forward⟩*}, H, F)
```

Algorithm 2 executes a depth-first search on the space of states. It first checks whether the current state has already been explored (Line 1). If that is the case, the method just returns. Otherwise, the algorithm creates the new state S' (Line 3). For this purpose, it considers all positioned functions in the forward direction (Lines 5-7). If any of these functions ends, the end counter is increased (Line 6). Otherwise, we advance the positioned function by one position. The $(*)$ means that if the positioned function happens to be the designated forward path function, then the advanced positioned function has to be marked as such, too. We then apply the procedure to the backwards-pointing functions (Lines 8-11).

Once that is done, there are several cases: If all functions ended, we have a plan (Line 12). In that case, we can stop exploring because no minimal plan can include an existing plan. Next, the algorithm considers the case where one function ended, and one function started (Line 13). If the function that ended were the designated forward path function, then we would have to add one more forward function. However, then the plan would contain two functions that start at the current state. Since this is not permitted, we just do not do anything (Line 14), and the execution jumps to Line 29. If the function that ended was some other function, then the ending and the starting function can form part of a valid plan. No other function can start or end at the current state, and hence we just move to the next state (Line 15).

Next, the algorithm considers the case where one function starts and no function ends (Line 16). In that case, it has to add another backward function. It tries out all functions (Line 17-19) and checks whether adding the function to the current state is consistent (as in Algorithm 1). If that is the case, the algorithm calls itself recursively with the new state (Line 19). Lines 20-23 do the same for a function that ended. Here again, the $(*)$ means that if f was the designated forward path function, then the new function has to be marked as such. Finally, the algorithm

considers the case where no function ended, and no function started (Line 24). In that case, we can just move on to the next state (Line 25). We can also add a pair of a starting function and an ending function. Lines 26-28 try out all possible combinations of a starting function and an ending function and call the method recursively. If none of the previous cases applies, then $end > 1$ and $start > 1$. This means that the current plan cannot be minimal. In that case, the method pops the current state from the stack (Line 29) and returns.

Algorithm 2: Search

Data: A state S with a designated forward path function, a set of states $\mathcal{H}$, a set of path functions $\mathcal{F}$
Result: Prints minimal weakly smart plans

1 **if** $S \in H$ **then** return
2 $H.push(S)$
3 $S' \leftarrow \emptyset$
4 $end \leftarrow 0$
5 **foreach** $\langle r_1...r_n, i, forward\rangle \in S$ **do**
6 **if** $i+1 > n$ **then** $end++$
7 **else** $S' \leftarrow S' \cup \{\langle r_1...r_n, i+1, forward\rangle^{(*)}\}$
8 $start \leftarrow 0$
9 **foreach** $\langle r_1...r_n, i, backward\rangle \in S$ **do**
10 **if** $i = 1$ **then** $start++$
11 **else** $S' \leftarrow S' \cup \{\langle r_1...r_n, i-1, backward\rangle\}$
12 **if** $S' = \emptyset$ **then** print(H)
13 **else if** $start = 1 \wedge end = 1$ **then**
14 **if** the designated function ended **then** pass
15 **else** search$(S', H, \mathcal{F})$
16 **else if** $start = 1 \wedge end = 0$ **then**
17 **foreach** $f \in \mathcal{F}$ **do**
18 $S'' \leftarrow S' \cup \{\langle f, |f|, backward\rangle\}$
19 **if** S'' is consistent **then** search$(S'', H, \mathcal{F})$
20 **else if** $start = 0 \wedge end = 1$ **then**
21 **foreach** $f \in \mathcal{F}$ **do**
22 $S'' \leftarrow S' \cup \{\langle f, 1, forward\rangle^{(*)}\}$
23 **if** S'' is consistent **then** search$(S'', H, \mathcal{F})$
24 **else if** $start = 0 \wedge end = 0$ **then**
25 search$(S', H, \mathcal{F})$
26 **foreach** $f, f' \in \mathcal{F}$ **do**
27 $S'' \leftarrow S' \cup \{\langle f, 1, forward\rangle, \langle f', |f'|, backward\rangle\}$
28 **if** S'' is consistent **then** search$(S'', H, \mathcal{F})$
29 $H.pop()$

Theorem 6.5.9 (Algorithm)**.** *Algorithm 1 is correct and complete, terminates on all inputs, and runs in time $\mathcal{O}(M!)$, where $M = |\mathcal{F}|^{2k}$ and k is the maximal number of atoms in a function.*

Proof. Algorithm 1 just calls Algorithm 2 on all possible starting states. Let us, therefore, concentrate on Algorithm 2. This algorithm always maintains one positioned function as the designated forward path function. These designated functions are chained one after the other, and they all point in the forward direction. No other function call can go to a position that is greater than the positions covered by the designated functions (Line 14). Hence, the designated functions form a forward

path. All other functions perform a walk in the forward path. Hence, all generated plans are bounded. The plans are also valid execution plans because whenever a function ends in one position, another function starts there (Lines 13, 16, 20). Hence, the algorithm generates only bounded plans, i.e., only weak smart plans.

At any state, the algorithm considers different cases of functions ending and functions starting (Lines 13, 16, 20). There cannot be any other cases, because, in a minimal weak smart plan, there can be at most one single function that starts and at most one single function that ends in any state. In all of the considered cases, all possible continuations of the plan are enumerated (Lines 17, 21, 26). There cannot be continuations of the plan with more than one new function, because minimal weak smart plans cannot have two functions starting at the same state. Hence, the algorithm enumerates all minimal weak smart plans.

We have already seen that the number of possible states is bounded (Theorem 6.5.8). Since the algorithm keeps track of all states that it encounters (Line 2), and since it never considers the same state twice (Line 1), it has to terminate. Furthermore, since M bounds the size of H, the algorithm runs in time $\mathcal{O}(M!)$. □

The worst-case runtime of $\mathcal{O}(M!)$ is unlikely to appear in practice. Indeed, the number of possible functions that we can append to the current state in Lines 19, 23, 28 is severely reduced by the constraint that they must coincide on their relations with the functions that are already in the state. In practice, very few functions have this property. Furthermore, we can significantly improve the bound if we are interested in finding only a single weakly smart plan:

Theorem 6.5.10. *Given an atomic query and a set of path function definitions $\mathcal{F}$, we can find a single weakly smart plan in $\mathcal{O}(|\mathcal{F}|^{2k})$, where k is the maximal number of atoms in a function.*

Proof. If we are interested in finding only a single plan, we replace H by a set in Line 3 of Algorithm 1. We also remove the pop operation in Line 29 of Algorithm 2. In this way, the algorithm will never explore a state twice. Since no minimal weak smart plan contains the same state twice, the algorithm will still find a minimal weak smart plan, if it exists (it will just not find two plans that share a state). Since there are only $|\mathcal{F}|^{2k}$ states overall (Theorem 6.5.8), the algorithm requires no more than $O(|\mathcal{F}|^{2k})$ steps. □

Functions with loops. If there is a function that contains a loop of the form $r.r^-$, then Algorithm 2 has to be adapted as follows: First, when neither functions are starting nor ending (Lines 24-28), we can also add a function that contains a loop. Let $f = r_1...r_i r_i^- ...r_n$ be such a function. Then the first part $r_1...r_i$ becomes the backward path, and the second part $r_i^- ...r_n$ becomes the forward path in Line 27.

When a function ends (Lines 20-23), we could also add a function with a loop. Let $f = r_1...r_i r_i^- r_n$ be such a function. The first part $r_1...r_i$ will create a forward state $\langle r_1...r_i, 1, \mathit{forward}\rangle$. The second part, $r_i^- ...r_n$ will create the backward state $\langle r_i^- ...r_n, |r_1...r_i|, \mathit{backward}\rangle$. The consistency check has to be adapted accordingly. The case when a function starts (Lines 16-19) is handled analogously. Theorems 6.5.9 and 6.5.10 remain valid, because the overall number of states is still bounded as before.

6.5.5 Generating Smart Plans

To generate smart plans instead of weakly smart plans, we will adapt Algorithm 1. The intuition is simple: if the query is $q(x) \leftarrow r(a, x)$, and if we found a weakly smart plan that contains the atom $r(y, x)$, then we have to add a filter $y = a$ to make the plan smart. For this, we have to make sure that y is an output variable. In Section 6.4.5, we show why this is sufficient to make a weakly smart plan smart. Besides, we show that we can restrict ourselves to smart plans whose semantics are path queries ending in $r(y, x)$ or $r^-(x, y)$. We now give more details about how we adapt the generation of weakly smart plans to generate also smart plans.

We begin by considering the case where the semantics of the plan is a path query ending in $r(y, x), y = a$. We have to modify Algorithm 1 in such a way that the last two variables of the sub-function that starts the search must be output variables. If the algorithm gives no result, we can conclude that no smart plan exists (because there exists no weakly smart plan). If the algorithm gives a result, then we know that we can add the filter $y = a$.

Now let us consider the case where the semantics of the plan is a path query ending with $r^-(x, y), y = a$. In this case, the restriction to sub-functions will remove the atom $r^-(x, a)$. Therefore, we have to adapt the initialisation of the algorithm in Section 6.5.4. We have two cases to consider:

- The last function call contains only one atom: $r^-(x, a)$. This happens when there is a sub-function with one atom r^- with one input and one output variable. In this case, we just have to find a weakly smart plan as in Section 6.5.4 and add this function at the end.

- The last call contains more than one atom. In this case, we have to look at all sub-functions ending in $r(z, y).r^-(y, x)$ where y and x are output variables. We then continue the initialisation as if the function was ending in $r(z, y)$, ignoring the last atom $r^-(y, x)$.

If this generation algorithm gives no result, we know there is no smart plan. Otherwise, we apply the filter on the last atom of the semantics to get $r^-(x, y), y = a$. This is possible because we made sure that y is an output variable.

We give the complete new algorithm in Algorithm 3.

6.5.6 Generating Weakly Sub-Smart Plans

The generation of weakly sub-smart plans follows a similar direction as the creation of weakly smart plans. First, as we already mentioned, all weakly smart plans are also weakly sub-smart. Therefore, we can reuse the results of Section 6.5.4 and, in what follows, we are going to focus on generating weakly sub-smart plans that are not weakly smart.

We first notice that as a function does not contain two consecutive inverse atoms $r.r^-$, no function can be both on the forward and backward path. This means that, in the characterisation given in Section 6.4.6, when we do a walk through the forward path to a position 1, we never go to position 0. Indeed, in the case of minimal plans, having a function ending at position 0 would mean that all function calls before

Algorithm 3: FindSmartPlans

Data: Query $q(a) \leftarrow r(a, x)$, set of path function definitions and all their sub-functions $\mathcal{F}$
Result: Prints minimal smart plans

```
if ∃f(x, y) = r(x, y) ∈ F then
    print(f)
H ← Stack()
foreach f(x, y, z) = r_1...r_n(x, y).r(y, z) ∈ F do
    foreach f' ∈ F consistent with r_n^-...r_1^- do
        foreach Weak Smart Plan in
          search({⟨f, n, backward⟩, ⟨f', 1, forward⟩*}, H, F) do
            Add a filter to create r(a, x)
foreach f(x, y, z) = r_1...r_n.r(x, y).r^-(y, z) ∈ F do
    foreach f' ∈ F consistent with r_n^-...r_1^- do
        foreach Weak Smart Plan in
          search({⟨f, n, backward⟩, ⟨f', 1, forward⟩*}, H, F) do
            Add a filter to create r^-(x, a)
foreach f(x, y) = r^-(x, y) ∈ F do
    FindMinimalWeakSmartPlans(q, F) + f(x, a)
```

going back to position 0 were useless (as we have to start a new function), and so could have been removed.

This observation makes it easy to adapt our algorithm to discover weak sub-smart plans: We have to filter the first function on the way forward to start with the query relation and then continue as if the function started at the second atom. Algorithm 4 gives the modified algorithm.

6.5.7 Generating Sub-Smart Plans

Finally, let us generate sub-smart plans. As we did for the generation of smart plans, we adapt the initialisation of the algorithm to have the filters at the correct plan. From Section 6.4.7, we know that the filter that is specific to sub-smart plans and that is not in smart plans is an atom $r^-(x, a)$. As for smart plans, we filter the last function call such that it is possible to create such a filter. Algorithm 5 gives the algorithm to find the minimal sub-smart plans.

6.6 Experiments

We have implemented the Susie Algorithm (Section 6.5.2), the equivalent rewriting [108], as well as our method (Section 6.5.4) in Python. The code is available on Github (https://github.com/Aunsiels). We conduct two series of experiments – one on synthetic data, and one on real Web services. All our experiments are run on a

Algorithm 4: FindMinimalWeakSubSmartPlans

Data: Query $q(a) \leftarrow r(a, x)$, set of path function definitions and all their sub-functions $\mathcal{F}$
Result: Prints minimal weakly sub-smart plans

if $\exists f = r \in \mathcal{F}$ **then**
  print(f)
$H \leftarrow Stack()$
foreach $f = r_1...r_n.r \in \mathcal{F}$ **do**
  foreach $f' \in \mathcal{F}$ *consistent with* $r_n^-...r_1^-$ **do**
    search($\{\langle f, n, backward\rangle, \langle f', 1, forward\rangle^*\}$, H, $\mathcal{F}$)
foreach $f = r_1...r_n \in \mathcal{F}$ **do**
  foreach $f' = r.r'_1...r'_m \in \mathcal{F}$ *with* $r'_1 \dots r'_m$ *consistent with* $r_n^-...r_1^-$ **do**
    search($\{\langle f, n, backward\rangle, \langle f', 2, forward\rangle^*\}$, H, $\mathcal{F}$)

Algorithm 5: FindSubSmartPlans

Data: Query $q(a) \leftarrow r(a, x)$, set of path function definitions and all their sub-functions $\mathcal{F}$
Result: Prints minimal sub-smart plans

if $\exists f(x, y) = r(x, y) \in \mathcal{F}$ **then**
  print(f)
$H \leftarrow Stack()$
foreach $f(x, \dots, y, z) = r_1...r_n(x, y).r(y, z) \in \mathcal{F}$ **do**
  foreach $f' \in \mathcal{F}$ *consistent with* $r_n^-...r_1^-$ **do**
    foreach *Weak Smart Plan in* *search*($\{\langle f, n, backward\rangle, \langle f', 1, forward\rangle^*\}$, H, $\mathcal{F}$) **do**
      Add a filter to create $r(a, x)$
foreach $f(x, \dots, y, z) = r_1...r_n.r(x, y).r^-(y, z) \in \mathcal{F}$ **do**
  foreach $f' \in \mathcal{F}$ *consistent with* $r_n^-...r_1^-$ **do**
    foreach *Weak Smart Plan in* *search*($\{\langle f, n, backward\rangle, \langle f', 1, forward\rangle^*\}$, H, $\mathcal{F}$) **do**
      Add a filter to create $r^-(x, a)$
foreach $f(x, y) = r^-(x, y) \in \mathcal{F}$ **do**
  FindMinimalWeakSmartPlans(q, $\mathcal{F}$) + $f(x, a)$
foreach $f(x, \dots, t, z) = r_1...r_n(x, y)r^-(y, z) \in \mathcal{F}$ **do**
  foreach $f' = r.r'_1...r'_m \in \mathcal{F}$ *with* $r'_1 \dots r'_m$ *consistent with* $r_n^-...r_1^-$ **do**
    foreach *Weak Sub-Smart Plan in* *search*($\{\langle f, n, backward\rangle, \langle f', 2, forward\rangle^*\}$, H, $\mathcal{F}$) **do**
      Add a filter to create $r^-(x, a)$

laptop with Linux, 1 CPU with four cores at 2.5GHz, and 16 GB RAM.

6.6.1 Synthetic Functions

In our first set of experiments, we use the methodology introduced by Romero et al [108] to simulate random functions. We consider a set of artificial relations $\mathcal{R} = \{r_1, ..., r_n\}$, and randomly generated path functions up to length 3, where all variables are existential except the last one. Then we try to find a smart plan for each query of the form $q(x) \leftarrow r(a, x), r \in \mathcal{R}$.

In our first experiment, we limit the number of functions to 30 and vary the number n of relations. All the algorithms run in less than 2 seconds in each setting for each query. Figure 6.6 shows which percentage of the queries the algorithms answer. As expected, when increasing the number of relations, the percentage of answered queries decreases, as it becomes harder to combine functions. The difference between the curve for weakly smart plans and the curve for smart plans shows that it was not always possible to filter the results to get exactly the answer of the query. (Weakly smart plans can answer more queries but at the expense of delivering only a superset of the query answers.) In general, we observe that our approach can always answer strictly more queries than Susie and the equivalent rewriting approach.

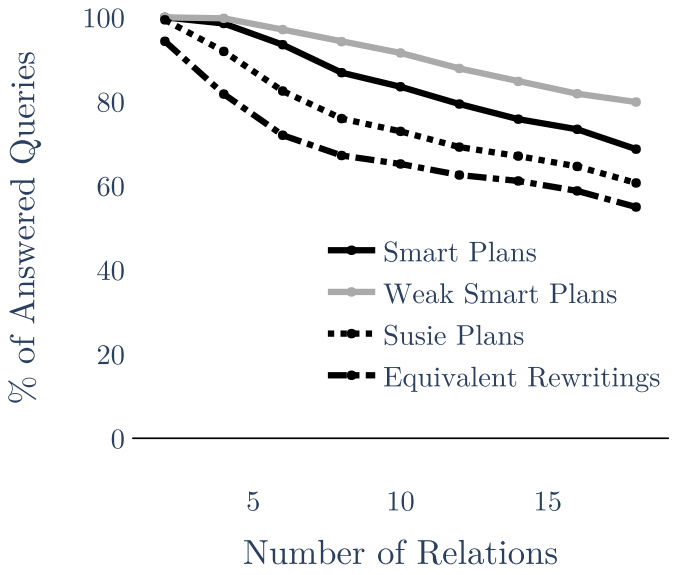

Figure 6.6: Percentage of answered queries

In our next experiment, we fix the number of relations to 10 and vary the number of functions. Figure 6.7 shows the results. As we increase the number of functions, we increase the number of possible function combinations. Therefore, the percentage of answered queries increases for all approaches. As before, our algorithm outperforms the other methods by a wide margin. The reason is that Susie cannot find all smart plans (see Section 6.5.2 again). Equivalent rewritings, on the other hand, can find only those plans that are equivalent to the query on all databases – which are very few in the absence of constraints.

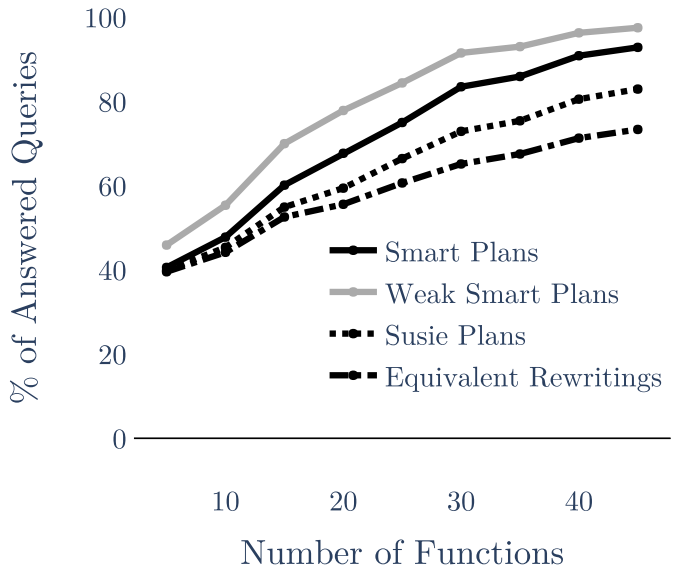

Figure 6.7: Percentage of answered queries

6.6.2 Real-World Web Services

In our second series of experiments, we apply the methods to real-world Web services. We use the same functions than in Chapter 5. Besides, as these Web services do not contain many existential variables, we added the set of functions based on information extraction techniques (IE) from Preda et al. [98].

Table 6.1 shows the number of functions and the number of relations for each Web service. Table 6.2 gives examples of functions. Some of them are recursive. For example, MusicBrainz allows querying for the albums that are related to a given album – like in Example 6.3.3. All functions are given in the same schema. Hence, in an additional setting, we consider the union of all functions from all Web services.

Note that our goal is not to call the functions. Instead, our goal is to determine whether a smart plan exists – before any functions have to be called.

Web Service	Functions	Relations
MusicBrainz	23	42
LibraryThing	19	32
Abe Books	9	8
LastFM	17	30
ISBNdb	14	20
Movie DB	12	18
UNION	74	82

Table 6.1: Our Web services

For each Web service, we considered all queries of the form $q(x) \leftarrow r(a, x)$ and $q(x) \leftarrow r^-(a, x)$, where r is a relation used in the function definitions of that Web service. We ran the Susie algorithm, the equivalent rewriting algorithm, and our

$getDeathDate(\underline{x}, y, z) \leftarrow hasId^{-}(x, y) \wedge diedOnDate(y, z)$
$getSinger(\underline{x}, y, z, t) \leftarrow hasRelease^{-}(x, y) \wedge released^{-}(y, z) \wedge hasId(z, t)$
$getLanguage(\underline{x}, y, z, t) \leftarrow hasId(x, y) \wedge released(y, z) \wedge language(z, t)$
$getTitles(\underline{x}, y, z, t) \leftarrow hasId^{-}(x, y) \wedge wrote^{-}(y, z) \wedge title(z, t)$
$getPublicationDate(\underline{x}, y, z) \leftarrow hasIsbn^{-}(x, y) \wedge publishedOnDate(y, z)$

Table 6.2: Examples of real functions (3 of MusicBrainz, 1 of ISBNdb, 1 of LibraryThing)

algorithm for each of these queries. The run-time is always less than 2 seconds for each query. Table 6.3 shows the ratio of queries for which we could find smart plans. We first observe that our approach can always answer at least as many queries as the other approaches can answer. Furthermore, there are cases where our approach can answer strictly more queries than Susie.

Web Service	Susie	Eq. Rewritings	Smart Plans
MusicBrainz (+IE)	48% (32%)	48% (32%)	48% (35%)
LastFM (+IE)	50% (30%)	50% (30%)	50% (32%)
LibraryThing (+IE)	44% (27%)	44% (27%)	44% (35%)
Abe Books (+IE)	75% (14%)	63% (11%)	75% (14%)
ISBNdb (+IE)	65% (23%)	50% (18%)	65% (23%)
Movie DB (+IE)	56% (19%)	56% (19%)	56% (19%)
UNION with IE	52%	50%	54%

Table 6.3: Percentage of queries with smart plans

The advantage of our algorithm is not that it beats Susie by some percentage points on some Web services. Instead, the crucial advantage of our algorithm is the guarantee that the results are complete. If our algorithm does not find a plan for a given query, it means that there cannot exist a smart plan for that query. Thus, even if Susie and our algorithm can answer the same number of queries on AbeBooks, only our algorithm can guarantee that the other queries cannot be answered at all. Thus, only our algorithm gives a complete description of the possible queries of a Web service.

Rather short execution plans can answer some queries. Table 6.4 shows a few examples. However, a substantial percentage of queries cannot be answered at all. In MusicBrainz, for example, it is not possible to answer $produced(a, x)$ (i.e., to know which albums a producer produced), $hasChild^{-}(a,x)$ (to know the parents of a person), and $marriedOnDate^{-}(a, x)$ (to know who got married on a given day). These observations show that the Web services maintain control over the data, and do not allow exhaustive requests.

Query	Plan
hasTrackNumber	*getReleaseInfoByTitle, getReleaseInfoById*
hasIdCollaborator	*getArtistInfoByName, getCollaboratorIdbyId, getCollaboratorsById*
publishedByTitle	*getBookInfoByTitle, getBookInfoById*

Table 6.4: Example Plans (2 of MusicBrainz, 1 of ABEBooks)

6.7 Discussion

In Chapter 5, the problem of finding equivalent rewritings is reduced to the problem of finding a word in a context-free grammar. This type of grammar makes it possible to check the emptiness of the language in polynomial time or to compute the intersection with a regular language. The emptiness operation can be used to check in advance if an equivalent rewriting exists, and the intersection can be used to restrict the space of solutions to valid execution plans.

In this chapter, we also define a language: the language of bounded plans. Unfortunately, it turns out that this language is not context-free.

Here we prove that we cannot represent smart plans with a context-free language in the general case. To do so, we will use a generalisation of Olden's Lemma presented in [9].

Lemma 6.7.1 (Bader-Moura's Lemma)**.** *For any context-free language L, $\exists n \in \mathbb{N}$ such that $\forall z \in L$, if d positions in z are "distinguished" and e positions are "excluded", with $d > n^{e+1}$, then $\exists u, v, w, x, y$ such that $z = uvwxy$ and:*

1. *vx contains at least one distinguished position and no excluded positions*
2. *if r is the number of distinguished positions and s the number of excluded positions in vwx, then $r \leq n^{s+1}$*
3. *$\forall i \in \mathbb{N}, u.v^i.w.x^i.y \in L$*

Theorem 6.7.2. *For $|\mathcal{R}| > 4$, the language of the smart plans is not context-free.*

Proof. We begin with a property for the words of our language.

Lemma 6.7.3. *Let $q(x) \leftarrow r(a, x)$ be an atomic query and $\mathcal{R}$ be a set of relations. For each relation r' different from r, the number of r' in a word of the language of the bounded plans is equal to the number of r'^- in this word.*

This lemma is easy to verify.

Let us now suppose our language of bounded plans is context-free. We define n according to the Barder-Moura's lemma. Let a, b, c be three distinct relations in $|\mathcal{R}|$ (they exist as $|\mathcal{R}| > 4$). Let $k = n^{10+1}$. We consider the word $z = a.b^k.a.c.c^-.a^-.b^{-k}.a^-.a.b^k.a.a^-.b^{-k}.a^-$ in which we distinguish all the b and b^- and exclude all the a, a^-, c, and c^-.

We write $z = u.v.w.x.y$. As $v.x$ contains no excluded positions, v contains only bs or only b^-s (and the same is right for x as well). As we must keep the same

number of b and b^- according to the previous lemma, either $v = b^j$ and $x = b^{-j}$ or $v = b^{-j}$ and $x = b^j$ (for some $j \in \mathbb{N}$).

In z, it is clear that $a.b^k.a.c$ is the forward path F as c appears only once. c^- and the last $a^-.b^{-k}.a^-$ are generated by the B. $a^-.b^{-k}.a^-.a.b^k.a$ is generated by L.

The forward-path defines the number of consecutive bs between two as (and so the number of consecutive b^-s between two a^-s). Under these conditions, it is impossible to make the four groups of b and b^- vary the same way, so it is impossible to define v and x, and so the language is not context-free. □

This has two consequences: First, it is not trivial to find the intersection with the valid execution plans as it was done in Chapter 5. Second, it explains the exponential complexity bound for our algorithm: our language is more complicated than a context-free grammar, and therefore, there is a priori no reason to believe that the emptiness problem is polynomial.

6.8 Conclusion

In this chapter, we have introduced the concept of smart execution plans for Web service functions. These are plans that are guaranteed to deliver the answers to the query if they deliver results at all. We have formalised the notion of smart plans, and we have given a correct and complete algorithm to compute smart plans. Our experiments have demonstrated that our approach can be applied to real-world Web services. All experimental data, as well as all code, is available at the URL given in Section 6.6. We hope that our work can help Web service providers to design their functions, and users to query the services more efficiently.

Chapter 7

Pyformlang

> Language is a process of free creation.
>
> *Noam Chomsky*

In Chapter 5 and Chapter 6, we have investigated a solution for query rewriting based on formal languages. During our experiments, we had to use algorithms that were rarely implemented, or in separated libraries. Therefore, we created a new Python library: Pyformlang. It implements most of textbooks algorithms and thus, we imagine it as a perfect companion for a lecture on formal languages.

This chapter comes from a full paper published at SIGCSE 2021 [106]:

> Romero, J.
> Pyformlang: An Educational Library for Formal Language Manipulation
> Full paper at SIGCSE 2021

****

Formal languages are widely studied and used in computer science. However, only a small part of this domain is brought to a broader audience, and so its uses are often limited to regular expressions and simple grammars. In this chapter, we introduce Pyformlang, a practical Python library for formal languages. Our library implements the most common algorithms of the domain, accessible by an easy-to-use interface. The code is written exclusively in Python, with a clear structure, so as to allow students to play and learn with it.

7.1 Introduction

Python has become one of the most popular programming languages over the past few years (https://insights.stackoverflow.com/survey/2019) and particularly among students. These students often start their journey in computer science by attending a lecture on formal languages. For example, in France, this subject is the first topic

taught in "classe préparatoire". However, there are few tools to manipulate formal languages with Python.

The range of applications of formal languages is infinite: They include information extraction with regular expressions for knowledge base construction as in Yago [123] or Quasimodo in Chapter 3, code parsing with context-free grammars, video game artificial intelligence using finite automata [101], or even query rewriting using complex manipulations of context-free grammars and regular expressions as in Chapter 5.

In this chapter, we present Pyformlang, a Python3 library for manipulating formal languages. In the first section, we will discuss other libraries for manipulating formal languages. Then, in the second section, we introduce Pyformlang and its components. Finally, we show how to use Pyformlang in practice.

7.2 Previous Work

The most used part of formal languages is surely regular expressions. Most programming languages implement them natively. In Python, the *re* library provides many possibilities to use regular expressions. Text processing is the primary use case of this library, and therefore it lacks more advanced formal language manipulation tools: For example, finite automata are missing from the standard library. Thus, one cannot transform a regular expression into its equivalent finite automaton. More operations like concatenation, union or Kleene stars are not available at the level of regular expressions.

For parsing natural language, the most common tool are context-free grammars. Some libraries are specialised in language parsing, such as Lark (https://pypi.org/project/lark-parser/) or more generally NLTK [81]. However, they are focused on parsing natural language, make no link with the rest of the theory, and often rely on an external file format (.lark for instance). As an example, none of the existing libraries allows the intersection between a context-free grammar and a regular expression or a finite state automaton (which is also a context-free grammar).

FAdo [104] is a library that focuses mainly on finite state automata, with few other functionalities. The language of FAdo is Python2, a deprecated language (https://www.python.org/dev/peps/pep-0373/) which is not compatible with Python3. The library implements regular expressions, deterministic, non-deterministic and general automata and the transformations to go from one to the other. One can also perform standard operations such as union, intersection, concatenation or emptiness. However, FAdo does not go further than finite state automata. For example, there is no link with context-free grammars.

JFLAP [105] is a tool written in Java to manipulate formal languages and it is the most advanced tool available. It implements regular expressions, deterministic and non-deterministic automata, context-free grammars, push-down automata, and Turing Machines. A user can also perform transformations between these structures. In addition, it provides roughly the same basic operations as FAdo for manipulating finite state automata. However, some advanced operations are missing (e.g. the intersection of context-free grammar and a regular expression). Furthermore, the

usage of Java can be a limiting factor for fresh computer science students, and it makes fast experimentation harder.

OpenFST [4] is a library to manipulate weighted finite-state transducers written in C++. Some Python wrappers were also implemented. OpenFST does not support operations beyond the realm of transducers.

Vaucanson [36] is an open-source C++ platform dedicated to the manipulation of finite weighted automata. The usage of C++ makes it very fast. It has bindings for Python3, but it cannot be installed directly from PyPi, making its usage more complex. The content on automata and transducers is very advanced, but there is no link with non-regular languages.

Some other tools exist to manipulate formal languages but they are either no longer available or very specialised. For example, Covenant [68] specialises in the intersection of context-free grammars (which are not necessary context-free) but is no longer accessible.

7.3 Pyformlang

Pyformlang is a Python3 library specialised in formal language manipulation. It is freely available on PyPI (https://pypi.org/project/pyformlang/) and the source code is on Github (https://github.com/Aunsiels/pyformlang). The full documentation can also be accessed on ReadTheDocs (https://pyformlang.readthedocs.io).

We chose to implement Pyformlang in Python3 in order to make it easily accessible. The implementation of each algorithm follows the one presented in the relevant textbooks. This way, a student can easily follow every detail of the lectures in Pyformlang. Unless otherwise mentioned, we used Hopcroft [75] as the source of most algorithms, since it is the most popular textbook for formal languages.

7.3.1 Regular Expressions

Pyformlang implements the operators of textbooks, which deviate slightly from the operators in Python.

- The concatenation can be represented either by a space or a dot (.)
- The union is represented either by | or +
- The Kleene star is represented by *
- The epsilon symbol can either be *epsilon* or $

It is also possible to use parentheses. All symbols except the space, ., |, +, *, (,) and $ can be part of the alphabet. All other common regex operators (such as [])) are syntactic sugar that can be reduced to our operators.

We deviate in one important point from the standard implementation of regular expressions. Usually, the alphabet consists of all single characters (minus special ones), thus creating a concatenation when encountering consecutive characters. In Pyformlang, we consider that consecutive characters are a single symbol. We wanted

to have a more general approach not bound to text processing. As an example, in normal Python, $na*$ will be interpreted as all words starting by n and ending by an indeterminate amount of a (like $naaa$). In Pyformlang it represents all strings composed of zero or more repetitions of the letter na (like $nanananana$). This deviation allows expressing a wider variety of words. Still, our library also allows the standard semantics though a wrapper of Python regular expressions.

As most users are familiar with the implementation in the Python standard library (which follows Perl's standard), our library can transform a Python regular expression into a regular expression compatible with our implementation. This transformation gives access to a more diversified function set for manipulating regular expressions and also makes it possible to have additional representations (such as finite automata, see Section 7.3.2) that are missing from the standard library. The transformation modifies the initial regular expression to reduce it to the fundamental operators presented above. Then, our parser can take the transformed regular expression and turn it into the internal representation.

As is the case in most systems, we used a tree structure to represent the regular expressions, where each node is an operator, and each leaf is a symbol in the alphabet.

Pyformlang contains the fundamental transformations on regular expressions, which produce again regular expressions: concatenation, Kleene star and unions. The transformation of a regular expression into an equivalent non-deterministic automaton with epsilon transitions gives access to additional operations.

Example 7.3.1. *We show here an example of how to use the regular expressions in our library:*

```
from pyformlang.regular_expression import Regex

regex = Regex("abc|d")

regex.accepts(["abc"])  # True
regex.accepts(["a", "b", "c"])  # False
regex.accepts(["d"])  # True

regex1 = Regex("a b")
regex_concat = regex.concatenate(regex1)
regex_concat.accepts(["d", "a", "b"])

print(regex_concat.get_tree_str())
# Operator(Concatenation)
#   Operator(Union)
#     Symbol(abc)
#     Symbol(d)
#   Operator(Concatenation)
#     Symbol(a)
#     Symbol(b)

# Give the equivalent finite-state automaton
regex_concat.to_epsilon_nfa()

from pyformlang.regular_expression import PythonRegex

```

```
p_regex = PythonRegex("a+[cd]")
p_regex.accepts(["a", "a", "d"])  # True
# As the alphabet the composed of single caracters, one
# could also write
p_regex.accepts("aad")  # True
p_regex.accepts(["d"])  # False
```

Listing 7.1: Regular Expression Example

7.3.2 Finite-State Automata

Pyformlang contains the three main types of non-weighted finite-state automata: deterministic, non-deterministic and non-deterministic with epsilon transitions.

We implemented the necessary possible transformations. First, it is possible to transform any non-deterministic finite state automaton with epsilon transitions into a non-deterministic finite without epsilon transitions. Then, it is possible to turn a non-deterministic finite automaton into a deterministic one. Besides, as finite-state automata and regular expressions are equivalent, we offer the possibility to go from one to the other. Finally, our library can transform an automaton into a finite state transducer, which accepts only the words in the language of the automaton and outputs the input word (see 7.3.6).

We implemented the fundamental operations on automata:

- Get the complementary, the reverse and the Kleene star of an automaton.
- Get the difference, the concatenation and the intersection between two automata.
- Check if an automaton is deterministic.
- Check if an automaton is acyclic.
- Check if an automaton produces the empty language.
- Minimise an automaton using the Hopcroft's minimisation algorithm for deterministic automata [74, 140].
- Check if two automata are equivalent.

Internally, the automaton is implemented using dictionaries and these dictionaries are accessible to the user who wants to manipulate them.

An advantage of finite-state automata is that they offer a nice visual representation (at least for smaller automata). Fundamentally, a finite-state automaton can be represented as a directed graph with two kinds of special nodes: Starting nodes and final nodes. We offer the possibility to turn a finite automaton into a Networkx [72] MultiDiGraph. In addition, the user can also save the finite-automaton into a dot file (https://graphviz.org/doc/info/lang.html) to print it in a GUI. An example is given in Figure 7.1.

Example 7.3.2. *We show here an example of how to use the finite automata in our library:*

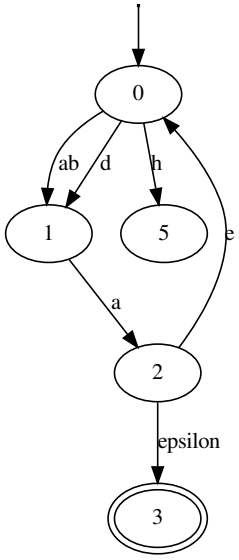

Figure 7.1: Visualisation of a finite-state automaton

```
from pyformlang.finite_automaton import EpsilonNFA

enfa = EpsilonNFA()
enfa.add_transitions(
    [(0, "abc", 1), (0, "d", 1), (0, "epsilon", 2)])
enfa.add_start_state(0)
enfa.add_final_state(1)
enfa.is_deterministic()  # False

dfa = enfa.to_deterministic()
dfa.is_deterministic()  # True
dfa.is_equivalent_to(enfa)  # True

enfa.is_acyclic()  # True
enfa.is_empty()  # False
enfa.accepts(["abc", "epsilon"])  # True
enfa.accepts(["epsilon"])  # False

enfa2 = EpsilonNFA()
enfa2.add_transition(0, "d", 1)
enfa2.add_final_state(1)
enfa2.add_start_state(0)
enfa_inter = enfa.get_intersection(enfa2)
enfa_inter.accepts(["abc"])  # False
enfa_inter.accepts(["d"])  # True
```

Listing 7.2: Finite Automata Example

7.3.3 Finite-State Transducer

Pyformlang implements non-weighted finite-state transducers and operators on them: the concatenation, the union and the Kleene star. Finite state transducers can be built and used to translate one word into another one. In addition, we

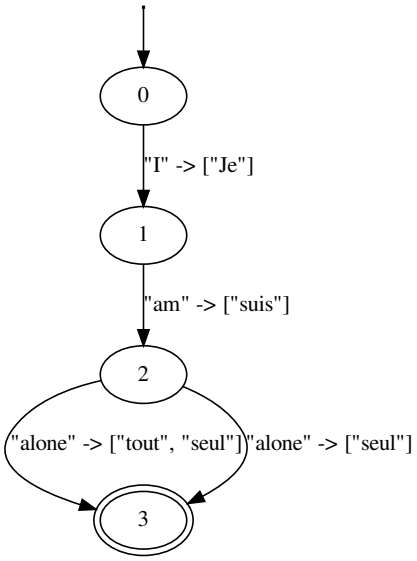

Figure 7.2: A finite-state transducer

offer an intersection function to intersect a finite-state transducer with an indexed grammar (see Section 7.3.6).

Just like finite state automata, it is possible to turn a finite-state transducer into a NetworkX graph and to save it into a dot file. Figure 7.2 shows the finite-state transducer used in Example 7.3.3.

Example 7.3.3. *We show here an example of how to use the finite state transducers in our library:*

```
from pyformlang.fst import FST

fst = FST()
fst.add_transitions(
    [(0, "I", 1, ["Je"]), (1, "am", 2, ["suis"]),
    (2, "alone", 3, ["tout", "seul"]),
    (2, "alone", 3, ["seul"])])
fst.add_start_state(0)
fst.add_final_state(3)
list(fst.translate(["I", "am", "alone"]))
# [['Je', 'suis', 'seul'],
#  ['Je', 'suis', 'tout', 'seul']]
```

Listing 7.3: Finite State Transducer Example

7.3.4 Context-Free Grammars

Pyformlang implements context-free grammars and the essential operations on them. One can construct a context-free grammar in two different ways. The first one consists in using internal representation objects to represent variables, terminals and production rules. This initialisation process can be quite wordy, and in most cases, it is not necessary. However, it is close to textbooks representations and allows a better understanding of context-free grammars. Besides, it is easier to use for a computer

program. The other way, used by most libraries, uses a string representation of the context-free grammar. Example 7.3.4 shows how this construction looks.

As many algorithms use the Chomsky Normal Form (CNF), our library offers the possibility to transform a context-free grammar into its CNF.

Context-free grammars are equivalent to push-down automata. So, our library allows transforming a context-free grammar into its equivalent push-down automaton accepting by empty stack.

Our context-free grammar also contains several operations linked to the closure properties: The concatenation, the union, the (positive) closure, the reversal and the substitution of terminals by another context-free grammar. In general, the intersection of two context-free grammars is not a context-free grammar, except if one of them is a regular expression or a finite automaton. Hopcroft [75] presents in algorithm that first transforms the grammar into an equivalent push-down automaton accepting by empty stack, and then into an equivalent push-down automaton accepting by final state. Next, it performs the intersection with the regular expression (or finite automaton) and transforms the push-down automaton obtained back into a context-free grammar (by transiting through a push-down automaton accepting by empty stack). However, this solution is costly, and we preferred the solution given by Bar-Hillel [7,12], which does not require push-down automata and is much more efficient. The algorithm of Bar-Hillel directly creates new non-terminals and the transitions between them from the context-free grammar and the regular expression.

We implemented various other operations, such as:

- Checking if a given context-free grammar produces a word
- Checking if a word is part of the language generated by the grammar
- Generate words in the grammar (through a generator as the language associated with the grammar is not necessarily finite)
- Checking whether a grammar is finite or not

Finally, we also added a LL(1) parser [3] which allows us to obtain parsing trees and derivations. Figure 7.3 shows an example of a parsing tree from the grammar defined in Example 7.3.4.

Example 7.3.4. *We show here an example of how to use the context-free grammars in our library:*

```
from pyformlang.cfg import CFG
from pyformlang.cfg.llone_parser import LLOneParser
from pyformlang.regular_expression import Regex

cfg = CFG.from_text("""
    S -> NP VP PUNC
    PUNC -> . | !
    VP -> V NP
    V -> buys | touches | sees
    NP -> georges | jacques | leo | Det N
    Det -> a | an | the
```

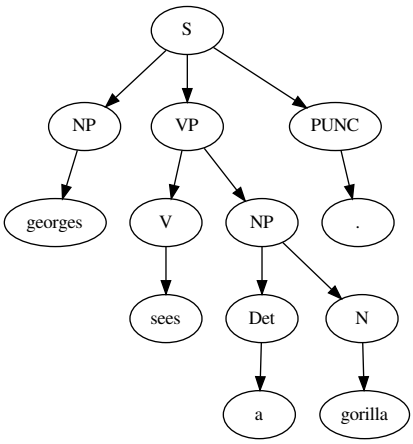

Figure 7.3: Parsing tree

```
    N -> gorilla | sky | carrots
""")
regex = Regex("georges touches (a|an) (sky|gorilla) !")

cfg_inter = cfg.intersection(regex)
cfg_inter.is_empty()  # False
cfg_inter.is_finite()  # True
cfg_inter.contains(["georges", "sees",
                    "a", "gorilla", "."])  # False
cfg_inter.contains(["georges", "touches",
                    "a", "gorilla", "!"])  # True

cfg_inter.is_normal_form()  # False
cnf = cfg.to_normal_form()
cnf.is_normal_form()  # True

llone_parser = LLOneParser(cfg)
parse_tree = llone_parser.get_llone_parse_tree(
    ["georges", "sees", "a", "gorilla", "."])
parse_tree.get_leftmost_derivation(),
# [[Variable("S")],
#  [Variable("NP"), Variable("VP"), Variable("PUNC")],
#  [Terminal("georges"), Variable("VP"),
#   Variable("PUNC")],
#  [Terminal("georges"), Variable("V"), Variable("NP"),
#   Variable("PUNC")],
#  [Terminal("georges"), Terminal("sees"),
#   Variable("NP"), Variable("PUNC")],
#  [Terminal("georges"), Terminal("sees"),
#   Variable("Det"), Variable("N"), Variable("PUNC")],
#  [Terminal("georges"), Terminal("sees"),
#   Terminal("a"), Variable("N"), Variable("PUNC")],
#  [Terminal("georges"), Terminal("sees"),
#   Terminal("a"), Terminal("gorilla"),
#   Variable("PUNC")],
#  [Terminal("georges"), Terminal("sees"),
```

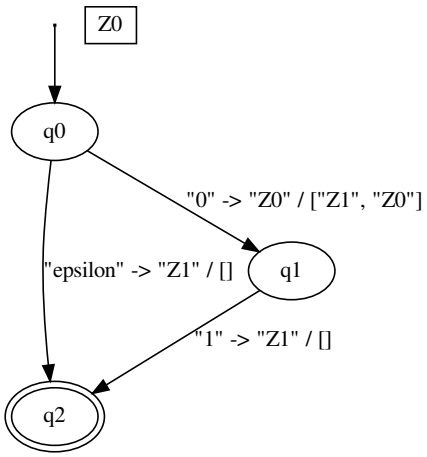

Figure 7.4: Visualisation of a push-down automaton

```
48 #    Terminal("a"), Terminal("gorilla"), Terminal(".")]]
```

Listing 7.4: Context-Free Grammar Example

7.3.5 Push-Down Automata

Pyformlang implements push-down automata accepting by final state or by empty stack. It also adds the possibility to go from one acceptance to the other. As context-free grammars are equivalent to push-down automata accepting by empty stack, we implemented the transformation to go from one to the other. Push-down automata have the same closure properties as context-free grammars. We implemented the intersection with a regular language using the native algorithm in Hopcroft [75].

Just like finite automata and finite state transducers, push-down automata can be visualised as a graph with NetworkX. Figure 7.4 shows the push-down automaton used in Example 7.3.5.

Example 7.3.5. *We show here an example of how to use the push-down automata in our library:*

```
1  from pyformlang.pda import PDA
2
3  pda = PDA()
4  pda.add_transitions(
5      [
6          ("q0", "0", "Z0", "q1", ("Z1", "Z0")),
7          ("q1", "1", "Z1", "q2", []),
8          ("q0", "epsilon", "Z1", "q2", [])
9      ]
10 )
11 pda.set_start_state("q0")
12 pda.set_start_stack_symbol("Z0")
13 pda.add_final_state("q2")
14
15 pda_final_state = pda.to_final_state()
16 cfg = pda.to_empty_stack().to_cfg()
```

```
17 cfg.contains(["0", "1"])  # True
```

Listing 7.5: Context-Free Grammar Example

7.3.6 Indexed Grammar

Aho [2] introduced indexed grammars, and they are an excellent example of non-context-free grammars. For example, they have application in natural language processing [69]. To the best of our knowledge, no library provides an implementation for this class of grammars. As indexed grammars are less constrained, they do not allow as many operations as context-free grammars: They can represent more languages but the languages are harder to manipulate. Still, Pyformlang implements essential functions such as checking if the grammar is empty or intersecting with a regular expression or a finite automaton. Indeed, indexed grammars are stable by this last operation.

As indexed grammars are not well documented, we give here more explanations about how they work, as well as the emptiness algorithm and the algorithm to intersect with a regular expression.

Definition

Indexed grammars [2] include the context-free grammars, but are strictly less expressive than context-sensitive grammars.

They generate the class of indexed languages, which contains all context-free languages. A nice feature of this class of languages is that it conserves closure properties and decidability results. In addition to the set of terminals and non-terminals from the context-free grammars, the indexed grammars introduce the set of *index symbols*. Following [75], an indexed grammar is a 5-tuple (N, T, I, P, S) where

- N is a set of variables or non-terminal symbols
- T is a set of terminal symbols
- I is a set of index symbols
- $S \in N$ is the start symbol, and
- P is a finite set of productions.

Each production in P takes one of the following forms:

$$\begin{aligned} A[\sigma] &\to a && \text{(end rule)} \\ A[\sigma] &\to B[\sigma]C[\sigma] && \text{(duplication rule)} \\ A[\sigma] &\to B[f\sigma] && \text{(production rule)} \\ A[f\sigma] &\to B[\sigma] && \text{(consumption rule)} \end{aligned}$$

Here, A, B, and C are non-terminals, a is a terminal, and f is an index symbol. The part in brackets [...] is the so-called stack. The left-most symbol is the top of the stack. σ is a special character that stands for the rest of the stack. For example, the duplication rule states that a non-terminal with a certain stack gives rise to two other non-terminals that each carry the same symbols on the stack.

Emptiness

The work of [2] gives an algorithm to determine the emptiness of an indexed grammar (Algorithm 6). The method takes as input an indexed grammar $G = (N, T, I, P, S)$. For every non-terminal $A \in N$, the algorithm keeps track of the set of unexpanded non-terminals from partial derivations that start at A. Every such set E_A has the property that there exists $\omega \in (T \cup E_A)^*$ such that $A \xrightarrow[G]{*} \omega$. For each non-terminal A, the algorithm maintains these sets in a collection $marked(A)$. This collection is at first initialised with the non-terminal A itself (Lines 1-2). For the end rules, we add the set of the empty set (Lines 3-4).

The algorithm then iteratively updates this set for the different types of rules (Lines 5-15). Only two kinds of rules create updates: duplication rules and production rules. For duplication rules, changes in the marked symbols for the right-hand-side non-terminals change the marked symbols for the non-terminal on the left-hand side. Production rules are treated in parallel with consumption rules. The code checks whether a symbol that was pushed will also be popped.

In order to prove that the grammar generates a non-empty language, we need to show that $\emptyset \in marked(S)$, where S is the start symbol. In this case, we have $S \xrightarrow[G]{*} \omega$, where $\omega \in T$. Therefore, we can say that the grammar is non-empty as soon as $\emptyset \in marked(S)$ (Lines 15-16). Otherwise, the grammar is empty.

Complexity. Let $G = (N, T, I, P, S)$ be an indexed grammar in reduced form. Let $n = |N|$ be the number of non-terminals. Let us consider the number of marked symbols in our algorithm. For each non-terminal, the maximum number of marked symbols is $\mathcal{O}(2^n)$ (the number of partitions of N). Then, the number of marked symbols is $\mathcal{O}(n2^n)$, which is the complexity of our algorithm. The original variant of the algorithm [2] had a complexity of $\Theta(p \times (2^{ln} + 2^{2n}))$, where p is the number of non-consumptions, and l is the maximum number of consumption rules for a production symbol, because the original variant of the algorithm generates all possible marked sets and all the rules to mark them in advance.

Intersection with a Regular Language

To perform the intersection of an indexed grammar with a regular expression, [2] proposed to use Nondeterministic Finite Transducers.

Definition 7.3.6 (Nondeterministic Finite Transducer)**.** *A Nondeterministic Finite Transducer (NFT) is a 6-tuple* $(Q, T, \Sigma, \delta, q_0, F)$ *such that:*

- Q *is a set of states*
- T *is a set of input symbols*

Algorithm 6: Emptiness Algorithm

Data: An indexed grammar $G = (N, T, I, P, S)$
Result: Whether the grammar is empty or not

foreach $A \in N$ **do**
  $marked(A) \leftarrow \{\{A\}\}$
foreach *end rule* $A[\sigma] \to a$ **do**
  $marked(A) \leftarrow marked(A) \cup \{\emptyset\}$
while *new sets are marked* **do**
  foreach *duplication rule* $A[\sigma] \to B[\sigma]C[\sigma]$ **do**
    $marked(A) \leftarrow marked(A) \cup \{E_B \cup E_C \mid E_B \in marked(B) \wedge E_C \in marked(C)\}$
  foreach *production rule* $A[\sigma] \to B[f\sigma]$ **do**
    if $\emptyset \in marked(B)$ **then**
      $marked(A) \leftarrow marked(A) \cup \emptyset$
    foreach $E_B \in marked(B)$ *such that* $E_B = \{B_1, B_2, \dots B_r\}$ *and* $\forall i \in \{1 \dots r\}, \exists$ *consumption* $B_i[f\sigma] \to C_i[\sigma]$ **do**
      for $i = 1, ..., r$ **do**
        let $\Gamma_i = \{C | \exists B_i[f\sigma] \to C[\sigma]\}$
      $marked(A) \leftarrow marked(A) \cup \{\bigcup marked(C_i) \mid \forall i \in [1; r], C_i \in \Gamma_i\}$
    foreach *consumption rule* $B[f\sigma] \to C[\sigma]$ **do**
      $marked(A) \leftarrow marked(A) \cup marked(C)$
  if $\emptyset \in marked(S)$ **then**
    return *The grammar is not empty*
return *The grammar is empty*

- Σ *is a set of output symbols*
- δ *is a transition function from* $Q \times T$ *into a finite subset of* $Q \times \Sigma^*$.
- $q_0 \in Q$ *is the initial state*
- $F \subseteq Q$ *is the set of final states.*

In short, NFT are finite state automata which give an output. These tranducers will be useful to filter our indexed grammar by using an NFT mapping.

Definition 7.3.7 (NFT Mapping)**.** *Let* $M = (Q, T, \Sigma, \delta, q_0, F)$ *be a NFT. For* $w \in T^*$*, we define* $M(w)$ *as the set of words output by* M *when giving* w *as an input. A NFT mapping on a language* $L \subseteq T^*$ *is* $M(L) = \cup_{w \in L} M(w)$.

Given a regular language R, and given a NFT M such that for $w \in R$, $M(w) = w$ and $M(w) = \emptyset$ otherwise, the intersection of a language L with R will be $M(L)$. Our language L is an indexed language, so we need to be able to construct another indexed grammar from L and R. Luckily, [2] tells us that $M(L)$ is also an indexed grammar.

Theorem 7.3.8. *Let L be an indexed language and M be a NFT. Then $M(L)$ is an indexed language.*

Proof. We present here the construction of $L(G')$. For the full proof, see [2].
Let $G = (N, T, F, P, S)$ be a indexed grammar in the reduced form and $M = (Q, T, \Sigma, \delta, q_0, K)$. We construct the indexed grammar $G' = (N', \Sigma, F', P', S')$ such that $L(G') = M(L(G))$, where the non-terminals are of the form (p, X, q) with p, q in Q and X in $N \cup T \cup \epsilon$, as follows:
For each rule r in P, we do:

- If r is a duplication rule $A[\sigma] \to B[\sigma]C[\sigma]$, we add the duplication rules $(p, A, q)[\sigma] \to (p, B, r)[\sigma](r, C, q)[\sigma]$ to P' for all p, q, r in Q.
- If r is a production rule $A[\sigma] \to B[f\sigma]$, we add the production rules $(p, A, Q)[\sigma] \to (p, B, q)[f\sigma]$ to P' for all p, q in Q
- If r is a consumption rule $A[f\sigma] \to B[\sigma]$, we add the consumption rule $(p, A, q)[f\sigma] \to (p, B, q)[\sigma]$ to P' for all p, q in Q.
- If r is an end rule $A[\sigma] \to a$, we add the duplication rules $(p, A, q)[\sigma] \to (p, a, q)[\sigma]T[\sigma]$ to P' for all p, q in Q and the end rule $T[\sigma] \to \epsilon$.

Then we add the following duplication rules to P':

- $(p, a, q)[\sigma] \to (p, a, r)[\sigma](r, \epsilon, q)[\sigma]$
- $(p, a, q)[\sigma] \to (p, \epsilon, r)[\sigma](r, a, q)[\sigma]$
- $(p, \epsilon, q)[\sigma] \to (p, \epsilon, r)[\sigma](r, \epsilon, q)[\sigma]$

Next, for all transitions in δ from $(p, a)[\sigma]$ to $(q, x)[\sigma]$, we add the end rule $(p, a, q)[\sigma] \to x$ to P'. For all $p \in Q$, we add the end rule $(p, \epsilon, p)[\sigma] \to \epsilon$ to P'.

Finally, we add the starting duplication rules $S'[\sigma] \to (q_0, S, p)[\sigma]T[\sigma]$ for $p \in K$ and the end rule $T[\sigma] \to \epsilon$.

□

Example 7.3.9. *We show here an example of how to use indexed grammars in our library:*

```
from pyformlang.indexed_grammar import Rules,
    ConsumptionRule, EndRule, ProductionRule,
    DuplicationRule, IndexedGrammar
from pyformlang.regular_expression import Regex

all_rules = [ProductionRule("S", "D", "f"),
             DuplicationRule("D", "A", "B"),
             ConsumptionRule("f", "A", "Afinal"),
             ConsumptionRule("f", "B", "Bfinal"),
             EndRule("Afinal", "a"),
             EndRule("Bfinal", "b")]
indexed_grammar = IndexedGrammar(Rules(all_rules))
indexed_grammar.is_empty()  # False
```

```
14 i_inter = indexed_grammar.intersection(Regex("a.b"))
15 i_inter.is_empty()  # False
```

Listing 7.6: Indexed-Grammar Example

7.4 Conclusion

In this chapter, we introduced Pyformlang, a modern Python3 library to manipulate formal languages, especially tailored for practical and pedagogical purposes. It contains the main functionalities to manipulate regular expressions, finite state automata, finite state transducers, context-free grammars, push-down automata and indexed grammars. The code is fully documented and is open sourced on GitHub (https://github.com/Aunsiels/pyformlang).

Chapter 8

Conclusion

> Any fool can know. The point is to understand.
>
> *Albert Einstein*

8.1 Summary

In this thesis, we addressed challenges related to knowledge base construction and exploitation. More precisely, we worked on the following topics:

Commonsense Knowledge Extraction. In Chapter 3, we introduced a methodology to automatically extract commonsense knowledge from various untypical Web resources. Although the problem is complicated, we provided a scalable framework that outperforms other state-of-the-art approaches in terms of precision and recall, and that performs as well as manually designed knowledge bases on some tasks. Our knowledge base Quasimodo contains more than four million facts, with a very high precision for the top-ranked statements.

Commonsense Knowledge Exploitation. In Chapter 4, we developed a software to display Quasimodo. Besides, we illustrated many applications that benefit from a commonsense knowledge base.

Equivalent Rewritings With Integrity Constraints. In Chapter 5, we introduced a practical setting that has many favourable properties when it comes to query rewritings. It exploits a particular set of integrity constraints, the unary inclusion dependencies, and a particular set of functions, the path functions. With these parameters, we showed that finding if there exists an equivalent rewriting is polynomial, and we provided an algorithm to enumerate all of them.

Query Rewriting Without Integrity Constraints. In Chapter 6, we extended the work of Chapter 5 by removing the integrity constraints from the setting. We then showed that equivalent rewritings are of little interest, and we introduced a new category of plans: the smart plans. We characterised these plans and gave an algorithm to find them.

Formal Language Framework In Python. In Chapter 7, we introduced Pyformlang, a formal language manipulation framework written in Python. It was

initially written to support the algorithm in Chapter 5 and ended up providing a set of functionalities that cannot be found in other libraries.

8.2 Outlook

There are still many challenges remaining in the area we worked in. We here give some possible research directions that would be worth exploring in the future.

Commonsense Knowledge Evaluation. In Chapter 3, we gave a score to each statement using logistic regression. This classifier is simple and could be improved a lot. Besides, we could include new features such as linguistic features into the scoring, especially by using neural networks such as BERT [40] or GPT [23]. The downside is that training and scoring would become slower and would require more computation resources. We are also using Web services with limited access, such as Google Books. We would need a smart strategy to know when to ask for such additional features. This would be in line with budgeted learning algorithms [80]. However, these approaches often have limiting computation time and would not scale easily to our data.

Integrating Commonsense Knowledge Into Applications. Currently, neural network approaches perform exceptionally well on various tasks such as question answering [77]. However, these methods are mostly based on statistical signals and would benefit from external knowledge [82]. For example, when we ask GPT to autocomplete the sentence "The colour of the sea is" (using https://transformer.huggingface.co/), it gives us nothing related to colour. However, a human would most likely finish the sentence with "blue.". Some work [139] focused on incorporating commonsense knowledge from ConceptNet into neural networks, but hybrid approaches are still rare.

A Pipeline for Stereotype Analysis. The pipeline we created in Chapter 3 could be reused to analyse stereotypes and their evolution. For example, Google autocompletion gives us "why do white people have thin lips", from which we can extract the hypothesis *white people, have, thin lips*. [11] used Google autocompletion to extract stereotypes, and we could extend their methods to a larger scale. Besides, we could also adapt the pipeline to support multiple languages and see how stereotypes vary depending on the language.

Polynomial Equivalent Rewriting Existence. One of the surprising results of Chapter 5 is that finding the existence of an equivalent rewriting is polynomial in some practical settings. We now need to understand how much we can generalise our setting, while still getting the complexity results. In particular, we could consider non-atomic queries of a particular shape, such as a path. We could also consider functions with a more general shape, such as a tree. We could even see what happens when functions have multiple inputs. Finally, we could try to vary the set of constraints by adding, for example, role inclusion or Horn rules.

Smart Plans Without Optional Edge Semantics. In Chapter 6, we characterised smart plans when we were under the "optional edge semantics". This allowed us to use sub-functions. In the future, we would like to investigate what happens when we remove this semantics, and to see what conclusions we can draw.

Link To Description Logic. The work in Chapter 5 and Chapter 6 seems related

to work done in description logic [18]. We want to investigate deeper what are the relations between the two and how one approach could benefit from the other.

Extending Formal Language Manipulation. Currently, Pyformlang contains most textbook algorithms and operations for formal languages. This is a definite advantage when it comes to applying Pyformlang in a pedagogical context. However, it could also be used for research. In Chapter 5, we showed a potential use case. More advanced functionalities could be implemented, such as weighted transducers or Turing machines.

Thus, while our work has advanced the state of the art, it has also opened the door to new challenges. We hope that we and others can build on what we achieved, so as to tackle these challenges as well.

Appendix A

Résumé en français

Le cosmos est mon campement.

Alain Damasio

A.1 Introduction

Internet est une jungle d'information qui connecte environ 4,5 milliards d'humains. Cependant, il y est difficile de démêler le vrai du faux tant les sources sont contradictoires. Certains sites, comme Wikipédia, se sont spécialisées dans la production de connaissances, mais, en dehors de ces zones de sécurité, rien n'est sûr. Il y a plusieurs raisons à cela. Tout d'abord, les experts sont rares dans certains domaines. Même si des sites comme le réseau Stack Exchange les regroupent sous un même toit, de nombreuses coquilles subsistent. Ensuite, certaines personnes diffusent intentionnellement du faux contenu. Dans le domaine des news, nous parlons généralement de Fake News ou d'Infox. Le but d'une telle démarche est souvent de soutenir une idéologie ou d'attirer une large audience à travers des titres aguicheurs.

Avec l'internet, nous avons un exemple parfait de la dichotomie entre l'information et la connaissance. D'un côté, l'information prend de nombreuses formes, est floue, peut être soit vraie, soit fausse et abonde en ligne. De l'autre, la connaissance est structurée, précise, se concentre sur la vérité et connaît des limites. Pourtant, les deux marchent main dans la main, l'un éclairant l'autre. Par exemple, sans connaissance a priori, il serait impossible de comprendre de quoi parle le moindre article de journal: Qui est Emmanuel Macron? Qu'est-ce que l'Europe?

Conscient de l'importance des connaissances pour clarifier l'information, les informaticiens ont mis au point des systèmes pour les collecter: on parle de bases de connaissance. On en trouve très rapidement dans l'histoire, avec, en 1984, l'émergence du premier projet d'ampleur: Cyc [79]. Ce dernier utilise un effort humain pour regrouper un maximum de données sur le monde. Plus tard, grâce à l'expansion de Wikipedia, de nombreuses bases de connaissances ont émergé. Les exemples les plus emblématiques sont Freebase [130], DBpedia [6], Yago [123] et Wikidata [131]. Ces dernières sont, en partie ou totalement, générées automatiquement, ce qui leur permet d'atteindre de grandes proportions.

Les bases de connaissances ont des applications dans des domaines très variés. Google les utilise pour clarifier les requêtes des utilisateurs [44]. Si nous demandons "train pour Paris" à un moteur de recherche, Paris doit être compris comme étant la capitale de la France et non le héros grec, qui ne peut être accédé en train. Il est aussi possible de répondre à des questions grâce à une base de connaissance, et ce que fit Watson [62], le logiciel d'IBM capable de jouer à Jeopardy!, un quiz inversé où il faut deviner les questions. Les applications ne s'arrêtent pas là: chatbot, compréhension du langage, vérification de faits, construction d'une argumentation, génération d'explications, décryptage d'une image, ... La connaissance est utile dans chaque activité humaine.

Dans cette thèse, nous nous focalisons sur deux tâches liées aux bases de connaissance. Tout d'abord, nous montrons comment en générer une pour le sens commun à partir de logs de requêtes et de forums de question-réponse. Ensuite, nous étudions comment répondre à des questions simples quand l'accès à une base de connaissance est limité.

A.2 Quasimodo

Collecter des connaissances sur le sens commun est un vieux thème de l'IA qui a été lancé dans les années 60 par McCarthy [85]. Le but est d'obtenir des connaissances sur les propriétés des objets de tous les jours, comme le fait que les bananes sont jaunes, sur les comportements humains et leurs émotions, par exemple en disant que la mort provoque de la tristesse, ou sur des concepts généraux. Par opposition, les connaissances encyclopédiques, présentes dans les bases de données classiques, sont en général enseignées: c'est à l'école que nous apprenons que la capitale de l'Angleterre est Londres.

Le sens commun est primordial dans de nombreuses applications comme celles que nous avons présentées plus tôt. Pourtant, il est très compliqué de les extraire automatiquement. En effet, ce genre de connaissance est rarement exprimé (car partagé par tous) et les sources, que ce soit internet, des livres ou directement des humains, sont souvent complexes, contextuelles et biaisées.

Dans cette thèse, nous présentons Quasimodo, une base de connaissance construite automatiquement à partir de sources surprenantes. Les résultats de nos travaux ont été publiés à CIKM 2019 [111], puis à CIKM 2020 [110] sous forme d'une démo. L'idée générale est d'utiliser des questions plutôt que des affirmations pour obtenir des faits. Nous remarquons qu'une question n'est jamais anodine et fait des présuppositions sur le monde. Par exemple, quand nous demandons "Pourquoi le ciel est bleu?", nous pouvons déduire que le ciel est bleu. Ce genre d'interrogations se trouve facilement dans certaines sources telles que les requêtes faites à un moteur de recherche ou encore les forums de question-réponse. L'humain étant curieux par nature, il questionne souvent le sens commun et cherche à connaître l'origine des faits les plus banals.

Une fois cette observation faite, nous collectons des questions traitant de sujets donnés et commençant par "pourquoi" et "comment". Pour cela, nous utilisons l'autocomplétion de moteurs de recherche comme Google (qui nous donne un accès indirect à son historique de requêtes) et les questions posées sur des forums tels que

Quora, Answers.com ou Reddit. Ensuite, nous transformons les questions en affirmations. Par exemple, de la question "pourquoi les astronautes vont dans l'espace", nous obtenons "les astronautes vont dans l'espace". Puis, nous séparons le sujet, le verbe (aussi appelé prédicat) et le complément d'objet (ou simplement objet) grâce un algorithme d'OpenIE. Dans notre exemple, nous obtenons le triplet (astronautes, vont, dans l'espace). Finalement, nous normalisons les triplets pour les uniformiser. Ici, nous avons (astronaute, aller, dans l'espace).

Passée l'étape de la génération de faits, nous leur donnons un score qui représente leur plausibilité. Par exemple, le triplet (banane, être, jaune) devrait avoir un meilleur score que (banane, être, mauve). Pour cela, nous utilisons des sources extérieures. En outre, nous vérifions si un fait est sur Wikipédia ou si un sujet et un objet apparaissent ensemble dans une image. Ces sources sont des signaux positifs ou négatifs que nous fusionnons afin d'obtenir un score.

Au bout du procédé, nous obtenons Quasimodo, une base de connaissance qui comporte plus de quatre millions de faits associés à un score. Ce nombre est bien supérieur à ceux des bases de connaissances construites manuellement telle que ConceptNet. Comparé aux autres bases de connaissance construites automatiquement, Quasimodo présente une plus grande précision et un plus grand rappel, comme nous le montrons dans nos expériences. De plus, Quasimodo arrive à surpasser ses concurrents sur de nombreuses tâches pratiques telles la sélection de réponse à une question.

Le code et les données de Quasimodo sont mises gratuitement en ligne pour aider la communauté. De nombreuses questions restent ouvertes. En particulier, nous pourrions nous demander comment donner un meilleur score à un fait. Une autre direction serait d'évaluer à quel point les modèles de langage tels que BERT capturent le sens commun et, si possible, les améliorer pour qu'ils en prennent compte.

Nous venons de voir comment créer une base de connaissance sur le sens commun. Maintenant, intéressons nous à notre second problème, à savoir comment obtenir des données d'une base de connaissance avec un accès limité.

A.3 Réécriture de requêtes

Comme nous l'avons vu plus haut, une base de connaissance apporte une grande valeur ajoutée à de nombreuses applications. Il est donc logique que certains acteurs de l'économie construisent et vendent des connaissances précises et spécialisées dans certains domaines. Quand l'accès se fait en ligne automatiquement par des machines, nous parlons de services web. Dans le secteur de la musique, MusicBrainz propose un grand nombre de fonctionnalités qui permettent, par exemple, de trouver l'interprète d'une chanson.

Toutefois, posséder des données implique de pouvoir en contrôler l'accès. Dès lors, les services web publient des méthodes d'accès qui donnent une vue partielle sur la base de connaissance. Celles-ci dépendent en général de l'abonnement payé par le client. Par exemple, nous pourrions imaginer que MusicBrainz donne gratuitement la possibilité de trouver l'interprète d'une chanson, mais demande une participation financière pour trouver les albums d'un compositeur.

Un utilisateur de tels services web va chercher à répondre à ses requêtes le plus efficacement possible. Il va donc devoir combiner des méthodes d'accès pour, d'une part, obtenir un résultat correct et, d'autre part, pour optimiser des critères qui lui sont propres (temporel ou financier par exemple).

Imaginons que nous ayons accès à un service web proposant des données généalogiques. Nous souhaitons connaître la date de naissance de Marie Curie. Cependant, seules deux méthodes d'accès nous sont données: une qui nous donne les parents d'un individu et une autre qui nous donne la date de naissance des petits-enfants d'une personne. Pour répondre à notre requête initiale, nous devons demander les parents de Marie Curie, puis leurs parents et enfin nous appelons la fonction donnant la date de naissance des petits-enfants.

Ce problème est ce que l'on appelle de la réécriture de requête: étant donné une requête (trouver la date de naissance de Marie Curie) et des méthodes d'accès (donner les parents d'une personne), peut-on répondre à la requête? Si la réponse est oui et la réécriture donne exactement les réponses à la requête dans tous les cas, nous parlons de réécriture équivalente. Cette tâche a été largement étudiée dans la communauté et nous donnons dans cette thèse une réponse nouvelle à un sous problème pratique.

En général, réécrire une requête peut être très coûteux. Dans notre cas, nous considérons des requêtes simples dites atomiques. Cela signifie qu'il y a une connexion directe entre le sujet de la question et la réponse. Dans la requête: "trouver la date de naissance de Marie Curie", Marie Curie et le 7 novembre 1867 sont liées par une relation "date de naissance". Au contraire, la requête: "trouver les grands-parents de Marie Curie" n'est pas atomique car Marie Curie et ses grands-parents sont séparés par deux relations "parent".

Ensuite, nous considérons des méthodes d'accès qui sont des fonctions chemins à une seule entrée. Cela signifie que les méthodes suivent un chemin de connexions dans la base de connaissance. Par exemple, la fonction pour obtenir la date de naissance des petits-enfants suit le chemin "parent", "parent", "date de naissance". À l'opposé, une fonction donnant accès aux parents et notre fratrie n'est pas un chemin. En effet, nous distinguons trois chemins: "parent", "frère" et "sœur".

À partir de requêtes atomiques et de fonctions chemins, nous étudions deux problèmes. Tout d'abord, nous considérons le cas où la base de connaissance respecte des contraintes sur la structure qui sont appelées des dépendances d'inclusion unitaire. Par exemple, nous pouvons dire que si une personne a une date de naissance, alors elle a des parents. Dans cette situation, on peut savoir en temps polynomial si le problème de la réécriture équivalente admet une solution, puis la générer. À travers nos expériences, nous montrons que notre approche exacte surpasse les méthodes plus générales dans notre scénario. Ces travaux ont été publiés à ESWC 2020 [108]. En complément de notre article, nous avons publié une démo [107] accessible à dangie.r2.enst.fr.

Dans le second problème, nous supposons que la base de connaissance ne respecte aucune contrainte. Dans ce cas, la notion de réécriture équivalente a peu de sens car nous ne savons rien a priori qui nous permette de résonner dans toutes les situations. Nous avons donc défini une notion de "plan intelligent" qui, même sans information additionnelle, donne des garanties sur les solutions. Dans cette thèse, nous prouvons

que ce nouveau problème admet une solution et nous donnons un algorithme pour la générer.

En conclusion, nous apportons dans cette thèse un éclairage nouveau sur un problème de la communauté en exhibant un cas particulier tractable. De plus, nous avons exhibé une nouvelle notion de "plan intelligent" qui donne des garanties, même sans information a priori. Dans le futur, nous pourrions chercher à étendre nos résultats à des requêtes non atomiques ou des fonctions non chemins, mais qui respectent toujours certaines contraintes.

Avant de conclure cette thèse, nous décrivons ici une dernière contribution plus pratique: la création d'une bibliothèque Python pour manipuler les langages formels.

A.4 Pyformlang

Notre travail sur la réécriture de requête équivalente a nécessité l'emploi de grammaires hors contextes et d'expressions régulières. En particulier, nous avons eu recours à des algorithmes non implémentés jusque là. Ce manque, et en particulier dans le paysage de Python, nous a poussé à développer Pyformlang, une bibliothèque de manipulation de langages formels.

Celle-ci permet d'interagir de manière intuitive avec les expressions régulières, les automates finis, les grammaires hors contextes, les automates à piles, les transducteurs finis ainsi que les grammaires indexées. Les algorithmes implémentés sont ceux des livres de cours classiques, ce qui permet aux étudiants d'ajouter une touche pratique à la théorie.

A.5 Conclusion

En conclusion, dans cette thèse, nous avons abordé deux problèmes majeurs. Tout d'abord, nous avons décrit comment construire une base de connaissance sur le sens commun à partir de sources inhabituelles. Ensuite, nous avons apporté deux solutions nouvelles et pratiques au problème de la réécriture de requête.

Nous espérons que nos recherches vont inspirer d'autres travaux et permettront des avancées majeures en informatique. Pour cela, tous nos résultats, codes et données sont accessibles en open source afin que la communauté puisse en profiter.

Appendix B

Additional Proofs

B.1 Proofs for Section 5.6

Let us first define some notions used throughout this appendix. Recall the definition of a *path query* (Definition 5.6.4) and of its skeleton. We sometimes abbreviate the body of the query (i.e. the part containing the path of relations) as $r_1...r_n(\alpha, x_1...x_n)$. We use the expression *path query with a filter* to refer to a path query where a body variable other than α is replaced by a constant. For example, in Figure 5.1, we can have the path query:

$$q(m, a) \leftarrow \text{sang}(m, s), \text{onAlbum}(s, a)$$

where m, a and s are variables representing respectively the singer, the album and the song. which asks for the singers with their albums. Its skeleton is *sang.onAlbum*.

Towards characterizing the path queries that can serve as a rewriting, it will be essential to study *loop queries*:

Definition B.1.1 (Loop Query)**.** *We call* ***loop query*** *a query of the form:* $r_1...r_n(a, a) \leftarrow r_1(a, x_1)...r_n(x_{n-1}, a)$ *where* a *is a constant and* $x_1, x_2, ..., x_{n-1}$ *are variables such that* $x_i = x_j \Leftrightarrow i = j$.

With these definitions, we can show the results claimed in Section 5.6.

B.1.1 Proof of Theorem 5.6.2

Theorem 5.6.2. *Given a query* $q(x) \leftarrow r(a, x)$*, a well-filtering plan* π_a*, the associated minimal filtering plan* π_a^{min} *and unary inclusion dependencies* $\mathcal{UID}$*:*

- *If* π_a^{min} *is not equivalent to* q*, then neither is* π_a*.*
- *If* π_a^{min} *is equivalent to* q*, then we can determine in polynomial time if* π_a *is equivalent to* π_a^{min}

First, let us show the first point. By Definition 5.4.5, we have that π_a contains $r(a, x)$ or $r^-(x, a)$. Let us suppose that π_a is equivalent to q. Let I be the database obtained by taking the one fact $r(a, b)$ and chasing by $\mathcal{UID}$. We know that the semantics of π_a has a binding returning b as an answer. We first argue that π_a^{min}

also returns this answer. As π_a^{min} is formed by removing all filters from π_a and then adding possibly a single filter, we only have to show this for the case where we have indeed added a filter. But then by definition, the added filter ensures that π_a^{min} is well-filtering. Therefore, it creates an atom $r(a, x)$ or $r^-(x, a)$ in the semantics of π_a^{min} and the binding of the semantics of π_a that maps the output variable to b is also a binding of π_a^{min}.

We then prove that π_a^{min} does not return any other answer. In the first case, as π_a^{min} is well-filtering, it cannot return any different answer than b on I. In the second case, we know by the explanation after Definition 5.4.5 that π_a^{min} is also well-filtering, so the same argument applies. Hence, π_a^{min} is also equivalent to q, which establishes the first point.

Let us now show the more challenging second point. We assume that π_a^{min} is equivalent to q. Recall the definition of a loop query (Definition B.1.1) and the grammar $\mathcal{G}_q$ defined in Definition 5.5.1, whose language we denoted as $\mathcal{L}_q$. We first show the following property:

Property B.1.2. *A loop query $r_1...r_n(a, a)$ is true on all database instances satisfying the unary inclusion dependencies $\mathcal{UID}$ and containing a tuple $r(a, b)$, iff there is a derivation tree in the grammar such that $B_r \xrightarrow{*} r_1...r_n$.*

Proof. We first show the backward direction. The proof is by structural induction on the length of the derivation. The length of the derivations is the number of rules applied to get a word. We first show the base case. If the length is 0, its derivation necessarily uses Rule 5.5.4, and the query $\epsilon(a, a)$ is indeed true on all databases.

We now show the induction step. Suppose we have the result for all derivations up to a length of $n-1$. We consider a derivation of length $n > 0$. Let I be a database instance satisfying the inclusion dependencies $\mathcal{UID}$ and containing the fact $r(a, b)$. Let us consider the rule at the root of the derivation tree. It can only be Rule 5.5.3. Indeed, Rule 5.5.4 only generates words of length 0. So, the first rule applied was Rule 5.5.3 $B_r \to B_r L_{r_i}$ for a given UID $r \rightsquigarrow r_i$. Then we have two cases.

The first case is when the next B_r does not derive ϵ in the derivation that we study. Then, there exists $i \in \{2, \dots, n-1\}$ such that $B_r \xrightarrow{*} r_1 \dots r_{i-1}$ and $L_{r_i} \xrightarrow{*} r_i \dots r_n$ (L_{r_i} starts by r_i). From the induction hypothesis we have that $r_1 \dots r_{i-1}(a, a)$ has an embedding in I. Note that the derivation tree to derive $B_r \xrightarrow{*} r_1 \dots r_{i-1}$ is at least of length 3. Indeed, we need to do at least a call to Rules 5.5.3, 5.5.4 and 5.5.5. So, the length of the derivation of $L_{r_i} \xrightarrow{*} r_i \dots r_n$ is less than $n-3$. Also, $r_i \dots r_n(a, a)$ is true on I if $B_{r_i} \xrightarrow{*} r_i \dots r_n$. Indeed, $B_{r_i} \xrightarrow{Rule 5.5.3} B_{r_i} L_{r_i} \xrightarrow{Rule 5.5.4} L_{r_i} \xrightarrow{*} r_i \dots r_n$. This derivation is of length less than $n-1$. This shows the first case of the induction step.

We now consider the case where the next B_{r_i} derives ϵ. Note that, as $r(a, b) \in I$, there exists c such that $r_i(a, c) \in I$. The next rule in the derivation is $L_{r_i} \to r_i B_{r_i^-} r_i^-$, then $B_{r_i^-} \xrightarrow{*} r_2 \dots r_{n-1}$, and $r_1 = r_i$ and $r_n = r_i^-$. By applying the induction hypothesis, we have that $r_2 \dots r_{n-1}(c, c)$ has an embedding in I. Now, given that $r_1(a, c) \in I$ and $r_n(c, a) \in I$ we can conclude that $r_1 \dots r_n(a, a)$ has an embedding in I. This establishes the first case of the induction step. Hence, by induction, we have shown the backward direction of the proof.

We now show the forward direction. Let I_0 be the database containing the single fact $r(a, b)$ and consider the database I_0^* obtained by chasing the fact $r(a, b)$ by the inclusion dependencies in $\mathcal{UID}$, creating a new null to instantiate every missing fact. This database is generally infinite, and we consider a tree structure on its domain, where the root is the element a, the parent of b is a, and the parent of every null x is the element that occurs together with x in the fact where x was introduced. Now, it is a well-known fact of database theory [1] that a query is true on every superinstance of I_0 satisfying $\mathcal{UID}$ iff that query is true on the chase I_0^* of I_0 by $\mathcal{UID}$. Hence, let us show that all loop queries $r_1 \dots r_n(a, a)$, which hold in I_0^* are the ones that can be derived from B_r.

We show the claim again by induction on the length of the loop query. More precisely, we want to show that, for all relation $r \in \mathcal{R}$, for all constants a and b, for all $n \geq 0$, for all loop queries $r_1 ... r_n(a, a)$ which is true on all database instances satisfying the unary inclusion dependencies $\mathcal{UID}$ and containing a tuple $r(a, b)$, we have:

1. $B_r \xrightarrow{*} r_1 ... r_n$
2. For a match of the loop query on I_0^*, if no other variable than the first and the last are equal to a, then we have: $L_{r_1} \xrightarrow{*} r_1 ... r_n$

In this proof, we index I_0^* by the first relation used to generate it. Thus, $I^*_{r,a,b}$ is the database obtained by chasing a fact $r(a, b)$.

Let $r \in \mathcal{R}$ and a and b be two constants. If the length of the loop query is 0, then it could have been derived by the Rule 5.5.4. The length of the loop query cannot be 1 as for all relations r', the query $r'(a, a)$ is not true on all databases satisfying the UIDs and containing a tuple $r(a, b)$ (for example it is not true on $I^*_{r,a,b}$).

Let us suppose the length of the loop query is 2 and let us write the loop query as $r_1(a, x), r_2(x, a)$ and let $r_1(a, c), r_2(c, a)$ be a match on $I^*_{r,a,b}$. The fact $r_1(a, c)$ can exist on $I^*_{r,a,b}$ iff $r \rightsquigarrow r_1$. In addition, due to the tree structure of $I^*_{r,a,b}$, we must have $r_2 = r_1^-$. So, we have $B_r \to L_{r_1} \to r_1 B_{r_1^-} r_1^- \to r_1^-$ and we have shown the two points of the inductive claim.

We now suppose that the result is correct up to a length $n - 1$ ($n > 2$), and we want to prove that it is also true for a loop query of length n.

Let $r \in \mathcal{R}$ and a and b be two constants. Consider a match $r_1(a, a_1), r_2(a_1, a_2), \dots, r_{n-1}(a_{n-2}, a_{n-1}), r_n(a_{n-1}, a)$ of the loop query on $I^*_{r,a,b}$. Either there is some i such that $a_i = a$, or there is none.

If there is at least one, then let us cut the query at all positions where the value of the constant is a. We write the binding of the loop queries on $I^*_{r,a,b}$: $(r_{i_0} \dots r_{i_1})(a, a).(r_{i_1+1} \dots r_{i_2})(a, a)...(r_{i_{k-1}+1} \dots r_{i_k})(a, a)$ (where $1 = i_0 < i_1 < ... < i_{k-1} < i_k = n$). As we are on $I^*_{r,a,b}$, we must have, for all $0 < j < k$, that $r \rightsquigarrow r_{i_j}$ (by construction of $I^*_{r,a,b}$). So, we can construct the derivation : $B_r \to B_r L_{r_{i_{k-1}}} \to B_r L_{r_{i_{k-2}}} L_{r_{i_{k-1}}} \xrightarrow{*} L_{r_{i_0}} \dots L_{r_{i_{k-1}}}$. We notice that, for all $0 \leq j < k$, we have $r_{i_j} ... r_{i_{j+1}}(a, a)$ true on all database instances where $r(a, b)$ is true as it is true on the canonical database. Then, from the induction hypothesis, we have that, for all $0 < j < k$, $L_{r_{i_j}} \xrightarrow{*} r_{i_j} ... r_{i_{j+1}}$ and so we get the first point of our induction hypothesis.

We now suppose that there is no i such that $a_i = a$. Then, we have $r \rightsquigarrow r_1$ (by construction of $I^*_{r,a,b}$). In addition, due to the tree structure of $I^*_{r,a,b}$, we must have $r_n = r_1^-$ and $a_1 = a_{n-1}$. We can then apply the induction hypothesis on $r_2...r_{n-1}(a_1, a_1)$, if it is true on all databases satisfying the unary inclusion dependencies $\mathcal{UID}$ and containing a tuple $r_1^-(a_1, a)$, then $B_{r_1^-} \xrightarrow{*} r_2...r_{n-1}$. To notice that $r_2...r_{n-1}(a_1, a_1)$ is indeed true on all databases satisfying the unary inclusion dependencies $\mathcal{UID}$ and containing a tuple $r_1^-(a_1, a)$, we notice that $I^*_{r,a,b}$ contains $I^*_{r_1,a,a_1}$ (up to a renaming of the variables), and that the constant a is never used. So, $r_2...r_{n-1}(a_1, a_1)$ is true on the canonical database and therefore on all databases satisfying the unary inclusion dependencies $\mathcal{UID}$ and containing a tuple $r_1^-(a_1, a)$. Finally, we observe that we have the derivation $B_r \xrightarrow{*} L_{r_1} \to r_1 B_{r_1^-} r_1^- \xrightarrow{*} r_1...r_n$ and so we have shown the two points of the inductive claim.

Thus, we have established the forward direction by induction, and it completes the proof of the claimed equivalence.

□

Next, to determine in polynomial time whether π_a is equivalent to π_a^{min} (and hence q), we are going to consider all positions where a filter can be added. To do so, we need to define the *root path* of a filter:

Definition B.1.3 (Root Path)**.** *Let π_a be well-filtered execution plan. Given a filter, there is a unique path query $r_1(a, x_1)...r_n(x_{n-1}, a)$ in the semantics of π_a, starting from the constant a and ending at the filter (which value is also a). We call this path the* root path of the filter.

The existence and uniqueness come from arguments similar to Property 5.4.2: we can extract a sequence of calls to generate the filter and then, from the semantics of this sequence of calls, we can extract the root path of the filter.

This definition allows us to characterise in which case the well-filtering plan π_a is equivalent to its minimal filtering plan π_a^{min}, which we state and prove as the following lemma:

Lemma B.1.4. *Let $q(x) \leftarrow r(a, x)$ be an atomic query, let $\mathcal{UID}$ be a set of UIDs, and let π_a^{min} be a minimal filtering plan equivalent to q under $\mathcal{UID}$. Then, for any well-filtering plan π_a defined for q, the plan π_a is equivalent to π_a^{min} iff for each filter and its associated root path $r_1(a, x_1)...r_n(x_{n-1}, a)$, there is a derivation tree such that $B_r \xrightarrow{*} r_1...r_n$ in the grammar $\mathcal{G}_q$.*

It is easy to show the second point of Theorem 5.6.2 once we have the lemma. We have a linear number of filters, and, for each of them, we can determine in PTIME if B_r generates the root path. So, the characterisation can be checked in PTIME over all filters, which allows us to know if π_a is equivalent to π_a^{min} in PTIME, as claimed.

Hence, all that remains to do in this appendix section to establish Theorem 5.6.2 is to prove Lemma B.1.4. We now do so:

Proof. We consider a filter and the root query $r_1(a, x_1)...r_n(x_{n-1}, a)$ obtained from its root path.

We first show the forward direction. Let us assume that π_a is equivalent to π_a^{min}. Then, π_a is equivalent to q, meaning that the loop query $r_1...r_n(a, a)$ is true on

all database instances satisfying the unary inclusion dependencies and containing a tuple $r(a, b)$. So, thanks to Property B.1.2, we conclude that $B_r \xrightarrow{*} r_1...r_n$.

We now show the more challenging backward direction. Assume that, for all loop queries $r_1...r_n(a, a)$ equal to the root path of each filter, there is a derivation tree such that $B_r \xrightarrow{*} r_1...r_n$. We must show that π_a^{min} is equivalent to π_a, i.e., it is also equivalent to q. Now, we know that π_a^{min} contains an atom $r(a, x)$ or $r^-(x, a)$, so all results that it returns must be correct. All that we need to show is that it returns all the correct results. It suffices to show this on the canonical database: let I be the instance obtained by chasing the fact $r(a, b)$ by the unary inclusion dependencies. As π_a^{min} is equivalent to q, we know that it returns b, and we must show that π_a also does. We will do this using the observation that all path queries have at most one binding on the canonical database, which follows from Property 5.4.4.

Let us call $\pi_a^{\text{no filter}}$ the execution plan obtained by removing all filters from π_a. As we have $B_r \xrightarrow{*} r_1...r_n$ for all root paths, we know from Property B.1.2 that $r_1...r_n(a, a)$ is true on all databases satisfying the UIDs, and in particular on I. In addition, on I, $r_1...r_n(a, x_1, ...x_n)$ has only one binding, which is the same than $r_1...r_n(a, x_1, ...x_{n-1}, a)$. So, the filters of π_a do not remove any result of π_a on I relative to $\pi_a^{\text{no filter}}$: as the reverse inclusion is obvious, we conclude that π_a is equivalent to $\pi_a^{\text{no filter}}$ on I.

Now, if π_a^{min} contains no filter or contains a filter which was in π_a, we can apply the same reasoning and we get that π_a^{min} is equivalent to $\pi_a^{\text{no filter}}$ on I, and so π_a^{min} and π_a are equivalent in general.

The only remaining case is when π_a^{min} contains a filter which is not in π_a. In this case, we have that the semantics of π_a contains two consecutive atoms $r(a, x)r^-(x, y)$ where one could have filtered on y with a (this is what is done in π_a^{min}). Let us consider the root path of π to y. It is of the form $r_1...r_n(a, a)r(a, x)r^-(x, y)$. We have $B_r \xrightarrow{*} r_1...r_n$ by hypothesis. In addition, as $r \rightsquigarrow r$ trivially, we get $B_r \to B_rL_r \to B_rrr^- \xrightarrow{*} r_1...r_n.r.r^-$. So, $r_1...r_n.r.r^-(a, a)$ is true on I (Property B.1.2). Using the same reasoning as before, π_a^{min} is equivalent to $\pi_a^{\text{no filter}}$ on I, and so π_a^{min} and π_a are equivalent in general. This concludes the proof.

□

B.1.2 Proof of Property 5.6.9

We show that we can effectively reverse the path transformation, which will be crucial to our algorithm:

Property 5.6.9. *Given a word w in $\mathcal{R}^*$, a query $q(x) \leftarrow r(a, x)$ and a set of path functions, it is possible to know in polynomial time if there exists a non-redundant minimal filtering execution plan π_a such that $\mathcal{P}(\pi_a) = w$. Moreover, if such a π_a exists, we can compute one in polynomial time, and we can also enumerate all of them (there are only finitely many of them).*

We are going to construct a finite-state transducer that can reverse the path transformation and give us a sequence of calls. To find one witnessing plan, it will suffice to take one run of this transducer and take the corresponding plan, adding

a specific filter which we know is correct. If we want all witnessing plans, we can simply take all possible outputs of the transducer.

To construct the transducer, we are going to use the regular expression P_r from Definition 5.5.2. We know that P_r faithfully represents plans (Theorem 5.6.13), and it is a regular expression. So we will be able to build an automaton from P_r on which we are going to add outputs to represent the plans.

The start node of our transducer is S, and the final node is F. The input alphabet of our transducer is $\mathcal{R}$, the set of relations. The output alphabet is composed of function names f for $f \in \mathcal{F}$, the set of path functions, and of output symbols OUT_i, which represents the used output of a given function. We explain later how to transform an output word into a non-redundant minimal filtering plan.

First, we begin by creating chains of letters from the $w_{f,i}$ defined in Definition 5.5.2. For a word $w_{f,i} = r_1...r_k$ (which includes the reverse atoms added at the end when $0 \leq i < n$), this chain reads the word $r_1...r_k$ and outputs nothing.

Next, we construct W_0 between two nodes representing the beginning and the end of W_0: S_{W_0} and F_{W_0}. From S_{W_0} we can go to the start of the chain of a final $w_{f,0}$ by reading an epsilon symbol and by outputting the function name f. Then, at the end of the chain of a final $w_{f,0}$, we go to F_{W_0} by reading an epsilon symbol and by outputting a OUT_1 letter.

Similarly, we construct W' between two nodes representing the beginning and the end of W': $S_{W'}$ and $F_{W'}$. From $S_{W'}$ we can go to the beginning of the chain of a final $w_{f,i}$ with $0 < i < n$ (as explained in Definition 5.5.2) by reading an epsilon symbol and by outputting the function name f. Then, at the end of the chain of a final $w_{f,i}$, we go to $F_{W'}$ by reading an epsilon symbol. The output symbol of the last transition depends on the last letter of $w_{f,i}$: if it is r, then we output OUT_i; otherwise, we output OUT_{i+1}. This difference appears because we want to create a last atom $r(a, x)$ or $r^-(x, a)$, and so our choice of output variable depends on which relation symbol we have.

Last, using the same method again, we construct W between two nodes representing the beginning and the end of W: S_W and F_W. From S_W we can go to the beginning of the chain of a $w_{f,i}$ with $0 < i \leq n$ (as explained in Definition 5.5.2) by reading an epsilon symbol and by outputting the function name f. Then, at the end of the chain of a final $w_{f,i}$, we go to $F_{W'}$ by reading an epsilon symbol and outputting OUT_i. In this situation, there is no ambiguity on where the output variable is.

Finally, we can link everything together with epsilon transitions that output nothing. We construct W^* thanks to epsilon transitions between S_W and F_W. Then, W^*W' is obtained by linking F_W to $S_{W'}$ with an epsilon transition. We can now construct $P_r = W_0|(W^*W')$ by adding an epsilon transition between S and S_{W_0}, S and S_W, F_{W_0} and F and $F_{W'}$ and F.

We obtain a transducer that we call $\mathcal{T}_{reverse}$.

Let w be a word of $\mathcal{R}^*$. To know if there is a non-redundant minimal filtering execution plan π_a such that $\mathcal{P}(\pi_a) = w$, one must give w as input to $\mathcal{T}_{reverse}$. If there is no output, there is no such plan π_a. Otherwise, $\mathcal{T}_{reverse}$ nondeterministically outputs some words composed of an alternation of function symbols f and output symbols OUT_i. From this representation, one can easily reconstruct the execution plan: The function calls are the f from the output word and the previous OUT

symbol gives their input. If there is no previous OUT symbol (i.e., for the first function call), the input is a. If the previous OUT symbol is OUT_k, then the input is the second argument of the k^{th} atom from the body. The last OUT symbol gives us the output of the plan. We finally add a filter with a on the constructed plan to get an atom $r(a, x)$ or $r^-(x, a)$ in its semantics in the last possible atom, to obtain a minimal filtering plan. Note that this transformation is related to the one given in the proof of Theorem 5.6.13 in Section B.1.4, where it is presented in a more detailed way.

Using the same procedure, one can enumerate all possible output words for a given input and then obtain all non-redundant minimal filtering execution plans π_a such that $\mathcal{P}(\pi_a) = w$. We can understand this from the proof of Theorem 5.6.13 in Section B.1.4, which shows that there is a direct match between the representation of w as words of $w_{f,i}$ and the function calls in the corresponding execution plan. Last, the reason why the set of output words is finite is because the transducer must at least read an input symbol to generate each output symbol.

B.1.3 Proof of Theorem 5.6.11

In this appendix, we finally show the main theorem of Section 5.6:

Theorem 5.6.11. *Given a set of unary inclusion dependencies, a set of path functions, and an atomic query q, the language $\mathcal{L}_q$ captures q.*

Recall that $\mathcal{L}_q$ is the language of the context-free grammar $\mathcal{G}_q$ from Definition 5.5.1. Our goal is to show that it is a capturing language.

In what follows, we say that two queries are *equivalent* under a set of UIDs if they have the same results on all databases satisfying the UIDs.

Linking $\mathcal{L}_q$ to equivalent rewritings.

In this part, we are going to work at the level of the words of $\mathcal{R}^*$ ending by a r or r^- (in the case $q(x) \leftarrow r(a, x)$ is the considered query), where $\mathcal{R}$ is the set of relations. Recall that the full path transformation (Definition 5.6.6) transforms an execution plan into a word of $\mathcal{R}^*$ ending by an atom r or r^-. Our idea is first to define which words of $\mathcal{R}^*$ are interesting and should be considered. In the next part, we are going to work at the level of functions.

For now, we start by defining what we consider to be the "canonical" path query associated to a skeleton. Indeed, from a skeleton in $\mathcal{R}^*$ where $\mathcal{R}$ is the set of relations, it is not clear what is the associated path query (Definition 5.6.4) as there might be filters. So, we define:

Definition B.1.5 (Minimal Filtering Path Query)**.** *Given an atomic query $q(a, x) \leftarrow r(a, x)$, a set of relations $\mathcal{R}$ and a word $w \in \mathcal{R}^*$ of relation names from $\mathcal{R}$ ending by r or r^-, the **minimal filtering path query** of w for q is the path query of skeleton w taking as input a and having a filter such that its last atom is either $r(a, x)$ or $r^-(x, a)$, where x is the only output variable.*

As an example, consider the word $onAlbum.onAlbum^-.sang^-$. The minimal filtering path query is: $q'(Jailhouse, x) \leftarrow onAlbum(Jailhouse,y), onAlbum^-(y,$

Jailhouse), $sang^-(Jailhouse, x)$, which is an equivalent rewriting of the atomic query $sang^-(Jailhouse, x)$.

We can link the language $\mathcal{L}_q$ of our context-free grammar to the equivalent rewritings by introducing a corollary of Property B.1.2:

Corollary B.1.6. *Given an atomic query $q(a, x) \leftarrow r(a, x)$ and a set $\mathcal{UID}$ of UIDs, the minimal filtering path query of any word in $\mathcal{L}_q$ is a equivalent to q. Reciprocally, for any query equivalent to q that could be the minimal filtering path query of a path query of skeleton w ending by r or r^-, we have that $w \in \mathcal{L}_q$.*

Notice that the minimal filtering path query of a word in $\mathcal{L}_q$ is well defined as all the words in this language end by r or r^-.

Proof. We first suppose that we have a word $w \in \mathcal{L}_q$. We want to show that the minimal filtering path query of w is equivalent to q. We remark that the minimal filtering path query contains the atom $r(a, x)$ or $r^-(x, a)$. Hence, the answers of the given query always include the answers of the minimal filtering path query, and we only need to show the converse direction.

Let I be a database instance satisfying the inclusion dependencies $\mathcal{UID}$ and let $r(a, b) \in I$ (we suppose such an atom exists, otherwise the result is vacuous). Let $q'(a, x)$ be the head atom of the minimal filtering path query. It is sufficient to show that $q'(a, b)$ is an answer of the minimal filtering path query to prove the equivalence. We proceed by structural induction. Let $w \in \mathcal{L}_q$. Let us consider a bottom-up construction of the word. The last rule can only be one of the Rule 5.5.1 or the Rule 5.5.2. If it is Rule 5.5.1, then $\exists r_1, \ldots, r_n \in \mathcal{R}$ such that $w = r_1 \ldots r_n r$ and $B_r \xrightarrow{*} r_1 \ldots r_n$. By applying Property B.1.2, we know that $r_1 \ldots r_n(a, a)$ has an embedding in I. Hence, $q'(a, b)$ is an answer. If the rule is Rule 5.5.2, then $\exists r_1, \ldots, r_n, \ldots, r_m \in \mathcal{R}$ such that $w = r_1 \ldots r_n r r_{n+1} \ldots r_m r^-$, $B_r \xrightarrow{*} r_1 \ldots r_n$ and $B_{r^-} \xrightarrow{*} r_{n+1} \ldots r_m$. By applying Property B.1.2 for the two derivations, and remembering that we have $r(a, b)$ and $r^-(b, a)$ in I, we have that $r_1 \ldots r_n(a, a)$ and $r_{n+1} \ldots r_m(b, b)$ have an embedding in I. Hence, also in this case, $q'(a, b)$ is an answer. We conclude that q' is equivalent to q.

Reciprocally, let us suppose that we have a minimal filtering path query of a path query of skeleton w, which is equivalent to q, and that $q'(a, x)$ is its head atom. We can write it either $q'(a, x) \leftarrow r_1(a, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, a) r(a, x)$ or $q'(a, x) \leftarrow r_1(a, x_1), r_2(x_1, x_2), ..., r_n(x_{n-1}, x), r^-(x, a)$. In the first case, as q' is equivalent to q, we have $r_1 ... r_n(a, a)$ which is true on all databases I such that I contains a tuple $r(a, b)$. So, according to Property B.1.2, $B_r \xrightarrow{*} r_1 ... r_n$, and using Rule 5.5.1, we conclude that $r_1 ... r_n.r$ is in $\mathcal{L}_q$. In the second case, for similar reasons, we have $B_r \xrightarrow{*} r_1 ... r_n r^-$. The last r^- was generated by Rule 5.5.5, using a non-terminal L_r which came from Rule 5.5.3 using the trivial UID $r \rightsquigarrow r$. So we have $B_r \rightarrow B_r L_r \rightarrow B_r r B_{r^-} r^- \xrightarrow{*} r_1 ... r_n r^-$. We recognise here Rule 5.5.2 and so $r_1 ... r_n r^- \in \mathcal{L}_q$. This shows the second direction of the equivalence, and concludes the proof. □

Linking the path transformation to $\mathcal{L}_q$.

In the previous part, we have shown how equivalent queries relate to the context-free grammar $\mathcal{G}_q$ in the case of minimal filtering path queries. We are now going to show how the full path transformation relates to the language $\mathcal{L}_q$ of $\mathcal{G}_q$, and more precisely, we will observe that the path transformation leads to a minimal filtering path query of a word in $\mathcal{L}_q$.

The path transformation operates at the level of the semantics for each function call, transforming the original tree-shaped semantics into a word. What we want to show is that after the path transformation, we obtain a minimal filtering path query equivalent to q iff the original execution path was an equivalent rewriting. To show this, we begin by a lemma:

Lemma B.1.7. *Let π_a a minimal filtering non-redundant execution plan. The query $\mathcal{P}'(\pi_a)(x)$ is a minimal filtering path query and it ends either by $r(a, x)$ or $r^-(x, a)$, where x was the variable name of the output of π_a.*

Proof. By construction, $\mathcal{P}'(\pi_a)(x)$ is a minimal filtering path query. Let us consider its last atom. In the case where the original filter on the constant a created an atom $r(a, x)$, then the result is clear: an atom $r^-(x, a)$ is added. Otherwise, it means the original filter created an atom $r^-(x, a)$. Therefore, as observed in the last point of the path transformation, the last atom is $r(a, x)$, where we created the new filter on a. □

The property about the preservation of the equivalence is expressed more formally by the following property:

Property B.1.8. *Let us consider a query $q(x) \leftarrow r(a, x)$, a set of unary inclusion dependencies $\mathcal{UID}$, a set of path functions $\mathcal{F}$ and a minimal filtering non-redundant execution plan π_a constructed on $\mathcal{F}$. Then, π_a is equivalent to q iff the minimal filtering path query $\mathcal{P}'(\pi_a)$ is equivalent to q.*

Proof. First, we notice that we have $\pi_a(I) \subseteq q(I)$ and $\mathcal{P}'(\pi_a)(I) \subseteq q(I)$ as $r(a, x)$ or $r^-(a, x)$ appear in the semantics of $\pi_a(x)$ and in $\mathcal{P}'(\pi_a)(x)$ (see Lemma B.1.7). So, it is sufficient to prove the property on the canonical database I_0 obtained by chasing the single fact $r(a, b)$ with $\mathcal{UID}$.

We first show the forward direction and suppose that π_a is equivalent to q. Then, its semantics has a single binding on I_0. Let us consider the i^{th} call $c_i(\underline{x_1}, ..., x_j, ...x_n)$ (x_j is the output used as input by another function or the output of the plan) in π_a and its binding: $r_1(y_1, y_2), ..., r_k(y_k, y_{k+1}), ..., r_m(y_m, y_{m+1})$. Then $r_1(y_1, y_2)$, ..., $r_{k-1}(y_{k-1}, y_k), r_k(y_k, y_{k+1})$, ..., $r_m(y_m, y_{m+1}), r_m^-(y_{m+1}, y_m)$, ..., $r_k^-(y_{k+1}, y_k)$ is a valid binding for the sub-semantics, as the reversed atoms can be matched to the same atoms than those used to match the corresponding forward atoms. Notice that the last variable is unchanged. So, in particular, the variable named x (the output of π_a) has at least a binding in I_0 before at step 3 of the path transformation. In step 4, we have two cases. In the first case, we add $r^-(x, a)$ to the path semantics. Then we still have the same binding for x as π_a is equivalent to q. In the second case, we added a filter in the path semantics, and we still get the same binding. Indeed, by Property 5.4.4, b has a single ingoing r-fact in I_0, which is $r^-(b, a)$. This observation

means that, in the binding of the path semantics, the penultimate variable was necessary a on I_0. We conclude that $\mathcal{P}'(\pi_a)$ is also equivalent to q.

We now show the backward direction and suppose $\mathcal{P}'(\pi_a)$ is equivalent to q. Let us take the single binding of $\mathcal{P}'(\pi_a)$ on I_0. Let us consider the sub-semantics of a function call $c_i(\underline{x_1}, ..., x_j, ...x_n)$ (where x_j is the output used as input by another function or the output of the plan): $r_1(y_1, y_2)$, ..., $r_{k-1}(y_{k-1}, y'_k), r_k(y'_k, y'_{k+1})$, ..., $r_m(y'_m, y_{m+1}), r_m^-(y_{m+1}, y_m)$, ..., $r_k^-(y_{k+1}, y_k)$. As all path queries have at most one binding on I_0, we necessarily have $y_k = y'_k$, ..., $y_m = y'_m$. Thus the semantics of π_a has a binding on I which uses the same values than the binding of $\mathcal{P}'(\pi_a)$. In particular, the output variables have the same binding. We conclude that π_a is also equivalent to q. □

We can now apply Corollary B.1.6 on $\mathcal{P}'(\pi_a)$ as it is a minimal filtering path query : $\mathcal{P}'(\pi_a)$ is equivalent to q iff $\mathcal{P}(\pi_a)$ is in $\mathcal{L}_q$. So, π_a is equivalent to q iff $\mathcal{P}(\pi_a)$ is in $\mathcal{L}_q$.

We conclude that $\mathcal{L}_q$ is a capturing language for q. As our grammar $\mathcal{G}_q$ for $\mathcal{L}_q$ can be constructed in PTIME, this concludes the proof of Theorem 5.6.11.

B.1.4 Proof of Theorem 5.6.13

Theorem 5.6.13. *Let $\mathcal{F}$ be a set of path functions, let $q(x) \leftarrow r(a, x)$ be an atomic query, and define the regular expression P_r as in Definition 5.5.2. Then the language of P_r faithfully represents plans.*

Proof. We first prove the forward direction: every word w of the language of P_r is achieved as the image by the full path transformation of a minimal filtering non-redundant plan. To show this, let w be a word in the language of P_r. We first suppose we can decompose it into its elements from W and W': $w = w_{f_1,i_1}...w_{f_{n-1},i_{n-1}}.w_{f_n,i_n}$ with w_{f_n,i_n} being final. Let π_a be composed of the successive function calls $f_1, \ldots, f_n$ where the input of f_1 is the constant a and the input of f_k ($k > 1$) is the i_{k-1}^{th} variable of f_{k-1}. For the output variable and the filter, we have two cases:

1. If the last letter of w_{f_n,i_n} is r, the output of the plan is the i_n^{th} variable of f_n and we add a filter to a on the $(i_n + 1)^{th}$ variable.

2. Otherwise, if the last letter of w_{f_n,i_n} is r^-, the output of the plan is the $(i_n+1)^{th}$ variable of f_n and we add a filter to a on the i_n^{th} variable (except if this is the input, in which case we do nothing).

We notice that π_a is non-redundant. Indeed, by construction, only the first function takes a as input, and all functions have an output used as input in another function. The added filter cannot be on the input of a function as $i_n > 0$. What is more, π_a is also a minimal filtering plan. Indeed, by construction, we create an atom $r(a, x)$ or an atom $r^-(x, a)$ (with x the output of the plan). Let us show that it is the last possible filter. If we created $r^-(x, a)$, it is obvious as x cannot be used after that atom. If we created $r(a, x)$, we know we could not have a following atom $r^-(x, y)$ where one could have filtered on y: this is what is guaranteed by the third point of the definition of the final $w_{f,i}$.

The only remaining point is to show that $\mathcal{P}(\pi_a) = w$. Indeed, for $k < n$, we notice that w_{f_k,i_k} is the skeleton of the sub-semantics of the k^{th} call in π_a. What is less intuitive is what happens for the last function call.

Let us consider the two cases above. In the first one, the output variable is the i_n^{th} variable of f_n. We call it x. The semantics of $\pi_a(x)$ contains $r_{i_n+1}(x, a) = r^-(x, a)$ (as the last letter of w_{f_n,i_n} is r). We are in the second point of step 4 of the path transformation. The skeleton of the end of the path semantics is not modified and it is w_{f_n,i_n}.

In the second case, the output variable is the $(i_n + 1)^{th}$ variable of f_n. We call it x. The semantics of $\pi_a(x)$ contains $r_{i_n+1}(a, x) = r(a, x)$ (as the last letter of w_{f_n,i_n} is r^-). We are in the first point of step 4 of the path transformation. The skeleton of the end of the path semantics is modified to append r^- and it is now $w_{f_n,i_n+1}r = w_{f_n,i_n}$ as expected.

This establishes that $\mathcal{P}(\pi_a) = w$ in the case where w can be decomposed as elements of W and W'. Otherwise, w is in the language of W_0, so $w = w_{f,0}$ where $w_{f,0}$ is final, ends by r^-, thus starts by r. We define π_a as the execution plan composed of one function call f, which takes as input a. The output of the plan is the first output variable of the function. The plan π_a is non-redundant as it contains only one function call. It is also minimal filtering. Indeed, by definition of $w_{f,0}$, the first output variable is on the first atom. So, the semantics of π_a contains an atom $r(a, x)$ where x is the output variable. Besides, it does not contain an atom $r^-(x, y)$ where y is an output of the f by the third point of the definition of a final $w_{f,i}$.

Finally, we have $\mathcal{P}(\pi_a) = w$, and this concludes the first implication. The transformation that we have here is what was performed by the transducer and the method presented in Section B.1.2. The only difference is that the technique in Section B.1.2 will consider all possible ways to decompose w into $w_{f,i}$ and into final $w_{f,i}$ to get all possible non-redundant minimal filtering plans.

We now show the converse direction of the claim: the full path transformation maps non-redundant minimal filtering plans to words in the language of P_r. Suppose that we have a non-redundant minimal filtering plan π_a such that $\mathcal{P}(\pi_a) = w$ and let us show that w is in the language of P_r. For all calls which are not the last one, it is clear that the sub-semantics of these calls are the w_{f,i_k} with $i_k > 0$ (as the plan is non-redundant). So the words generated by the calls that are not the last call are words of the language of W^*.

For the last function call, we have several cases to consider.

First, if $\pi_a(x)$ contains a filter, then it means that either the first atom in the semantics of $\pi_a(x)$ is not $r(a, x)$ or, if it is, it is followed by an atom $r^-(x, a)$.

If we are in the situation where the semantics of $\pi_a(x)$ starts by $r(a, x), r^-(x, a)$, then π_a is composed of only one function call f (otherwise, it would be redundant). Then, it is clear that $\mathcal{P}(\pi_a) = w_{f,1}$ with $w_{f,1}$ being final, and we have the correct form.

If we are in the situation where the semantics of $\pi_a(x)$ does not start by $r(a, x)$, we have the two cases (corresponding to the two cases of the forward transformation). We suppose that the last function call is on f, and the output variable is the i^{th} one in f.

If π_a does not contain an atom $r(a, x)$, then it contains an atom $r^-(x, a)$ and

the result is clear: the skeleton of the path semantics is not modified and ends by the sub-semantics of f whose skeleton is $w_{f,i}$ and has the correct properties: the last atom is r, the variable after x is not existential (it is used to filter) and the atom after $r(a, x)$ cannot be $r^-(x, y)$ with y an output variable of f as π_a is minimal filtering.

If π_a contains an atom $r(a, x)$, then in the definition of the path transformation, we append an atom $r^-(x, a)$ after the sub-semantics of the f. We then have the path semantics ending by the atom names $w_{f,i}.r^- = w_{f,i-1}$ and $w_{f,i-1}$, which is final, has the adequate properties.

This shows that w is in the language of P_r in the case π_a has a filter because the word generated by the last call is in W'.

Now, we consider the case when π_a does not have a filter. It means that the semantics of π_a starts by $r(a, x)$ and is not followed by an atom $r^-(x, y)$ where y is the output of a function (as π_a is well-filtering). Then, π_a is composed of only one function call f and it is clear that $\mathcal{P}(\pi_a) = w_{f,0}$ which is final. So, in this case, the word w belongs to W_0.

So, w is in the language of P_r in the case π_a does not have a filter. This concludes the proof of the second direction, which establishes the property. □

B.2 Proofs for Section 5.4 and 5.5

In this section, we give the missing details for the proof of the claims given in the main text, using the results from the previous appendices. We first show Theorem 5.4.7 in Appendix B.2.1, and show Proposition 5.4.8 in Appendix B.2.2.

B.2.1 Proof of Theorem 5.4.7

In this appendix, we show our main theorem:

Theorem 5.4.7. *There is an algorithm which, given an atomic query $q(x) \leftarrow r(a, x)$, a set $\mathcal{F}$ of path function definitions, and a set $\mathcal{UID}$ of UIDs, decides in polynomial time if there exists an equivalent rewriting of q. If so, the algorithm enumerates all the non-redundant plans that are equivalent rewritings of q.*

We start by taking the grammar $\mathcal{G}_q$ with language $\mathcal{L}_q$ used in Theorem 5.6.11 and defined in Definition 5.5.1 and the regular expression P_r used in Theorem 5.6.13 and defined in Definition 5.5.2. We make the following easy claim:

Property B.2.1. *$\mathcal{L}_q \cap P_r$ is a capturing language that faithfully represents plans, and it can be constructed in PTIME.*

Proof. By construction, P_r represents all possible skeletons obtained after a full path transformation (Theorem 5.6.13).

So, as P_r represents all possible execution plans, and as $\mathcal{L}_q$ is a capturing language (proof of Theorem 5.6.11), then $\mathcal{L}_q \cap P_r$ is a capturing language.

The only remaining part is to justify that it can be constructed in PTIME. First, observe that the grammar $\mathcal{G}_q$ for $\mathcal{L}_q$, and the regular expression for P_r, can be computed in PTIME. Now, to argue that we can construct in PTIME a context-free

grammar representing their intersection, we will use the results of [76] (in particular, Theorem 7.27 of the second edition). First, we need to convert the context-free grammar $\mathcal{G}_q$ to a push-down automaton accepting by final state, which can be done in PTIME. Then, we turn P_r into a non-deterministic automaton, which is also done is PTIME. Then, we compute a push-down automaton whose language is the intersection between the push-down automaton and the non-deterministic automaton using the method presented in [76]. This method is very similar to the one for intersecting two non-deterministic automata, namely, by building their product automaton. This procedure is done in PTIME. In the end, we obtain a push-down automaton that we need to convert back into a context-free grammar, which can also be done in PTIME. So, in the end, the context-free grammar $\mathcal{G}$ denoting the intersection of $\mathcal{L}_q$ and of the language of P_r can be constructed in PTIME. This concludes the proof. □

So let us now turn back to our algorithm and show the claims. By Property B.2.1, we can construct a grammar for the language $\mathcal{L}_q \cap P_r$ in PTIME, and we can then check in PTIME if the language of this new context-free grammar is empty. If it is the case, we know that is no equivalent plan. Otherwise, we know there is at least one. We can thus generate a word w of the language of the intersection – note that this word is not necessarily of polynomial-size, so we do not claim that this step runs in PTIME. Now, as P_r faithfully represents plans (Theorem 5.6.13), we deduce that there exists an execution plan π_a such that $\mathcal{P}(\pi_a) = w$, and from Property 5.6.9, we know we can inverse the path transformation in PTIME to gut such a plan.

To get all plans, we enumerate all words of $\mathcal{L}_q \cap P_r$: each of them has at least one equivalent plan in the preimage of the full path transformation, and we know that the path transformation maps every plan to only one word, so we never enumerate any duplicate plans when doing this. Now, by Property 5.6.9, for any word $w \in \mathcal{L}_q \cap P_r$, we can list all its preimages by the full path transformation; and for any such preimage, we can add all possible filters, which is justified by Theorem 5.6.2 and Property B.1.2. That last observation establishes that our algorithm indeed produces precisely the set of non-redundant plans that are equivalent to the input query under the input unary inclusion dependencies, which allows us to conclude the proof of Theorem 5.4.7.

B.2.2 Proof of Proposition 5.4.8

Proposition 5.4.8. *Given a set of unary inclusion dependencies, a set of path functions, an atomic query $q(x) \leftarrow r(a, x)$ and a non-redundant execution plan π_a, one can determine in PTIME if π_a is an equivalent rewriting of q.*

First, we check if π_a is well-filtering, which can easily be done in PTIME. If not, using Lemma 5.4.6 we can conclude that π_a is not an equivalent rewriting. Otherwise, we check if π_a is equivalent to its associated minimal filtering plan. This verification is done in PTIME, thanks to Theorem 5.6.2. If not, we know from Theorem 5.6.3 that π_a is not an equivalent rewriting. Otherwise, it is sufficient to show that π_a^{min} is an equivalent rewriting. To do so, we compute $w = \mathcal{P}(\pi_a^{min})$ in PTIME and check if w is a word of the context-free capturing language defined in

Theorem 5.6.11. This verification is done in PTIME. By Theorem 5.6.11, we know that w is a word of the language iff π_a is an equivalent rewriting, which concludes the proof.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Alfred V Aho. Indexed grammars—an extension of context-free grammars. *Journal of the ACM (JACM)*, 15(4):647–671, 1968.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[4] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. Openfst: A general and efficient weighted finite-state transducer library. In *International Conference on Implementation and Application of Automata*, pages 11–23. Springer, 2007.

[5] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020.

[6] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a Web of Open Data. *Semantic Web*, 2008.

[7] Y. Ba-Hillel, M. Prles, and E. Shamir. On formal properties of simple phrase structure grammars. z. phonetik, sprachwissen. komm. 15 (i961), 143-172. *Y. Bar-Hillel, Language and Information, Addison-Wesley, Reading, Mass*, pages 116–150, 1965.

[8] Franz Baader, Diego Calvanese, Deborah McGuinness, Peter Patel-Schneider, Daniele Nardi, et al. *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[9] Christopher Bader and Arnaldo Moura. A generalization of ogden's lemma. *J. ACM*, 29(2), April 1982.

[10] Ricardo Baeza-Yates and Alessandro Tiberi. Extracting semantic relations from query logs. In *KDD*, 2007.

[11] Paul Baker and Amanda Potts. 'why do white people have thin lips?'google and the perpetuation of stereotypes via auto-complete search forms. *Critical discourse studies*, 10(2):187–204, 2013.

[12] Richard Beigel and William Gasarch. A proof that the intersection of a context-free language and a regular language is context-free which does not use push-down automata. *http://www.cs.umd.edu/~gasarch/BLOGPAPERS/cfg.pdf*, .

[13] Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura. *Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation*. Synthesis Lectures on Data Management. Morgan & Claypool, 2016.

[14] Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. PDQ: Proof-driven query answering over web-based data. *VLDB*, 7(13), 2014.

[15] Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. Querying with access patterns and integrity constraints. *PVLDB*, 8(6), 2015.

[16] Michael K Bergman. White paper: the deep web: surfacing hidden value. *Journal of electronic publishing*, 7(1), 2001.

[17] Timothy J Berners-Lee and Robert Cailliau. Worldwideweb: Proposal for a hypertext project. *CERN*, 1990.

[18] Meghyn Bienvenu, Magdalena Ortiz, and Mantas Simkus. Regular path queries in lightweight description logics: Complexity and algorithms. *JAIR*, 53, 2015.

[19] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the web (LDOW2008). In *WWW*, 2008.

[20] Antoine Bosselut, Hannah Rashkin, Maarten Sap, Chaitanya Malaviya, Asli Çelikyilmaz, and Yejin Choi. COMET: commonsense transformers for automatic knowledge graph construction. In *ACL*, 2019.

[21] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000.

[22] A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *WWW*, 2012.

[23] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin,

Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[24] Fei Cai and Maarten De Rijke. A survey of query auto completion in information retrieval. *Foundations and Trends in Information Retrieval*, 2016.

[25] Andrea Calì, Diego Calvanese, and Davide Martinenghi. Dynamic query optimization under access limitations and dependencies. In *J. UCS*, 2009.

[26] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *Journal of Artificial Intelligence Research*, 48:115–174, 2013.

[27] Andrea Calì and Davide Martinenghi. Querying data under access limitations. In *ICDE*, 2008.

[28] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka, and Tom M Mitchell. Toward an architecture for never-ending language learning. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[29] Manuel Castells. *The information age*, volume 98. Oxford Blackwell Publishers, 1996.

[30] S. Ceri, A. Bozzon, and M. Brambilla. The anatomy of a multi-domain search infrastructure. In *ICWE*, 2011.

[31] Yohan Chalier, Simon Razniewski, and Gerhard Weikum. Joint reasoning for multi-faceted commonsense knowledge, 2020.

[32] Janara Christensen, Stephen Soderland, Oren Etzioni, et al. An analysis of open information extraction based on semantic role labeling. In *K-CAP*, 2011.

[33] Alexis Conneau and Douwe Kiela. Senteval: An evaluation toolkit for universal sentence representations, 2018.

[34] Niall J Conroy, Victoria L Rubin, and Yimin Chen. Automatic deception detection: Methods for finding fake news. *Proceedings of the Association for Information Science and Technology*, 52(1):1–4, 2015.

[35] Bhavana Dalvi, Niket Tandon, and Peter Clark. Domain-targeted, high precision knowledge extraction. In *TACL*, 2017.

[36] Akim Demaille, Alexandre Duret-Lutz, Sylvain Lombardy, and Jacques Sakarovitch. Implementation concepts in Vaucanson 2. In Stavros Konstantinidis, editor, *Proceedings of Implementation and Application of Automata, 18th International Conference (CIAA'13)*, volume 7982 of *Lecture Notes in Computer Science*, pages 122–133, Halifax, NS, Canada, July 2013. Springer.

[37] Daniel Deutch and Tova Milo. A quest for beauty and wealth (or, business processes for database researchers). In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, 2011.

[38] Daniel Deutch and Tova Milo. *Business Processes: A Database Perspective*. Synthesis Lectures on Data Management. Morgan & Claypool, 2012.

[39] Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. In *Theor. Comput. Sci.*, 2007.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[41] Dennis Diefenbach, Vanessa Lopez, Kamal Singh, and Pierre Maret. Core techniques of question answering systems over knowledge bases: a survey. *Knowledge and Information systems*, 55(3):529–569, 2018.

[42] Sebastian Dieguez. *Total bullshit!: au coeur de la post-vérité*. Presses Universitaires de France, 2018.

[43] Chris H. Q. Ding, Tao Li, Wei Peng, and Haesun Park. Orthogonal nonnegative matrix tri-factorizations for clustering. In *KDD*, 2006.

[44] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–610, 2014.

[45] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.

[46] Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive query plans for data integration. In *J. Log. Program.*, 2000.

[47] Carissa Schoenick et al. Moving beyond the Turing test with the Allen AI science challenge. *Communications of the ACM*, 2017.

[48] Christopher Manning et al. The Stanford CoreNLP natural language processing toolkit. In *ACL*, 2014.

[49] Huanhuan Cao et al. Context-aware query suggestion by mining click-through and session data. In *KDD*, 2008.

[50] Luisa Bentivogli et al. Revising the wordnet domains hierarchy: Semantics, coverage and balancing. In *COLING*, 2004.

[51] Mandar Joshi et al. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In *ACL*, 2017.

[52] Marius Pasca et al. Weakly-supervised acquisition of open-domain classes and class attributes from web documents and query logs. In *ACL*, 2008.

[53] Niket Tandon et al. WebChild: harvesting and organizing commonsense knowledge from the web. In *WSDM*, 2014.

[54] Niket Tandon et al. Commonsense in parts: Mining part-whole relations from the web and image tags. In *AAAI*, 2016.

[55] Niket Tandon et al. Reasoning about actions and state changes by injecting commonsense knowledge. In *EMNLP*, 2018.

[56] Sreyasi Nag Chowdhury et al. VISIR: visual and semantic image label refinement. In *WSDM*, 2018.

[57] Stanislav Malyshev et al. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *ISWC*, 2018.

[58] Tom Young et al. Augmenting end-to-end dialogue systems with commonsense knowledge. In *AAAI*, 2018.

[59] Oren Etzioni, Michael Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S Weld, and Alexander Yates. Web-scale information extraction in knowitall: (preliminary results). In *Proceedings of the 13th international conference on World Wide Web*, pages 100–110, 2004.

[60] Manaal Faruqui, Jesse Dodge, Sujay K Jauhar, Chris Dyer, Eduard Hovy, and Noah A Smith. Retrofitting word vectors to semantic lexicons. *arXiv preprint arXiv:1411.4166*, 2014.

[61] Edward A Feigenbaum. Knowledge engineering. *Annals of the New York Academy of Sciences*, 1984.

[62] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, et al. Building watson: An overview of the deepqa project. *AI magazine*, 31(3):59–79, 2010.

[63] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000.

[64] Le Figaro. Quand christine boutin cite sans sourciller le site parodique le gorafi.

[65] Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, 1999.

[66] Harry G Frankfurt. *On bullshit*. Princeton University Press, 2009.

[67] Luis Galárraga, Geremy Heitz, Kevin Murphy, and Fabian M. Suchanek. Canonicalizing open knowledge bases. In *CIKM*, 2014.

[68] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. A tool for intersecting context-free grammars and its applications. In *NASA Formal Methods Symposium*, pages 422–428. Springer, 2015.

[69] Gerald Gazdar. Applicability of indexed grammars to natural languages. In *Natural language parsing and linguistic theories*, pages 69–94. Springer, 1988.

[70] Tomasz Gogacz and Jerzy Marcinkowski. Red spider meets a rainworm: Conjunctive query finite determinacy is undecidable. In *SIGMOD*, 2016.

[71] Le Gorafi. Le gorafi - toute l'information selon des sources contradictoires.

[72] Aric Hagberg, Dan Schult, Pieter Swart, D Conway, L Séguin-Charbonneau, C Ellison, B Edwards, and J Torrents. Networkx. *URL http://networkx. github. io/index. html*, 2013.

[73] Alon Y. Halevy. Answering queries using views: A survey. In *VLDB J.*, 2001.

[74] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

[75] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[76] John E Hopcroft and 1942 Ullman, Jeffrey D. *Introduction to automata theory, languages, and computation*. Reading, Mass. : Addison-Wesley, 1979.

[77] Daniel Khashabi, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. Unifiedqa: Crossing format boundaries with a single qa system, 2020.

[78] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: A survey of techniques and tools. *ACM Comput. Surv.*, 48(3), December 2015.

[79] Douglas B Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 1995.

[80] Daniel J Lizotte, Omid Madani, and Russell Greiner. Budgeted learning of naive-bayes classifiers. *arXiv preprint arXiv:1212.2472*, 2012.

[81] Edward Loper and Steven Bird. Nltk: the natural language toolkit. *arXiv preprint cs/0205028*, 2002.

[82] Gary Marcus. The next decade in ai: Four steps towards robust artificial intelligence, 2020.

[83] David L. Martin, Massimo Paolucci, Sheila A. McIlraith, Mark H. Burstein, Drew V. McDermott, Deborah L. McGuinness, Bijan Parsia, Terry R. Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia P. Sycara. Bringing semantics to web services: The OWL-S approach. In *SWSWPC*, 2004.

[84] Mausam. Open information extraction systems and downstream applications. In *IJCAI*, 2016.

[85] John McCarthy. *Programs with common sense*. RLE and MIT computation center, 1960.

[86] Christophe Michel and Patrick Baud. La fiabilité des médias.

[87] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 1995.

[88] Alan Nash and Bertram Ludäscher. Processing unions of conjunctive queries with negation under limited access patterns. In *EDBT*, 2004.

[89] Roberto Navigli. Word sense disambiguation: A survey. *ACM Comput. Surv.*, 2009.

[90] Roberto Navigli and Simone Paolo Ponzetto. Babelnet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 193:217–250, 2012.

[91] Allen Institute of AI. AI2 science questions v2.1, 2017. http://data.allenai.org/ai2-science-questions.

[92] Harinder Pal et al. Demonyms and compound relational nouns in nominal open IE. In *AKBC*, 2016.

[93] Madhavi Parchure, M Sasikumar, and Ankit Garg. Veda: an online assessment and question banking system. *International Conference on Management Technology for Educational Practice*, 2009.

[94] Marius Pasca. Open-domain fine-grained class extraction from web search queries. In *EMNLP*, 2013.

[95] Marius Pasca. The role of query sessions in interpreting compound noun phrases. In *CIKM*, 2015.

[96] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. Yago 4: A reason-able knowledge base. In *Proceedings of the Extended Semantic Web Conference (ESWC), 2020*, 2020.

[97] N. Preda, G. Kasneci, F. M. Suchanek, T. Neumann, W. Yuan, and G. Weikum. Active Knowledge : Dynamically Enriching RDF Knowledge Bases by Web Services. In *SIGMOD*, 2010.

[98] Nicoleta Preda, Fabian M. Suchanek, Wenjun Yuan, and Gerhard Weikum. SUSIE: Search Using Services and Information Extraction. In *ICDE*, 2013.

[99] Ken Q. Pu, Vagelis Hristidis, and Nick Koudas. Syntactic rule based approach to Web service composition. In *ICDE*, 2006.

[100] Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*, 2008.

[101] Steven Rabin. *Game AI pro 2: collected wisdom of game AI professionals*. AK Peters/CRC Press, 2015.

[102] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[103] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Logic-based web services composition: From service description to process model. In *ICWS*, 2004.

[104] Rogério Reis and Nelma Moreira. Fado: tools for finite automata and regular expressions manipulation. *https://www.dcc.fc.up.pt/ nam/publica/dcc-2002-2.pdf*, 2002.

[105] Susan H Rodger and Thomas W Finley. *JFLAP: an interactive formal languages and automata package*. Jones & Bartlett Learning, 2006.

[106] Julien Romero. Pyformlang: An Educational Library for Formal Language Manipulation. In *SIGCSE*, 2021.

[107] Julien Romero, Nicoleta Preda, Antoine Amarilli, and Fabian Suchanek. Computing and illustrating query rewritings on path views with binding patterns. *Proceedings of the 29th ACM International Conference on Information and Knowledge Management - CIKM '20*, 2020.

[108] Julien Romero, Nicoleta Preda, Antoine Amarilli, and Fabian Suchanek. Equivalent rewritings on path views with binding patterns, 2020. Extended version with proofs. https://arxiv.org/abs/2003.07316.

[109] Julien Romero, Nicoleta Preda, and Fabian Suchanek. Query rewriting on path views without integrity constraints, 2020.

[110] Julien Romero and Simon Razniewski. Inside quasimodo: Exploring construction and usage of commonsense knowledge. *Proceedings of the 29th ACM International Conference on Information and Knowledge Management - CIKM '20*, 2020.

[111] Julien Romero, Simon Razniewski, Koninika Pal, Jeff Z. Pan, Archit Sakhadeo, and Gerhard Weikum. Commonsense properties from query logs and question answering forums. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management - CIKM '19*, 2019.

[112] Swarnadeep Saha and Mausam. Open information extraction from conjunctive sentences. In *COLING*, 2018.

[113] Swarnadeep Saha, Harinder Pal, and Mausam. Bootstrapping for numerical open IE. In *ACL*, 2017.

[114] Maarten Sap, Ronan LeBras, Emily Allaway, Chandra Bhagavatula, Nicholas Lourie, Hannah Rashkin, Brendan Roof, Noah A Smith, and Yejin Choi. Atomic: An atlas of machine commonsense for if-then reasoning. *AAAI*, 2018.

[115] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *ISWC*, 2011.

[116] Oded Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 237–249, 1987.

[117] Push Singh, Thomas Lin, Erik T Mueller, Grace Lim, Travell Perkins, and Wan Li Zhu. Open mind common sense: Knowledge acquisition from the general public. In *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, pages 1223–1237. Springer, 2002.

[118] Snopes. Donald trump protester speaks out: "i was paid $3,500 to protest trump's rally".

[119] Robyn Speer and Catherine Havasi. ConceptNet 5: A large semantic network for relational knowledge. In *Theory and Applications of Natural Language Processing*, 2012.

[120] Robyn Speer and Catherine Havasi. Representing general relational knowledge in ConceptNet 5. In *LREC*, 2012.

[121] Steffen Staab and Rudi Studer. *Handbook on ontologies*. Springer Science & Business Media, 2010.

[122] OASIS Standard. Web services business process execution language. https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, April 2007.

[123] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706, 2007.

[124] Fabian M Suchanek, Jonathan Lajus, Armand Boschin, and Gerhard Weikum. Knowledge representation and rule mining in entity-centric knowledge bases. In *Reasoning Web. Explainable Artificial Intelligence*, pages 110–152. Springer, 2019.

[125] Fabian M. Suchanek, Mauro Sozio, and Gerhard Weikum. SOFIE: a self-organizing framework for information extraction. In *WWW*, 2009.

[126] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. *CoRR*, abs/1811.00937, 2018.

[127] Niket Tandon, Gerard de Melo, and Gerhard Weikum. WebChild 2.0: Fine-grained commonsense knowledge distillation. In *ACL*, 2017.

[128] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. Yago 4: A reason-able knowledge base. *The Semantic Web*, 12123:583 – 596, 2020.

[129] Inkscape team. Inkscape: A vector drawing tool, 2020.

[130] Metaweb Technologies. The freebase project. http://freebase.com.

[131] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 2014.

[132] Ryen W. White, Matthew Richardson, and Wen-tau Yih. Questions vs. queries in informational search tasks. In *WWW*, 2015.

[133] WSML working group. WSML language reference. http://www.wsmo.org/wsml/, 2008.

[134] Wentao Wu, Hongsong Li, Haixun Wang, and Kenny Qili Zhu. Probase: a probabilistic taxonomy for text understanding. In *SIGMOD*, 2012.

[135] http://www.w3.org/RDF/.

[136] Frank F Xu, Bill Yuchen Lin, and Kenny Zhu. Automatic extraction of commonsense locatednear knowledge. In *ACL*, 2018.

[137] Mark Yatskar, Vicente Ordonez, and Ali Farhadi. Stating the obvious: Extracting visual common sense knowledge. In *NAACL*, 2016.

[138] Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. SWAG: A large-scale adversarial dataset for grounded commonsense inference. In *EMNLP*, 2018.

[139] Wanjun Zhong, Duyu Tang, Nan Duan, Ming Zhou, Jiahai Wang, and Jian Yin. Improving question answering by commonsense-based pre-training. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 16–28. Springer, 2019.

[140] Hang Zhou. Implementation of the hopcroft's algorithm. *https://www.irif.fr/ carton//Enseignement/Complexite/ENS/Redaction/2009-2010/hang.zhou.pdf*, 2009.

ECOLE DOCTORALE

Titre: Collecte de connaissances cachées et du sens commun à partir de services web

Mots clés: Base de connaissance, sens commun, extraction d'information, réécriture de requête, base de donnée, service web

Résumé: Dans cette thèse, nous collectons sur le web deux types de connaissances. Le premier porte sur le sens commun, i.e. des connaissances intuitives partagées par la plupart des gens comme "le ciel est bleu". Nous utilisons des logs de requêtes et des forums de questions-réponses pour extraire des faits essentiels grâce à des questions avec une forme particulière. Ensuite, nous validons nos affirmations grâce à d'autres ressources comme Wikipedia, Google Books ou les tags d'images sur Flickr. Finalement, nous groupons tous les signaux pour donner un score à chaque fait. Nous obtenons une base de connaissance, *Quasimodo*, qui, comparée à ses concurrents, montre une plus grande précision et collecte plus de faits essentiels.
Le deuxième type de connaissances qui nous intéresse sont les connaissances cachées, i.e. qui ne sont pas directement données par un fournisseur de données. En effet, les services web donnent généralement un accès partiel à l'information. Il faut donc combiner des méthodes d'accès pour obtenir plus de connaissances: c'est de la réécriture de requête. Dans un premier scénario, nous étudions le cas où les fonctions ont la forme d'un chemin, la base de donnée est contrainte par des "dépendences d'inclusion unitaires" et les requêtes sont atomiques. Nous montrons que le problème est alors décidable en temps polynomial. Ensuite, nous retirons toutes les contraites et nous créons un nouvelle catégorie pertinente de plans: les "smart plans". Nous montrons qu'il est décidable de les trouver.

Title: Harvesting Commonsense and Hidden Knowledge From Web Services

Keywords: Knowledge base, Commonsense, Information Extraction, Query rewritings, database, web services

Abstract: In this thesis, we harvest knowledge of two different types from online resources. The first one is commonsense knowledge, i.e. intuitive knowledge shared by most people like "the sky is blue". We extract salient statements from query logs and question-answering sites by carefully designing question patterns. Next, we validate our statements by querying other web sources such as Wikipedia, Google Books, or image tags from Flickr. We aggregate these signals to create a final score for each statement. We obtain a knowledge base, *Quasimodo*, which, compared to its competitors, has better precision and captures more salient facts.
The other kind of knowledge we investigate is hidden knowledge, i.e. knowledge not directly given by a data provider. More concretely, some Web services allow accessing the data only through predefined access functions. To answer a user query, we have to combine different such access functions, i.e. we have to rewrite the query in terms of the functions. We study two different scenarios: In the first scenario, the access functions have the shape of a path, the knowledge base respects constraints called "Unary Inclusion Dependencies", and the query is atomic. We show that the problem is decidable in polynomial time, and we provide an algorithm with theoretical evidence. In the second scenario, we remove the constraints and create a new class of relevant plans called "smart plans". We show that it is decidable to find these plans, and we provide an algorithm.

Institut Polytechnique de Paris
91120 Palaiseau, France