ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2018-ENST-0043

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TÉLÉCOM ParisTech

Spécialité « Informatique »

présentée et soutenue publiquement par

Thomas Rebele

le 2018-07-19

Extending the YAGO knowledge base

Directeur de thèse: **Fabian M. Suchanek**

Jury

M. Bernd AMANN, Professeur, Sorbonne Université	Examinateur
M. Felix NAUMANN, Professeur, Université de Potsdam	Rapporteur
M. Benjamin NGUYEN, Professeur, Université d'Orléans	Examinateur
Mme. Nathalie PERNELLE, Maître de Conférences, Université Paris-Sud	Rapporteure
M. Fabian SUCHANEK, Professeur, Télécom ParisTech	Directeur de thèse
Mme. Katerina TZOMPANAKI, Maître de Conférences, Université de Cergy-Pontoise	Examinatrice

T H È S E

TÉLÉCOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

Abstract

English

A knowledge base is a set of facts about the world. YAGO was one of the first large-scale knowledge bases that were constructed automatically. This thesis focuses on extending the YAGO knowledge base along two axes: extraction and preprocessing.

The first main contribution of this thesis is improving the number of facts about people. The thesis describes algorithms and heuristics for extracting more facts about birth and death date, about gender, and about the place of residence. The thesis also shows how to use these data for studies in Digital Humanities.

The second main contribution are two algorithms for repairing a regular expression automatically so that it matches a given set of words. Experiments on various datasets show the effectiveness and generality of these algorithms. Both algorithms improve the recall of the initial regular expression while achieving a similar or better precision.

The last contribution is a system for translating database queries into Bash scripts. This approach allows preprocessing large tabular datasets and knowledge bases by executing Datalog and SPARQL queries, without installing any software beyond a Unix-like operating system. Experiments show that the performance of our system is comparable with state-of-the-art systems.

French

Une base de connaissances est un ensemble de faits sur le monde. Parmi elles se trouve YAGO, une des premières à être générée automatiquement à grande échelle. Cette thèse se concentre sur l'extension de la base de connaissances YAGO en améliorant l'extraction de contenu et son accès.

La première contribution principale consiste en l'augmentation de la quantité des faits sur les personnes. Pour se faire, cette thèse décrit des algorithmes et des heuristiques permettant d'extraire d'avantage de dates de naissance et de décès, d'indications sur le sexe et de lieux de résidence. Ces données sont ensuite utilisées dans le cadre d'études en humanités numériques.

La deuxième contribution principale présente deux algorithmes permettant de réparer automatiquement une expression régulière afin qu'elle corresponde à un ensemble de mots donnés. Des expériences sur divers jeux de données montrent l'efficacité et la généralité de l'approche. Comparés aux travaux précédents, le rappel est amélioré tout en conservant une précision similaire, voire supérieure.

La dernière contribution est un système de traduction de requêtes sur des bases de données en scripts Bash. Cela permet de prétraiter des jeux de données en utilisant des requêtes Datalog et SPARQL sans installer de logiciel au-delà d'un système d'exploitation de type Unix. Les expériences montrent que les performances de notre approche sont comparables à celles des meilleures solutions du marché.

Acknowledgements

First, I want to thank my supervisor, Fabian Suchanek, who guided me through the years of my PhD. This thesis would not have been possible without his help and his support.

I would also like to thank my coworkers, who have helped and motivated me, each in their own way. Foremost, Katerina Tzompanaki, with whom I had many research discussions; Thomas Pellissier Tanon, who worked with me on my most recent research project; and Arash Nekoei, for his persistence and his amazing capability to find faults in the data. I want to thank Hiep Le for making it possible to participate in the first international conference for me, even if it was in Germany. I thank the researchers from the Max-Planck Institute for Informatics in Saarbrücken: Johannes Hoffart, Joanna "Asia" Biega, Erdal Kuzey, and Gerhard Weikum. They made it possible for me to participate in my "first real" international conference, i.e., outside of Germany.

Many thanks also to the researchers from my laboratory at Télécom ParisTech: Albert, Antoine, Arnaud, Atef, Camille, Danai, Hana, Jacob, James, Jean-Benoît, Jean-Louis, Jesse, Jonathan, Luis, Manthos, Marc, Marie, Maroua, Mauro, Maximilian, Mikaël, Miyoung, Mostafa, Mouhamadou, Oana, Pierre-Alexandre, Pierre, Quentin, Raman, Roxana, Sébastien, Siwar, Talel, and Ziad. Especially to Ziad, who helped me more than he knows. I'll keep the time of my PhD in a good memory. The countless hours of eating lunch together, during which we discussed about life, the universe, and everything. The seminars, teaching hours, and conferences that we visited together. And the numerous social events.

Thanks to my friends, and people I met during my PhD: Adrian, Alex, Amelie, Anna, Ashley, Bettina, Camille, Christian, Elena, Filipe, Francesco, Giang, Gleison, Inés, Jia, Johannes, Katie, Kelley, Kilian, Luise, Marcel, Mariem, Michael, Mohammed, Ophélie, Roberto, Roland, Samih, Simon, Sonja, Stefan, Tess, Vivian, Yuri, and the people from the dancing class and the fanfare of Télécom ParisTech. They made my time much more enjoyable, and helped me during some difficult periods of my life. I had a lot of good experiences with every one of them. This includes having nice discussions, cooking and/or eating together, doing exercises, or just having a good time.

Last, but not least, I want to thank my family and relatives. Their support made it possible for me to study, and to work as a researcher: My father Franz, my mother Angelika, my brothers Martin and Peter; my relatives Erich, Karin, Rainer, Monika, Helmut, Holger, Philip, and Berta.

In the memory of my grandparents, Mathias, Amalie, and Lorenz.

This research was partially supported by Labex DigiCosme (project ANR-11-LABEX-0045-DIGICOSME) operated by ANR as part of the program "Investissement d'Avenir" Idex Paris Saclay (ANR-11-IDEX-0003-02).

Contents

1 Introduction **13**

1.1 Motivation 13

1.2 Outline and Contributions 15

2 Preliminaries **19**

2.1 Definitions 19

2.1.1 Entity, Identifier, and Label 19

2.1.2 Class, Subclass, and Taxonomy 20

2.1.3 Relation and Fact 21

2.1.4 Knowledge Base 22

2.2 Semantic Web 23

2.2.1 RDF 23

2.2.2 OWL 24

2.2.3 SPARQL 25

3 The YAGO Knowledge Base **27**

3.1 Introduction 27

3.2 Related Work 28

3.3 History 29

3.4 Content 29

3.5 Construction of YAGO 31

3.5.1 Sources 31

3.5.2 Extraction Process 33

3.5.3 Evaluation . . . 35
3.6 Infrastructure . . . 37
3.7 Applications of YAGO . . . 38
3.8 Outlook . . . 39
3.9 Summary . . . 40

4 Extending Information about People in YAGO 41
4.1 Introduction . . . 41
4.2 Related Work . . . 42
4.3 Problem Statement . . . 43
4.4 Methods . . . 43
4.4.1 Gender . . . 43
4.4.2 Dates . . . 45
4.4.3 Place of Residence . . . 46
4.5 Results . . . 48
4.6 Case Studies . . . 49
4.6.1 Population Size over Time . . . 49
4.6.2 Life Expectancy over Time . . . 50
4.6.3 Births per Month . . . 53
4.6.4 Full Moon Myth . . . 54
4.6.5 Age at First Child Birth . . . 55
4.6.6 Rise and Fall of Civilizations . . . 55
4.7 Summary . . . 59

5 Adding Missing Words to Regular Expressions 61
5.1 Introduction . . . 62
5.2 Related Work . . . 63
5.3 Preliminaries . . . 66
5.3.1 Regular Expressions . . . 66
5.3.2 Problem Statement . . . 68
5.3.3 Finding the Matchings . . . 69
5.4 Simple Algorithm . . . 72

5.5 Adaptive Algorithm . . . 73
5.6 Experiments . . . 79
5.6.1 Setup . . . 79
5.6.2 Experimental Evaluation and Results . . . 81
5.7 Demo . . . 86
5.8 Summary . . . 87
5.9 Outlook . . . 88

6 Answering Queries with Unix Shell **89**
6.1 Introduction . . . 89
6.2 Example . . . 91
6.3 Related Work . . . 93
6.4 Preliminaries . . . 95
6.5 Approach . . . 100
6.5.1 Datalog Dialect . . . 100
6.5.2 Loading Datalog . . . 101
6.5.3 Producing Bash Commands . . . 104
6.5.4 Recursion . . . 105
6.5.5 Materialization . . . 106
6.5.6 Optimization . . . 108
6.6 Experiments . . . 112
6.6.1 Lehigh University Benchmark . . . 113
6.6.2 Reachability . . . 116
6.6.3 YAGO and Wikidata . . . 119
6.7 Web Interface . . . 121
6.8 Summary . . . 124
6.9 Outlook . . . 124

7 Conclusion **127**
7.1 Summary . . . 127
7.2 Outlook . . . 128

Appendices 141

A Generated Unix Shell Script 143

A.1 Datalog Mode . . . 143

A.2 SPARQL/OWL Mode . . . 145

B Résumé en français 147

B.1 Introduction . . . 147

B.1.1 Motivation . . . 147

B.1.2 Outline . . . 149

B.2 La base de connaissances YAGO . . . 150

B.2.1 Historique . . . 150

B.2.2 Contenu . . . 150

B.2.3 Construction de YAGO . . . 151

B.2.4 Résumé . . . 152

B.3 Extension des informations sur les personnes dans YAGO . . . 152

B.3.1 Introduction . . . 152

B.3.2 Méthodes . . . 153

B.3.3 Résultats . . . 155

B.3.4 Études de cas . . . 155

B.3.5 Résumé . . . 157

B.4 Ajout de mots manquants aux expressions régulières . . . 157

B.4.1 Préliminaires . . . 158

B.4.2 Algorithme simple . . . 160

B.4.3 Algorithme adaptatif . . . 160

B.4.4 Expériences . . . 163

B.4.5 Résumé . . . 163

B.5 Répondre aux requêtes avec Unix Shell . . . 164

B.5.1 Exemple . . . 164

B.5.2 Préliminaires . . . 164

B.5.3 Approche . . . 165

B.5.4 Expériences . . . 169

B.5.5 Résumé 173
B.6 Conclusion 173

Chapter 1

Introduction

1.1 Motivation

A knowledge base is a computer-processable collection of knowledge about the world. A knowledge base usually contains *entities*, such as Albert Einstein, the Technical University of Munich (TUM), or Paris. It also contains *facts* about these entities, such as the fact that Albert Einstein is a physicist, that TUM is a university, or that Paris is located in France. In the recent decades, knowledge bases have started to pop up in everyday life. For example, when a user searches for information about a well-known person on one of the two major search engines – Google or Bing –, she receives not only a list of web pages, but also an infobox in tabular form. For instance, if she searches for "Albert Einstein", the search engines show – among other things – his date of birth and death, his wives, and some of his works. This allows the user to obtain basic information about a person, without the need to skim through the web pages.

Both search engines also provide infoboxes for other types of queries. For example, the query "12 angry men" shows that it is a drama movie from 1957, directed by Sidney Lumet, and "Hotel California" shows information about the song from the band Eagles from 1976. Given the unfathomable huge number of people, movies, and songs, the user might wonder how the search engines know details about these entities.

Such detailed answers are possible, because Google and Bing each have a huge knowledge base that contains structured information about entities, such as people, movies, and books. But knowledge bases do not just support search engines. They are also used in machine translation, semantic search, question answering, intelligent assistants, and data mining [Suchanek and Weikum, 2016].

It took several years, from the conception and implementation of knowledge base projects, until the technology became mature enough to be useful for the end user. The first knowledge bases were constructed manually. One of the first projects in this direc-

tion was Cyc, a commercial system [Lenat and Guha, 1989]. Cyc started in 1984, and is still actively developed. Cyc contains basic concepts, and rules about how the world works. Another notable early knowledge base is WordNet, a linguistic database for the English language [Miller, 1995]. This knowledge base contains mainly English words and their relationships, e.g., synonym, antonym, hyponym, or hypernym.

However, it takes a lot of time and effort to create a knowledge base manually. Therefore research began looking into the automatic construction of knowledge bases. With the growth of the Web, more and more approaches constructed knowledge bases automatically by extracting information from Web corpora. This development was accelerated in the early years with the creation of Wikipedia, an online crowd-sourced encyclopedia. Wikipedia contains textual description, infoboxes, and a system to categorize articles. Especially the latter two make information extraction particularly fruitful. Information extraction techniques, such as regular expressions, could quickly generate large numbers of facts.

Some of the more prominent approaches along these lines are YAGO, DBpedia, Wikidata, NELL, and Google's Knowledge Vault. The YAGO knowledge base, in particular, was one of the earliest approaches in this direction. It combined the strengths of two of the previously mentioned projects: WordNet has a hand-crafted and well-designed taxonomy (class hierarchy), but only few entities. On the other hand, Wikipedia contains many entities, but does not provide a clean taxonomy. YAGO closed the gap by merging the two resources. It distinguishes itself from similar projects by focusing on the precision. This is the ratio of true facts with respect to a ground truth. Optimizing the precision often goes hand-in-hand with reducing the coverage, i.e., the total number of facts. Nevertheless YAGO achieves a precision of 95%, while containing millions of facts.

During my doctoral studies, I was actively involved in the maintenance, and the evaluation of the YAGO knowledge base. Therefore, the first part of the thesis is dedicated to the description of this knowledge base.

Despite the good precision and coverage of YAGO, there is still room for improvement. YAGO is still incomplete – even when using Wikipedia as ground truth. For example, the original version of YAGO knew the place of residence for only 53% of the people, and the gender for only 63% of them. This thesis shows how to improve the coverage of four essential types of information about people in YAGO: the birth and death dates, the gender, and the place of residence. For example, we can extract the gender for 90% of the people now. This newly gained wealth of data allows using YAGO for historical studies. For example, we can see that women postpone having their first child in the recent centuries.

To increase the coverage, one has to debug and adapt regular expressions. For example, the previous version of YAGO extracted the year from dates with three digits, such

as “100 BC”, but it did not extract the year from dates with two digits, such as “44 BC”. Correcting the regular expressions is tedious, because the regular expressions are quite complex. This observation led to the idea to automate the adaptation of regular expressions. This thesis presents two algorithms that can automatically modify a regular expression so that it matches a given word. Our experiments show that our algorithms can generalize regexes based on small training data.

During the work on maintaining and implementing new algorithms for the YAGO knowledge base, I often encountered the task of searching all facts in YAGO about a certain entity. Loading the entire knowledge base into a database or a triple store would have take too much time and effort. The easiest way to obtain the result was normally to write a short Unix command, such as grep or AWK, and wait for some minutes. Over the years I became more and more experienced with the Unix commands, their command line parameters, and Bash, the command line interpreter.

This know-how inspired the last part of the thesis: An algorithm that transforms database queries to Bash scripts. The user can enter her query in Datalog or SPARQL in a web interface. This query is transformed automatically to an optimized Bash script. She can then download the Bash script, and run it on the knowledge base. In this way, the user can obtain the results of her query without importing the knowledge base into a database. Our experiments show that our Bash scripts are often faster than traditional databases or triple stores.

1.2 Outline and Contributions

This work consists of four parts.

Describing YAGO as resource. In the first part, this thesis introduces the YAGO knowledge base, and describes its history, construction, content, maintenance, and applications. In this chapter I also describe my contributions to the YAGO knowledge base in terms of its maintenance, evaluation, and development. This part of the thesis is based on the following publication:

> Rebele, T., Suchanek, F. M., Hoffart, J., Biega, J., Kuzey, E., and Weikum, G. (2016). YAGO: A Multilingual Knowledge Base from Wikipedia, Wordnet, and Geonames. In *The Semantic Web - ISWC - 15th International Semantic Web Conference*, volume 9982 of *Lecture Notes in Computer Science*, pages 177–185. URL: `https://doi.org/10.1007/978-3-319-46547-0_19` (resource paper)

Digital humanities. The second part of this thesis studies the application of YAGO to the digital humanities. The knowledge base is extended with new facts about birth and

death dates, locations, and gender of people. Several case studies show the usefulness of such data. This work has led to the following publication:

> Rebele, T., Nekoei, A., and Suchanek, F. M. (2017a). Using YAGO for the Humanities. In *Proceedings of the Second Workshop on Humanities in the Semantic Web (WHiSe II) co-located with 16th International Semantic Web Conference (ISWC)*, volume 2014 of *CEUR Workshop Proceedings*, pages 99–110. CEUR-WS.org. URL: `http://ceur-ws.org/Vol-2014/paper-11.pdf` (workshop paper)

Repairing regular expressions. The third part focuses on a problem that often occurs in information extraction: a regular expression might not match all the words that it should match. Two algorithms are presented, which modify a regex, so that the regex matches also a set of given words. This part of the thesis is based on the following publications:

> Rebele, T., Tzompanaki, K., and Suchanek, F. M. (2017b). Visualizing the addition of missing words to regular expressions. In *Proceedings of the ISWC Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC)*, volume 1963 of *CEUR Workshop Proceedings*. CEUR-WS.org. URL: `http://ceur-ws.org/Vol-1963/paper457.pdf` (demonstration)

> Rebele, T., Tzompanaki, K., and Suchanek, F. (2018b). Adding Missing Words to Regular Expressions. In *Advances in Knowledge Discovery and Data Mining - 22nd Pacific-Asia Conference, PAKDD, Proceedings*, Lecture Notes in Computer Science. URL: `https://www.thomasrebele.org/publications/2018_pakdd.pdf` (full paper)

> Rebele, T., Tzompanaki, K., and Suchanek, F. (2018c). Technical Report: Adding Missing Words to Regular Expressions. Technical report, Telecom ParisTech. URL: `https://hal.archives-ouvertes.fr/hal-01745987v1` (technical report)

Answering Datalog queries with Bash. Last but not least, the fourth part of this thesis describes a system that translates Datalog programs or SPARQL queries into Bash scripts. This allows a user to prepossess large tabular data on a UNIX system without software installation. Interestingly, our approach can obtain query results in a time comparable with state-of-the-art systems. This work gave rise to the following paper, which is currently under submission:

> Rebele, T., Tanon, T. P., and Suchanek, F. M. (2018a). Technical Report: Answering Datalog Queries with Unix Shell Commands. Technical report. URL: `https://www.thomasrebele.org/publications/2018_report_bashlog.pdf` (technical report, under review)

Conclusion. Chapter 7 concludes the thesis. It discusses the scope of these contributions, and suggests potential research directions.

Chapter 2

Preliminaries

This chapter introduces the basic notions and the background knowledge that are helpful to understand the thesis in general. Topics that are used in only one chapter are introduced in that chapter.

2.1 Definitions

First, we will define concepts that often appear when talking about a "knowledge base", as well as the concept of knowledge bases itself. The definitions in this chapter are based on [Suchanek and Weikum, 2016].

2.1.1 Entity, Identifier, and Label

Definition (Entity). *An* entity *is any abstract or concrete object of fiction or reality.*

Definition (Identifier). *An* identifier *is a string referring to a single entity.*

Bertrand Russel aptly describes an entity as "whatever may be an object of thought" [Russell, 1903]. We have already seen examples for entities in the introduction of this thesis: Albert Einstein, the Technical University Munich, or Paris. But also a chair, a pen, or this thesis are entities. Examples for abstract entities are the mathematical operation of multiplication, the theory of evolution, or the flat earth model. This thesis uses the notation <Albert_Einstein> for identifiers to denote entities of a knowledge base. For the scope of this chapter, we will write id_e as an identifier of entity e. In the context of this thesis, we will not distinguish identifiers from entities in general, and just talk of entities instead.

Definition (Literal). *A* literal *is an entity that represents a value, e.g., a number, a date, or a string.*

Definition (Label). *A* label *of an entity is a human-readable string that represents the entity's name.*

Literals are a special case of abstract entities. They occur quite frequently, and so they are treated in a special way. Take for example the literal "Technische Universität München". It is the German name of the entity <Technical_University_Munich>. This literal is thus a label of the university.

2.1.2 Class, Subclass, and Taxonomy

Definition (Class). *A* class, *or* type *is a named group of entities that share some common characteristics. The name of the class is its identifier. The extension of the class is the set of these entities. An entity that is an element of that set is called an* instance *of that class. If the extension of class A is a proper subset of the extension of class B, we call class A a* subclass *of B.*

Definition (Taxonomy). *A* taxonomy *is a directed graph, with class identifiers as nodes, and an edge from class identifier id_A to class identifier id_B, if A is a subclass of B. More formally, a taxonomy is a graph (E, V), where E is the set of class identifiers, and V is the set of edges, $V = \{(id_{C_1}, id_{C_2}) \mid C_1 \subset C_2\}$, where C_1 and C_2 are the extensions of the classes with the identifiers id_{C_1} and id_{C_2}.*

Every class is an entity. We do not distinguish between a class and its extension if the distinction is clear from the context. For example, we will say that a class contains an entity. We assume a class of classes, *Class*, i.e., a class whose extension contains all classes, including itself. We avoid the problem of expressing self-contained classes by writing down a class as a set of identifiers. We model literals, such as strings or dates, as their own classes, such as *String* and *Date*, respectively.

Real world examples of classes could be <Scientist>, <University>, and <City>. The entities <Albert_Einstein>, <Technical_University_Munich>, and <Paris> would be members of the classes <Scientist>, <University>, and <City>, respectively. Those three classes would be subclasses of <Person>, of <Institute>, and of <Location>, respectively. The three superclasses themselves would be subclasses of <Entity>. The subclass relationship builds a taxonomy, which is illustrated in Figure 2.1 on the facing page.

Note that an entity in one knowledge base might be modeled as a class in another knowledge base. Take for example the book "Introduction to Algorithms". We could use <Introduction_to_Algorithms> to refer to a specific book with that title in the library of Télécom ParisTech. We could also use <Introduction_to_Algorithms> to refer to an edition of that book, or to that book in general. In those cases the identifier would refer to a non-class entity. However, if we use <Introduction_to_Algorithms> to refer to all books with that title, it would be a class. Even if we use the identifier <Introduction_to_Algorithms> to refer to the specific book at Télécom ParisTech,

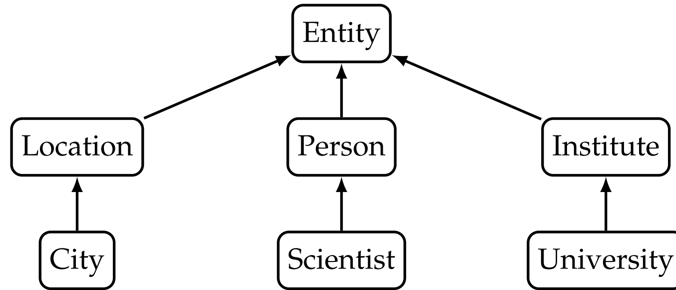


Figure 2.1 – Example taxonomy

a physicist might argue that we should model it as class referring to all points in space-time that represent that book. We leave such detailed considerations to ontologists, and assign identifiers to entities and classes according to common sense. The reader is referred to [Guarino, 1998; Suchanek and Weikum, 2016] for a detailed discussion.

2.1.3 Relation and Fact

Definition (Relation). *A* relation *is a named relationship between entities. The name of the relation is its identifier. The extension of the relation is a subset of the Cartesian product of the extensions of n classes. The number n is called the* arity *of the relation. Each element of the extension of the relation is a* tuple *of the form $\langle e_1, \ldots, e_n \rangle$. A* binary relation, *or* property, *is a 2-ary relation.*

Definition (Fact). *A* fact, *or* statement, *is a relation identifier with an element of the extension of that relation. A fact of a relation r can be written as $id_r(id_{e_1}, \ldots, id_{e_n})$, where id_r is the relation identifier, and $id_{e_1}, \ldots, id_{e_n}$ are identifiers of entities. For binary relations, a fact can be written as a* triple $id_{e_1}\ id_r\ id_{e_2}$. *We call the three parts of the triple the* subject, *the* predicate, *and the* object, *respectively.*

A relation itself is an entity. In the context of this thesis, we do not distinguish between a relation and its extension. For example, we will say that a tuple is an element of the relation. Similar to classes, relationships between relations can be expressed by using their identifiers. The relation identifier usually starts with a lower case character.

As an example, assume that we want to model the birth of a person. We could model it as ternary relation $birth \subseteq Person \times Location \times Date$. The fact that Albert Einstein was born in Ulm on March 14, 1879 would then be represented as <birth>(<Albert_Einstein>, <Ulm>, <1879-03-14>).

Every n-ary relation r can be expressed using binary relations: we introduce a new entity x for every fact $id_r(id_{e_1}, \ldots, id_{e_n})$, and rewrite the fact as triples $id_x\ id_{r,1}\ id_{e_1}$, $\ldots$, $id_x\ id_{r,n}\ id_{e_n}$, with new identifiers $id_{r,1}, \ldots, id_{r,n}$. In our example, we could write <Albert_Einstein> <wasBornIn> <Ulm>, and <Albert_Einstein>

<wasBornOnDate> <1879-03-14>. Knowledge bases in general use binary relations, as they are more flexible. For example, we can state the birth date of Albert Einstein even if we do not know his birth place – something that is not so easily possible with the ternary relation.

Relations can also express relationships between different kinds of entities, such as between an entity and a class, between two classes, or between two relations, using their respective identifiers. For example, the relation <type> specifies the "is-a" relationship between an entity and a class (via its class identifier). This is expressed in triple notation as follows: <Albert_Einstein> <type> <Scientist>.

Definition (Database). *A* database *is a set of relations.*

A database is normally used within a database management system (DBMS), which allows the user to easily carry out frequent tasks, such as querying, or inserting new data. In general, a database is stored as a collection of tables. Several types of data organization are possible. A table could be stored row-wise or column-wise. In summary, databases focus on the relations. Knowledge bases, in contrast, focus on the facts, as we will see in the next section.

2.1.4 Knowledge Base

In this thesis, we restrict ourselves to knowledge bases that contain facts in the form of triples. In the literature, there are many variations of definitions for the terms "ontology" and "knowledge base". For this thesis, we stick to the following pragmatic definitions, roughly following those of [Baader and Nutt, 2003]:

Definition (Instance). *An* instance *is an entity that is neither a class, nor a relation, nor a literal.*

Definition (ABox). *An* ABox *("Assertions") is a set of facts that concern instances, i.e., facts whose subject or object are instances.*

Definition (TBox). *A* TBox *("Terminologies"), is a set of facts that do not concern instances, i.e., facts where neither subject nor object are instances.*

Definition (Knowledge base). *A* knowledge base *consists of an ABox and a TBox.*

In more practical words, a knowledge base is a computer-processable collection of knowledge about the world. In the context of this thesis, we call the TBox component an *ontology*. As described in the introduction, a knowledge base contains facts about entities, such as the fact that <Paris> is a <City>. That fact is part of the ABox. Relationships between classes, such as the fact that every <City> is a <Location>, are part of the TBox. As a consequence, the entire taxonomy is part of the TBox.

2.2 Semantic Web

2.2.1 RDF

The *Resource Description Framework*[1] (RDF) [W3C, 2014] is a formalism for specifying knowledge bases. It describes the syntax, formats, and conventions for writing down facts. RDF stores only triples (binary relations). This is not a constraint, because we can express every n-ary relation as triples, as shown above.

In RDF, entities are generally identified by IRIs, a generalization of URIs. In that way, the entities have a globally unique identifier. The other types of entities are literals, and blank nodes. Blank nodes allow stating facts about an entity, without giving it an IRI. RDF has the limitation that the subject and the predicate of triples cannot be literals. RDF Schema (RDFS) can be used to express TBox facts. The RDFS relation `rdfs:subClassOf` expresses the subclass relation.

RDF comes in several formats: Turtle, TriG, RDFa, N-Triples, N-Quads, and JSON-LD. We describe first the more concise Turtle format, and then the N-Triples format, which can be parsed more easily.

Turtle. The Turtle[2] format is also called Terse RDF Triple Language (TTL). The following snippet states that Albert Einstein is a scientist, and that every scientist is a person:

```
@prefix kb:   <http://example.org/knowledge-base#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

kb:Albert_Einstein  rdf:type         kb:Scientist .
kb:Scientist        rdfs:subClassOf  kb:Person .
```

To demonstrate the use of blank nodes, we will express that Albert Einstein worked for a university, without specifying which university.

```
kb:Albert_Einstein  kb:worksFor      _:b1 .
_:b1                rdf:type         kb:University .
```

The blank node identifier `_:b1` allows us to express facts about the blank node. It is only valid within the file that contains it. We can also use a more concise syntax:

```
kb:Albert_Einstein  kb:worksFor      [
    rdf:type  kb:University
] .
```

1. https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/
2. https://www.w3.org/TR/turtle/

The []-notation introduces a blank node, which is the subject of the statement fragments within the brackets.

N-Triples. The N-Triples[3] format is a subset of the Turtle format. It does not allow the prefix notation, and so the IRI of an entity must be written out fully. It supports blank nodes using blank node identifiers, such as `_:b1`. In this thesis we will refer to the canonical N-Triples format, which requires that the subject, predicate, and object are separated with a single space. Take as example the following N-Triples listing, which states that Albert Einstein is a scientist:

```
<http://example.org/knowledge-base#Albert_Einstein> <http://www.w3.org ↩
    /1999/02/22-rdf-syntax-ns#type> <http://example.org/knowledge-base# ↩
    Scientist> .
```

2.2.2 OWL

The *Web Ontology Language*[4] (OWL) is an extension of RDF, which allows defining semantic constraints of a knowledge base. We continue the RDF example, and show how to specify rules in OWL. For the sake of our example, we assume that all people working at an university are scientists. In first-order logic this could be written as $\forall x : Person(x) \wedge (\exists y : worksFor(x, y) \wedge University(y)) \implies Scientist(x)$. We need to translate this rule into a triple form in order to express this in OWL. The rule is expressed in OWL as follows:

```
kb:Scientist  owl:intersectionOf (
    kb:Person
    [
      rdf:type            owl:Restriction;
      owl:onProperty      kb:worksFor;
      owl:someValuesFrom  kb:University
    ]
) .
```

The ()-notation specifies an RDF collection. Every RDF collection can be expressed in the triple format by using concepts from functional programming. The interested reader is referred to the RDF documentation[5].

3. `https://www.w3.org/TR/n-triples/`
4. `https://www.w3.org/TR/owl2-overview/`
5. `https://www.w3.org/TR/rdf11-mt/#rdf-collections`

2.2.3 SPARQL

The *SPARQL Protocol and RDF Query Language* [6] (SPARQL) is a query language for RDF. A user can write a query in SPARQL to obtain a list of entities, or even a list of tuples of entities that fulfill certain conditions. As an example, the following SPARQL query returns all people, their birth date, and their place of birth from the knowledge base:

```
PREFIX kb:  <http://example.org/knowledge-base#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?person ?city ?date
WHERE {
  ?person kb:wasBornOnDate ?date .
  ?person kb:wasBornIn ?city .
}
```

6. https://www.w3.org/TR/sparql11-query/

Chapter 3

The YAGO Knowledge Base

YAGO is a large knowledge base that is built automatically from Wikipedia, WordNet, and GeoNames. YAGO focuses on extraction quality, and achieves a manually evaluated precision of 95%. In this chapter, we explain how YAGO is built from its sources, how its quality is evaluated, how a user can access it, and how other projects utilize it.

This chapter is based on the following publication:

> Rebele, T., Suchanek, F. M., Hoffart, J., Biega, J., Kuzey, E., and Weikum, G. (2016). YAGO: A Multilingual Knowledge Base from Wikipedia, Wordnet, and Geonames. In *The Semantic Web - ISWC - 15th International Semantic Web Conference*, volume 9982 of *Lecture Notes in Computer Science*, pages 177–185. URL: `https://doi.org/10.1007/978-3-319-46547-0_19` (resource paper)

3.1 Introduction

The YAGO knowledge base (Yet Another Great Ontology) [Suchanek et al., 2007] was the first academic project to build a knowledge base from Wikipedia, closely followed by the DBpedia project [Auer et al., 2007]. Today, YAGO is a larger project at the Max Planck Institute for Informatics, and Télécom ParisTech University. The knowledge base draws on several sources, including WordNet and GeoNames, and has grown to 16 million entities and more than 100 million facts. YAGO combines information from Wikipedias in 10 different languages into a coherent whole, thus giving the knowledge a multilingual dimension. It also attaches spatial and temporal information to many facts, and thus allows the user to query the data over space and time. It is part of the Linked Open Data cloud.

YAGO has found numerous applications. Its taxonomy is used in the DBpedia project. Data from YAGO was also used in the IBM Watson system [Kalyanpur et al., 2011], which beat the human champion in the Jeopardy! quiz show in 2011. YAGO has also

found applications in named entity disambiguation, news paper analysis, and question answering. The knowledge base is freely available for download.

This chapter presents the YAGO knowledge base, its schema, construction, and applications in detail. It is structured as follows. Section 3.2 describes other knowledge base projects. Section 3.3 gives an overview of the YAGO project. Section 3.4 describes the content, and Section 3.5 describes the construction of the knowledge base. Section 3.6 illustrates data formats and tools. Section 3.7 shows applications of YAGO. Section 3.8 shows future research directions, before Section 3.9 recapitulates.

3.2 Related Work

Following YAGO, several projects have started to create large knowledge bases. Among the most visible projects are the following [Suchanek and Weikum, 2014]:

DBpedia [Lehmann et al., 2015a] extracts structural data from Wikipedias in different languages. The DBpedia community maps infobox attributes in order to combine the Wikipedias to a single knowledge base [Lehmann et al., 2015b].
NELL [Carlson et al., 2010] – short for Never-ending Language Learning – performs two steps repeatedly. First, it extracts data based on input classes, relations, and patterns. Then, it infers new classes, relations and patterns from the new data.
BabelNet [Navigli and Ponzetto, 2012] provides a multilingual lexicographic knowledge base. It combines Wikipedia and WordNet, and links concepts by using human and machine translation, and similarity between concepts.
Wikidata [Vrandecic and Krötzsch, 2014] is a crowd-sourced knowledge base. It also integrates data from other projects, most notably Freebase [1].
Knowledge Vault [Dong et al., 2014] is a probabilistic knowledge base developed by Google. It combines existing knowledge bases with information extraction from the Web, and supervised machine learning.

Further academic and commercial projects are KnowItAll [Etzioni et al., 2004], ConceptNet [Speer et al., 2017], DeepDive [Niu et al., 2012], ImageNet [Deng et al., 2009], WikiNet [Nastase et al., 2010], WikiTaxonomy [Ponzetto and Strube, 2008], Freebase (until 2016) [Bollacker et al., 2008], EntityCube [Nie et al., 2012], and Probase [Wu et al., 2012].

Unlike the Knowledge Vault, YAGO is publicly available for download. Unlike DBpedia and Wikidata, YAGO is not constructed through crowd-sourcing, but through information extraction and merging. The YAGO project puts a particular focus on the quality of its data, which is assessed through regular manual evaluations. It also has a rather elaborate taxonomy in comparison to other projects, which it inherits from Word-

1. see `https://www.wikidata.org/wiki/Wikidata:WikiProject_Freebase`

Net [Fellbaum, 1998]. YAGO also integrates several multilingual sources into a single knowledge base. Finally, YAGO pays particular attention to the anchoring of the facts in time and space.

3.3 History

The YAGO project started in 2006 from a simple idea: Wikipedia contains a large number of instances, such as singers, movies, or cities. However, its hierarchy of categories is not directly suitable as a taxonomy. WordNet, on the other hand, has a very elaborate taxonomy, but a rather low recall on instances. It thus seemed promising to combine both resources to obtain the best of the two worlds.

The first version of YAGO [Suchanek et al., 2007] extracted facts mainly from the category names of the English Wikipedia. With the first upgrade of YAGO in 2008 [Suchanek et al., 2008], the project started extracting also from the infoboxes. In 2010, the YAGO team started working on the extraction of temporal and geographical meta-facts, which resulted in YAGO2 [Hoffart et al., 2011a; Hoffart et al., 2013]. The system architecture was completely restructured for YAGO2s [Biega et al., 2013a] in 2013. This helped to develop YAGO3 [Mahdisoltani et al., 2015], which added extraction from 10 different Wikipedia languages in 2015.

In 2014, I contributed to the project by developing regular expressions for extracting literals from infobox attributes. In the next year, I joined the project as a PhD student. My first task was to set up and to manage the evaluation of the YAGO3 release. I contributed by correcting errors in the extraction algorithms, and by optimizing their performance and memory footprint. I also extended tools which are used internally. In 2017, I took an active part in creating and publishing the YAGO 3.1 release, and in making the YAGO source code available to the public.

3.4 Content

YAGO facts follows the RDF model, which was described in Chapter 2. Facts are represented by triples of a subject, a predicate, and an object. An example is

<Barack_Obama> <wasBornOnDate> "1961-08-04"^^xsd:date.

In YAGO, entities (such as <Barack_Obama>) and relations (such as <wasBornOnDate>) have human readable names. YAGO also gives each fact a *fact identifier*. The identifier is generated by combining hash values of the subject, predicate, and object. For example, the above fact has the fact identifier <id_1km2mmx_1xk_17y5fnj>. This fact identifier

theme	fact count	theme	fact count
TAXONOMY	95 422 928	MULTILINGUAL	787 651
SIMPLETAX	17 402 744	LINK	4 623 709
CORE	55 509 326	WIKIPEDIA	296 661 088
GEONAMES	39 619 387	OTHER	471 429 325
META	203 977 824	total	1 185 433 982

Table 3.1 – Number of facts in themes of YAGO3

allows YAGO to state temporal or spatial information, or the origin of facts. We can say, e.g., that the above fact was extracted from the English Wikipedia page about Barack Obama:

<id_1km2mmx_1xk_17y5fnj> <extractionSource>
<http://en_wikipedia_org/wiki/Barack_Obama>.

This fact, in turn, has the fact identifier <id_17n9463_115_1wydgxv>, which is used to include provenance information. This second fact identifier allows YAGO to express that the fact about the birth date of Barack Obama was extracted from the infobox of his English Wikipedia page, using the extractor named InfoboxExtractor:

<id_17n9463_115_1wydgxv> <extractionTechnique>
"InfoboxExtractor from <infobox/en/birthdate>"

YAGO covers topics of general interest, such as geographical entities, personalities of public life or history, movies, and organizations. For this, YAGO uses a manually predefined set of 76 relations. As of YAGO3, the knowledge base contains 16 927 153 entities, and 1 185 433 982 triples in total. The triples are partitioned into *themes*. Themes allow research projects that need only a certain subset of facts, to downloaded them separately. YAGO has the following groups of themes (number of triples in parentheses, m: million, k: thousand):

— TAXONOMY: taxonomy-related facts; the class hierarchy (570 k), types (16 m), their transitive closure (78 m), and schema information (486).
— SIMPLETAX: a simplified taxonomy with just three layers. It contains the leaf levels of the WordNet taxonomy, the main YAGO branches (person, organization, building, artifact, abstraction, physical entity, and geographical entity), and the root node owl:Thing.
— CORE: the main facts, i.e., relations between entities (5 m), facts with dates (3 m), facts with other literals (1 m), and labels (45 m)

— GEONAMES: facts imported from GeoNames, mainly types, labels, and coordinates of geographic entities.

— META: facts about facts, i.e., facts about the extraction source (201 m), as well as their time and location (2 m)

— MULTILINGUAL: labels for classes in various languages from the Universal WordNet (788 k)

— LINK: links to other knowledge bases, notably to DBpedia (4 m), GeoNames (117 k), and WordNet identifiers (156 k)

— WIKIPEDIA: raw information from Wikipedia in RDF, which other projects can use to avoid parsing of Wikipedia. YAGO provides infobox attributes of entities (72 m), the infobox templates that an entity has on its Wikipedia page (5 m), the infobox attributes per template (262 k), Wikipedia-links between the entities (63 m), and the source facts for all of these.

— OTHER: redirect links and hyperlink anchor texts from Wikipedia (471 m)

3.5 Construction of YAGO

In the previous section, we what kinds of facts YAGO contains. Now we will see how these facts were extracted from YAGO's sources.

3.5.1 Sources

YAGO derives its instances from Wikipedia pages and GeoNames entities. Its classes stem from Wikipedia categories, WordNet synsets [2] [Fellbaum, 1998], and GeoNames classes. The instances of these classes follow the form <wikicat____>, <wordnet__>, and <geoclass__> respectively.

Wikipedia. Most of the information in YAGO comes from Wikipedia, the community-driven online encyclopedia. Wikipedia contains not just textual material, but also a hierarchical category system, and structured data in the form of *infoboxes*. As a rule of thumb, each Wikipedia page becomes an entity in YAGO. Facts about these entities are created mainly from Wikipedia Infoboxes, using a set of manually compiled mappings from Infobox attributes to YAGO relations. The types of entities are extracted from the Wikipedia leaf level categories. The upper part of the Wikipedia class hierarchy is discarded.

2. set of synonymous words associated to one meaning

Temporal Knowledge. In previous sections we simplified YAGO's knowledge harvesting approach by assuming that facts do not change over the time. This is appropriate for some relations, e.g., for birth dates. However, it is not sufficient for the facts that are valid during a certain time period, such as presidencies, marriages, or playing in a football club. In this part, we go beyond entity-relationship style facts, and describe the YAGO approach to attach temporal meta-information to facts that change over time. In other words, we capture the time spans during which particular relationships hold.

YAGO extracts the time span of facts by hand-crafted regular expressions from the Wikipedia infoboxes and categories. For example, from the infobox excerpt from Cristiano Ronaldo's Wikipedia page

| years2 = 2003–2009 | clubs2 = [[Manchester United F.C.]]

YAGO extracts the the start time and end time of the fact

<Cristiano_Ronaldo> <playsForTeam> <Manchester United F_C_>

This fact has the fact identifier <id_12bk998_1ul_134t38k>. YAGO stores time points as `xsd:date` literals attached to the fact id of the original fact. Thus, start time and end time are expressed as meta facts:

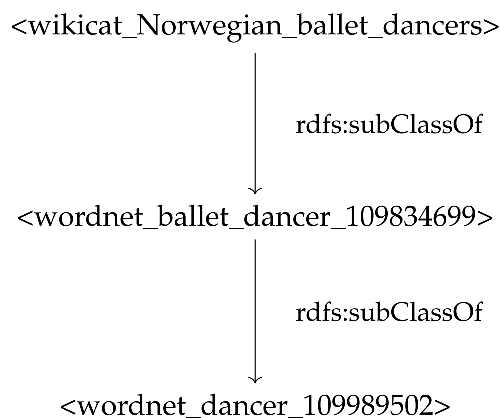
<id_12bk998_1ul_134t38k> <since> "2003-##-##"^^xsd:date.
<id_12bk998_1ul_134t38k> <until> "2009-##-##"^^xsd:date.

Dates are denoted in the standard format YYYY-MM-DD (ISO 8601 [3]) by using the xsd:date data type. If a date contains only the year and month, YAGO uses placeholders, as in "2003-12-##" with # as a wildcard symbol.

YAGO extracts temporal information from Wikipedia categories in a similar manner. YAGO3 attached temporal information to 2 715 985 facts with a 95% precision.

WordNet. The WordNet knowledge base [Fellbaum, 1998] is a lexical database of the English language [Miller, 1995]. Among other things, it defines a taxonomy of nouns, e.g., "ballet dancer" is a hyponym of "dancer". YAGO takes the leaves of the Wikipedia category hierarchy, and links them to WordNet synsets. This yields, e.g.

3. `https://en.wikipedia.org/wiki/ISO_8601`



YAGO includes WordNet Domains [Magnini and Cavaglia, 2000], which groups words into 167 thematic domains, and allows, e.g., searching for entities related to "computer science". The Universal WordNet [de Melo and Weikum, 2009] extends WordNet to over 200 languages, and YAGO uses it to add labels in many languages to the WordNet classes in YAGO.

GeoNames. The GeoNames knowledge base [4] contains 7 million geographical entities, such as villages, cities, and notable buildings. It contains a class hierarchy and facts, such as <locatedIn> facts for cities and countries. Each entity is classified with exactly one type, which enables their integration into the YAGO type system. GeoNames provides links to Wikipedia for about 117 000 GeoNames entities, which we use to map the entities to YAGO entities. The GeoNames classes are mapped to WordNet classes by a heuristic. Potential matching candidates are all WordNet classes with a geographical meaning. YAGO chooses the WordNet class with the highest token-overlap with the description of the GeoNames class. The precision of this matching heuristic is 94%, with a recall of 87% [Hoffart et al., 2013].

3.5.2 Extraction Process

Architecture. In YAGO, an *extractor* is a small code module that is responsible for a single, well-defined extraction subtask. An extractor takes certain themes as input, and produces certain themes as output. Therefore, the architecture of the YAGO extraction system can be represented as a bipartite graph of extractors and themes. This architecture allows for parallelization of the extraction process: Each extractor provides a list of input themes, and a list of output themes, and each extractor is started by a scheduler as soon as its input becomes available [Biega et al., 2013a]. The YAGO team provides an

4. http://www.geonames.org/

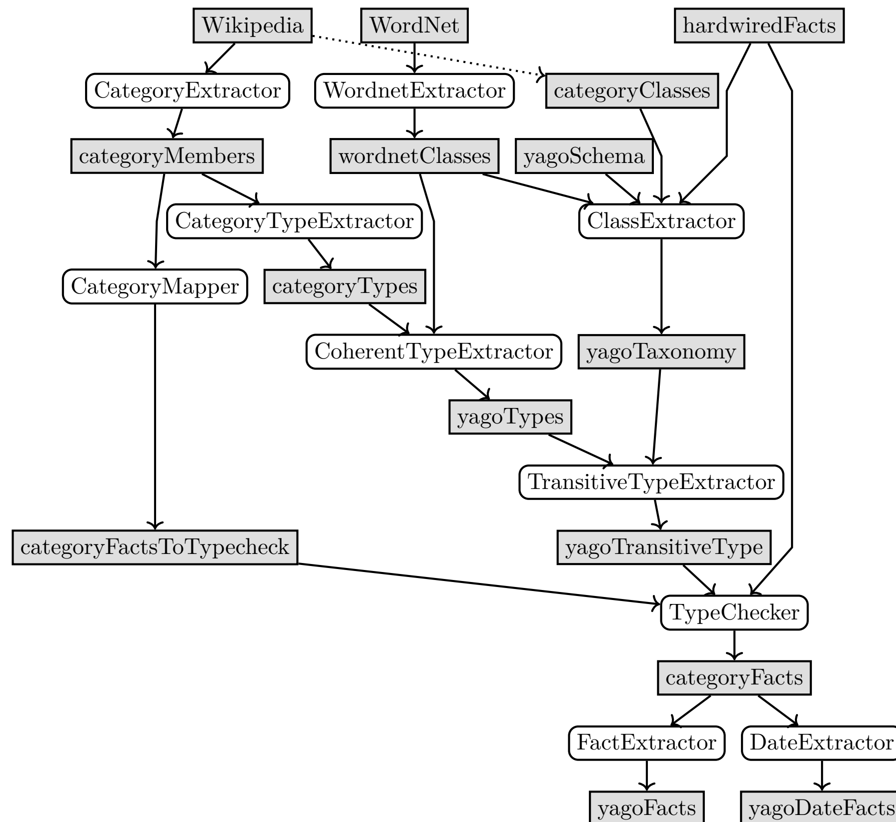


Figure 3.1 – Simplified excerpt of the architecture of YAGO. Gray boxes are themes and input files. White boxes are extractors. Only the themes with the prefix "yago" are part of a YAGO release. An arrow from an extractor to a theme means that the theme was produced by the extractor. An arrow from a theme to an extractor means that the theme is an input for the extractor. The dotted arrow represents a path with several themes and extractors.

online demo for exploring the architecture of a previous version[5] [Biega et al., 2013b]. See Figure 3.1 for an excerpt of YAGO's architecture. It shows the extractors and the intermediate themes for extracting facts from the categories in Wikipedia. These facts include, for example, facts from the relations <wasBornOnDate> and <isCitizenOf>.

The dependency graph of extractors is a directed acyclic graph (DAG). Take for example the following path of the DAG in Figure 3.1: *CategoryExtractor* → ··· → *TransitiveTypeExtractor* → *TypeChecker* → *FactExtractor*. First, the *CategoryExtractor* is executed, and the other extractors need to wait until it finishes. The previous version of YAGO would discover which themes are missing, and would regenerate those. Now imagine that a developer modifies the CategoryMapper to improve the extraction

5. http://resources.mpi-inf.mpg.de/yago-naga/yago/www2013demo/yago_demo_static/

of facts from the categories. She deletes the *categoryFactsToTypecheck* theme in order to recreate it. The previous version of YAGO would update *categoryFactsToTypecheck*, but the *categoryFacts* theme would still contain the old data.

We have developed a new scheduling algorithm that solves this problem as follows. First, the algorithm sorts the extractors topologically into a list. In that list, an extractor depends only on the preceding extractors. Then, the algorithm iterates over the list, and checks for every extractor whether it needs to be rerun. That is the case, if any of its output themes is missing, or if any of its input themes was regenerated. In that way, the developer can ensure consistency between themes.

Filtering. While the initial extractors are responsible for extracting raw facts from the sources, the following extractors are responsible for cleaning these facts. The facts first undergo redirection, a process where entities are replaced by their canonical versions in Wikipedia. They are then de-duplicated, and sent through various syntactic and semantic checks. Most notably, the facts are checked for compliance with the type signatures of the relations [Biega et al., 2013a; Hoffart et al., 2011a; Hoffart et al., 2013; Kasneci et al., 2008; Suchanek et al., 2008].

This modular architecture proved useful when YAGO was made multilingual [Mahdisoltani et al., 2015]. Only 3 major new extractors had to be added for the translation of entities. After that, the translated facts undergo the same procedures as the facts obtained from the English Wikipedia [Mahdisoltani et al., 2015]. A novelty in the pipeline was the attribute mapping algorithm: for foreign Wikipedias, the mapping between Infobox attributes and YAGO relations is inferred automatically based on the original English mapping, the English facts, and the raw foreign facts.

3.5.3 Evaluation

Every major release of YAGO is evaluated for quality. Since there is no high quality gold standard of comparable size, this evaluation is done manually. Since the large number of facts in YAGO makes a complete manual evaluation infeasible, a random sample of facts is selected from every relation and evaluated. Only facts obtained by information extraction are evaluated (not, e.g., imported facts), with respect to the extraction source (Wikipedia). The statistical significance is verified by help of the Wilson interval [Brown et al., 2001]. The Wilson interval calculates a confidence interval, in which the true ratio lies with a probability α.

The YAGO team has developed a Web tool to carry out the evaluation. The Web tool presents a fact together with the relevant Wikipedia pages to a human judge. The judge clicks on “correct”, “incorrect”, or “ignore”, and proceeds to the next fact. As YAGO3 extracts facts from Wikipedias in several languages, we extended the tool during this

Figure 3.2 – Screenshot of the evaluation tool

YAGO3 Evaluation - Current Standings

Overall State of the Evaluation

98.07% of 4412 evaluations were judged to be correct. This gives a weighted average Wilson center of 95.03% (4.19 % width)

Evaluation Results for Relations

Evaluation Target	Evaluations	Correct	Ratio (%)	Wilson Center (%)	Wilson Width (%)	Progress
<happenedIn>	87	87	100	97.89	2.11	
<byTransport>	120	119	99.17	97.64	2.21	
<hasExpenses>	135	133	98.52	97.18	2.42	
<hasExport>	60	60	100	96.99	3.01	
<hasISBN>	59	59	100	96.94	3.06	
<exports>	58	58	100	96.89	3.11	

Figure 3.3 – Screenshot of the evaluation progress

thesis. The new version shows the Wikipedia pages of the corresponding language, and of the time of the Wikipedia dump. A screenshots is depicted in Figure 3.2. The Web tool also provides an overview of the evaluation progress, see Figure 3.3.

The last evaluation of YAGO was made in 2015 with version 3.0.2, and took two months. 15 people participated and evaluated 4 412 facts of 76 relations, which contain 60m facts in total. They judged 98% of the facts in the sample to be correct. Weighted by the number of facts, the Wilson interval has a center of 95% and a width of 4.19%. This means that the true ratio of correct facts in YAGO lies between 91% and 99%, with α = 95% probability. See `https://w3id.org/yago/statistics` for the complete statistics.

3.6 Infrastructure

Data format. YAGO is provided in two formats, Turtle (see Chapter 2) and TSV (Tab Separated Values). The Turtle format allows using YAGO with standard Semantic Web software, such as Apache Jena. Since Turtle does not support fact identifiers directly, it stores a fact identifier in a comment that precedes the fact. The TSV format allows users to easily import the facts into a database, or to handle the data programmatically. The format also allows storing fact identifiers as an additional column. The YAGO team provides a script for importing the TSV files into an SQL database.

We published the current YAGO version 3.1 in August 2017. Users can always download the latest version of YAGO from the web page of the Max-Planck Institute for Informatics [6]. The YAGO knowledge base is also published on Datahub [7]. The Creative Commons Attribution 3.0 License allows everyone to use YAGO, as long as the origin of the data is credited. YAGO follows the FAIR principles (Findable, Accessible, Interoperable, and Re-usable), thanks to the use of the standard Turtle format, its copious metadata, and its open license. In addition to the YAGO 3.1 release, the YAGO team made the source code available to the public [8] in 2017.

YAGO is an active research project, and the teams at the Max-Planck Institute for Informatics, and at Télécom ParisTech provide support and maintenance. Since every major revision of YAGO is evaluated manually, YAGO is updated in the rhythm of months or years.

Tools. The YAGO team provides several tools to explore the data in YAGO. A graph browser [9] visualizes an entity with its in- and outgoing edges arranged in a star shape. See Figure 3.4 on the next page for a screenshot. Users can navigate the graph by clicking on an entity. Edges with the same direction and label are grouped together. Flags indicate the origin of the particular fact. The SPOTLX browser (Subject, Predicate, Object, Time, Location, conteXt) [10] allows querying YAGO with spatial and temporal visualizations. Some example queries are provided online. Users can ask questions, such as "Which politicians born before 1900 were also scientists?". The Data Science Center of Paris-Saclay offers a SPARQL endpoint for YAGO [11], together with example SPARQL queries [12].

6. https://w3id.org/yago
7. https://datahub.io/dataset/yago
8. https://github.com/yago-naga/yago3/
9. https://w3id.org/yago/svgbrowser
10. https://w3id.org/yago/demo
11. https://w3id.org/yago/sparql
12. https://w3id.org/yago/dataset

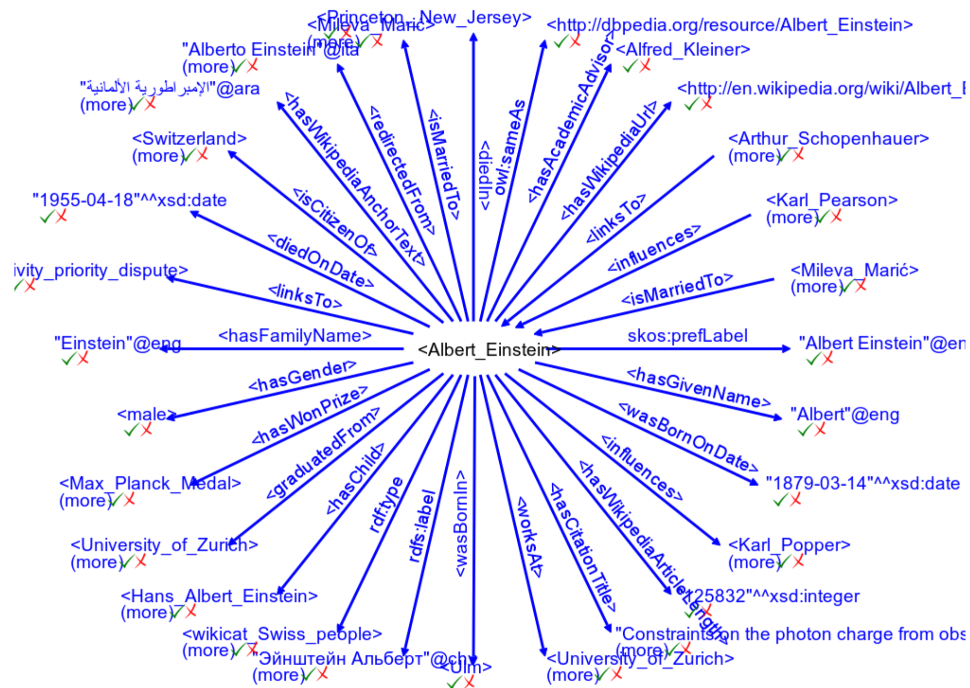


Figure 3.4 – Screenshot of the graph browser: facts of Albert Einstein in YAGO

3.7 Applications of YAGO

DBpedia. This project [Auer et al., 2007] is a community effort to extract a knowledge base from Wikipedia. The knowledge base uses two taxonomies in parallel: a hand-crafted one from its contributors, and the YAGO taxonomy. For this purpose, the `type` and `subclassOf` facts from YAGO are imported into a proper namespace in DBpedia.

IBM Watson. The Watson system [Ferrucci et al., 2010] can answer questions in natural language. It uses several data sources, among them the type hierarchy of YAGO. Watson participated in the TV quiz show *Jeopardy* together with human players, and was awarded the first place.

AIDA. The AIDA system [Hoffart et al., 2011b] can find names of entities in text documents, and map them to the corresponding YAGO entities. For example, in the sentence "When *Page* played *Kashmir* at *Knebworth*, his *Les Paul* was uniquely tuned.", AIDA recognizes the names in italics using a graph algorithm, and entity similarity measures. AIDA can understand that "Page" here refers to *Jimmy Page* of Led Zeppelin fame (and

not, e.g., to *Larry Page*), and that "Kashmir" means the song, not the region. Such a combination of structured and unstructured knowledge opens a new perspective to semantic analytics of text. For example, noun phrases, such as "the presidency of Obama" or "the second term of Merkel" [Kuzey et al., 2016]. The temporal intention of such phrases can be easily determined once the context of the particular phrases is leveraged by mapping them to the semantic components in the knowledge base. Thus, the input text is semantically and temporally tagged. In this way, it is possible to gather statistics over implicit time periods, such as: What are the all demonstrations during the second term of Merkel?

Semantic Culturomics. YAGO has been used to annotate articles of the French journal *Le Monde* with entities from the knowledge base [Huet et al., 2013]. These annotations allow to compute statistics on entities over time, such as: What are the countries where many foreign companies operate (are mentioned)? What is the proportion of women mentioned in Le Monde, and how did it change over time? The combination of structured knowledge (from YAGO) and unstructured knowledge (from Le Monde) illustrates correlations not visible in these resources alone. In Chapter 4 we will discuss how to use YAGO's data for similar studies.

3.8 Outlook

For future work, YAGO could be extended along the following dimensions:

Release cycle. Reducing the manual work required for the evaluation could shorten the release cycle. Many relations, such as <isLocatedIn> or <wasBornOnDate>, retain almost all facts from the previous version. We could therefore reuse a part of the previously evaluated facts, and combine them with a manually evaluated proportional sample of the new facts. We plan to investigate how to assure the validity of the precision estimation.

Commonsense knowledge. This concept comprises the problem of knowing facts about everyday life and everyday objects (e.g., that spiders have eight legs), and being able to reason with them. It is an essential part in general intelligence, and still poses a major challenge. Properties of everyday objects and general concepts are of importance for text understanding, sentiment analysis, and even object recognition in images and videos [Tandon et al., 2015; Tandon et al., 2014]. Commonsense knowledge can also take the form of rules. For example, active sports athletes hardly ever hold political positions [Galárraga et al., 2015a; Galárraga et al., 2015b].

Textual extensions. Representing commonsense knowledge might require ternary and higher-arity relations (such as descriptions of events in time and space). Those can be cast into triples only by reification, making them tedious to query and work with. Finally, the textual source of the facts often contains additional subtleties that cannot be captured in triples.

3.9 Summary

YAGO is a knowledge base that unifies information from Wikipedia, WordNet, and GeoNames into a coherent whole. This chapter has described the sources, the extraction process, and the applications of YAGO.

Even though YAGO contains a huge number of facts about many different entities, it still represents only a tiny amount of the facts that hold true in the real world. In the next chapter, we see how we can extend the knowledge of YAGO.

Chapter 4

Extending Information about People in YAGO

As we have seen in the previous chapter, recent advances in information extraction have led to more and more data about the real world in structured form in the YAGO knowledge base. In this chapter, we study how data from YAGO can be used for case studies in the Humanities. More precisely, we use the YAGO knowledge base to study life expectancy, birth rates, the age at childbirth, and the pertinence of locations over time. We also discuss the information extraction methods that we used to make YAGO sufficiently complete for these analyses to work.

This chapter is based on the following publication:

> Rebele, T., Nekoei, A., and Suchanek, F. M. (2017a). Using YAGO for the Humanities. In *Proceedings of the Second Workshop on Humanities in the Semantic Web (WHiSe II) co-located with 16th International Semantic Web Conference (ISWC)*, volume 2014 of *CEUR Workshop Proceedings*, pages 99–110. CEUR-WS.org. URL: `http://ceur-ws.org/Vol-2014/paper-11.pdf` (workshop paper)

4.1 Introduction

In parallel to the development of knowledge bases, the Digital Humanities have established themselves as a research field in their own right, using computational methods to support researchers in their quest to understand history, language, or the arts. Recently, the Digital Humanities have found common ground with research on knowledge bases [Adamou et al., 2016; Rapti et al., 2015], with the insight that the data of knowledge bases can help deliver answers to questions in the Digital Humanities. In this chapter, we want to push this fruitful symbiosis a bit further, and investigate in how far data from YAGO can help study questions of history and society.

Our first observation is that the data in YAGO is very incomplete. For example, only half of the people in YAGO have a place of residence. Incompleteness is a general problem on the Semantic Web: In DBpedia [Lehmann et al., 2015a], only 0.2% of people have a gender, and in Wikidata [Vrandecic and Krötzsch, 2014], only 3% of people have father [Razniewski et al., 2016]. Therefore, our first challenge is to make the data more complete.

As described in Section 3.5, YAGO extracts its information by automated means from the online encyclopedia Wikipedia, and other sources. A large part of the incompleteness in YAGO stems from the fact that these extraction mechanisms miss important pieces of information in this process. In this chapter, we describe how these extraction mechanisms can be improved so that they distill even more information from the source. We show that our techniques improve the coverage of YAGO on certain attributes by up to 60%.

To show the usefulness of this new data, we then proceed to several case studies: We use the data in YAGO to trace the evolution of life expectancy across history, by gender. We use the data to refute the myth that full moon days see more births than other days. We trace the age at child birth over time. Finally, we use population size to track the pertinence of locations over the centuries. With these analyses, we show that the data from the Semantic Web can help shed light on questions of the humanities.

This chapter is structured as follows: Section 4.2 discussed related work. Section 4.4 presents our improved methods for information extraction, which we evaluate in Section 4.5. Section 4.6 presents our case studies, before Section 4.7 summarizes.

4.2 Related Work

The idea of using data from the Semantic Web to support the Digital Humanities is in its infancy [Adamou et al., 2016; Rapti et al., 2015]. Schich et al. [Schich et al., 2014] use Freebase to trace the birth and death locations of intellectuals. However, their study was limited to only 150,000 people, while we aim an order of magnitude higher. De La Croix et al [de la Croix and Licandro, 2012] trace the longevity of famous people across history. Likewise, their study was limited to the 300,000 people in the "Index bio-bibliographicus notorum hominum", while we aim to show the value of Semantic Web data, which is much more ample. Gergaud et al. [Gergaud et al., 2017] come closest to our approach: They build a database of 1,1m people from Wikipedia, and study the economic impact of these individuals. In this chapter, we show how to build a database that contains twice as many people from Wikipedia. We compare the coverage of our improved YAGO to DBpedia [Lehmann et al., 2015a], which was also derived from Wikipedia.

4.3 Problem Statement

Our goal is to increase the number of facts about people in YAGO. First, we want to increase the number of facts of birth and death dates. As we want to analyze historical tendencies, we aim to to extract at least the year dates. Next, we want to increase the coverage of the places of residence and nationality. We subsume those two properties, and speak of places of residence only. We interpret those locations as place of residence, where a person stayed for an extended period of time. This excludes vacations and business trips. Finally, we want to know the gender of each person. For our work, we avoid discussing gender identity and gender roles. Thus, with gender we mean a traditional categorization into male and female.

One particularity of YAGO is that it has a manually evaluated precision of 95% with respect to Wikipedia. This means that, statistically, only 1 triple out of 20 does not correspond to the facts in Wikipedia. All of our improvements have to respect this quality constraint.

4.4 Methods

As described in Chapter 3, YAGO's facts are extracted from the infoboxes in Wikipedia, and the category names. The extraction process uses a modular architecture, in which extractors produce themes. These themes are then post-processed by other extractors: they are cleaned, de-duplicated, and checked for consistency. This results in a sequence of themes of ever cleaner data, of which the final themes constitute the YAGO knowledge base. The improvements that we propose follow this schema: we propose to add new extractors, and link them into this process.

4.4.1 Gender

The infoboxes of Wikipedia do not mention the gender of a person. Therefore, earlier versions of YAGO did not have gender information. Gender was added only in YAGO3, and it was extracted from the occurrence of pronouns on the page. The *GenderPronoun-Extractor* (Algorithm 1, Lines 1-4) counts the number of occurrences of "he" and "she" in the articles. If the number of "he" is at least twice the number of occurrences of the word "she", and if the number of occurrences is at least 10, the gender is assumed to be male (and vice versa). This worked well, but it had a rather low coverage. Only 61% of people had a gender.

Algorithm 1 Extract gender (helper methods)

```
1: procedure GENDERPRONOUNEXTRACTOR(article)
2:     count occurrence of "he" and "she" in first paragraph
3:     if count > 10 then output most frequent gender
4: end procedure
5: procedure GENDERCATEGORYEXTRACTOR(article)
6:     if a category contains "male" or "female" then output gender
7: end procedure
```

We improved this method by making use of the gender-specific categories in Wikipedia. For example, Cleopatra is in the category *Female people from Alexandria*. We wrote the *GenderCategoryExtractor* (Algorithm 1, Lines 5-7), which considers every article x, and produces the fact x <hasGender> <female> if the article is in a category that contains the substring *female*. We proceed analogously for the male categories. This works well, but it still has a low coverage.

We improve upon this as follows: We wrote the *GenderNameExtractor* (Algorithm 2), which infers the gender from the first names of a person. The same first name ν may be associated to people of different gender – either because the name is used for both males and females, or because of errors in Wikipedia or the extraction process. Our goal is to determine whether ν is associated to one gender in the majority of cases. Let us say that our sample for name ν has a proportion of p% males. We want to know what is the proportion of male people with the name ν in the real world. For this purpose, we run a statistical test: We assume that our set of people with first name ν is a sample from the real world. Then we generalize the proportion of males in our sample to the proportion of males in the real world. We use the Wilson estimator for this purpose, with $\alpha = 5\%$. Only if the lower bound of the Wilson interval is at least 95%, we assign the name to the male gender (analogously for female). The values of α and 95% are those that the YAGO evaluation [Suchanek et al., 2007] used.

Algorithm 2 first defines a helper procedure which creates a map from first name to gender. The helper function extracts the first name and the gender of the articles using the *GenderCategoryExtractor* (Line 2-3). Then, it applies the statistical test on each first name (Line 5-9) to generate a mapping from first name to gender. Lines 12-15 list the function that is actually maps an article to a gender based on the first name, if possible.

All of these three extractors produce sets of facts in the YAGO framework. These are then collected by a fourth extractor, which assigns at most one gender to each person, giving priority to the *GenderCategoryExtractor*, followed by the *GenderNameExtractor* and the *GenderPronounExtractor* (Algorithm 3).

Algorithm 2 Extract gender (helper methods)

```
1: procedure MapNameToGender
2:     M ← empty mapping from first name to gender to count
3:     fill M by extracting first name, and gender using categories
4:     N ← empty mapping from name to gender
5:     for every first name v do
6:         if support for male or female > threshold then
7:             N[v] = male or female, respectively
8:         end if
9:     end for
10:    return N
11: end procedure
12: procedure GenderNameExtractor(article, mapping)
13:    v ← ExtractFirstName(article)
14:    if v in mapping then output mapping[v]
15: end procedure
```

Algorithm 3 Extract gender

```
1: mapping ← MapNameToGender()
2: for every article do
3:     if GenderCategoryExtractor(article) then output gender
4:     else if GenderNameExtractor(article, mapping) then output gender
5:     else if GenderPronounExtractor(article) then output gender
6:     end if
7: end for
```

4.4.2 Dates

YAGO harvests the birth dates and death dates of people from two sources: From the infoboxes of Wikipedia, and from the categories of Wikipedia. The infoboxes contain a list of attribute-value-pairs, of the form *date-of-birth = Jan 8, 1935*. As for the categories, there exists one category per birth year (e.g., *1935 births*). The years were extracted using regular expressions.

The algorithm for extracting dates proceeds in two steps: recognition and parsing. Recognition is the scanning of a document for text spans that specify a date. Parsing is the transformation of such a text span into a standardized format. With our improvements, YAGO also recognizes birth and death dates from categories containing “BC” or “century”, such as “123 BC births” and “2nd century BC deaths”. The previous version

did not parse years with less than three digits, as it would otherwise introduce many false positives. We noticed that these year dates often appear with the prefix "AD", or with the suffix "BC" or "BCE". We therefore added support for year dates, such as "AD 42" and "5 BC". Furthermore, we added support for dates like "12th-century" and "1st millennium".

Many Wikipedia pages contain more than one date in the infobox attributes and in the categories. However, people can have only one birthday. The previous versions of YAGO used the date from the infobox where available, and defaulted to the year from the category otherwise. The problem with this approach is that the extraction quality is higher for the categories than for the infoboxes. This is due to more varied date formats in the infoboxes (*01/08/1935, 08/01/1935, 8 Jan 1935*, etc.), and dates that not relevant, such as publication dates of references.

We illustrate the new algorithm based on the example of extracting birth dates, see Algorithm 4. First, we extract dates from infoboxes and categories as previously (Line 1-2). Then, we analyze the birth dates of each person (Lines 4-9). If there are category dates (Line 6), then we keep only those infobox dates in I whose year coincides with one category date in C. Finally, we output the first infobox date, and if there is none, then the first category date (Line 9). We proceed analogously for death dates.

Algorithm 4 Extract Birth Dates

```
1:  extract birth dates from infoboxes
2:  extract birth dates from categories
3:  for every person p do
4:      C ← birth dates of person p from categories
5:      I ← birth dates of person p from infoboxes
6:      if C ≠ ∅ then
7:          I ← infobox dates whose year matches a year in C
8:      end if
9:      output first of I, otherwise first of C
10: end for
```

4.4.3 Place of Residence

YAGO extracts the birth place, death place, and place of residence from the infoboxes of Wikipedia. The problem with this approach is that the data in the infoboxes is sparse. However, some categories give away the nationality or the place of residence of a person, as in *Egyptian queens regnant*.

Algorithm 5 Extract places of residence

```
1: M ← mapping from location name to locations
2: for every person do
3:     C ← collect categories
4:     L ← empty mapping from location to occurrence count
5:     for category c in C do
6:         N ← all location names that are a substring of c
7:         N ← loc. names of N that are not contained in another location name in N
8:         increment counters in L for locations whose location name is in N
9:     end for
10:    output most occurring location in L
11: end for
```

Therefore, we have written a new extractor that harvests also the categories of Wikipedia. The extractor is shown in Algorithm 5. We first compile a list of location names (Line 1) – in part from the Wikipedia list of demonyms [1] [2], and in part from the lists of empires on Wikipedia [3].

These lists build the basis for a mapping of a string of characters to (one or more) locations, such as “Roman Empire” to the location <Roman_Empire>. Some Wikipedia categories mention only a part of the location name. For example, Julius Caesar has the category “Ancient Roman generals”. We therefore extend the mapping as follows. If a location name contains general nouns, such as “Empire” or “Caliphate”, we remove it from the location name, and add it to the mapping. In the example, we would also map “Roman” to <Roman_Empire>. We manually fix problems with that mapping, e.g., “Roman Catholic” maps to the empty set. The final mapping assigns locations to 5896 location names.

We use this mapping to assign each category to a set of locations (Lines 5-9). We take a category, for example <Nobility of the Holy Roman Empire>, and search for all location names and parts that are contained in the category’s name (Line 6). That would be “Holy Roman Empire”, “Holy Roman”, “Roman Empire”, and “Roman”. It is obvious that we do not want to map it to the location <Roman_Empire>. Thus, we remove all those findings that occur within another one (Line 7). In the example, only the location name “Holy Roman Empire” remains after the removal. The application of the mapping to the remaining findings leads to the set of locations for that category.

Our extractor then scans the categories of a person x, and counts the number of times each location appears (Line 8). For every location y that appears most often, we create a

1. denotation of people living at a certain place
2. https://en.wikipedia.org/wiki/Demonym, and https://en.wikipedia.org/wiki/List_of_adjectival_and_demonymic_forms_of_place_names
3. https://en.wikipedia.org/wiki/List_of_empires

fact x <livedIn> y (Line 10). Our rationale is that the categories can neither determine the birth place nor the nationality reliably, but that they can at least indicate a place of residence.

> *Example (Place of residence):* We show how the algorithm works by an example. First we initialize the mapping M as follows:
>
> — M(“Roman Empire”) = {*<Roman_Empire>*}
>
> — M(“Holy Roman Empire”) = {*<Holy_Roman_Empire>*}
>
> — M(“German”) = {*<Germany>*}
>
> Let <Martin_Luther> be a member of the categories “People of the Holy Roman Empire”, “German male writers”, and “German translators”. We then iterate over the categories. For the category “People of the Holy Roman Empire”, we set N = {“Roman Empire”, “Holy Roman Empire”} (Line 6), and remove “Roman Empire” in the next step, as it is a substring of “Holy Roman Empire”. Then, we update the counter: L(*<Holy_Roman_Empire>*) = 1. The categories “German male writers”, and “German translators” both result in N = {“German”}, and therefore L(*<Germany>*) = 2. Finally, we output the locations with the highest count in L, resulting in the fact <Martin_Luther> <livedIn> <Germany>.

4.5 Results

In this section, we study the effect that our new extraction methods have. We consider two axes: Precision and coverage. For precision, we took a random sample of 100 facts per method, and checked it manually against Wikipedia, as is we usually do for YAGO (see Chapter 3). For coverage, we counted the number of unique people who have a certain attribute. We compared our coverage to the previously implemented methods of YAGO, run on the same version of Wikipedia. We also report the coverage in DBpedia, the other big knowledge base extracted from Wikipedia.

These numbers have to be taken with a grain of salt: First, there exist different versions of DBpedia, one for each Wikipedia language. We take here the English one (which is the largest). YAGO, in contrast, extracts from 10 Wikipedias. Second, the precision of DBpedia is not known. We did not want to make statements about the quality of DBpedia in place of its authors. Thus, our results should not be understood as a direct comparison.

Table 4.1 shows our results. YAGO contains 2,215,360 people. Our precision values are very good. The 2% wrong results for genders were exclusively due to type errors, not to the extraction itself. The extraction of residences also suffered from that problem. In addition, it produced 8% of anachronistic residencies (such as *German Empire* instead of

Extraction	YAGO before	YAGO now	Precision	DBpedia (en)
Birth Dates	1,651,528	1,687,943	100%	819,371 [4]
Death Dates	805,377	821,780	100%	322,212 [5]
Gender	1,354,763	1,983,737	98%	4,419 [6]
Place of Residence	1,179,355	1,916,695	98%	683,854 [7]

Table 4.1 – Coverage and precision of our methods

Germany). We counted them as correct for the purposes of our study. The precision of birth and death dates is extraordinary.

In addition, our methods have significantly increased the number of data points for all attributes that we consider. Genders, e.g., increased by 46%. Places of residence increased even by 62%.

4.6 Case Studies

In this section, we aim to show the usefulness of our data in case studies. Most of them deal with the evolution of age or population size over time. The generality of the case studies is limited by the data source, Wikipedia. The people that we can find on Wikipedia represent only a tiny fraction of the human population. Furthermore, the available information might be heavily biased, as the Wikipedia project originated in the Western culture.

4.6.1 Population Size over Time

We first show for each century, how many people of that century have facts in YAGO. We take all people with birth or death date before 2017 into account. We consider that the $(x+1)$-th century starts at year "x00", and we translate a year y BCE (before common era) to $-y$. This gives rise to a virtual year 0, reducing the actual number of years the 1st century by 1%.

The diagram is obtained by assigning every person to a century, based on the year of birth or death. If a person has both – a birth and a death date, we take the arithmetic average of both year dates, and assign the person to the century of that average year.

4. from the DBpedia statistics Web page
5. from the DBpedia statistics Web page
6. "Gender" dataset
7. union of birth-place, death-place, and residence from "Mapping based objects"

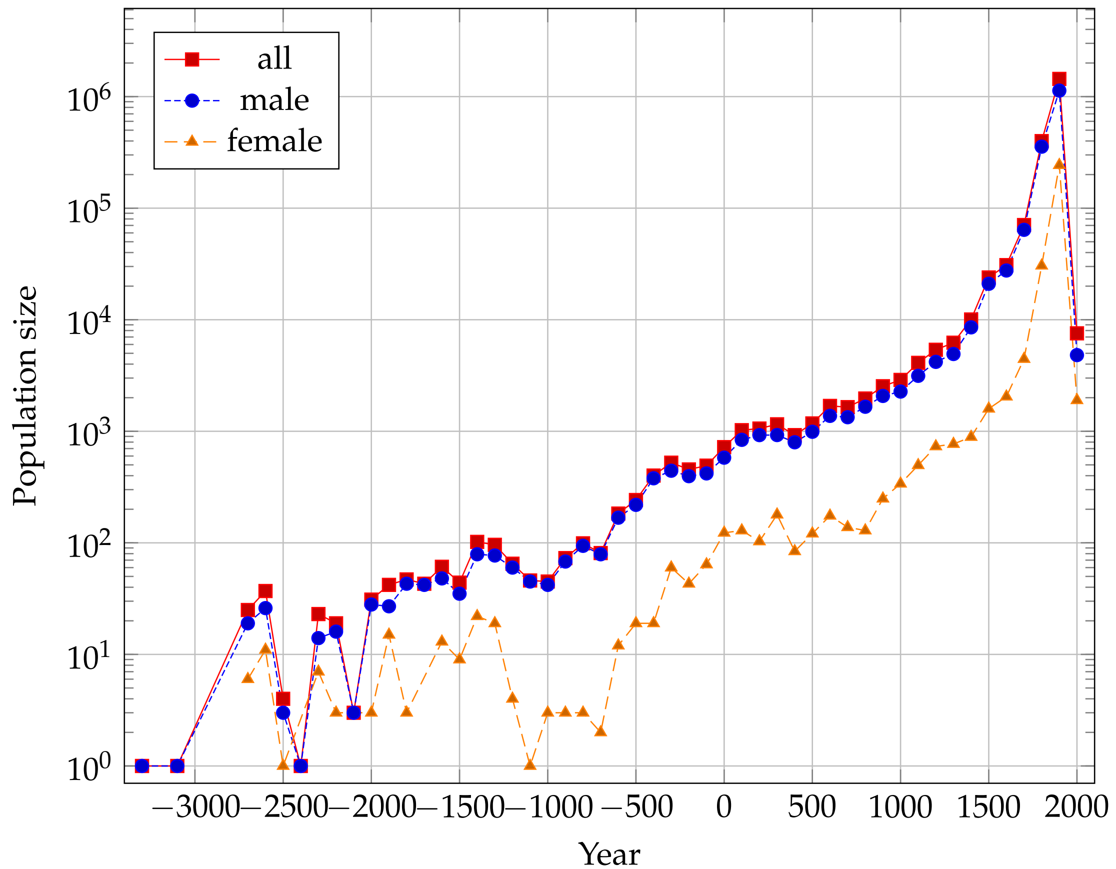


Figure 4.1 – Population size in YAGO

The result is shown in Figure 4.1. Naturally, YAGO knows only a few people from the early centuries 1000 BCE or earlier. Also, the number of female people in YAGO is only a fraction of male people. The drop in the 21st century – the century in which this thesis was written – looks astonishing at first glance, if one thinks about all the newspaper articles about new world population records. This drop can be easily explained by the following three reasons: First, that century has datapoints only from 17 years, thus a much smaller time span than the 100 years for the previous centuries. Second, people that will be important in the 21st century are still in their childhood, adolescence, or may not even be born yet. Finally, many people born at the end of the 20th century are alive, and are thus counted in that century.

4.6.2 Life Expectancy over Time

This case study (Figure 4.2) investigates the pattern of life spans across history. It does so by plotting the average life span of males and females against time. We restricted

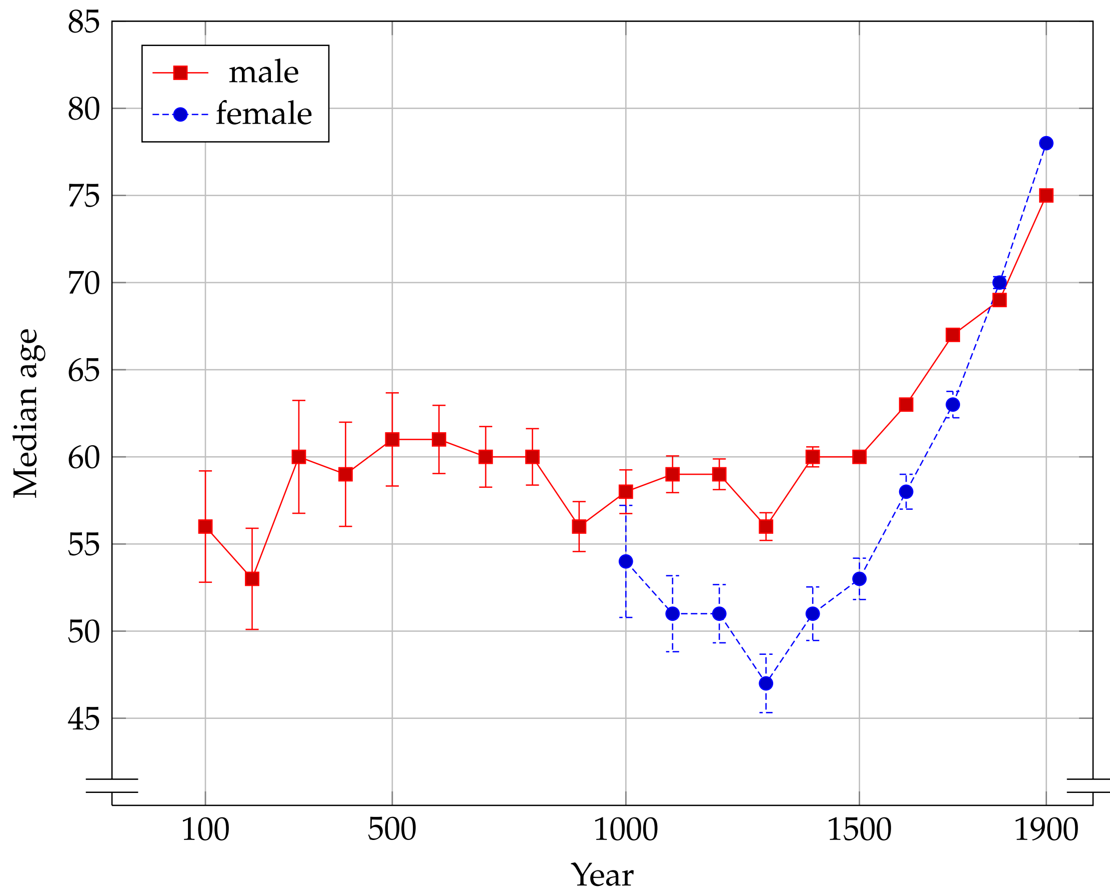


Figure 4.2 – Median age over time, by year of birth (with the Student's t confidence interval at $\alpha = 95\%$).

the study to centuries where we had more than 100 men and women, respectively. Interestingly, there is no trend in the data until the 15th century, with the average age fluctuating around 53 and 60 years for females and males, respectively. The effects of the Black Death are clearly mirrored in our data: life expectancy decreases in the 13th century.

Beyond that, there is a steady increase in the life span across genders during the last 500 years. We also see that women generally had a shorter life time in our data. This changes, however, in the 19th century: Women live longer than men. As our data is quite dense from the 19th century onwards, this fact is statistically significant in our data.

We can also drill down into the historical life expectancy per country. Figure 4.3 shows 4 countries that existed continuously over the past 1000 years, together with the median age of their population (only for centuries with more than 100 data points). The figure

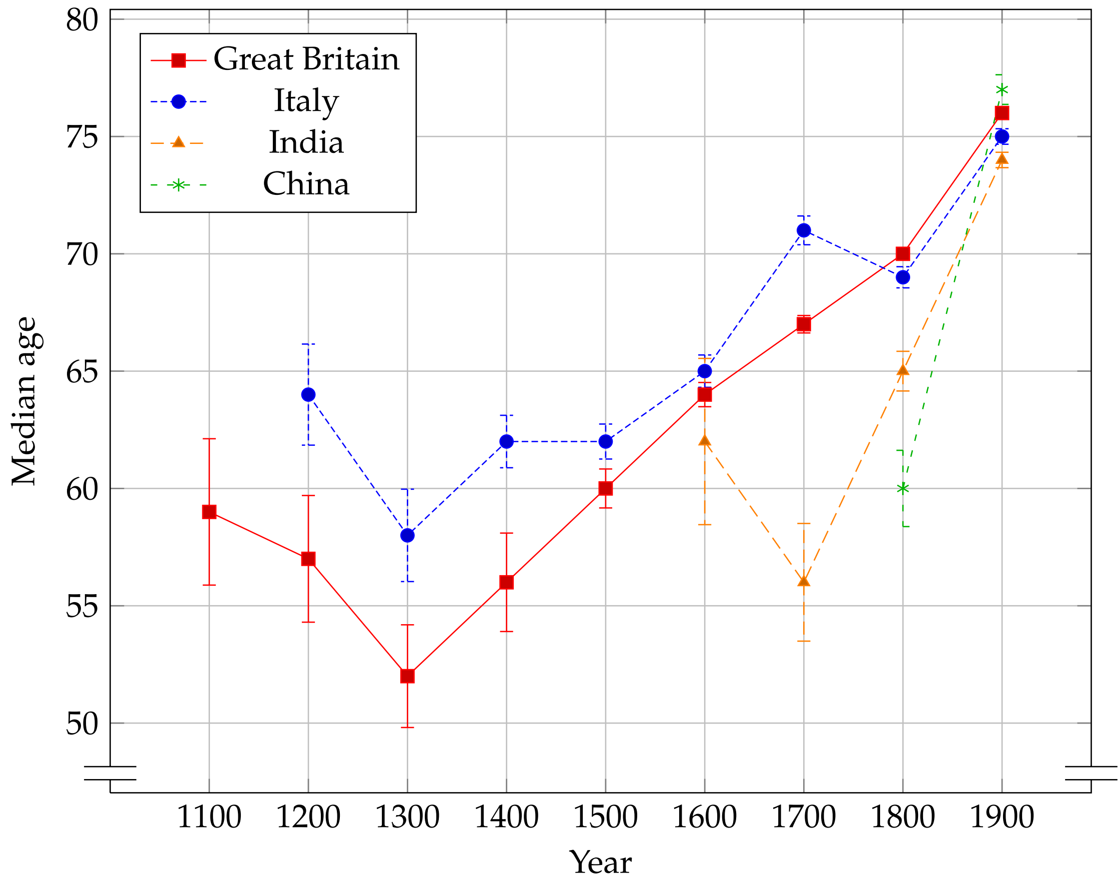


Figure 4.3 – Median age over time, by year of birth (with the Student's t confidence interval at $\alpha = 95\%$).

shows that the life span has been increasing since the Black Death in the two developed countries in the sample, Italy and Great Britain. We also see a catch-up effect: India and China have been experiencing a much larger increase in life span during the last 200 years. Another interesting observation is the take-over of Italy by Great Britain. This is mainly due to a deceleration of Italy rather that an acceleration of Great Britain, and it is thus difficult to argue that is related to industrial revolution. Today, all 4 countries have a comparable life expectancy in our data. This result has to be taken with a grain of salt: Wikipedia (and hence YAGO) contains mainly the elite population. Results may thus not generalize to the full population of a country. Our next study will shed light on that divergence.

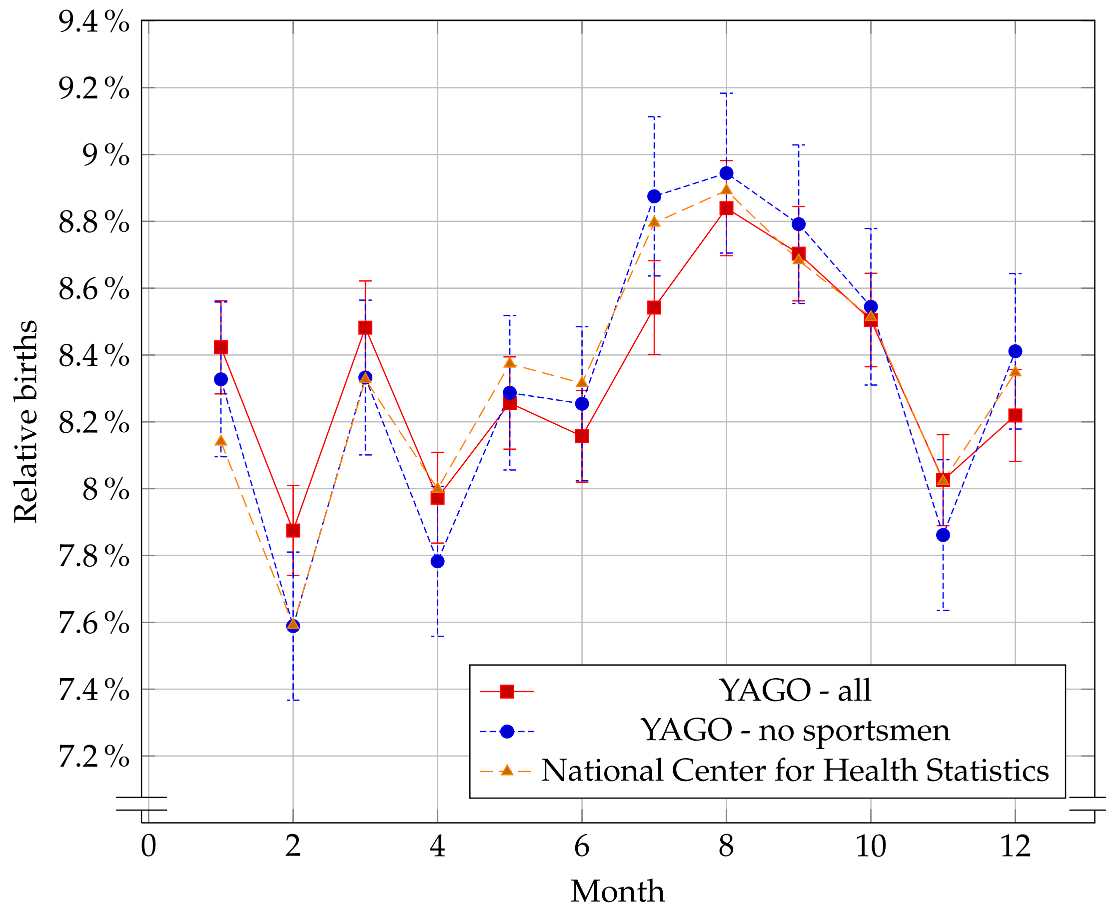


Figure 4.4 – Births per month (with the Student's t confidence interval at $\alpha = 95\%$).

4.6.3 Births per Month

Our next study tests whether the likelihood of making it to Wikipedia/YAGO is associated with the month of birth (Figure 4.4). For this purpose, we plotted the number of births in the US per month according to our data ("YAGO – all"), and compared it to the number of births per month according to the U.S. birth registry [8] ("National Center for Health Statistics"). We find that generally, both graphs show the same peaks. However, we find that people born in January are slightly more likely to be in Wikipedia than expected.

One potential explanation for this pattern is that our data comprises many sportsmen, who benefit from the relative age effect [9]: Sportsmen born early in the year are slightly

8. `http://abcnews.go.com/Health/Science/story?id=990641`, [US DHHS, 2017]
9. `https://en.wikipedia.org/wiki/Relative_age_effect`

older, and thus slightly more mature than sportsmen born later in the year with whom they usually compete. This makes them more successful, and hence slightly more likely to appear in Wikipedia. Indeed, if sportsmen are removed from the graph ("YAGO – no sportsmen"), our curve becomes more similar to the US census data.

4.6.4 Full Moon Myth

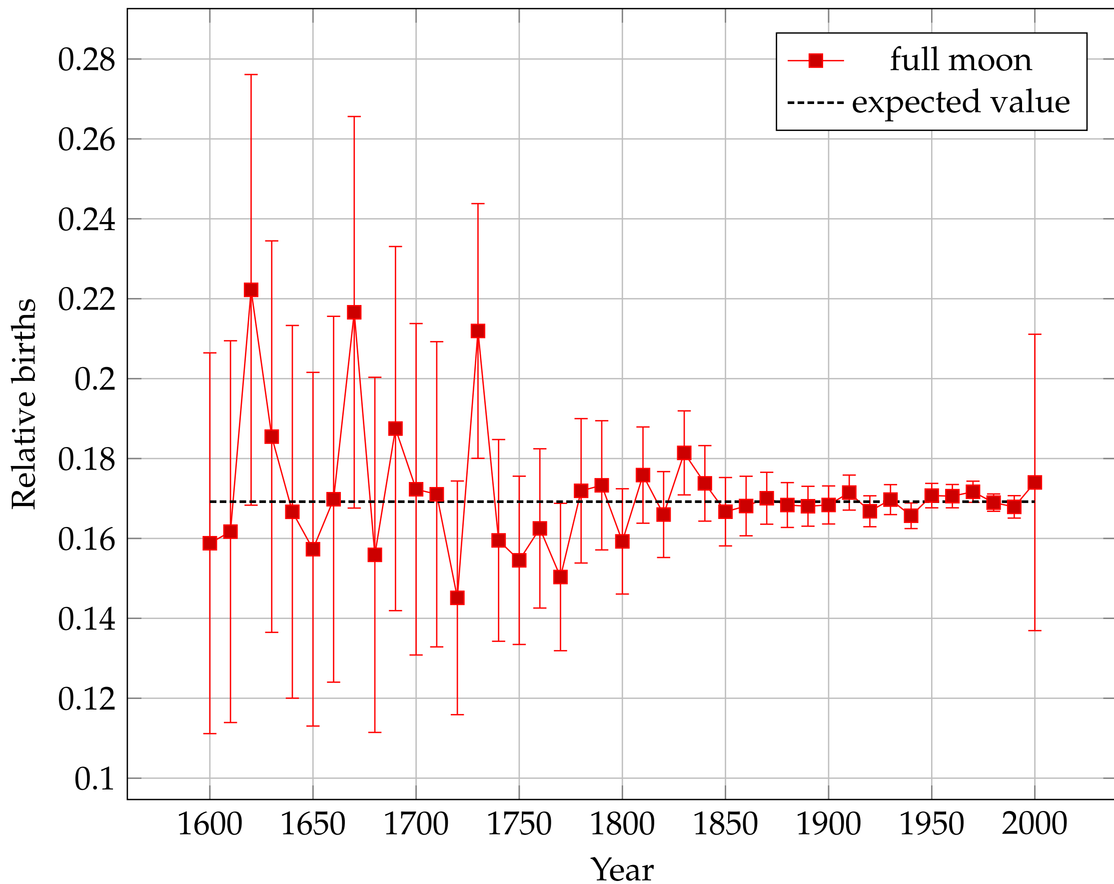


Figure 4.5 – Proportion of births on a full moon day (±2.5 days, with a Wilson score interval at $\alpha = 95\%$).

A popular myth has it that there is an increase in births on full moon days [10]. We wanted to analyze this conjecture with our data. For this purpose, we computed the time points of the full moon over the past centuries. We restricted our analysis to 1600 to 2000, as our full moon calculation bases on the Gregorian calendar introduced in 1582. We also

10. `https://www.google.com/#q=full+moon+birth`

did point checks with known historical full moon dates to make sure our computation is correct.

We consider the two days that precede, and the two days that follow a full moon day as "full moon days". Among the 146,463 days between 1600 and 2000, we classified 24,793 days as full moon days. This gives a total ratio of full moon days of 16.9%. During that time period, 691,616 people were born, of which 117,081 fall on a full moon day. This gives a total ratio of full moon births of 16.9% – exactly as expected. The Wilson score interval at $\alpha = 95\%$ is 16.9%±0.1%. Thus, the full moon has no influence on the birth rates in our data.

Additionally, we plotted the proportion of people born on or around a full moon day between 1600 and 2000 per century. Figure 4.5 shows our results. The great fluctuation seems to indicate a relationship between the period of the moon, and the frequency of births. However, this fluctuation is mainly due to sparse data. As the confidence interval shrinks, the proportion of full moon births converges to the expected value of 16.9%.

4.6.5 Age at First Child Birth

We now turn towards the age at which people become parents. Figure 4.6 on the next page shows the age at which people have their first and last child during the last millennium. Again, we restricted our analysis to centuries where we had at least 100 parent-child pairs.

Males show an increase in the age at which they have their first child, moving from around 28 to around 32. For females, we see similar story, with one important difference: for females, the increase is concentrated during the last 2 centuries. However, the age at which males have their last child is almost unchanged until 1700s, and declining since then. For females, the age at the birth of last child is unchanged.

We speculate that two demographic phenomena are creating these patterns. First, women having less children by mainly postponing the age at which they have their first child. Second, the age difference between fathers and mothers is declining. These two forces together can create an increasing first-child age of mothers, and a declining last-child of fathers with constant last-child age of mothers, and first-child of fathers.

4.6.6 Rise and Fall of Civilizations

In our last case study, we investigate how we can use YAGO's data for analyzing the rise and fall of civilizations. For the purpose of this section, a "civilization" is a location, such as countries, nations, states, or empires, that influenced humanity. That influence can express itself, among others, in science, art, politics, military, or sport. We assume

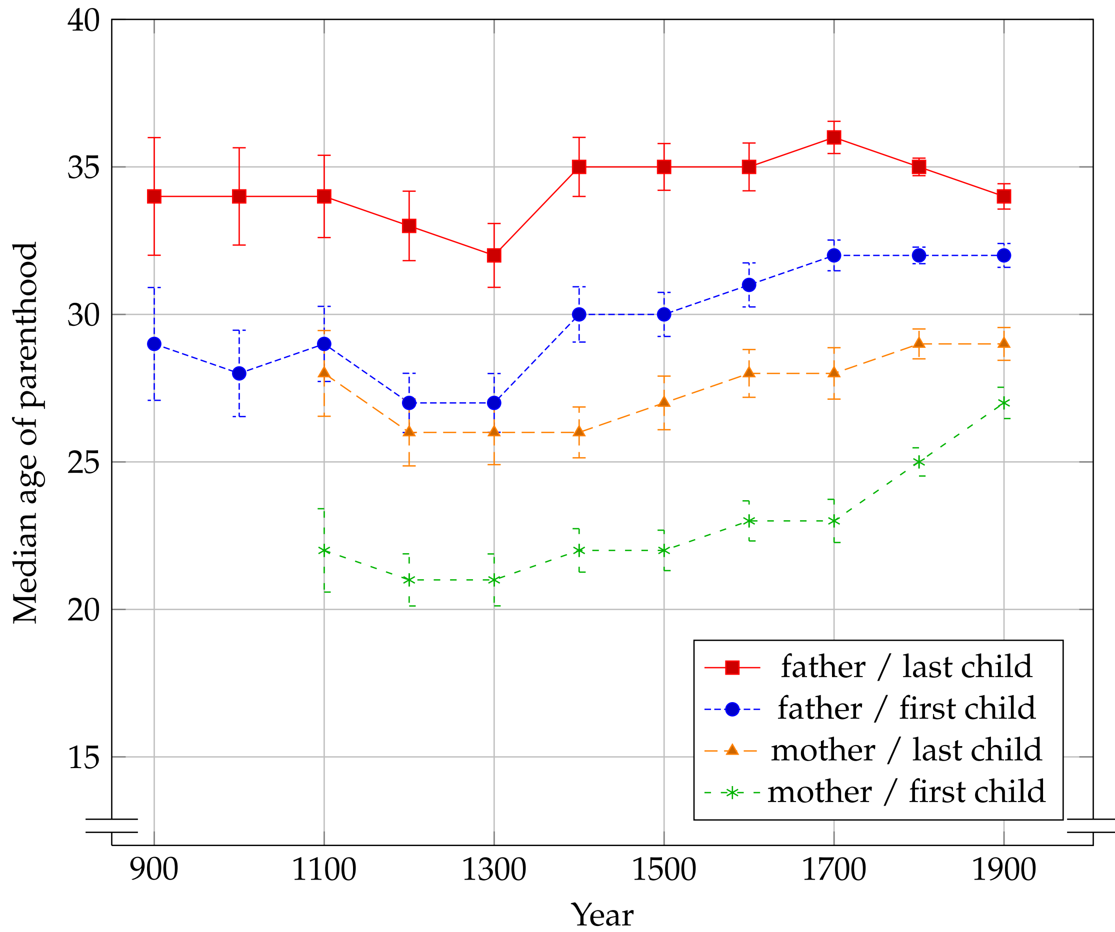


Figure 4.6 – Age of parenthood, by year of birth (with the Student's t confidence interval at α = 95%)

that a civilization with a higher influence has a higher chance to go down in history, and that a higher proportion of its citizens have their own article on Wikipedia. Therefore, we analyze for every century, in which civilizations the people in Wikipedia lived.

This case study faces the obstacle of assigning people to civilizations. The area of those civilizations change over time. They expand, shrink, divide, merge, or simply disappear. Cities could belong to several different civilizations over the centuries. Take for example Mainz. Today it is a German city, but previously it was a city in the Holy Roman Empire, and it began as a fortification in the Roman Empire. Thus, if YAGO tells us that a person is born in Mainz, we do not know her or his civilization.

We therefore take the following approach: we use the <locatedIn> attribute from the YAGO knowledge base to map locations to today's civilizations. We then search for lo-

cations that could not be mapped this way, and their population size. We then manually examined the 985 most populous of those locations, and assigned each location to a civilizations of our time, if that civilization contains the majority of the area of the location. We reuse the century assignments from Subsection 4.6.1.

The relative population size is the number of people living in one location in century x, divided by all people with known location that live in century x.

For readability, we show only the most pertinent locations. We aim to define pertinence in a way such that we include locations that had either a strong influence during a short period of time, or a medium importance for a long time. Therefore, we define

$$\text{pertinence}(y) = \sum_{\text{century } x} \text{relative population size of civilization } y \text{ in century } x$$

The pertinence measure of a civilization is an approximation of the area under the curve of the civilization in Figure 4.7 on page 60. We took the 11 most pertinent civilizations, and depicted the evolution of their relative population size in Figure 4.7. We cut off this list after 11 civilizations so that our study includes the United States, the civilization with the highest relative population size of our century. We removed the data points where a civilization has less than 1% relative population size.

In the diagram we can observe the succession of the most pertinent locations. We analyze the diagram with the help of Wikipedia [11]. Foremost, Egypt tops all other locations as the civilization with the highest pertinence, and also the first civilization that appears in our data. Egypt's dominance lasts until the end of the 10th century BCE. Starting from around 2000 BCE, two other civilizations in the Middle East establish themselves: the Babylonian Empire, and the Assyrian Empire, listed as "Syria" in the diagram. Those three civilizations decline in the Late Bronze Age collapse [12].

According to our data, China flourished from the 8th century to the 6th century BCE. This corresponds to the time of the Spring and Autumn period [13], during which the Zhou dynasty experienced a social change. Well-known Chinese philosophers, such as Confucius and Laozi, wrote influential works during this period.

The 6th century BCE marks the beginning of the Greek philosophy. The following centuries until the 3rd century BCE are known as "Classical Greece" [14]. This period advanced, among others, politics, philosophy, and literature. Well-known people from that time include Socrates, Plato, and Alexander the Great.

The ancient Rome [15] was a civilization from the 7th century BCE until the 5th century CE. It provided political stability in their vast territories ("Pax romana"). According

11. see pages in the "Centuries" category, `https://en.wikipedia.org/wiki/Category:Centuries`
12. `https://en.wikipedia.org/wiki/Late_Bronze_Age_collapse`
13. `https://en.wikipedia.org/wiki/Spring_and_Autumn_period`
14. `https://en.wikipedia.org/wiki/Classical_Greece`
15. `https://en.wikipedia.org/wiki/Ancient_Rome`

to our data, ancient Rome was most the most important civilization from the 1st century BCE until the 1st century CE. Historians still discuss the reason why ancient Rome declined in the 5th century [16].

The period from the 6th century to the 10th century CE is coeval with the Early Middle Ages [17], sometimes called the "Dark Ages". According to Wikipedia, Spain flourished during that time. Unfortunately, we cannot see that in our data. Many people living in that region at that time are not mapped to Spain, but to the Caliphate of Córdoba. Also, there were many small kingdoms, which our algorithm did include in the list of civilizations. Britain was the most prevalent civilization during that period according to our data, but we could not find a historical explanation.

In the High and Late Middle Ages [18], Europe consisted of monarchies, mainly kingdoms. Those in France, Italy, Britain, and Germany, were very influential, among them the Holy Roman Empire. Religion played a major role in society, and this period was also the time of the crusades [19]. The Gothic architecture [20] fits into the zeitgeist. The architecture lead to famous cathedrals, such as Notre Dame in Paris or the Cologne Cathedral.

The next period from the 15th to the 17th century was the Renaissance [21]. It started in Italy in the 15th century, which corresponds to the peak of Italy in our diagram. Then, the Renaissance spread over the European continent. This period was also the start of the modern history [22]. Modern nations and countries started to form. The United States declared their independence in the 18th century. In the following centuries, the United States thrived and became one of today's superpowers.

We compare our results with one of the few similar works that we could find: The Histomap [23] by John B. Sparks from 1931. We manually extracted a chronological list of the most prevalent civilizations: Egypt (20th century BCE - 12th century BCE); Assyria and Babylonian Empire (11th century BCE - 6th century BCE); Greece and Persia (5th century BCE - 3rd century BCE); ancient Rome (2nd century BCE - 3rd century CE); Mongolians (4th century - 5th century); Arabs, Byzantine Empire and Franks (6th century - 10th century); Holy Roman Empire (11th century - 12th century); Mongolians and

16. https://en.wikipedia.org/wiki/Fall_of_the_Western_Roman_Empire: "Alexander Demandt enumerated 210 different theories on why Rome fell"
17. https://en.wikipedia.org/wiki/Early_Middle_Ages
18. https://en.wikipedia.org/wiki/High_Middle_Ages, and https://en.wikipedia.org/wiki/Late_Middle_Ages
19. https://en.wikipedia.org/wiki/Crusades
20. https://en.wikipedia.org/wiki/Gothic_architecture
21. https://en.wikipedia.org/wiki/Renaissance
22. https://en.wikipedia.org/wiki/Modern_history
23. https://www.davidrumsey.com/luna/servlet/detail/RUMSEY~8~1~200375~3001080:The-Histomap-

Ottoman Turks (13th century - 16th century); France and Britain (18th century - 19th century); and finally, the United States (20th century).

We observe that our study misses some civilizations, such as the Mongolians and the Byzantine Empire. This is because we focused on the 11 most prevalent civilizations. Furthermore, some civilizations are under-represented in Figure 4.7, such as the Holy Roman Empire. Others are over-represented, such as Egypt before the 10th century BCE. Overall, our study manages to replicate the succession of the most prevalent countries quite well. While our studies are only preliminary, it shows that the data extracted with our algorithm can enhance historical studies.

4.7 Summary

In this chapter, we have investigated how data from the Semantic Web can help research in the Digital Humanities. More precisely, we have used the YAGO knowledge base to study the life expectancy of people across different times; the age at first childbirth; the myth that full moon days see more births; and the pertinence of civilizations over time. We have also presented methods to improve the coverage of YAGO. Our methods are integrated into the YAGO infrastructure, and the source code is included in the open source release of YAGO.

The extension of YAGO has relied partially on regular expressions. We have seen that even experienced developers have difficulties to develop regular expressions that match all intended strings. The expressions often have to be adapted, so that they match all strings that we intended to match. In the next chapter, we will see how to adapt regular expressions automatically.

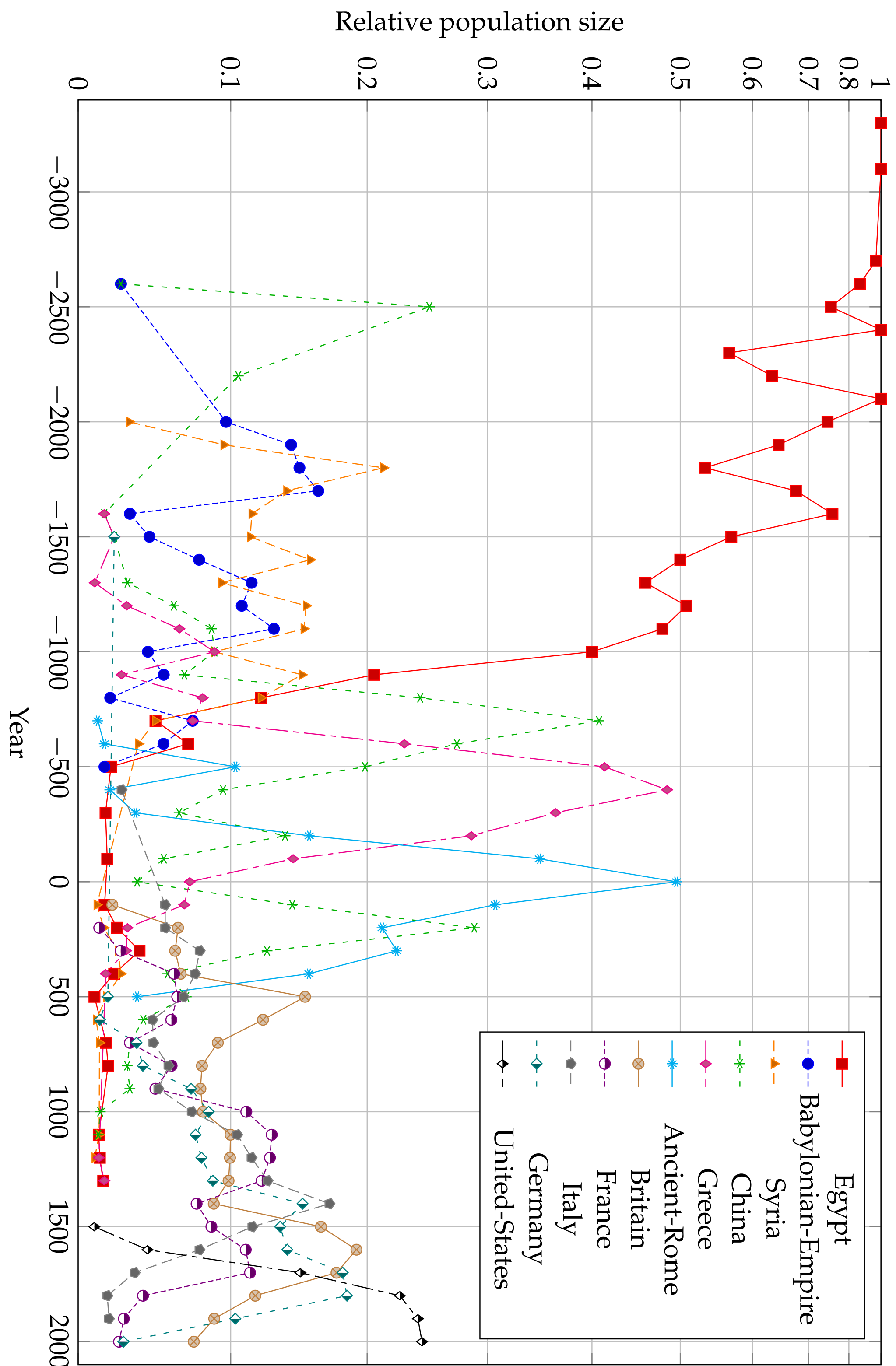


Figure 4.7 – Relative population size, by century. The y-axis is scaled by a quadratic function.

Chapter 5

Adding Missing Words to Regular Expressions

In the previous chapter we have encountered the task of extracting dates from text by regular expressions. We have seen that even hand-crafted regular expressions may fail to match all the intended words. This problem appears not just in YAGO, but in many data-intensive applications that use regexes for information extraction.

In this chapter, we propose a novel way to automatically generalize a regular expression so that it matches a set of given words. Our method finds an approximate match between the missing word and the regular expression, and adds disjunctions for the unmatched parts appropriately. Our goal is to improve the recall of the initial regular expression without deteriorating its precision. We show the effectiveness and generality of our technique by experiments on various information extraction datasets.

This chapter is based on the following publications:

> Rebele, T., Tzompanaki, K., and Suchanek, F. M. (2017b). Visualizing the addition of missing words to regular expressions. In *Proceedings of the ISWC Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC)*, volume 1963 of *CEUR Workshop Proceedings*. CEUR-WS.org. URL: `http://ceur-ws.org/Vol-1963/paper457.pdf` (demonstration)
>
> Rebele, T., Tzompanaki, K., and Suchanek, F. (2018b). Adding Missing Words to Regular Expressions. In *Advances in Knowledge Discovery and Data Mining - 22nd Pacific-Asia Conference, PAKDD, Proceedings*, Lecture Notes in Computer Science. URL: `https://www.thomasrebele.org/publications/2018_pakdd.pdf` (full paper)
>
> Rebele, T., Tzompanaki, K., and Suchanek, F. (2018c). Technical Report: Adding Missing Words to Regular Expressions. Technical report, Telecom ParisTech. URL: `https://hal.archives-ouvertes.fr/hal-01745987v1` (technical report)

5.1 Introduction

Regular expressions (regexes) are a popular tool for extracting, processing, and categorizing text spans. They find applications in information extraction, DNA structure descriptions, document classification, spell-checking, spam email identification, deep packet inspection, or in general for obtaining compact representations of string languages. To create regexes, several approaches learn them automatically from example words. These approaches take as input a set of positive and negative example words, and output a regular expression.

However, in many cases, the regexes are hand-crafted. Information extraction projects, such as DBpedia [Lehmann et al., 2015a], GATE [Cunningham et al., 2002], SystemT [Krishnamurthy et al., 2008], and YAGO all rely (also) on manually crafted regexes, as we have seen in the previous chapter. These regexes have been developed by human experts over the years. They form a central part of a delicate ecosystem, and most likely contain domain knowledge that goes beyond the information contained in the training sets.

One of the challenges in such settings is to correct a regex if it does not match a word that it is supposed to match. Take as example the following simple regex for phone numbers: \d{10}. After running the regex over a text, the user may find that she missed the phone number 01 43 54 65 21. An easy way to repair the regex would be to add this number in a disjunction, as in \d{10}|01 43 54 65 21. Obviously, this would be a too specific solution, and any new missing words would have to be added in the same way. A more flexible repair would *split* the repetition in the original regex and *inject* the alternatives, as in (\d{2} ?){4}\d{2}.

In this chapter, we propose an algorithm that achieves such generalizations automatically. More precisely, given a regex and a small set of missing words, we show how the regex can be modified so that it matches the missing words, while maintaining its assumed intention. This is a challenging endeavor, for several reasons.

First, the new word has to be mapped onto the regex, and there are generally several ways to do this. Take, e.g., the regex <h1>.*</h1> and the word <h1 id=a>ABC</h1>. It is obvious to a human that the id has to go into the first tag. However, a standard mapping algorithm could just as well map the entire string id=a>ABC onto the part .*. This would yield <h1>?.*</h1> as a repair – which is clearly not intended. Second, there is a huge search space of possible ways to repair the regex. In the example, <h1(>.*| id=a>ABC)</h1> is certainly a possible repair – but again clearly not the intended one.

Finally, the repair itself is non-trivial. Take, e.g., the regex (abc|def)* and the word abcabXefabc. To repair this regex, one has to find out that the word is indeed a sequence of abc and def, except for two iterations. In the first iteration, the last character of abc is missing. In the second iteration, the first character of def has to be replaced by

an X. Thus, the repair requires descending into the disjunction, removing part of the left disjunct and part of the right one, before inserting the X into one of them, yielding (abc?|[dX]ef)* as one possible repair.

Another difficulty in repairing a regular expression is that existing approaches typically require a large number of positive examples as input in order to repair or learn a regex from scratch. This means that the user has to come up with a large number of cases where the regular expression does not work as intended – a task that requires time, effort, and in some cases continuous user interaction (see Section 5.2 for examples). We want to relieve the user from this effort. Our approach requires not more that 10 non-matching words to produce meaningful generalizations. The contributions described in this chapter are as follows:

— we provide an algorithm to generalize a given regex by appropriately adding unmatched substrings to the regex;

— we show how such repairs can be performed even if the set of provided examples is small;

— we show how to extended our approach to improve the quality of the repaired regex, while reducing its length at the same time;

— we run extensive and comparative experiments on standard datasets, which show that our approach can improve the performance of the original regex in terms of recall and precision.

This chapter is structured as follows. Section 5.2 starts with a survey of related work. Section 5.3 introduces preliminaries. Section 5.4 presents the basic algorithm, which is improved in Section 5.5. Section 5.6 shows their experimental evaluation. Section 5.7 presents a demonstration tool, before Section 5.8 recapitulates.

5.2 Related Work

In this chapter we consider repairing regular expressions that fail to match a set of words provided by the user. We want to modify only the regex, and we assume that our input examples are correct. Thus we do not deal with data cleaning [Raman and Hellerstein, 2001; Li et al., 2016]. We discuss work relevant to our problem along three axes: (1) matching regular expressions to strings, (2) automatic generation of regular expressions from examples, and (3) transformation of an existing regular expression based on examples.

Regex matching. Many algorithms (e.g., [Ficara et al., 2008]) aim to match a regex efficiently on a text. Another class of algorithms deals with matching the input regex to a given input word – even though the regex does not match the string entirely [My-

ers and Miller, 1989; Knight and Myers, 1995]. Other algorithms for approximate regex matching optimize for efficiency [Wu et al., 1995; Navarro, 2004]. We build on [Myers and Miller, 1989; Knight and Myers, 1995] for approximate regex matching, which itself builds on the well-known work of Wagner-Fisher [Wagner and Fischer, 1974] for string-to-string matching. Other algorithms for approximate regular expression matching [Wu et al., 1995; Navarro, 2004] optimize for efficiency.

Different from all these approaches, our work aims not just to match a regex, but also to repair it.

Regex learning. Several approaches allow learning a regex automatically from examples. One approach [Brauer et al., 2011] uses rules to infer regular expressions from positive examples for entity identifiers. In this work, the authors observe that most entity identifiers share certain patterns in the beginning or the end. Thus, the proposed method constructs suffix and prefix automata for the input examples, and uses the minimum length principle to choose between the alternative patterns.

Other work [Bartoli et al., 2012; Bartoli et al., 2014; Bartoli et al., 2016] uses genetic programming techniques to derive the best regular expression for identifying given substrings in a given set of strings. The fitness of the derived regular expression is measured in terms of edit distance and regular expression size (length). Both negative and positive examples are required for the training phase.

The work presented in [Prasse et al., 2015] follows a learning approach to derive regular expressions for spam email campaign identification. The training requires sets of strings along with representative regular expressions designed by experts. Here, the given regular expressions are the correct target values used for the cost function minimization. This is different from our setting where a given regular expression has to be improved.

All of these works take as input a set of positive and negative examples, for which they construct a regular expression from scratch. In our setting, in contrast, we want to repair a given regex. Furthermore, we have only very few positive examples.

In the slightly different context of combining various input strings to construct a new one, the work of [Gulwani, 2011] proposes a language to synthesize programs, given input-output examples.

In the same spirit, the authors of [Le and Gulwani, 2014] proposed an interactive framework in which users can highlight example subparts of text documents for data extraction purposes. In case of dis-accordance with the proposed extracted entities, the user can propose more positive or negative examples in an attempt to refine the produced extraction. The framework is based on an algebra with extraction operators, which can be used to create an extraction language for each corresponding file type (such as web-

pages or spreadsheets). The extraction algorithm relies on the hierarchical declaration of the extracted regions, and the structure of the text.

Our problem, however, also takes advantage of the expert's knowledge of the domain; thus we build our regex inference model both on (positive and negative) examples, and on an initial regex provided by an expert.

Regex transformation. There are several approaches that aim to improve a given regex. The orthogonal problem of enhancing the *precision* of a regular expression has been addressed in [Li et al., 2008]. As in our setting, the goal is to maximize the F-measure of the regular expression. Li et al.'s system takes as input a set of positive and negative examples as well as an initial regular expression to be improved. The authors propose an algorithm that restricts the language of an initial regex by restricting quantifiers and character classes, or by introducing negative dictionaries. The transformation steps consist in repeatedly restricting the language of sub-expressions of the initial regex until no further improvement can be observed. This work makes the regex match less words. Our goal is different: We aim to *relax* the initial regular expression, so that it covers words that it did not match before.

Similar to us, the work of [Murthy et al., 2012] attempts to relax an initial regular expression. The approach generalizes character classes and quantifiers. It enumerates all possible sub-expression relaxations, and checks whether the desired precision is maintained above a threshold while the coverage is improved. Different from our approach, this work requires user feedback and statistics over the document words. Our approach, in contrast, is fully automated. Furthermore, their regexes do not allow for alternatives or stars, while we want to support all of these.

Only one approach [Babbar and Singh, 2010] can relax a given regex automatically. That work also addresses the problem of relaxing a regular expression in order to improve its F-measure. The regular expression is split into its parts, which are gradually relaxed until a performance threshold is reached. In each iteration, the relaxed regex is matched over a document, and the positive matches are clustered. The final regex is a disjunction of the regexes obtained for each cluster.

However, as we will see in our experiments, that approach produces very long regexes (usually over 100 characters). Our approach, in contrast, produces much shorter expressions – at comparable or even better precision.

Even though [Babbar and Singh, 2010] addresses the same problem as we do, our approaches are inherently different. First of all we do not need the labeling of all positive examples. Moreover, we do not greedily relax the character classes and ranges of the initial regex. Instead, we consider a small set of missing words, based on which we add alternatives to relevant sub-expressions of the initial regex. We show in our experiments

that our approach achieves comparable results to [Babbar and Singh, 2010] with a much smaller set of positive examples.

5.3 Preliminaries

As regular expressions are the backbone of this work, we start by defining the exact family of regular expressions that we consider, along with other relevant notions. Then, we proceed to stating our problem formally.

5.3.1 Regular Expressions

In all of the following, we assume a fixed ordered set of characters Σ, the *alphabet*. A *string* is a sequence of characters of Σ. We write $|s|$ for the length of a string s, and s_i for its i-th character.

Definition (Regular expression). *A* regular expression *is either*

— *the empty string* ϵ*, or*
— *an element of* Σ*, or a string of the form*
— AB *(a* concatenation*),*
— $(A|B)$ *(a* disjunction*), or*
— $(A)^*$ *(a* Kleene star*), where* A *and* B *are regexes.*

As is common, we allow the following shorthand notations for a regex A and integers n, m:

— $A\{0,0\} := \epsilon$,
— $A\{0,n\} := (\epsilon|A\{1,n-1\})$,
— $A\{n,m\} := AA\{n-1,m\}$,
— $A\{n\} := A\{n,n\}$,
— $A? := (\epsilon|A)$, and
— $A^+ := AA^*$.

We abbreviate a disjunction $(a|...|z)$ of single characters $a,\dots,z \in \Sigma$ as $[a...z]$. We also allow the character classes $[a-z] := [a...z]$ for $a, z \in \Sigma, a < z$, and regular expression syntax supported by popular languages, such as `\d=[0-9]`, `\w=[A-Za-z0-9_]`, or the single dot, which stands for a disjunction of all characters of Σ. In our context, we define the length of a regular expression as the total number of characters in it, if seen as a string. For example, the length of `(\d{2} ?){4}\d{2}` is 17.

Definition (Language). *The* language $L(r)$ *of a regex* r *is the set of strings defined as follows:*

— $L(\epsilon) := \varnothing$
— $L(a) := a$*, for* $a \in \Sigma$

— $L(AB) := \{w_1 w_2 : w_1 \in L(A), w_2 \in L(B)\}$
— $L(A|B) := L(A) \cup L(B)$
— $L(A*) := \{\epsilon\} \cup L(AA*)$

Definition (Match). *A string s* matches *a regex r (or, equivalently, r matches s), if* $s \in L(r)$.

For example, the language of the regex `ab*c` is the set of strings that start with an "a", have 0 or more occurrences of a "b", and finish with a "c". Thus, any of the strings *abc, ac, abbc* match `ab*c`.

Each regex r is associated to a syntax tree, whose leaf nodes are characters or character classes:

Definition (Syntax tree). *The* syntax tree *T of a regex is an ordered labeled tree defined as follows:*

— *$T(A \dots B)$ consists of a node labeled with "&", with children $T(A), \dots, T(B)$.*
— *$T(A| \dots |B)$ consists of a node labeled with " | ", with children $T(A), \dots, T(B)$.*
— *$T(A*)$ consists of a node labeled "*" with one child, $T(A)$.*
— *For any $a \in \Sigma$, $T(a)$ is a single node (a* leaf node*) labeled with a.*

Figure 5.1 shows the syntax tree for the regex `ab*c`.

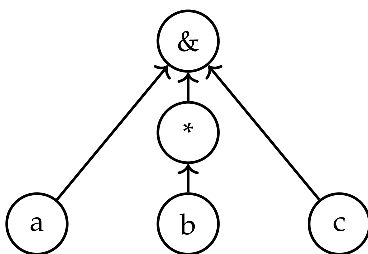


Figure 5.1 – The syntax tree $T(r)$ for the regex r = `ab*c`

We use *n.label*, *n.children*, *n.parent*, and *n.index* to refer to the label, children, parent, and index among its siblings, respectively, of a node n. We write $|r|$ to denote the number of leaf nodes of $T(r)$. We write r_i to refer to the i-th leaf node in the pre-order traversal of $T(A)$. For two leaf nodes r_i, r_j, we write $r_i < r_j$ to mean $i < j$.

Now we define an automaton that determines whether a word is part of the language of a regex. We will use a simplified version of the the *Glushkov automaton* [Glushkov, 1961].

The *exit nodes* E of a node n in a syntax tree are defined as follows: If n is a leaf node, then $E(n) = \{n\}$. If n is a concatenation, then $E(n)$ are the exit nodes of n's last child. If n is a disjunction, then $E(n) = \bigcup_{n' \in n.children} E(n')$. If n is a Kleene star, then $E(n) = E(n.children_1) \cup P(n)$, where $P(n)$ is the set of n's predecessor nodes.

The *predecessor nodes* P of a node n in a syntax tree are defined as follows: The σ ("start") represents the beginning of the regex. If n has no parents, then $P(n) = \{\sigma\}$. If n is the first child of a concatenation, then $P(n) = P(n.parent)$. For any other child n of a concatenation, $P(n) = E(n.parent.children_{n.index-1})$, i.e., the exit nodes of the preceding child. If n is the child of a disjunction, then $P(n) = P(n.parent)$. If n is the child of a Kleene star, then $P(n) = E(n) \cup P(n.parent)$.

We can allow our automaton to ignore the empty word, as we will only use it to match non-empty words. With this, the automaton of a regex r is the graph that has the leaf nodes of $T(r)$ as nodes, and an edge (u, v) if $u \in P(v) \setminus \{\sigma\}$. The *entry nodes* (or start states) are all leaf nodes n with $\sigma \in P(n)$, i.e., those nodes whose predecessor node would be the beginning of the regex. The *exit nodes* (or end states) are the elements of $E(r) \setminus \{\sigma\}$. Figure 5.2 shows the automaton for the regex `ab*c`. A string $s \neq \epsilon$ matches a regex r if there exists a path from an entry node to an exit node in the automation of r, such that the sequence of nodes corresponds to the sequence of characters in s. We say that a leaf node is *to the left* of another leaf node, if the latter can be reached from the former in the Glushkov automaton of r.

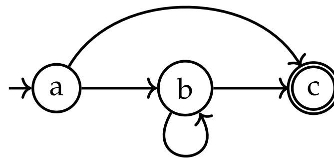


Figure 5.2 – Automaton $A(r)$ for the regex r = `ab*c`

5.3.2 Problem Statement

The problem that we address is the following:

Problem Statement. *Given a regular expression r, a set S of positive examples, $\varepsilon \notin S$, and a set E^- of negative examples, such that $|S| \ll |E^-|$, find a "good" regular expression r' such that $L(r) \subseteq L(r')$, $S \subseteq L(r')$, and $|L(r') \cap E^-|$ small.*

In other words, we want to generalize the regex so that it matches all strings it matched before, plus the new positive ones. For example, given a regex r =`[0-9]+` and a string s ="12 34 56", a possible regex to find is r' =`([0-9] ?)+`. This regex matches all strings that r matched, and it also matches s.

Now there are obviously trivial solutions to this problem. One of them is to propose r =`.*`. This solution matches s. However, it does most likely not capture the intention of the original regex, because it matches arbitrary strings. Therefore, one input to the problem is a set of negative examples E^-. The regex shall be generalized, but only so

much that it does not match many words from E^-. The rationale for having a small set S and a large set E^- is that it is not easy to provide a large set of positive examples: these are the words that the hand-crafted regex does not – but should – match, and they are usually few. In contrast, it is somewhat easier to provide a set of negative examples. It suffices to provide a document that does not contain the target words. (such as an encyclopedia article for the task of matching phone numbers). All strings in that document can make up E^-, as we show in our experiments.

Another trivial solution to our problem is to say $r' = r|s$. However, this solution does not capture the intention of the regex either. In the example, the regex r' = `[0-9]+|12 34 56` does match s, but it does not match any other sequence of numbers and spaces. Hence, the goal is to *generalize* the input regex appropriately, i.e., to find a "good" regex that neither over-specializes nor over-generalizes. To quantify the goodness of a regex, we follow the same approach as [Babbar and Singh, 2010; Murthy et al., 2012], and evaluate the precision, recall, and F1 measure on a test set.

5.3.3 Finding the Matchings

Matchings. Our goal is to modify the regex in such a way that it still matches all strings that it matched before. Our strategy is to limit the modifications of the regex to a minimum. For this purpose, we first match the string to the regex in an approximate manner. Figure 5.3 shows an example. This allows us to find out which parts (substrings) of the string already match. The parts that match can then be kept intact. The set of unmatched parts of the string are passed to the subsequent repairing step of the algorithm. We begin by defining a *matching*, using the notation $f|_A$ to denote the restriction of function f to domain A:

Definition (Matching). *Given a string s and a regex r, a* matching *is a partial function m from string indices $\{1,...,|s|\}$, to leaf nodes of r's syntax tree $T(r)$. Let $I = \{i_1,...,i_n\} \subseteq \{1,...,|s|\}$ be the domain of m, where $i_1 < \cdots < i_n$. Function m is a matching, if and only if one of the following conditions applies:*

- *r is a character or character class and $I = \varnothing$, or $I = \{i\}$, $m(i) = r$, and $s_i \in L(r)$*
- *$r = pq$ and there exists a j such that $m|_{i_1,...,i_j}$ is a matching for p and $m|_{i_{j+1},...,i_n}$ is a matching for q*
- *$r = p|q$ and m is a matching for p or for q*
- *$r = p*$ and $\exists k_1,\ldots,k_j : k_1 = i_1 \wedge k_1 < \cdots < k_j \wedge k_j = i_n + 1 \wedge \forall l \in \{1,\ldots,j-1\} : m|_{k_l,...,(k_{l+1})-1}$ is a matching for p*

A matching is *maximal*, if there is no other matching that is defined on more positions of the string. Figure 5.3 shows a maximal matching for the regex `ab*c` and the (non-matching) string `aabbdc`.

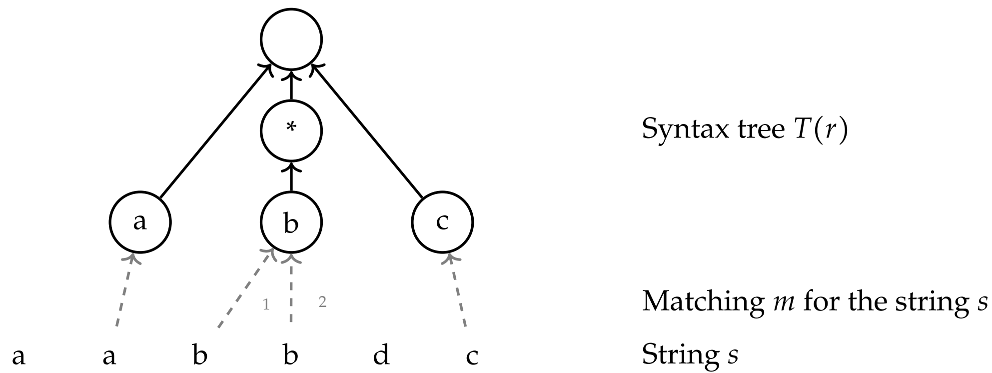


Figure 5.3 – A matching m for regex `ab*c` and string `aabbdc`

A matching indicates the parts of the string that are already covered by the regex. It can map different characters onto the same leaf node (see edges labeled 1 and 2 in Figure 5.3). Our goal is now to find a *maximal matching*, i.e., a matching that is defined on as many positions of the string as possible. There can be several maximal matchings for a given string and a given regex. For example, for r =`/a+[a-z]/` and s =`/aa/`, there are two maximal matchings: one maps both `a`'s onto `a+`, and leaves `[a-z]` unmapped. The other maps the first `a` to `a+`, and the second `a` to `[a-z]`. Obviously, we prefer the second matching, because it covers the regex entirely, and because it does not introduce a "gap" in the regex. A *gap in the regex* for a given matching is either a pair of two consecutive characters in the string that are mapped to two regex leaves a and b that are not predecessors of each other, i.e., $a \notin P(b)$, or the first mapped or the last mapped character of the string that is not mapped to an entry or exit node, respectively. Vice versa, a *gap in the string* is either a pair of two matched characters that have unmatched characters in-between, or a sequence of unmatched characters at the beginning or at the end of the string. For example, for r = `ac` and s = `abc`, a gap in s occurs between `a` and `c`.

Finding Maximal Matchings. To find a maximal matching with few gaps, we use the dynamic programming algorithm developed by Myers et al. in [Myers and Miller, 1989]. Myers' algorithm first constructs an automaton for the regex. Then it uses a matrix, in which the rows are the characters of the input string, and the columns are the leaves of the regex. The algorithm then proceeds similarly to the dynamic programming algorithms for computing the edit distance between two strings in order to find a matching that maximizes the number of matched characters. As described in the preliminaries, we unfold regexes of the form $A\{n,m\}$.

The goal of the dynamic programming algorithm is to find a sequence of operations that transform the initial word into a word that is matched by the regex. Every operation can

	I	S	B	N	(9	7	8 |	9	7	9 |)	\d	_
I	**1**	1	1	1	1	1	1	1	1	1	1	_
9	.	.	.	.	**2**	2	2	**2**	2	**2**	**2**	_
7	.	.	.	.	.	**3**	3	.	**3**	3	**3**	_
8	.	.	.	.	.	.	**4**	.	.	.	**4**	_
0	.	.	.	.	.	.	.	.	.	.	**5**	_
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Figure 5.4 – Excerpt of the table used by the dynamic programming algorithm to find the maximal matching of the string `I9780321455369` and the regex `ISBN(978|979|)\d{10}`

modify at most one character. Every operation is assigned a weight, and we would like to find the sequence of operations with the minimal cost.

The original algorithm computes a string-regex edit distance based on 3 operations: insertion, deletion, and substitution. Substitution can mean that a character s_i matches a regex leaf r, or that the character needs to be replaced by a different one. For our purposes, we split this case: if $s_i \in L(r)$, we call the operation a *match*, otherwise we call it a *substitution*. Finally, we introduced an operation called *gap* that is evoked whenever there is a gap in the regex or in the string.

Figure 5.4 illustrates this process in a simplified way, by tracking only match operations. On the left of the table is the string, and on the top of the table is the regex automaton. Each cell counts how many match operations occurred for the prefix of the string until that row, and for the automaton "until" the state represented by the column.

Myers' algorithm can be configured with different weights for each operation. The original paper proposes a weight of -1 for deletions. For our purposes, all types of modifications of the original regex should be penalized in the same way, and hence we use that weight as well for insertions, and for gaps. The weight for matches has to be positive, so that the algorithm is encouraged to match characters. Since one match shall count as much as one deletion plus its corresponding gap, the weight has to be 2. Finally, the weight of substitution is $-\infty$, because we want to avoid substitutions at all cost, giving preference to deletions followed by insertions. We used these values for all our experiments.

In order to obtain the match operations of the optimal alignment, Myers et al. propose to track the provenance of the information in each cell of the matrices in the form of back pointers. Figure 5.4 illustrates this process for the string `I9780321455369` and the regex `ISBN(978|979|)\d{10}`. Additionally, we store character and regex leaf in the cells, if they match. We obtain the maximal matching as follows: We start from the bottom right

cell of the matrix, follow the back pointers, and add (character, regex leaf) pairs to the matching if appropriate.

Let us now apply this algorithm to our running example of phone numbers.

> *Example (Matching):* Consider the regex `(\d{3}-){2}\d{4}` (Table 5.1). Assume that we want to match the word `(234) 235-5678`. We first expand the regex to `\d\d\d-\d\d\d-\d\d\d\d`. Then, the matcher aligns the substring 234 to the first `\d\d\d` group, and the substring 235-5678 to `\d\d\d-\d\d\d\d`. Using index i to refer to the i-th regex leaf, we obtain the following matching m: m(2)=$\texttt{\textbackslash d}_1$, m(3)=$\texttt{\textbackslash d}_2$, m(4)=$\texttt{\textbackslash d}_3$, m(7)=$\texttt{\textbackslash d}_5$, _, m(14)=$\texttt{\textbackslash d}_{12}$.

Original regex r:	`(\d{3}-){2}\d{4}`
String $s \in S$:	`(234) 235-5678`
Repaired regex:	`\(?\d{3}(\) |-)\d{3}-\d{4}`

Table 5.1 – Example regex reparation by the simple algorithm

5.4 Simple Algorithm

In this section, we describe our first algorithm for solving the problem described in Section 5.3.2 on page 68: Given a regex r, a set of strings S, our goal is to extend r so that it matches the strings in S. Table 5.1 shows an example with only a single string $s \in S$, and no negative examples. We assume that the syntactic sugar of the regex is unfolded, so that it contains only Kleene stars, disjunctions, and concatenations.

Our algorithm is shown in Algorithm 6 on the facing page. The algorithm inserts the non-matching parts of the string in the appropriate places of the regex, and it makes the non-matched parts of the regex optional. First, we mark the start and the end of the regex (Lines 1-2) to simplify the treatment of the boundaries. We make a copy of the regex (Line 3) for the case that we need to revert the repairing. Then, we iterate over the set of positive examples S (Line 4), and find a maximal matching (Line 5). This is done with Myers' algorithm as described in the previous section. We repair the regex r for s with Algorithm 7 (Line 6). We check whether the repaired regex performs well on the set of negative examples (Line 8). If the original regex has fewer false positives, we output a disjunction of the original regex and the missing words instead (Line 9). Finally, we remove the characters that we added in Lines 1-2.

Now we will describe the actual repairing step, which is shown in Algorithm 7 on page 74. It takes as input a regex r, a string s, and a matching m. It runs left to right through the string (Line 1). For each mapped character position, it finds the next

Algorithm 6 Repair regex (simple)

INPUT: regex r, set of strings S, negative examples E^-
OUTPUT: modified regex r
1: $r \leftarrow xry$, where characters x, y do not occur in r or s
2: $s \leftarrow xsy$
3: $r_{orig} \leftarrow$ copy of r
4: **for** $s \in S$ **do**
5: $m \leftarrow findMatching(r, s)$
6: $r \leftarrow fixRegexSimple(r, s, m)$
7: **end for**
8: **if** number of matches of r in E^- > number of matches of r_{orig} in E^- **then**
9: $r \leftarrow r_{orig}|s_1|\dots|s_n$, where $\{s_1, \dots, s_n\} = S$
10: **end if**
11: **return** r', where $r = xr'y$

mapped character position (Line 2), and the lowest common ancestor (LCA) of both leaf nodes in the syntax tree of r (Line 3). In a Kleene star, it may happen that the next character is mapped to a leaf node that lies to the left of the leaf node of the current character. Consider, e.g., the regex `(abc)*` and the string `abfabc`. Here, the second `a` is mapped to a leaf node that lies to the left of the node of `b`: the links cross. In such a case, we find the next character that is mapped to the right of the current one (Line 5), and we add the obstructing substring as an optional concatenation (Line 7). In the example, we obtain `(ab(fab)?c)*`. If no such fix can be found, we remove the mapping (Line 10), and proceed. If the next character is mapped to a regex leaf node that succeeds the current leaf node, we trace the path from the left leaf node to the LCA, and make all children to the right of that path optional (Lines 14-16). We proceed analogously with the right leaf node (Lines 17-19). Finally, we insert the unmatched substring below the LCA (Lines 20-21).

The algorithm manages to repair the regex reasonably well if the positive examples consist of only one string. However, if one repair added a disjunction at the wrong place, then performing more repairs might add more disjunctions within the previously added disjunction. As a consequence, the regex becomes more and more convoluted. Therefore, we have developed a more sophisticated algorithm, which we present in the next section.

5.5 Adaptive Algorithm

In this section we present a more elaborate version of the previous algorithm, which incorporates the following ideas: we do not expand quantifiers $A\{\dots\}$, but treat them

Algorithm 7 Fix regex (simple)

INPUT: regex r, string s, matching m
OUTPUT: modified regex r
1: **for** $i \in dom(m)$ in increasing order **do**
2: j $\leftarrow argmin_j$ ($j > i \wedge m(j)$ is defined)
3: $n \leftarrow$ lowest common ancestor of $m(i)$ and $m(j)$ in the syntax tree of r
4: **if** $m(j)$ left of $m(i)$ in r **then**
5: $k \leftarrow \text{argmin}_k \quad k > j \wedge m(k)$ right of $m(i)$
6: **if** k exists **then**
7: insert $(w_{i+1} \dots w_{k-1})$? below n between $m(i)$ and $m(k)$
8: $i \leftarrow k$
9: **else**
10: $m(j) \leftarrow$ undefined
11: **end if**
12: **go to** Line 2
13: **end if**
14: **for** node n' on the path from $m(i)$ to n in the syntax tree of r **do**
15: **if** n' is concatenation **then** make children of n' to the right optional
16: **end for**
17: **for** node n' on the path from $m(j)$ to n in the syntax tree of r **do**
18: **if** n' is concatenation **then** make children of n' to the left optional
19: **end for**
20: $c_1, \dots, c_k \leftarrow$ children of n between $m(i)$ and $m(j)$
21: replace $c_1, \dots, c_k$ by a disjunction $c_1 \dots c_k | w_{i+1}, \dots, w_{j-1}$
22: **end for**

as their own type of node in the syntax tree; we group the modifications, and carry them out at the same time; we check possible places for adding a missing part as a disjunction, and accept such a disjunction if it does not deteriorate precision; and we collect the strings where introducing disjunctions would deteriorate precision, infer generalized regexes from these strings, and add the generalized regexes instead.

Before we present the algorithm, we will introduce the concept of a *cycle*. Assume that r contains a Kleene star q^* that is not contained within another Kleene star. The *cycle* of a Kleene star q^* for a character s_i, $m(i) \in T(q)$, in a matching m is the minimal $n \geq 1$ so that $m|_{1,\dots,i}$ restricted on $q*$ could be transformed to a matching for $q\{n\}$, but not to a matching for $q\{n-1\}$ (while still matching the same characters of the string). This definition can be extended naturally to quantifiers $A\{\dots\}$. For nested Kleene stars (and quantifiers), the outer Kleene stars (and quantifiers) are unfolded sufficiently often, and then the cycle is determined. Take as example the string `bab` and the regex `(ab*)*`. The

cycle of both Kleene stars for the character b_1 is 1, while the cycle of the outer Kleene star for b_2 is 2, and the cycle of the inner Kleene star `b*` for b_2 is 1.

The general outline of our method is shown in Algorithm 8. It takes as an additional input parameter a threshold α. This threshold indicates how many negative examples the repaired regex is allowed to match. Higher values for α allow a more aggressive repair, which matches more negative examples. $\alpha = 1$ makes the algorithm more conservative. In that case, the method tries to match at most as many negative examples as the original regex did. The algorithm proceeds in 4 steps, which we will now discuss in detail.

Algorithm 8 Repair regex (adaptive)

INPUT: regex r, set of strings S, negative examples E^-, threshold $\alpha \geq 1$
OUTPUT: modified regex r
1: $M \leftarrow \bigcup_{s \in S} \mathit{findMatching}(r, s)$
2: $gaps \leftarrow \bigcup_{m \in M} findGaps(m)$
3: $\mathit{findGapOverlaps}(r, gaps)$
4: $\mathit{addMissingParts}(r, S, gaps, E^-, \alpha)$

Step 1: Finding the Matchings. For each word $s \in S$, our algorithm finds the maximal matchings (Algorithm 8, Line 1, Method *findMatching*), as described in Section 5.3.3. The maximal matchings for all the words in S are collected in a set M.

Step 2: Finding the Gaps. The matching tells us which parts of the regex match the string. To fix the regex, we are interested in the parts that do *not* match the string. For this purpose, we introduce a data structure for the gaps in the string (i.e., for the parts of the string that are not mapped to the regex). In the context of the adaptive algorithm, a gap g for string s in a matching m is a tuple of the following:

— $g.start$: The index in the string where the gap starts (possibly 0).
— $g.end$: The index in the string where the gap ends (possibly $|s|+1$).
— $g.span$: The substring between $g.start$ and $g.end$ (excluding both).
— $g.m$: The matching m, which we store in the gap tuple for later access.
— $g.parts$: An (initially empty) set that stores sequences of concatenation child nodes. The sequences are disjoint, and partition the regex. Each $p \in g.parts$ is a possibility to inject $g.span$ into the regex as `(`p`|`$g.span$`)`.

> *Example (Finding the gaps):* When matching the word `(234) 235-5678` onto the regex `\d\d\d-\d\d\d-\d\d\d\d`, we encounter two gaps: gap_1 embraces the sub-

> string "`(`" with $gap_1.start = 0$, $gap_1.end = 2$. gap_2 embraces the substring "`)  `" with $gap_2.start = 4$, $gap_2.end = 7$.

For each matching m, we find all gaps g that have no matching character in between (i.e., $\nexists k : g.start < k < g.end \wedge m(k)$ is defined), and where at least one of the following holds

— there is a character in the string between $g.start$ and $g.end$, i.e., $g.start < g.end - 1$

— there is a gap in the regex between $m(g.start)$ and $m(g.end)$, i.e., $m(g.start) \notin previous(m(g.end))$.

This set of gaps is returned by the method *findGaps* in Algorithm 8, Line 2.

Step 3: Finding Gap Overlaps. Gaps can overlap. Take for example the regex r = `01234567`, and the missing words `0x567` and `012y7`. One possible repair is `0(12)?(34|x|y)?(56)?7`. We can find this repair only if we consider the overlap between the gaps. In this example, we have two gaps: one with span `1234` and one with span `3456`. We have to partition the concatenation for the first gap into `12` and `34`, and for the second gap into `34` and `56`.

This is what Algorithm 9 on the facing page does. It takes as input the regex r and the set of gaps $gaps$. It walks through the regex recursively, and treats each node of the regex. We split quantifiers $r\{min, max\}$ with $max < 100$ into $r\{\ldots\}r\{\ldots\}$, if the gap occurs between different cycles of $r\{min, max\}$. For other quantifiers, Kleene stars, and disjunctions, we descend recursively into the regex tree (Line 6).

For concatenation nodes, we determine all gaps that have their start point or their end point inside the concatenation (Line 9). Then, we determine the partitioning boundaries (Line 10; $l \in r$ means that regex r has a leaf node l). We consider each gap g (Line 11). We find whether the start point or the end point of any other gap falls inside g. This concerns only the boundaries between child indices s and e of the concatenation $c_1 \ldots c_n$ (Lines 12-13). We partition the concatenation subsequence $c_s \ldots c_{e-1}$ by cutting at the boundaries (Line 14). Finally, the method is called recursively on the children of the concatenation that contain the start point or the end point of any gap (Lines 16-18).

Step 4: Adding Missing Parts. The previous step has given us, for each gap, a set of possible partitionings. In our example of the regex `01234567`, the word `012y7`, and the gap `3456`, we have obtained the partitioning `34|56`. This means that both `34` and `56` have to become optional in the regex, and that we can insert the substring `y` as an alternative to either of them: `012(34|y)(56)?7` or `012(34)?(56|y)7`. Algorithm 10 takes this decision based on which solution performs better on the set E^- of negative examples. It may also happen that none of these solutions is permissible, because they all match too many negative examples. In that case, it is preferable to add the word as

Algorithm 9 Find Gap Overlaps

INPUT: regex r, gaps $gaps$
OUTPUT: modified regex r, modified gaps $gaps$

1: **if** r is quantifier $q\{min, max\}$ with $max < 100$,
 and gap occurs between different cycles of $r\{min, max\}$ **then**
2: $r \leftarrow q\{\ldots\}q\{\ldots\}$ with appropriate ranges
3: *findGapOverlaps*$(r, gaps)$
4: **else if** r is disjunction or Kleene star or quantifier **then**
5: **for** child c of r **do**
6: *findGapOverlaps*$(c, gaps)$
7: **end for**
8: **else if** r is concatenation $c_1 \ldots c_n$ **then**
9: $gaps' \leftarrow \{g \mid g \in gaps \wedge g.m(g.start) \in r \vee g.m(g.end) \in r\}$
10: $idx \leftarrow \{i+1 \mid g \in gaps' \wedge g.m(g.start) \in c_i\} \cup \{i \mid g \in gaps' \wedge g.m(g.end) \in c_i\}$
11: **for** $g \in gaps'$ **do**
12: $s \leftarrow$ i+1 if $g.m(g.start) \in c_i$, else 1
13: $e \leftarrow$ i if $g.m(g.end) \in c_i$, else $n+1$
14: $g.parts \leftarrow g.parts \cup \{c_i \ldots c_{j-1} \mid i, j \in idx \wedge s \le i < j \le e \wedge (\nexists k : k \in idx \wedge i < k < j)\}$
15: **end for**
16: **for** $c_i \in \{c_i \mid \exists g \in gaps'.\ g.m(g.start) \in c_i \vee g.m(g.end) \in c_i\}$ **do**
17: *findGapOverlaps*$(c_i, gaps)$
18: **end for**
19: **end if**

a disjunction to the original regex, as in `01234567|012y7`. To make these decisions, the algorithm compares the number of negative examples matched by the repaired regex with the number of negative examples matched by the original regex. The ratio of these two should be bounded by the threshold α.

Algorithm 10 takes as input a regex r, a set of gaps $gaps$ with partitionings, negative examples E^-, and a threshold α. The algorithm first makes a copy of the original regex (Line 1) and treats each gap (Line 2). For each gap, it considers all parts (Line 3). In the example, we consider the part `34` and the part `56` of the gap `3456`. The algorithm transforms the part into a disjunction of the part and the span of the gap. In the example, the part `34` is transformed into `(34|y)` (Line 4). If the number of matched negative examples does not exceed the number of negative examples matched by the original regex times α (Line 5), the algorithm chooses this repair, and stops exploring the other parts of the gap (Lines 6-7). In Line 11, the algorithm collects all positive examples that are still not matched. The changes that were made for these words are undone (Line 12). Line 13 generalizes these words into one or several regexes. The algorithm then checks

Algorithm 10 Add Missing Parts

INPUT: regex r, set of strings S, gaps $gaps$, negative examples E^-, threshold $\alpha \geq 1$
OUTPUT: modified regex r

1: $org \leftarrow r$
2: **for** g ∈ gaps **do**
3:     **for** part p = $(c_i \cdots c_j) \in g.parts$ **do**
4:         $r' \leftarrow r$ with $c_i \cdots c_j$ replaced by $(c_i \cdots c_j | g.span)$, and all other parts $c_x \ldots c_y$ in $g.parts$ made optional with $(c_x \ldots c_y |)$
5:         **if** $|E^- \cap L(r')| \leq \alpha \cdot |E^- \cap L(org)|$ **then**
6:             $r \leftarrow r'$
7:             **break**
8:         **end if**
9:     **end for**
10: **end for**
11: $S' \leftarrow S \setminus L(r)$
12: undo all changes for $s \in S'$ not required by other repairs
13: $G \leftarrow$ generalize words in S'
14: **for** g ∈ G **do**
15:     **if** $|E^- \cap L(r|g)| \leq \alpha \cdot |E^- \cap L(org)|$ **then**
16:         $r \leftarrow r|g$
17:     **else**
18:         **for** $s \in L(g) \cap S'$ **do**
19:             $r \leftarrow r|s$
20:         **end for**
21:     **end if**
22: **end for**

if the regex obtained this way is good enough (Line 15). If this is the case, the regex is added as a disjunction (Line 16). Otherwise the words that contributed to that group are added disjunctively (Lines 18-19).

The generalization is adapted from [Babbar and Singh, 2010]. First, we assign a group key to every word. The key is obtained by substituting substrings consisting only of digits with a (single) `\d`, lower or upper case characters with a `[a-z]` or `[A-Z]`, and remaining characters with character class `\W` or `\w`. Finally we obtain the group regex r by adding $\{min, max\}$ after every character class, such that r matches all strings in that group.

Time complexity. We analyze the time needed to run our algorithm. Let M be the sum of the length of the missing words. Let N the length of the (unexpanded) regex. Let N' the length of the expanded regex. Furthermore let t be the runtime of applying a regex of

length $O(N')$ on the negative examples E^-. Step 1 is a dynamic programming algorithm which fills up a table with M rows and $O(N')$ columns. It runs in $O(N'M)$ [Myers and Miller, 1989]. Step 2 iterates over the characters of the words. It checks two conditions, both in constant time reusing $previous(\dots)$ of step 1. Thus complexity of step 2 is $O(M)$. A word of length l can lead to at most $l+1$ gaps. Therefore step 2 finds at most $O(M)$ gaps. Step 3 recursively descends into every node of the regex tree. For each of the concatenation nodes it might to iterate over $O(M)$ gaps. In doing so it produces at most $O(N)$ part boundaries per gap, therefore at most $O(N)$ parts per gap. The splitting of quantifier might lead to an exponential increase in the length of the regex. However, this happens also with the expansion of the regex in step 1, so the number of nodes that Algorithm 9 is limited by $O(N')$, so in total, step 3 runs in $O(N'M)$. Step 4 iterates over $O(M)$ gaps, each with at most $O(N)$ parts. Every part might require one application of the regex on the examples E^-. Therefore the first part of Algorithm 10 (Lines 1-10) runs in $O(NMt)$. Undoing (Lines 11-12) can be done in less than that time. The group keys for generalization can be obtained in $O(M)$, producing at most $O(M)$ generalized regexes. Applying those on the examples takes at most $O(Mt)$. In total, step 4 runs in time $O(NMt)$. In summary, our approach is dominated by adding the missing parts, and applying the candidates of repaired regexes on the examples. Its total runtime is thus $O(NMt + N'M)$.

5.6 Experiments

5.6.1 Setup

Measures. To evaluate our algorithms, we follow related work in the area [Babbar and Singh, 2010; Murthy et al., 2012], and use a gold standard of positive example strings, $E^+ \supset S$. With this, the *precision* of a regex r is the fraction of positive examples matched among all examples matched:

$$prec(r) = \frac{|E^+ \cap L(r)|}{|L(r) \cap (E^+ \cup E^-)|}$$

The *recall* of r is the fraction of positive examples matched:

$$rec(r) = \frac{|E^+ \cap L(r)|}{|E^+|}$$

As usual, the *F1 measure* is the harmonic mean of these two measures:

$$F_1(r) = \frac{2 \cdot prec(r) \cdot rec(r)}{prec(r) + rec(r)}$$

Baselines. We compare our methods to different baselines. We call the first baseline "dis", short for a disjunction. Given a set of missing words $S = \{s_1, \ldots, s_n\}$, it just adds each word as a disjunct, i.e., the repaired regex is $r|s_1| \ldots |s_n$. The second baseline is the "star" baseline, short for Kleene star. It just takes the regex `.*` as the repaired regex.

Competitors. We compare our approaches to the other method [Babbar and Singh, 2010] (B&S) that can generalize given regexes automatically (see Section 5.2). The code for B&S was not available upon request. We therefore had to re-implemented the approach. We think that our implementation is fair, because it achieves a higher F1-value (87% and 84%, as opposed to 84% and 82%) when run on the same datasets as in B&S (see below) with a full E^+ as input.

Datasets. We use 3 datasets from related work. The *ReLIE dataset* [1] [Li et al., 2008] includes four tasks (phone numbers, course numbers, software names, and URLs). Each task comes with a set of documents. Each document consists of a span of words, and 100 characters of context to the left, and to the right. Each span is annotated as a positive or a negative example, thus making up our sets E^+ and E^-, respectively. We manually cleaned the dataset by fixing obvious annotation errors, e.g., where a word is marked as a positive and a negative example in the same task. In total, the dataset contains 90807 documents.

task	documents	avg. size	$\lvert E^+ \rvert$	regexes
from [Li et al., 2008]				
ReLIE/phone	41 896	211	2 657	5
ReLIE/course	569	210	314	5
ReLIE/software	44 413	185	2 307	5
ReLIE/urls	3 929	176	735	5
following [Babbar and Singh, 2010]				
Enron/phone	225	1 452	145	1
Enron/date	225	1 452	392	1
created with YAGO				
YAGO/dates	100 000	25	109 824	15
YAGO/numbers	100 000	57	131 149	15

Table 5.2 – Statistics of the datasets

1. http://dbgroup.eecs.umich.edu/regexLearning/

The *Enron-Random dataset*[2] [Minkov et al., 2005] contains a set of emails of Enron staff. The work of [Babbar and Singh, 2010] uses it to extract phone numbers and dates. Unfortunately, there is no gold standard available for these tasks, and the authors of [Babbar and Singh, 2010] could not provide one. Therefore, we manually annotated phone numbers and dates on 200 randomly selected files, which gives us E^+. As in [Babbar and Singh, 2010], any string that is matched on these 200 documents, and that is not a positive example is considered a negative example. In this way, we obtain a large number of negative examples.

The *YAGO-Dataset* consists of Wikipedia infobox attributes, where the dates and numbers that were used to build YAGO have been annotated as positive examples. As in *Enron-Random*, all strings that are not annotated as positive examples count as negative examples.

We thus have 8 tasks: 4 for the ReLIE dataset, 2 for the Enron dataset, and 2 for the YAGO dataset. Each task comes with positive examples E^+ and negative examples E^-. Our algorithms need as input an initial regex that shall be repaired. For the Enron task, we used the initial regexes given in [Babbar and Singh, 2010]. For ReLIE and YAGO, we asked our colleagues to produce regexes by hand. For this purpose, we provided them with 10 randomly chosen examples from E^+ for each task, and asked them to write a regex. This gives us 5 initial regexes for each ReLIE task, and 15 initial regexes for each YAGO task. Table 5.2 summarizes our datasets.

Runs. The algorithms do not take as input the entire set of positive examples E^+, but a small subset S of positive examples. To simulate a real setting for the algorithms, we randomly select S from E^+, with $|S| = 10$ (and less than 10, if $|E^+| < 10$) . We average our results over 10 different random draws of S. For each draw, we use each initial regex that we have at our disposal, and average our results over these. We average over a 5×2 train/test splits, i.e., we split the dataset in two halves, train with one half, and test with the other, and vice versa. In summary, we provide the algorithms with 10 sets of words, randomly drawn from all missing words, each with at most 10 words, and run it on 10 train/test splits. Thus, we run each algorithm 500 times for each ReLIE task, 100 times for each Enron-Random task, and 1500 times for each YAGO task, and we average the obtained numbers over these. We provide the B&S competitor and the simple algorithm with additional positive examples obtained from running the original regex on the input dataset. The adaptive algorithm does not need this step. All algorithms are implemented in Java 8. The experiments were run on an Intel Xeon with 2.70GHz and 250GB memory.

5.6.2 Experimental Evaluation and Results

2. http://www.cs.cmu.edu/~einat/datasets.html

		baseline		competitor		adaptive approach		
task	original	dis	star	B&S	simple	α = 1.0	α = 1.1	α = 1.20
ReLIE/phone	81.6	82.1	12.3	-0.5 ▼	+0.3	+0.8	+1.6 △	**+2.3** ▲
ReLIE/course	45.8	46.0	48.4	**+6.4** ▲	+0.2 ▲	+0.5 ▲	+1.3 ▲	+1.4 ▲
ReLIE/software	9.2	12.4	9.9	+0.1 ▲	0.0	+0.7 ▲	+3.9	**+4.6**
ReLIE/urls	55.2	56.0	31.5	-30.3 ▼	+0.3	+2.9	**+4.2** ▲	**+4.2** ▲
Enron/phone	61.7	61.7	0.1	-7.7	+5.3 △	**+21.0** ▲	**+21.0** ▲	**+21.0** ▲
Enron/date	72.3	72.4	0.0	-49.6 ▼	0.0	**+0.6**	**+0.6**	+0.4
YAGO/number	40.1	40.1	31.0	-11.5 ▼	+1.8 ▲	**+3.4**	+2.2	+2.4
YAGO/date	70.1	70.1	34.3	-26.3 ▼	+0.3 ▲	**+6.9** ▲	+6.7 ▲	+6.6 ▲

Table 5.3 – F1 measure for different values of the parameter α, improvement over the dis-baseline in percentage points. Bold numbers indicate the maximum F1 measure within each row. ▲ (and △) indicates significant improvement relative to the dis-baseline for a significance level of 0.01 (and 0.05).

		baseline		competitor		adaptive approach		
task	original	dis	star	B&S	simple	α = 1.0	α = 1.1	α = 1.20
ReLIE/phone	71.2	71.8	100.0	-0.8 ▼	+0.5	+1.5	+2.8 △	**+3.9** ▲
ReLIE/course	88.1	88.4	100.0	-39.2 ▼	+1.0 ▲	+2.2 ▲	+5.3 ▲	**+6.6** ▲
ReLIE/software	79.2	83.9	100.0	-58.0 ▼	0.0	+0.4 ▼	+5.2 ▲	**+6.0** ▲
ReLIE/urls	64.8	66.1	100.0	-49.7 ▼	+1.1	+6.0	+10.0 ▲	**+10.0** ▲
Enron/phone	45.9	45.9	8.3	-7.2	+6.8 △	**+29.1** ▲	**+29.1** ▲	**+29.1** ▲
Enron/date	58.4	58.6	0.5	-39.1 ▼	0.0	+4.6	+4.6	**+4.9**
YAGO/number	53.8	53.8	39.1	-34.9 ▼	+2.5 ▲	**+4.2**	+2.8	+3.1
YAGO/date	68.9	68.9	48.4	-31.2 ▼	+0.2	**+13.2** ▲	+13.1 ▲	+13.0 ▲

Table 5.4 – Recall for different values of the parameter α, improvement over the dis-baseline in percentage points. Bold numbers indicate the maximum recall within the competitor and our system. ▲ (and △) indicates significant improvement relative to the dis-baseline for a significance level of 0.01 (and 0.05).

		baseline		competitor		adaptive approach		
task	original	dis	star	B&S	simple	$\alpha = 1.0$	$\alpha = 1.1$	$\alpha = 1.20$
ReLIE/phone	97.8	97.8	6.5	0.0	0.0	0.0	0.0	0.0
ReLIE/course	31.0	31.1	32.0	**+25.8** ▲	+0.1 △	+0.3 ▲	+0.5 ▲	+0.5 ▲
ReLIE/software	5.7	8.7	5.2	**+55.0** ▲	0.0	+0.3 ▼	+3.1	+3.7
ReLIE/urls	65.7	65.9	18.7	**+26.5**	+0.2	+0.7	+1.5 ▲	+1.5 ▲
Enron/phone	95.5	**95.5**	0.1	-2.8	-0.2	-1.9 ▼	-1.9 ▼	-1.9 ▼
Enron/date	96.1	**95.9**	0.0	-62.2 ▼	**0.0**	-5.2	-5.2	-6.0
YAGO/number	35.2	35.2	25.7	**+36.2** ▲	+1.4 ▲	+1.8	+0.8	+0.8
YAGO/date	81.3	81.3	26.6	-21.2 ▼	**+0.1** ▲	-4.2 ▽	-4.5 ▽	-4.7 ▼

Table 5.5 – Precision for different values of the parameter α, improvement over the dis-baseline in percentage points. Bold numbers indicate the maximum precision within each row. ▲ (and △) indicates significant improvement relative to the dis-baseline for a significance level of 0.01 (and 0.05).

F1 measure. Table 5.3 shows the F1-measure on all datasets for different algorithms: the original regex, the disjunction-baseline, the star-baseline, the method from [Babbar and Singh, 2010], the simple algorithm, and the adaptive algorithm with different values for α. The table shows the improvement of the $F1$ measure with respect to the dis-baseline, in percentage points. For example, for *ReLIE/phone* and $\alpha = 1.2$, the adaptive algorithm achieves an F1 value of 82.1%+2.3%=84.4%. Detailed results on recall and precision can be found in Table 5.4 and Table 5.5. We verified the significance of the F1 measures

		baseline		competitor		adaptive approach		
task	original	dis	star	B&S	simple	$\alpha = 1.0$	$\alpha = 1.1$	$\alpha = 1.20$
ReLIE/phone	41.6	230.3	2.0	94.6	221.2	**46.8**	50.0	53.1
ReLIE/course	22.2	203.2	2.0	280.4	181.1	**33.7**	48.2	52.5
ReLIE/software	43.2	168.2	2.0	594.7	168.2	**54.4**	67.3	67.8
ReLIE/urls	52.4	630.3	2.0	5826.1	570.2	**70.6**	74.7	74.7
Enron/phone	17.0	199.1	2.0	243.8	164.8	**41.4**	**41.4**	**41.4**
Enron/date	17.0	170.6	2.0	581.0	170.6	**34.2**	**34.2**	35.6
YAGO/number	65.4	223.2	2.0	19471.0	207.9	**119.4**	120.5	120.7
YAGO/date	191.4	336.2	2.0	4337.0	313.6	203.6	203.6	**203.5**

Table 5.6 – Length of the repaired regexes (# of characters). Bold numbers indicate the shortest ones (without the original).

with a micro sign test [Yang and Liu, 1999]. We can see that, across almost all tasks and settings, our algorithms outperform the original regex as well as the dis-baseline.

The dis baseline has nearly no effect: It increases the recall by at most n words – too little to show up in the numbers. Likewise, the star baseline is unusable: It increases the recall trivially to 100 % in the ReLIE dataset, but precision suffers badly. In the Enron, where the task is to identify the positive example inside a full document, the regex cannot even achieve any recall. This is because it cannot discriminate the example from the rest of the document. For the YAGO dataset, about 31-34% of the documents consist of only the positive example. In the remaining documents, the regex also cannot discriminate the example from the rest of the document.

For most tasks, the simple algorithm improves the F1 measure only marginally. The reason is that in many cases the simple fixing algorithm made necessary parts of the original regex optional. As a consequence the intermediate regex becomes too general, and the simple algorithm reverts to the same regex as the dis-baseline.

The adaptive algorithm, however, manages to increase the F1 measure over all tasks. If α is small, the algorithm is conservative, and tends towards the dis-baseline. If α is larger, the algorithm performs repairs even if this generates more negative examples. As we can see, the impact of α is small. We take this as an indication that our method is robust to the choice of the parameter.

Regex length. Up to now, we have shown that our method achieves a comparable, and usually better F1-value than the baselines and the competitor. Now we look at the length of the repaired regexes to see how close the repaired regexes are to the original regex. Table 5.6 shows the average length of the generated regexes (in number of characters). The simple approach produces longer regexes because it often rejects the intermediate repaired regex, and reverts to adding the missing words as a disjunction. Therefore the length of the regexes of the simple approach is more similar to the length of the dis-baseline than to the length of the original regex.

The real value of the adaptive algorithm, however, comes from the much shorter regexes that it generates. For the adaptive algorithm, the length depends on α: If the value is large, the algorithm tends to integrate the words into the original regex. Then, the words are no longer subject to the generalization mechanism. Still, the impact of α is marginal: No matter the value, our algorithm produces regexes that are up to 8 times shorter than the dis-baseline, and nearly always at least twice as short as the competitor or the simple algorithm – at comparable or better precision and recall. We thus conclude that the regexes produced by the adaptive algorithm is better suited for the tasks than the competing algorithms.

Runtime. For the ReLIE and Enron dataset, all approaches take a time in the order of

	competitor		adaptive approach		
dataset	B&S	simple	$\alpha = 1.0$	$\alpha = 1.1$	$\alpha = 1.2$
ReLIE/phone	0.5	**0.3**	2.3	2.3	2.4
ReLIE/course	0.2	0.2	0.2	**0.1**	**0.1**
ReLIE/software	9.1	**0.6**	3.1	2.8	2.8
ReLIE/urls	**1.6**	4.4	1.9	2.0	**1.6**
Enron/phone	**0.4**	1.5	1.1	1.0	1.0
Enron/date	**0.2**	1.6	1.1	1.0	1.0
YAGO/number	16721.9	**17.9**	109.5	116.1	126.4
YAGO/date	396.5	**5.8**	58.3	54.1	58.0

Table 5.7 – Average runtime of the different algorithms, in seconds. Bold numbers indicate the shortest runtime.

seconds for repairing one regex. Due to the much larger E^+, runtimes differ for the YAGO dataset: fastest system is the simple algorithm with 12 s on average, followed by the adaptive algorithm with 84 s. The runtime of our reimplementation of [Babbar and Singh, 2010] lies in the order of minutes for the large YAGO dataset, as we did not optimize for runtime efficiency.

Runtimes for all systems are shown in Table 5.7. Table 5.8 shows the runtime for the different phases of the adaptive algorithm: the matching (Step 1), the fixing (Step 2-4), and the matching of the intermediate regexes on the set of negative examples E^- (which we call "feedback" in the table). In general, the runtime is in the order of seconds. For more complicated use cases (with longer regexes, longer missing words, and many negative examples), our method takes longer to execute. The matching phase takes more time if the regex contains nested quantifiers with a high maximal repetition number, as in `[a-z0-9]{0,30}){0,30}`. Most of the runtime is spent for the feedback. However, for reasonable inputs, our algorithm runs very fast: repairing a regex takes only some seconds.

Example. Table 5.9 shows an example of a repaired regex in the ReLIE/phone task for the adaptive algorithm. Our algorithm successfully identifies the non-matched characters `:` and > at the beginning of a phone number. It introduces them as options at the beginning of the original regex, leaving the rest of the regex intact. The dis-baseline, in contrast, would take the original regex, and add all words in a large disjunction. It is easy to see that our solution is more general and more syntactically similar to the original regex.

	$\alpha = 1.0$				$\alpha = 1.1$				$\alpha = 1.2$			
dataset	m.	f.	fb.	total	m.	f.	fb.	total	m.	f.	fb.	total
ReLIE/phone	0.2	0.0	2.1	2.3	0.1	0.0	2.2	2.3	0.1	0.0	2.2	2.4
ReLIE/course	0.1	0.0	0.1	0.2	0.1	0.0	0.0	0.1	0.1	0.0	0.0	0.1
ReLIE/software	0.3	0.1	2.8	3.1	0.2	0.0	2.5	2.8	0.3	0.0	2.6	2.8
ReLIE/urls	1.5	0.1	0.3	1.9	1.6	0.1	0.3	2.0	1.2	0.1	0.3	1.6
Enron/phone	0.1	0.0	1.0	1.1	0.1	0.0	0.9	1.0	0.1	0.0	0.9	1.0
Enron/date	0.1	0.0	1.0	1.1	0.0	0.0	0.9	1.0	0.0	0.0	0.9	1.0
YAGO/number	1.3	0.0	108.1	109.5	1.4	0.0	114.7	116.1	1.4	0.0	125.0	126.4
YAGO/date	16.1	0.0	42.2	58.3	15.7	0.0	38.4	54.1	16.1	0.0	41.8	58.0

Table 5.8 – Runtime of the different phases of the adaptive algorithm: matching (m.), fixing (f.), feedback (fb.), and for comparison the total time

<table>
<tr><td>Original regex:</td><td colspan="3"><code>({0,1} \d{3}){0,1}(-|\.|) \d{3}(-|\.|) d{4}</code></td></tr>
<tr><td>Missing words:</td><td><code>:734-763-2200</code></td><td><code>>317.569.8903</code></td><td><code>>443.436.0787</code></td></tr>
<tr><td></td><td><code>>512.289.1407</code></td><td><code>>734-615-9673</code></td><td><code>>734-647-8027</code></td></tr>
<tr><td></td><td><code>>734-763-5664</code></td><td><code>>734.647.3256</code></td><td></td></tr>
<tr><td></td><td><code>>773.339.3223</code></td><td><code>>804.360.2356</code></td><td></td></tr>
<tr><td>Repaired regex:</td><td colspan="3"><code>((|:|>)?\d{3})?(-|\.|)\d{3}(-|\.|)\d{4}</code></td></tr>
</table>

Table 5.9 – Example of a scenario for the ReLIE/phone task

5.7 Demo

We have developed a demo that allows the user to visualize the regex repairs of the simple algorithm. In our interface, the user can enter a regex and a missing word. For example, the user can choose the regex `\d{2}/\d{2}/\d{4}`, and the missing word `1/8/1935`. The demo then proceeds to show the user the maximal matching between the regex and the missing word, as computed by Myers' algorithm (Figure 5.5 on the facing page). In the following, the user can walk step by step through the modifications that our algorithm proposes. In the example, the algorithm proposes to make first the first digit optional, and then the third (Figure 5.6 left). Finally, the algorithm simplifies the regex to `(\d?\d/){2}\d{4}` (Figure 5.6 right).

Alternatively, the user can choose one of our pre-defined sets of words. We offer the ReLIE dataset, the YAGO dataset, and the Enron dataset. The user can then enter a regex that they think captures the words. We show the user which words are not captured, and the user can choose to add one of them to the regex. The user can then run the

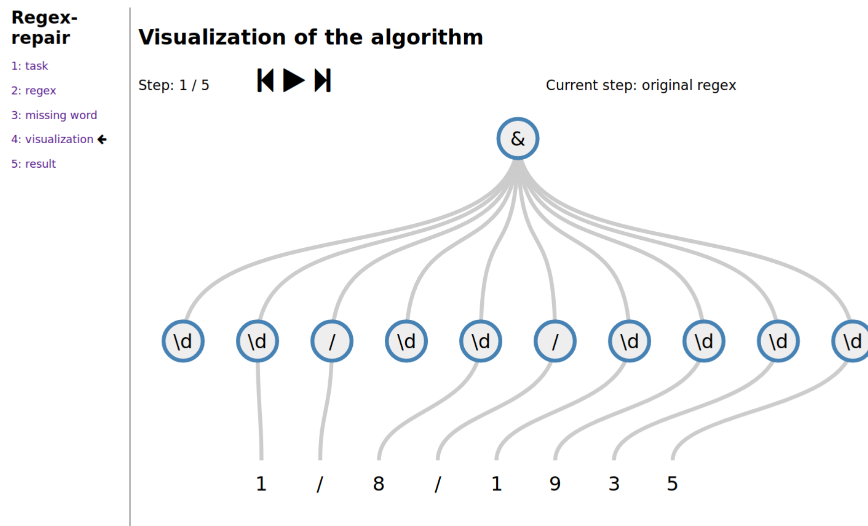


Figure 5.5 – Demo screenshot

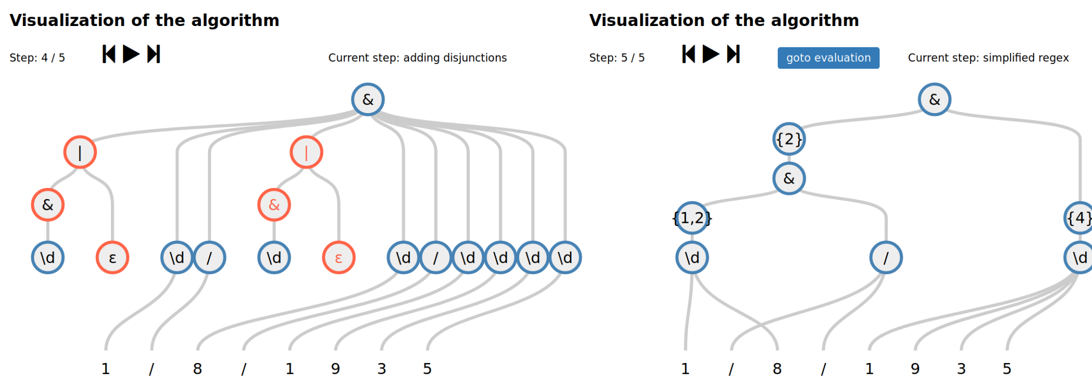


Figure 5.6 – Demo screenshots

modified regex again to see how many more words are captured, and which ones are still missing. This way, the user can see how the algorithm performs on real data.

5.8 Summary

In this chapter, we have proposed an algorithm that can add missing words to a given regular expression. With only a small set of positive examples, our method generalizes the input regex, while maintaining its structure. In this way, our approach improves the precision and recall of the original regex. Our method maintains the parts of the regular

expression that already match the input words. It adds alternatives at the right points of the regex, so as to match the substrings that were not yet covered.

We have evaluated our method on various datasets, and we have shown that already with a very small set of positive examples, we can improve the F1 measure on the ground truth. This is a remarkable result, because it shows that regexes can be generalized based on very small training data. What is more, the adaptive algorithm produces regexes that are significantly shorter than the baseline, the competitor, and the simple algorithm. This shows that our method generalizes the regexes in a meaningful and non-trivial way.

Experimental results are available online at `https://thomasrebele.org/projects/regex-repair`. Both the source code for the simple and the adaptive algorithm can be obtained at `https://github.com/thomasrebele/regex-repair`.

5.9 Outlook

The approach presented in this chapter can be extended in several ways. First, the algorithm could detect when to generalize the components of the regex into character classes. Second, there are often several possibilities to add the missing parts in disjunctions. A machine learning approach could determine the best positions of the newly created disjunctions. Finally, more advanced algorithm for simplifying the final regex could be developed. Even though the adaptive algorithm produces quite short regexes, they are still considerably longer than the original regex.

Until now we have investigated algorithms for extracting more information from the sources. This raises the question of how such large knowledge bases can be handled efficiently. In the next chapter we will see how we can preprocess large knowledge bases.

Chapter 6

Answering Queries with Unix Shell

As we have seen in the previous chapters, some knowledge bases contain a huge number of facts. Dealing with such large datasets often requires extensive preprocessing. This preprocessing happens only once, so that loading and indexing the data in a database or triple store may be an overkill. In this chapter, we present an approach that allows preprocessing large tabular data in Datalog – without indexing the data. The Datalog query is translated to Unix Bash and can be executed in a shell. Our experiments show that, for the use case of data preprocessing, our approach is competitive with state-of-the-art systems in terms of scalability and speed, while at the same time requiring only a Bash shell, and a Unix-compatible operating system. We also provide a basic SPARQL and OWL 2 to Datalog converter to make our system interoperable with Semantic Web standards.

This chapter is based on the following publication:

> Rebele, T., Tanon, T. P., and Suchanek, F. M. (2018a). Technical Report: Answering Datalog Queries with Unix Shell Commands. Technical report. URL: `https://www.thomasrebele.org/publications/2018_report_bashlog.pdf` (technical report, under review)

6.1 Introduction

Many data analytics tasks work on tabular data. Such data can take the form of relational tables, TAB-separated files, or knowledge bases from the Semantic Web in the form of subject-predicate-object triples. Quite often, such data has to be preprocessed before the analysis can be made. In this chapter, we focus on preprocessing in the form of select-project-join-union operations with recursion. This may include the removal of superfluous columns, the selection of rows of interest, or the amendment of certain rows by performing a join with another tabular dataset. In the case of knowledge bases, the

preprocessing may involve extracting all instances of a certain class; in the case of graph data, the preprocessing may involve finding all nodes that are reachable from a certain node. These operations require recursion.

The defining characteristic of such pre-processing steps is that they are executed only once on the data in order to constitute the dataset of interest for the later analysis. It is only after such preprocessing that the actual data analysis task begins. This one-time pre-processing is the task that this chapter is concerned with.

While there exist databases (or triple stores) to help with this preprocessing, loading large amounts of data into these systems may take hours or even days. Wikidata [Vrandecic and Krötzsch, 2014], for example, one of the largest knowledge bases on the Semantic Web, contains 267 GB of data. If only a small portion of the data is needed afterwards, then it is an overkill in terms of time and space consumption to first load and index the entire dataset. After loading, purging the superfluous elements may again take several days, because indexes have to be rebuilt. All of this is frustratingly slow, as people who have worked with such data can confirm.

There are a number of systems that can work directly on the data, such as DLV [Leone et al., 2006] or RDFox [Motik et al., 2014]. However, these systems load the data into memory. While this works well for small datasets, it does not work for larger ones, such as Wikidata (as we show in our experiments). We are thus facing the problem of preprocessing large datasets that are not indexed, and that do not fit into main memory. There are tools to help with this (such as Spark [Zaharia et al., 2010], Flink [Carbone et al., 2015], Dryad [Isard et al., 2007], Impala [Kornacker et al., 2015]), but these require the installation of particular software, the knowledge of particular programming languages, or even a particular distributed infrastructure.

In this chapter, we develop a method to preprocess tabular file data without indexing it. We propose to express the preprocessing steps in Datalog [Abiteboul et al., 1995]. Datalog is a particularly simple language, which has just a single syntactic construction, and no reserved keywords. Nevertheless, it is expressive enough to deal with joins, unions, projections, selections, negation, and in particular also with the recursivity that is required for preprocessing knowledge bases and graphs. We propose to compile this Datalog program automatically to Unix Bash Shell commands. We offer a Web page to this end: `https://www.thomasrebele.org/projects/bashlog`. The user can just enter the Datalog program, and click a button to obtain the Bash code. The Bash code can be copy-pasted into a Unix Shell, and executed without any prerequisites. Our method automatically optimizes the Datalog program with standard relational algebra optimization techniques, re-uses previously computed intermediate results, and produces a highly parallelized Shell script. For this purpose, our method employs pipes and process substitution. Our experiments on a variety of datasets show

that this method is competitive in terms of runtime with state-of-the-art database systems, Datalog query answering engines, and triple stores.

More concisely, our approach allows the one-time preprocessing of tabular data in the form of select-project-join operations with negation and recursion

1. without any software installation beyond a Unix Bash shell on a POSIX compliant operating system
2. without any knowledge of programming or query languages other than Datalog
3. in a time that is competitive with conventional systems

The contributions of this chapter are:

— a method that compiles Datalog to Unix Bash Shell commands
— the optimization of such programs
— extensive experiments on real datasets that show the viability of our method

This chapter is structured as follows. We start with a motivating example in Section 6.2. Section 6.3 discusses related work, before Section 6.4 introduces preliminaries. Section 6.5 presents our approach, and Section 6.6 evaluates it. Section 6.7 shows how to use the Web interface, before Section 6.8 concludes.

6.2 Example

Setting. Since our approach may appear slightly unorthodox, let us illustrate our method by a concrete example. Consider a knowledge base of the Semantic Web – for example BabelNet, DBpedia, YAGO, or Wikidata. As described in Section 2.1, these knowledge bases contain instances (such as <New_York_City> or <USA>), and these instances belong to certain classes (such as <City> or <Country>). The classes of a knowledge base form a hierarchy, where more specific classes (such as <President>) are included in more general classes (such as <Politician>). This data is typically stored in RDF. For simplicity and readability, assume that the data resides in a TAB-separated file *facts.tsv*:

```
<Empire_State_Building>    <locatedIn>    <Manhattan>
<Manhattan>                <locatedIn>    <New_York_City>
<New_York_City>            <locatedIn>    <USA>
```

Figure 6.1 – An excerpt from a knowledge base (facts.tsv)

Now consider a data engineer who wants to recursively extract all places located in the United States. Figure 6.2 shows how this query can be expressed in our Datalog dialect.

The first line says that the predicate *fact* can be computed by printing out the file *facts.tsv*. Note the tilde, which signals that the body of the rule is a Unix command. The second line of the program says that the *locatedIn* predicate is obtained by selecting those facts with the predicate *locatedIn*. The third and fourth line say that we are interested in all places that are located in the United States.

```
fact(X, R, Y)  :~  cat facts.tsv
locatedIn(X, Y) :- fact(X, "<locatedIn>", Y) .
locatedIn(X, Y) :- locatedIn(X, Z), locatedIn(Z, Y) .
main(X) :- locatedIn(X, "<USA>") .
```

Figure 6.2 – A Datalog program for finding places in the United States.

Translation. We propose to compile such Datalog programs automatically into Unix Bash Shell commands. For this purpose, the user can just visit our Web page and copy-paste the Datalog program there. She will then obtain a script similar to the code shown in Figure 6.3 (for readability, we have omitted a number of parameters, *sort* commands, and optimizations in this example). The code first extracts the <locatedIn> facts from *facts.tsv*. From these facts, it extracts the places directly located in the <USA>, and stores them in a file *delta.tmp* and in a file *full.tmp*. In the following *while* loop, the delta file is joined with all <locatedIn> facts. The classes that had already been found previously are filtered out, and the remaining ones are added to *full.tmp* and put into the delta file. If that delta file is empty, a fixed point has been reached, and the loop stops.

```
awk '$2 == "<locatedIn>" {print $1 "\t" $3}' facts.tsv > li.tmp
awk '$2 == "<USA>" {print $0}' li.tmp | tee full.tmp > delta.tmp
while
    join li.tmp delta.tmp | comm -23 - full.tmp > new.tmp
    mv  new.tmp  delta.tmp
    sort -m -o full.tmp  full.tmp delta.tmp
    [ -s delta.tmp ];
do continue; done
cat full.tmp
```

Figure 6.3 – The Datalog program in Bash (simplified).

This code can either be saved in a Shell script file, or else directly copy-pasted into the command-line prompt. When run, the code produces the list of places in the United States. This list is written to the standard output, and can be saved in a file.

Rationale. Such a solution has several advantages. First, it does not require any soft-

ware installation. Installing and getting to run a complex system, such as BigDatalog [Shkapsky et al., 2016], e.g., can take several hours. Our solution just requires a visit to a Web site. Second, the Bash shell has been around for several decades, and the commands are not just tried and tested, but actually continuously developed. Modern implementations of the *sort* command, e.g., can split the input into several portions that fit into memory, and sort them individually. Finally, the Bash shell allows executing several processes in parallel, and their communication is managed by the operating system.

6.3 Related Work

Relational Databases. Relational database management systems can handle data in the form of tables. Such systems include Oracle, IBM DB2, Postgres, and MySQL, as well as newer systems, such as MonetDB [Boncz et al., 2008], and NoDB [Alagiannis et al., 2012].

All of these systems (except NoDB) require loading the data and indexing it in its entirety. If the preprocessing is executed only once, then this time overhead may not pay off. We show in our experiments that just loading the data can take much longer than the entire preprocessing with our method. Furthermore, all of these systems (including NoDB) require the installation and setting up of software. Our approach, in contrast, can be run as a simple jar file, or even just as a service on the Web. The resulting Bash script then runs in a common shell console without any further prerequisites.

Triple Stores. Another class of systems target RDF knowledge bases. These are called *triple stores* and include OpenLink Virtuoso [Erling and Mikhailov, 2009], Stardog, Jena [Carroll et al., 2004], and others. Again, these require the loading and indexing of the data, and we will show that this is slower than our method for the purpose of preprocessing. Several approaches aim to speed up this loading: HDT [Fernández et al., 2013] is a binary format for RDF, which can be used with Jena. Still, we find that this combination cannot deliver the speed of Bash. Linked Data Fragments [Verborgh et al., 2016] aim to strike a balance between downloading an RDF data dump and querying it on a server. The method thus addresses a slightly different problem. Apart from this, all of these approaches require the installation of software, while our approach works in a Bash shell.

NoSQL Databases. Several other data management systems target non-tabular data. These can be full-text indexing systems or key-value stores. Approaches, such as Cas-

sandra, HBase, and Google's BigTable [Chang et al., 2008], target particularly large data. Our method, in contrast, aims at tabular data.

Distributed Processing. Distributed batch processing is a well known problem. The major paradigm used is Map-Reduce [Dean and Ghemawat, 2008]. Dryad [Isard et al., 2007] provides a DAG dataflow system where the user can specify their own functions. These ideas have been implemented in systems, such as Apache Tez [Saha et al., 2015]. SCOPE [Zhou et al., 2010], Impala [Kornacker et al., 2015], Apache Spark [Zaharia et al., 2010], and Apache Flink [Carbone et al., 2015] provide advanced features, such as support for SQL or streams. While all these systems address our problem, they require the installation of particular software. What is more, they also require a distributed infrastructure. Our approach, in contrast, requires neither the installation of software nor a particular physical infrastructure. It just requires a Bash shell.

Datalog. The execution of Datalog is an active research topic, and the parallel processing of Datalog has been studied for over twenty years [Wolfson and Silberschatz, 1988; Ganguly et al., 1990; Ganguly et al., 1992]. Several recent works have taken to improve the performance of Datalog execution by the use of modern data processing systems. For example, the work of [Shaw et al., 2012] ports the usual semi-naive evaluation algorithm [Abiteboul et al., 1995] to Hadoop. The work of [Bu et al., 2012] executes Datalog on top of both Map-Reduce [Dean and Ghemawat, 2008] and Pregel [Malewicz et al., 2010]. Myria [Wang et al., 2015] provides a parallel distributed pipeline to evaluate Datalog programs and uses Postgres for storing facts. Recent works have used Apache Spark [Zaharia et al., 2010]. BigDatalog [Shkapsky et al., 2016], in particular, tackles the problem of recursion in Spark. DatalogRA [Rogala et al., 2016] deals with Datalog with data aggregation, and the work of [Wu et al., 2016] uses the naive evaluation strategy to evaluate Datalog programs and OWL ontologies. There is also recent work on recursive query evaluation on top of Spark [Katsogridakis et al., 2017]. The RDFox system [Motik et al., 2014] is specialized on Datalog queries on RDF data. There are also systems that can preprocess RDF datasets by filtering their content by SPARQL queries [Marx et al., 2013]. An example of such systems is RDFSlice [Marx et al., 2017]. It supports simple filtering and certain types of joins.

All of these systems address the same problem as us. Then again, all of these systems require the installation of software. The parallelized systems also require a distributed infrastructure. Our approach, in contrast, requires none of these. Nevertheless, we show in our experiments that the performance of our approach is competitive with the state of the art in the domain.

OWL Reasoners. OWL is an ontology language for Semantic Web data. Several sys-

tems can perform OWL reasoning. These include, e.g., Pellet [Sirin and Parsia, 2004], HermiT [Shearer et al., 2008], RACER [Haarslev and Möller, 2001], and Fact++ [Tsarkov and Horrocks, 2006]. Jena also supports OWL reasoning. These systems support negation, existential variables, and functional constraints. In this chapter, we aim at a much simpler pre-processing language, Datalog. Datalog corresponds to a subset of the OWL 2 RL profile [Cao et al., 2011]. Thus, OWL reasoners are an overkill for our scenario. We make this point by comparing our approach with Jena and the Pellet successor Stardog.

6.4 Preliminaries

Datalog. We follow the definition of Datalog with negation from [Abiteboul and Hull, 1988; Abiteboul et al., 1995]. In all of the following, we assume 3 distinct sets of identifiers: predicates $\mathcal{P}$, variables $\mathcal{V}$, and constants $\mathcal{C}$. An n-ary *atom* is of the form $p(a_1, \dots, a_n)$, with $p \in \mathcal{P}$ and $a_i \in \mathcal{C} \cup \mathcal{V}$ for $i = 1 \dots n$. An atom is *grounded* if it does not contain variables. A *rule* takes the form

$$H :- B_1, \dots, B_n, \neg N_1, \dots, \neg N_m.$$

Here, H is the *head atom*, and $B_1, \dots, B_n, N_1, \dots, N_m$ are the *body atoms*. For $n = 0$, the rule simply takes the form "H.". We say that the body atoms $N_1, \dots, N_m$ are *negated*. A rule is *safe* if each variable in the head or in a negated atom also appears in at least one positive body atom. We consider only safe rules in this work. A *Datalog program* is a set of rules. A set M of grounded atoms is a *model* of a program P, if the following holds: M contains an atom a if and only if P contains a rule $H :- B_1, \dots, B_n, \neg N_1, \dots, \neg N_m$, such that there exists a substitution $\sigma : \mathcal{V} \to \mathcal{C}$ with $\sigma(B_i) \in M$ for $i = 1 \dots n$ and $\sigma(N_i) \notin M$ for $i = 1 \dots m$ and $a = \sigma(H)$. A model is *minimal* if no proper subset is a model. In order to ensure the existence and the uniqueness of a minimal model for each given program with negation, we restrict ourselves to *stratified* Datalog programs [Abiteboul and Hull, 1988; Abiteboul et al., 1995]. A Datalog program is stratified, if there exists a function σ from predicates to $\mathbb{N}$ such that for all rules of the form $H :- \dots, B_i, \dots$, we have $\sigma(H) \geq \sigma(B_i)$, and for all rules of the form $H :- \dots, \neg N_j, \dots$, we have $\sigma(H) > \sigma(N_j)$.

Relational Algebra. *Relational algebra* [Codd, 1970; Abiteboul et al., 1995] provides the semantics of relational database operations. There exist many different variants of relational algebra. Here, we want to use a variant that is equivalent to Datalog. A *table* is a set of tuples of the same arity. We write $arity(\cdot)$ for the arity of a tuple or the arity of the tuples in a set. We call *SPJAU unnamed relational algebra* the following set of operators on tables T and T' [Abiteboul et al., 1995]:

Select (column equality): For $i, j \in [1, \dots, arity(T)]$,
$\sigma_{i=j}(T) = \{t \in T \mid t(i) = t(j)\}$.

Select (column-value equality): For $i \in [1, \dots, arity(T)]$,
$\sigma_{i=a}(T) = \{t \in T \mid t(i) = a\}$.

Project: For $i_1, \dots, i_k \in [1, \dots, arity(T)]$,
$\pi_{i_1,\dots,i_k}(T) = \{\langle t(i_1), \dots, t(i_k)\rangle \mid t \in T\}$

Constant Introduction: For $i \in [1, \dots, arity(T) + 1]$,
$\pi_{i:a}(T) = \{\langle t(1), \dots, t(i-1), a, t(i), \dots, t(arity(T))\rangle \mid t \in T\}$.

Join: For $i_1, \dots, i_k \in [1, \dots, arity(T)], i'_1, \dots, i'_k \in [1, \dots, arity(T')]$,
$T \bowtie_{i_1=i'_1,\dots,i_k=i'_k} T' = \{\langle t, t'\rangle \mid t \in T \wedge t' \in T' \wedge t(i_1) = t'(i'_1) \wedge \dots \wedge t(i_k) = t'(i'_k)\}$

Anti-join: For $n = arity(T')$ and $i_1, \dots, i_n \in [1, \dots, arity(T)]$,
$T \rhd_{i_1,\dots,i_n} T' = \{t \mid t \in T \wedge \neg \exists t' \in T' : t(i_1) = t'(1) \wedge \dots \wedge t(i_n) = t'(n)\}$

Union: $T \cup T'$ is the usual set union,
$T \cup T' = \{t \mid t \in T \vee t \in T'\}$

We often consider relational algebra expressions as syntax tree. The outermost operator represents the root. A node a is a child of another node b, if the output of a serves as input of b. Descendants of a node a are all nodes, who are children of a or children of a descendant of a.

To map Datalog programs to relational algebra, we need an operator for recursive programs. For this, the work of [Aho and Ullman, 1979] introduces a *least fixed point operator* (LFP).[1] For a function f from a table to a table, $LFP(f)$ is the least fix point of f for the $\subseteq$ relation. The least fixed point can be computed with the semi-naive algorithm [Abiteboul et al., 1995], as shown in Algorithm 11. This operator allows expressing all Datalog programs with stratified negation. We call *SPJAUR unnamed relational algebra* the SPJAU algebra extended with this operator. This algebra has the same expressivity as safe stratified Datalog programs [Abiteboul et al., 1995].

Algorithm 11 Computation of $LFP(f)$ using the semi-naive algorithm

INPUT: function f
OUTPUT: table
1: $result \leftarrow \emptyset$
2: $\Delta \leftarrow \emptyset$
3: **repeat**
4: $\quad \Delta \leftarrow f(\Delta) \setminus result$
5: $\quad result \leftarrow result \cup \Delta$
6: **until** $\Delta = \emptyset$
7: **return** result

1. The work of [Agrawal, 1988] introduces a different relational algebra operator called α. However, α can express only transitive closures, and not arbitrary recursions.

> *Example (Relational Algebra):* Assume that there is a table *subclass* (which contains classes with their superclasses). Then the following expression computes the transitive closure of this table:
>
> $$LFP(\lambda x: \text{ subclass} \cup \pi_{1,4}(x \bowtie_{2=1} x))$$
>
> This expression computes the least fix point of a function. The function is given by a lambda expression. To compute the result of this expression, we execute the function first with the empty table, $x = \emptyset$. Then the function returns the *subclass* table. Then we execute the function again on this result. This time, the function joins *subclass* with itself, projects the resulting 4-column table on the first and last column, and adds in the original *subclass* table. We repeat this process until no more changes occur. This process terminates eventually, because the operators of our algebra are all monotonous – with the exception of the anti-join. Since our programs are stratified, x does not occur as the second argument of an anti-join, and thus the second argument does not change between iterations.

Unix. Unix is a family of multitasking computer operating systems. Unix and Unix-like systems are widely used on servers, on smartphones (e.g., Android OS), and on desktop computers (e.g., Apple's MacOS). One of the characteristics of Unix is that "Everything is a file", which means that files, pipes, the standard output, the standard input, and other resources can all be seen as streams of bytes [2]. For the present work, we are interested only in *TAB-separated* byte streams, i.e., byte streams that consist of several *rows* (sequences of bytes separated by a newline character), which each consist of the same number of *columns* (sequences of bytes separated by a tabulator character). When printed, these byte streams look like a table.

The Bourne-again shell (Bash) is a command-line interface for Unix-like operating systems. It is the default interactive shell for users on most Linux and MacOS systems [Wikipedia, 2017]. A *Bash command* is either a built-in keyword, or a small program. We are here concerned mainly with those commands of the POSIX standard that take one or several byte streams as input, and that produce one byte stream by printing to the standard output. We will use the following commands with the following parameters:

cat b_1 **...** b_n
: Prints the byte streams b_1 ... b_n one after the other.

sort -t $'\t' -k**$c_1$ **_-kc_n b
: Sorts the byte stream b on columns $c_1, \ldots, c_n$ and prints the result.

2. according to Linus Torvalds, the creator of Linux, `http://yarchive.net/comp/linux/everything_is_file.html`

sort -u -m -o b_0 b_1 b_2
: Merges the sorted byte streams b_1 and b_2, eliminating duplicate lines, and prints the output to b_0.

comm -23 b_1 b_2
: Prints the lines that appear in the sorted byte stream b_1, but not in the sorted byte stream b_2.

join -t $'\t' -1**$c_1$ **-2c_2 **-o** d **[-v1]** b_1 b_2
: Joins the byte streams b_1 and b_2 on column c_1 of b_1 and column c_2 of b_2, and prints the output columns d of the result. For this, b_1 has to be sorted on column c_1, and b_2 has to be sorted on column c_2. The command supports joining on a single column only. With `-v1`, the command outputs those lines of b_1 that could not be joined.

echo -n > f
: Creates an empty file f, overwriting f if it exists.

mv f_1 f_2
: Renames file f_1 to f_2.

AWK. We will also use the command `awk`. It implements an interpreter for the AWK programming language. We use awk commands of the following form

```
awk -F$'\t'  'p' b
```

The `-F` option makes awk use the TAB character for column separation. This character can then be referred to as `FS`. b denotes the input byte stream, and p is an awk program of the following form:

c **{ print $**$i_1$ **FS** ... **FS $**$i_k$ **[>> "**f**"] }**
: This AWK program prints out the columns $i_1, \ldots, i_k$ of the input byte stream, if a certain condition c is fulfilled. A condition is either a column equality `$`i `==` `$`j, a column-value equality `$`i `== "value"`, or a combination of several conditions c_1 && ... && c_n. An empty condition always succeeds. If the optional >> "f" is given, the output is appended to file f.

{ print $0 FS $**$i_1$ s ... s **$**$i_k$ **[>> "f**"] }**
: This AWK program prints a line of the input byte stream, and appends a single column to it. This single column is the concatenation of the columns $i_1, \ldots, i_k$, separated by the character s. We used the ASCII character 002 for this purpose, but another character can be used, as long as it does not appear in the Datalog program[3]. If a file f is given, the result is appended to f. We use this program to create a column on which we can run the `join` command.

3. other excluded characters are ASCII characters 000, 001, and TAB

Finally, we make use of the Bash control structure `while`, which we use as follows:

```
while c [ -s f ];
    do continue;
done
```

This code runs the sequence of commands *c* repeatedly until the file *f* is empty.

Pipes. When a command or control structure is executed, it becomes a *process*. In the Unix-like operating systems, processes can communicate through *pipes*. A pipe is a byte stream that can be filled by one process, and read by another process. If the producing process is faster than the receiving one, the pipe buffers the stream, or blocks the producing process if necessary. In Bash, pipes can be constructed as follows:

```
p1 | p2
```

This construction sends the output of process p_1 as input to process p_2. If p_2 is a command, the input byte stream no longer has to be specified explicitly. A process *p* can also send its output byte stream to two other byte streams b_1 and b_2 (including pipes or files), as follows:

```
p | tee b1 [b3 ...] > b2
```

A pipe can also be constructed "on the fly" by a so-called *process substitution*, as follows:

```
p1 <( p2 )
```

This construction runs the process p_2, and pipes its output stream into the first argument of the process p_1. Finally, it is possible to create a *named pipe n* with the command `mkfifo` *n*. Such a pipe can be closed with the Bash command `exec` *d*`>`*n*`; exec` *d*`>&-`, where *d* is an integer greater than 2, representing a not yet used file descriptor.

We will now see how these constructions can be used to execute Datalog programs.

6.5 Approach

6.5.1 Datalog Dialect

In our concrete application of Datalog, we assume that the set $\mathcal{P}$ of predicates is the set of strings that consist of letters, and that start with a lower-case letter. The set of variables $\mathcal{V}$ is the set of strings that consist of letters, and that start with an upper-case letter. The set $\mathcal{C}$ of constants is the set of all strings that start and end with an ASCII double quotation mark. Constants may not contain ASCII double quotation marks other than the two delimiters. They may also not contain TAB characters, newline characters, or the separator character that we use in the AWK programs. Future versions of our compiler may relax these restrictions, but for the present work we stay with these conventions for readability.

For our purposes, the Datalog program has to refer to files or byte streams of data. For this reason, we introduce an additional type of rules, which we call *command rules*. A command rule takes the following form:

> $p(x_1, ..., x_n)$:~ c

Here, p is a predicate, $x_1, \ldots, x_n$ are variables, and c is a Bash command. Such a rule ends syntactically not with a dot (because Bash commands often contain dots), but with a new line. Notice the tilde in the place of the usual hyphen to distinguish command rules from ordinary rules. Semantically, this rule means that executing c produces a TAB-separated byte stream of n columns, which will be referred to by the predicate p in the Datalog program. In the simplest case, the command c just prints a file, as in `cat facts.tsv`. However, the command can also be any other Bash command, such as `ls -1`.

Our goal is to compute a certain output with the Datalog program. This output is designated by the distinguished head predicate `main`. An *answer* of the program is a grounded variant of the head atom of this rule that appears in the minimal model of the program. See again Figure 6.2 on page 92 for an example of a Datalog program in our dialect. We emphasize that our dialect is a generalization of standard Datalog, so that any normal Datalog program can be run directly in our system.

Our approach can also work in "RDF mode". In that mode, the input consists of a SPARQL query [Harris et al., 2013], a TBox in the form of OWL 2 RL [Cao et al., 2011], and an ABox in the form of an N-Triples file F (see Section 2.2.1 on page 23). We convert the OWL ontology and the SPARQL query to Datalog rules, and we include the following AWK command in the Bash script to transform file F to a TSV file:

```
fact(S, P, O)  :~  awk '{ sub(" ", "\t");  sub(" ", "\t");  ↩
    sub(/ \.$/, "");  print $0 }' F
```

The command replaces the spaces that separate the three parts by TAB characters, and removes the dot character at the end. If necessary, a similar AWK command can transform the output of the Bash script back to the N-Triples format.

The Datalog program contains predicate `main`, which returns the result of the SPARQL query on the file F, while having used the provided ontology for expansion. For example, we are able to produce the rule `hasParent`(X, Y) :− `hasFather`(X, Y) from the OWL axiom *subPropertyOf*(*hasFather*, *hasParent*). Like RDFox [Motik et al., 2014], we assume that all classes and properties axioms are provided by the ontology, and that they are not queried by the SPARQL query. This assumption allows us to produce efficient programs. We do not yet support OWL axioms related to literals. Our SPARQL implementation supports basics graph patterns, property paths without negations, OPTIONAL, UNION and MINUS.

6.5.2 Loading Datalog

Our approach takes as input a Datalog program, and produces as output a Bash Shell script. For this purpose, our approach first builds a relational algebra expression for the `main` predicate of the Datalog program with Algorithm 12 on the following page. The algorithm takes as input a predicate p, a cache, and a Datalog program P. The method is initially called with p=`main`, *cache*=∅, and the Datalog program that we want to translate. The cache stores already computed relational algebra plans. In all of the following, we assume that p always appears with the same arity in P. If that is not the case, p can be replaced by different predicates, one for each arity. The full materialization of predicate p is the least fixed point of the union of the application of all the rules producing p.

Our algorithm first checks whether p appears in the cache (Line 2-4). In that case, p is currently being computed in a previous recursive call of the method, and the algorithm returns a variable x indexed by p (Line 3). This is the variable for which we compute the least fix point.

Then, the algorithm traverses all rules with p in the head (Line 7). For every rule

$$p_h(H_1, \ldots, H_{n_h}) :- r_1(X_1^1, \ldots, X_{n_1}^1), \ldots r_n(X_1^n, \ldots, X_{n_n}^n), \\ \neg q_1(Y_1^1, \ldots, Y_{m_1}^1), \ldots, \neg q_m(Y_1^m, \ldots, Y_{m_m}^m)$$

the algorithm recursively retrieves the plan for the r_i (Line 9-15), and the q_j (Line 16-22). It then adds a nested $\sigma_{j=k}$ if there are j, k such that $X_j^i = X_k^i$ (Line 11-13), and $Y_j^i = Y_k^i$ (Line 18-20) respectively. Then it combines these expressions pair-wise from left to right

Algorithm 12 Translation from Datalog to SPJAUR algebra

INPUT: predicate p, cache, Datalog program P
OUTPUT: algebra plan for p

1: **procedure** *mapPred*(p, *cache*, P)
2: **if** $p \in cache$ **then**
3: **return** x_p
4: **end if**
5: $plan \leftarrow \varnothing$
6: $newCache \leftarrow cache \cup \{p\}$
7: **for** rule $p(H_1, \ldots, H_{n_h}) :- r_1(X^1_1, \ldots, X^1_{n_1}), \ldots, r_n(X^n_1, \ldots, X^n_{n_n}), \neg q_1(Y^1_1, \ldots, Y^1_{m_1}), \ldots, \neg q_m(Y^m_1, \ldots, Y^m_{m_m})$ in P **do**
8: $bodyPlan \leftarrow \{()\}$
9: **for** $r_i(X^i_1, \ldots, X^i_{n_i})$ **do**
10: $atomPlan \leftarrow mapPred(r_i, newCache, P)$
11: **for** $(X^i_j, X^i_k) \mid X^i_j = X^i_k, j \neq k$ **do**
12: $atomPlan \leftarrow \sigma_{X^i_j = X^i_k}(atomPlan)$
13: **end for**
14: $bodyPlan \leftarrow bodyPlan \bowtie atomPlan$
15: **end for**
16: **for** $\neg q_i(Y^i_1, \ldots, Y^i_{m_i})$ **do**
17: $atomPlan \leftarrow mapPred(q_i, newCache, P)$
18: **for** $(Y^i_j, Y^i_k) \mid Y^i_j = Y^i_k, j \neq k$ **do**
19: $atomPlan \leftarrow \sigma_{Y^i_j = Y^i_k}(atomPlan)$
20: **end for**
21: $bodyPlan \leftarrow bodyPlan \rhd atomPlan$
22: **end for**
23: $plan \leftarrow plan \cup \pi_{H_1, \ldots, H_{n_h}}(bodyPlan)$
24: **end for**
25: **for** rule $p(H_1, \ldots, H_{n_h}) :\sim c$ in P **do**
26: $plan \leftarrow plan \cup \pi_{H_1, \ldots, H_{n_h}}([c])$
27: **end for**
28: **return** $LFP(\lambda x_p : plan)$
29: **end procedure**

by adding the relevant join constraints between the r_i (Line 14). It also adds the anti-join constraints between the results of the combinations of the left elements and the q_j (Line 21). At the end of the for-loop, the algorithm puts the resulting formula into a project-node that extracts the relevant columns (Line 23). Then, the algorithm processes all command rules, wraps each command in a project-node, and adds it to the plan (Line 26). Finally, the algorithm wraps the plan in a least fixed point operator (Line 28). A subsequent optimization step removes this operator if it is not necessary.

We can add the anti-join constraints here, because the program is stratified. That means, applying `mapPred` on a negated atom q_j never reaches a rule with head p again. Furthermore, the program is safe, so all columns returned by the second parameter of the anti-join appear in the columns of the first parameter.

The implementation of the algorithm builds a directed acyclic graph (DAG) instead of a tree. When the function `mapPred` is called with the same arguments as in a previous call, it returns the result of the previous call. This implementation allows us to re-use the same sub-plan multiple times in the final query plan, thereby reducing its size. The technique also allows the Bash programs to re-use results that have already been computed.

Example (Datalog Translation): Assume that there is a two-column TAB-separated file `subclass.tsv`, which contains each class with its subclasses. Consider the following Datalog program P:

```
1 directSubclass(x,y)  :~  cat subclass.tsv
2 main(x,y) :- directSubclass(x,y).
3 main(x,z) :- directSubclass(x,y), main(y,z).
```

We call $mapPred(\texttt{main}, \varnothing, P)$. Our algorithm goes through all rules with the head predicate `main`. These are Rule 2 and Rule 3. For Rule 2, the algorithm recursively calls $mapPred(\texttt{directSubclass}, \{\texttt{main}\}, P)$. This returns

$$LFP(\lambda x_{\texttt{directSubclass}} : \varnothing \cup [\texttt{cat subclass.tsv}]).$$

Since the lambda-expression does not contain the variable $x_{\texttt{directSubclass}}$, this is equivalent to `[cat subclass.tsv]`.

For Rule 3, we call $mapPred(\texttt{directSubclass}, \{\texttt{main}\}, P)$, which returns [cat subclass.tsv] just like before. Then we call $mapPred(\texttt{main}, \{\texttt{main}\}, P)$, which returns $x_{\texttt{main}}$, because main is in the cache. Thus, Rule 3 yields

$$\pi_{1,4}(\texttt{[cat subclass.tsv]} \bowtie_{2=1} x_{\texttt{main}})).$$

Finally, the algorithm constructs the result

$$LFP(\lambda x_{\texttt{main}} : \texttt{[cat subclass.tsv]} \cup \pi_{1,4}(\texttt{[cat subclass.tsv]} \bowtie_{2=1} x_{\texttt{main}}))$$

6.5.3 Producing Bash Commands

The previous step has translated the input Datalog program to a relational algebra expression. Now, we translate this expression to a Bash command by the function b, which is defined as follows:

$b([c]) = c$
: An expression of the form $[c]$ is already a Bash command, and hence we can return directly c.

$b(e_1 \cup \ldots \cup e_n)$
: To remove possible duplicates, we translate a union into

```
sort -u <(b(e1)) ... <(b(en))
```

$b(e_1 \bowtie_{x=y} e_2)$
: A join of two expressions e_1 and e_2 on a single variable at position x and y, respectively, gives rise to the command

```
join -t$'\t'  -1x -2y \
    <(sort -t$'\t'  -kx <(b(e1))) \
    <(sort -t$'\t'  -ky <(b(e2)))
```

This command sorts the byte streams of $b(e_1)$ and $b(e_2)$, and then joins them on the common column.

$b(e_1 \bowtie_{x=y,\ldots} e_2)$
: The Bash join command can perform the join on only one column. If we want to join on several columns, we have to add a new column to each of the byte streams. This new column concatenates the join columns into a single column. This can be achieved with the following AWK program, which we run on both $b(e_1)$ and $b(e_2)$:

```
{ print $0 FS $j_1 s $j_2 s ... s $j_n }
```

Here, the indices $j_1, \ldots, j_n$ are the positions of the join columns in the input byte stream, and s is the separation character (see Section 6.4). Once we have done this with both byte streams, we can join them on this new column in the same way as described above for simple joins. This join also removes the additional column.

$b(e_1 \triangleright_x e_2)$

Just as a regular join, an anti-join becomes a `join` command. We use the parameter `-v1`, so that the command outputs only those tuples emerging from e_1 than cannot be joined with those from e_2. We deal with anti-joins on multiple columns in the same way as with multi-column joins.

$b(\pi_{i_1,\ldots i_n}(e))$

A projection becomes the following AWK program, which we run on $b(e)$:

```
{ print $i_1 FS ... FS $i_n }
```

$b(\pi_{i:a}(e))$

A constant introduction becomes the following AWK program, which we run on $b(e)$:

```
{ print $1 FS ... $(i-1) FS a FS $i FS ... $n }
```

$b(\sigma_{i=v}(e))$

A selection node gives rise to the following AWK program, which we run on $b(e)$:

```
$i == "v" { print $0 }
```

This command can be generalized easily to a selection on several columns.

Note that several of these translations produce process substitutions. In such cases, Bash starts the parent process and the inner process in parallel. The parent process will block while it cannot read from the inner processes. Thus, only the innermost processes run in the beginning. Every process is run asynchronously as soon as input and CPU capacity is available. Thus, our Bash program is not subject to the forced synchronization that appears in Map-Reduce systems.

6.5.4 Recursion

We have just defined the function b that translates a relational algebra expression to a Bash command. We will now see how to define b for the case of recursion. A node $LFP(f)$ becomes

```
echo -n > delta.tmp;  echo -n > full.tmp
while
    sort b(f(delta.tmp)) | comm -23 - full.tmp > new.tmp;
    mv  new.tmp  delta.tmp;
    sort -u -m -o full.tmp  full.tmp <(sort delta.tmp);
    [ -s delta.tmp ];
do continue; done
cat full.tmp
```

This code uses 3 temporary files to compute the least fix point of f: `full.tmp` contains all facts inferred until the current iteration. `delta.tmp` contains newly added facts of an iteration. `new.tmp` is used as swap file.

The code first creates `delta.tmp` and `full.tmp` as empty files. It then runs f on the delta file. The `comm` command compares the sorted outcome of f to the (initially empty) file `full.tmp`, and writes the new lines to the file `new.tmp`. This file is then renamed to `delta.tmp`. This procedure updates the file `delta.tmp` to contain the newly added facts. The `comm` command cannot write directly to `delta.tmp`, because this file also serves as input to the command produced by $b(f($`delta.tmp`$))$.

The following `sort` command merges the new lines into `full.tmp`, and writes the output to `full.tmp` (the `sort` command can write to a file that also serves as input). Now, all facts generated in this iteration have been added to `full.tmp`. The `[...]` part of the code lets the loop run while the file `delta.tmp` is not empty, i.e., while new lines are still being added. If no new lines were added, the code quits the loop, and prints all facts. Note that, due to the monotonicity of our relational algebra operators, and due to the stratification of our programs, we can afford to run f only on the newly added lines.

Our method generates such a loop for each recursion. Since such loops can run in several processes in parallel, we generate different temporary file names for each recursion. We also take care to delete the temporary files after the Bash program finishes.

6.5.5 Materialization

Materialization nodes. To avoid re-computing a relational algebra expression that has already been computed we materialize some intermediate computations. For this purpose, we introduce a new type of operator to the algebra, the materialization node. A materialization node $\boxdot(m,(\lambda y : p_{m\rightarrow y}))$ has two sub-plans: m is the plan that is used multiple times, and that we will materialize. The function $(\lambda y : p_{m\rightarrow y})$ is the main plan, and takes the materialized plan as parameter. The plan $p_{m\rightarrow y}$ is the original plan p with all occurrences of m replaced by the variable y.

A sub-plan m will be materialized if one of the following applies:

— m is used by different nodes of the query plan, i.e., m has multiple parents in the relational algebra expression DAG.

— there exists a node $LFP(\lambda x_q : f)$ where f contains m, but m does not contain x_q.

We proceed in two phases: We first check whether to materialize a sub-plan, and then decide where to materialize it. First, we discuss the simple case where no LFP node is the ancestor of another LFP node. We start with the deepest node m that fulfills one of these conditions. If m does not contain a node x of an LFP node, then and we replace the query plan p by ⧉$(m, (\lambda y : p_{m \to y})$. If m contains a node x_q of an $LFP(\lambda x_q : q)$, and occurs at least twice in q, we materialize m within the LFP node: $LFP(\lambda x_q :$ ⧉$(m, (\lambda y : q_{m \to y})))$. We repeat this procedure until no more nodes can be materialized. Finally, if m contains variables x of several LFP nodes, these LFP nodes are nested. We materialize m directly below the deepest LFP node whose variable x is contained in m.

Translation. A node ⧉$(m,(\lambda y : p))$ gives rise to the following translation to Bash commands:

```
mkfifo lock_t
(
    b(m) > t
    mv lock_t done_t
    cat done_t &
    exec 3> done_t
    exec 3>&-
) &
b_{y→t}(p)
rm t
```

Here, t is a temporary file name. Each materialization node uses its own temporary file t, because ⧉-nodes can be nested. Our translation allows us to execute several materialization operations in parallel. The function b is the Bash translation function defined in Section 6.5.3. Commands that use t have to wait until $b(m)$ finishes. We ensure this by making these commands read from the named pipe `lock_t`. Since this pipe contains no data, the commands block. When $b(m)$ finishes, the two `exec` commands close the named pipe, thus unblocking the commands that need t. There can be a rare race condition: $b(m)$ may finish before any process that listens on the pipe was started. In that case, the two `exec` commands try to close a pipe that has no listeners. In such cases, the `exec` command would block. We solve this problem by reading from the pipe with a `cat` command that runs in the background. This way, the pipe has at least one listener, and the `exec` commands close the pipe. This, however, brings a second problem: If the processes that listen on the pipe were still not started, they would try to listen to a

closed pipe. To avoid this problem, we rename the pipe from lock_t to done_t. Such a renaming does not affect any processes that already listen on the pipe, but it prevents any new processes from listening on the pipe under the old name.

Finally, we actually use the materialized plan. The function $b_{y \to t}$ extends b as follows: $b_{y \to t}(y)$ generates the bash code `cat t`; and all plan nodes p_i that have a child y generate the bash code

```
cat lock_t 2> /dev/null
b(p_i)
```

As explained above, the `cat` command blocks the execution until t is materialized. The part "`2> /dev/null`" removes the error message in case `cat` is executed when the pipe was already renamed.

6.5.6 Optimization

Algebra Optimizations. We apply the usual optimizations on our relational algebra expressions: we push selection nodes as close to the source as possible; we merge unions; we merge projects; we apply a simple join re-ordering. Additionally, we remove LFP when there is no recursion; and we extract from an LFP node the non-recursive part of the inner plan (so that it is computed only once at the beginning of the fixed point computation).

Removing superfluous calls. In the Datalog program, a recursive call of a predicate occurs, if the predicate takes itself as input (eventually with mediating rules). In the algebra, a recursive call corresponds to the x of a LFP node. We remove recursive calls when they are not contributing new output. We call these calls "superfluous". The following example illustrates the concept of superfluous calls.

> *Example (Superfluous calls):* We want to obtain a list of professors. The rules of the knowledge base are as follows:
>
> ```
> Professor(X) :- Person(X), teachesCourse(X,Y).
> Professor(X) :- successorOf(X,Y), Professor(Y).
> Person(X) :- Employee(X).
> Person(X) :- Professor(X).
> ```
>
> The predicates `teachesCourse`, `successorOf`, and `Employee` represent input relations. The first and last rule combined are equivalent to the rule

```
Professor(X) :- Professor(X), teachesCourse(X,Y)}.
```

It is obvious that this rule does not add any new professors to the output. If X is not a professor, then the first atom of the body `Professor(X)` is not part of the model, and so the rule will not apply. If X is a professor, then the first atom is part of the model. However, in that case, applying the rule will infer `Professor(X)`, which is already in the model, so nothing changes.

If a rule contains the head atom in the body (with the same variables in the same order), we can safely ignore the rule. We will apply this principle on the algebra plan. The algebra plan of this example is shown in Figure 6.4. Only the recursive call x in the left subtree is superfluous.

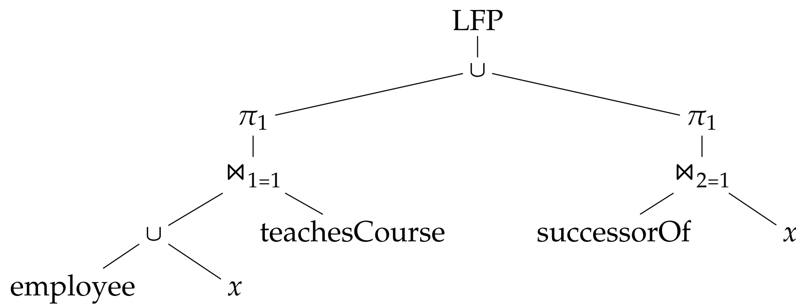


Figure 6.4 – Algebra plan for removing superfluous calls example

First, we determine whether a recursive call x is superfluous, by tracing its columns until the LFP node. If they arrive completely, and in order, at the LFP node, all output computed in previous iterations would just pass trough the path from x to the LFP node.

In more detail, we detect superfluous recursive calls of a $LFP(\lambda x : f)$ node as follows. First, we define a function that returns all paths from an algebra tree node to its leaves, including the position of the arguments:

- $paths(t) = \{t\}$, if t is an input table
- $paths(\square(t)) = \{\square/ p \mid p \in paths(t)\}$, where $\square$ is a relational algebra operator of the form $\sigma_{i=\ldots}$, or $\pi_{i\ldots}$
- $paths(\square(t_1, t_2)) = \bigcup_i \{\square\ i/ p \mid p \in paths(t_i)\}$, where $\square$ is an operator of the form $\bowtie_{\ldots=\ldots}$, $\triangleright_{\ldots=\ldots}$, or $\cup$
- $paths(LFP(\lambda x : f) = \{LFP/p \mid p \in paths(f)\}$

Example (Paths): We will list here all paths of the child node of the least fixed point in Figure 6.4: $\{ \cup 1/\pi_1/\bowtie_{1=1} 1/\cup 1/\text{employee}, \quad \cup 1/\pi_1/\bowtie_{1=1} 1/\cup 2/x, \quad \cup 1/\pi_1/\bowtie_{1=1} 2/\text{teachesCourse}, \quad \cup 2/\pi_1/\bowtie_{2=1} 1/\text{successorOf}, \quad \cup 2/\pi_1/\bowtie_{2=1} 2/x \}$

Next, we define how the columns of a variable x of an $LFP(\lambda x : f)$ node propagate through the relational algebra expression. Let z be any constant, representing a column with an unknown value, and let $l = arity(LFP(\lambda x : f))$. The function $columns$ maps a path to a word over the alphabet $\{z, c_1, \ldots, c_l\}$. The character c_i represents the i-th column of x. Let n be the arity of the first relational operator of the argument of $columns$, and let $z^n = z \ldots z$, the word that repeats the character z n-times. The function $columns$ is defined recursively as follows:

— $columns(x) = c_1 \ldots c_l$
— $columns(\sigma_{i=\ldots}/p) = columns(p)$
— $columns(\pi_{i_1,\ldots,i_k}/p) = w_{i_1} \ldots w_{i_k}$, where $w_1 \ldots w_j = columns(p)$
— $columns(\pi_{i:a}/p) = w_1 \ldots w_{i-1}\, z\, w_i \ldots w_j$, where $w_1 \ldots w_j = columns(p)$
— $columns(\bowtie_{\ldots=\ldots} 1/p) = columns(p) \circ z^{n-j}$
— $columns(\bowtie_{\ldots=\ldots} 2/p) = z^{n-j} \circ columns(p)$
— $columns(\triangleright_{\ldots} 1/p) = columns(p)$
— $columns(\cup\, i/p) = columns(p)$, for $i = 1$ and $i = 2$
— otherwise, $columns(p) = z^n$

Example (Columns): We apply the function $columns$ to the recursive calls x in Figure 6.4: In our example, $columns(x) = c_1$, as the query outputs only a single column. For the left, $columns(\cup 1/\pi_1/\bowtie_{1=1} 1/\cup 2/x) = c_1$, so the variables of the left recursive call are passed through the tree. For the right, $columns(\cup 2/\pi_1/\bowtie_{2=1} 2/x \}) = z$, which means that the column of x did not arrive at the LFP node.

The superfluous variables x of $LFP(\lambda x : f)$ are the paths from the LFP node to a x node whose output columns are all columns of x in the right order:

$$superfluous_calls = \{p \mid p = p_1 \ldots p_n \in paths(f) \wedge p_n = x \wedge columns(p) = columns(x)\}$$

After detecting superfluous calls, we replace them with an empty table, and apply the usual optimizations.

Example (Superfluous calls): The only path with $columns(p) = columns(x)$ is $\cup 1/\pi_1/\bowtie_{1=1} 1/\cup 2/x$. It is the only superfluous call. We therefore simplify the plan in Figure 6.4 by replacing the left x with the empty table $\varnothing$. Figure 6.5 shows the simplified result.

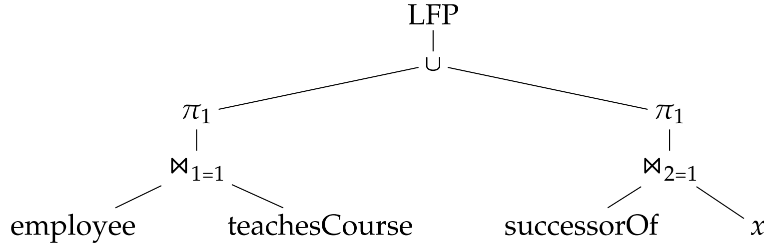


Figure 6.5 – Algebra plan after removing superfluous calls

Semi-naive evaluation. We also optimize the fix point computation by using the same idea as the semi-naive Datalog evaluation [Abiteboul et al., 1995]: For every expression of the form $p = LFP(\lambda x_p : f)$, we introduce a placeholder δ_p that represents the facts created by the last iteration of the fix-point computation. Then we apply the operation I_δ on f, which is defined as follows:

- $I_\delta(x) = \delta_p$
- $I_\delta(R_1 \ltimes_s R_2) = (I_\delta(R_1) \ltimes_s R_2) \cup (R_1 \ltimes_s I_\delta(R_2))$ if both R_1 and R_2 contain occurrences of x.
- in all other cases $I_\delta(\phi)$, we apply I_δ recursively on the children of ϕ.

It is easy to see that $LFP(\lambda x : f) = LFP(\lambda x_p : I_\delta(f))$.

If f represents a simple transitive closure, i.e.,

$$f = \pi_{1,4}(x_p \ltimes_{2=1} x_p) \text{ or } f = \pi_{2,3}(x_p \ltimes_{1=2} x_p),$$

we avoid the second join, and use the plan

$$LFP(\lambda x_p : \pi_{1,4}(\delta_p \ltimes_{2=1} x_p)).$$

Join reordering. Reordering join operations can greatly reduce the cardinality of intermediate results. There is an extensive corpus of work about different techniques to this end. For now, we apply only a simple join-reordering, and reserve more advanced reordering algorithms for future work. Let n be the top-most join-node in our relational algebra expression. We do a depth-first search that stops at every non-join node, and collect them into a list L, from left to right. From L, we construct a new ordered list L' as follows: We choose node p of $L \setminus L'$ that has the most join conditions with the nodes in L'. If there are no common join conditions, we choose the node that has the most join conditions overall. We add p to L', and repeat the procedure until L' contains all nodes of L.

We select the first two elements of L', and join them with all join conditions that can be applied to their columns. We join every following element in L' to the right of the previous join with all applicable join conditions. Finally, we wrap the so constructed join tree into a projection node, in order to retrieve the same columns that the original join n would have produced. We apply this method recursively to all nodes in L'.

Parallel file scanning. Preliminary experiments showed that the first phase of query execution was IO bound. It may happen that several commands read simultaneously from the same file, selecting on different conditions, or projecting different columns. If all commands access the file at the same position, the operating system can buffer the relevant block of the file. However, since our commands are not synchronized, they usually read from different blocks of the file, which makes the access very slow. To mitigate this problem, we collect different AWK commands that select or project on the same file into a single AWK command. This command runs only once through the file, and writes out all selections and projections into several files, one for each original AWK command.

Post-processing. Our Bash program may nest several `sort` commands. This can happen, e.g., if a union is the object of a join. We detect such cases, and remove redundant `sort` commands. To make sure that the final output of our program contains only unique results, we run `sort -u` on the final output.

Astonishingly, `sort` and `join` use a different character order. This means that `join` with input from `sort` warns about unsorted input. To mitigate this problem, we add the following as a first line to our program:

```
export LC_ALL=C
```

This construction forces all commands to use the default language for input and output, and to sort bytewise. This ensures, in particular, that our command works with UTF-8 encoded files. It also improves the processing speed.

6.6 Experiments

To show the viability of our approach, we ran our method on several datasets, and compared it to several competitors. All experiments were run on a laptop with Ubuntu 16.04, an Intel Core i7-4610M 3.00 GHz CPU, 16 GB of memory, and 3.8 TB of disk space. We used GNU coreutils 8.25 for POSIX commands, and mawk 1.3.3 for AWK.

We emphasize that our goal is not to be faster than each and every system that currently exists. For this, the corpus of related work is simply too large (see Section 6.3). This

is also not the purpose of Bash Datalog. The purpose of Bash Datalog is to provide a preprocessing tool that runs without installing any additional software besides a Bash shell. This is an advantage that no competing approach offers. Our experiments then serve mainly to show that our approach is generally comparable in terms of speed and scalability with the state of the art.

6.6.1 Lehigh University Benchmark

Dataset. Our first dataset is the Lehigh University Benchmark (LUBM) [Guo et al., 2005]. LUBM is a standard dataset for Semantic Web repositories. It models universities, their employees, students, and courses. The dataset is parameterized by the number of universities, and hence its size can be varied. LUBM comes with 14 queries, which test a variety of different usage patterns. These queries are expressed in SPARQL. For our purposes, we translated the queries to Datalog.

Competitors. We compare our approach to the following competitors:

DLV[4] is a disjunctive logic programming system. It can handle Datalog queries out of the box [Leone et al., 2006].

RDFox[5] is an in-memory RDF triple store that supports shared memory parallel Datalog reasoning [Motik et al., 2014].

Jena[6] is an open-source RDF triple store written in Java. It can execute SPARQL queries [Carroll et al., 2004]. We used the TDB implementation of Jena.

Jena+HDT[7] is a combination of Jena with the binary format HDT for triple data [Fernández et al., 2013].

Stardog[8] is commercial knowledge graph platform that allows answering SPARQL queries on RDF data.

Virtuoso[9] is a commercial platform that also allows answering SPARQL queries on RDF data [Erling and Mikhailov, 2009].

Postgres[10] is a relational database system that is developed as open-source. It supports SQL queries.

4. `http://www.dlvsystem.com/dlv/`, v. Dec 17 2012
5. `https://www.cs.ox.ac.uk/isg/tools/RDFox/`
6. `https://jena.apache.org/`, v. 3.4.0
7. `http://www.rdfhdt.org/`, v. 1.1.2
8. `https://www.stardog.com/`, v. 5.2.0
9. `https://virtuoso.openlinksw.com/`, v. 7.2.5 OS Edition
10. `https://www.postgresql.org/`, v. 10.1
11. `https://github.com/HBPMedical/PostgresRAW/tree/6ae475`, v. 9.6.5

Query	Bash	DLV	RDFox	Jena + TDB	Jena + HDT	Stardog	Virtuoso	Postgres* – I	Postgres* + I	NoDB*	MonetDB* – I	MonetDB* + I	RDF-Slice*
1	**0.7**	9.6	2.2	25.7	26.4	12.8	11.7	4.8	27.5	>600	1.8	2.6	12.6
2	**1.3**	9.3	2.2	281.3	>600	13.6	11.8	–	–	–	–	–	–
3	**0.9**	9.2	2.2	26.7	27.0	12.7	11.5	7.8	30.5	292.9	1.9	2.7	–
4	**1.9**	9.3	2.2	>600	>600	13.2	12.2	14.7	37.4	>600	2.3	3.1	–
5	**1.4**	9.3	2.2	>600	>600	12.9	–	28.9	51.6	–	2.2	3.0	–
6	**1.9**	9.4	2.4	>600	>600	17.6	–	21.2	43.9	–	3.9	4.7	–
7	2.4	9.5	**2.2**	>600	>600	13.4	–	21.6	44.3	>600	3.0	3.9	–
8	2.5	9.3	**2.3**	>600	>600	15.3	–	–	–	–	–	–	–
9	3.1	9.4	**2.3**	>600	>600	13.4	–	71.1	93.8	>600	25.5	26.8	–
10	**2.0**	9.3	2.2	>600	>600	13.5	–	23.0	45.7	>600	5.8	7.1	–
11	**0.9**	9.3	2.2	25.3	35.7	13.0	11.8	–	–	–	–	–	–
12	**1.4**	9.2	2.2	>600	>600	13.1	–	–	–	–	–	–	–
13	**1.4**	9.2	2.2	>600	>600	12.9	–	8.4	31.1	>600	4.3	5.4	–
14	**0.8**	9.5	2.3	34.5	24.8	13.5	12.0	4.8	27.5	19.1	1.9	2.7	3.7
of which loading:				16.8	7.4	11.0	5.9	4.4	24.3		1.7	2.5	

Table 6.1 – Runtime for the LUBM queries with 10 universities (155 MB), in seconds.
* = no support for querying with a TBox. We folded the TBox into the query.
+/–I = with/without indexes. A dash means that the query is not supported.

NoDB[11] is an extension of Postgres, which can execute SQL queries directly on TAB-separated files.

MonetDB[12] is an open source column-oriented database management system, which also supports SQL queries [Boncz et al., 2008].

RDFSlice[13] is a tool for filtering RDF triples, implementing the Extract-Transform-Load paradigm for RDF data [Marx et al., 2017].

For the database systems (Postgres, NoDB, and MonetDB), we translated the queries to SQL. For this purpose, we used the relational algebra expression computed in Algorithm 12. Not all systems support all types of queries. MonetDB does not support recursive SQL queries. Postgres supports only certain types of recursive queries[14]. The same applies to NoDB. Virtuoso currently does not support intersections. RDFSlice aims at the slightly different problem of RDF-Slicing. It supports only a specific type of join. Also, it does not support recursion.

We ran every competitor on all queries that it supports, and averaged the runtime over 3 runs. Since most systems finished in a matter of seconds, we aborted systems that took longer than 10 minutes. The databases were run with and without indexing. NoDB comes with its own adaptive indexing mechanism that cannot be switched off.

LUBM10. Table 6.1 shows the runtimes of all queries for the different systems on LUBM with 10 universities. The runtimes include the loading and indexing times. For systems where we could determine these times explicitly, we noted them in the last row of the table. Among the 4 triple stores (Jena+TDB, Jena+HDT, Stardog, and Virtuoso), only Stardog can finish on all queries in less than 10 minutes. Jena is mostly too slow, no matter with which back-end. Virtuoso performs slightly faster than Stardog, but it cannot deal with all queries. The triple stores are generally slower than our 5 database competitors (Postgres, NoDB, and MonetDB – with and without indexes). Among these, we find that MonetDB is much faster than Postgres and NoDB. Postgres and MonetDB are fastest without indexes, which is to be expected with such small datasets.

The best performing systems overall are clearly the 3 Datalog systems (Bash Datalog, DLV, and RDFox). Not only can they answer all queries, but they are generally also faster than the other systems. Among the three, DLV is the slowest. RDFox shines with a very short and nearly constant time for all queries. We suspect that this time is given

12. `https://www.monetdb.org/`, v. Jul2017-SP3

13. `http://aksw.org/Projects/RDFSlice.html`, `https://bitbucket.org/emarx/rdfslice/src`, v. 2016-12-01

14. For example, Datalog programs of the following shape cannot be translated into an SQL query supported by Postgres:

```
level(X,Y) :- level(X,Z), level(Y,Z).
level(Y,X) :- level(X,Z), level(Y,Z).
```

by the loading time of the data, and that it dominates the answer computation time. Nevertheless, Bash Datalog is faster than RDFox on nearly all queries on LUBM 10.

LUBM100 to LUBM1000. Based on the previous experiment, we chose the fastest systems in each group as competitors: RDFox for the Datalog systems, Stardog and Virtuoso for the triple stores, and MonetDB with and without indexes for the databases. We then increased the number of universities in our LUBM dataset from 10 to 100, 200, 500, and 1000. Table 6.2 and 6.3 shows the sizes of the datasets and the runtimes of the systems. Across all datasets, our system performs best on more than half of the queries. The only system that can achieve a similar performance is RDFox. As before, RDFox always needs just a constant time to answer a query, because it loads the dataset into main memory. This makes the system very fast. However, this technique does not work if the dataset is too large, as we shall see next.

6.6.2 Reachability

Datasets. Our next datasets are graph datasets. We used the LiveJournal and com-orkut graphs from [Leskovec and Krevl, 2014], and the friendster graph [Kunegis, 2013]. These datasets represent the graph structure of online social networks. They allow us to test the performance of our algorithm on real world data. Table 6.4 shows the number of nodes and edges of these datasets.

As our competitors, we chose again RDFox, Stardog, and Virtuoso. We could not use MonetDB, because the reachability query is recursive. As an additional competitor, we chose BigDatalog [Shkapsky et al., 2016]. BigDatalog is a distributed Datalog implementation running on Apache Spark. BigDatalog was already run on the same LiveJournal and com-orkut graphs in the original paper [Shkapsky et al., 2016].

Query. For all of these datasets, we used a single query: We asked for the set of nodes that can be reached from a given node *id*. We used the following Datalog program to this end, adapted from [Shkapsky et al., 2016]:

```
reach(Y) :- arc(id, Y).
reach(Y) :- reach(X), arc(X, Y).
```

In order to avoid that RDFox materializes the entire transitive closure, we modified the program as follows for RDFox:

```
reach(id, Y) :- arc(id, Y).
```

	LUBM 100 (1.5 GB)								LUBM 200 (3.1 GB)							
Query	Bash	DLV	RDFox	Stardog	Virtuoso	MonetDB (no indices)	MonetDB (indices)	RDFSlice	Bash	DLV	RDFox	Stardog	Virtuoso	MonetDB (no indices)	MonetDB (indices)	RDFSlice
1	**8**	111	25	59	119	15	17	100	**9**	227	50	110	302	31	36	196
2	**11**	108	25	61	120				**21**	222	51	112	302			
3	**7**	108	25	59	119	16	18		**14**	223	50	110	302	32	37	
4	**17**	109	24	60	119	21	23		**37**	224	50	110	303	42	47	
5	**12**	108	25	59		19	21		**24**	224	50	110		59	64	
6	**17**	110	26	93		30	32		**41**	944	53	173		76	82	
7	**24**	110	25	61		28	31		**44**	227	51	112		74	80	
8	**22**	109	25	61					**45**	225	50	113				
9	28	111	**26**	62		34	36		61	227	**52**	113		138	138	
10	**18**	108	25	60		27	30		**44**	224	50	111		72	79	
11	**6**	108	24	60	119				**11**	222	50	110	302			
12	**10**	108	24	59					**21**	222	50	110				
13	**12**	107	25	59		19	21		**24**	222	50	110		58	64	
14	**5**	108	26	63	122	15	18	33	**11**	601	52	118	309	31	36	65
of which load:				57	118	14	16					108	301	28	33	

Table 6.2 – Runtime for the LUBM queries, in seconds.

	LUBM 500 (7.8 GB)							LUBM 1000 (16 GB)					
Query	Bash	RDFox	Stardog	Virtuoso	MonetDB (no indices)	MonetDB (indices)	RDFSlice	Bash	RDFox	Stardog	MonetDB (no indices)	MonetDB (indices)	RDFSlice
1	**27**	131	582	1577	83	97	229	**75**	273	1955	185	210	1042
2	**53**	132	683	1580				**118**	278	2030			
3	**35**	131	609	1578	88	101		**89**	276	1955	186	217	
4	**95**	129	583	1579	118	131		307	**273**	1962	522	471	
5	**62**	131	498		290	364		**168**	278	1956	894	793	
6	**93**	137	1011		866	797		354	**287**	2361	2066	1934	
7	**122**	134	673		898	753		544	**279**	2005	1809	2016	
8	151	**132**	768					447	**274**	1967			
9	250	**136**	749		2669	3064		712	**283**	2018	3275	3090	
10	**95**	132	678		587	491		334	**277**	1959	1845	1834	
11	**28**	130	498	1576				**64**	273	1957			
12	**56**	130	682					**164**	273	1959			
13	**63**	132	669		312	287		**174**	277	1969	908	955	
14	**28**	136	787	1595	85	99	74	**63**	284	2069	181	217	334
of which load:			489	1575	72	92				1946	160	194	

Table 6.3 – Runtime for the LUBM queries, in seconds.

dataset	Nodes	Edges
LiveJournal [Leskovec and Krevl, 2014]	4.8 M	69 M
orkut [Leskovec and Krevl, 2014]	3.1 M	117 M
friendster [Kunegis, 2013]	68 M	2 586 M

Table 6.4 – Statistics for the reachability datasets.

dataset	Bash	RDFox	BigDatalog	Stardog	Virtuoso
LiveJournal	117	**70**	532	941	>1500
orkut	225	**121**	1838	1123	>3000
friendster	**16306**	OOM	OOS	>36000	

Table 6.5 – Runtime for the reachability query, in seconds (OOM=Out of memory; OOS=Out of space).

```
reach(id, Y) :- reach(id, X), arc(X, Y).
```

For the experiment, we chose 3 random nodes (and thus generated 3 queries) for LiveJournal and com-orkut. We chose one random node for Friendster.

Results. Table 6.5 shows the runtime for each system (averaged over the 3 queries for LiveJournal and com-orkut). Virtuoso was the slowest system, and we aborted it after 25 min and 50 min, respectively. We did not run it on the Friendster dataset, because Friendster is 20 times larger than the other two datasets. Stardog performs better. Still, we had to abort it after 10 hours on the Friendster dataset. BigDatalog performs well, but fails with an out of space error on the Friendster dataset. The fastest system is RDFox. This is because it can load the entire data into memory. This approach, however, fails with the Friendster dataset. It does not fit into memory, and RDFox is unable to run. Bash Datalog runs 50% slower than RDFox. In return, it is the only system that can finish in reasonable time on the Friendster dataset (4:30h). We believe that this is because Bash Datalog can rely on the highly optimized implementations of the Bash commands, which can deal with large files even if they cannot be loaded into memory.

6.6.3 YAGO and Wikidata

Datasets. Our final series of experiments tests our system on knowledge bases. For this purpose, we used YAGO 3.1, and Wikidata [Vrandecic and Krötzsch, 2014]. The YAGO data comes in 3 different files, one with the 12 M facts (814 MB), one with the taxonomy

with 1.8 M facts (154 MB), and a last one with the 24 M type relations (1.6 GB in size). The Wikidata simple dataset contains 2.1 billion triples and has an uncompressed size of 267 GB.

Queries. We designed 4 queries that are typical for such datasets (Table 6.6). Table 6.7 shows the TBox that we used. These queries and the TBox are slightly adapted to work with the different schema of the two knowledge bases. Query 1 asks for all subclasses of the class <Person>. Query 2 asks for the parents of Louis XIV, and Query 3 asks recursively for the ancestors of Louis XIV. Query 4 asks for all people born in a place in Andorra. These queries are not difficult. The difficulty comes from the fact that the data is so large.

```
query1(X) :- subClassOf(X, "<Person>") .
query1(X) :- subClassOf(X, Y), query1(Y) .
query2(X) :- hasParent(X, "<Louis_XIV>") .
query3(X) :- hasAncestor(X, "<Louis_XIV>") .
query4(X) :- hasBirthPlace(X, Y),
             isLocatedIn(Y, "<Andorra>") .
```

Figure 6.6 – Knowledge Base queries

```
hasParent(X,Y) :- hasChild(Y,X) .
hasAncestor(X,Y) :- hasParent(X,Y) .
hasAncestor(X,Z) :- hasAncestor(X,Y),
                    hasParent(Y,Z) .
isLocatedIn(X,Y) :- containsLocation(Y,X) .
containsLocation(X,Y) :- isLocatedIn(Y,X) .
isLocatedIn(X,Y) :- isLocatedIn(X,Y),
                    isLocatedIn(Y,Z) .
```

Figure 6.7 – Knowledge Base rules

Results. Table 6.6 shows the results of RDFox and our system on both datasets. On YAGO, RDFox is much slower than our system, because it needs to instantiate all rules in order to answer queries. On Wikidata, the data does not fit into main memory, and hence RDFox cannot run at all. Our system, in contrast, scales effortlessly to the larger sizes of the data.

One may think that a database system, such as Postgres, may be better adapted for such large datasets. This is, however, not the case. Postgres took 104 seconds to load

	YAGO		Wikidata	
query	Bash	RDFox	Bash	RDFox
1	**8**	483	**2259**	OOM
2	**5**	483	**2254**	OOM
3	**293**	483	**10171**	OOM
4	**5**	481	**2270**	OOM

Table 6.6 – Runtime for the Wikidata/YAGO benchmark in seconds. (OOM = out of memory error)

the YAGO dataset, and 190 seconds to build the indexes. In this time, our system has already answered nearly all the queries.

Discussion. All of our experiments evaluate only the setting that we consider in this chapter, namely the setting where the user wants to execute a single query in order to preprocess the data. These are the cases that our system is designed for. We also want to emphasize again that our goal is not to be faster than all existing systems. This is not our point. Our point is that Bash Datalog can preprocess tabular data without the need to install any particular software. In addition, our approach is competitive in both speed and scalability to the state of the art.

6.7 Web Interface

Our system can be used online at `https://www.thomasrebele.org/projects/bashlog`. Figure 6.8 on the following page shows a screenshot. Our interface provides three modes: a Datalog mode, a SPARQL mode, and an API.

Datalog mode. The user can enter her Datalog program in a text box. After she clicks on "Convert to Bash script", the Datalog program is transmitted to a server, which translates it to a Bash script. The Bash script then appears in the second text box. The user can copy and paste the script into a terminal and execute it. To help the user get started, we provide an example dataset based on YAGO, together with example queries.

> *Example (Datalog mode):* Let us, e.g., walk through the query shown in Figure 6.9 on page 123. It extracts all people that have an ancestor born in Italy from the knowledge base. Line 1 specifies the only command rule, which is responsible for

https://www.thomasrebele.org/projects/bashlog

Bash Datalog
Answering Datalog Queries with Unix Shell Commands

This project translates datalog programs to Unix shell scripts. It can be used to preprocess large tabular datasets. It has a datalog mode, a SPARQL/OWL mode, and an API. We describe how it works in the technical report.

Datalog program

```
type("albert", "person").
type("marie", "person").
people(X) :- type(X, "person").
```

Convert to bash script | Download script

Bash script

```
#!/bin/bash
###########################################################
##########
# This script was generated by bashlog
# For more information, visit
thomasrebele.org/projects/bashlog
###########################################################
##########

export LC_ALL=C
mkdir -p tmp
rm -f tmp/*
if type mawk > /dev/null; then awk="mawk"; else
awk="awk"; fi
```

How to try it:

1. Copy one of the examples into the "Datalog program" textbox
2. Click on the `Convert to bash script` button
3. Copy the content of the "Bash script" textbox into a file named `query.sh` in the folder with the .tsv files (or click on `Download script`)

Examples:

You can try the examples on this dataset (source). Unpack the dataset archive in a new folder (if unzip is installed: `unzip sample.zip`).

- Find people that died in the city where they were born

```
facts(_, S, P, O) :~ cat *.tsv
main(X) :-
   facts(_, X, "<wasBornIn>", Y),
   facts(_, X, "<diedIn>", Y).
```

- Living people

```
facts(_, S, P, O) :~ cat *.tsv
born(X) :- facts(_, X, "<wasBornIn>", Y).
born(X) :- facts(_, X, "<wasBornOnDate>", Y).
dead(X) :- facts(_, X, "<diedIn>", Y).
dead(X) :- facts(_, X, "<diedOnDate>", Y).

main(X) :- born(X), not dead(X).
```

(you can find deceased people by removing `not`)

- All people

```
facts(_, S, P, O) :~ cat *.tsv

type(X, Y) :- facts(_, X, "rdf:type", Y).
subclass(X, Y) :- facts(_, X, "rdfs:subclassOf", Y).
type(X, Z) :- type(X, Y), subclass(Y, Z).

main(X) :- type(X, "<wordnet_person_100007846>").
```

Figure 6.8 – Screenshot of the web interface

reading the data from the disk. Lines 3-5 are shorthand predicates for the relations that we want to use in the query. Lines 7-9 define the TBox, which states that `hasAncestor` can be computed as a transitive closure. Lines 11-13 are the actual query. Once the user clicks on "Generate", our interface translates the query into the Bash script shown in Appendix A.1 on page 143.

SPARQL/OWL mode. This mode allows preprocessing knowledge bases using Semantic Web standards. It takes a SPARQL query, and a TBox in the OWL 2 RL format. For the input of the ABox, we currently support RDF data in the form of N-Triples.

Example (SPARQL/OWL mode): Let us consider the same example as in Figure 6.9. Let us assume this time that the user formulates the query not in Datalog, but in SPARQL. Figure 6.11 shows the query. To add the semantics to the hasAncestor predicate, the user specifies the transitivity of the predicate in OWL (shown in Figure 6.10). When our system receives this input, it translates it to a Datalog program, and proceeds as before. The generated Bash script is shown in Appendix A.2 on page 145.

```
1  fact(Id, S, P, O) :~ cat *.tsv
2
3  hasChild(X, Y) :- fact(_, X, "<hasChild>", Y).
4  wasBornIn(X,Y) :- fact(_, X, "<wasBornIn>", Y).
5  isLocatedIn(X,Y) :- fact(_, X, "<isLocatedIn>", Y).
6
7  hasParent(X,Y) :- hasChild(Y,X).
8  hasAncestor(X,Y) :- hasParent(X,Y).
9  hasAncestor(X,Z) :- hasAncestor(X,Y), hasParent(Y,Z).
10
11 main(X,Y) :- hasAncestor(X,Y),
12              wasBornIn(Y,Z),
13              isLocatedIn(Z, "<Italy>").
```

Figure 6.9 – Example of a Datalog query that can be used with the Web interface. It finds all people in the YAGO knowledge base that have an ancestor that was born in Italy.

API. We also provide a way to use the Web interface from a command line interface, without opening a browser. Using the API requires a command which supports sending HTTP POST requests, such as `curl`. Unfortunately the POSIX standard does not include a command for HTTP requests. The command `curl` is widely spread and provides the right functionality for our purpose. Here we show the `curl` command for a SPARQL/OWL query:

```
curl --data-urlencode owl@ontology.owl \
```

```
1  @prefix owl: <http://www.w3.org/2002/07/owl#> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3  @prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4  @prefix kb: <http://yago-knowledge.org/resource/> .
5
6  kb:hasParent
7    owl:inverseOf kb:hasChild;
8    rdfs:subPropertyOf kb:hasAncestor.
9
10 kb:hasAncestor
11   rdf:type owl:TransitiveProperty.
```

Figure 6.10 – Example of an OWL TBox

```
PREFIX kb: <http://yago-knowledge.org/resource/>
SELECT ?X ?Y WHERE {
        ?X kb:hasAncestor ?Y .
        ?Y kb:wasBornIn ?Z .
        ?Z kb:isLocatedIn kb:Italy .
}
```

Figure 6.11 – Example of a SPARQL query

```
      --data-urlencode sparql@query.sparql \
      --data-urlencode nTriples=kb.ntriples \
      https://www.thomasrebele.org/projects/bashlog/api/sparql
```

The command expects the TBox and the query as files `ontology.owl`, and `query.sparql`, respectively. The file `kb.ntriples` is the path to the knowledge base in N-Triples format. The "=" specifies that only the path, but not the file content, will be sent to the server. The command sends the content of the files `ontology.owl` and `query.sparql`, and the string `kb.ntriples` to the server. The server executes our algorithm, and sends the Bash script in the response of the HTTP request. The result of the command is a Bash script that can be saved in a file and executed locally.

6.8 Summary

In this chapter, we have presented a method to compile Datalog programs into Unix Bash scripts. This allows executing Datalog queries on tabular datasets without installing any software. We show that our method is competitive in terms of speed with state-of-the-art systems. In particular, our method takes often less time to answer a query than a database system needs to load the data. Furthermore, our system can process datasets even if they do not fit in memory. This means that our approach is a good choice for preprocessing large tabular datasets.

6.9 Outlook

There are still ways to extend our system. For now, we have restricted ourselves to Datalog with negations, extended with rules to call external commands. It would be useful to enhance our Datalog dialect by adding numerical comparison operators, and aggregations (such as maximum), and counting. This would also allow us to support more SPARQL features.

Bash is a powerful scripting language, which makes it relatively easy to leverage the power of POSIX based operating systems. However, the language comes with its own limitations. Most notably, we could not implement a disk-based hash-join efficiently. AWK provides associative arrays. However, those are limited in size by the available main memory. Implementing an efficient hash function in AWK proved too difficult.

One possibility would thus be to compile the Datalog program not to Bash, but to a programming language, such as C. Every POSIX compatible operating system also provides a C compiler [15]. We tried this idea, and implemented a translator for generating C code. It works similarly to the approach described in Section 6.5.3. The C code uses threads instead of processes. We implemented a basic hash-join algorithm with an in-memory hash table backed by a file on disk. Preliminary experiments indicated a good performance. However, these experiments also showed that the approach using C was slower than the approach using Bash scripts. More work would be necessary to improve the C-based approach. We thus believe that our idea of compiling Datalog and Bash hits a sweet spot in terms of simplicity, scalability, and speed.

15. see `http://pubs.opengroup.org/onlinepubs/009695399/utilities/c99.html`

Chapter 7

Conclusion

7.1 Summary

This thesis has illustrated how to extend the YAGO knowledge base, and how to make it more useful for its users. It has worked on and contributed to the following research topics:

The YAGO Knowledge Base. In Chapter 3, we have described the YAGO knowledge base. YAGO is one of the first large-scale automatically constructed knowledge bases. Its approach of combining WordNet with Wikipedia has shown to be fruitful. YAGO has a high quality, i.e., it achieves a precision of over 95%. This quality distinguishes YAGO from similar approaches. Manual evaluations are conducted for every major release, in order to validate YAGO's quality. However, although YAGO contains millions of facts, it still covers only a small fraction of human knowledge.

Extending information about people in YAGO. In Chapter 4, we have addressed the problem of increasing the coverage of facts about people in YAGO. We showed how we can improve the information extraction algorithms by adding new heuristics. In this way, we extracted more birth and death dates, places of residence, and genders. Compared to the previous version of YAGO, we increased the number of people for which YAGO knows a place of residence by 63%. Similarly, we increased the number of people for which YAGO knows the gender by 46%.

We used the improved version of YAGO for historical case studies, among them the life expectancy of males and females over the centuries, and the age at first child birth. These studies are limited to the people that have their own Wikipedia page, i.e., mainly elites such as wealthy and famous people. Nevertheless, it might help to analyze history

from a different, more statistical point of view, and obtain insights about social changes over the span of centuries.

Working on the algorithms responsible for extracting dates from categories and infobox attributes, we often encountered the problem of adapting regular expressions. This problem was addressed in the next chapter.

Adding missing words to regular expressions. We showed how to automatically adapt a regular expression, so that it also matches a given set of words. For this, we need only a small set of positive examples as input. The repaired regex should achieve a better F1 measure, and a precision similar to, or better than the original regex. We provided two algorithms for that purpose. Both build on approximate regex matching to find the non-matched parts of the string. The first, simpler algorithm descends into the regex syntax tree, and inserts the non-matched parts as alternatives. The second, adaptive algorithm tracks several insertions at once, and additionally checks the quality of all modifications. In the experiments, both algorithms were competitive with state-of-the-art. The adaptive algorithm produced regexes with the best F1 measure of all systems, and output the shortest regexes. We also developed an online demonstration, where the user can see the actions of the simple algorithm step by step.

Answering queries with Unix shell. In this chapter, we developed a system for translating database queries into Bash scripts. Our approach allows a user to execute database queries on TSV files on a Unix system, without additional software. We support Datalog, or SPARQL queries with OWL constraints. In the case of SPARQL queries, we first translated them to Datalog. We transformed the Datalog program to a plan of relational algebra operators, extended with a least fixed point operator (LFP). We applied the usual relational algebra optimizations, and described how to optimize the plans that contain LFPs. Finally, we translated the plan to an optimized Bash script. In the experiments we showed that our approach is comparative in speed with state-of-the-art systems.

7.2 Outlook

Even though the work described in this thesis has improved the YAGO knowledge base, there are still many challenges to meet. This thesis concludes by outlining research directions that build on the here presented advances:

YAGO. Today's knowledge bases may be correct, but they are hardly ever complete [Razniewski et al., 2016]. Wikipedia contains much more data than YAGO. As we have seen with people in YAGO, simple heuristics, and algorithms, might increase

the coverage drastically. The first step is to find out, which entities or relations are complete or incomplete [Galárraga et al., 2017; Lajus and Suchanek, 2018].

Currently, YAGO contains only facts in form of triples. Future versions might also contain non-triple facts about narrative stories, text references, and even negated facts. Furthermore, meta facts that describe the completeness of the facts will become more important.

Digital Humanities. For future work, we aim to find more cases where our data can be of use. We are currently working on an interdisciplinary project that analyzes the importance of different groups of people in different locations over time. These groups can be based on social classes, such as politicians, scientists, or artists, gender, or social classes of their parents.

Repairing regexes. We aim to shorten the produced regexes further, by generalizing the components into character classes. We hope that this line of research can make regexes ever more useful to their users. Furthermore, we want to look into ways to apply our algorithm more automatically in information extraction pipelines.

Querying with Unix shell. We aim to explore extensions of this work, to make it even more useful. First, we would like to add support of numerical comparisons to our Datalog dialect. Second, we would like to add support of aggregations. This would allow the user to obtain the maximum, the minimum, or the sum of a list of values. Last, but not least, we plan to add support for redirecting intermediate results to external programs. In that way, our approach would act as a general Unix data modelling pipeline.

Declarative YAGO. The extraction algorithms that construct YAGO are quite complex. The source code consists of over 100 Java classes. Including its libraries, this number raises to over 200. This makes it challenging to find the cause of problems, and to fix them. Modeling the extraction pipeline in a declarative language, such as Datalog, might reduce the complexity of the source code substantially. Furthermore, it would allow adding more detailed provenance information with relative ease. It might even be possible to learn such an information extraction pipeline with machine learning techniques.

Résumé. Knowledge bases have become more and more important in everyday life. They enhance our experience with search engines, they help us organize information, and they might become a building block for the artificial intelligence systems of the future. However, knowledge bases are still incomplete, and – especially automatically constructed knowledge bases – may contain faulty data. In the future, knowledge bases

will become more complete and more accurate. This thesis has contributed to that endeavor by improving the extraction of facts about people from Wikipedia, by providing algorithms for repairing regular expressions, and by providing a tool for preprocessing and querying large amounts of data.

Bibliography

[Abiteboul and Hull, 1988] Abiteboul, S. and Hull, R. (1988). Data Functions, Datalog and Negation (Extended Abstract). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 143–153. ACM Press.

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[Adamou et al., 2016] (2016). *Workshop on Humanities in the Semantic Web*, volume 1608 of *CEUR Workshop Proceedings*.

[Agrawal, 1988] Agrawal, R. (1988). Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries. *IEEE Transactions on Software Engineering*, 14(7):879–885.

[Aho and Ullman, 1979] Aho, A. V. and Ullman, J. D. (1979). The Universality of Data Retrieval Languages. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 110–120. ACM Press.

[Alagiannis et al., 2012] Alagiannis, I., Borovica, R., Branco, M., Idreos, S., and Ailamaki, A. (2012). NoDB: efficient query execution on raw data files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 241–252. ACM.

[Auer et al., 2007] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. G. (2007). DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC + ASWC*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer.

[Baader and Nutt, 2003] Baader, F. and Nutt, W. (2003). Basic Description Logics. In *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95. Cambridge University Press.

[Babbar and Singh, 2010] Babbar, R. and Singh, N. (2010). Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text. In *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data, AND (in conjunction with CIKM)*, pages 43–50. ACM.

[Bartoli et al., 2012] Bartoli, A., Davanzo, G., Lorenzo, A. D., Mauri, M., Medvet, E., and Sorio, E. (2012). Automatic generation of regular expressions from examples with genetic programming. In *Genetic and Evolutionary Computation Conference, GECCO, Companion Material Proceedings*, pages 1477–1478. ACM.

[Bartoli et al., 2014] Bartoli, A., Davanzo, G., Lorenzo, A. D., Medvet, E., and Sorio, E. (2014). Automatic Synthesis of Regular Expressions from Examples. *IEEE Computer*, 47(12):72–80.

[Bartoli et al., 2016] Bartoli, A., Lorenzo, A. D., Medvet, E., and Tarlao, F. (2016). On the Automatic Construction of Regular Expressions from Examples (GP vs. Humans 1-0). In *Genetic and Evolutionary Computation Conference, GECCO, Companion Material Proceedings*, pages 155–156. ACM.

[Biega et al., 2013a] Biega, J., Kuzey, E., and Suchanek, F. M. (2013a). Inside YAGO2s: a transparent information extraction architecture. In *22nd International World Wide Web Conference, WWW*, pages 325–328. International World Wide Web Conferences Steering Committee / ACM.

[Biega et al., 2013b] Biega, J., Kuzey, E., and Suchanek, F. M. (2013b). Inside YAGO2s: a transparent information extraction architecture. In *22nd International World Wide Web Conference, WWW, Companion Volume*, pages 325–328. International World Wide Web Conferences Steering Committee / ACM.

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

[Boncz et al., 2008] Boncz, P. A., Kersten, M. L., and Manegold, S. (2008). Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77–85.

[Brauer et al., 2011] Brauer, F., Rieger, R., Mocan, A., and Barczynski, W. M. (2011). Enabling information extraction by inference of regular expressions from sample entities. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM*, pages 1285–1294. ACM.

[Brown et al., 2001] Brown, L. D., Cai, T. T., and DasGupta, A. (2001). Interval estimation for a binomial proportion. *Statistical science*, pages 101–117.

[Bu et al., 2012] Bu, Y., Borkar, V. R., Carey, M. J., Rosen, J., Polyzotis, N., Condie, T., Weimer, M., and Ramakrishnan, R. (2012). Scaling Datalog for Machine Learning on Big Data. *CoRR*, abs/1203.0160.

[Cao et al., 2011] Cao, S. T., Nguyen, L. A., and Szalas, A. (2011). On the Web Ontology Rule Language OWL 2 RL. In *Computational Collective Intelligence. Technologies and Applications - Third International Conference, ICCCI*, volume 6922 of *Lecture Notes in Computer Science*, pages 254–264. Springer.

[Carbone et al., 2015] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38.

[Carlson et al., 2010] Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Jr., E. R. H., and Mitchell, T. M. (2010). Toward an Architecture for Never-Ending Language Learning. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI*. AAAI Press.

[Carroll et al., 2004] Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters, WWW*, pages 74–83. ACM.

[Chang et al., 2008] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26.

[Codd, 1970] Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387.

[Cunningham et al., 2002] Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. (2002). A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 168–175. ACL.

[de la Croix and Licandro, 2012] de la Croix, D. and Licandro, O. (2012). The longevity of famous people from Hammurabi to Einstein. CORE Discussion Papers 2012052, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE).

[de Melo and Weikum, 2009] de Melo, G. and Weikum, G. (2009). Towards a universal wordnet by learning from combined evidence. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM*, pages 513–522. ACM.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L., Li, K., and Li, F. (2009). ImageNet: A large-scale hierarchical image database. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255. IEEE Computer Society.

[Dong et al., 2014] Dong, X., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., Strohmann, T., Sun, S., and Zhang, W. (2014). Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, pages 601–610. ACM.

[Erling and Mikhailov, 2009] Erling, O. and Mikhailov, I. (2009). RDF Support in the Virtuoso DBMS. In *Networked Knowledge - Networked Media - Integrating Knowledge Management, New Media Technologies and Semantic Systems*, volume 221 of *Studies in Computational Intelligence*, pages 7–24.

[Etzioni et al., 2004] Etzioni, O., Cafarella, M. J., Downey, D., Kok, S., Popescu, A., Shaked, T., Soderland, S., Weld, D. S., and Yates, A. (2004). Web-scale information extraction in knowitall: (preliminary results). In *Proceedings of the 13th international conference on World Wide Web, WWW*, pages 100–110. ACM.

[Fellbaum, 1998] Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. Language, speech, and communication. MIT Press.

[Fernández et al., 2013] Fernández, J. D., Martínez-Prieto, M. A., Gutiérrez, C., Polleres, A., and Arias, M. (2013). Binary RDF representation for publication and exchange (HDT). *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22–41.

[Ferrucci et al., 2010] Ferrucci, D. A., Brown, E. W., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J. M., Schlaefer, N., and Welty, C. A. (2010). Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3):59–79.

[Ficara et al., 2008] Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., and Pietro, A. D. (2008). An improved DFA for fast regular expression matching. *SIGCOMM Computer Communication Review*, 38(5):29–40.

[Galárraga et al., 2017] Galárraga, L., Razniewski, S., Amarilli, A., and Suchanek, F. M. (2017). Predicting Completeness in Knowledge Bases. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM*, pages 375–383. ACM.

[Galárraga et al., 2015a] Galárraga, L., Symeonidou, D., and Moissinac, J. (2015a). Rule Mining for Semantifying Wikilinks. In *Proceedings of the Workshop on Linked Data on the Web, LDOW, co-located with the 24th International World Wide Web Conference (WWW)*, volume 1409 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Galárraga et al., 2015b] Galárraga, L., Teflioudi, C., Hose, K., and Suchanek, F. M. (2015b). Fast rule mining in ontological knowledge bases with AMIE+. *The VLDB Journal*, 24(6):707–730.

[Ganguly et al., 1990] Ganguly, S., Silberschatz, A., and Tsur, S. (1990). A Framework for the Parallel Processing of Datalog Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 143–152. ACM Press.

[Ganguly et al., 1992] Ganguly, S., Silberschatz, A., and Tsur, S. (1992). Parallel Bottom-Up Processing of Datalog Queries. *J. Log. Program.*, 14(1&2):101–126.

[Gergaud et al., 2017] Gergaud, O., Laouenan, M., and Wasmer, E. (2017). A Brief History of Human Time. Exploring a database of notable people. *Sciences Po Economics Discussion Papers*.

[Glushkov, 1961] Glushkov, V. M. (1961). The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53.

[Guarino, 1998] Guarino, N. (1998). Formal Ontology and Information systems. In *Proceedings of the first international conference (FOIS)*, volume 46. IOS press.

[Gulwani, 2011] Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 317–330. ACM.

[Guo et al., 2005] Guo, Y., Pan, Z., and Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182.

[Haarslev and Möller, 2001] Haarslev, V. and Möller, R. (2001). RACER System Description. In *Automated Reasoning, First International Joint Conference, IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–706. Springer.

[Harris et al., 2013] Harris, S., Seaborne, A., and Prud'hommeaux, E. (2013). SPARQL 1.1 query language. *W3C recommendation*. URL: `http://www.w3.org/TR/sparql11-query/`.

[Hoffart et al., 2011a] Hoffart, J., Suchanek, F. M., Berberich, K., Lewis-Kelham, E., de Melo, G., and Weikum, G. (2011a). YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In *Proceedings of the 20th International Conference on World Wide Web, WWW (Companion Volume)*, pages 229–232. ACM.

[Hoffart et al., 2013] Hoffart, J., Suchanek, F. M., Berberich, K., and Weikum, G. (2013). YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence*, 194:28–61.

[Hoffart et al., 2011b] Hoffart, J., Yosef, M. A., Bordino, I., Fürstenau, H., Pinkal, M., Spaniol, M., Taneva, B., Thater, S., and Weikum, G. (2011b). Robust Disambiguation of Named Entities in Text. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 782–792. ACL.

[Huet et al., 2013] Huet, T., Biega, J., and Suchanek, F. M. (2013). Mining history with Le Monde. In *Proceedings of the workshop on Automated knowledge base construction, AKBC@CIKM*, pages 49–54. ACM.

[Isard et al., 2007] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference*, pages 59–72. ACM.

[Kalyanpur et al., 2011] Kalyanpur, A., Murdock, J. W., Fan, J., and Welty, C. A. (2011). Leveraging Community-Built Knowledge for Type Coercion in Question Answering. In *The Semantic Web - ISWC - 10th International Semantic Web Conference*, volume 7032 of *Lecture Notes in Computer Science*, pages 144–156. Springer.

[Kasneci et al., 2008] Kasneci, G., Ramanath, M., Suchanek, F. M., and Weikum, G. (2008). The YAGO-NAGA approach to knowledge discovery. *SIGMOD Record*, 37(4):41–47.

[Katsogridakis et al., 2017] Katsogridakis, P., Papagiannaki, S., and Pratikakis, P. (2017). Execution of Recursive Queries in Apache Spark. In *Euro-Par: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing*, volume 10417 of *Lecture Notes in Computer Science*, pages 289–302. Springer.

[Knight and Myers, 1995] Knight, J. R. and Myers, E. W. (1995). Approximate Regular Expression Pattern Matching with Concave Gap Penalties. *Algorithmica*, 14(1):85–121.

[Kornacker et al., 2015] Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., Joshi, I., Kuff, L., Kumar, D., Leblang, A., Li, N., Pandis, I., Robinson, H., Rorke, D., Rus, S., Russell, J., Tsirogiannis, D., Wanderman-Milne, S., and Yoder, M. (2015). Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR, Seventh Biennial Conference on Innovative Data Systems Research*. www.cidrdb.org.

[Krishnamurthy et al., 2008] Krishnamurthy, R., Li, Y., Raghavan, S., Reiss, F., Vaithyanathan, S., and Zhu, H. (2008). SystemT: a system for declarative information extraction. *SIGMOD Record*, 37(4):7–13.

[Kunegis, 2013] Kunegis, J. (2013). KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW*, pages 1343–1350. International World Wide Web Conferences Steering Committee / ACM.

[Kuzey et al., 2016] Kuzey, E., Setty, V., Strötgen, J., and Weikum, G. (2016). As Time Goes By: Comprehensive Tagging of Textual Phrases with Temporal Scopes. In *Proceedings of the 25th International Conference on World Wide Web, WWW*, pages 915–925. ACM.

[Lajus and Suchanek, 2018] Lajus, J. and Suchanek, F. M. (2018). Are All People Married?: Determining Obligatory Attributes in Knowledge Bases. In *Proceedings of the World Wide Web Conference on World Wide Web, WWW*, pages 1115–1124. ACM.

[Le and Gulwani, 2014] Le, V. and Gulwani, S. (2014). FlashExtract: a framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 542–553. ACM.

[Lehmann et al., 2015a] Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., and Bizer, C. (2015a). DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195.

[Lehmann et al., 2015b] Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., and Bizer, C. (2015b). DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195.

[Lenat and Guha, 1989] Lenat, D. and Guha, R. (1989). *Building large knowledge-based systems: representation and inference in the Cyc project*. Addison-Wesley Pub. Co.

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562.

[Leskovec and Krevl, 2014] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`. URL: `http://snap.stanford.edu/data`, accessed on 2018-01-17.

[Li et al., 2008] Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., and Jagadish, H. V. (2008). Regular Expression Learning for Information Extraction. In *Conference on Empirical Methods in Natural Language Processing, EMNLP, Proceedings of the Conference, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 21–30. ACL.

[Li et al., 2016] Li, Z., Wang, H., Shao, W., Li, J., and Gao, H. (2016). Repairing Data through Regular Expressions. *PVLDB*, 9(5):432–443.

[Magnini and Cavaglia, 2000] Magnini, B. and Cavaglia, G. (2000). Integrating Subject Field Codes into WordNet. In *Proceedings of the Second International Conference on Language Resources and Evaluation, LREC*. European Language Resources Association.

[Mahdisoltani et al., 2015] Mahdisoltani, F., Biega, J., and Suchanek, F. M. (2015). YAGO3: A Knowledge Base from Multilingual Wikipedias. In *CIDR, Seventh Biennial Conference on Innovative Data Systems Research, Online Proceedings*. www.cidrdb.org.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 135–146. ACM.

[Marx et al., 2013] Marx, E., Shekarpour, S., Auer, S., and Ngomo, A. N. (2013). Large-Scale RDF Dataset Slicing. In *IEEE Seventh International Conference on Semantic Computing*, pages 228–235. IEEE Computer Society.

[Marx et al., 2017] Marx, E., Shekarpour, S., Soru, T., Brasoveanu, A. M. P., Saleem, M., Baron, C., Weichselbraun, A., Lehmann, J., Ngomo, A. N., and Auer, S. (2017). Torpedo: Improving the State-of-the-Art RDF Dataset Slicing. In *11th IEEE International Conference on Semantic Computing, ICSC*, pages 149–156. IEEE Computer Society.

[Miller, 1995] Miller, G. A. (1995). WordNet: A Lexical Database for English. *Commun. ACM*, 38(11):39–41.

[Minkov et al., 2005] Minkov, E., Wang, R. C., and Cohen, W. W. (2005). Extracting Personal Names from Email: Applying Named Entity Recognition to Informal Text. In *HLT/EMNLP, Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 443–450. The Association for Computational Linguistics.

[Motik et al., 2014] Motik, B., Nenov, Y., Piro, R., Horrocks, I., and Olteanu, D. (2014). Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 129–137. AAAI Press.

[Murthy et al., 2012] Murthy, K., Padmanabhan, D., and Deshpande, P. M. (2012). Improving Recall of Regular Expressions for Information Extraction. In *Web Information Systems Engineer-*

ing - WISE - 13th International Conference. Proceedings, volume 7651 of *Lecture Notes in Computer Science*, pages 455–467. Springer.

[Myers and Miller, 1989] Myers, E. W. and Miller, W. (1989). Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5 – 37.

[Nastase et al., 2010] Nastase, V., Strube, M., Boerschinger, B., Zirn, C., and Elghafari, A. (2010). WikiNet: A Very Large Scale Multi-Lingual Concept Network. In *Proceedings of the International Conference on Language Resources and Evaluation, LREC*. European Language Resources Association.

[Navarro, 2004] Navarro, G. (2004). Approximate Regular Expression Searching with Arbitrary Integer Weights. *Nord. J. Comput.*, 11(4):356–373.

[Navigli and Ponzetto, 2012] Navigli, R. and Ponzetto, S. P. (2012). BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 193:217–250.

[Nie et al., 2012] Nie, Z., Wen, J., and Ma, W. (2012). Statistical Entity Extraction From the Web. *Proceedings of the IEEE*, 100(9):2675–2687.

[Niu et al., 2012] Niu, F., Zhang, C., Ré, C., and Shavlik, J. W. (2012). DeepDive: Web-scale Knowledge-base Construction using Statistical Learning and Inference. In *Proceedings of the Second International Workshop on Searching and Integrating New Web Data Sources*, volume 884 of *CEUR Workshop Proceedings*, pages 25–28. CEUR-WS.org.

[Ponzetto and Strube, 2008] Ponzetto, S. P. and Strube, M. (2008). WikiTaxonomy: A Large Scale Knowledge Resource. In *ECAI - 18th European Conference on Artificial Intelligence, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 751–752. IOS Press.

[Prasse et al., 2015] Prasse, P., Sawade, C., Landwehr, N., and Scheffer, T. (2015). Learning to identify concise regular expressions that describe email campaigns. *Journal of Machine Learning Research*, 16:3687–3720.

[Raman and Hellerstein, 2001] Raman, V. and Hellerstein, J. M. (2001). Potter's Wheel: An Interactive Data Cleaning System. In *VLDB , Proceedings of 27th International Conference on Very Large Data Bases*, pages 381–390. Morgan Kaufmann.

[Rapti et al., 2015] Rapti, A., Tsolis, D., Sioutas, S., and Tsakalidis, A. K. (2015). A Survey: Mining Linked Cultural Heritage Data. In *Workshop Proceedings of the 16th International Conference on Engineering Applications of Neural Networks, EANN*, pages 24:1–24:6. ACM.

[Razniewski et al., 2016] Razniewski, S., Suchanek, F. M., and Nutt, W. (2016). But What Do We Actually Know? In *Proceedings of the 5th Workshop on Automated Knowledge Base Construction, AKBC@NAACL-HLT*, pages 40–44. The Association for Computer Linguistics.

[Rebele et al., 2017a] Rebele, T., Nekoei, A., and Suchanek, F. M. (2017a). Using YAGO for the Humanities. In *Proceedings of the Second Workshop on Humanities in the Semantic Web (WHiSe II) co-located with 16th International Semantic Web Conference (ISWC)*, volume 2014 of *CEUR Workshop Proceedings*, pages 99–110. CEUR-WS.org. URL: `http://ceur-ws.org/Vol-2014/paper-11.pdf`.

[Rebele et al., 2016] Rebele, T., Suchanek, F. M., Hoffart, J., Biega, J., Kuzey, E., and Weikum, G. (2016). YAGO: A Multilingual Knowledge Base from Wikipedia, Wordnet, and Geonames. In *The Semantic Web - ISWC - 15th International Semantic Web Conference*, volume 9982 of *Lecture Notes in Computer Science*, pages 177–185. URL: `https://doi.org/10.1007/978-3-319-46547-0_19`.

[Rebele et al., 2018a] Rebele, T., Tanon, T. P., and Suchanek, F. M. (2018a). Technical Report: Answering Datalog Queries with Unix Shell Commands. Technical report. URL: `https://www.thomasrebele.org/publications/2018_report_bashlog.pdf`.

[Rebele et al., 2018b] Rebele, T., Tzompanaki, K., and Suchanek, F. (2018b). Adding Missing Words to Regular Expressions. In *Advances in Knowledge Discovery and Data Mining - 22nd Pacific-Asia Conference, PAKDD, Proceedings*, Lecture Notes in Computer Science. URL: `https://www.thomasrebele.org/publications/2018_pakdd.pdf`.

[Rebele et al., 2018c] Rebele, T., Tzompanaki, K., and Suchanek, F. (2018c). Technical Report: Adding Missing Words to Regular Expressions. Technical report, Telecom ParisTech. URL: `https://hal.archives-ouvertes.fr/hal-01745987v1`.

[Rebele et al., 2017b] Rebele, T., Tzompanaki, K., and Suchanek, F. M. (2017b). Visualizing the addition of missing words to regular expressions. In *Proceedings of the ISWC Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC)*, volume 1963 of *CEUR Workshop Proceedings*. CEUR-WS.org. URL: `http://ceur-ws.org/Vol-1963/paper457.pdf`.

[Rogala et al., 2016] Rogala, M., Hidders, J., and Sroka, J. (2016). DatalogRA: datalog with recursive aggregation in the spark RDD model. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 3. ACM.

[Russell, 1903] Russell, B. (1903). *The Principles of Mathematics*. Number vol. 1. W. W. Norton & Company.

[Saha et al., 2015] Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A. C., and Curino, C. (2015). Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1357–1369. ACM.

[Schich et al., 2014] Schich, M., Song, C., Ahn, Y.-Y., Mirsky, A., Martino, M., Barabási, A.-L., and Helbing, D. (2014). A network framework of cultural history. *Science*, 345:558–562.

[Shaw et al., 2012] Shaw, M., Koutris, P., Howe, B., and Suciu, D. (2012). Optimizing Large-Scale Semi-Naïve Datalog Evaluation in Hadoop. In *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0*, volume 7494 of *Lecture Notes in Computer Science*, pages 165–176. Springer.

[Shearer et al., 2008] Shearer, R., Motik, B., and Horrocks, I. (2008). HermiT: A Highly-Efficient OWL Reasoner. In *Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC)*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Shkapsky et al., 2016] Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., and Zaniolo, C. (2016). Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the International Conference on Management of Data, SIGMOD Conference*, pages 1135–1149. ACM.

[Sirin and Parsia, 2004] Sirin, E. and Parsia, B. (2004). Pellet: An OWL DL Reasoner, poster. In *Proceedings of the International Workshop on Description Logics (DL2004)*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Speer et al., 2017] Speer, R., Chin, J., and Havasi, C. (2017). ConceptNet 5.5: An Open Multilingual Graph of General Knowledge. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 4444–4451. AAAI Press.

[Suchanek et al., 2007] Suchanek, F. M., Kasneci, G., and Weikum, G. (2007). YAGO: A Core of Semantic Knowledge. In *Proceedings of the 16th International Conference on World Wide Web, WWW*, pages 697–706. ACM.

[Suchanek et al., 2008] Suchanek, F. M., Kasneci, G., and Weikum, G. (2008). YAGO: A Large Ontology from Wikipedia and WordNet. *Web Semantics*, 6(3):203–217.

[Suchanek and Weikum, 2014] Suchanek, F. M. and Weikum, G. (2014). Knowledge Bases in the Age of Big Data Analytics. *PVLDB*, 7(13):1713–1714.

[Suchanek and Weikum, 2016] Suchanek, F. M. and Weikum, G. (2016). Knowledge Representation in Entity-Centric Knowledge Bases. In *RUSSIR Summer School*.

[Tandon et al., 2015] Tandon, N., de Melo, G., De, A., and Weikum, G. (2015). Knowlywood: Mining Activity Knowledge From Hollywood Narratives. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM*, pages 223–232. ACM.

[Tandon et al., 2014] Tandon, N., de Melo, G., Suchanek, F. M., and Weikum, G. (2014). WebChild: harvesting and organizing commonsense knowledge from the web. In *Seventh ACM International Conference on Web Search and Data Mining, WSDM*, pages 523–532. ACM.

[Tsarkov and Horrocks, 2006] Tsarkov, D. and Horrocks, I. (2006). FaCT++ Description Logic Reasoner: System Description. In *Automated Reasoning, Third International Joint Conference, IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 292–297. Springer.

[US DHHS, 2017] United States Department of Health and Human Services (US DHHS), Centers for Disease Control and Prevention (CDC), National Center for Health Statistics (NCHS), Division of Vital Statistics (2017). Natality public-use data 2003-2006, and 2007-2015, on CDC WONDER Online Database. URL: `https://wonder.cdc.gov/natality.html`, accessed on 2017-07-24.

[Verborgh et al., 2016] Verborgh, R., Sande, M. V., Hartig, O., Herwegen, J. V., Vocht, L. D., Meester, B. D., Haesendonck, G., and Colpaert, P. (2016). Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 37-38:184–206.

[Vrandecic and Krötzsch, 2014] Vrandecic, D. and Krötzsch, M. (2014). Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85.

[W3C, 2014] W3C (2014). RDF 1.1 Concepts and Abstract Syntax. URL: `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`, accessed on 2018-04-21.

[Wagner and Fischer, 1974] Wagner, R. A. and Fischer, M. J. (1974). The String-to-String Correction Problem. *J. ACM*, 21(1):168–173.

[Wang et al., 2015] Wang, J., Balazinska, M., and Halperin, D. (2015). Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines. *PVLDB*, 8(12):1542–1553.

[Wikipedia, 2017] Wikipedia (2017). Bash (Unix shell) — Wikipedia, The Free Encyclopedia. URL: `https://en.wikipedia.org/w/index.php?title=Bash_(Unix_shell)`, accessed on 2017-11-08.

[Wolfson and Silberschatz, 1988] Wolfson, O. and Silberschatz, A. (1988). Distributed Processing of Logic Programs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 329–336. ACM Press.

[Wu et al., 2016] Wu, H., Liu, J., Wang, T., Ye, D., Wei, J., and Zhong, H. (2016). Parallel Materialization of Datalog Programs with Spark for Scalable Reasoning. In *Web Information Systems Engineering - WISE - 17th International Conference*, volume 10041 of *Lecture Notes in Computer Science*, pages 363–379.

[Wu et al., 1995] Wu, S., Manber, U., and Myers, E. W. (1995). A Subquadratic Algorithm for Approximate Regular Expression Matching. *J. Algorithms*, 19(3):346–360.

[Wu et al., 2012] Wu, W., Li, H., Wang, H., and Zhu, K. Q. (2012). Probase: a probabilistic taxonomy for text understanding. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 481–492. ACM.

[Yang and Liu, 1999] Yang, Y. and Liu, X. (1999). A Re-Examination of Text Categorization Methods. In *SIGIR: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 42–49. ACM.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*. USENIX Association.

[Zhou et al., 2010] Zhou, J., Larson, P., and Chaiken, R. (2010). Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of the 26th International Conference on Data Engineering, ICDE*, pages 1060–1071. IEEE Computer Society.

Appendices

Appendix A

Generated Unix Shell Script

This appendix shows the complete code listings of two Unix shell scripts that were created with the system of Chapter 6.

A.1 Datalog Mode

The following script was translated from the Datalog query in Figure 6.9 on page 123 in Section 6.7. Lines 1-17 are the header. The header is included in all generated scripts, and checks for available optimizations. Lines 20-24 read the relevant facts from the input file, and saves them in three temporary files. Lines 27-32 sort one of the just created files, so that it can be reused in the `while` loop. Lines 34-58 show the actual query plan, and Line 60 cleans up the temporary files.

```
1  #!/bin/bash
2  ###############################################################
3  # This script was generated by bashlog
4  # For more information, visit thomasrebele.org/projects/bashlog
5  ###############################################################
6
7  export LC_ALL=C
8  mkdir -p tmp
9  rm -f tmp/*
10 if type mawk > /dev/null; then awk="mawk"; else awk="awk"; fi
11 sort="sort "
12 check() { grep -- $1 <(sort --help) > /dev/null; }
```

```
13  check "--buffer-size" && sort="$sort --buffer-size=20% "
14  check "--parallel"    && sort="$sort --parallel=2 "
15
16  read_ntriples() { $awk -F" " '{ sub(" ", "\t"); sub(" ", "\t"); sub(/ \.$ ↩
        /, ""); print $0 }' "$@"; }
17  conv_ntriples() { $awk -F$'\t' '{ print $1 " " $2 " " $3 " ." }'; }
18
19
20  touch tmp/mat0 tmp/mat1 tmp/mat2
21  $awk -v FS=$'\t' ' ($4 == "<Italy>" && $3 == "<isLocatedIn>") { print $2   ↩
        >> "tmp/mat0" }
22   ($3 == "<hasChild>") { print $4 FS $2 >> "tmp/mat1" }
23   ($3 == "<wasBornIn>") { print $2 FS $4 >> "tmp/mat2" }
24    ' *.tsv
25
26
27  mkfifo tmp/lock_mat3; (
28     $sort -t $'\t' -k 1 tmp/mat1 > tmp/mat3;
29     mv tmp/lock_mat3 tmp/done_mat3;
30     cat tmp/done_mat3 > /dev/null & exec 3> tmp/done_mat3;
31     exec 3>&-;
32  ) &
33
34  # plan
35  $sort -t $'\t' -k 1 -k 2 -u \
36  <(join -t $'\t' -1 1 -2 1 -o 1.2,1.3 \
37     <($sort -t $'\t' -k 1 \
38        <(join -t $'\t' -1 1 -2 2 -o 1.2,2.1,2.2 \
39           <($sort -t $'\t' -k 1 tmp/mat2) \
40           <($sort -t $'\t' -k 2 \
41              <($sort -t $'\t' -k 1 -k 2 -u tmp/mat1 \
42                     | tee tmp/full4 > tmp/delta4
43                 while
44
45                 $sort -t $'\t' -k 1 -k 2 -u \
46                    <(cat tmp/lock_mat3 1>&2 2>/dev/null ;  \
47                       join -t $'\t' -1 2 -2 1 -o 1.1,2.2 \
48                       <($sort -t $'\t' -k 2 tmp/delta4) tmp/mat3) \
49                     | comm -23 - tmp/full4 > tmp/new4;
50
51                 mv tmp/new4 tmp/delta4 ;
52                 $sort -u --merge -o tmp/full4 tmp/full4 tmp/delta4;
53                 [ -s tmp/delta4 ];
54                 do continue; done
55
56                 rm tmp/delta4
57                 cat tmp/full4)))) \
58     <($sort -t $'\t' -k 1 -u tmp/mat0))
59
60   rm -f tmp/*
```

A.2 SPARQL/OWL Mode

The following script was translated from the SPARQL query and OWL TBox from Figures 6.11 and 6.10 on page 123. Lines 1-17 are the header. Lines 20-33 store the relevant facts in temporary files. Lines 35-58 show the query plan, and Line 60 cleans up temporary files.

```
1  #!/bin/bash
2  ###############################################################
3  # This script was generated by bashlog
4  # For more information, visit thomasrebele.org/projects/bashlog
5  ###############################################################
6
7  export LC_ALL=C
8  mkdir -p tmp
9  rm -f tmp/*
10 if type mawk > /dev/null; then awk="mawk"; else awk="awk"; fi
11 sort="sort "
12 check() { grep -- $1 <(sort --help) > /dev/null; }
13 check "--buffer-size" && sort="$sort --buffer-size=20% "
14 check "--parallel"    && sort="$sort --parallel=2 "
15
16 read_ntriples() { $awk -F" " '{ sub(" ", "\t"); sub(" ", "\t"); sub(/ \.$ ↩
       /, ""); print $0 }' "$@"; }
17 conv_ntriples() { $awk -F$'\t' '{ print $1 " " $2 " " $3 " ." }'; }
18
19
20 touch tmp/mat0 tmp/mat1 tmp/mat2
21 $awk -v FS=$'\t' '
22   BEGIN {
23    mat0_out0c2_cond1["<http://yago-knowledge.org/resource/hasAncestor>"] = ↩
        "1";
24    mat0_out0c2_cond1["<http://yago-knowledge.org/resource/hasParent>"] =  ↩
       "1";
25    mat0_out2c0_cond1["<http://yago-knowledge.org/resource/hasChild>"] =  ↩
       "1";
26   }
27
28  (($2) in mat0_out0c2_cond1){ print $1 FS $3 >> "tmp/mat0" }
29  (($2) in mat0_out2c0_cond1){ print $3 FS $1 >> "tmp/mat0" }
30  ($3 == "<http://yago-knowledge.org/resource/Italy>" && $2 == "<http:// ↩
       yago-knowledge.org/resource/isLocatedIn>") { print $1 >> "tmp/mat1" }
31  ($2 == "<http://yago-knowledge.org/resource/wasBornIn>") { print $1 FS $3 ↩
        >> "tmp/mat2" }
32   ' \
33 <(read_ntriples yago-sample.ntriples)
34
```

```
35  $sort -t $'\t' -k 1 -k 2 -u \
36  <(join -t $'\t' -1 1 -2 1 -o 1.2,1.3 \
37      <($sort -t $'\t' -k 1 \
38          <(join -t $'\t' -1 1 -2 2 -o 1.2,2.1,2.2 \
39              <($sort -t $'\t' -k 1 tmp/mat2) \
40              <($sort -t $'\t' -k 2 \
41                  <($sort -t $'\t' -k 1 -k 2 -u tmp/mat0 \
42                           | tee tmp/full3 > tmp/delta3
43                      while
44
45                      $sort -t $'\t' -k 1 -k 2 -u \
46                          <(join -t $'\t' -1 2 -2 1 -o 1.1,2.2 \
47                              <($sort -t $'\t' -k 2 tmp/delta3) \
48                              <($sort -t $'\t' -k 1 tmp/full3)) \
49                           | comm -23 - tmp/full3 > tmp/new3;
50
51                      mv tmp/new3 tmp/delta3 ;
52                      $sort -u --merge -o tmp/full3 tmp/full3 tmp/delta3 ;
53                      [ -s tmp/delta3 ];
54                      do continue; done
55
56                      rm tmp/delta3
57                      cat tmp/full3)))) \
58      <($sort -t $'\t' -k 1 -u tmp/mat1))
59
60   rm -f tmp/*
```

Appendix B

Résumé en français

B.1 Introduction

B.1.1 Motivation

Une base de connaissances est une collection de connaissances sur le monde qui peut être traitée par ordinateur. Une base de connaissances contient généralement des *entités*, comme Albert Einstein, l'Université technique de Munich (TUM) ou Paris. Elle contient également des *faits* sur ces entités, comme le fait qu'Albert Einstein est un physicien, que TUM est une université ou que Paris se trouve en France.

Au cours des dernières décennies, les bases de connaissances ont commencé à apparaître dans la vie quotidienne. Par exemple, lorsqu'un utilisateur recherche des informations sur une personne connue sur l'un des deux principaux moteurs de recherche - Google ou Bing -, il reçoit non seulement une liste de pages Web, mais également une infobox sous forme de tableau. Par exemple, si il recherche «Albert Einstein», les moteurs de recherche montrent, entre autres choses, sa date de naissance et de décès, ses épouses et certaines de ses œuvres. Les bases de connaissances sont également utilisées dans la traduction automatique, la recherche sémantique, les assistants intelligents et l'exploration de données [Suchanek and Weikum, 2016].

Il a fallu plusieurs années, de conception et de mise en œuvre de projets de base de connaissances, pour que la technologie devienne suffisamment mature et utile à l'utilisateur final. Une base de connaissances précoce notable est WordNet, une base de données linguistiques pour la langue anglaise [Miller, 1995].

Cependant, il faut beaucoup de temps et d'efforts pour créer une base de connaissances manuellement. La recherche a donc commencé à examiner la construction automatique de bases de connaissances. Avec la croissance du Web, de plus en plus d'approches construisent automatiquement des bases de connaissances en extrayant des informa-

tions de corpus Web, comme Wikipedia. Les techniques d'extraction d'informations, comme les expressions régulières, peuvent rapidement générer un grand nombre de faits.

Certaines des approches les plus importantes dans ce domaine sont YAGO, DBpedia, Wikidata, NELL et Google Knowledge Vault. La base de connaissances YAGO, en particulier, a été l'une des premières approches dans ce sens. Elle a combiné les forces de deux des projets mentionnés précédemment. WordNet a une taxonomie faite à la main et bien conçue (hiérarchie de classes), mais peu d'entités. D'autre part, Wikipedia contient de nombreuses entités, mais ne fournit pas une taxonomie propre. YAGO a comblé l'écart en fusionnant les deux ressources. YAGO se distingue des projets similaires en se concentrant sur la précision. C'est le rapport des vrais faits par rapport aux données de référence. L'optimisation de la précision va souvent de pair avec la réduction de la couverture, c'est-à-dire le nombre total de faits. Néanmoins, YAGO atteint une précision de 95 %, tout en contenant des millions de faits.

Lors des études de doctorat, j'ai participé activement à la maintenance et à l'évaluation de la base de connaissances YAGO. Par conséquent, la première partie de ma thèse est consacrée à la description de cette base de connaissances.

Malgré la bonne précision et la couverture de YAGO, des améliorations sont encore possibles. YAGO est encore incomplet - même en utilisant Wikipedia comme données de référence. Par exemple, la version originale de YAGO connaissait le lieu de résidence pour seulement 53 % des personnes et le sexe pour seulement 63 %. Cette thèse montre comment améliorer la couverture de quatre types d'informations essentielles sur les personnes à YAGO : les dates de naissance et de décès, le sexe et le lieu de résidence. Par exemple, nous pouvons maintenant extraire le sexe pour 90 % des personnes. Cette nouvelle richesse de données permet d'utiliser YAGO pour des études historiques.

Pour augmenter la couverture, il peut être nécessaire de déboguer et d'adapter des expressions régulières. Par exemple, la version précédente de YAGO a extrait l'année des dates à trois chiffres, comme «100 BC», mais elle n'a pas extrait l'année des dates à deux chiffres, comme «44 BC». Corriger les expressions régulières est fastidieux, car les expressions régulières sont assez complexes. Cette observation a conduit à l'idée d'automatiser l'adaptation des expressions régulières. Cette thèse présente deux algorithmes capables de modifier automatiquement une expression régulière pour qu'elle corresponde à un mot donné. Nos expériences montrent que nos algorithmes peuvent généraliser des expressions rationnelles basées sur de petites données d'entraînement.

Pendant le travail sur le maintien et l'implémentation de nouveaux algorithmes pour la base de connaissances YAGO, j'ai souvent été confronté à la tâche de rechercher tous les faits dans YAGO à propos d'une certaine entité. Le chargement de toute la base de connaissances dans une base de données ou dans un triplestore demanderait trop de temps et d'efforts. Le moyen le plus simple d'obtenir le résultat était normalement

d'écrire une commande Unix courte, telle que grep ou AWK, et d'attendre quelques minutes. Au fil des années, je suis devenu de plus en plus expérimenté avec les commandes Unix, leurs paramètres et Bash, l'interpréteur de ligne de commande.

Ce savoir-faire a inspiré la dernière partie de ma thèse : Un algorithme qui transforme les requêtes de base de données en scripts Bash. L'utilisateur peut saisir sa requête dans Datalog ou SPARQL dans une interface Web. Cette requête est automatiquement transformée en script Bash optimisé. Il peut ensuite télécharger le script Bash et l'exécuter dans la base de connaissances. De cette façon, l'utilisateur peut obtenir les résultats de sa requête sans importer la base de connaissances dans une base de données. Nos expériences montrent que nos scripts Bash sont souvent plus rapides que les bases de données traditionnelles ou les triplestores.

B.1.2 Outline

Ce travail se compose de quatre parties.

Décrire YAGO comme ressource. Dans la première partie, cette thèse présente la base de connaissances YAGO et décrit son historique, sa construction, son contenu, sa maintenance et ses applications. Dans ce chapitre, je décris également mes contributions à la base de connaissances YAGO en termes de maintenance, d'évaluation et de développement.

Digital humanities. La deuxième partie de cette thèse étudie l'application de YAGO aux humanités numériques. La base de connaissances est enrichie de nouveaux faits concernant les dates de naissance et de décès, les lieux et le sexe des personnes. Plusieurs études de cas montrent l'utilité de telles données.

Réparation d'expressions régulières. La troisième partie se concentre sur un problème qui survient souvent lors de l'extraction d'informations : une expression régulière peut ne pas correspondre à tous les mots auxquels elle doit correspondre. Deux algorithmes sont présentés, qui modifient une expression régulière, de sorte que l'expression régulière corresponde également à un ensemble de mots donnés.

Répondre aux requêtes Datalog avec Bash. Enfin, la quatrième partie de cette thèse décrit un système qui traduit les programmes Datalog ou les requêtes SPARQL en scripts Bash. Cela permet à un utilisateur de disposer de données tabulaires volumineuses sur un système UNIX sans installation de logiciel. Il est intéressant de noter que notre approche permet d'obtenir des résultats de recherche dans un délai comparable à celui des systèmes de pointe.

B.2 La base de connaissances YAGO

YAGO est une grande base de connaissances qui est construite automatiquement à partir de Wikipedia, WordNet et GeoNames. YAGO couvre des sujets d'intérêt général, comme les entités géographiques, les personnalités publiques ou historiques, les films et les organisations. YAGO se concentre sur la qualité de l'extraction et atteint une précision évaluée manuellement à 95 %. Dans cette section, j'explique comment YAGO est construit à partir de ses sources et comment sa qualité est évaluée.

B.2.1 Historique

Le projet YAGO a démarré en 2006 à partir d'une idée simple : Wikipedia contient un grand nombre d'instances, comme des chanteurs, des films ou des villes. WordNet a une taxonomie très élaborée. Il semblait donc prometteur de combiner les deux ressources pour obtenir le meilleur des deux mondes.

La première version de YAGO [Suchanek et al., 2007] a extrait des faits principalement des noms de catégories de Wikipedia en Anglais. Avec la première mise à jour de YAGO en 2008 [Suchanek et al., 2008], le projet a également commencé à extraire des infoboxes. En 2010, l'équipe YAGO a commencé à travailler sur l'extraction de méta-faits temporels et géographiques, qui ont abouti à YAGO2 [Hoffart et al., 2011a; Hoffart et al., 2013]. YAGO3 [Mahdisoltani et al., 2015] a ajouté une extraction de 10 différentes langues Wikipedia en 2015.

J'ai contribué au projet, entre autres, en développant des expressions régulières pour extraire les littéraux des attributs infobox, gérer l'évaluation, corriger les erreurs dans les algorithmes d'extraction et contribuer à la version Open Source.

B.2.2 Contenu

Les faits de YAGO suivent le modèle de RDF décrit dans le chapitre 2 page 19. Les faits sont représentés par des triples d'un sujet, un prédicat et un objet. Un exemple est

<Barack_Obama> <wasBornOnDate> "1961-08-04" ^^ xsd :date.

Dans YAGO, les entités (comme <Barack_Obama>) et les relations (comme <wasBornOnDate>) ont des noms lisibles par l'homme. YAGO donne également à chaque fait un *identifiant de fait*. L'identifiant est généré en combinant des valeurs de hachage du sujet, du prédicat et de l'objet. Par exemple, le fait ci-dessus a l'identifiant de fait <id_1km2mmx_1xk_17y5fnj>. Cet identifiant de fait permet à YAGO d'indiquer des informations temporelles ou spatiales, ou l'origine des faits.

B.2.3 Construction de YAGO

Dans YAGO, un *extracteur* est un petit module de code qui est responsable d'une seule sous-tâche d'extraction bien définie. Un extracteur prend certains thèmes en entrée et produit certains thèmes en sortie. Le graphe de dépendance des extracteurs est un graphe acyclique dirigé (DAG).

En règle générale, chaque page Wikipedia devient une entité dans YAGO. Les faits sur ces entités sont créés principalement à partir des infoboxes de Wikipedia. YAGO prend les feuilles de la hiérarchie des catégories Wikipedia et les relie aux synsets WordNet.

Alors que les extracteurs initiaux sont responsables de l'extraction des données brutes, les extracteurs suivants sont responsables du nettoyage de ces faits. Les faits sont d'abord redirigés, un processus où les entités sont remplacées par leurs versions canoniques de Wikipedia. Ils sont ensuite dédupliqués et envoyés à travers différentes vérifications syntaxiques et sémantiques. Plus particulièrement, la vérification de la conformité des faits aux signatures de type des relations [Biega et al., 2013a; Hoffart et al., 2011a; Hoffart et al., 2013; Kasneci et al., 2008; Suchanek et al., 2008].

B.2.3.1 Evaluation

Chaque version majeure de YAGO est évaluée selon la qualité. Comme il n'existe pas un test de référence de qualité comparable, cette évaluation est effectuée manuellement. Étant donné que le grand nombre de faits dans YAGO rend une évaluation manuelle complète irréalisable, un échantillon aléatoire de faits est sélectionné à partir de chaque relation et évalué. Seuls les faits obtenus par extraction d'information sont évalués (hors faits importés), par rapport à la source d'extraction (Wikipedia). La signification statistique est vérifiée à l'aide de l'intervalle de Wilson [Brown et al., 2001]. L'intervalle de Wilson calcule un intervalle de confiance, dans lequel le rapport réel repose sur une probabilité α.

La dernière évaluation de YAGO a été réalisée en 2015 avec la version 3.0.2, et a duré deux mois. 15 personnes ont participé et ont évalué 4 412 faits sur 76 relations, qui contiennent au total 60 millions de faits. Ils ont jugé que 98 % des faits de l'échantillon étaient corrects. Pondéré par le nombre de faits, l'intervalle de Wilson a un centre de 95 % et une largeur de 4,19 %. Cela signifie que le vrai rapport des faits corrects dans YAGO se situe entre 91 % et 99 %, avec $\alpha = 95\%$ de probabilité.

B.2.4 Résumé

YAGO est une base de connaissances qui unifie les informations de Wikipedia, WordNet et GeoNames en un tout cohérent. Cette section a décrit les sources, le processus d'extraction et les applications de YAGO.

Même si YAGO contient un grand nombre de faits sur un grand nombre d'entités différentes, il ne représente qu'une infime partie des faits qui sont vrais dans le monde réel. Dans la section suivante, nous voyons comment nous pouvons étendre les connaissances de YAGO.

B.3 Extension des informations sur les personnes dans YAGO

Dans ce chapitre, nous étudions comment les données de YAGO peuvent être utilisées pour des études de cas dans les sciences humaines. Nous discutons également des méthodes d'extraction de l'information que nous avons utilisées pour rendre YAGO suffisamment complet pour que ces analyses fonctionnent.

B.3.1 Introduction

Récemment, les humanités numériques ont trouvé un terrain d'entente avec la recherche sur les bases de connaissances [Adamou et al., 2016; Rapti et al., 2015], sachant que les données des bases de connaissances peuvent aider à répondre aux questions des humanités numériques. Dans ce chapitre, nous souhaitons pousser un peu plus loin cette symbiose fructueuse et étudier dans quelle mesure les données de YAGO peuvent aider à étudier des questions d'histoire et de société.

L'incomplétude est un problème général sur le Web sémantique : dans YAGO, seule la moitié des habitants ont un lieu de résidence ; dans DBpedia [Lehmann et al., 2015a], seulement 0,2 % des personnes ont un genre et dans Wikidata [Vrandecic and Krötzsch, 2014], seulement 3 % des personnes ont un père [Razniewski et al., 2016]. Par conséquent, notre premier défi consiste à rendre les données plus complètes.

Dans ce chapitre, nous décrivons comment les mécanismes d'extraction peuvent être améliorés afin qu'ils extraient encore plus d'informations de la source. Nous montrons que nos techniques améliorent jusqu'à 60 % la couverture de YAGO sur certains attributs.

Pour montrer l'utilité de ces nouvelles données, nous procédons ensuite à plusieurs études de cas. Entre autres, nous utilisons les données de YAGO pour suivre l'évolution de l'espérance de vie à travers l'histoire, par genre. Avec ces analyses, nous montrons

que les données du Web sémantique peuvent aider à éclairer les questions des sciences humaines.

B.3.2 Méthodes

B.3.2.1 Genre

Les infoboxes de Wikipédia ne mentionnent pas le genre d'une personne. Le genre a été ajouté seulement dans YAGO3. Il a été extrait grace aux pronoms sur la page par le *GenderPronounExtractor* (Algorithme 1, Lines 1-4). Cela a bien fonctionné, mais la couverture était plutôt faible. Seulement 61 % des gens avaient un genre.

Nous l'améliorons comme suit : Nous avons écrit le *GenderNameExtractor*, qui déduit le genre à partir du prénom d'une personne. Disons que notre échantillon pour le prénom v a une proportion de p % d'hommes. Nous utilisons l'estimateur de Wilson (avec α = 5 %) pour généraliser la proportion d'hommes dans notre échantillon à la proportion d'hommes dans le monde réel. Si la limite inférieure de l'intervalle de Wilson est d'au moins 95 %, nous attribuons le nom au genre masculin (de manière analogue pour les femmes). Les valeurs de α et 95 % sont celles que l'évaluation de YAGO [Suchanek et al., 2007] a utilisé.

L'algorithme 2 page 45 définit d'abord une fonction auxiliaire qui associe le genre au prénom. La fonction auxiliaire extrait le prénom et le genre des articles en utilisant le *GenderCategoryExtractor* (Ligne 2-3). Ensuite, il applique le test statistique sur chaque prénom (Ligne 5-9) pour générer une association prénom-genre. Les lignes 12-15 listent la fonction qui fait correspondre un article à un genre en se basant sur le prénom, si possible.

Ces trois extracteurs produisent des ensembles de faits dans le cadre de YAGO. Celles-ci sont ensuite collectés par un quatrième extracteur, qui attribue au plus un genre à chaque personne, en donnant la priorité au *GenderCategoryExtractor*, suivi du *GenderNameExtractor* et le *GenderPronounExtractor* (algorithme 3 page 45).

B.3.2.2 Dates

YAGO récolte les dates de naissance et les dates de décès des personnes de deux sources : les infoboxes et les catégories de Wikipedia. Les infoboxes contiennent une liste de paires attribut-valeur, de la forme *date-de-naissance = 8 janvier 1935*. En ce qui concerne les catégories, il existe une catégorie par année de naissance (par exemple, *Naissance en 1935*). Les années ont été extraites en utilisant des expressions régulières.

De nombreuses pages Wikipedia contiennent plus d'une date dans les attributs infobox et dans les catégories. Cependant, les gens ne peuvent avoir qu'un seul anniversaire. Les

versions précédentes de YAGO utilisaient la date de l'infobox si disponible et, sinon, l'année de la catégorie par défaut. Le problème avec cette approche est que la qualité de l'extraction est plus élevée pour les catégories que pour les infoboxes. Cela est dû à la diversité des formats de date dans les infoboxes (*01/08/1935, 08/01/1935, 8 janv. 1935*, etc.) et aux dates non pertinentes, comme les dates de publication des références.

Nous illustrons le nouvel algorithme basé sur l'extraction de dates de naissance, voir Algorithme 4 page 46. Tout d'abord, nous extrayons les dates des infoboxes I et des catégories C comme précédemment (Ligne 1-2). Ensuite, nous analysons la date de naissance de chaque personne (Lignes 4-9). S'il existe des dates de catégorie (ligne 6), alors nous ne conservons que les dates des infoboxes dans I dont l'année coïncide avec une date de catégorie dans C. Enfin, nous affichons la première date de l'infobox, et s'il n'y en a pas, la première date de catégorie (Line 9). Nous procédons de manière analogue pour les dates de décès.

B.3.2.3 Lieu de résidence

YAGO extrait le lieu de naissance, le lieu de décès et le lieu de résidence des infoboxes de Wikipedia. Le problème avec cette approche est que les données dans les infoboxes sont rares. Cependant, certaines catégories donnent la nationalité ou le lieu de résidence d'une personne, comme dans *Egyptian queens regnant*.

Par conséquent, nous avons écrit un nouvel extracteur qui récolte également les catégories de Wikipedia. Nous compilons d'abord une liste de noms de lieux (Line 1) - en partie à partir de la liste Wikipedia des démonymes [1] [2], et en partie des listes d'empires sur Wikipedia [3].

Nous utilisons cette correspondance pour attribuer chaque catégorie à un ensemble d'emplacements (lignes 5 à 9). Nous prenons une catégorie, par exemple <Nobility of the Holy Roman Empire>, et recherchons tous les noms de lieux et parties contenus dans le nom de la catégorie (Ligne 6) : «Holy Roman Empire», «Holy Roman», «Roman Empire» et «Roman». Il est évident que nous ne voulons pas l'associer à l'emplacement <Roman_Empire>. Ainsi, nous supprimons toutes les occurences qui se produisent dans une autre occurence (Ligne 7). Dans l'exemple, seul le nom de lieu «Holy Roman Empire» reste après le retrait. L'application de la correspondance aux résultats restants conduit à l'ensemble des emplacements pour cette catégorie.

Notre extracteur analyse alors les catégories d'une personne x et compte le nombre de fois où chaque emplacement apparaît (Ligne 8). Pour chaque emplacement y le plus

1. dénotation des personnes vivant à un certain endroit
2. https://en.wikipedia.org/wiki/Demonym, et https://en.wikipedia.org/wiki/List_of_adjectival_and_demonymic_forms_of_place_names
3. https://en.wikipedia.org/wiki/List_of_empires

Extraction	YAGO avant	YAGO maintenant	Précision
Dates de naissance	1 651 528	1,687,943	100 %
Dates de décès	805,377	821 780	100 %
Le genre	1 354 763	1 983 737	98 %
Lieu de résidence	1 179 355	1,916,695	98 %

TABLE B.1 – Couverture et précision de nos méthodes

fréquent, nous créons un fait x <livedIn> y (Ligne 10). Notre raisonnement est que les catégories ne peuvent ni déterminer le lieu de naissance ni la nationalité de manière fiable, mais qu'elles peuvent au moins indiquer un lieu de résidence.

B.3.3 Résultats

Dans cette section, nous étudions l'effet de nos nouvelles méthodes d'extraction. Pour la précision, nous avons pris un échantillon aléatoire de 100 faits par méthode et l'avons vérifié manuellement par rapport à Wikipedia, comme c'est généralement le cas pour YAGO (voir le chapitre 3 page 27). Pour la couverture, nous avons compté le nombre de personnes uniques dans YAGO qui ont un certain attribut. Nous avons comparé notre couverture aux méthodes précédemment implémentées de YAGO, exécutées sur la même version de Wikipedia.

La table 4.1 page 49 montre nos résultats. YAGO contient 2 215 360 personnes. Nos valeurs de précision sont très bonnes. Les 2 % de mauvais résultats pour les genres étaient exclusivement dus à des erreurs de type et non à l'extraction elle-même. L'extraction des résidences a également souffert de ce problème. En outre, il a produit 8 % de résidences anachroniques (comme *German Empire* au lieu de *Germany*). Nous les avons considérés comme corrects aux fins de notre étude. La précision des dates de naissance et de décès est extraordinaire.

De plus, nos méthodes ont considérablement augmenté le nombre de points de données pour tous les attributs que nous considérons. Les genres, par exemple, ont augmenté de 46 %. Les lieux de résidence ont même augmenté de 62 %.

B.3.4 Études de cas

Dans cette section, nous visons à montrer l'utilité de nos données dans des études de cas. Ce résumé décrit deux études de cas : l'espérance de vie au fil du temps et l'âge à la naissance du premier enfant.

B.3.4.1 Taille de la population au fil du temps

En préambule, nous montrons pour chaque siècle combien de personnes de ce siècle ont des faits dans YAGO. Nous prenons en compte toutes les personnes avec la date de naissance ou de décès avant 2017.

Le diagramme est obtenu en assignant chaque personne à un siècle, basé sur l'année de naissance ou de décès. Le résultat est montré dans la figure 4.1 page 50. Naturellement, YAGO ne connaît que quelques personnes des premiers siècles 1000 avant notre ère ou plus tôt. En outre, le nombre de femmes à YAGO ne représente qu'une fraction des hommes.

B.3.4.2 Espérance de vie au fil du temps

Notre premier cas (Figure 4.2 page 51) étudie la structure de la vie à travers l'histoire. Il le fait en traçant la durée de vie moyenne des hommes et des femmes en fonction du temps. Nous avons limité l'étude aux siècles où nous avions plus de 100 hommes ou 100 femmes. Il est intéressant de noter qu'il n'y a pas de tendance dans les données avant le 15ème siècle, l'âge moyen fluctuant autour de 53 ans pour les femmes et 60 ans pour les hommes. Les effets de la peste noire se reflètent clairement dans nos données : l'espérance de vie diminue au 13ème siècle.

Au-delà de cela, il y a une augmentation constante de la durée de vie entre les genres au cours des 500 dernières années. Nous constatons également que les femmes ont généralement une durée de vie plus courte dans nos données. Cela change cependant au 19ème siècle : les femmes vivent plus longtemps que les hommes. Comme nos données sont assez denses à partir du 19ème siècle, ce fait est statistiquement significatif dans nos données.

B.3.4.3 Âge à la naissance du premier enfant

Nous nous tournons maintenant vers l'âge auquel les gens deviennent parents. La figure 4.6 page 56 indique l'âge auquel les personnes ont leurs premier et dernier enfants au cours du dernier millénaire. Encore une fois, nous avons limité notre analyse aux siècles où nous avions au moins 100 paires parent-enfant.

Les hommes montrent une augmentation de l'âge auquel ils ont leur premier enfant, passant d'environ 28 à environ 32 ans. Pour les femmes, nous voyons une histoire similaire, avec une différence importante : l'augmentation est concentrée au cours des 2 derniers siècles. Cependant, l'âge auquel les hommes ont leur dernier enfant est presque inchangé jusqu'en 1700 et a diminué depuis. Pour les femmes, l'âge à la naissance du dernier enfant est inchangé.

Nous supposons que deux phénomènes démographiques sont à l'origine de ces tendances. Tout d'abord, les femmes ont moins d'enfants en repoussant l'âge auquel elles ont leur premier enfant. Deuxièmement, la différence d'âge entre les pères et les mères diminue. Ensemble, ces deux causes peuvent créer un accroissement de l'âge de la mère au premier enfant et un diminution de l'âge du père au dernier enfant, avec un âge constant de la mère au dernier enfant et du père au premier enfant.

B.3.5 Résumé

Dans ce chapitre, nous avons étudié comment les données du Web sémantique peuvent aider à la recherche dans les humanités numériques. Plus précisément, nous avons utilisé la base de connaissances YAGO pour étudier - entre autres - l'espérance de vie des personnes au fil de siècles et l'âge de parents à la naissance de leur premier enfant. Nous avons également présenté des méthodes pour améliorer la couverture de YAGO. Nos méthodes sont intégrées à l'infrastructure YAGO et le code source est inclus dans la version open source de YAGO.

L'extension de YAGO repose en partie sur des expressions régulières. Les expressions doivent parfois être adaptées, de sorte qu'elles correspondent à toutes les chaînes que nous avons voulu faire correspondre. Dans le prochain chapitre, nous verrons comment adapter automatiquement les expressions régulières.

B.4 Ajout de mots manquants aux expressions régulières

Dans le chapitre précédent, nous avons étudié la tâche d'extraire des dates dans du texte par des expressions régulières (ou regexes). Nous avons vu que même des expressions régulières conçues à la main pourraient ne pas correspondre à tous les mots prévus. Ce problème apparaît non seulement dans YAGO, mais aussi dans de nombreuses applications gourmandes en données qui utilisent des expressions regulières pour l'extraction d'informations.

Dans ce chapitre, nous proposons une nouvelle méthode pour généraliser automatiquement une regex afin qu'elle corresponde à un ensemble de mots donnés. Notre méthode trouve une correspondance approximative entre le mot manquant et la regex, et ajoute des disjonctions pour les parties qui n'ont pas été appariées de manière appropriée. Notre objectif est d'améliorer le rappel de la regex initiale sans nuire à sa précision. Nous evoluons l'efficacité et la généralité de notre technique par des expériences sur divers jeux de données d'extraction d'informations.

B.4.1 Préliminaires

Comme les expressions régulières sont la fondation de ce travail, nous commençons par définir la famille des regexes que nous considérons, ainsi que d'autres notions pertinentes. Ensuite, nous donnons la définition formelle de notre problème.

Une *expression régulière* (ou regex) est soit

— l'expression vide ϵ, ou
— un élément de Σ, ou une expression de la forme
— AB (une *concaténation*),
— $(A|B)$ (une *disjonction*), ou
— $(A)^*$ (une *étoile de Kleene*), où A et B sont des expressions regulières.

Nous utiliserons aussi les notations courantes pour une regex A et des entiers n, m :

— $A\{0,0\} := \epsilon$,
— $A\{0,n\} := (\epsilon|A\{1,n-1\})$,
— $A\{n,m\} := AA\{n-1,m\}$,
— $A\{n\} := A\{n,n\}$,
— $A? := (\epsilon|A)$, et
— $A^+ := AA^*$.

Le problème que nous étudions est le suivant :

Énoncé du problème. *Étant donné une regex r, un ensemble S d'exemples positifs, $\varepsilon \notin S$ et un ensemble E^- d'exemples négatifs, comme $|S| \ll |E^-|$, trouver une «bonne», regex r' telle que $L(r) \subseteq L(r')$, $S \subseteq L(r')$ et $|L(r') \cap E^-|$ petit.*

En d'autres termes, nous voulons généraliser la regex afin qu'elle accepte toutes les chaînes de caractères auxquelles elle correspondait précédemment, plus les nouveaux examples positives. Par exemple, avec une regex r = `[0-9]+` et une chaîne de caractères s = "12 34 56", une regex que nous pourrions vouloir trouver est r' = `([0-9] ?)+`. Cette regex accepte toutes les chaînes de caractères qui correspondent à r et également à s.

Maintenant, il y a évidemment des solutions triviales à ce problème. L'une d'elles est de proposer r = `.*`. Cette solution correspond à s. Cependant, elle ne capture probablement pas l'intention de la regex originale, car elle correspond à des chaînes de caractères arbitraires. C'est pourquoi une des entrées du problème est un ensemble d'exemples négatifs E^-. La regex doit être généralisée, mais à un point tel qu'elle ne correspond pas à trop de mots de E^-. La définition utilise un petit ensemble S et un grand ensemble E^- car il n'est pas facile de fournir un grand nombre d'exemples positifs : ce sont les mots que la regex conçue à la main n'accepte pas - mais devrait accepter - et ils sont généralement peu nombreux. En revanche, il est un peu plus facile de fournir une série d'exemples négatifs. Il suffit de fournir un document qui ne contient pas les mots cibles (comme un article d'encyclopédie lorsqu'on souhaite extraire les numéros de téléphone). Toutes

les chaînes de caractères de ce document peuvent être utilisé dans E^-, comme nous le montrons dans nos expériences.

Une autre solution triviale à notre problème est de dire $r' = r|s$. Cependant, cette solution ne capture pas non plus l'intention de la regex. Dans l'exemple, la regex r' = `[0-9]+|12 34 56` correspond bien à s, mais elle ne correspond à aucune autre séquence de nombres et d'espaces. Par conséquent, le but est de *généraliser* la regex d'entrée de manière appropriée, c'est-à-dire de trouver une "bonne" regex qui ne se spécialise pas trop ni ne sur-généralise. Pour quantifier la qualité d'une regex, nous suivons la même approche que [Babbar and Singh, 2010; Murthy et al., 2012] et évaluons la précision, le rappel et la mesure F1 sur un ensemble de tests.

Trouver les correspondances

Notre objectif est de modifier la regex de manière à ce qu'elle corresponde toujours à toutes les chaînes de caractères précédemment acceptées. Notre stratégie est de limiter les modifications de la regex au minimum. Pour cela, nous faisons correspondre la chaîne de caractères à la regex de manière approximative. La figure 5.3 page 70 en donne un exemple. Cela nous permet de savoir quelles parties (sous-chaînes de caractères) de la chaîne de caractères correspondent déjà. Les parties qui correspondent peuvent alors être conservées intactes ! L'ensemble des parties non appariées de la chaîne de caractères est transmis à l'étape de réparation suivante de l'algorithme. Nous commençons par définir un *correspondant*, en utilisant la notation $f|_A$ pour indiquer la restriction de la fonction f au domaine A :

Definition (Correspondance). *Étant donné une chaîne de caractères s et une regex r, une* correspondance *est une fonction partielle m des indices de la chaîne de caractères* $\{1, ..., |s|\}$, *aux feuilles de l'arbre syntaxique* $T(r)$ *de r . Soit* $I = \{i_1, ..., i_n\} \subseteq \{1, ..., |s|\}$ *le domaine de m, où* $i_1 < \cdots < i_n$. *La fonction m est une correspondance, si et seulement si l'une des conditions suivantes s'applique :*

— *r est un caractère ou une classe de caractères et* $I = \emptyset$, *ou* $I = \{i\}$, $m(i) = r$ *et* $s_i \in L(r)$

— $r = pq$ *et il existe un j tel que* $m|_{i_1,...,i_j}$ *est une correspondance pour p et* $m|_{i_{j+1},...,i_n}$ *est une correspondance pour q*

— $r = p|q$ *et m est une correspondance pour p ou pour q*

— $r = p*$ *et* $\exists k_1, \ldots, k_j : k_1 = i_1 \wedge \{k_1 < \cdots < k_j\} \wedge \{k_j = i_n + 1\} \wedge \forall l \in \{1, \ldots, j-1\} :$ $m|_{k_l,...,(k_{l+1})-1}$ *est une correspondance pour p*

Une correspondance est *maximale*, s'il n'y a pas d'autre correspondance définie sur davantage de positions de la chaîne de caractères. Pour trouver une correspondance maximale avec quelques lacunes, nous utilisons l'algorithme de programmation dynamique développé par Myers et al. [Myers and Miller, 1989]. La figure 5.3 page 70 montre une

correspondance maximale pour la regex `ab * c` et la chaîne de caractères (que la regex n'accepte pas) `aabbdc`.

B.4.2 Algorithme simple

Dans cette section, nous décrivons notre premier algorithme pour résoudre le problème décrit dans la section 5.3.2 page 68 : Étant donné une regex r, un ensemble de chaînes de caractères S, notre objectif est d'étendre r pour qu'elle corresponde aux chaînes de caractères de S. Le tableau 5.1 page 72 montre un exemple avec une seule chaîne de caractères $s \in S$, et aucun exemple négatif. Nous supposons que la regex a été réécrite pour éliminer le sucre syntaxique, de sorte qu'elle ne contient que des étoiles de Kleene, des disjonctions et des concaténations.

Notre algorithme est présenté dans l'algorithme 6 page 73. L'algorithme insère les parties non correspondantes de la chaîne de caractères aux endroits appropriés de la regex, et rend facultatives les parties non correspondantes de la regex. Tout d'abord, nous marquons le début et la fin de la regex (Lignes 1-2) pour simplifier le traitement des limites. Nous faisons une copie de la regex (Ligne 3) pour le cas où nous devons annuler la réparation. Ensuite, nous parcourons l'ensemble des exemples positifs S (Ligne 4) et trouvons une correspondance maximale (Ligne 5). Nous réparons la regex r pour s avec l'algorithme 7 page 74 (ligne 6). Nous vérifions si la regex réparée fonctionne bien sur l'ensemble des exemples négatifs (Ligne 8). Si la regex d'origine a moins de faux positifs, nous produisons une disjonction de la regex originale et des mots manquants (Ligne 9). Enfin, nous supprimons les caractères que nous avons ajoutés dans les Lignes 1-2.

B.4.3 Algorithme adaptatif

Dans cette section, nous présentons une version plus élaborée de l'algorithme précédent, qui incorpore les idées suivantes : nous n'élargissons pas les quantificateurs $A\{\ldots\}$, mais les traitons comme leur propre type de nœud dans l'arbre de syntaxe ; nous regroupons les modifications et les exécutons en même temps ; nous vérifions les endroits possibles pour ajouter une partie manquante à l'aide d'une disjonction et acceptons une telle disjonction si elle ne fait pas diminuer la précision ; et nous collectons les chaînes de caractères où l'introduction de disjonctions détériorerait la précision, déduirait les regex généralisées à partir de ces chaînes de caractères et ajouterait plutôt les regexes généralisées.

Le schéma général de notre méthode est présenté dans l'algorithme 8 page 75. Il prend comme paramètre d'entrée supplémentaire un seuil α. Ce seuil indique à combien d'exemples négatifs la regex réparée est autorisée à correspondre. Des valeurs plus élevées pour α permettent une réparation plus agressive. Dans ce cas, la regex réparée correspond à davantage d'exemples négatifs. À l'invers, $\alpha = 1$ rend l'algorithme plus

conservateur. Dans ce cas, la méthode tente de faire correspondre au maximum autant d'exemples négatifs que la regex d'origine. L'algorithme se déroule en 4 étapes, que nous allons maintenant présenter en détail.

Algorithm 13 Réparer les regexes (algorithme adaptatif)

INPUT: regex r, ensemble de chaînes de caractères S, exemples négatifs E^-, seuil $\alpha \geq 1$
OUTPUT: regex modifiée r
1: $M \leftarrow \bigcup_{s \in S} \mathit{findMatching}(r, s)$
2: $gap \leftarrow \bigcup_{m \in M} findGaps(m)$
3: $\mathit{findGapOverlaps}(r, gap)$
4: $\mathit{addMissingParts}(r, S, gap, E^-, \alpha)$

Étape 1 : Recherche des correspondances. Pour chaque mot $s \in S$, notre algorithme trouve les correspondances maximales (Algorithme 13, Ligne 1, Méthode *findMatching*). Les correspondances maximales pour tous les mots de S sont collectées dans un ensemble M.

Étape 2 : Trouver les lacunes. La correspondance nous indique quelles parties de la regex correspondent à la chaîne de caractères. Pour corriger la regex, nous souhaitons identifier les parties qui *ne* correspondent *pas*. À cette fin, nous introduisons une structure de données pour les lacunes dans la chaîne de caractères (c.-à-d. pour les parties de la chaîne de caractères qui ne sont pas mappées sur la regex).

Etape 3 : Recherche de chevauchements des lacunes. L'algorithme 9 page 77 prend en entrée la regex r et l'ensemble des lacunes $gaps$. Il parcourt récursivement la regex et traite chaque nœud de la regex. Nous divisons les quantificateurs $r\{min, max\}$ avec $max < 100$ en $r\{\ldots\}r\{\ldots\}$, si la lacune se produit entre différents cycles de $r\{min, max\}$. Pour d'autres quantificateurs, pour les étoiles de Kleene et pour les disjonctions, nous descendons récursivement dans l'arbre de la regex (Ligne 6).

Pour les nœuds de concaténation, nous déterminons tous les intervalles qui ont leur début ou leur fin dans la concaténation (Ligne 9). Nous déterminons ensuite les limites du partitionnement (Ligne 10 ; $l \in r$ signifie que la regex r a une feuille l). Nous considérons chaque lacune g (Ligne 11). Nous trouvons si le début ou la fin d'un autre intervalle tombe dans g. Cela ne concerne que les limites entre les indices enfants s et e de la concaténation $c_1 \ldots c_n$ (Lignes 12-13). Nous partitionnons la sous-séquence de concaténation $c_s \ldots c_{e-1}$ en coupant aux limites (Ligne 14). Enfin, la méthode est appelée récursivement sur les enfants de la concaténation qui contiennent le début ou la fin de toute lacune (Lignes 16-18).

Etape 4 : Ajout de parties manquantes. L'étape précédente nous a donné, pour chaque lacune, un ensemble de partitions possibles. Dans notre exemple de regex `01234567`, le mot `012y7` et la lacune `3456`, nous avons obtenu le partitionnement `34 | 56`. Cela signifie que `34` et `56` doivent devenir facultatifs dans la regex, et que nous pouvons insérer la sous-chaîne `y` comme alternative à l'une d'entre elles : `012(34|y)(56)?7` ou `012(34)?(56|y)7`. L'algorithme 10 page 78 prend cette décision en fonction de la solution la plus performante sur l'ensemble E^- d'exemples négatifs. Il se peut également qu'aucune de ces solutions ne soit permise, car elles correspondent toutes à trop d'exemples négatifs. Dans ce cas, il est préférable d'ajouter le mot comme une disjonction à la regex d'origine, comme dans `01234567|012y7`. Pour prendre ces décisions, l'algorithme compare le nombre d'exemples négatifs correspondant à la regex réparée avec le nombre d'exemples négatifs correspondant à la regex d'origine. Le rapport de ces deux quantités devrait être limité par le seuil α.

L'algorithme 10 page 78 prend en entrée une regex r, un ensemble de lacunes gap avec des partitions, des exemples négatifs E^- et un seuil α. L'algorithme commence par copier la regex original (Ligne 1) et traite chaque lacune (Ligne 2). Pour chaque lacune, il considère toutes les parties (Ligne 3). Dans l'exemple, nous considérons la partie `34` et la partie `56` de la lacune `3456`. L'algorithme transforme la partie en une disjonction de la partie et du passage de la lacune. Dans l'exemple, la partie `34` est transformée en `(34 | y)` (Ligne 4). Si le nombre d'exemples négatifs correspondants ne dépasse pas le nombre d'exemples négatifs correspondant à la regexes initiale multiplié par α (Ligne 5), l'algorithme choisit cette réparation et cesse d'explorer les autres parties de la lacune (Lignes 6-7). En Ligne 11, l'algorithme recueille tous les exemples positifs qui ne sont toujours pas associés. Les modifications apportées à ces mots sont annulées (Ligne 12). La Ligne 13 généralise ces mots en une ou plusieurs regexes. L'algorithme vérifie alors si la regex obtenue de cette façon est suffisamment bonne (Ligne 15). Si tel est le cas, la regex est ajoutée en tant que disjonction (Ligne 16). Sinon, les mots qui ont contribué à ce groupe sont ajoutés de manière disjonctive (Lignes 18-19).

La généralisation est adaptée de [Babbar and Singh, 2010]. Tout d'abord, nous assignons une clé de groupe à chaque mot. La clé est obtenue en substituant des sous-chaînes de caractères constituées uniquement de chiffres par un (unique) `\d`, des caractères minuscules ou majuscules avec un `[a-z]` ou un `[A-Z]`, et les caractères restants avec la classe de caractères `\W` ou `\w`. Enfin, nous obtenons le groupe regex r en ajoutant $\{min, max\}$ après chaque classe de caractères, de sorte que r correspond à toutes les chaînes de caractères de ce groupe.

B.4.4 Expériences

Mesure F1. Le Tableau 5.3 page 82 montre la mesure F1 sur tous les jeux de données pour différents algorithmes : la regex originale, la ligne de base de la disjonction, la référence en étoile, la méthode de [Babbar and Singh, 2010], l'algorithme simple et l'algorithme adaptatif avec des valeurs différentes pour α. Le tableau montre l'amélioration de la mesure $F1$ par rapport à la ligne de base, en points de pourcentage. Par exemple, pour *ReLIE / phone* et $\alpha = 1,2$, l'algorithme adaptatif atteint une valeur F1 de 82,1 % + 2,3 % = 84,4 %. Des résultats détaillés sur le rappel et la précision peuvent être trouvés dans la table 5.4 page 82 et la table 5.5 page 83. Nous avons vérifié que les mesures F1 sont significatives avec un test de micro-signe [Yang and Liu, 1999]. Nous pouvons voir que, pour presque toutes les tâches et tous les paramètres, nos algorithmes dépassent les regex d'origine ainsi que la base de référence.

Longueur de la regex. Jusqu'à présent, nous avons montré que notre méthode permet d'obtenir une valeur F1 comparable et généralement meilleure que les bases de référence. Nous examinons maintenant la longueur des regexes réparées pour voir à quel point les regexes réparées sont proches de la regex originale. La table 5.6 page 83 indique la longueur moyenne des regexes générées (en nombre de caractères). L'approche simple produit des regexes plus longues car elle rejette souvent la regex réparée et revient à ajouter les mots manquants en tant que disjonction. Par conséquent, la longueur des regexes de l'approche simple est plus proche de la longueur de la base de référence que de la longueur de la regex.

La valeur réelle de l'algorithme adaptatif provient toutefois des regexes beaucoup plus courtes qu'il génère. Pour l'algorithme adaptatif, la longueur dépend de α : si la valeur est grande, l'algorithme a tendance à intégrer les mots dans la regex d'origine. Ensuite, les mots ne sont plus soumis au mécanisme de généralisation. Néanmoins, l'impact de α est marginal : quelle que soit la valeur, notre algorithme produit des regexes jusqu'à 8 fois plus courtes que la base de référence, et presque toujours au moins deux fois plus courtes que l'algorithme concurrent - à une précision et un rappel comparables ou meilleurs. Nous concluons donc que les regexes produites par l'algorithme adaptatif sont mieux adaptées aux tâches étudiés que celles produites par les algorithmes concurrents.

B.4.5 Résumé

Dans ce chapitre, nous avons proposé un algorithme permettant d'ajouter des mots manquants à une regex donnée. Avec seulement un petit nombre d'exemples positifs, notre méthode généralise la regex d'entrée, tout en maintenant sa structure. De cette

façon, notre approche améliore la précision et le rappel de la regex originale. Notre méthode conserve les parties de la regex qui correspondent déjà aux mots d'entrée. Il ajoute des alternatives aux points droits de la regex, afin de correspondre aux sous-chaînes de caractères qui n'étaient pas encore couvertes.

Nous avons évalué notre méthode sur différents jeux de données et nous avons montré que déjà, avec un très petit nombre d'exemples positifs, nous pouvons améliorer la mesure F1 sur le test de référence. C'est un résultat surprenant, car il montre que les regexes peuvent être généralisées sur la base de très petites données d'entraînement. De plus, l'algorithme adaptatif produit des regexes beaucoup plus courtes que la base de référence, les approches concurrentes et l'algorithme simple. Cela montre que notre méthode généralise les regexes de manière significative et non triviale.

B.5 Répondre aux requêtes avec Unix Shell

Comme nous l'avons vu dans les chapitres précédents, certaines bases de connaissances contiennent un grand nombre de faits. Traiter de tels jeux de données nécessite souvent un prétraitement important. Ce prétraitement ne se produit qu'une seule fois, de sorte que le chargement et l'indexation des données dans une base de données ou un triplestore peuvent être excessifs. Dans ce chapitre, nous présentons une approche qui permet de prétraiter de grandes données tabulaires dans Datalog sans indexer les données. La requête Datalog est traduite en Bash Unix et peut être exécutée dans un shell. Nos expériences montrent que, pour le cas d'utilisation du prétraitement des données, notre approche est compétitive par rapport aux systèmes de pointe en termes d'évolutivité et de vitesse, tout en ne nécessitant qu'un shell Bash, et un système d'exploitation compatible Unix. Nous fournissons également un outil convertinant une partie de SPARQL et OWL2 vers Datalog pour rendre notre système interopérable avec les standards du Web sémantique.

B.5.1 Exemple

B.5.2 Préliminaires

Datalog. Nous suivons la définition de Datalog avec négation de [Abiteboul and Hull, 1988; Abiteboul et al., 1995]. Un programme *Datalog Program* est un ensemble de règles. Un *règle* prend la forme

$$H :- B_1, \ldots, B_n, \neg N_1, \ldots, \neg N_m.$$

Ici, H est *l'atome de tête*, et $B_1, \ldots, B_n$ sont les atomes de *corps* présents et $N_1, \ldots, N_m$ sont les atomes de corps niés. Nous nous limitons aux règles sûres et aux *programmes stratifiés* [Abiteboul and Hull, 1988; Abiteboul et al., 1995].

Algèbre relationnel. Un *table* est un ensemble de tuples de la même arité. Nous écrivons $arity(\cdot)$ pour l'arité d'un tuple ou l'arité des tuples d'un ensemble. Nous appelons *algèbre relationnelle SPJAUR* l'ensemble d'opérateurs suivant [Abiteboul et al., 1995] : sélection σ, projet π, jointure $\bowtie$, anti-jointure $\triangleright$, union $\cup$, et le plus petit point fixe LFP (least fixed point). Pour une fonction f d'une table à une table, $LFP(f)$ est le plus petit point fixe de f pour la relation $\subseteq$. Le plus petit point fixe peut être calculé avec l'algorithme semi-naïf [Abiteboul et al., 1995], comme indiqué dans l'algorithme 11 page 96.

Unix. Unix est une famille de systèmes d'exploitation multitâches. Pour le présent travail, nous nous intéressons uniquement aux flux d'octets *séparés par tabulation*, c'est-à-dire les flux d'octets composés de plusieurs *lignes* (séquences d'octets séparées par un caractère de nouvelle ligne), constitués chacun du même nombre de *colonnes* (séquences d'octets séparées par un caractère tabulateur). Lorsqu'ils sont imprimés, ces flux d'octets ressemblent à une table.

Le shell Bourne-again (Bash) est une interface de ligne de commande pour les systèmes d'exploitation de type Unix. C'est le shell interactif par défaut pour les utilisateurs sur la plupart des systèmes Linux et MacOS [Wikipedia, 2017]. Une commande *Bash* est soit un mot-clé intégré, soit un petit programme. Nous sommes ici principalement concernés par les commandes du standard POSIX qui prennent en entrée un ou plusieurs flux d'octets et qui produisent un flux d'octets en imprimant sur la sortie standard. Nous utiliserons les commandes suivantes : `cat`, `sort`, `comm`, `join`, `echo`, `mv`, `awk`. Nous utilisons également des pipes, principalement sous la forme de *substitution de processus* : $command_1$ <($command_2$). Cette construction exécute le processus $command_2$ et dirige son flux de sortie dans le premier argument du processus $command_1$. .

B.5.3 Approche

B.5.3.1 Dialecte Datalog

Pour nos besoins, le programme Datalog doit faire référence à des fichiers ou à des flux d'octets de données. Pour cette raison, nous introduisons un type supplémentaire de règles, que nous appelons *règles de commande*. Une règle de commande prend la forme suivante :

> $p(x_1, ..., x_n)$:~ c

Sémantiquement, cette règle signifie que l'exécution de c produit un flux d'octets séparés par tabulation de n colonnes, qui sera désigné par le prédicat p du programme Datalog. Notre approche peut également fonctionner en «mode RDF». Dans ce mode, l'entrée consiste en une requête SPARQL [Harris et al., 2013], un TBox sous la forme de OWL 2 RL [Cao et al., 2011] et un ABox sous la forme d'un fichier N-Triples F (voir la section 2.2.1 page 23). Nous convertissons l'ontologie OWL et la requête SPARQL en règles Datalog et nous incluons une commande AWK dans le script Bash pour transformer le fichier F en fichier TSV.

B.5.3.2 Chargement du datalog

Notre approche prend en entrée un programme Datalog et produit en sortie un script Bash Shell. Pour ce faire, notre approche construit d'abord une expression algébrique relationnelle pour le prédicat `main` du programme Datalog avec Algorithm 12 page 102.

B.5.3.3 Production des commandes Bash

Maintenant, nous traduisons cette expression en commande Bash par la fonction b, définie comme suit :

$b([c]) = c$
: Une expression de la forme $[c]$ est déjà une commande Bash, et nous pouvons donc retourner directement c.

$b(e_1 \cup \ldots \cup e_n)$
: Pour supprimer les doublons possibles, nous traduisons une union en

```
sort -u <(b(e1)) ... <(b(en))
```

$b(e_1 \bowtie_{x=y} e_2)$
: Une jointure de deux expressions e_1 et e_2 sur une seule variable à la position x et y respectivement, mène à la commande

```
join -t$'\t'  -1x -2y \
    <(sort -t$'\t'  -kx <(b(e1))) \
    <(sort -t$'\t'  -ky <(b(e2)))
```

Cette commande trie les flux d'octets de $b(e_1)$ et $b(e_2)$, puis les joint dans la colonne commune.

$b(e_1 \bowtie_{x=y,\ldots} e_2)$
: La commande Bash `join` peut effectuer la jointure sur une seule colonne. Si nous

voulons joindre plusieurs colonnes, nous devons ajouter une nouvelle colonne à chacun des flux d'octets. Cette nouvelle colonne concatène les colonnes de jointure en une seule colonne. Cela peut être réalisé avec le programme AWK suivant, que nous exécutons à la fois sur $b(e_1)$ et $b(e_2)$:

```
{ print $0 FS $j1 s $j2 s ... s $jn }
```

Ici, les indices $j_1, \ldots, j_n$ sont les positions des colonnes de jointure dans le flux d'octets d'entrée, et s est le caractère de séparation (voir la section 6.4 page 95). Une fois que nous l'avons fait avec les deux flux d'octets, nous pouvons les joindre sur cette nouvelle colonne de la même manière que celle décrite ci-dessus pour les jointures simples. Cette jointure supprime également la colonne supplémentaire.

$b(e_1 \rhd_x e_2)$
: Juste comme une jointure régulière, un anti-jointure devient une commande `join`. Nous utilisons le paramètre `-v1`, de sorte que la commande ne génère que les tuples émergeant de e_1 qui ne peuvent pas être joints à ceux de e_2. Nous traitons les anti-jointures sur plusieurs colonnes de la même manière qu'avec les jointures multi-colonnes.

$b(\pi_{i_1,\ldots i_n}(e))$
: Une projection devient le programme AWK suivant, que nous exécutons sur $b(e)$:

```
{ print $i1 FS ... FS $in }
```

$b(\pi_{i:a}(e))$
: Une introduction constante devient le programme AWK suivant, que nous exécutons sur $b(e)$:

```
{ print $1 FS ... $(i-1) FS a FS $i FS ... $n }
```

$b(\sigma_{i=v}(e))$
: Un noeud de sélection donne lieu au programme AWK suivant, que nous exécutons sur $b(e)$:

```
$i == "v" { print $0 }
```

Cette commande peut être généralisée facilement à une sélection sur plusieurs colonnes.

B.5.3.4 Récursivité

Nous venons de définir la fonction b qui traduit une expression algèbre relationnelle en une commande Bash. Nous allons maintenant voir comment définir b pour le cas de la récursivité. Un nœud $LFP(f)$ devient

```
echo -n > delta.tmp;  echo -n > full.tmp
while
    sort b(f(delta.tmp)) | comm -23 - full.tmp > new.tmp;
    mv  new.tmp  delta.tmp;
    sort -u -m -o full.tmp  full.tmp <(sort delta.tmp);
    [ -s delta.tmp ];
do continue; done
cat full.tmp
```

Ce code utilise 3 fichiers temporaires pour calculer le moindre point de correction de f : `full.tmp` contient tous les faits inférés jusqu'à l'itération en cours. `delta.tmp` contient les faits nouvellement ajoutés d'une itération. `new.tmp` est utilisé comme fichier d'échange. La boucle s'exécute alors que de nouvelles lignes sont encore ajoutées. Si aucune nouvelle ligne n'a été ajoutée, le code quitte la boucle et imprime tous les faits. Notez qu'en raison de la monotonie de nos opérateurs d'algèbre relationnelle et de la stratification de nos programmes, nous ne pouvons nous permettre d'exécuter f que sur les lignes nouvellement ajoutées.

B.5.3.5 Matérialisation

Pour éviter de recalculer une expression d'algèbre relationnelle déjà calculée, nous matérialisons des calculs intermédiaires. Pour cela, nous introduisons un nouveau type d'opérateur à l'algèbre, le nœud de matérialisation. Un noeud de matérialisation $⧉(m, (\lambda y : p_{m \to y}))$ a deux sous-plans : m est le plan utilisé plusieurs fois et que nous matérialiserons. La fonction $(\lambda y : p_{m \to y})$ est le plan principal et prend le plan matérialisé comme paramètre. Le plan $p_{m \to y}$ est le plan original p avec toutes les occurrences de m remplacées par la variable y.

Un sous-plan m sera matérialisé s'il devait être évalué plus d'une fois lors de l'exécution du plan algébrique. Le code bash d'un nœud de matérialisation stocke le résultat dans un fichier temporaire t. Un mécanisme de verrouillage permet d'exécuter la matérialisation en parallèle avec d'autres commandes.

B.5.3.6 Optimisation

Nous appliquons les optimisations habituelles sur nos expressions algébriques relationnelles : nous poussons les nœuds de sélection le plus près possible de la source ; nous fusionnons des unions ; nous fusionnons des projections ; nous appliquons une simple commande de réassociation. De plus, nous supprimons le LFP quand il n'y a pas de récursivité ; et nous extrayons d'un nœud LFP la partie non récursive du plan interne (de sorte qu'elle ne soit calculée qu'une seule fois au début du calcul du point fixe).

Dans le programme Datalog, un appel récursif d'un prédicat se produit, si le prédicat se prend comme entrée (éventuellement avec des règles de médiation). Dans l'algèbre, un appel récursif correspond au x d'un nœud LFP. Nous supprimons les appels récursifs lorsqu'ils ne fournissent pas de nouvelles sorties. Nous appelons ces appels «superflus».

Dans Datalog, c'est le cas si une règle contient l'atome principal dans le corps (avec les mêmes variables dans le même ordre). Nous pouvons ignorer la règle en toute sécurité. Nous appliquerons ce principe sur le plan de l'algèbre.

Nous déterminons si un appel récursif x est superflu, en traçant ses colonnes jusqu'au nœud LFP. S'ils arrivent complètement, et dans l'ordre, au nœud LFP, toutes les sorties calculées dans les itérations précédentes passeraient simplement par le chemin de x au nœud LFP. Cela signifie que nous pouvons supprimer x du plan algébrique.

B.5.4 Expériences

Pour montrer la viabilité de notre approche, nous avons utilisé notre méthode sur plusieurs ensembles de données et l'avons comparée à plusieurs concurrents. Toutes les expériences ont été exécutées sur un ordinateur portable avec Ubuntu 16,04, un processeur Intel Core i7-4610M 3,00 GHz, 16 Go de mémoire et 3,8 To d'espace disque. Nous avons utilisé GNU coreutils 8.25 pour les commandes POSIX et mawk 1.3.3 pour AWK.

B.5.4.1 Lehigh University Benchmark

Dataset. Notre premier jeu de données est le Benchmark (LUBM) de l'Université Lehigh [Guo et al., 2005]. LUBM est un ensemble de données standard pour les référentiels Web sémantiques. Il modélise les universités, leurs employés, leurs étudiants et leurs cours. L'ensemble de données est paramétré par le nombre d'universités et sa taille peut donc varier. LUBM est livré avec 14 requêtes qui testent différents modèles d'utilisation. Ces requêtes sont exprimées en SPARQL. Pour nos besoins, nous avons traduit les requêtes en Datalog.

Compétiteurs. Nous comparons notre approche aux groupes de concurrents suivants :

— systèmes basés sur les journaux de données : DLV, RDFox

— triplestores : Jena, Jena avec HDT, Stardog, Virtuoso

— systèmes de gestion de base de données : Postgres, NoDB, MonetDB

— systèmes de données : RDFSlice

Pour les systèmes de base de données (Postgres, NoDB et MonetDB), nous avons traduit les requêtes en SQL et ajouté la TBox dans la requête. Nous avons lancé tous les concurrents sur toutes les requêtes qu'il prend en charge, et nous avons effectué une moyenne de 3 exécutions. Comme la plupart des systèmes se terminent en quelques secondes, nous avons interrompu les systèmes qui ont duré plus de 10 minutes. Les bases de données ont été exécutées avec et sans indexation. NoDB est livré avec son propre mécanisme d'indexation adaptatif qui ne peut pas être désactivé.

LUBM10. La table 6.1 page 114 montre les temps d'exécution de toutes les requêtes pour les différents systèmes sur LUBM avec 10 universités. Les temps d'exécution incluent les temps de chargement et d'indexation. Pour les systèmes où nous pouvions déterminer ces temps explicitement, nous les avons notés dans la dernière ligne du tableau. Parmi les 4 triplestores (Jena + TDB, Jena + HDT, Stardog et Virtuoso), seul Stardog peut terminer toutes les requêtes en moins de 10 minutes. Jena est la plupart du temps trop lent, peu importe avec quel back-end. Virtuoso fonctionne légèrement plus vite que Stardog, mais il ne peut pas traiter toutes les requêtes. Les triplestores sont généralement plus lents que nos 5 concurrents de bases de données (Postgres, NoDB et MonetDB - avec et sans index). Parmi ceux-ci, nous constatons que MonetDB est beaucoup plus rapide que Postgres et NoDB. Postgres et MonetDB sont les plus rapides sans index, ce qui est prévisible avec de si petits ensembles de données.

Les systèmes les plus performants sont clairement les 3 systèmes Datalog (Bash Datalog, DLV et RDFox). Non seulement ils peuvent répondre à toutes les requêtes, mais ils sont généralement plus rapides que les autres systèmes. Parmi les trois, DLV est le plus lent. RDFox brille avec un temps très court et presque constant pour toutes les requêtes. Nous soupçonnons que ce temps est donné par le temps de chargement des données et qu'il domine le temps de calcul de la réponse. Néanmoins, Bash Datalog est plus rapide que RDFox sur presque toutes les requêtes sur LUBM 10.

LUBM100 à LUBM1000. Sur la base de l'expérience précédente, nous avons choisi comme concurrents les systèmes les plus rapides de chaque groupe : RDFox pour les systèmes Datalog, Stardog et Virtuoso pour les triplestores et MonetDB avec et sans index pour les bases de données. Nous avons ensuite augmenté le nombre d'universités de notre ensemble de données LUBM de 10 à 100, 200, 500 et 1000. Le tableau 6.2

page 117 et 6.3 page 118 montre les tailles des ensembles de données et les temps d'exécution des systèmes. Sur tous les jeux de données, notre système fonctionne mieux sur plus de la moitié des requêtes. RDFox est le seul système capable de réaliser des performances similaires. Comme auparavant, RDFox a toujours besoin d'un temps constant pour répondre à une requête, car il charge le jeu de données dans la mémoire principale. Cela rend le système très rapide. Cependant, cette technique ne fonctionne pas si le jeu de données est trop grand, comme nous le verrons ensuite.

B.5.4.2 Accessibilité

Datasets. Nos prochains jeux de données sont des jeux de données graphiques. Nous avons utilisé les graphes LiveJournal et com-orkut de [Leskovec and Krevl, 2014] et le graphe friendster [Kunegis, 2013]. Ces ensembles de données représentent la structure graphique des réseaux sociaux en ligne. Ils nous permettent de tester les performances de notre algorithme sur des données réelles. Tableau 6.4 indique le nombre de noeuds et les arêtes de ces jeux de données.

En tant que concurrents, nous avons de nouveau choisi RDFox, Stardog et Virtuoso. Nous n'avons pas pu utiliser MonetDB, car la requête d'accessibilité est récursive. En tant que concurrent supplémentaire, nous avons choisi BigDatalog [Shkapsky et al., 2016]. BigDatalog est une implémentation de Datalog distribuée exécutée sur Apache Spark. BigDatalog était déjà exécuté sur les mêmes graphiques LiveJournal et com-orkut dans le document original [Shkapsky et al., 2016].

Query. Pour tous ces jeux de données, nous avons utilisé une seule requête : nous avons demandé l'ensemble des noeuds pouvant être atteints à partir d'un noeud donné *id*. Nous avons utilisé le programme Datalog suivant à cette fin, adapté de [Shkapsky et al., 2016] :

```
reach(Y) :- arc(id, Y).
reach(Y) :- reach(X), arc(X, Y).
```

Afin d'éviter que RDFox ne matérialise la totalité de la fermeture transitive, nous avons modifié le programme comme suit pour RDFox :

```
reach(id, Y) :- arc(id, Y).
reach(id, Y) :- reach(id, X), arc(X, Y).
```

Pour l'expérience, nous avons choisi 3 nœuds aléatoires (et donc généré 3 requêtes) pour LiveJournal et com-orkut. Nous avons choisi un nœud aléatoire pour Friendster.

Results. Tableau 6.5 affiche le temps d'exécution pour chaque système (en moyenne sur les 3 requêtes pour LiveJournal et com-orkut). Virtuoso était le système le plus lent, et nous l'avons abandonné après 25 min et 50 min respectivement. Nous ne l'avons pas exécuté sur le jeu de données Friendster, car Friendster est 20 fois plus gros que les deux autres jeux de données. Stardog fonctionne mieux. Cependant, nous avons dû l'abandonner après 10 heures sur le jeu de données Friendster. BigDatalog fonctionne bien, mais échoue avec une erreur d'espace insuffisant sur le jeu de données Friendster. Le système le plus rapide est RDFox. C'est parce qu'il peut charger toutes les données en mémoire. Cette approche échoue cependant avec le jeu de données Friendster. Il ne rentre pas dans la mémoire et RDFox ne peut pas s'exécuter. Bash Datalog s'exécute 50 % plus lentement que RDFox. En retour, c'est le seul système capable de terminer dans un délai raisonnable sur le jeu de données Friendster (4,5 h). Nous pensons que c'est parce que Bash Datalog peut s'appuyer sur les implémentations hautement optimisées des commandes Bash, qui peuvent traiter des fichiers volumineux même s'ils ne peuvent pas être chargés en mémoire.

B.5.4.3 YAGO et Wikidata

Datasets. Notre dernière série d'expériences teste notre système sur des bases de connaissances. A cette fin, nous avons utilisé YAGO 3.1 et Wikidata [Vrandecic and Krötzsch, 2014]. Les données YAGO sont disponibles en 3 fichiers différents, l'un avec les 12 M faits (814 MB), l'autre avec la taxonomie avec 1,8 M faits (154 MB), et le dernier avec le 24 M relations de type (1,6 taille en Go). Le jeu de données simple Wikidata contient 2,1 milliards de triples et a une taille non compressée de 267 Go.

Queries. Nous avons conçu 4 requêtes typiques pour ces jeux de données (Tableau 6.6 page 120). Le tableau 6.7 page 120 montre le TBox que nous avons utilisé. Ces requêtes et la TBox sont légèrement adaptées au schéma différent des deux bases de connaissances. Query 1 demande toutes les sous-classes de la classe <Person>. Query 2 demande les parents de Louis XIV, et Query 3 demande récursivement les ancêtres de Louis XIV. Query 4 demande à toutes les personnes nées en Andorre. Ces requêtes ne sont pas difficiles. La difficulté vient du fait que les données sont très grandes.

Results. La table 6.6 page 121 affiche les résultats de RDFox et de notre système sur les deux jeux de données. Sur YAGO, RDFox est beaucoup plus lent que notre système, car il doit instancier toutes les règles pour répondre aux requêtes. Sur Wikidata, les données ne rentrent pas dans la mémoire principale et RDFox ne peut donc pas être exécuté du tout. Notre système, en revanche, s'adapte facilement aux grandes tailles de données.

On peut penser qu'un système de base de données, tel que Postgres, pourrait être mieux adapté à de tels ensembles de données volumineux. Ceci est cependant pas le cas. Postgres a mis 104 secondes pour charger le jeu de données YAGO et 190 secondes pour générer les index. Pendant ce temps, notre système a déjà répondu à presque toutes les requêtes.

Discussion. Toutes nos expériences n'évaluent que le scénario que nous considérons dans cette section, à savoir le scénario où l'utilisateur veut exécuter une seule requête afin de prétraiter les données. Ce sont les cas pour lesquels notre système est conçu. Nous tenons également à souligner une fois de plus que notre objectif n'est pas d'être plus rapide que tous les systèmes existants. Ce que nous voulons dire est que Bash Datalog peut prétraiter les données tabulaires sans avoir besoin d'installer un logiciel particulier. De plus, notre approche est compétitive en termes de rapidité et d'évolutivité par rapport à l'état de l'art.

B.5.5 Résumé

Dans cette séction, nous avons présenté une méthode pour compiler des programmes Datalog dans des scripts Unix Bash. Cela permet d'exécuter des requêtes Datalog sur des jeux de données tabulaires sans installer de logiciel. Nous montrons que notre méthode est compétitive en termes de vitesse avec les systèmes de pointe. De plus, notre système peut traiter des jeux de données même s'ils ne rentrent pas dans la mémoire.

B.6 Conclusion

Cette thèse a illustré comment étendre la base de connaissances YAGO et la rendre plus utile pour ses utilisateurs.

La base de connaissances YAGO. Nous avons décrit la base de connaissances YAGO. YAGO est l'une des premières bases de connaissances à grande échelle construites automatiquement. YAGO a une haute qualité, c'est-à-dire qu'il atteint une précision de plus de 95 %. Des évaluations manuelles sont effectuées pour chaque version majeure afin de valider la qualité de YAGO. Cependant, bien que YAGO contienne des millions de faits, il ne couvre encore qu'une petite partie des connaissances humaines.

Extension des informations sur les personnes dans YAGO. Nous avons abordé le problème de l'augmentation de la couverture des faits sur les personnes dans YAGO. Nous avons montré comment améliorer les algorithmes d'extraction d'informations en ajou-

tant de nouvelles heuristiques. De cette manière, nous avons extrait plus de dates de naissance et de décès, de lieux de résidence et de genre.

Nous avons utilisé la version améliorée de YAGO pour des études de cas historiques, notamment l'espérance de vie des hommes et des femmes au cours des siècles et l'âge des parents au moment de la naissance de leur premier enfant. Ces études se limitent aux personnes qui ont leur propre page Wikipedia, c'est-à-dire principalement des élites comme des personnes riches et célèbres. Néanmoins, il pourrait être utile d'analyser l'histoire d'un point de vue différent, plus statistique, et d'obtenir des informations sur les changements sociaux au cours des siècles.

Ajout de mots manquants aux expressions régulières. Nous avons montré comment adapter automatiquement une expression régulière pour qu'elle corresponde également à un ensemble de mots donné. Pour cela, nous n'avons besoin que d'un petit ensemble d'exemples positifs. La regex réparé devrait obtenir une meilleure mesure F1 et une précision similaire ou supérieure à celle du regex originale. Nous avons fourni deux algorithmes à cette fin. Les deux s'appuient sur l'appariement approximatif des regex pour rechercher les parties non correspondantes de la chaîne. Le premier algorithme, plus simple, descend dans l'arbre syntaxique regex et insère les parties non correspondantes comme alternatives. Le second algorithme adaptatif suit plusieurs insertions à la fois et vérifie en plus la qualité de toutes les modifications. Dans les expériences, les deux algorithmes étaient compétitifs avec l'état de l'art. L'algorithme adaptatif a produit des expressions rationnelles avec la meilleure mesure F1 de tous les systèmes et produit les expressions rationnelles les plus courtes.

Répondre aux requêtes avec le shell Unix. Dans ce chapitre, nous avons développé un système de traduction des requêtes de base de données en scripts Bash. Notre approche permet à un utilisateur d'exécuter des requêtes de base de données sur des fichiers TSV sur un système Unix, sans logiciel supplémentaire. Nous prenons en charge les requêtes Datalog ou SPARQL avec des contraintes OWL. Dans le cas des requêtes SPARQL, nous les avons d'abord traduites en Datalog. Nous avons transformé le programme Datalog en un plan d'opérateurs d'algèbre relationnelle, étendu avec un opérateur de type plus petit point fixe (LFP). Nous avons appliqué les optimisations habituelles d'algèbre relationnelle et décrit comment optimiser les plans contenant des LFP. Enfin, nous avons traduit le plan en un script Bash optimisé. Dans les expériences, nous avons montré que notre approche est comparable en termes de vitesse avec les systèmes de pointe.

Étendre la base de connaissances YAGO

Thomas Rebele

RESUME: Une base de connaissances est un ensemble de faits sur le monde. Parmi elles se trouve YAGO, une des premières à être générée automatiquement à grande échelle. Cette thèse se concentre sur l'extension de la base de connaissances YAGO en améliorant l'extraction de contenu et son accès.
La première contribution principale consiste en l'augmentation de la quantité des faits sur les personnes. Pour se faire, cette thèse décrit des algorithmes et des heuristiques permettant d'extraire d'avantage de dates de naissance et de décès, d'indications sur le sexe et de lieux de résidence. Ces données sont ensuite utilisées dans le cadre d'études en humanités numériques.
La deuxième contribution principale présente deux algorithmes permettant de réparer automatiquement une expression régulière afin qu'elle corresponde à un ensemble de mots donnés. Des expériences sur divers jeux de données montrent l'efficacité et la généralité de l'approche. Comparés aux travaux précédents, le rappel est amélioré tout en conservant une précision similaire, voire supérieure.
La dernière contribution est un système de traduction de requêtes sur des bases de données en scripts Bash. Cela permet de prétraiter des jeux de données en utilisant des requêtes Datalog et SPARQL sans installer de logiciel au-delà d'un système d'exploitation de type Unix. Les expériences montrent que les performances de notre approche sont comparables à celles des meilleures solutions du marché.

MOTS-CLEFS: Bases de connaissance, humanités numériques, expression régulière

ABSTRACT: A knowledge base is a set of facts about the world. YAGO was one of the first large-scale knowledge bases that were constructed automatically. This thesis focuses on extending the YAGO knowledge base along two axes: extraction and preprocessing.
The first main contribution of this thesis is improving the number of facts about people. The thesis describes algorithms and heuristics for extracting more facts about birth and death date, about gender, and about the place of residence. The thesis also shows how to use these data for studies in Digital Humanities.
The second main contribution are two algorithms for repairing a regular expression automatically so that it matches a given set of words. Experiments on various datasets show the effectiveness and generality of these algorithms. Both algorithms improve the recall of the initial regular expression while achieving a similar or better precision.
The last contribution is a system for translating database queries into Bash scripts. This approach allows preprocessing large tabular datasets and knowledge bases by executing Datalog and SPARQL queries, without installing any software beyond a Unix-like operating system. Experiments show that the performance of our system is comparable with state-of-the-art systems.

KEY-WORDS: Knowledge bases, Digital Humanities, regular expressions