HAL
open science

On the foundations for the compilation of web data queries : optimization and distributed evaluation of SPARQL

Louis Jachiet

To cite this version:

Louis Jachiet. On the foundations for the compilation of web data queries : optimization and distributed evaluation of SPARQL. Information Retrieval [cs.IR]. Université Grenoble Alpes, 2018. English. NNT : 2018GREAM038 . tel-01891444v3

HAL Id: tel-01891444

https://theses.hal.science/tel-01891444v3

Submitted on 19 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Louis JACHIET

Thèse dirigée par **Nabil LAYAIDA**
et codirigée par **Pierre GENEVES**, CNRS
préparée au sein du **Laboratoire Institut National de Recherche en Informatique et en Automatique**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Sur la compilation des langages de requêtes pour le web des données : optimisation et évaluation distribuée de SPARQL

On the foundations for the compilationof web data queries: optimization anddistributed evaluation of SPARQL.

Thèse soutenue publiquement le **13 septembre 2018**,
devant le jury composé de :

Monsieur NABIL LAYAÏDA
DIRECTEUR DE RECHERCHE, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Directeur de thèse
Monsieur DARIO COLAZZO
PROFESSEUR, UNIVERSITE PARIS-DAUPHINE, Rapporteur
Madame IOANA MANOLESCU
DIRECTRICE DE RECHERCHE, INRIA CENTRE SACLAY- ÎLE-DE-FRANCE, Rapporteur
Monsieur PIERRE GENEVES
CHARGE DE RECHERCHE, CNRS DELEGATION ALPES, Co-directeur de thèse
Monsieur JERÔME EUZENAT
DIRECTEUR DE RECHERCHE, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Président
Monsieur PATRICK VALDURIEZ
DIRECTEUR DE RECHERCHE, INRIA CENTRE S. ANTIPOLIS-MEDITERRANEE, Examinateur

Abstracts

Abstract

The topic of my PhD is the compilation of web data query languages. More particularly, the analysis and the distributed evaluation of a such language: SPARQL. My main contributions concern the evaluation of web data queries especially for recursive queries or for distributed settings.

In this thesis, I introduce μ-algebra: it is a kind of relational algebra equipped with a fixpoint operator. I present its syntax, semantics, and a translation from SPARQL with Property Paths (a new feature of SPARQL allowing some form of recursion) to this μ-algebra.

I then present a type system and show how μ-algebra terms can be rewritten to terms with equivalent semantics using either classical rewrite rules of the relational world or new rules that are specific to this μ-algebra. We demonstrate the correctness of these new rules that are introduced to handle the rewriting of fixpoints: they allow to push filters, joins and projections inside fixpoints or to combine several fixpoints (when some condition holds).

I demonstrate how these terms could be evaluated both from a general perspective and in the specific case of a distributed evaluation. I devise a cost model for μ-algebra terms inspired by this evaluation. With this cost model and this evaluator, several terms that are semantically equivalent can be seen as various Query Execution Plans (QEP) for a given query. I show that the μ-algebra and its rewrite rules allow the reach of QEP that are more efficient than all QEP considered in other existing approaches and confirm this by an experimental comparison of several query evaluators on SPARQL queries with recursion.

I investigate the use of an efficient distributed framework (Spark) to build a fast SPARQL distributed query evaluator. It is based on a fragment of μ-algebra, limited to operators that have a translation into fast Spark code. The result of this has been used to implement SparqlGX, a state of the art distributed SPARQL query evaluator.

Finally, my last contribution concerns the estimation of the cardinality of solutions to a μ-algebra term. Such estimators are key in the optimization. Indeed, most cost models for QEP rely on such estimators and are therefore necessary to determine the most efficient QEP. I specifically consider the conjunctive query fragment of μ-algebra (which corresponds to the well-known Basic Graph Pattern fragment of SPARQL). I propose a new cardinality estimation based on statistics about the data and implemented the method into SparqlGX. Experiments show that this method improves the performance of SparqlGX.

Résumé

Ma thèse porte sur la compilation des langages de requêtes orientés web des données. Plus particulièrement, ma thèse s'intéresse à l'analyse, l'optimisation et l'évaluation distribuée d'un tel langage : SPARQL.

Ma contribution principale est l'élaboration d'une méthode nouvelle particulièrement intéressante pour des requêtes contenant de la récursion ou dans le cadre d'une évaluation distribuée. Cette nouvelle méthode s'appuie sur un nouvel outil que nous introduisons : la μ-algèbre. C'est une variation de l'algèbre relationnelle équipée d'un opérateur de point fixe. Nous présentons sa syntaxe et sémantique ainsi qu'une traduction vers la μ-algèbre depuis SPARQL avec Property Paths (une fonctionnalité introduite dans le dernier standard SPARQL qui autorise une forme de récursion).

Nous présentons ensuite un système de types et nous montrons comment les termes de la μ-algèbre peuvent être réécrits en d'autres termes (de sémantique équivalente) en utilisant soit des règles de réécriture provenant de l'algèbre relationnelle soit des règles nouvelles, spécifiques à la μ-algèbre. Nous démontrons la correction des nouvelles règles qui sont introduites pour réécrire les points fixes : elles permettent de pousser les filtres, les jointures ou les projections à l'intérieur des points fixes (dépendant des certaines conditions sur le terme).

Nous présentons ensuite comment ces termes peuvent être évalués, d'abord de manière générale, puis en considérant le cas particulier d'une évaluation sur une plateforme distribuée. Nous présentons aussi un modèle de coût pour l'évaluation des termes. À l'aide du modèle de coût et de l'évaluateur, plusieurs termes qui sont équivalents d'un point de vue sémantiques peuvent maintenant être vus comme différentes manières d'évaluer les termes avec différents coûts estimés.

Nous montrons alors que les termes qui sont considérés grâce aux nouvelles règles de réécritures que nous avons introduites, permettent une exécution plus efficace que ce qui était possible dans les autres approches existantes. Nous confirmons ce résultat théorique par une expérimentation comparant plusieurs exécuteurs sur des requêtes SPARQL contenant de la récursion. Nous avons investigué comment utiliser une plateforme de calcul distribuée (Apache Spark) pour produire un évaluateur efficace de requêtes SPARQL. Cet évaluateur s'appuie sur un fragment de la μ-algèbre, limité aux opérateurs qui ont une traduction en code Spark efficace. Le résultat de ces investigations à résultat en l'implémentation de SPARQLGX, un évaluateur SPARQL distribué en pointe par rapport à l'état de l'art.

Pour finir, ma dernière contribution concerne l'estimation de la cardinalité des solutions à un terme de la μ-algèbre. Ces estimateurs sont particulièrement utiles pour l'optimisation. En effet, les modèles de coût reposent généralement sur de telles estimations pour choisir quel sera le terme le plus efficace parmi plusieurs termes équivalents. Pour cette estimation nous nous intéressons tout particulièrement au fragment conjonctif de la μ-algèbre (ce qui correspond au fragment bien connu Basic Graph Pattern de SPARQL). Notre nouvelle estimation de cardinalité s'appuie sur des statistiques sur les données et a été implémenté dans SPARQLGX. Nos expériences montrent que cette méthode permet de grandement accélérer l'évaluation de SPARQL sur SPARQLGX.

Publications

Published in peer reviewed international conferences

[SJMR18] **ProvSQL: Provenance and Probability Management in PostgreSQL.**
Pierre Senellart, Louis Jachiet, Silviu Maniu, Yann Ramusat.
Proceedings of the VLDB Endowment, Volume 11 Number 12.

[ABJM17] **A Circuit-Based Approach to Efficient Enumeration.**
Antoine Amarilli, Pierre Bourhis, Louis Jachiet, Stefan Mengel.
44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland.

[GJGL16a] **SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark.**
Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
The 15th International Semantic Web Conference ISWC, Oct 2016, Kobe, Japan.

[GJGL16b] **SPARQLGX in action: Efficient Distributed Evaluation of SPARQL with Apache Spark.**
Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
The 15th International Semantic Web Conference ISWC, Oct 2016, Kobe, Japan.

Published only in national conferences (peer reviewed)

[JGLG17] **Une nouvelle algèbre pour SPARQL permettant l'optimisation des requêtes contenant des Property Paths.**
Louis Jachiet, Pierre Geneves, Nabil Layaïda et Nils Gesbert
Poster at Bases de données et application 2017, Nancy, France

[GJGL17] **Une classification expérimentale multi-critères des évaluateurs SPARQL répartis.**
Damien Graux, Louis Jachiet, Pierre Genevès et Nabil Layaïda
Bases de données et application 2017, Nancy, France

[JGGL18] **On the optimization of recursive relational queries.**
Louis Jachiet, Nils Gesbert, Pierre Geneves & Nabil Layaïda
Bases de données et application 2018, Bucarest, Romania

Awaiting publication

- **The SPARQLGX System for Distributed Evaluation of SPARQL Queries.**
  Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda
  `https://hal.inria.fr/hal-01621480`

- **Optimizing sparql query evaluation with a worst-case cardinality estimation based on statistics on the data.**
  Louis Jachiet, Pierre Genevès, Nabil Layaïda
  `https://hal.archives-ouvertes.fr/hal-01524387`

- **An efficient translation from a modal μ-calculus over finite trees with converse to tree automata.**
  Louis Jachiet, Pierre Genevès, Nabil Layaïda
  `https://hal.archives-ouvertes.fr/hal-01117830`

Remerciements

Je tiens à remercier mes deux encadrants de thèse : Nabil Layaïda et Pierre Genevès. Ils m'ont intégré à l'équipe Wam en stage de licence puis dans l'équipe Tyrex pour un second stage et enfin ont accepté d'encadrer ma thèse. Je tiens aussi à remercier Nils Gesbert. Au-delà des conversations intéressantes (scientifiques ou non), Nils a suivi mes recherches, s'est régulièrement plongé dans mes écrits (souvent obscurs) et m'a fourni de nombreux retours pertinents ! Merci !

Je tiens ensuite à remercier tous les membres du jury. Je suis honoré par la présence de chacune de ces personnes : Nabil et Pierre, mes encadrants de thèse ; Mme Manolescu et M. Colazzo, à qui j'adresse un remerciement tout spécial pour avoir accepté de rapporter mon manuscrit et pour avoir fourni de nombreux conseils qui m'ont permis d'améliorer le manuscrit actuel ; MM. Euzenat et Valduriez qui ont accepté d'examiner ma thèse et avec lesquels j'ai eu l'occasion à plusieurs reprises de discuter de recherche et de mon futur dans ce milieu.

Je remercie aussi les nombreuses personnes qui sont passées par l'équipe Tyrex et avec qui j'ai eu plaisir à travailler : Abdullah, Canard, Cécile, Graux, Helen, Mathieu, Thibaud, Thomas, ainsi que les "nouveaux" : Fateh, Sarah, Raouf, Muideen, Laurent.

J'ai passé cette dernière année de thèse, en ATER, à l'École Normale Supérieure. Je remercie très fortement Pierre Senellart, pour m'avoir aidé à candidater puis intégré dans son équipe. Travailler aux côtés de Pierre a été très instructif, je regrette de n'avoir eu que peu de temps à consacrer à la recherche dans cette équipe dont les sujets étaient intéressants et l'ambiance de travail très stimulante. Je remercie tous les membres de l'équipe Valda qui m'ont accueilli et n'ont jamais hésité à répondre à mes questions. Plus généralement, je tiens à remercier tous les membres du haut du DI pour l'excellente année passée avec elleux.

J'ai eu l'occasion durant ma thèse de discuter avec de nombreuses personnes et je me suis beaucoup nourri de ces échanges. Je tiens particulièrement à remercier la communauté d'ulminfo pour les intenses discussions sur de multiples sujets. Et pour des échanges plus scientifiques, passionnants, qui ont mené à des collaborations : Pierre B., Pierre S., Vincent L., a3nm.

J'en viens maintenant à remercier ma famille, à qui je dois mon goût des sciences et qui m'a toujours soutenu dans cette voie et même poussé, quand il le fallait, à aller toujours plus loin. Cette thèse a été un éloignement physique mais c'est toujours un plaisir de vous retrouver.

Je voudrais aussi remercier mes amis. Nombreuses sont les personnes que j'ai déjà citées qui sont bien plus que des collègues et je les en remercie. En plus de ceux-ci, je voudrais remercier les personnes suivantes qui ont été présentes tout au long de ma thèse : les grotas (aaz, amiel, grogrodile, grotarrel, grotenedicte, grotillon, haveo, helene, jonas, Lea, Machin, maud, Mc, MLB, p4bl0, pandarion, picomango, snoopy, Ted) et la grotadhérence (trop peuplée pour être listée), groupe d'ami·e·s formé pendant mes études et dont je suis resté très proche malgré la distance ; les pensionnaires de divers canaux d'ulminfo (a3nm, Bibi, olasd, iXce, et bien d'autres) et enfin les nombreux amis que je me suis faits à Grenoble : les colocs, les voisins, les rôlistes, les litt&Arts (les voisines !).

Enfin je tiens à remercier Camille Brouzes, qui m'a supporté moralement, hébergé et qui m'a beaucoup aidé dans les dernières étapes de cette thèse.

Un grand merci !

General introduction

The semantic web is an extension to the World Wide Web whose goal is to make the Web carry more semantic information in a machine-readable format. Towards this goal, the W3C developed several standards such as RDF which models knowledge using graphs and SPARQL which is a query language for RDF graphs.

The field of research in databases is more than fifty years old but since its introduction by Codd at the beginning of the 70s, the relational model has received most of the attention from both the academia and the databases vendors. In particular, the optimization and the fast evaluation of relational queries has been extensively studied.

As new data models and query languages raise new challenges and as the RDF+SPARQL model diverges from the relational model, an interrogation naturally appears: *can we develop a new model based on the relational model and its thorough body of research to tackle languages such as* SPARQL*?*

In order to answer this question, we will present models of data and queries in the first part of this thesis. We will start in chapter 1 with the semantic web world and describe the RDF data model and its companion query language SPARQL. SPARQL will be presented via the SPARQL–algebra which is a formal modeling of SPARQL queries provided by the W3C and used by several query evaluators to represent and manipulate queries internally. We will also briefly look into existing query evaluators and assess their performance. This performance review will, in particular, investigate two salient points where we observe counter performances: on complex queries (e.g. with recursion) and on very large datasets.

As the relational model was at the center of a huge body of research, and as it will serve as the basis for our proposed approach, we will present, in chapter 2, the relational model and algebra along with the Datalog query language. We will also compare the relational model with the RDF+SPARQL model which will justify that our approach does not use the plain relational model.

As we will have demonstrated by going through both models, there are remaining challenges for the optimization of SPARQL queries and particularly for complex queries with recursion. For this type of queries, even traditional relational query evaluators cannot handle well very simple queries. Also, at the opposite side, distributed query evaluators for very large datasets could also benefit from performance improvements. That is why, in the second part of this thesis, we will introduce our general approach to SPARQL evaluation using our new model: the μ-algebra.

An overall depiction of our approach is presented in figure 1. It mimics the optimization strategy of SQL, which can summed up as this process: first the query is translated into a

logical representation; then, using rewrite rules, we produce many equivalent terms (in the sense that they have the same answers) to the term obtained through the translation. Each of those terms can be seen as a Query Execution Plans for the original query and, thus, for each term, we estimate the time this term will take to be evaluated. Finally we select the term that takes an estimated minimal time to be evaluated and evaluate it. In this thesis our perspective is the optimization of SPARQL query evaluation but since our optimization efforts will concentrate on this μ-algebra any language that could be converted into μ-algebra would benefit from our work.

The question of adapting the relational algebra that models relational queries to model the evaluation of languages à la SPARQL will be dealt with in chapter 3. In this chapter, we will present the syntax and semantics of our μ-algebra. We will also present a translation from a large fragment of SPARQL to this query language. This chapter corresponds to the first arrow of figure 1.

We will then dedicate chapter 4 to present our rewriting strategy for μ-algebra terms. This strategy, as shown in figure 1, distinguishes between rewrite rules that *produce* new terms and rewrite rules that serve a *normalizing* purpose. The idea behind producing rules is to create new query execution plans while the idea behind the normalizing rules is to reduce the numbers of μ-algebra terms considered. In order to describe when those rules can be applied, this chapter introduces a typing mechanism for μ-algebra terms. This chapter also includes definitions, lemmas and theorems in order to prove the validity of the new rewrite rules we introduced.

Given all the terms produced by our rewriting strategy, the natural next step is to devise a cost model for our terms, allowing us to select the term that is the most efficient. However, in order to devise a cost model, we need a precise idea of how our terms will be evaluated. Chapter 5 explores this problematic and proposes a general way to evaluate μ-algebra terms and describes two prototypes we implemented: a μ-algebra evaluator called `musparql` and a distributed evaluator called SparqlGX based on Apache Spark. SparqlGX is limited to a fragment of the μ-algebra (and thus a fragment of SPARQL). Using these evaluators we can devise a cost model that can, for instance, predict accurately the time to compute the join of two sets A and B knowing the size of A and B. However, to decide the best term, this cost model will, in turn, depend on a cardinality estimation.

This quest for a cardinality estimation scheme for the μ-algebra is tackled in the chapter 6 with a new technique that provides a worst-case cardinality estimation based on a new tool: collection summaries. Collection summaries capture the implicit schema often found in RDF datasets. Our new tool captures this schema even in the presence of a few violations.

To validate our approach, chapter 7 compares it with the state of the art from both theoretical and empirical standpoints. The theoretical part compares query execution plans possible with the μ-algebra with query execution plans of various other approaches. We demonstrate that, even on very simple queries there are graphs where our new plans have a better complexity: in the scenario we present they have a linear complexity (in the size of the graph) while all other approaches are, at least, quadratic. We then rediscover experimentally this result by comparing our implementation based on the μ-algebra with SPARQL, SQL and Datalog engines. Finally we show that our distributed evaluator of SPARQL queries has state of the art performance and that our cardinality estimation allow us to improve further the performance of SparqlGX.

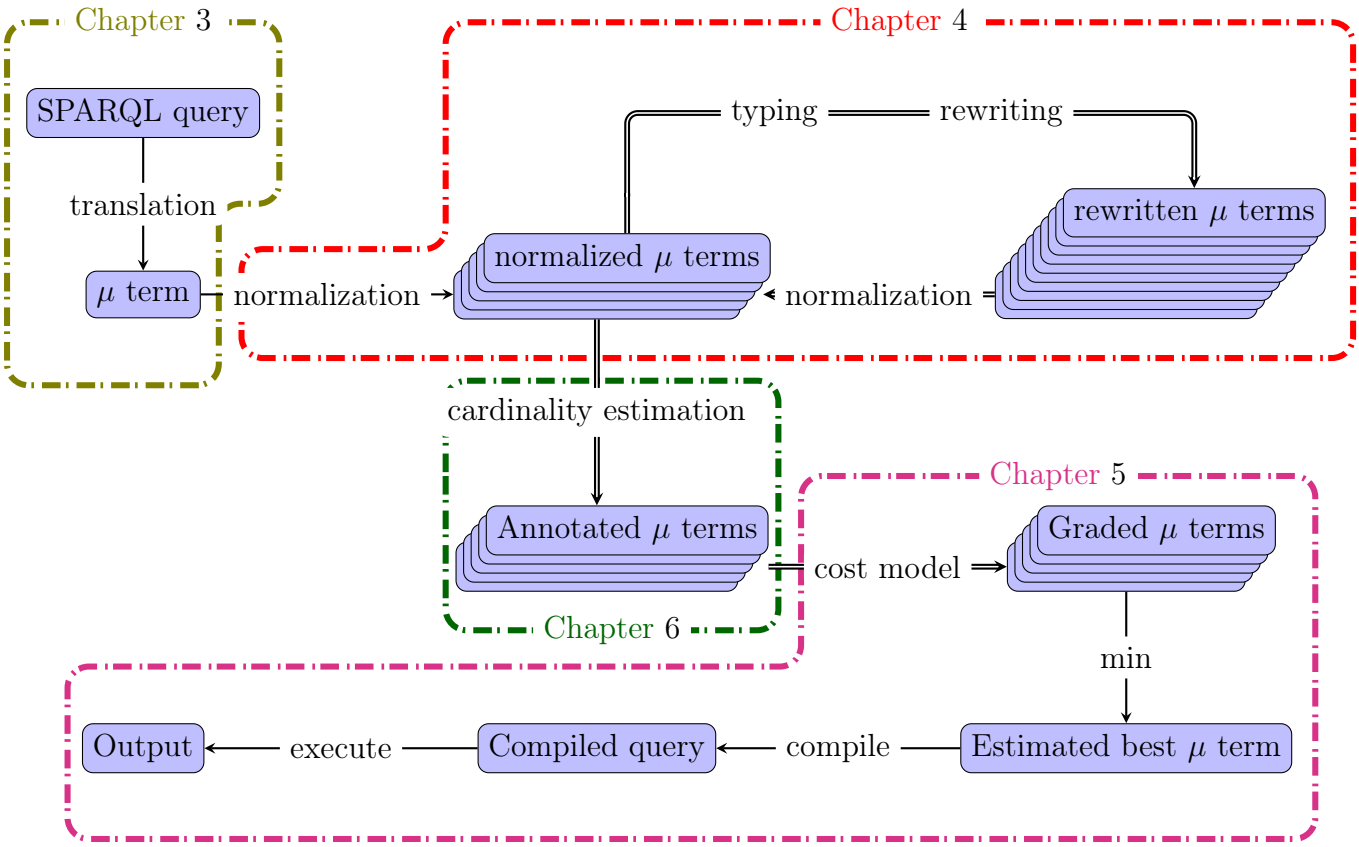


Figure 1: Schematic representation of our approach and the chapter decomposition

Contents

Abstracts **3**

General introduction **9**

Contents **14**

I Preliminaries 15

1 The SPARQL language **19**
1.1 The RDF data model 21
1.2 The SPARQL query language and the SPARQL-algebra 26
1.3 SPARQL query evaluators and their optimization 35

2 The relational model and beyond for graphs queries **39**
2.1 The relational model 41
2.2 Datalog 49
2.3 Encoding SPARQL queries into relational query languages 51

II The μ-algebra for the execution of SPARQL queries 55

3 The μ-algebra **61**
3.1 Syntax & Semantics 63
3.2 Examples of μ-algebra terms 68
3.3 Restriction on μ-algebra: constant and linear recursions 70
3.4 Relationship between μ-algebra and existing relational variants 72
3.5 Translation from SPARQL 73

4 Analysis & transformation of μ-algebra terms **81**
4.1 Preliminaries: several examples of terms to be rewritten 83
4.2 Effects of μ-algebra terms 84
4.3 Decomposed fixpoints 90
4.4 Typing μ-algebra terms 93

4.5 Normalizing rules for μ-algebra terms . . . 96
4.6 Producing rules . . . 100
4.7 Ad-hoc rules . . . 104
4.8 Rewriting algorithm . . . 105
4.9 Example of rewriting . . . 107

5 Evaluation of μ-algebra terms 111
5.1 General bottom-up evaluation for μ-algebra terms . . . 113
5.2 Bottom-Up cost model for μ-algebra terms . . . 117
5.3 Single core bottom-up evaluation of μ-algebra . . . 119
5.4 Towards a distributed μ-algebra evaluator . . . 119

6 Cardinality estimation of μ-algebra terms 125
6.1 Summaries . . . 127
6.2 Computing collection summaries representing the solutions of a single TP . . 133
6.3 Optimization of distributed BGP query plans with an over-estimation . . . 135
6.4 Extensions . . . 137

III Results & conclusion 139

7 Comparision of our approach with the state of the art 141
7.1 Theoretical comparison of μ-algebra bottom-up evaluation and other approaches on recursive queries . . . 143
7.2 An experiment for μ-algebra bottom-up with a recursive query . . . 147
7.3 Pushing the benchmark further . . . 152
7.4 Efficiency of distributed SPARQL query evaluators . . . 158
7.5 Experimental of SparqlGX with our cardinality estimation . . . 161

8 Conclusions & Perspectives 167
8.1 Conclusions . . . 167
8.2 Perspectives . . . 169

Appendices 170

Bibliography 171

A Proofs 183
A.1 Proofs of chapter 3 . . . 183
A.2 Relationship between μ-algebra and existing relational variants . . . 186
A.3 Proofs of chapter 4 . . . 192

B Details of our second benchmark 205

Part I

Preliminaries

Table of Contents

1 The SPARQL language **19**
1.1 The RDF data model 21
1.2 The SPARQL query language and the SPARQL-algebra 26
1.3 SPARQL query evaluators and their optimization 35

2 The relational model and beyond for graphs queries **39**
2.1 The relational model 41
2.2 Datalog 49
2.3 Encoding SPARQL queries into relational query languages 51

CHAPTER 1

The SPARQL language

The semantic web *corresponds to an extension of today's* World Wide Web. *While the World Wide Web was designed to enable humans to share information with each others, the goal of the* semantic web *is to share information in a machine-readable format carrying semantic information.*

Sharing information via the semantic web allows machines to reason and process information created from multiple outside sources in a federated manner. In the "old" syntactic web, one can create one parser per source of data and then gather them into a large silo in order to query them. In the semantic web, one would simply write a query that would in turn query each source for its data and gather the needed data at run time. There would be no need for a parser per source as the data would be served in a common unifying language. One could, for instance, imagine a trip planning system combining flight and train information extracted from several companies and then coupling that with landmarks information (using e.g. dbPedia) and even with some meteorological forecast data. And all of that without the need to centralize the data.

In this chapter we present the RDF *data model and the* SPARQL *query language which are both specifications from the World Wide Web Consortium (*W3C*) to express semantic information on the web and to query it. We will skip over many aspects of these specifications as they are large and we will focus on the essential parts needed for the evaluation of* SPARQL *queries on* RDF *data. We will then look at query evaluators and their optimization techniques.*

1.1 The RDF data model

The Resource Description Framework (RDF) is a language standardized by the W3C [RCM14]. RDF models knowledge about resources. An RDF dataset is a set of statements where each statement gives either a relationship between resources or describes a resource. Before diving into the details of RDF we first need to introduce some terminology.

Serialization and RDF Serialization consists in producing a string representation of typed data. One of the goal of RDF is the ability for RDF–compatible software to understand data coming out of other tools therefore a large part of the RDF standard focuses on the serialization.

This chapter will give some hindsight about this serialization but the details are left to the official standard and, for the sake of simplicity, after this chapter, we will refer to arbitrary strings to escape the details of the encoding and focus on the essential challenges.

1.1.1 Terminology

Ressources

The *resources* described in a RDF dataset can be any object or class that we want to describe. For instance in a dataset stating that `Socrate is human` and that `human is mortal`, All of `Socrate`, `human` and `mortal` would be resources but the verb `is` would also be a resource as it is an entity that could be described. For instance, the dataset could declare that `is` has a transitive meaning which here would imply that given `Socrate is human` and `human is mortal` we also have `Socrate is mortal`.

In an RDF dataset, resources are either represented directly as an IRI or as anonymous resources and represented as *Blank nodes*.

Internationalized Resource Identifier (IRI)

An Internationalized Resource Identifier (IRI) is a string of characters that identifies a *resource*. The IRI scheme extends the URI scheme by allowing any unicode character (and not just ASCII as in the URI scheme).

We refer the reader to the RFC 3987 for a precise description of IRI but as a simplification, an IRI can be seen as a string of the form `scheme:path` (e.g. `https://www.inria.fr` uses the scheme `https` and the path is `//www.inria.fr`).

The reader probably has an intuitive notion of IRI as this format is very similar to the URL scheme which is used to encode internet addresses but note that an IRI does not necessarily correspond to an actually internet-accessible address nor that the scheme used has to be known (even though the schemes `http` and `https` are very widely used, in which case the path often corresponds to a web accessible resource).

Remember that the goal of RDF is to represent knowledge for the semantic web. In order to do that, we need to be able to identify resources across datasets. Therefore, when the same IRI i is present in two different datasets D_1, D_2 then the i in D_1 and the i in D_2 are considered to identify the same object.

Prefixes

For space and readability reasons, RDF datasets and SPARQL queries often use a set of prefix mappings. A prefix mapping is a pair (prefix `p`, iri `<i>`) and indicates that the strings of the form `p:v` is equivalent to its expanded form `<iv>` (where `iv` is the concatenation of the string `i` with the string `v`).

In RDF and in SPARQL queries, an IRI should either appear enclosed in angle brackets (of the form `<scheme:path>`) or be prefixed (of the form `prefix:subpath`). In this latter case, there should be a unique matching prefix (`prefix`,`iri`) and thus the `prefix:subpath` can be replaced by the concatenation of `iri` and `subpath` enclosed in angle brackets (thus `<iri subpath>`).

Notice that prefixes are only here for concision and readability purposes as IRI tend to be long. In fact, prefixes are not even part of the formal RDF data model. A prefixed IRI should always be treated as its expanded version (and thus a simple way to do just that is to replace them by their expanded version).

The prefixes we will use in this thesis are described in the following table :

Prefix	Expansion	Meaning
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#	The RDF built-in vocabulary
rdfs	http://www.w3.org/2000/01/rdf-schema#	The RDF Schema vocabulary
xsd	http://www.w3.org/2001/XMLSchema#	The RDF-compatible built-in XSD types
foaf	http://xmlns.com/foaf/0.1/	An RDF vocabulary to link people
ex	http://example.org/	A dummy prefix to shorten examples

Blank Nodes

An RDF dataset might include anonymous resources represented as *blank nodes*. A *blank node* is of the form `_:subpath` where `_:` designates exactly the string `_:` while `subpath` can be any valid IRI path.

The idea behind blank nodes is to describe resources that exist and have some relations with other (named or anonymous) resources. For instance if someone wishes to represent an ordered list of three authors (represented as `ex:authorA`,`ex:authorB`,`ex:authorC`) of an article (represented by `ex:paper1`) this person could create a node representing this ordered set with a blank node (e.g. `_:abc`) and state that `paper` has been authored by `_:abc`. The use of a blank node ensures that there will be no conflict with IRI existing outside of this datasets.

In contrast with `iri`, if a blank node b is present in D_1 and in D_2, there is no reason to suppose b in D_1 and b in D_2 are describing the same object and they should be treated as different.

As explained in the RDF 1.1 recommandation, it is possible to replace blank nodes with IRI using a skolemnization process. But in order to respect the RDF semantics, we need to replace blank nodes with IRI that are 1) cannot be used in other datasets , 2) can be identified unambiguously as blank nodes (as e.g. SPARQL allows to test whether a resource is a blank node).

Literals

A literal is a string of characters (enclose in a pair of `"`) eventually adjoined with a tag language information (concatenated at the end of the string and starting with a `@`) or tag

data type (also concatenated at the end of the string but starting with a `^^`). The RDF standard only supports a tag language in a literal when its data type is `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`, therefore the data type is thus often omitted for literals with language information. When there is no datatype nor language information, the data type is considered to be `http://www.w3.org/2001/XMLSchema#string`.

For instance the literal `"42"^^xsd:integer` represents the string "42" and indicates that it should be treated as an "`xsd:integer`" (which are the integers as defined by the XML standard). The literal `"Londres"@fr` represents the string "Londres" with the indication that it is in french and `"London"@en` represents the string "London" with the indication that it is written in english.

The values encoded via literals could also be encoded as resources and thus via an IRI. But some types of things (such as string, integers, dates, boolean) are more easily manipulated with some type information than with IRI. For instance in SPARQL, the comparison on typed data will not use the same algorithm to compare value depending on the type of the values compared (e.g. `xsd:boolean`, `xsd:dateTime`, etc.).

Note that it is possible to create custom data types however this is not supported by all evaluators and thus the types actually used correspond to the types defined by the RDF standard and whose semantics is imported from the XML standard (and thus their IRI starts with `http://www.w3.org/2001/XMLSchema#`).

1.1.2 RDF Triples

RDF datasets represent knowledge through sets of statements. Each of these statements is an RDF *triple*. As its name suggests, a triple is composed of three elements (s p o):

- the *subject* s which designates a resource and thus is either a blank node or an IRI;
- the *predicate* p which always is an IRI;
- the *object* o which can by either a resource (blank node or IRI) or a literal.

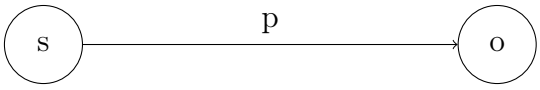
Predicates can be thought as of predicate in the predicate calculus or in theories of grammars. The idea is that each predicate p carries a semantic relationship between the subject s and the object o.

Following the W3C notation, we note $\mathcal{I}$ the set of valid IRI, RDF–$\mathcal{B}$ the set of valid blank nodes names and RDF–$\mathcal{L}$ the set of valid literals then these three sets $\mathcal{I}$, RDF–$\mathcal{B}$ and RDF–$\mathcal{L}$ are disjoint and an RDF triple can be seen as an element of $(\mathcal{I} \cup \text{RDF} - \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \text{RDF} - \mathcal{B} \cup \text{RDF} - \mathcal{L})$.

Finally, the standard denotes as the set of RDF–terms the set $\mathcal{T} = \mathcal{I} \cup \text{RDF} - \mathcal{B} \cup \text{RDF} - \mathcal{L}$ (which is also the set of valid objects in an RDF triple). In this view, an RDF triple is an element of $\mathcal{T}^3$ (however, not all elements of $\mathcal{T}^3$ qualify as triples).

1.1.3 RDF graphs

A triple can also be seen as a part of labeled directed graph. The triple (s, p, o) links the node s to the node o via a directed arc labeled with p as depicted below.



A set of triples $\{(s_1, p_1, o_1), \ldots, (s_n, p_n, o_n)\}$ is thus seen by the RDF standard as a directed labeled multigraph $G = (V, E)$ whose nodes are the subjects and objects (i.e. $V = \{s_1, \ldots, s_n\} \cup \{o_1, \ldots, o_n\}$) and there is an edge per triple (s_i, p_i, o_i), starting from s_i going to o_i labeled with p_i (i.e. the edges are $E = \{s_1 \xrightarrow{p_1} o_1, \ldots, s_n \xrightarrow{p_n} o_n\}$).

Nodes in RDF graphs are therefore RDF-terms but labels can only be IRI. Note that an IRI appearing as the label of a transitions (and thus as a predicate) can also be a node of the graph.

An RDF *dataset* is a set $\{G, (< u_1 >, G_1), \ldots, (< u_n >, G_n)\}$ where each of the $G, G_1, \ldots, G_n$ corresponds to an RDF graph and each of the u_i is a valid IRI. The graph G_i is *named* $< u_i >$ while the graph G is the *default* graph.

1.1.4 An example

Let us consider the following example describing the phylogeny of penguins and tyrannosaurus and showing that they are not so distant cousin.

```
ex:tyrex ex:subTaxon ex:saurichien .
ex:tyrex rdfs:type ex:taxon
ex:tyrex rdfs:label "Tyrannosaur"@en .
ex:tyrex rdfs:label "Tyrannosaure"@fr .

ex:birds rdfs:type ex:taxon .
ex:birds rdfs:label "Aves" .
ex:birds ex:subTaxon _:missingLink .

_:missingLink ex:subTaxon ex:saurichien .
_:missingLink rdfs:type ex:taxon .

ex:penguins rdfs:type ex:taxon .
ex:penguins rdfs:label "Sphenisciformes"@en .
ex:penguins ex:subTaxon ex:birds .

ex:saurichien rdfs:label "Saurichia"@en .
```

Note that we used a blank node (`_:missingLink`) to represent the existence of a missing link between birds and saurichias. The same dataset in a graph representation :

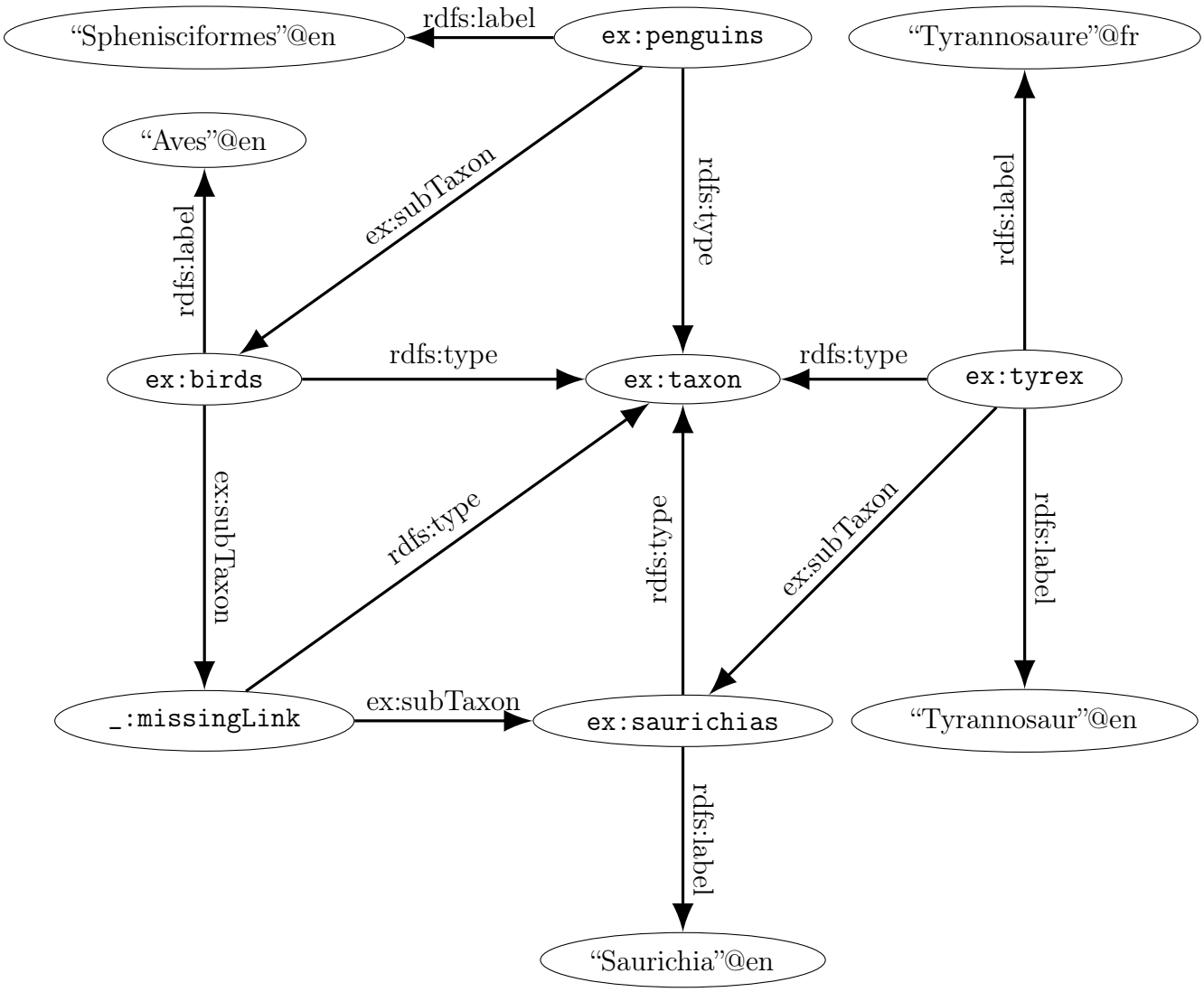


Figure 1.1: Graph representation of our example dataset

1.1.5 Entailment

The RDF technology stack also comes with an *ontology* mechanism to access data. An ontology is a set of rules (eventually also encoded in RDF) to deduce new statements implied by the set of statements. It allows, for instance, to write a set of rules such that an RDF–equivalent of the statement `socrate is mortal` can be automatically deduced from the RDF–equivalents of the statements `socrate is human` and `human is mortal`.

The entailment regimes in RDF are very rich and interesting but they are not in the set of topics that we will cover in this thesis. It does not mean, however, that our method can not be applied to data with entailment. One of the ways to treat entailment is to have a pre-processing phase enriching the data with the knowledge discovered by an entailment regime. If our method does not treat the enrichment part, our query evaluator is obviously capable of treating enriched data. Another method consists in rewriting the query q that we want to evaluate on a database using an ontology o into a query q_o where the semantics of q_o (without ontology) is the semantics of q with the ontology o. Depending on q and o,

the resulting query q_o might be much more complex than q (and may not be expressible in SPARQL).

1.2 The SPARQL query language and the SPARQL-algebra

The SPARQL acronym stands for SPARQL Protocol and RDF Query Language. SPARQL is, as its name suggests, a query language for RDF data. SPARQL has been widely adopted since its standardization by the W3C [HSP13] to query and update RDF data and it is now in its 1.1 recommendation (which is the second major version). In this section we will not present the whole SPARQL but rather we will present the multiset fragment of the SPARQL–algebra which is an algebraic representation of SPARQL queries that helps to formally define the semantics of SPARQL queries.

1.2.1 Definitions

Definition 1. *The* SPARQL *language contains* variables *(that we will call* SPARQL-variables*). The set of valid* SPARQL*-variable names is noted $\mathcal{V}$ and is disjoint with the set $\mathcal{T}$ of* RDF*–terms. In this manuscript* SPARQL*–variables will be spaceless strings of characters starting by '?' or '$'.*

The SPARQL standard formalizes query answers as *solution mappings* whose domain contains only SPARQL–variables (i.e. elements of $\mathcal{V}$) and whose range are RDF terms (i.e. $\mathcal{T}$). Solution mappings thus have the type $\mathcal{V} \to \mathcal{T}$, however, during the computation of a solution mapping, we also have to compute an *instance mapping* which is a partial mapping whose domain is the set of blank nodes (i.e. a mapping of type $\mathcal{B} \to \mathcal{T}$). In SPARQL–algebra the combination of such an instance mapping with a solution mapping is called a *Pattern Instance Mapping* (therefore a mapping of type $\mathcal{V} \cup \mathcal{B} \to \mathcal{T}$. As the difference between the several parts of *Pattern Instance Mapping* will only appear during the translation, we will call *mappings* for all the three types of mappings and remember which part is from the instance mapping and which part is solution mapping (which is easy since the set of SPARQL–variables is disjoint with the set of RDF terms).

Definition 2. *A* binding *is a pair (*SPARQL*–variable,* RDF*–terms) or (blank node,* RDF*–term).*

Definition 3. *A* mapping *is a partial function $m : \mathcal{V} \cup \mathcal{B} \to \mathcal{T}$ and the domain of m (noted $dom(m)$) is the subset of $\mathcal{V} \cup \mathcal{B}$ where m is defined.* Mappings *have a finite domain.*

Alternatively, a mapping *can be seen as a finite set of bindings where the* SPARQL*–variables and blank nodes in the bindings are all distinct.*

Definition 4. *Two mappings m_1 and m_2 are said* compatible *when for all $c \in dom(m_1) \cap dom(m_2)$ we have $m_1(c) = m_2(c)$.*

Given two compatible mappings m_1 and m_2 we can define $m_1 + m_2$ as the union of their bindings or alternatively as the mapping whose domain is $dom(m_1) + dom(m_2)$ and such that

$$(m_1 + m_2)(c) = \begin{cases} m_1(c) & \textit{when } c \in dom(m_1) \\ m_2(c) & \textit{otherwise} \end{cases}$$

1.2.2 A first SPARQL Example

This example is based on the RDF dataset presented earlier. We consider the SPARQL query asking for the name of all taxons and, when available, the name of the parent taxon. This could be written in SPARQL as:

```
SELECT ?name, ?parentName WHERE {
  ?taxon rdfs:type ex:taxon .
  ?taxon rdfs:label ?name .
  OPTIONAL {
     ?taxon ex:subTaxon _:parent .
     _:parent rdfs:label ?parentName .
     }
}
```

In this query, there is one blank node `_:parent`; three variables `?taxon`, `?name` and `?parentName` and 4 constants : `ex:taxon`, `rdfs:type`, `rdfs:label`, and `ex:subTaxon`. The solutions of such a query will necessarily be mappings whose domain always contains `?name` and might also comprise a `?parentName`.

Against the example dataset of figure 1.1 we have:

A

name	parentName
"*Sphenisciformes*"	"*Aves*"
"*Tyrannosaure*"	"*Saurichia*"
"*Tyrannosaur*"	"*Saurichia*"
"*Aves*"	
"*Saurichia*"	

Here "Aves" has a parent taxon (`_:missingLink`) but it has no name and "Saurichia" does not have a parent taxon.

1.2.3 Operations

SPARQL queries are evaluated against RDF datasets. As we explained in the RDF presentation, RDF datasets are composed of several graphs: one default graph and several named graphs. During the evaluation of SPARQL queries we will maintain a *current* or *active* graph that corresponds to the graph queried in the RDF dataset. At first, the active graph will be the default graph of the RDF dataset but the SPARQL standard also allows to change this active graph to one of the named graphs.

We now present the multiset fragment of the SPARQL–algebra along with its semantics. The fragment corresponds to the SPARQL–algebra where we removed the operators: Exists, ToList, OrderBy, Slice. We choose this fragment because it allows us to present only the multiset semantics of SPARQL: the last three removed operators (ToList, OrderBy, Slice) operate on a list semantic (a multiset where the elements are ordered) while the Exists operator has a complex semantics (it triggers the evaluation of subqueries) but it has been shown that SPARQL queries can be rewritten without this Exists (as shown in [KKG17]) even though this rewriting is non-trivial.

Triple Patterns

The building blocks of the SPARQL queries are Triple Patterns (TP). Similarly to an RDF triple, a TP is composed of three parts :

- a *subject* s that is either an RDF–term or a SPARQL–variable (i.e. $s \in \mathcal{V} \cup \mathcal{T}$) ;
- a *predicate* p that is either an IRI or a SPARQL–variable (i.e. $p \in \mathcal{V} \cup \mathcal{I}$) ;
- an *object* o that is either a SPARQL–variable or an RDF–term (i.e. $o \in \mathcal{V} \cup \mathcal{T}$).

Note that there is a discrepancy between SPARQL TP and RDF triples, the former allowing subjects to be literals while the latter does not. A TP (s, p, o) is an element of $(\mathcal{T} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{T} \cup \mathcal{V})$, but, as for RDF triples, it can also be seen as an element of $(\mathcal{T} \cup \mathcal{V})^3$ (but not all elements of $(\mathcal{T} \cup \mathcal{V})^3$ are valid SPARQL TP).

let G be the current graph, a solution mapping m is solution of $TP(s, p, o)$ when $dom(m) = \{s, p, o\} \cap (\mathcal{B} \cup \mathcal{V})$ and $\tilde{m}(s)\ \tilde{m}(p)\ \tilde{m}(o)$ is a triple of G (for $\tilde{m}(x) = x$ when $x \in dom(m)$ and $\tilde{m}(x) = x$ otherwise). Note that blank nodes that appear in queries acts just like SPARQL–variables (except that they are not included in the output and cannot be used in expressions).

Property Paths

Property Paths (PP) are a novelty of the 1.1 version of the SPARQL standard. A TP can be seen as a mean of expressing the connection between two nodes in a graph (where both nodes and the label of the edge can also be variables). PP extend TP by allowing the predicates to be Regular Path Expressions.

The syntax for these Regular Path Expressions is the following:

u	an IRI
r_1/r_2	the concatenation of two paths
$r_1 \mid r_2$	the alternative choice between two paths
r^{-1}	a reversed path
$!\{: i_1 \cdots : i_n\}$	a negated property set
$\{: i_1 \cdots : i_n\}$	a property set
	(any path of length one labeled by something else than one of the i_j)
$r?$	optional path
r^+	the transitive closure of the path r
r^*	the transitive reflexive closure of the path r

Let G be the active graph, a solution mapping m is a solution of $PP(s\ r\ o)$ when $dom(m) = \{s, o\} \cap (\mathcal{V} \cup \mathcal{B})$ and there is a path $\tilde{m}(s) = p_1 \xrightarrow{l_1} p_2 \ldots p_k \xrightarrow{l_k} p_{k+1} = \tilde{m}(o)$ in G such that the sequence of labels $l_1 \ldots l_k$ matches the regular path expression r. Note that as there might be several paths from $\tilde{m}(s)$ to $\tilde{m}(o)$ the mapping m might be present more than once as explained in the paragraph multiset semantics of Property Paths.

For SPARQL, blank nodes appearing in the query play a special role but without an entailment regime, they act as a form of local variable that is not included in the output (see BGP below).

Technically, PP are not an extension of TP since variables cannot occur in the predicate of a PP. However we can allow the predicate to be either a variable or a regular path query (constant are a special case of regular path queries) and a PP that strictly extend the TP.

A TP has a multiset semantics (as all others SPARQL operators) but each assignment of variables and blank nodes either appear zero or once. That is why, TP can be evaluated under a set semantics.

Example For instance on the graph of section 1.1.4, we can ask for the pairs of `?childTaxon`, `?ancestorTaxon` using PP(`?childTaxon ex:subTaxon`$^+$ `?ancestorTaxon`) or we can even ask for the label of ancestor taxons : PP(`?taxon ex:subTaxon`*`/rdfs:label ?label`) which gives us:

?taxon	**?label**
ex:saurichien	"Saurichia"@fr
ex:penguins	"Sphenisciformes"@en
ex:penguins	"Aves"@en
ex:penguins	"Saurichia"@en
ex:birds	"Aves"@en
ex:birds	"Saurichia"@en
ex:tyrex	"Tyrannosaure"@fr
ex:tyrex	"Tyrannosaur"@en
ex:tyrex	"Saurichia"@en
_:b0	"Saurichia"@en

Note that the blank node `_:missingLink` has been renamed to `_:1`. This is perfectly legal and normal as blank node names are always relative to a dataset.

The multiset semantics of Property Paths During the elaboration of the standard, Property Paths had a multiset semantics. However, such semantics create problems on property paths such as: `?from` (`:a`)$^+$ `?to`. Indeed, how many times a given assignment of (`?from`, `?to`) such be included in the result? Loops create an obvious problem but even without loops counting the number of paths is very hard. To solve this problem the standard imposes that all the operators except | and / have a set semantics (when | and / are nested inside another operator they also fall back to set semantics).

To simplify the set vs multiset problem of PP, we can transform patterns $(r_1|r_2)$ and (r_1/r_2) appearing on top using *Union* and *Join* (described below). After this transformation all PP can be evaluated under the set semantics without changing the semantics.

A pattern A $(r_1|r_2)$ B is translated into $Union(A\ r_1\ B, A\ r_2\ B)$ and a pattern A (r_1/r_2) B into $Join(A\ r_1$ `_:tmp`, `_:tmp` $r_2\ B)$ with `_:tmp` a new blank node name not appearing in the query.

For instance to look for the pairs of taxons that are related we can use the property path `?A ex:subTaxon`*/(`ex:subTaxon`$^{-1}$)* `?B` which is equivalent to `?A ex:subTaxon`* `_:commonAncestor` and `_:commonAncestor` (`ex:subTaxon`$^{-1}$)* `?B`. Here, a mapping for (`?A`, `?B`) will appear as many times as there are common ancestors.

Join

Given two SPARQL–algebra terms t_1 and t_2 we can combine them through a *Join* operator with $Join(t_1, t_2)$. For any pair of elements e_1, e_2 solutions of t_1 and t_2 such that e_1 is compatible with e_2 then $e_1 + e_2$ is a solution of $Join(t_1, t_2)$.

Note that this definition is on multisets. A mapping is thus solution as many times as there are pairs producing it in the solutions of t_1 and t_2.

Union

The operator $Union(t_1, t_2)$ simply refers to the multiset union. A mapping m occurs in the solutions of $Union(t_1, t_2)$ when it occurs as a solution of t_1 or t_2 and the number of occurrences of m in the solutions of $Union(t_1, t_2)$ is the sum of the number of occurrences in the solutions of t_1 and the solutions of t_2.

Note that the mappings solution of a PP all share the same domain which is the set of variables appearing in the PP. And the mappings solutions of join of two (or more) PP also share the same domain (which is the set of variables appearing in at least one of the PP). However this is not the case for union.

Project

Given a mapping m, its projection on the set $s \subseteq \mathcal{B} \cup \mathcal{V}$ is the mapping m' whose domain is $dom(m') = dom(m) \cap s$ and where the remaining values are unchanged, i.e. $\forall x \in dom(m') : m'(x) = m(x)$.

The project operator $Project(C, t)$ changes the mappings solutions of t by projecting them onto the set C.

Basic Graph Patterns

A Basic Graph Pattern (BGP) is a set of PP. A $BGP((s_1\ p_1\ o_1) \ldots (s_n\ p_n\ o_n))$ can be seen as syntactic sugar for the join of individual PP where we keep in the domain of solution mappings only the variables (and not the blank nodes). In algebraic terms, it gives us $BGP(s_1\ p_1\ o_1) \ldots (s_n\ p_n\ o_n)) = (Project(V, Join(PP(s_1\ p_1\ o_1), \ldots, PP(s_n\ p_n\ o_n))))$ where $V = \{s_1, p_1, o_1, \ldots, s_n, p_n, o_n\} \cap \mathcal{V}$.

Example We can use blank nodes and PP to capture the pairs of animals coming from the same taxon (possibly all animals if our dataset is complete):

$$
\begin{aligned}
BGP(\{\ & PP(\texttt{?a subTaxon}^* \texttt{ \_:commonAncestor}),\\
& PP(\texttt{?b subTaxon}^* \texttt{ \_:commonAncestor}),\\
& PP(\texttt{\_:commonAncestor rdfs:type ex:taxon})\})
\end{aligned}
$$

Without the last PP all nodes (including the literals) would be included as solutions since `ex:subTaxon`* is a star operation that matches all pairs of twice the same node appearing in the RDF graph. Also with this query, the solution `?a` = `_:penguins` and `?b` = `_:penguins` would appear thrice as they have three common ancestors (`_:penguins`, `_:birds` and `ex:saurichias`) and one might want to ask to remove pairs of the same animals using filters.

Expressions

SPARQL includes a language of expressions. Expressions are used in filter condition, in *Extend* and in aggregates (see below). We present the main types of expressions and refer the reader to the SPARQL standard for a complete description.

The expressions are inspired by the syntax and semantics of expressions in the XQuery standard and thus operate on the built-in types of the namespace `xsd` such as `xsd:integer`, `xsd:double`, `xsd:boolean`, `xsd:datetime`, *etc.*.

The expressions contain the standard operations on the above types (e.g. addition and multiplication for the numeric types and time manipulation for the `xsd:datetime`). The expressions can also be comparison (equality and inequality tests) comparing either two columns of the mappings or a column of the mappings with a constant. The inegality comparisons in SPARQL depend on the type for the literals. For the literal datatype `xsd:integer` or `xsd:numeric`, SPARQL–algebra will rely on the usual number comparison, for the literals typed `xsd:DateTime` it will rely on a datetime comparison, etc.

Expressions can also be one of the built-in tests. The most important ones being: $bnd(\texttt{?c})$ to test whether the mappings contain a binding for the SPARQL–variable `?c`, $isIRI(\texttt{?c})$, (respectively $isBlank(\texttt{?c})$ and $isLiteral(\texttt{?c})$) testing whether the value bound by `?c` is an IRI (respectively a blank node or a literal). But we can also extract the language tag of a literal or get its datatype.

Expressions are themselves typed and can be in turn composed. For instance. instead of comparing directly the values associated to SPARQL–variables in the mappings, it also possible to perform computations on those. For instance, $\texttt{?a} + \texttt{?b} = \texttt{?c}$ is a valid expression returning true for mappings m such that $\{\texttt{?a}, \texttt{?b}, \texttt{?c}\} \subseteq dom(m)$ and $m(\texttt{?c}) = m(\texttt{?a}) + m(\texttt{?b})$ (when treated as numbers).

As shown earlier, the solutions of a given SPARQL–algebra term might not all share the same domain. Expressions are thus built to handle such situations and return an error value when we try to access a value that is not defined. Therefore tests such as `?a` = "42"^^`xsd:integer` and `xsd:not`(`?a` = "42"^^`xsd:integer`) would respectively return true and false when `?a` is defined and is equal to "42"^^`xsd:integer`. They would respectively return false and true for a `?a` defined as any other value and finally they would both return an error when `?a` is not defined.

An expression accessing an undefined value generally returns an error and errors from subexpressions are generally propagated. The exceptions are *bnd* (which tests whether a value is defined) and the special case of logical connectives && (for the logical AND), || (for the logical OR) and `xsd:not` (for the logical negation). The truth table of three-valued logic is given below using the notation $\top$, $\bot$ and *Error* for true, false and error (we only present the cases containing errors as the others follow the normal logical rules). The idea is that when the value does not change when replacing errors with true or false then the value of connective is equal to that and otherwise the value is error. This special treatment of boolean expression allows a concise way of treating filter conditions.

A	B	$A \&\& B$	$A \mid\mid B$
Error	$\top$	*Error*	$\top$
Error	$\bot$	$\bot$	*Error*
Error	*Error*	*Error*	*Error*
$\top$	*Error*	*Error*	$\top$
$\bot$	*Error*	$\bot$	*Error*

A	$not(A)$
Error	*Error*
$\top$	$\bot$
$\bot$	$\top$

Filter

A filter operator $Filter(f, t)$ returns the mappings solutions of the SPARQL–algebra t passing the filter f. A filter is passed when its value cast to the boolean is true (i.e. not false and not an error).

Diff and LeftJoins

The diff operator is needed to define the semantic of the LeftJoin. The diff operator computes the conditional difference and takes three parameters, two SPARQL–algebra terms t_1, t_2 and a filter condition f. The $Diff(t_1, t_2, f)$ will return mappings m that are the solutions of t_1 such that all the mappings m' that are solution of t_2 and compatible with m are such that $m + m'$ passes the filter condition $not(f)$ (remember that since filters are three-valued it is not exactly the same as not passing f, the filter could also evaluate to error in which case $not(f)$ will also evaluate to error).

When the condition f is always true (which means that $not(f)$ is always false), the $Diff(t_1, t_2, f)$ corresponds to the set of mappings of t_1 such that there are no compatible mappings in t_2.

Similarly to *Diff*, the *LeftJoin* in the SPARQL–algebra is conditional. *LeftJoin* takes three parameters: two SPARQL–algebra terms t_1 and t_2 and a filter condition f. The solution to $LeftJoin(t_1, t_2, f)$ is equal to the union of two sets. The first set corresponds to the join of t_1 with t_2 filtered by the condition f. The second set corresponds to the mappings of t_1 that were not used the first set (i.e. the mappings m solutions of t_1 such that all mappings m' of t_2 are such that $m + m'$ do not pass the filter f). The first set corresponds to the term $Filter(f, Join(t_1, t_2))$ and the second set corresponds to $Diff(t_1, t_2, f)$ and thus we have $LeftJoin(t_1, t_2, f) = Union(Filter(f, LeftJoin(t_1, t_2)), Diff(t_1, t_2, f))$.

Minus

The $Minus$ operation takes two SPARQL–algebra terms t_1 and t_2 and returns the mappings from t_1 such that all mappings t_2 are incompatible or have a disjoint domain.

The idea behind the $Minus$ operator is to remove all mappings from t_1 such that there is a compatible mapping in t_2 but we want to avoid the special case where mappings from t_1 and t_2 have disjoint domains and are vacuously compatible.

Extend

The $Extend$ operation takes three parameters a SPARQL–algebra term t, a variable name v and an expression e. The $Extend$ modifies each solution m of t by computing the value of $e(m)$ and binding v to this value.

Graph

As we explained in section 1.2.3, the evaluation of SPARQL queries is performed w.r.t. an *active* graph which is the graph against which TP and PP will be evaluated. The operator *Graph* takes two arguments: a graph name and a SPARQL–algebra term. The graph name can be:

- An IRI i in which case the solutions of $Graph(i, t)$ are the solutions of t when evaluated with G_k if there exists a binding $< u_k >, G_k$ with $u_k = i$ in the RDF dataset. The solutions of $Graph(i, t)$ are the empty multiset when there are no such u_k.

- A SPARQL–variable v in which case the multiset of solutions of $Graph(v, t)$ is the union for all named graphs $(< u_k >, G_k)$ of $Extend(Graph(u_k, t), v, u_k)$, i.e. the SPARQL–variable v ranges over the named graphs and for each graph, the term t is evaluated against this graph.

Distinct and reduced

As we have seen, SPARQL queries are evaluated in a multiset semantics: each mapping can be present several times in a query answer. The $Distinct(t)$ operator returns the multiset of the distinct values that are solutions of t. The *Reduced* operator is more relaxed: the multiset of solution to $Reduced(t)$ can be anything comprised between the multiset of solutions of $Distinct(t)$ and the multiset of solutions of t.

Group / Aggregation / AggregateJoin

The aggregation mechanism in SPARQL uses three auxiliary functions: *Group*, *Aggregation* and *AggregateJoin*. *Group* groups the mappings along a set of keys, *Aggregation* performs the computation of the aggregate and *AgregateJoin* merge together several aggregations.

Group The *Group* operator takes n expressions $e_1, \ldots, e_n$ and a term t. The *Group* operator is always used with an Aggregate and thus does not return a multiset of mappings but a function from n-uplet of RDF–terms to multisets of mappings.

Given a *Group* operator, we define for each solution of t its *key* as the n-uplets of values where the i-th value is obtained by computing e_i with the mapping m. The solution of a *Group* is a function from keys to the the multiset of mappings having that key (the function is only defined for keys corresponding to at least one mapping).

Aggregation The *Aggregation* operator takes k expressions $e_1, \ldots, e_k$, a function $func$ with some initial arguments $scalarvals$ and a term t (which has to be a *Group* operation) and returns a set of pairs of keys and values (where the key is the one produced by the *Group* operator).

The idea is that since t is a *Group* operation, its output will be of the form $\{k_1 \rightarrow M_1, \ldots, k_n \rightarrow M_n\}$ and the *Aggregation* will replace the M_i (which is a multiset of mappings) with $func((e_1, \ldots, e_k)(M_i), scalarvals)$ where $(e_1, \ldots, e_k)(M_i)$ designates the multiset of k-uplets which is the image of M_i by the function $m \rightarrow (e_1(m), \ldots, e_k(m))$. The function $func$ is expected to take the k-uplets and the $scalarvals$ and return a single RDF–term.

Therefore, if the solution of t is $\{k_1 \rightarrow M_1, \ldots, k_n \rightarrow M_n\}$ then the solution of $Aggregation((e_1, \ldots, e_k), func, scalarvals, t)$ is

$$\{k_1 \rightarrow func((e_1, \ldots, e_k)(M_1), scalarvals), \ldots, func((e_1, \ldots, e_k)(M_n), scalarvals)\}$$

AggregateJoin The idea of *AggregateJoin* is to patch together several *Aggregation* operators. Given k terms $t_1, \ldots, t_k$ that are all *Aggregation* over the same *Group* then the t_i will be evaluated to a set of the form $\{k_1 \to v_1^i, \ldots, k_n \to v_n^i\}$ where the $(k_j)_j$ are shared among all the the $(t_j)_j$ since they correspond to the same *Group* operator while the $(v_j^i)_j$ might be different for two different values of i (they all correspond to different aggregations).

The evaluation of $AggregateJoin(t_1, \ldots, t_k)$ will be be the multiset containing one mapping per key of the *Group* operator: the mapping $\text{agg}_1 \to v_j^1, \ldots, \text{agg}_k \to v_j^k$.

Note that the $(\text{agg}_i)_i$ refer to constants defined by the standard, the solutions of *AggregateJoin* are thus generally renamed afterward to the actual aggregate names (but it is not mandatory, one can ask for an aggregate without naming it, in which case the aggregate will be named agg_i for some i).

1.2.4 An example

Building on our example (see section 1.1.4), we can ask to concatenate all the labels corresponding to taxons descending from saurichias. First, we get the nodes corresponding to sub taxons of saurichias with the PP `?species ex:subTaxon*` `ex:saurichias` and enforce that they actually correspond to taxons with `?species rdfs:type ex:taxon` then we get their label with the PP `?species rdfs:label ?label`. This gives us the following *Group* operator that we name G:

$G = Group($ `?species`,
$BGP(\{$ `?species ex:subTaxon*` `ex:saurichias`,
`?species rdfs:type ex:taxon`
`?species rdfs:label ?label`
$\}))$

Then we compute two aggregates: one that gets the concatenation of labels and one that get species node (the `?species` is computed using an aggregation even if it is one of the columns that is in the *Group*). The term $t_1 = Aggregation(\texttt{?label}, Concat, separator =' |', G)$ computes the concatenation of labels: `?label` extracts the label from the multiset of G, *Concat* corresponds to the concatenation function with an initial argument $separator =' |'$ (the *scalarvals*) to indicate that between two concatenated terms there should be a $'|'$. The term $t_2 = Aggregation(\texttt{?species}, Sample, \emptyset, G)$ computes the `?species` for each `?species` by just taking any element (since they are grouped per `?species`). Finally we get the result with $A = AggregateJoin(t_1, t_2)$. Solutions of A will have two columns, agg_1 for the concatenation of labels and agg_2 for the species name.

The solutions of the *BGP* and G are:

BGP

mapping	*?species*	*?label*
m_1	$ex:tyrex$	$"Tyrannosaure"@fr$
m_2	$ex:tyrex$	$"Tyrannosaur"@en$
m_3	$ex:birds$	$"Aves"@en$
m_4	$ex:penguins$	$"Sphenisciformes"@en$

G

key	*val*
$ex:tyrex$	$\{m_1, m_2\}$
$ex:birds$	$\{m_3\}$
$ex:penguins$	$\{m_4\}$

The solutions of t_1, t_2 are:

t_1

key	val
$ex:tyrex$	$"Tyrannosaure\|Tyrannosaur"$
$ex:birds$	$"Aves"$
$ex:penguins$	$"Sphenisciformes"$

t_2

key	val
$ex:tyrex$	$ex:tyrex$
$ex:birds$	$ex:birds$
$ex:penguins$	$ex:penguins$

Finally the solutions of A are:

A

agg_2	agg_1
$ex:tyrex$	$"Tyrannosaure\|Tyrannosaur"$
$ex:birds$	$"Aves"$
$ex:penguins$	$"Sphenisciformes"$

1.2.5 Analysis of SPARQL–algebra

One advantage of the SPARQL–algebra is to pave the way for the formal study of SPARQL queries and in particular their transformation, optimization and evaluation. In particular, given a SPARQL–algebra term, we will concentrate on how to rewrite this term or what are the SPARQL variables that might appear in the solutions of a given term. These questions are actually linked since the rewriting of terms usually depends on the variables bound by SPARQL–algebra terms.

Early works (such as [SML10a]) on SPARQL already tackled these two questions together. They introduced a set of rewriting rules for a fragment of SPARQL–algebra (designed for a fragment of SPARQL1.0 under the set semantics). This set of rewriting rules depended on $cVars(t)$ and $pVars(t)$ that are syntactical functions returning a set of SPARQL variables that are *certain* (for $cVars$) to appear in a mapping solution of t and can *possibly* (for $pVars$) appear in a mapping solution of t.

1.3 SPARQL query evaluators and their optimization

There exists a wide variety of SPARQL query evaluation techniques and SPARQL query evaluators. In this section, we will overview a few existing SPARQL query evaluation techniques implemented in various SPARQL engines.

1.3.1 Storage of RDF graphs

The SPARQL query engines can be split into two main families: the direct engines and the data stores. A direct engine is not responsible for storing data and therefore for each query, we also need to pass the data that needs to be queried. A data store is a query engine that is capable of storing data and at each query the data store will answer with the query it has stored.

Both techniques have their advantages and drawbacks, a data store is generally more efficient as it has been able to store the data in its own format and therefore a data store can retrieve data very fast (for instance the data is generally indexed). On the opposite side,

a direct engine needs to read the whole dataset before being sure that something does not exist. While a direct engine is generally slower at the querying phase, a direct engine does not have a loading phase; therefore if we are only interested in a few queries over a dataset or if the dataset is very volatile, then the total time for a direct engine to read and answer might be less then the time needed by a data store to load the data and then process all theses queries.

Vertical partitioning

The Vertical Partitioning [AMMH07] is a technique introduced by Abadi *et al.* to store an RDF graph. The idea is to notice that in most RDF datasets there is only a limited number of distinct predicates. And by splitting the dataset among the various predicates, we obtain a gain in space usage and more importantly and a significant improvement in query response time for queries where all predicates are constants (which that is the case of most queries as pointed out in [AFMPF11]). This partitioning can be seen as a lightweight indexing method that is useful due to the shape of SPARQL queries and performs well in practice because is well compatible with distributed storage techniques.

Extended Vertical Partitioning

S2RDF [SPZSL16] relies on an Extended Vertical Partitioning (ExtVP), that mixes the ideas of Vertical Partitioning with the idea to partially precompute joins.

We will consider the sets of pairs (s, o) of subjetcs,objects for each predicate. For each pair A and B of such sets, we might compute $A \bowtie B$, the extVP will thus compute the set $A_{|B} \subseteq A$ that can be used in $A \bowtie B$. The idea is that $A \bowtie B$ is always equals to $A_{|B} \bowtie B_{|A}$ but $A_{|B}$ and $B_{|A}$ might be much smaller than A and B and thus $A_{|B} \bowtie B_{|A}$ might be faster to compute

Storing for each pair A, B the sets $A_{|B}$ and $B_{|A}$ might be very large. But they actually store these reduced sets ($A_{|B}$ and $B_{|A}$) only when they are empty or small compared with A and B.

Multiple indexes

RDF-3X [NW08] is a SPARQL query engine that put the focus on speed. Given a set of triples, RDF-3X build indexes for all of the 6 permutations of "subject", "predicate" and "object" which allows to retrieve the solutions in an ordered fashion for any triple pattern. RDF-3X then creates the solutions to a BGP with a set of merge joins. The join ordering is decided using a cardinality estimation that mixes statistics on the six permutations plus statistics over a set of frequent paths.

RYA [PCR12] is a SPARQL query engine based on Apache Accumulo for the data storage and on the OpenRDF Sesame framework to process RDF reasoning and SPARQL querying. RYA improves on the RDF-3X storage. The authors notice that storing three of the six permutations is enough for an efficient retrieval of solutions to individual TP.

Graph partitioning

Some distributed SPARQL query evaluators adopt a *partitioning* technique. In this partitioning scheme, the graph is split among the several workers with respect to the graph connec-

tivity. The goal of the partitioning phase is to split the data among the several workers so that the two sets of nodes stored in two different workers don't have much connections.

This technique is particularly useful for SPARQL stores that adopt a "partial evaluation and assembly" method of evaluation. At the query evaluation time, the query to be sent to all workers, each solving the query on its local graph. Then, the local solutions are merged, when needed. This technique is at the heart of several SPARQL query evaluators (e.g. [PZO+16]). Note that in the *partitioning* scheme, each worker has to store its local graph which generally requires an indexing technique for this local data.

1.3.2 Query Execution Plan optimization techniques

Most of the SPARQL query engines have a Query Execution Plan, that is an internal representation of SPARQL queries. This QEP is initially just a translation of the SPARQL query but then the query engine optimizes it in several passes.

Some query engines (such as Apache Jena ARQ or Pig Latin [SPZL11]) can rely on heuristic optimizations. Heuristic optimization consists in detecting changes in the QEP that will almost always lead to a more efficient QEP. For instance, when possible, pushing filters into joins always reduces the size of the join which generally improves the query time (and never increases it by a large factor).

Heuristic optimization is used by query engines that to not have a preprocessing phase where they load the data but other stores sometimes also rely on heuristics as successfully exploiting statistics is often a challenge. CliqueSquare [GKM+15] is a SPARQL query engine relying on Hadoop Map-Reduce and focusing on the fast evaluation of complex BGP. The query optimization performed by CliqueSquare consists in finding "flat" plans in two steps: first they decompose the queries into cliques (hence the name), these cliques will correspond to the first map-Reduce step. Then they combine partial results using n-ary joins while trying achieve "flat plans" (i.e. DAG of joins with a minimal depth). The data is stored into the Hadoop file system so that the first pass of join can be performed locally (without a shuffle phase).

The query engines that do load the data into memory usually take advantage of this phase to collect some statistics over the data they are loading. This allows them to consider more QEP and decide which one they will be using by relying on the statistics they collected.

Conclusion

In this chapter we have presented RDF and SPARQL. RDF is a data model describing graphs. SPARQL is a rich language to query RDF graphs. SPARQL query evaluation raises challenges: how to evaluate such a rich query language on graphs that are, by nature, very large. More precisely: how can we find efficient query plans to handle even simple queries but on very large datasets? How can we find efficient plans for complex queries on which today's evaluators tends to fail even on relatively small datasets? Since the relational model was at the center of most of the research in database, before addressing those questions, we need to examine into the relational model and what it can offer to evaluate SPARQL queries.

CHAPTER 2

The relational model and beyond for graphs queries

While the computers take their name from their ability to perform long and tedious computations, storing and handling large quantities of data has been one of the main task for computers since their inception. In fact, machines capable of handling data predate the computers by half a century. For instance, the tabulating machine was invented for the 1890 US census.

The relational model and its tuple calculus were introduced by Codd in 1970. Compared with models existing in 1970, the great novelty of relational model was not directly its storage model but the fact that it had a high-level query language and allowed to move the burden of optimizing data access from the programmer to the query evaluator. The users would store their data as relational tables and eventually specify indexes on them. The users would not specify how the data is stored nor how it is accessed but they would rather only specify what data they wanted (in the form of a query) and the database engine would automatically translate this query into efficient code.

These ideas quickly gained traction, as they enabled

users to write data-crunching programs more easily, and were the basis for the Structured Query Language (SQL) which soon became the de facto standard for databases. Furthermore it fostered the adoption of rich, high-level query languages such as SPARQL.

A lot a research was poured into the relational model and as SPARQL *bears some resemblance with it, it is natural to try to adapt this research for the optimization of* SPARQL. *That is why, in this chapter, we give the fundamentals of the relational model and algebra, we then present some extensions to this model designed to handle more powerful features (such as the recursion that appears in* SPARQL*), we continue with the presentation of another powerful data-handling formalism: Datalog and we finish by exploring the relation between relational query languages and* SPARQL.

2.1 The relational model

The relational model was introduced by Codd at the beginning of the 70s [Cod70]. The novelty of the approach does not lie so much much within its data model but, as we will see, in the query languages and the optimization on these query languages allowed by the relational model.

2.1.1 Definitions

A relational database describes objects with relations. These objects are values with a type (such as integers, strings, custom types, etc.). A relational database is simply a set of *relations* between these objects. We now define the properties needed to introduce relational database but as we will see, relations can be seen as tables and that is why we sometimes use "column" and "row".

Definition 5. *The relational model supposes a set of column names C (we can think of it as the set of strings).*

Definition 6. *A* tuple schema *is a sequence of* attributes *and an* attribute *is a pair (d, c) where d is a domain of values and c belongs to C. The domain d is the* column type *and c is the* column name.

Definition 7. *A* tuple *(or* row*) is a sequence of ordered pair of values and domain. A tuple $(v_1, d_1), \ldots, (v_n, d_n)$ is* compatible *with a tuple schema $(d'_1, c_1), \ldots, (d'_m, c_m)$ when $m = n$ and for all $1 \leq i \leq n$, $d'_i = d_i$ (the domains match).*

Definition 8. *A* database schema *is a triple (D, R, h) where:*

- *D is the domain of atomic values,*
- *R is a finite set of relations names,*
- *h is a function from R to tuple schemas.*

Definition 9. *A* relation *is a pair of a tuple schema and a set of tuples compatible with this schema. The tuple schema is also called the* heading *while the tuples are called the* body *of the relation.*

Definition 10. *A* relational database *$S = (D, R, h)$ contains a set of relations R such that heading of $r \in R$ is $h(r)$ and the values of tuples in the relation all belong to D.*

As we explained, relations can be seen as tables where the columns are typed and named by attributes (or column names and column types) and the entries are the tuples (or rows). Note however that the relational model does not define any order between the tuples. In this view, a relational database is a set of named tables.

2.1.2 An example

Let us consider the following example. We have a first relation describing theaters, with their name, their address and the number of rooms and a second relation describing shows with their name, the date of the representation and the name of theater they take place in. In table representation, we have:

Theaters

Name	Address	Rooms
"La Nef"	"bd Edourd Rey"	7
"Le Mélies"	"caserne de Bonne"	3
"Le Club"	"rue Phalanstère"	3

Movies

Title	Date	Theater
"Inception"	12/08/2010 20h	"Le Mélies"
"Toy Story 3"	13/08/2010 17h	"Le Club"
"Toy Story 3"	13/08/2010 20h	"Le Club"
"Toy Story 3"	10/08/2010 17h	"Le Mélies"
"Akmareul boatda"	10/08/2010 16h	"Le Club"
"How to train your dragon"	12/03/2010 18h	"Pathé Chavant"

In this example the relation $Theaters$ has schema $(Name, String), (Address, String), (Rooms, Integ$ while the relation $Movies$ has schema $(Title, String), (Date, Datetime), (Theater, String)$ using the data type $String$, $Integer$ and $Datetime$.

2.1.3 The relational algebra

The relational algebra is a query language introduced by Codd for the relational model. There exists other formalisms such as the tuple relational calculus and the domain relational calculus which are very "logical" query language (you specify the data you want with a formula and the model of this formula is your answer). In contrast, the relational algebra is actually much closer to a classical programming language. In a sense, a relational algebra term not only expresses what data we are interested in but it also has a compositional bottom-up semantics that describes a way to retrieve and compute the data (even though the compositional semantics provided by the query language might actually not be the most efficient way to actually compute the solutions of a query).

Relational algebra terms

The relational algebra terms are recursively made up of:

- Relations (any $r \in R$ e.g. $Movies$ for our running example);
- Selections noted $\sigma_f(\varphi)$ where f is a filter condition and φ is a relational algebra term. The filter condition is a boolean formula comparing the columns of $FC(f)$ (with $FC(f) \subseteq type(\varphi)$, see next section for the notion of type);

- Projections noted $\pi_P(\varphi)$ where φ is a relational algebra term and P is a subset of the type of φ;
- Renaming noted $\rho_{a/b}(\varphi)$ where φ is a relational algebra term and $a \in C \setminus type(\varphi)$ and $b \in type(\varphi)$;
- Natural joins noted $\varphi_1 \bowtie \varphi_2$ where φ_1 and φ_2 are two algebra terms;
- Antijoins noted $\varphi_1 \triangleright \varphi_2$ where φ_1 and φ_2 are two algebra terms.
- Unions noted $\varphi_1 \cup \varphi_2$ but where the union is restricted to pairs of terms that have the same "type".

Interpretation of the relational algebra terms

Relational algebra terms have, by design, a compositional semantics which means that the evaluation of a term can be performed bottom-up as the evaluation of a subterm depends only one the subterm and not on its context. More precisely, the interpretation of a term φ depends on the database (and on φ). When the term is executed, it returns a set of tuples that are all compatible with a schema s which is the *type* of φ.

The interpretation of a relational algebra term against a database (D, R, h) is recursively defined as follows:

- For a relation $r \in R$, its interpretation is simply the set of tuples the database associates with this relation and its type is $h(r)$.
- For a selection $\sigma_f(\varphi)$ its interpretation is the set of tuples that are solution of φ and passing the filter condition f. The type of $\sigma_f(\varphi)$ is the type of φ.
- For a projection $\pi_P(\varphi)$, its interpretation returns the set of tuples that can be obtained by restricting a tuple solution of φ to the attributes listed by P. The projection supposes that the type of φ contains the columns P and the resulting type (after projection) is the restriction of this type to P and thus P.
- The interpretation of a renaming $\rho_{a/b}(\varphi)$ corresponds to the interpretation of φ but where we change the name of the column b to a.
- The set of tuples that are solutions to $\varphi_1 \bowtie \varphi_2$ corresponds to the set of tuples such that the restriction to $type(\varphi_1)$ corresponds to a solution of φ_1 and the restriction to $type(\varphi_2)$ corresponds to a solution of φ_2.
- The set of solutions of the antijoin $\varphi_1 \triangleright \varphi_2$ is the subset of solutions of φ_1 that cannot be joined with mappings of φ_2, hence the name antijoin. Equivalently, it is the set of t that are solutions of φ_1 such that for all t' such that the restriction of t' to $type(\varphi_1)$ is t then the restriction of t' to $type(\varphi_2)$ is not solution of φ_2.
- The set of tuples that are solutions of $\varphi_1 \cup \varphi_2$ is the union of solutions to φ_1 and φ_2. The union is only defined for terms that have the same type and the type of the union is their common type.

Example We can ask for the dates and addresses of the projections of "Toy Story 3" with:

$$\pi_{Address,Date}\Big(\sigma_{(Title=\text{“Toy Story 3”})}\big(Theathers \bowtie \rho_{Name/Theater}(Movies)\big)\Big)$$

And to retrieve the set of names of theaters appearing in $Movies$ that do not have an entry in theaters:

$$\pi_{Name}\Big(\rho Name/Theater(Movies) \triangleright Theaters\Big)$$

2.1.4 Rewriting relational algebra terms

As we have seen, the semantics of relational algebra terms directly provides a way to evaluate them. This "naive" evaluation might not be the most efficient way of evaluating terms. For instance $A \cup A$ would imply to compute A twice and in $\sigma_{(Title=\text{“Toy Story 3”})}\big(Theathers \bowtie \rho_{Name/Theater} Movies\big)$ it would be more efficient to compute the equivalent $Theathers \bowtie \rho_{Name/Theater}\big(\sigma_{(Title=\text{“Toy Story 3”})}(Movies)\big)$

One of the nice properties of the relational algebra is that terms can be rewritten into equivalent terms, i.e., terms computing the same set of solutions. This allows for a first optimization strategy: we rewrite the term into several others computing the same set of tuples as the initial term; we select heuristically the most efficient and then we evaluate this new term.

The rewriting rules are depicted in figure 2.1

2.1.5 Codd theorem

As we briefly mentioned, there exists several query languages:

- the tuple relational calculus, originally introduced by Codd
- the domain relational calculus, a simplification of the tuple relational calculus,
- and the relational algebra (here considered without extensions).

A natural question is the relation between these languages and one of the main theorems for the relational model is the Codd theorem. The Codd theorem proves that these three languages actually have the same expressive power. This theorem allows us to define the notion of *relational completeness* which is the property for a query language to have the same expressive power as the query language we presented above.

2.1.6 Common extensions

New operators in the relational algebra as syntactic sugar

It is also possible to include other operators in the relational algebra such as the division, the equijoin, or the set difference. However from a logical point of view they do not increase the expressiveness of the language (i.e. they do not allow to express new queries, they only simplify some queries).

For instance the set difference is just a particular case of the antijoin. The set difference, noted $\varphi_1 - \varphi_2$, corresponds to the set of mappings φ_1 that are not solution of φ_2. The

$\sigma_{f_1}(\sigma_{f_2}(\varphi)) \leftrightarrow \sigma_{f_1}(\sigma_{f_2}(\varphi))$		commutativity of selections
$\varphi_1 \bowtie \varphi_2 \leftrightarrow \varphi_2 \bowtie \varphi_1$		commutativity of joins
$\varphi_1 \cup \varphi_2 \leftrightarrow \varphi_2 \cup \varphi_1$		commutativity of unions
$\varphi_1 \bowtie (\varphi_2 \bowtie \varphi_3) \leftrightarrow (\varphi_1 \bowtie \varphi_2) \bowtie \varphi_3$		associativity of joins
$\varphi_1 \cup (\varphi_2 \cup \varphi_3) \leftrightarrow (\varphi_1 \cup \varphi_2) \cup \varphi_3$		associativity of unions
$(\varphi_1 \triangleright \varphi_2) \triangleright \varphi_3 \leftrightarrow (\varphi_1 \triangleright \varphi_3) \triangleright \varphi_2$		"associativity" of antijoins
$\pi_{P_1}(\pi_{P_2}(\varphi)) \leftrightarrow \pi_{P_1}(\varphi)$	when $P_1 \subseteq P_2 \subseteq type(\varphi)$	involutivity of projections
$\sigma_f(\varphi_1 \bowtie \varphi_2) \leftrightarrow \sigma_f(\varphi_1) \bowtie \varphi_2$	when $FC(f) \subseteq type(\varphi_1)$	distributivity of selection
$\sigma_f(\varphi_1 \bowtie \varphi_2) \leftrightarrow \sigma_f(\varphi_1) \bowtie \varphi_2$	when f concerns columns of φ_1	distributivity of selection
$\pi_P(\varphi_1 \bowtie \varphi_2) \leftrightarrow \pi_{P \cap type(\varphi_1)}(\varphi_1) \bowtie \pi_{P \cap type(\varphi_2)}(\varphi_2)$	when $type(\varphi_1) \cap type(\varphi_2) \subseteq P$	distributivity of projection
$\pi_P(\varphi_1 \cup \varphi_2) \leftrightarrow \pi_P(\varphi_1) \cup \pi_P(\varphi_2)$	when $type(\varphi_1) = type(\varphi_2)$	distributivity of projection
$\sigma_f(\varphi_1 \triangleright \varphi_2) \leftrightarrow \sigma_f(\varphi_1) \triangleright \varphi_2$		distributitivty of selection
$\sigma_f(\varphi_1 \triangleright \varphi_2) \leftrightarrow \sigma_f(\varphi_1) \triangleright \sigma_f(\varphi_2)$	when $FC(f) \subseteq \varphi_2$	distributitivty of selection

Figure 2.1: Rewrite rules of the relational algebra

set difference is only defined for pairs of relational terms sharing the same type and thus it corresponds to a special case of the antijoin.

Another example is the equijoin. In the natural join $\varphi_1 \bowtie \varphi_2$, we combine solutions of φ_1 with solutions of φ_2 that agree on their common domain (for all $c \in type(\varphi_1) \cap type(\varphi_2)$ we need that the pair of tuples have the same value for c). An equijoin extends this set of constraints with a set of ordered pairs $(a_1 = b_1) \dots (a_k = b_k)$ and ask for all solution t and for all $1 \leq i \leq k$ that the a_i value of t is equal to its b_i value. Note that equijoin do not add expressiveness as they can be encoded with a filter or a set of renaming operations but they sometimes simplify terms.

For instance, in the query retrieving the set of pair (date,address) of projections of "Toy Story 3" we had to join $Theater$ and $Movies$ so that the the $Name$ of $Theaters$ is the same as the $Theater$ of $Movies$ which we can note $Theater \bowtie_{(Name=Theater)} Movie$

Outer joins

The left outer join $\varphi_1 ⟕ \varphi_2$ corresponds to the natural join $\varphi_1 \bowtie \varphi_2$ augmented with the antijoin $\varphi_1 \triangleright \varphi_2$. However in the relational algebra, all tuples that are solutions to a given term should have the same type, that we why the mappings solutions of $\varphi_1 \triangleright \varphi_2$ are "augmented" with a constant designing a missing value (a NULL in the SQL semantics) to be placed in the columns $type(\varphi_2) \setminus type(\varphi_1)$.

This missing value constant is usually not considered as a standard value. For instance, in SQL, the boolean formulas are evaluated under a three-valued logic that is similar to the three-valued logic of SPARQL. And, still in SQL, a NULL value is not equal to anything, not even to itself. It makes sense as NULL represents missing values and therefore there is no reason to suppose that two NULL represent the same value but it has a profound effect on the semantics of SQL. For instance, a join of two expressions $\varphi_1 \bowtie \varphi_2$ never returns tuples t such that the restriction of t onto $type(t_1) \cap type(t_2)$ contains a NULL.

We presented here the left outer join but by symmetry it is possible to define the right outer join (the right outer join between φ_1 and φ_2 being the left outer join between φ_2 and φ_1) and the full outer join (which is the union of left and right outer joins).

Note that since the NULL value is not equal to anything (including itself) it changes the intuitive definition of join that we presented here to include such values and it also changes the set of valid rewrite rules. For instance, in the relational algebra we have $A \bowtie A = A$ but this is not true anymore if we include NULL values and that A contains NULL.

Computing over values

In the "traditional" relational model it is not possible to compute over values. For instance one cannot ask about the tuple such that the a column times the b column is equal to the c column (where a, b and c are e.g. integers). In general values of the domain can only be tested for equality.

We can add to the relational calculi formulas computing over some data types. For instance, we can ask that the integers are equipped with a comparison symbol $\leq$ or with a $+$ operator.

Similarly to the SPARQL algebra *Extend* operator, we can also enrich the syntax of relational algebra with expressions and an *extended projection* that compute new columns using these expressions.

Aggregates

One other common extension of the relational algebra consists in adding aggregates. An aggregate groups tuples based on a set of columns G (two tuples belong to the same group if they have the same value on the columns of G) and then for each group, the aggregate performs a set of computation. It is once again very similar to SPARQL aggregation as SPARQL aggregation was inspired by the SQL syntax.

Bag semantics

The relational model queries have a set semantic which means the solutions to a query are defined as a set. However similarly to what has been presented for SPARQL we can define a semantic for the relational algebra terms that has a multiset (or bag) semantics. In fact, one the main differences between SQL and the relational algebra is the use by SQL of such a bag semantics.

It is possible [GMUW09] to adapt the work and theorems for the relational algebra with a bag semantics but many theorems become invalid or we need to add some additional conditions.

Transitive closures

The extensions we have presented up to now are either syntactic, or they extend the relational model with new values (the default value for outer joins) or they suppose that we can perform computations on the values of D. However they are natural and interesting queries that we cannot express in the relational calculi nor in the relational algebra.

Suppose for instance that the database contains a binary relation R (e.g. representing a directed graph). The transitive closure (noted R^+) of R is the pair of nodes (a, b) such that b is reachable from a through the relation R. The transitive closure of relations belong to these queries that cannot be expressed in the relational algebra or the relational calculi (see the theorem 1 of Aho and Ullman [AU79]).

One other way to introduce transitive closures to the relational algebra is to add an atomic formula R^+ where R is a relation of the database. This is essentially what has been done in e.g. the XPath language where we can only use the transitives closure on the atomic relations. However when φ is a relational algebra term representing a binary relation R we might want to include in the language a way to represent R^+.

A proposal by Agrawal [Agr88] follows up this direction. Its α-extended relational algebra comprised the same rules of formation for formula than the relational algebra but it also adds a new one: if φ is a term computing a binary relation between two columns of the same type then $\alpha(\varphi)$ also is a term, with the same type, and it computes the transitive closure. Agrawal [Agr88] notes that pushing selections into transitive closure is sometimes possible but does not provide an effective criterion when it is possible.

Recursive queries

Transitive closures are a special kind of recursive queries for binary queries. There exists multiple ways of introducing recursion into the relational model from both syntactical and semantical perspectives. One key difference from a semantical perspective consists in distinguishing between inflationary fixpoints and noninflationary ones.

For the relational calculi the standard notation is an operator $\mu X.\varphi$ with X a free variable of φ in reference to the μ-calculus. We note CALC+μ for this calculus and CALC+μ^+ for the inflationary version. The semantics of μ in CALC+μ^+ corresponds to the limit of U_n in $n \to \infty$ with $U_0 = \emptyset$ and $U_{n+1} = U_n \cup \varphi(X = U_n)$ (with X a free variable of the relational calculus term φ) while in the second case it corresponds to the limit with $U_{n+1} = \varphi(X = U_n)$. This second case is more general (as the union is possible) but does not necessarily converge.

Aho *et al* proposed to introduce a least fixpoint operator in the relational algebra and even showed in their theorem 2 that it is possible to push some selections inside of fixpoint. A similar proposal [HA92] (inspired by the μ-calculus and thus diverging only marginally from the notations of Aho and Ullman [AU79]) provides a notation for recursive queries with mutually recursive fixpoints. However their work on optimization only provides criteria for pushing the selection inside the transitive closure of regular binary relations which is therefore quite similar to the Theorem 2 of Aho and Ullman.

Recursive queries and transitive closures in particular have now their equivalent in the SQL standard since its 1999 version of the standard that allows for recursive Common Table Expression (and multiple vendors have their own dialect for recursive queries even before that). But as we will see, it is not well supported nor well optimized by SQL engines.

2.1.7 Optimization of relational algebra and SQL

One of the great success of the relational algebra was the ability for a user to write its query only specifying the data she expects and not how to retrieve it.

SQL is very close with the relational algebra by design. However the reach of static analysis methods in the realm of SQL is much more limited due to the bag semantic and that prevents some type of optimization (such as [CM77]). Once the SQL query has been translated into an algebraic form, the main optimization process consists in finding equivalent forms through the use of rewrite rules.

For each rewritten term, the SQL optimizer chose the one that has an estimated running time, or *cost*, that is the lower. This cost estimation thus mainly rely on a *cost model* describing the cost of each individual operation (in terms of time spent processing it per solution) plus a cardinality estimation scheme (to estimate the number of solution to all the subterms).

2.1.8 Cardinality estimation schemes

Estimating the number of solutions for a query has long been viewed as a key element in the optimization of queries and it is a well-studied problem in the relational world [PSC84]. There exists a wide variety of techniques that have been very successful in the relational world such as histograms [Ioa03, OR00].

These techniques have been, however, less successful for the semantic web [EM09, NM11]. The main reasons for that is the heterogeneous nature of RDF [NM11] and the fact that SPARQL queries usually contain a lot of self-joins that are notoriously hard to optimize [PT08].

Various works have tackled the specific issue of cardinality estimation for SPARQL. A line of work [SSB+08] introduced the "selectivity estimation" now in use in several SPARQL evaluators [ZYW+13]. The "selectivity estimation" assumes the statistical independence of the various parts of a TP. Variants of this method have been implemented in popular SPARQL query evaluators (e.g. in RDF-3X see below).

A second line of work [KRA17] takes as input an actual schema and produces an optimized query plan based based on information extracted from the schema. A third line of work [NM11, GN14] tries to derive the implicit schema of an RDF graph by fitting nodes into characteristic sets, or by summarizing [GSMT14] the graph into large entities.

2.2 Datalog

Prolog is a general purpose programming language. Datalog is a fragment of Prolog focused on data. The specificity of Prolog (and thus Datalog) is that it is a declarative logic programming language. The programmers describe the data they expect and the interpreter computes the data in a supposedly efficient manner.

2.2.1 Datalog programs

There exists many variations of Datalog in the literature. We will present here a Datalog with recursion and negation but where the interaction between recursion and negation is limit to only allow so-called stratified Datalog programs. A stratified Datalog program consists in a sequence of strata. Each stratum defines a certain number of relations through a set of rules per relation. There are three types of rules and each definition of a rule is of one of these three forms:

- it can be given as a n-uplet for which the relation holds. This kind of rules can be seen as an "input". This rule is written as "$R(x_1, \ldots, x_n)$." (where the (x_i) are constants);
- it can be a difference between two n-ary relations. This rule is written "$R(X_1, \ldots, X_n) : -R_a(X_1, \ldots, X_n),\ \text{not}\ R_b(X_1, \ldots, X_n)$" where the upper case $X_1, \ldots, X_n$ refer to variables. In this type of rules, the negated relation (i.e. R_b) has to be defined by a stratum that is strictly below the current stratum while R_a can be defined in a stratum below or at the current stratum;
- it can be defined as a rule corresponding to the conjunction of several other relations that are defined as the same stratum or below; it is written as "$R(X_1, \ldots, X_n) : -R_1(X_1^1, \ldots, X_1^{k_1}), \ldots, R_\ell(X_\ell^1, \ldots, X_\ell^{k_\ell})$" where the set of $X_1 \ldots X_n$ plus a set of existential variables $Y_1, \ldots, Y_K$ is equal to the set of variables appearing in the R_j (i.e. the set of X_i^j).

We can have multiple rules defining the same R but all the rules defining R must belong to the same stratum and be of the same form. A relation defined by the set of n-uplets for which the relation holds is called an extensional rule in opposition to the two others kinds of rules that define intentional relations.

2.2.2 Interpretation of Datalog programs

The interpretation of a Datalog program is done stratum by stratum and returns for each relation the set of n-uplet satisfying the relation. For each stratum we compute the smallest fixpoint such that:

- in an extensional relation, its interpretation contains the n-uplets of its definition;

- for each rule $R(X_1, \ldots, X_n) : -R_a(X_1, \ldots, X_n), notR_b(X_1, \ldots, X_n)$ it is the set of n-uplets solution of R_a that are not solution of R_b;

- for each rule $R(X_1, \ldots, X_n) : -R_1(X_1^1, \ldots, X_1^{k_1}), \ldots, R_\ell(X_\ell^1, \ldots, X_\ell^{k_\ell})$ if it exists values for $X_1 = x_1, \ldots, X_n = x_n, Y_1 = y_1, \ldots, Y_k = y_k$ such that $R_1(X_1^1, \ldots, X_1^{k_1}), \ldots, R_\ell(X_\ell^1, \ldots, X_\ell^{k_\ell})$ then $x_1, \ldots, x_n$ has to be included in R.

Note that the restriction we imposed on negation (that the negated part has to come from a stratum that is below) is mandatory to have a fixpoint that always terminate. We can only make sure that an n-uplet is not solution of a relation after the fixpoint has been reached, therefore we can only negate relations that are below the stratum on which we are computing the fixpoint. Furthermore it avoids the problem of "counter-intuitive" relations such as $R(x) : -\ not\ R(x)$.

2.2.3 Example

Suppose that we have the relations of example 2.1.2 encoded as the extensional relations $Movies(X, Y, Z)$ (where X represents the $Title$, Y the $Date$ and Z the $Theater$) and $Theaters(X, Y, Z)$ (representing respectively the $Name$ for X, the $Address$ for Y and the $Rooms$ for Z).

To query the dates and addresses for projections of "Toy Story 3" we have the following Datalog program:

```
date_address(X,Y) :- Movies(``Toy Story 3'', X, Z), Theaters(Z,Y,W)
```

And to query the theaters referenced by the relation $Movies$ but not by $Theaters$:

```
TheathersT(X) :- Theaters(X,Y,Z)
MoviesT(X) :- Movies(X,Y,Z)

missingTheaters(X) :- MoviesT(X), not TheatersT(X)
```

2.2.4 Variants of Datalog

There are several extensions of Datalog. Some include aggregates, some include user-defined functions or the handling of `NULL` values *à la* SQL. We present here two variants of Datalog that will correspond to fragment of the algebra we propose (the constant and the linear fragments).

Non recursive Datalog

A Datalog program is said non recursive when there exists a stratification such that all rule bodies refer to rules defined in strata that are below the stratum of the rule definition. It has been proven that non recursive Datalog actually is relational complete, it has exactly the same expressive power as the relational algebra.

Linear Datalog

A stratified Datalog program is said to be linear when at each stratum and in each conjunctive rule $R(X_1, \ldots, X_\ell) : -R_1(X_1^1, \ldots, X_{k_1}^1) \wedge \cdots \wedge R_n(X_1^n, \ldots, X_{k_n}^n)$ of the stratum contains at most one the conjunctive part (i.e. at most one of the R_i) that is defined at the current stratum.

2.3 Encoding SPARQL queries into relational query languages

Using the relational model to encode RDF graphs is relatively easy. The RDF graphs can, for instance, be stored as quadruplets of strings (s, p, o, g) where s, p, o represents an edge from s to o labeled with p and this edge belongs to the graph g.

As we have seen in the first chapter, SPARQL queries can be translated into a SPARQL–algebra that has a compositional semantics much like the relational algebra. However there are fundamental differences in the semantics of SPARQL and relational query languages which makes the encoding of SPARQL into a relational query language hard.

There has been, however, a variety of papers on the subject. For instance, [Cyg05] tackled this subject even before the official release of SPARQL 1.0, [CLF09] studied the translation from the complete SPARQL 1.0, the translation of Property Paths (which were introduced in SPARQL 1.1) was examined in [YGG13]. To the best of our knowledge, there has no publication on the direct translation from SPARQL 1.1 to SQL but as we have explained in chapter 1, there have been attempts to translate the more advanced features directly into a simpler fragment of SPARQL which can then be translated into SQL.

Furthermore, there are implementations of such translations. For instance, we can cite OnTop which is a software designed to query relational databases using the SPARQL language. We now review the main difficulties of such translations and whether our approach will suffer from such difficulties or not.

2.3.1 Missing values

SPARQL terms do not describe relations as all the solutions to a given SPARQL–algebra term do not necessarily share a common domain. For instance, in a term such as `{?a ?b ?c} UNION { ?a ?b ?d }`, only one of the variables `?c` and `?d` is defined in each solution.

As we have seen, there are relational query languages supporting missing values; for instance SQL has a special value `NULL`. If we can detect which variables might be undefined in solution, we can translate the join in the relational algebra using a special condition. For instance if R_1 and R_2 are translations of two SPARQL–algebra terms sharing a variable $?a$ that might not be defined then the join condition would be `(R1.a=R2.a OR R1.a IS NULL OR R2.a IS NULL)`. And the same problem arises for filter conditions.

In order to use these `NULL`, we need to be able to detect which variables are *possibly* bound by a SPARQL–algebra term. Furthermore, to produce more efficient terms (that do not contain a lot of `IS NULL`), we also want to be able to determine which variables are *certainly* bound. This analysis of *cVars* (certainly bound variables) and *pVars* (possibly bound variables) has been proposed in [SML10a].

This analysis is only a syntactic approximation but there is a good reason why the analysis is sometime imprecise. It can be shown that a precise analysis of whether a column will be bound or not is, at least, as hard as detecting whether a term is included into another which is an undecidable task. Note that for the translation to be valid, the approximation needs to overapproximate $pVar$ and underapproximate $cVarq$.

2.3.2 Filtered Optional

Another mismatch between relational query languages and SPARQL is for Optional. Optional are translated into the SPARQL–algebra operator $LeftJoin$ which is a conditional left join. Translating such conditional fixpoint into unconditional left join introduce a lot of redundancies. Our algebra will be equipped with only an unconditional left join and therefore we will present how to handle conditional left joins in section 3.5.6.

2.3.3 Recursive queries

The property paths introduced in SPARQL 1.1 can contain recursive part. It is possible since its 99 version to express recursion in SQL. For instance, [YGG13] proposes to translates property paths such as p^* can be expressed with the SQL query:

```
WITH closure(s,o) AS (
     SELECT P1.s, P1.o FROM P1
     UNION ALL
     SELECT C.s, P1.o FROM closure C, P1 WHERE C.o = P1.s
) CYCLE s SET cyclemark TO 'Y' DEFAULT 'N' USING cyclepath,

  P(s,o) AS (
     SELECT DISTINCT s, o FROM closure
     UNION
     SELECT T.s as s, T.s as o FROM triples T
  )
```

Such a translation raises two problems for SQL databases vendors. First, not all SQL databases support the recursivity (for instance, the popular `mysql` does not support it) such a query is hard for database vendors to optimize. In fact, most database optimizers do not try to optimize recursive queries.

Conclusion

In this chapter, we briefly presented the relational model and its query languages. The research on the relational algebra and SQL in particular are a great basis on which we can build query evaluators and optimizers. However, in their current form, relational-based SPARQL query evaluator suffer from inefficiencies and there are multiple reasons at the root of these inefficiencies:

- relational query evaluators do not optimize well some type of queries such as recursive queries;

- relational query languages and SPARQL do not match well semantically;
- and finally some features of SPARQL are hard to evaluate efficiently.

To address these two first points, we will now present our μ-algebra inspired by the relational algebra but adapted to handle both the recursivity part and the non-relational (missing values) part.

Part II

The μ-algebra for the execution of SPARQL queries

Table of Contents

3 The μ-algebra **61**

3.1 Syntax & Semantics . . . 63
3.2 Examples of μ-algebra terms . . . 68
3.3 Restriction on μ-algebra: constant and linear recursions . . . 70
3.4 Relationship between μ-algebra and existing relational variants . . . 72
3.5 Translation from SPARQL . . . 73

4 Analysis & transformation of μ-algebra terms **81**

4.1 Preliminaries: several examples of terms to be rewritten . . . 83
4.2 Effects of μ-algebra terms . . . 84
4.3 Decomposed fixpoints . . . 90
4.4 Typing μ-algebra terms . . . 93
4.5 Normalizing rules for μ-algebra terms . . . 96
4.6 Producing rules . . . 100
4.7 Ad-hoc rules . . . 104
4.8 Rewriting algorithm . . . 105
4.9 Example of rewriting . . . 107

5 Evaluation of μ-algebra terms **111**

5.1 General bottom-up evaluation for μ-algebra terms . . . 113
5.2 Bottom-Up cost model for μ-algebra terms . . . 117
5.3 Single core bottom-up evaluation of μ-algebra . . . 119
5.4 Towards a distributed μ-algebra evaluator . . . 119

6 Cardinality estimation of μ-algebra terms **125**

6.1 Summaries . . . 127
6.2 Computing collection summaries representing the solutions of a single TP . . . 133
6.3 Optimization of distributed BGP query plans with an over-estimation . . . 135
6.4 Extensions . . . 137

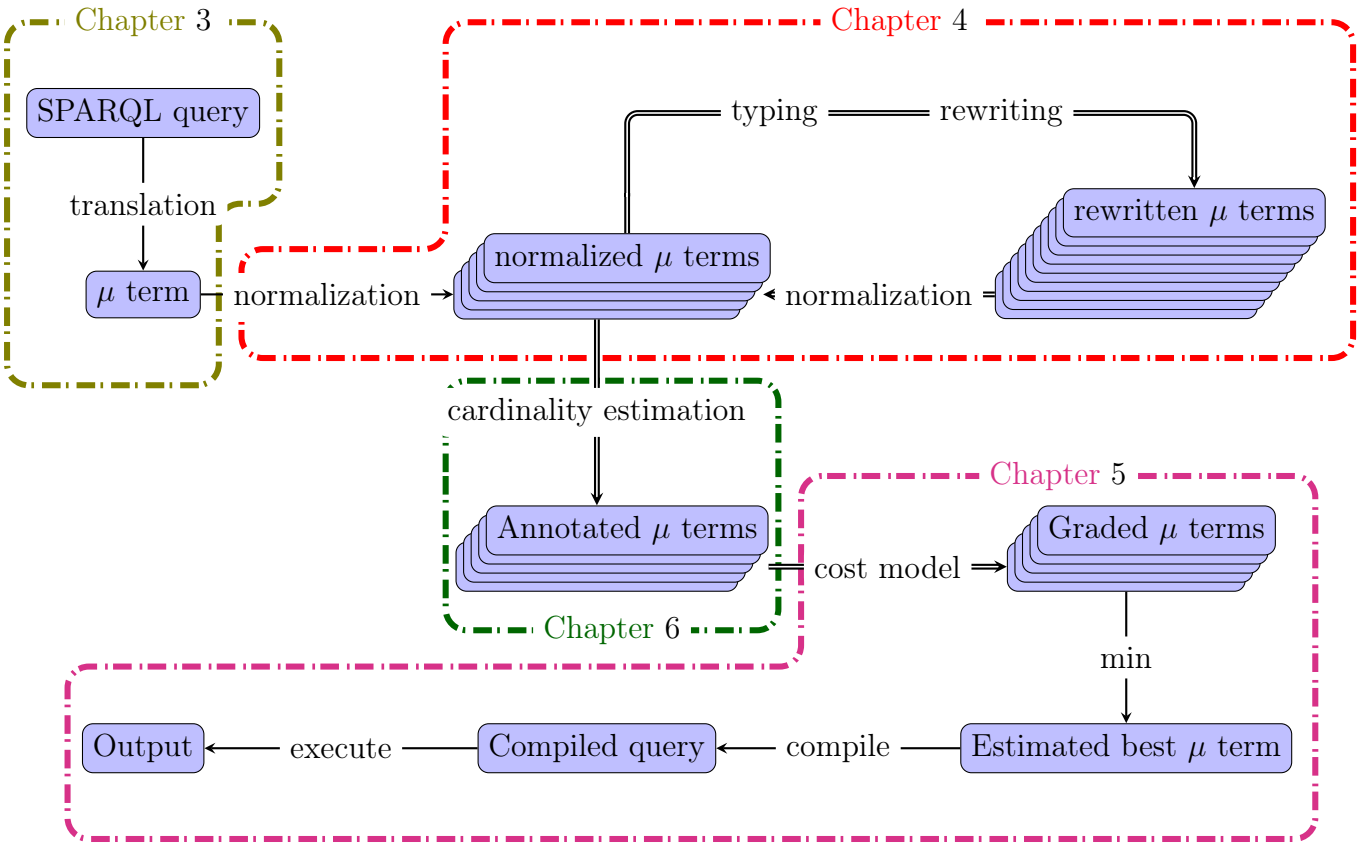


Figure 2.2: Schematic representation of our approach and the chapter decomposition

Notation for this part of the thesis

We use the following notations:

- an uppercase Γ corresponds to a abstract environment;
- μ-algebra terms are designated by greek letters $\varphi, \psi, \xi, \kappa$;
- the letter f usually stands for a **f**ilter condition;
- the letter g usually stands for a user defined function;
- a lowercase letter c designates a column;
- uppercase letters A, B, C, D and E correspond to sets of columns;
- uppercase letters U, S and W correspond to sets of mappings;
- uppercase letters such as X and Y correspond to μ-algebra variables;
- the uppercase letter V corresponds to an environment (a mapping from μ-algebra variables to sets of mappings);
- lowercase letters such as m, w correspond to mappings;
- the notation $\varphi[X/\psi]$ indicates that the μ-algebra term φ where we replace syntactical occurrences of X with ψ
- the notation $A[B \to C]$ indicates the set A where B is replaced by C if $B \subseteq A$, i.e. $A \setminus B \cup C$ when $B \subseteq A$ and A otherwise;
- the notation $A[B \leftrightarrow C]$ indicates the set A where B and C and simultaneously replaced i.e. it represents the $\big(A \setminus (C \cup B)\big) \cup C' \cup B'$ where $C' = \begin{cases} C & \text{when } B \subseteq A \\ \emptyset & \text{otherwise} \end{cases}$ and $B' = \begin{cases} B & \text{when } C \subseteq A \\ \emptyset & \text{otherwise} \end{cases}$.

CHAPTER 3

The μ-algebra

As shown in the previous chapters and as we will demonstrate in chapter 5, the optimization of recursive queries is still in is infancy. In order to tackle the optimization of SPARQL *queries and in particular of* SPARQL *queries containing recursion, we introduce a new algebra μ-algebra that we present in this chapter.*

The μ-algebra is heavily inspired by the SPARQL *algebra and the relational algebra and borrows features from both. It can be seen as a variant of the relational algebra extended with let-binders and a new fixpoint operators (which we denote with the letter μ, hence the name μ-algebra) using several traits specific to the* SPARQL *Algebra: the filters, the user-defined functions and the fact that our terms do not describe relations but rather sets of mappings where all mappings do not necessarily share the same domain.*

As we will see along the next few chapters, our μ-algebra can be seen as a Query Execution Plan (QEP) and allows us to rewrite queries and consider plans that

the existing methods did not and could not represent. We will demonstrate in the next chapters that this increased plan space is useful and that will validate our approach but we first need to describe the μ-algebra and how it captures SPARQL.

This chapter starts by presenting the general syntax and semantics of our μ-algebra. We then present restrictions on terms we will actually consider. Using these restrictions we then present a few properties that restricted μ-algebra terms enjoyed that will lay the foundations for the proof of our rewriting scheme.

We will investigate the relationship between our language and various relational languages such as Linear Datalog and inflationary Datalog$\neg$.

The μ-algebra was conceived with the ambition to capture a large fragment of SPARQL. *In the last section of this chapter, we will therefore explore this relationship. We will however present the μ-algebra in a generic way as it can capture large fragments of various other query languages on which the new optimizations that we provide for* SPARQL *could apply.*

3.1 Syntax & Semantics

In this section, we present the formalism of our language. As we will see, the syntax and semantics draw from the relational algebra, however our language is equipped with "missing values" à la SPARQL, and the semantics therefore uses mappings instead of relations.

3.1.1 Mappings

The mappings we manipulate here are very close to the Pattern Instance Mappings of the SPARQL standard. In fact, one can see these mappings as SPARQL mappings where we changed the domains; SPARQL mapping are from SPARQL variables or blank nodes to RDF terms, our mappings are functions with a finite domain and a signature $\mathcal{S} \to \mathcal{S}$ where $\mathcal{S}$ stands for a countable set. For the sake of simplicity we will suppose it is the set of strings (since other countable types can easily be serialized into strings we don't really need any other types except for further optimization).

Mappings are the building blocks of our μ-algebra. The μ-algebra terms that we define return sets of mappings and their semantics operate on mappings. As we will see in the SPARQL to μ-algebra translation, mappings are solutions (or partial solutions) of queries: the set of solutions is composed of mappings from the variables appearing in the query towards nodes in the graphs. But as SPARQL solutions can also comprise more than nodes from the queried graph (such as aggregates) our mappings can contain arbitrary values that we suppose encoded into strings.

Definition 11. Mappings *are functions with a finite domain. We often represent a mapping m as a set of the form $\{c_1 \to v_1, \ldots, c_n \to v_n\}$ for an element c, $m(c)$ is defined only if there exists i such that $c_i = c$ and $m(c) = v_i$. Such a i is necessarily unique (m is a function).*

Definition 12. *The* domain *of a mapping m is noted $dom(m)$ and corresponds to the set of c such that $m(c)$ is defined.*

Definition 13. *An element c of the domain $dom(m)$ of a mapping m is called a* column.

3.1.2 Syntax

The syntax of the language is heavily inspired by the syntax of the relational algebra. As we will see in the semantics section, there are key differences, notably to ensure the precise translation of SPARQL. The general syntax is presented in Figure 3.1 but we add another constraint: fixpoints need to be increasing. This requirement is a semantical property (see below in semantics) but we enforce it syntactically by forbidding recursive variables to appear in the "negative" operators: left-joins and minuses (normal, strict and set minuses).

3.1.3 Semantics

The μ-algebra language operates on mappings. When evaluated, a μ-algebra term returns a set (possibly empty) of mappings. The evaluation of a μ-algebra term also depends on the variables used in this term that is why we evaluate terms under an environment.

φ	::=		formula
	|	$\varphi_1 \cup \varphi_2$	union
	|	$\varphi_1 \backslash\backslash \varphi_2$	normal minus
	|	$\varphi_1 - \varphi_2$	set minus
	|	$\varphi_1 \setminus \varphi_2$	strict minus
	|	$\varphi_1 \ ⋊ \ \varphi_2$	left-join
	|	$\varphi_1 \ \bowtie \ \varphi_2$	join
	|	$\rho_a^b(\varphi)$	column exchange (or rename)
	|	$\pi_a(\varphi)$	column dropping
	|	$\beta_a^b(\varphi)$	column multiplying
	|	$\theta(\varphi, g : \mathcal{C} \to \mathcal{D})$	apply a function to mappings
	|	$\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$	reduce
	|	$\sigma_{filter}(\varphi)$	row filtering
	|	$\mu(X = \varphi)$	fixpoint
	|	let $(X = \varphi)$ in ψ	let-binder
	|	X	variable
	|	$\emptyset$	no mapping
	|	$\lvert c_1 \to v_1, \ldots, c_n \to v_n \rvert$	a mapping
$filter$	::=		filter expression
	|	$filter \wedge filter$	conjunction
	|	$filter \vee filter$	disjunction
	|	$\neg filter$	negation
	|	$bnd(c)$	presence
	|	$test(c_1, \ldots, c_n)$	test on values

Figure 3.1: Grammar of μ-algebra

Definition 14. *An* environment *is a mapping from μ-algebra variables to sets of mappings. Given an environment V, $V[X/S]$ corresponds to the environment V where the image of X is S. Environment are complete functions: they associate to each variable a set set of mappings. But the number of variables not pointing to the empty set of mappings (noted $\emptyset$) is always finite.*

The semantics of a μ-algebra term φ under the environment V is noted $[\![\varphi]\!]_V$. The definition of $[\![\varphi]\!]_V$ is recursive and given in figure 3.2. We give here the intuition behind the μ-algebra. Basic μ-algebra terms are variables and terms can be combined or modified via the following operators:

$\varphi_1 \cup \varphi_2$ Union on the sets of mappings from φ_1 and φ_2;

$\varphi_1 \bowtie \varphi_2$ Join operator: the combinations of pairs of mappings from φ_1 and φ_2 that are compatible;

$\varphi_1 ⋊ \varphi_2$ Left-join: the combinations of pairs of compatible mappings from φ_1 and φ_2 plus the mappings of φ_1 that are not compatible with any mapping from φ_2;

$\varphi_1 \setminus \varphi_2$ Strict Minus: the mappings from φ_1 for which there is no compatible mapping in φ_2;

$\varphi_1 \backslash\backslash \varphi_2$ Minus: the mappings from φ_1 for which all compatible mappings in φ_2 have a disjoint domain (note that two mappings with two disjoint domains are always compatible);

$\varphi_1 - \varphi_2$ Set Minus: the set difference on the set of mappings from φ_1 with the set of mappings of φ_2 (this is different from previous minuses: two compatible mappings are equal if and only if they have the same domain). The set of solutions of $\varphi_1 \setminus \varphi_2$ is always contained in the set of solutions of $\varphi_1 \backslash\backslash \varphi_2$ which is itself always contained in the set of solutions of $\varphi_1 - \varphi_2$ (except for the special case of the empty mapping $\{\}$, if it is solution of both φ_1 and φ_2 then it is solution of $\varphi_1 \backslash\backslash \varphi_2$ but not of $\varphi_1 - \varphi_2$);

$\rho_a^b(\varphi)$ Exchanges the value associated with a and b in the mappings solution of φ; when a (resp. b) is not bound then this operator just renames b into a (resp. a into b);

$\pi_a(\varphi)$ Removes the column a from the mappings of φ;

$\beta_a^b(\varphi)$ The effect of this operator is that in all mappings of φ, b is mapped to the same value as a (or no value if a is not in the mapping).

$\theta(\varphi, g : \mathcal{C} \to \mathcal{D})$ Maps the mappings m solution of φ through the function g. The notation $\theta(\varphi, g : \mathcal{C} \to \mathcal{D})$ indicates that when g is given a mapping m with $dom(m) \supseteq \mathcal{C}$ it returns the mapping $m_{|\bar{\mathcal{D}}} + g(m_{|\mathcal{C}})$ (where $m_{|\bar{\mathcal{D}}}$ denotes the mapping m where we remove the $\mathcal{D}$ part and $m_{|\mathcal{C}}$ denotes the mapping m limited to the domain $\mathcal{C}$). When $\mathcal{C} \not\subseteq dom(m)$, our operator leaves the mapping unchanged.

$\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$ Corresponds to a reduce operation using the user-defined function g and where the key is the complement of $\mathcal{C}$; i.e. two (or more) mappings m_1, m_2 are mapped to the same value when $dom(m_1) \setminus \mathcal{C} = dom(m_2) \setminus \mathcal{C}$ and $\forall c \in dom(m_1) \setminus \mathcal{C} : m_1(c) = m_2(c)$; $\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$ can be seen as the reduce or combine where $\theta(\varphi, g : \mathcal{C} \to \mathcal{D})$ is the map in a map-Reduce setup; for all M, we have $g(M) = \mathcal{D}$. The mappings m such that $\mathcal{C} \not\subseteq dom(m)$ are discarded.

$let\ X\ = \varphi\ in\ \psi$ Binds the variable X with the set of mappings of φ then returns the set of mappings of ψ. This operator does not add expressiveness (up to a renaming of variables we can expand $let\ X = \varphi\ in\ \psi$ into $\psi[X/\varphi]$) but it allows us to sometimes limit the combinatory explosion that might arise by expanding let-binding;

$\sigma_f(\varphi)$ Keeps only the mappings m such that $f(m)$ evaluates to true (i.e. $eval(f, m) = \top$);

$\mu(X = \varphi)$ Corresponds to the fixpoint of the function $S \to S \cup [\![\psi]\!]_{V[X/S]}$ (where $[\![\psi]\!]_{V[X/S]}$ is the semantic of ψ where the variable X is mapped to the set of mappings S). We suppose in $\mu(X = \varphi)$ that φ is increasing in X (i.e. $A \subseteq B \Rightarrow [\![\varphi]\!]_{V[X \to A]} \subseteq [\![\varphi]\!]_{V[X \to B]}$); that can be forced as a syntatic criterion (see section 3.3).

$|c_1 \to v_1, \ldots, c_n \to v_n|$ Corresponds to a single constant mapping.

$\emptyset$ Corresponds to the empty set of mappings.

X Corresponds to a variable, the solution is given by the environment $V(X)$.

$$
\begin{array}{rcl}
[\![\varphi_1 \bowtie \varphi_2]\!]_V &=& \{a+b \mid a \in [\![\varphi_1]\!]_V \wedge\ b \in [\![\varphi_2]\!]_V \wedge\ a \sim b\} \\
[\![\varphi_1 \backslash\!\backslash\, \varphi_2]\!]_V &=& \{m \in [\![\varphi_1]\!]_V \mid \nexists n \in [\![\varphi_2]\!]_V\ n \sim m \wedge dom(n) \cap dom(m) \neq \emptyset\} \\
[\![\varphi_1 \setminus \varphi_2]\!]_V &=& \{m \in [\![\varphi_1]\!]_V \mid \nexists n \in [\![\varphi_2]\!]_V\ n \sim m\} \\
[\![\varphi_1 - \varphi_2]\!]_V &=& \{m \in [\![\varphi_1]\!]_V \mid m \notin [\![\varphi_2]\!]_V\} \\
[\![\varphi_1 ⟕ \varphi_2]\!]_V &=& [\![\varphi_1 \bowtie \varphi_2]\!]_V \cup [\![\varphi_1 \setminus \varphi_2]\!]_V \\
[\![\pi_a(\varphi)]\!]_V &=& \{\{c \rightarrow v \mid (c \rightarrow v) \in m \wedge c \neq a\} \mid m \in [\![\varphi]\!]_V\} \\
[\![\beta_a^b(\varphi)]\!]_V &=& \{\{c \rightarrow v \mid (c \neq b \wedge c \rightarrow v \in m) \vee (c = b \wedge a \rightarrow v \in m)\} \mid m \in [\![\varphi]\!]_V\} \\
[\![\rho_a^b(\varphi)]\!]_V &=& \{\{c \rightarrow v \mid (c = a \wedge b \rightarrow v \in m) \vee (c = b \wedge a \rightarrow v \in m) \vee (c \neq a \wedge c \neq b \wedge c \rightarrow v \in m)\} \\
&& \mid m \in [\![\varphi]\!]_V\} \\
[\![\sigma_f(\varphi)]\!]_V &=& \{m \mid m \in [\![\varphi]\!]_V\ \wedge eval(f,m) = \top\} \\
[\![\Theta(\varphi, g, \mathcal{C}, \mathcal{D})]\!]_V &=& \{m + g(M) \mid \mathcal{C} \cap dom(m) = \emptyset \\
&& \text{and } M = \{m' \mid m + m' \in [\![\varphi]\!]_V \text{ and } dom(m') = \mathcal{C}\} \text{ and } M \neq \emptyset\} \\
[\![\theta(\varphi, g : \mathcal{C} \rightarrow \mathcal{D})]\!]_V &=& \{\{c \rightarrow v \in m \mid c \notin \mathcal{D}\} + f(\{c \rightarrow v \in m \mid c \in \mathcal{C}\}) \mid \mathcal{C} \subseteq dom(m)\} \cup \{m \mid \mathcal{C} \nsubseteq dom(m)\}
\end{array}
$$

$$
\begin{array}{rclrcl}
[\![|c_1 \rightarrow v_1, \ldots, c_n \rightarrow v_n|]\!]_V &=& \{\{c_1 \rightarrow v_1, \ldots, c_n \rightarrow v_n\}\} & [\![\emptyset]\!]_V &=& \emptyset \\
[\![\text{let } (X = \varphi) \text{ in } \psi]\!]_V &=& [\![\psi]\!]_{V[X \rightarrow [\![\varphi]\!]_V]} & [\![X]\!]_V &=& V(X) \\
[\![\mu(X = \varphi)]\!]_V &=& [\![X]\!]_{V[X \rightarrow U]} & [\![\varphi_1 \cup \varphi_2]\!]_V &=& [\![\varphi_1]\!]_V \cup [\![\varphi_2]\!]_V
\end{array}
$$

Where $U_0 = \emptyset$, $U_{i+1} = U_i \cup [\![\varphi]\!]_{V[X \rightarrow U_i]}$ and $U = lim_{n \rightarrow \infty} U_i$.

Figure 3.2: Semantics of μ-algebra

3.1.4 Semantics of filters

Our μ-algebra language is equipped with a filter operator $\sigma_f(\varphi)$ where φ is a μ-algebra term and f a filter expression. This $\sigma_f(\varphi)$ only keeps the mappings m of φ such that $f(m)$ evaluates to true (true is noted $\top$).

Our filter expressions correspond to the SPARQL filter expressions, which have a rich syntax. The syntax of filter expressions in μ-algebra is presented in figure 3.1, but we omit the detailled presentation of *test* functions. In the filter syntax, a test function is just any boolean function operating on values but if any of the values considered by the test function is missing, the function will evaluate to a special error symbol. Filter functions are thus three-valued boolean functions ($\top$ for true, $\bot$ for false and *Error* for error).

The exact set of test functions is not relevant to this manuscript as we treat all test functions alike. We simply suppose that we have all test functions needed to translate SPARQL filters and that they include SPARQL filters which means we have the typed comparisons such as $?price < 42$ or $?price_1 = ?price_2$ to keep the mappings m such that $m(?price) < 42$ when treated as integer or $m(?price_1) = m(?price_2)$), the regular expressions (such as $regex(?title, "Web")$ to keep mappings m such that $m(?title)$ contains the string Web).

As important kinds of filters, we also have: the test of column presence (written $bnd(c)$), which evaluates to true on mappings whose domain contains c; the various classical logical composition of filters: conjunction (with $\wedge$), disjunction (with $\vee$), and negation (with $\neg$). And we suppose that other tests are encoded as n-ary functions that we will treat as black boxes. For instance our filters include comparisons between values and columns ($less(?a, b)$ to compare the value b and the value bounded by $?a$ and $lessthant42(?a)$ to compare the column $?a$ with 42), etc. The formal semantics of filters is given in figure 3.3.

$$
\begin{aligned}
eval(f_1 \wedge f_2, m) &= \begin{cases} \top \text{ when } (eval(f_1, m) = \top) \wedge (eval(f_2, m) = \top) \\ \bot \text{ when } (eval(f_1, m) = \bot) \vee (eval(f_2, m) = \bot) \\ Error \text{ otherwise} \end{cases} \\
eval(f_1 \vee f_2, m) &= \begin{cases} \top \text{ when } (eval(f_1, m) = \top) \vee (eval(f_2, m) = \top) \\ \bot \text{ when } (eval(f_1, m) = \bot) \wedge (eval(f_2, m) = \bot) \\ Error \text{ otherwise} \end{cases} \\
eval(\neg f, m) &= \begin{cases} \top \text{ when } eval(f, m) = \bot \\ \bot \text{ when } eval(f, m) = \top \\ Error \text{ otherwise} \end{cases} \\
eval(bnd(c), m) &= \begin{cases} \top \text{ when } c \in dom(m) \\ \bot \text{ when } c \notin dom(m) \end{cases} \\
eval(test(c_1, \ldots, c_n), m) &= \begin{cases} Error \text{ when } \{c_1, \ldots, c_n\} \not\subseteq dom(m) \\ \top \text{ when } test(m(c_1), \ldots, m(c_n)) = \top \\ \bot \text{ otherwise} \end{cases}
\end{aligned}
$$

Figure 3.3: Semantics of filters in μ-algebra

3.1.5 Free columns of a filter

Definition 15. *The set $FC(f)$ of* free columns *of a filter f is the set of columns that the filter depends on. Given a column $c \notin FC(f)$ then for all mappings m with $c \notin dom(m)$ and all values v we have $eval(f)(m) = eval(f)(m + \{c \to v\})$. We define $FC(f)$ as the following syntactic criterion:*

$$
\begin{aligned}
FC(f_1 \wedge f_2) &= FC(f_1) \cup FC(f_2) \\
FC(f_1 \vee f_2) &= FC(f_1) \cup FC(f_2) \\
FC(\neg f) &= FC(f) \\
FC(bnd(c)) &= \{c\} \\
FC(test(c_1, \ldots, c_n)) &= \{c_1, \ldots, c_n\}
\end{aligned}
$$

Proposition 1. *For all pairs of mappings (m_1, m_2) and all filters f such that $dom(m_1) \cap FC(f) = dom(m_2) \cap FC(f)$ and $\forall c \in dom(m_1) \cap FC(f)$ $m_1(c) = m_2(c)$ then $eval(f)(m_1) = eval(f)(m_2)$ and thus the definition of FC works as expected.*

Proof Sketch. See full proof in appendix A.1.

It follows the definition of both FC and $eval$ by induction. □

3.2 Examples of μ-algebra terms

As the formal definition of the language has been presented in the last section, let us present in this section some examples of terms and interpretation of these terms on actual datasets to build some intuition about our new language.

3.2.1 Example datasets

Let us present a few examples of how these operators work and build some intuition. We consider the two following sets of mappings (represented in tabular view, where an empty case represents a column not bound by the mapping):

Set A with 4 mappings

name	pseudo	phone
John	Johnny	
Luke	Lucky	3
	Ed	3
Camille		7

Set B with 5 mappings

name	phone
John	1
Bob	
Luke	3
	4
Alice	5

Figure 3.4: Example datasets.

These datasets can also be represented as μ-algebra terms using the constants and union. For instance the two first items of B can be represented as $|name \to John, phone \to 1| \cup |name \to Bob|$.

3.2.2 Unary operators

Here are examples of unary operators operating on the sets described above.

Results of $\sigma_{phone=3}(A)$

name	pseudo	phone
Luke	Lucky	3
	Ed	3

Results of $\pi_{pseudo}(A)$

name	phone
John	
Luke	3
	3
Camille	7

Results of $\rho_{phone}^{number}(A)$

name	pseudo	number
John	Johnny	
Luke	Lucky	3
	Ed	3
Camille		7

Results of $\beta_{name}^{name_2}(A)$

name	$name_2$	pseudo	phone
John	John	Johnny	
Luke	Luke	Lucky	3
		Ed	3
Camille	Camille		7

Figure 3.5: Results for unary operators

And now let us present an example of an aggregation. We have a listing where each mapping corresponds to a player with an indication of its name and its team. The query corresponds to counting the number of different names per team. Notice that the mapping $\{team \to D\}$ is ignored because it does not have a name. If we wanted to actually count the number of players per team we should have used C' defined as $C \rtimes \{name \to undef\}$ to make sure that all fields used in the aggregation were defined.

Set C

name	team
Alice	A
Bob	A
Charlie	B
Daisy	C
Eve	
	D

Results of $\Theta\left(C, cnt = count(name), \{name\}, \{cnt\}\right)$

team	cnt
A	2
B	1
C	1
	1

Figure 3.6: Example of the aggregation operator.

Finally let us present an example of a fixpoint. Let us suppose that E corresponds to a directed graph with mapping binding $from$ and to and let us look to the transitive closure of E.

Graph E in table form

from	to
A	B
B	C
A	D
D	E
F	G

Results of $\mu\big(X = E \cup \pi_m\left(\rho^m_{to}\left(E\right) \bowtie \rho^m_{from}\left(X\right)\right)\big)$

from	to
A	B
A	C
A	D
A	E
B	C
D	E
F	G

Figure 3.7: Example of a fixpoint.

3.2.3 Binary operators: joins and minuses

The results of Joins and Minuses for A and B are shown in figure 3.8. The difference between $\bowtie$ and $\rtimes$ is the inclusion of the mapping $\{name \to Camille, phone \to 7\}$ of A which does not match with any mapping of B. In this example $A \backslash\backslash B$ and $A \setminus B$ have the same semantics because all pairs of mappings from A and B share a column thus we also considered the set $\pi_{phone}\left(A\right)$ instead of A.

Results of $A \bowtie B$

name	pseudo	phone
John	Johnny	1
John	Johnny	4
Luke	Lucky	3
Luke	Ed	3
Bob	Ed	3

Results of $A \ltimes B$

name	pseudo	phone
John	Johnny	1
John	Johnny	4
Luke	Lucky	3
Luke	Ed	3
Bob	Ed	3
Camille		7

Results of $A - B$

name	pseudo	phone
John	Johnny	
Luke	Lucky	3
	Ed	3
Camille		7

Results of $A \backslash\backslash B$ or $A \setminus B$

name	pseudo	phone
Camille		7

Results of $\pi_{phone}(A) \setminus B$

name	pseudo

Results of $\pi_{phone}(A) \backslash\backslash B$

name	pseudo
Camille	
	Ed

Figure 3.8: Results for joins and minuses between A and B

3.3 Restriction on μ-algebra: constant and linear recursions

In this section we define what *constant* in the variable X means and similarly to linear datalog (in a sense that will explained further in section 3.4), we introduce a distinction between *linear* and *non linear* recursions. A non linear recursion corresponds to a recursion where several mappings are simultaneously used to produce a new one while in linear recursions mapping depends on only one mapping. The distinction is then used to prove that our fixpoint operator is well-founded in the case of linear recursions.

3.3.1 Definitions

A *recursive* variable is a variable introduced by a fixpoint operator.

Definition 16. *The function $sim(\varphi, X)$ computes the degree of linearity of a term φ in the variable X. φ is said* constant *in X when $sim(\varphi, X) = 0$,* linear *in X when $sim(\varphi, X) \leq 1$. With this function we will limit the possible μ-algebra terms to a set of terms that behave well.*

$$
\begin{array}{rcll}
sim(\varphi_1 \backslash\backslash \varphi_2, X) & = & sim(\varphi_1, X) + 2 * sim(\varphi_2, X) & \\
sim(\varphi_1 \setminus \varphi_2, X) & = & sim(\varphi_1, X) + 2 * sim(\varphi_2, X) & \\
sim(\varphi_1 - \varphi_2, X) & = & sim(\varphi_1, X) + 2 * sim(\varphi_2, X) & \\
sim(\varphi_1 \bowtie \varphi_2, X) & = & sim(\varphi_1, X) + 2 * sim(\varphi_2, X) & \\
 & & & \\
sim(\varphi_1 \cup \varphi_2, X) & = & max(sim(\varphi_1, X), sim(\varphi_2, X)) & \\
sim(\varphi_1 \bowtie \varphi_2, X) & = & sim(\varphi_1, X) + sim(\varphi_2, X) & \\
 & & & \\
sim(\rho_a^b(\varphi), X) & = & sim(\varphi, X) & \\
sim(\pi_a(\varphi), X) & = & sim(\varphi, X) & \\
sim(\beta_a^b(\varphi), X) & = & sim(\varphi, X) & \\
sim(\sigma_f(\varphi), X) & = & sim(\varphi, X) & \\
sim(\theta(\varphi, g : \mathcal{C} \to \mathcal{D}), X) & = & sim(\varphi, X) & \\
sim(\Theta(\varphi, g, \mathcal{C}, \mathcal{D}), X) & = & sim(\varphi, X) \times 2 & \\
sim(\mu(Y = \varphi), X) & = & 0 & when\ sim(\varphi, X) = 0 \vee X = Y \\
sim(\mu(Y = \varphi), X) & = & 2 & when\ sim(\varphi, X) > 0 \wedge X \neq Y \\
sim(let\ (Y = \varphi)\ in\ \psi, X) & = & sim(\psi, Y) \times sim(\varphi, X) + sim(\psi, X) & when\ X \neq Y \\
sim(let\ (Y = \varphi)\ in\ \psi, Y) & = & 0 & \\
sim(X, X) & = & 1 & \\
sim(Y, X) & = & 0 & when\ X \neq Y \\
sim(\emptyset, X) & = & 0 & \\
sim(|c_1 \to v_1, \ldots, c_n \to v_n|, X) & = & 0 &
\end{array}
$$

Definition 17. *Let R be the set of recursive variables appearing in φ, φ is* valid *if there are no non-linear subterm, i.e. for all subterms φ we have $\forall r \in R \quad sim(\varphi, r) \leq 1$;*

Remark 1. *Unless explicitly stated otherwise, all the terms we consider in this thesis are valid in the sense of definition 17.*

3.3.2 Properties of the linear μ-algebra

Lemma 1 shows that constant terms are truly constant. This lemma we will very useful in the proofs of upcoming lemmas and theorem.

Lemma 1. *Given a term φ valid in the sense of definition 17 and such that $sim(\varphi, X) = 0$ then $[\![\varphi]\!]_V$ does not depend on $V(X)$, i.e. $\forall S \quad [\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\emptyset]}$.*

Proof Sketch. See full proof in appendix A.1.1.

The idea is simple: when $sim(\varphi, X) = 0$, we generally have that $sim(\psi, X) = 0$ for all subformulas ψ of φ. In particular it is true for base terms, which means X does not belong to the set of subformulas.

The only case where $sim(\psi, X) \neq 0$ is when ψ appears as a subformula of κ in let $(Y = \kappa)$ in ξ and that Y does not appear in ξ. In which case, the let $(Y = \kappa)$ in ξ could actually be simplified in ξ. □

We will now present a lemma and theorem that show that linear μ-algebra (or simply μ-algebra in the rest of this thesis) enjoy useful properties for fixpoints.

Lemma 2. *Given a valid term φ we have that $A \subseteq B \Rightarrow [\![\varphi]\!]_{V[X/A]} \subseteq [\![\varphi]\!]_{V[X/B]}$.*

Proof Sketch. See full proof in appendix A.1.2.

The idea is simple. *sim* is designed to prevent X from appearing negatively — i.e. as the subformula in the right side of a minus — by using the value 2. Therefore all subterms are monotone in X and in minuses the left side are constant in X. □

Theorem 1. *Given a fixpoint expression $\mu(X = \varphi)$ that is valid in the sense of definition 17 and an environment V, the function $f(S) = [\![\varphi]\!]_{V[X \to S]}$ has the following properties:*

1. $\forall W \neq \emptyset \quad f(W) = \bigcup_{w \in W} f(\{w\})$
2. $\forall A, B, C \quad A = B \cup C \Rightarrow f(A) = f(B) \cup f(C)$
3. *f has a least fixpoint P with $P = [\![\mu(X = \varphi)]\!]_V$*

Proof Sketch. See full proof in appendix A.1.3.

Point 2 and 3 follow easily from point 1. The idea of the proof of point 1. is that *sim* is designed such that when we combine two terms (e.g. $\varphi_1 \bowtie \varphi_2$) then X can only appear in one of them.

Notice that while lemma 2 was resting only on the fact the X does not appear negatively in φ, this theorem needs simultaneously this positive appearance of X AND that the combination of mappings cannot appear from mapping coming out of X.

□

Lemma 2 and Theorem 1 show that fixpoints are least fixpoints and can be computed in various ways. If we have a μ-algebra term with a negative appearance of X such as $\mu(X = \varphi - X)$ then our semantics is defined (here it will be $[\![\mu(X = \varphi - X)]\!]_V = [\![\varphi]\!]_V$) but it does not correspond to a fixpoint semantics (e.g. $[\![\varphi - X]\!]_{V[X/[\![\varphi]\!]_V]} = \emptyset$).

If we have a non-linear μ-algebra term but where X appear positively such as $\mu(X = \varphi \cup \pi_c(\rho_a^c(X) \bowtie \rho_b^c(X)))$ then it is increasing, our semantics of this fixpoint is a fixpoint semantics but we do not have $[\![\mu(X = \varphi)]\!]_V = \bigcup_{x \in [\![\mu(X=\varphi)]\!]_V} [\![\varphi]\!]_{V[X/\{x\}]}$ in general.

3.4 Relationship between μ-algebra and existing relational variants

Our language μ-algebra is very powerful in terms of expressive power. It is easy to see that if we do not put limit on user-defined functions and that we want to compute a function f then we can concatenate all the mappings into one with an aggregation then apply the function f. However, we can also show that, if we allow the map operator and only one simple function ($x \to x + 1$), then our μ-algebra has, at least, the expressive power of a Turing machine (see appendix A.2.1).

Since we have the negation, we can even show that our language is capable of solving the halting problem, which means that our μ-algebra has more expressive power than a Turing Machine. Obviously such programs are not very interesting for a practical application as a computer would never be able to answer.

If we look carefully at the proof, this more-than-Turing-complete happens because the map operator is combined with a fixpoint and that can create an infinite amount of distinct

values. If we remove the map and aggregation operators from our language, we can easily see that values in mappings are restricted to values that appear either in the term or in the input and since all fixpoints are monotonic our programs are in PTIME.

More precisely, in appendix A.2.4 we show that without the linearity, map and aggregation, we have the expressive power of inflationary Datalog$\neg$ but with the addition of the linearity, we have the expressive power of linear Datalog, which strictly less than the expressive power of Datalog. For instance, with linearity but without map and aggregation, we cannot find pair of nodes in a graph that are linked by a path labeled by two constants `:A` and `:B` with as many of each.

3.5 Translation from SPARQL

One of the main reason for the inception of the μ-algebra was to study to optimization of SPARQL. In this section, we present a translation from SPARQL towards our μ-algebra. This translation will allow us to put the focus in the next chapters on the evaluation and optimization of μ-algebra terms as we will know how to translate SPARQL queries into μ-algebra terms.

3.5.1 SPARQL variables

We recall that the SPARQL standard that defines queries has a notion of *variable* and each solution of a SPARQL query is a mapping from a subset of these SPARQL *variables* appearing in the query towards strings that represent IRI, blank nodes or literals. In our encoding, the solutions will be mappings whose domains are SPARQL *variables* and whose images are IRI, blank nodes or literals.

It is therefore important to not get confused in the terminology between μ-algebra variables that are evaluated to sets of mappings and are introduced by a let $(X = \dots)$ in $\dots$ or a $\mu(X = \dots)$ and SPARQL variables that will be translated to column names and will be part of the domains of mappings.

Since they do not represent the same kind object, for the sake of clarity, SPARQL variables will be called SPARQL variables whereas μ-algebra variables will be simply be called variables.

3.5.2 Encoding of the graph structure

We recall that an RDF graph G is given by a set of triples, $G \subset U \times U \times U$. When $(s, p, o) \in G$ we say that the node s is linked to the node o via an edge labeled by p. An RDF dataset is a set of RDF graphs $G_{i_1}\ dots G_{i_n}$ each identified by an IRI. As such an RDF dataset can be seen as a set of quadruples: one (s, p, o, g) for each $(s, p, o) \in G_g$.

The encoding of all graphs is done using a unique μ-algebra variable Q that contains one mapping per quadruple, each of these mappings has for domain: 's' for subject, 'p' for predicate, 'o' for object and 'g' for graph.

We then introduce a term T_n for each graph named n and a variable T referring to the default graph. All of them bind three columns: 's', 'p', 'o' (for subject, predicate and object).

Finally we have a variable N_n for the set of all nodes of each graph named n plus a variable N referring to the default graph. Each node is represented by a mapping whose domain is $\{s\}$ (even though N also contains nodes that only appear as objects).

Note that we can derive all of these variables from Q: for each graph name *name*, we use $let\ T_{name} = \pi_g(\sigma_{g=name}(Q))\ in\ldots$ and as SPARQL allows us to change the graph queried, during the course of the translation we maintain T to be the set of triples (s, p, o) corresponding to the current graph using let binders and finally, $N_n = \rho_s^o(\pi_p(\pi_s(T_n))) \cup \pi_p(\pi_o(T_n))$.

3.5.3 Regular Path Expressions

$$
\begin{array}{rcl}
rpe(Constant(u)) & = & \pi_p(\sigma_{p=''u''}(T)) \\
rpe(s-Variable(v)) & = & \rho_p^v(T) \\
rpe((r_1/r_2)) & = & \pi_m(\rho_o^m(rpe(r_1)) \bowtie \rho_m^m(rpe(r_2))) \\
rpe((r_1|r_2)) & = & rpe(r_1) \cup rpe(r_2) \\
rpe((r^{-1})) & = & \rho_s^o(rpe(r)) \\
rpe((!\{:i_1\cdots:i_n\})) & = & \pi_p\left(\sigma_{p\notin\{:i_1\cdots:i_n\}}(T)\right) \\
rpe((r?)) & = & rpe(r) \cup \beta_s^o(N) \\
rpe((r*)) & = & \mu(X = \beta_s^o(N) \cup \pi_m(\rho_o^m(X) \bowtie \rho_s^m(rpe(r))))
\end{array}
$$

Figure 3.9: Translation of a PP.

The hard part of translating a PP to a μ-algebra term is the translation of the regular path expression. The basic idea is to recursively translate the regular path expression r to a μ-algebra term such that each mapping solution of this term maps s (resp. o) to s' (resp. o') with s' linked to o' via a path accepted by r. However, we included here SPARQL variables in regular path expressions and the translation might bind more column names than just s and o.

The translation of a regular path expression r is $rpe(r)$ as presented in figure 3.9. Our SPARQL expression differ from the usual SPARQL algebra because we provide a syntactic differentiation between constants and variables for the reader by introducing $Constant(u)$ instead of u when u is an IRI and $s-Variable(v)$ for a SPARQL variable. We also suppose that there are no blank nodes in the query as we have shown how to remove them in section 1.2.

Note that $rpe(r*)$ introduces a fixpoint with always the same variable X, but in practice we can use a fresh variable there. Furthermore note that the terms produced by our translation are indeed *valid* in the sense of definition 17.

3.5.4 Property Paths

We note $gp(a)$ our translation of the SPARQL algebra term a into μ-algebra. The translation of the SPARQL algebra term $PP(s, r_p, o)$ (where r_p is a regular path expression) into $gp(PP(s, r_p, o))$ is done in two steps: first, we obtain a μ-algebra term $rpe(r_p)$ representing the set of mappings solution of r_p; then, there are four cases depending on whether s is a value or a SPARQL variable and whether t is a value (w) or a SPARQL variable ($?w$). With $p = pp(r_p)$, $gp(PP(s, r_p, o))$ is equal to:

	$o = Constant(w)$	$o = s - Variable(?w)$
$s = Constant(v)$	$\pi_s(\pi_o(\sigma_{s=v}(\sigma_{o=w}(p))))$	$\pi_s\left(\sigma_{s=v}\left(\rho_o^{?w}(p)\right)\right)$
$s = s - Variable(?v)$	$\pi_o\left(\sigma_{o=w}\left(\rho_s^{?v}(p)\right)\right)$	$\rho_o^{?w}\left(\rho_s^{?v}(p)\right)$

In this translation, we implicitly supposed that if s or o are SPARQL variables then they are different from each other and they do not appear in r_p. In case $o = s - Variable(?v)$ and $?v$ appears somewhere else in the PP (in s or in r_p), we replace it with $?v'$: we translate the PP $(s, r_p, s - Variable(?v'))$ to a μ-algebra term φ and finally the translation is $\pi_{v'}(\sigma_{v=v'}(\varphi))$. Similarly, if $s = s - Variable(?w)$, and $?w$ appears in r_p, we replace it with $?w'$.

3.5.5 Translation of SPARQL algebra terms

SPARQL algebra terms are defined recursively and are very similar by design to our μ-algebra terms. A SPARQL algebra term can therefore be easily translated into our μ-algebra:

- we have seen the translation for property paths;
- join can be translated to our join;
- unconditional optionals can be translated to our left join and conditional optionals are translated using their definition as the union of a filtered join and a conditional difference;
- filters translate to our filter operator;
- unions translate to our union;
- graph selections translate to let bindings;
- the semantics of $FilterExists$ and $FilterNotExist$ are ill-defined but considering a reasonable semantics, it is possible [KKG17] to translate them to simpler SPARQL operators;
- $Minus$ to a minus ($\setminus$);
- the aggregations can be translated to an aggregation;
- the conditional difference is non-trivial to translate.

3.5.6 Conditional difference

Conditional difference ($Diff(\Omega_1, \Omega_2, cond)$ in the W3C literature) is defined as the set of mappings m_1 from Ω_1 such that all compatible mappings m_2 from Ω_2 are such that $cond(m_1 + m_2)$ is not true (false or error). Therefore $Diff(\Omega_1, \Omega_2, cond) = \Omega_1 - \Omega'$ where Ω' captures the mappings n from Ω_1 such that there is a mapping m from Ω_2 with $cond(n + m)$. In order to compute Ω', we need to keep the Ω_1 part of solutions of $\sigma_{cond}(\Omega_1 \bowtie \Omega_2)$. In many cases, this would only mean to remove the columns specific to Ω_2 but since, in some cases, the presence of some columns might be contingent in the mappings of Ω_1 and Ω_2 then we need to "backup" the columns specific to Ω_1 before joining.

Let $c_1, \ldots, c_n$ be the set of columns that can appear in in some mapping solution of Ω_1 and in some mappings solution of Ω_2 but not sure to be present in Ω_1. Let $t_1, \ldots, t_n$ a set of "fresh" column names, we "backup" the columns $(c_1, \ldots, c_n)$ of Ω_1 onto $(t_1, \ldots, t_n)$ with $\beta_{c_1}^{t_1}\left(\ldots \beta_{c_n}^{t_n}(\Omega_1)\right)$ then we join with Ω_2, we filter, and finally we restore the backup. To restore

```
SELECT *
{
 {  ?b firstname john . }
  UNION
 {  ?b firstname john . }
}
```

Figure 3.10: Example SPARQL query with a set semantics different from the bag semantics

the backup we need to remove all columns $j_1, \ldots, j_m$ that might appear in Ω_1 or Ω_2 except those used to backup then exchange the $(t_i)_i$ with the $(c_i)_i$. This gives us:

$$Diff(\Omega_1, \Omega_2, cond) = \text{let } (O = \Omega_1) \text{ in } O - \rho_{c_1}^{t_1}\left(\ldots \rho_{c_n}^{t_n}\left(\pi_{j_1}\left(\ldots \pi_{j_n}\left(\sigma_{cond}(\beta_{c_1}^{t_1}\left(\ldots \beta_{c_n}^{t_n}(O)\right) \bowtie \Omega_2)\right)\right)\right)\right)$$

3.5.7 Set semantics and bag semantics

The set semantics consists in producing an answer with the set of bindings that are solutions while the bag semantics consists in producing the same answers but with multiplicity. For instance, if $?a =' bob'$ is twice a solution, then it should appear twice as an answer in the bag semantics, and once in the set semantics.

SPARQL is evaluated, by default, with a bag semantics. For a query such as the one in figure 3.10 a SPARQL query evaluator should give a binding for $?b$ repeated twice for each $?b$ solution. If SPARQL uses the bag semantics by default, most SPARQL queries, or, at least, most part of most SPARQL queries can be evaluated under a set semantics for the following reasons:

- the set semantics and the bag semantics often agree, for instance on the semantics of Triple Patterns and Basic Graph Pattern;
- in SPARQL most Property Paths need to be evaluated under the set semantics as counting the number of paths matching a Regular Path Query would often be intractable;
- finally, the user can specify that a query that it has to be evaluated under the set semantics using the keyword `DISTINCT`. The semantics of `REDUCED` is that the evaluator can answer anything comprised between the set semantics and the bag semantics. Therefore it can be evaluated using the set semantics when using the keyword `REDUCED`.

On the opposite, the main causes of differences between the bag semantics and the set semantics are the following:

- the unions (as portrayed in figure 3.10)
- the joins or left-joins between terms comprising unions or left-joins
- the projection (i.e. when select explicitly lists only a subset of the SPARQL variables appearing in the query).

Our μ-algebra is equipped with a set semantics that follow the set semantics of SPARQL. Our μ-algebra can be adapted to deal with the bag semantics of SPARQL. The idea is to first translate as if we were evaluating terms under the set semantics but tag terms that need to be evaluated under the bag semantics. Then we change terms that need to evaluated under the bag semantics by adding either a Cardinality (Card) field or an Identifier (ID) field to solutions.

We will not detail here the general method but present a few examples. Let us suppose that A and B can be evaluated under the set semantics but that $A \cup B$ needs to be evaluated under the bag semantics. We first modify the solutions of A and B to add a cardinality information. Since both A and B can be evaluated under the set semantics, it means that each solution has to be counted once. Therefore we compute $A' = A \bowtie |Card \rightarrow 1, ID \rightarrow ID_A|$ and $B' = B \bowtie |Card \rightarrow 1, ID \rightarrow ID_B|$ then we compute the union $A' \cup B'$ and reduce the solutions: $\Theta\left(A' \cup B', \sum Card, \{ID, Card\}, \{Card\}\right)$.

If a term A has been computed with the method presented above then we can replace the term $\pi_c(A)$ with the following term $\Theta\left(A, Card = \sum Card, \{c, Card\}, \{Card\}\right)$

This method works well but induce a cost since we need to perform a reduce operation after each computation. Another method consists in adding a Unique ID (UID). For instance if we have a term A and a term B and we want to evaluate $A \bowtie B$ under the bag semantic we add a column UID_A to the mappings solutions of A (all mappings in A receive a different UID_A) and a column UID_B to the mappings of B then we perform the join normally, and, finally, we can replace the pair of columns (UID_A, UID_B) with a column $UID_{A \bowtie B}$.

This second method is lightweight per solution as mapping a set of solution through a function to tag them with an ID is very cheap (at least compared with the actual computation of a join) but in practice, it might sometimes be actually more expensive than the first method as in the first method, a mapping that is present n times is actually represented once while in this second method a mapping is present as many times as it is a solution. That is why, in practice, we rely on a mixed method, preferring the first method for joins and unions and the second for projections.

3.5.8 Final translation

The translation for a SPARQL–algebra is given in figure 3.12 for the translation in set semantics and in figure 3.11 for the bag semantics. For the sake of simplicity, we encoded expression of the form $Graph(v, t)$ as an union ranging v over the possible graph names which supposes a knowledge of the queried dataset at compile time. However this constraint can be overcame by maintaining the graph name in the case of variable graph names, the only difference being for minuses where we need to ensure that the a removed column share the same graph name. The $Minus$ can thus be expressed as a conditional $Diff$.

The function $multi(t)$ presented in figure 3.11 produces terms that are evaluated in bag semantics.

3.5.9 Example of translation

In the case of the simple recursive query presented in figure 3.13. This query asks for the pairs $?a$ and $?b$ such that there is a path where the edges of the path are labeled by *knows* from $?a$ to $?b$ and there is a path between $?b$ and the node `:John` labeled by `:Firstname`.

$$
\begin{aligned}
multi(Join(a,b)) &= multi(a) \bowtie multi(b) \\
multi(Union(a,b)) &= multi(a) \cup multi(b) \\
multi(Project(V,t)) &= \pi_{a_1,\ldots,a_n}(multi(t)) && \{a_1,\ldots,a_n\} \text{ variables to remove} \\
multi(BGP(p_1,\ldots,p_n)) &= \pi_{a_1,\ldots,a_n}(\theta(distinct(p_1) \bowtie \ldots \bowtie distinct(p_n), m \to |id_h \to fresh()| : \emptyset \to \{id_h\})) \\
& && \{b_1,\ldots,b_n\} \text{ blank nodes to remove} \\
multi(Filter(f,t)) &= \sigma_f(multi(t)) \\
multi(Diff(f,t_1,t_2)) &= Diff(f, multi(t_1), distinct(t_2)) \\
multi(Minus(t_1,t_2)) &= multi(t_1) \setminus distinct(t_2) \\
multi(Extend(t,a,v)) &= \beta_v^a(multi(t)) && \text{when } v \text{ is a column} \\
multi(Extend(t,a,v)) &= multi(t) \bowtie |a \to v| && \text{when } v \text{ is a constant} \\
multi(Extend(t,a,v)) &= \theta(multi(t), (m \to |a \to v(m)| : FC(v) \to \{a\}) \\
multi(Graph(i,t)) &= \text{let } (T = T_v) \text{ in } multi(t) && \text{for } v \text{ IRI} \\
multi(Graph(v,t)) &= \text{let } (T = T_{u_1}) \text{ in } multi(t) \bowtie |v \to u_1| \bowtie \ldots \text{let } (T = T_{u_k}) \text{ in } multi(t) \bowtie |v \to u_k| \\
& && \text{for } v \text{ variable}
\end{aligned}
$$

$$\text{with } G = Group(e'_1,\ldots,e'_j,t)$$

$$multi(AggregationJoin(Aggregation(e^1_1,\ldots,e^1_{i_1}, f_1, scalarvals, G),\ldots,Aggregation(e^l_1,\ldots,e^l_{i_l}, f_l, scalarvals, G))) =$$

$$\Theta\,(computeKeys, computeAgg, \{k_1,\ldots,k_j\}, \{\text{agg}_1,\ldots,\text{agg}_l, id_h\})$$

where

$$computeAgg = \big(M \to \text{agg}_1 = f_1((e^1_1,\ldots,e^1_{i_1})(M)),\ldots,\text{agg}_l = f_l((e^l_1,\ldots,e^l_{i_l})(M)), id_h = fresh()\big)$$

and

$$computeKeys = \theta(multi(t), m \to (k_1 = e'_1(m),\ldots,k_j = e'_j(m)) : * \to \{k_1,\ldots,k_j\})$$

In the translation of terms id_h has to be different for each different term so that id never conflicts. The function $fresh()$ is executed at the evaluation time to provide a unique id for mappings for the bag semantics.

Figure 3.11: Translation of a query in bag semantics.

$$
\begin{array}{rcll}
distinct(Join(a,b)) & = & distinct(a) \bowtie distinct(b) & \\
distinct(Union(a,b)) & = & distinct(a) \cup distinct(b) & \\
distinct(Project(V,t)) & = & \pi_{a_1,\ldots,a_n}(distinct(t)) & \{a_1,\ldots,a_n\} \text{ variables to remove} \\
distinct(PP(a)) & = & PP(a) & \\
distinct(Filter(f,t)) & = & \sigma_f(distinct(t)) & \\
distinct(Diff(f,t_1,t_2)) & = & Diff(f, distinct(t_1), distinct(t_2)) & \\
distinct(Minus(t_1,t_2)) & = & distinct(t_1) \setminus distinct(t_2) & \\
distinct(Extend(t,a,v)) & = & \beta_v^a(distinct(t)) & \text{when } v \text{ is a column} \\
distinct(Extend(t,a,v)) & = & distinct(t) \bowtie |a \to v| & \text{when } v \text{ is a constant} \\
distinct(Extend(t,a,v)) & = & \theta(distinct(t), (m \to |a \to v(m)| : FC(v) \to \{a\}) & \\
distinct(Graph(i,t)) & = & \text{let } (T = T_v) \text{ in } distinct(t) & \text{for } v \text{ IRI} \\
distinct(Graph(v,t)) & = & \text{let } (T = T_{u_1}) \text{ in } distinct(t) \bowtie |v \to u_1| \bowtie \ldots \text{let } (T = T_{u_k}) \text{ in } distinct(t) \bowtie |v \to u_k| & \\
 & & & \text{for } v \text{ variable}
\end{array}
$$

$$\text{with } G = Group(e'_1,\ldots,e'_j,t)$$

$$multi(AggregationJoin(Aggregation(e^1_1,\ldots,e^1_{i_1},f_1,scalarvals,G),\ldots,Aggregation(e^l_1,\ldots,e^l_{i_l},f_l,scalarvals,G))) =$$

$$\Theta\left(computeKeys, computeAgg, \{k_1,\ldots,k_j\}, \{\text{agg}_1,\ldots,\text{agg}_l\}\right)$$

where

$$computeAgg = \left(M \to \text{agg}_1 = f_1((e^1_1,\ldots,e^1_{i_1})(M)),\ldots,\text{agg}_l = f_l((e^l_1,\ldots,e^l_{i_l})(M))\right)$$

and

$$computeKeys = \theta(multi(t), m \to (k_1 = e'_1(m),\ldots,k_j = e'_j(m)) : * \to \{k_1,\ldots,k_j\})$$

Figure 3.12: Translation of a distinct query.

```
SELECT *
{
  ?a :knows* ?b .
  ?b :firstname :John .
  ?a :lastname :Doe .
}
```

Figure 3.13: Example SPARQL query

First, this query is translated into a SPARQL Algebra term:

$$Join(Join(PP(?b\ \texttt{:firstname :John}), PP(?a\ \texttt{:knows}^*\ ?b)), PP(?b\ : lastname\ \texttt{:Doe})$$

Then each PP is translated:

$$\begin{aligned}
pp(?b\ \texttt{:firstname :John}) &= \rho_s^{?b}\left(\pi_o\left(\sigma_{o=\texttt{:John}}(\pi_p\left(\sigma_{p=\texttt{:firstname}}(T)\right)))\right)\right) \\
pp(?a\ \texttt{:lastname :Doe}) &= \rho_s^{?a}\left(\pi_o\left(\sigma_{o=\texttt{:Doe}}(\pi_p\left(\sigma_{p=\texttt{:firstname}}(T)\right)))\right)\right) \\
pp(?a\ \texttt{:knows}^*\ ?b) &= \rho_s^{?a}\left(\rho_o^{?b}\left(rpe(\texttt{:knows}^*)\right)\right) \\
rpe(\texttt{:knows}^*) &= \mu(X = \beta_s^o\left(N\right) \cup \pi_m\left(\rho_o^m\left(X\right) \bowtie \rho_s^m\left(\pi_p\left(\sigma_{p=\texttt{:knows}}(T)\right)\right)\right))
\end{aligned}$$

Finally, the total term is: $rpe(?a\ \texttt{:knows}^*\ ?b) \bowtie rpe(?b\ \texttt{:firstname :John}) =$

$$\left(\rho_s^{?b}\left(\pi_o\left(\sigma_{o=\texttt{:John}}(\pi_p\left(\sigma_{p=\texttt{:firstname}}(T)\right))\right)\right)\right) \bowtie \left(\rho_s^{?a}\left(\pi_o\left(\sigma_{o=\texttt{:Doe}}(\pi_p\left(\sigma_{p=\texttt{:lastname}}(T)\right))\right)\right)\right) \bowtie$$

$$\left(\rho_s^{?a}\left(\rho_o^b\left(\mu(X = \beta_s^o\left(N\right) \cup \pi_m\left(\rho_o^m\left(X\right) \bowtie \rho_s^m\left(\pi_p\left(\sigma_{p=\texttt{:knows}}(T)\right)\right)\right)\right)\right)\right)$$

Conclusion

In this chapter we presented a new algebra: the μ-algebra. This algebra takes its inspiration from the relational algebra but diverges on some essential points: as it name suggests the relational algebra describes relations while solutions to our algebra are set of mappings that do not necessarily share the same domain. Furthermore our algebra is equipped with a novel fixpoint operator μ. For this fixpoint to be well-defined we restrict ourselves to "linear" terms and introduce a translation from a large fragment of the SPARQL–algebra to our μ-algebra.

Since our algebra is a variation of the relational algebra and since the relational algebra optimization is based on rewrite rules, the direction we will now follow is to investigate the rewrite rules and strategy that can be applied for our μ-algebra.

CHAPTER 4

Analysis & transformation of μ-algebra terms

As we have now presented the μ-algebra and a translation from SPARQL*, we can now dive into the process of optimization. This optimization process is divided into several steps the first of which being the production of several terms equivalent to the initial term. While all these terms are* semantically *equivalent they correspond to diverse ways of actually computing the solutions. Therefore by producing several terms we are considering several ways of computing the solutions among which some might be better than directly using the initial term.*

More precisely, in this chapter we present the mechanisms to produce many terms equivalent to given a μ-algebra term using rewriting rules*. These rewriting rules are local patterns that could be replaced with other patterns.*

While some of the rewrite rules are relatively straightforward to prove, others will require some theoretical framework to even enunciate. The first part of this

chapter presents definitions, lemmas and theorems that are the core machinery that will later translate into proven rewrite rules. In particular, several theorems we introduce translate into powerful new rewrite rules for queries with recursion.

The validity of our rewrite rules will also depend on some contextual information. For instance, the rule $\pi_c(\varphi_1 \cup X) \rightarrow \pi_c(\varphi_1) \cup X$ *is only true in environment* V *such that the domain of the mappings of* $V[X]$ *do not contain* c*. That is why our chapter starts with the presentation of a type system for* μ*-algebra terms that will provide us with this contextual information.*

The chapter continues by presenting our various rules and when to apply them. Our set of rules is split between normalizing *rules and* producing *rules.* Normalizing *rules are used to produce canonical versions of our terms while* producing *rules produce terms that are really new. The presentation of these sets of rules is the subject of the two sections preceding the one on our algorithm.*

This chapter ends by the presentation of our efficient algorithm to produce the rewritten forms of μ*-algebra terms and an example of term rewritten.*

4.1 Preliminaries: several examples of terms to be rewritten

In order to motivate the heavy framework of notations that we will present after, we start this chapter with a few examples of queries and what kind of rewriting we would like to apply on those queries.

4.1.1 Filters

Let us imagine we have a graph database containing edges labeled with the predicate `:locatedIn` we might want to use this predicate in a transitive way. For instance, if we have `:Grenoble :locatedIn :Isère` and `:Isère :locatedIn :France` then the query below would retrieve ($?a \rightarrow$`:Grenoble`, $?b \rightarrow$`:France`) as expected.

```
SELECT * WHERE {
   ?a :locatedIn+ ?b .
   FILTER(?b = :France)
}
```

From an evaluation point of view, we do not want the query optimizer to choose a plan where it computes the whole transitive closure `:locatedIn` as it is reasonable to think that only a portion of all objects are located in `:France`. In terms of rewrite rules we want the filter condition to pushed inside the computation of the fixpoint. Note that it is possible for the query evaluator to push some kind of filters inside fixpoints but not all fixpoints can be pushed without changing the semantics. For instance if the μ-algebra term corresponding to `?a :locatedIn+ ?b` starts with a solution $(?a, ?b)$ then transform at each iteration a solution $(?a, ?b)$ into a solution $(?a', ?b)$ then it is possible to push a filter condition that filters $?b$ (as $?b$ do not change). In the same term it would not be possible to push the filter if the filter was considering the column $?a$. Indeed if we have $(?a_1, ?b)$ at iteration 1, $(?a_2, ?b)$ at iteration 2, then $?a_2$ might have passed the filter while $?a_1$ might have not. In which case pushing the filter would remove the solution $(?a_2, ?b)$.

4.1.2 Joins

In a query such as the following:

```
SELECT * WHERE {
   ?a :locatedIn+ ?b .
   ?b label "France" .
}
```

The join act as a filter. Therefore depending on the fixpoint and the column bound by the join, it might or might not be pushed into the fixpoint. The condition in general to push filters into fixpoint is actually more complicated than the condition to push filters as pushing a join can introduce new columns that can interact with the fixpoint.

For instance in a μ-algebra term such as $\mu\big(X = \pi_m\left(\rho_m^b\left(X\right) \bowtie \varphi\right) \cup \psi\big)$ if neither φ nor ψ binds m then the recursive solutions of X will not bind m. If we push a join that bind m then this m will be renamed into b and might create new condition for the join with φ.

4.1.3 N-ary fixpoints

Let us suppose we have the following query:

```
SELECT * WHERE {
   ?a :locatedIn+/:sameAs+ ?b .
}
```

One way to compute the solution the solutions to this query is to use the equivalent following query:

```
SELECT ?a ?b WHERE {
   ?a :locatedIn+ ?c .
   ?c :sameAs+ ?b .
}
```

Then the computation performs two fixpoints and joins the results. It seems however inefficient because `:sameAs` might concern a lot of objects that are not places and thus do not have a `:locatedIn` predicate. Conversely, many places might not have a `:sameAs` information. In practice, what we would like the evaluator to do in this setting is to start by computing the pairs $(?a, ?b)$ of solutions to `?a :locatedIn/:sameAs ?b` and then recursively build on that, i.e. at each iteration we create either $(?a', ?b)$ (or $(?a, ?b')$) from $(?a, ?b)$ such that $?a'$ `:locatedIn` $?a$ (or $?b'$ `:sameAs` $?b$). This way, our query execution plan is optimal in the sense that the only partial solutions that we manipulate are actual solutions.

4.2 Effects of μ-algebra terms

In this section, we will develop a logical framework (with definitions, lemmas and theorem) that will help us analyze the behavior of μ-algebra terms. More precisely we will have tools describing which variables are defined along with a substitution method, how a fixpoint uses recursively its columns and how we can modify its type without changing its semantics. This logical framework will then be put into practice to demonstrate rewrite rules concerning fixpoints.

4.2.1 Expansion of let binders

Given a term let $(X = \varphi)$ in ψ we might be tempted to rewrite that into $\psi[X/\varphi]$, i.e. the term ψ where the occurrences of X are replaced by φ. However that is not always semantically equivalent: we need to take into account the fact that variables might be unbound in φ and bound ψ or change scope (if the same variable is bound twice). For instance with $\varphi = Y$ and $\psi = \text{let } (Y = |c_1 \to v_1, \ldots, c_n \to v_n|) \text{ in } (X \bowtie Y)$ we have $[\![\text{let } (X = \varphi) \text{ in } \psi]\!]_V = [\![\text{let } (X = Y) \text{ in } \psi]\!]_V =$
$[\![\psi]\!]_{V[X/V(Y)]} = [\![\text{let } (Y = |c_1 \to v_1, \ldots, c_n \to v_n|) \text{ in } (X \bowtie Y)]\!]_{V[X/V(Y)]} =$
$[\![X \bowtie Y]\!]_{V[X/V(Y),Y/\{\{c_1 \to v_1,\ldots,c_n \to v_n\}\}]} = [\![|c_1 \to v_1, \ldots, c_n \to v_n| \bowtie Y]\!]_V$ which is generally different from $[\![\psi[X/Y]]\!]_V = [\![\text{let } (Y = |c_1 \to v_1, \ldots, c_n \to v_n|) \text{ in } Y \bowtie Y]\!]_V =$
$[\![Y \bowtie Y]\!]_{V[Y/\{c_1 \to v_1,\ldots,c_n \to v_n\}]} = \{\{c_1 \to v_1, \ldots, c_n \to v_n\}\}$.

The definition 18 defines the intuitive notion of the set of variables defined in a term and definition 19 defines a context-aware replacement scheme while the lemmas 4 and 3 show we can replace variables with their definition when φ is constant in the variables defined in ψ.

Definition 18. *The set $def(\varphi)$ of variables defined in φ in defined recursively as:*

$$
\begin{array}{lcl}
def(\varphi_1 \cup \varphi_2) & = & def(\varphi_1) \cup def(\varphi_2) \\
def(\varphi_1 \backslash\backslash \varphi_2) & = & def(\varphi_1) \cup def(\varphi_2) \\
def(\varphi_1 \setminus \varphi_2) & = & def(\varphi_1) \cup def(\varphi_2) \\
def(\varphi_1 - \varphi_2) & = & def(\varphi_1) \cup def(\varphi_2) \\
def(\varphi_1 \rtimes \varphi_2) & = & def(\varphi_1) \cup def(\varphi_2) \\
def(\varphi_1 \bowtie \varphi_2) & = & def(\varphi_1) \cup def(\varphi_2) \\
 & & \\
def(\rho_a^b(\varphi)) & = & def(\varphi) \\
def(\pi_a(\varphi)) & = & def(\varphi) \\
def(\beta_a^b(\varphi)) & = & def(\varphi) \\
def(\theta(\varphi, g : \mathcal{C} \to \mathcal{D})) & = & def(\varphi) \\
def(\Theta(\varphi, g, \mathcal{C}, \mathcal{D})) & = & def(\varphi) \\
def(\sigma_f(\varphi)) & = & def(\varphi) \\
def(X) & = & \emptyset \\
def(\emptyset) & = & \emptyset \\
def(|c_1 \to v_1, \ldots, c_n \to v_n|) & = & \emptyset \\
def(\mu(Y = \varphi)) & = & \{Y\} \cup def(\varphi) \\
def(let\ \ (Y = \varphi)\ \ in\ \psi) & = & \{Y\} \cup def(\varphi) \cup def(\psi)
\end{array}
$$

Definition 19. *The replace operator $replace(\varphi, X, \psi)$, replaces free occurrences of X in φ with ψ.*

$$
\begin{array}{lcll}
replace(\varphi_1 \cup \varphi_2, X, \psi) & = & replace(\varphi_1, X, \psi) \cup replace(\varphi_2, X, \psi) & \\
replace(\varphi_1 \backslash\backslash \varphi_2, X, \psi) & = & replace(\varphi_1, X, \psi) \backslash\backslash\, replace(\varphi_2, X, \psi) & \\
replace(\varphi_1 \setminus \varphi_2, X, \psi) & = & replace(\varphi_1, X, \psi) \setminus replace(\varphi_2, X, \psi) & \\
replace(\varphi_1 - \varphi_2, X, \psi) & = & replace(\varphi_1, X, \psi) - replace(\varphi_2, X, \psi) & \\
replace(\varphi_1 \rtimes \varphi_2, X, \psi) & = & replace(\varphi_1, X, \psi) \rtimes replace(\varphi_2, X, \psi) & \\
replace(\varphi_1 \bowtie \varphi_2, X, \psi) & = & replace(\varphi_1, X, \psi) \bowtie replace(\varphi_2, X, \psi) & \\
 & & & \\
replace(\rho_a^b(\varphi), X, \psi) & = & \rho_a^b(replace(\varphi, X, \psi)) & \\
replace(\pi_a(\varphi), X, \psi) & = & \pi_a(replace(\varphi, X, \psi)) & \\
replace(\beta_a^b(\varphi), X, \psi) & = & \beta_a^b(replace(\varphi, X, \psi)) & \\
replace(\theta(\varphi, g : \mathcal{C} \to \mathcal{D}), X, \psi) & = & \theta(replace(\varphi, X, \psi), g : \mathcal{C} \to \mathcal{D}) & \\
replace(\Theta(\varphi, g, \mathcal{C}, \mathcal{D}), X, \psi) & = & \Theta(replace(\varphi, X, \psi), g, \mathcal{C}, \mathcal{D}) & \\
replace(\sigma_f(\varphi), X, \psi) & = & \sigma_f(replace(\varphi, X, \psi)) & \\
replace(\mu(Y = \varphi), X, \psi) & = & \mu(Y = replace(\varphi, X, \psi)) & X \neq Y \\
replace(\mu(X = \varphi), X, \psi) & = & \mu(X = \varphi) & \\
replace(X, X, \psi) & = & \psi & \\
replace(\emptyset, X, \psi) & = & \emptyset & \\
replace(|c_1 \to v_1, \ldots, c_n \to v_n|, X, \psi) & = & |c_1 \to v_1, \ldots, c_n \to v_n| & \\
replace(let\ \ (X = \varphi)\ \ in\ \xi, X, \psi) & = & let\ \ (X = \varphi)\ \ in\ \xi & \\
replace(let\ \ (Y = \varphi)\ \ in\ \xi, X, \psi) & = & let\ \ (Y = replace(\varphi, X, \psi))\ \ in\ replace(\xi, X, \psi) &
\end{array}
$$

Lemma 3. *Let φ and ψ be terms such that for all variables $Y \in def(\psi)$, $sim(\varphi, Y) = 0$, then $[\![replace(\psi, X, \varphi)]\!]_V = [\![\psi]\!]_{V[X/[\![\varphi]\!]_V]}$.*

Lemma 4. *Let φ and ψ be terms such that for all variables $Y \in def(\psi)$, $sim(\varphi, Y) = 0$ and $X \notin def(\psi)$; then $[\![\psi[X/\varphi]]\!]_V = [\![\psi]\!]_{V[X/[\![\varphi]\!]_V]}$.*

Proof. Lemma 4 is a consequence of lemma 3 as $\psi[X/\varphi] = replace(\psi, X, \varphi)$ when $X \notin def(\psi)$.

The idea of the proof relies on the fact that capturing the set $S = [\![\varphi]\!]_V$ of solutions for X before evaluating $[\![\psi]\!]_{V[X/S]}$ is equivalent to computing $[\![\varphi]\!]_{V'}$ for each free occurrence of X in ψ. In general $[\![\varphi]\!]_{V'}$ and $[\![\varphi]\!]_V$ might differ but since V and V' only differ for variables in $def(\psi)$; for all $Y \in def(\psi)$, $sim(\varphi, Y) = 0$ and thanks to lemma 1 thus $[\![\varphi]\!]_V = [\![\varphi]\!]_{V'}$.

□

4.2.2 Image of a variable

In this section we use the notation $p(c) = \perp$ to indicate that the image of c through the function (or mapping) p is not defined. We introduce the *perm* operator. The operator $perm(\varphi, X, C)$ computes the different ways an element of $m \in [\![\varphi]\!]_{X[V/\{w\}]}$ that is not in $[\![\varphi]\!]_{X[V/\emptyset]}$ can be obtained (more specifically, how m and w are related see lemma 6).

This *perm* operator will later be used as a criterion for rewriting rules over fixpoints.

Definition 20. *We note $perm(\varphi, X, C)$ the set of permutations on the set C of columns of the variable X in the term φ. It is defined recursively as:*

$$
\begin{array}{lcl}
perm(\varphi_1 \cup \varphi_2, X, C) & = & perm(\varphi_1, X, C) \cup perm(\varphi_2, X, C) \\
perm(\varphi_1 \backslash\backslash \varphi_2, X, C) & = & perm(\varphi_1, X, C) \\
perm(\varphi_1 \setminus \varphi_2, X, C) & = & perm(\varphi_1, X, C) \\
perm(\varphi_1 - \varphi_2, X, C) & = & perm(\varphi_1, X, C) \\
perm(\varphi_1 \rtimes \varphi_2, X, C) & = & perm(\varphi_1, X, C) \\
perm(\varphi_1 \bowtie \varphi_2, X, C) & = & perm(\varphi_1, X, C) \cup perm(\varphi_2, X, C)
\end{array}
$$

$$
\begin{array}{lcl}
perm(\rho_a^b(\varphi), X, C) & = & \{p' \mid p \in perm(\varphi, X, C) \quad p = p' \ \ except\ p'(a) = p(b)\ /\ p'(b) = p(a)\} \\
perm(\pi_a(\varphi), X, C) & = & \{p' \mid p \in perm(\varphi, X, C) \quad p = p' \ \ except\ p'(a) = \perp\} \\
perm(\beta_a^b(\varphi), X, C) & = & \{p' \mid p \in perm(\varphi, X, C) \quad p = p' \ \ except\ p'(b) = p(a)\} \\
perm(\sigma_f(\varphi), X, C) & = & perm(\varphi, X, C) \\
perm(\mu(Y = \varphi), X, C) & = & \emptyset \\
perm(X, X, C) & = & \{x \to x \mid x \in C\} \ \ (the\ identity\ function) \\
perm(\emptyset, X, C) & = & \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
perm(\theta(\varphi, g : \mathcal{C} \to \mathcal{D}), X, C) & = & \{p' \mid p \in perm(\varphi, X, C) \quad p = p' \ \ except\ c \in D \Rightarrow p'(c) = \perp\} \\
perm(\Theta(\varphi, g, \mathcal{C}, \mathcal{D}), X, C) & = & \emptyset \\
perm(|c_1 \to v_1, \ldots, c_n \to v_n|, X, C) & = & \emptyset \\
perm(let\ \ (X = \varphi)\ \ in\ \psi, X, C) & = & \emptyset
\end{array}
$$

$$
perm(let\ \ (Y = \varphi)\ \ in\ \psi, X, C) =
$$

$$
perm(\psi, X, C) \cup \{p_2 \circ p_1 \mid p_1 \in perm(\varphi, X, C)\ p_2 \in perm(\psi, Y, im(p_1))\}
$$

Lemma 5. *Given a term φ, a variable X, if $sim(\varphi, X) = 0$ then for all set of columns C we have $perm(\varphi, X, C) = \emptyset$*

Proof. Follows easily from the definition of *sim* and *perm*: if X does not appear as subterm of φ then $perm(\varphi, X, C) = \emptyset$. □

Lemma 6. *Let C be such that $C \subseteq dom(w)$, for all $w \in [\![\varphi]\!]_{V[X/\{w\}]} \setminus [\![\varphi]\!]_{V[X/\emptyset]}$ there exists $p \in perm(\varphi, X, C)$ such that $\forall c\ p(c) = \perp \vee (m(c) \neq \perp \wedge m(c) = w(p(c)))$.*

Proof Sketch. See full proof in appendix A.3.1.

The proof relies on the fact that the linearity syntactically forbid the combination of mappings coming out from a given variable. The *perm* operator tracks the possible provenance of a mapping of a variable and annotates which operations are applied on the mapping for each provenance. □

4.2.3 Adding columns to variables

With our rewriting rules we sometimes change the domains of mappings that are solutions of a fixpoint. In this section we introduce a function *canAdd* allowing us to derive some sufficient conditions for when the impact on the fixpoint semantics of extending the domain of a mapping is only on the extended part of the domain.

Definition 21. *Given a term φ and a column c, the function $canAdd(\varphi, X, c)$ determines whether modifying the column c of variable X has an impact on the solutions of φ, it is defined as:*

$$
\begin{array}{lcl}
canAdd(\varphi_1 \cup \varphi_2, X, c) & = & canAdd(\varphi_1, X, c) \wedge canAdd(\varphi_2, X, c) \\
canAdd(\varphi_1 \bowtie \varphi_2, X, c) & = & canAdd(\varphi_1, X, c) \wedge canAdd(\varphi_2, X, c) \\
 & & \\
canAdd(\varphi_1 \backslash\backslash \varphi_2, X, c) & = & canAdd(\varphi_1, X, c) \wedge canAdd(\varphi_2, X, c) \\
canAdd(\varphi_1 \setminus \varphi_2, X, c) & = & canAdd(\varphi_1, X, c) \wedge canAdd(\varphi_2, X, c) \\
canAdd(\varphi_1 - \varphi_2, X, c) & = & canAdd(\varphi_1, X, c) \wedge (sim(\varphi_1, X) = 0) \\
canAdd(\varphi_1 \ltimes \varphi_2, X, c) & = & canAdd(\varphi_1, X, c) \wedge canAdd(\varphi_2, X, c) \\
 & & \\
canAdd(\rho_a^b(\varphi), X, c) & = & canAdd(\varphi, X, c) \wedge c \notin \{a, b\} \\
canAdd(\pi_a(\varphi), X, c) & = & canAdd(\varphi, X, c) \ \ when\ c \neq a \\
canAdd(\pi_c(\varphi), X, c) & = & sim(\varphi, X) = 0 \\
canAdd(\beta_a^b(\varphi), X, c) & = & canAdd(\varphi, X, c) \wedge c \notin \{a, b\} \\
canAdd(\theta(\varphi, g : \mathcal{C} \to \mathcal{D}), X, c) & = & can(\varphi, X, c) \wedge c \notin \mathcal{C} \cup \mathcal{D} \\
canAdd(\Theta(\varphi, g, \mathcal{C}, \mathcal{D}), X, c) & = & \top \qquad (since\ sim(\Theta(\varphi, g, \mathcal{C}, \mathcal{D})) = 0) \\
canAdd(\sigma_f(\varphi), X, c) & = & canAdd(\varphi, X, c) \wedge c \notin FC(f) \\
canAdd(\mu(Y = \varphi), X, c) & = & canAdd(\varphi, X, c) \\
canAdd(Y, X, c) & = & \top \\
canAdd(|c_1 \to v_1, \ldots, c_n \to v_n|, X, c) & = & c \notin \{c_1, \ldots, c_n\} \\
canAdd(let\ \ (Y = \varphi)\ \ in\ \psi, X, c) & = & canAdd(\psi, X, c) \wedge canAdd(\varphi, X, c) \wedge canAdd(\psi, Y, c)
\end{array}
$$

Lemma 7. *Given a term φ, an environment V, a variable X and a mapping w, If $c \notin dom(w)$, $\forall Y, sim(\varphi, Y) = 0 \vee (\forall m \in V(Y) c \notin dom(m))$, and $canAdd(\varphi, X, c)$ then we have for all v (with $w(v) = w + \{c \to v\}$):*

1. $\forall m \in [\![\varphi]\!]_{V[X/\{w\}]}\ c \notin dom(m)$;
2. $[\![\varphi]\!]_{V[X/\{w\}]} = [\![\pi_c(\varphi)]\!]_{V[X/\{w(v)\}]}$;
3. $\forall m \in [\![\varphi]\!]_{V[X/\{w(v)\}]}\ \ c \notin dom(m) \vee m(c) = v$.

Proof Sketch. See full proof in appendix A.3.2.

The proof of point 1 and 3 relies on the fact that *canAdd* enforces that c syntactically appears in a subformula that is also a subformula of $\pi_c(\varphi)$. Therefore the c part of the domain cannot be altered since the term cannot reference it.

The proof of point 2 relies on the fact that there is only one possible value (v) for the c image a mapping solution. □

As we have seen *canAdd* provides information on whether a given column can be added to a variable without essentially changing the semantics of a formula (except on this column). We now present lemma 8 which reinforce this by extending *perm* for free for the added column and thus complete the result of lemma 7 with the conclusions of lemma 6.

Lemma 8. *Let φ be a μ-algebra term, X a variable, c a column and C a set of columns with $c \notin C$ and $canAdd(\varphi, X, c)$ then $perm(\varphi, X, C \cup \{c\}) = \{p \cup \{c \to c\} \mid p \in perm(\varphi, X, C)\}$.*

Proof Sketch. See full proof in appendix A.3.3.

The definition of *canAdd* essentially enforces that c does not appear syntactically while $perm(\varphi, X, C)$ returns the elements of subformulas and changes only the elements that appear syntactically. □

4.2.4 Application 1: A theorem on filtered fixpoints

We now present the applications of our lemmas. Theorem 2 present a sufficient condition for pushing filters inside fixpoints. Pushing filter down the evaluation tree is usually a good heuristic as it limit the size of intermediary results. However in the case of filter, it is not always possible.

For instance, consider the term that computes whether there exists a chain $john = p_1 ... p_n = bob$ of people with $\{s = p_i, o = p_{i+1}\} \in K$. A term computing that is $\sigma_{s=\text{"}john\text{"} \wedge o=\text{"}bob\text{"}}(\mu(X = \pi_{tmp}(\rho_o^{tmp}(K) \bowtie \rho_s^{tmp}(X)) \cup K))$ in which intermediate results do not pass the filter but might be essential to provide a solution. Indeed, the term where the filter is pushed $\mu(X = \sigma_{s=\text{"}john\text{"} \wedge o=\text{"}bob\text{"}}(\pi_{tmp}(\rho_o^{tmp}(K) \bowtie \rho_s^{tmp}(X)) \cup K))$ does not really compute a fixpoint, it computes K. This term does not meet the criterion 2 of the theorem because $perm(\pi_{tmp}(\rho_o^{tmp}(K) \bowtie \rho_s^{tmp}(X)) \cup K, X, \{s, o\}) = \{\{o \to o\}\}$ and thus we do not have $\{o \to o\}(s) = s$.

Theorem 2. *Let $\mu(X = \varphi)$ be a fixpoint, V an environment, C and a filter f with:*

1. $\forall m \in [\![\mu(X = \varphi)]\!]_V \quad C \subseteq dom(m)$
2. $\forall p \in perm(\varphi, X, C) \quad \forall d \in FC(f) \quad p(d) = d$

we have: $[\![\sigma_f(\mu(X = \varphi))]\!]_V = [\![\mu(X = \sigma_f(\varphi))]\!]_V$.

Proof Sketch. See full proof in appendix A.3.4.

The idea is that, in a linear fixpoint, a mapping m solution of $[\![\mu(X = \varphi)]\!]_V$ is either the solution of the initial term $m \in [\![\varphi]\!]_{V[X/\emptyset]}$ or the image of another mapping solution ($m \in [\![\varphi]\!]_{V[X/\{w\}]}$ for some w). By condition 2 of the theorem and lemma 6 we have that the image m of a mapping w passes the filter if and only w passes the filter. Therefore we can remove the element not passing the filter. □

4.2.5 Application 2: A theorem on fixpoints joined with other μ-algebra terms

The objective of this section is to present criteria allowing a fixpoint of the form $\mu(X = \varphi) \bowtie \psi$ to be rewritten into $\mu(X = \varphi \bowtie \psi)$. In order to better explain the chosen criteria, here are various ways of why $[\![\mu(X = \varphi) \bowtie \psi]\!]_V$ might differ from $[\![\mu(X = \varphi \bowtie \psi)]\!]_V$:

1. As for filters, performing the join of φ and ψ might remove some mappings that are not solution of $\mu(X = \varphi) \bowtie \psi$ but necessary to produce other solutions. For instance, the toy example of section 4.2.4 can also be expressed with a join: $\sigma_{s="john" \wedge o="bob"}(\varphi) = \varphi \bowtie |o \to "bob", s \to "john"|$. In this case the conditions for allowing the join should be similar to the condition allowing a filter.

2. Joins can also add some columns that might change the semantics of a fixpoint. For instance on the fixpoint $\mu(X = \pi_{tmp}(\rho_o^{tmp}(K) \bowtie \rho_s^{tmp}(X)) \cup K)$ if the mappings of X also bind the column tmp then the semantics changes: in $\mu(X = (\pi_{tmp}(\rho_o^{tmp}(K) \bowtie \rho_s^{tmp}(X)) \cup K) \bowtie |tmp \to v|)$ all recursive solutions m (i.e. all solutions except those from $K \bowtie |c \to tmp|$) will have $r(s) = r(tmp) = v$ which is generally not the same as $\mu(X = \pi_{tmp}(\rho_o^{tmp}(K) \bowtie \rho_s^{tmp}(X)) \cup K) \bowtie |tmp \to v|$ where solutions m of the fixpoint are simply extended with $m(tmp) = v$.

3. Finally the column we add might also interact with themselves. For instance, let $\mu(X = \varphi)$ be a fixpoint where a and b are not columns of solutions and do not appear. Then $\mu(X = \varphi \bowtie (|a \to v| \cup |b \to v|))$ has not the same solutions as $\mu(X = \varphi) \bowtie (|a \to v| \cup |b \to v|)$ because in the first term we will compute $(|a \to v| \cup |b \to v|)$ joined with itself which has 3 solutions $(\{a \to v\}, \{b \to v\}, \{a \to v, b \to v\})$ while $(|a \to v| \cup |b \to v|)$ has only two solutions.

The problem raised by point 1 is treated by lemma 9. The problem raised by point 2 is treated by lemma 10 and finally Theorem 3 wraps everything to provide a sufficient criterion for pushing joins into fixpoints.

Lemma 9. *Let φ and ψ be μ-algebra terms, V an environment and D and C be sets of columns such that:*

1. $\forall m \in [\![\psi]\!]_V \quad dom(m) \subseteq D \subseteq C$
2. $\forall m \in [\![\mu(X = \varphi)]\!]_V \quad D \subseteq C \subseteq dom(m)$
3. $\forall p \in perm(\varphi, X, C), c \in D \quad p(c) = c$

Then we have $[\![\psi \bowtie \mu(X = \varphi)]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$.

Proof Sketch. See full proof in appendix A.3.5.

The proof of this lemma is pretty similar to the proof of Theorem 2, we show that the join cannot add or change mapping (it thus acts as a filter) and that the removed mappings cannot be used to produce solutions passing the join. □

Lemma 10. *Let φ and ψ be μ-algebra terms, V an environment, c a column and D and C be sets of columns such that:*

1. $\forall m \in [\![\psi]\!]_V \quad dom(m) \subseteq D \cup \{c\} \subseteq C \cup \{c\}$
2. $\forall m \in [\![\mu(X = \varphi)]\!]_V \quad D \subseteq C \subseteq dom(m) \wedge c \notin dom(m)$
3. $\forall p \in perm(\varphi, X, C), c \in D \quad p(c) = c$
4. $canAdd(\varphi, X, c) = \top$

Then we have $[\![\psi \bowtie \mu(X = \varphi)]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$.

Proof Sketch. See full proof in appendix A.3.6.
The idea is to take leverage of lemma 7 to ensure the result holds on one iteration of the fixpoint and then proceed on induction (to follow the fixpoint semantics). □

Theorem 3. *Let $\mu(X = \varphi)$ be a fixpoint and ψ be a μ-algebra term, V an environment, and C, D, E sets with:*

1. $\forall m \in [\![\psi]\!]_V \quad dom(m) = E \cup D$
2. $\forall m \in [\![\mu(X = \varphi)]\!]_V \quad (D \subseteq C \subseteq dom(m)) \ \wedge \ (dom(m) \cap E = \emptyset)$
3. $\forall p \in perm(\varphi, X, C), d \in D \quad p(d) = d$
4. $\forall c \in E \quad canAdd(\varphi, X, c) = \top$
5. $sim(\psi, X) = 0$

we have: $[\![\psi \bowtie \mu(X = \varphi)]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$.

Proof Sketch. See full proof in appendix A.3.7.
The idea is to proceed by induction on the number of columns defined by ψ and at each column c depending on whether $c \in E$ or $c \in D$ we use either lemma 10 or lemma 9. □

Remark 2. *We could extend this theorem to allow one of the column from $D \cup E$ to appear only in some of the mappings (and not all). But we could not allow two of them: for instance if $[\![\psi]\!]_V$ contains the mappings $\{c \to 3, a \to 1\}$ and $\{c \to 3, b \to 2\}$ then $[\![\psi \bowtie \psi]\!]_V$ also contains $\{a \to 1, b \to 2, c \to 3\}$.*

4.3 Decomposed fixpoints

In this section we present the decomposition of fixpoints into constant and recursive parts and show some specialized version of the theorems proved above for decomposed fixpoints.

4.3.1 Recursive terms

Definition 22. *A term ψ is said* recursive *in X when for all V, $[\![\psi]\!]_{V[X/\emptyset]} = \emptyset$.*

Definition 23. *Given a term φ and a variable X the term φ is said* syntactically recursive in X *when $rec(\varphi, X) = \top$. rec is defined recursively as:*

$$
\begin{array}{rcl}
rec(\varphi_1 \backslash\backslash \varphi_2, X) & = & rec(\varphi_1, X) \\
rec(\varphi_1 \setminus \varphi_2, X) & = & rec(\varphi_1, X) \\
rec(\varphi_1 - \varphi_2, X) & = & rec(\varphi_1, X) \\
rec(\varphi_1 \rtimes \varphi_2, X) & = & rec(\varphi_1, X) \\
 & & \\
rec(\varphi_1 \cup \varphi_2, X) & = & rec(\varphi_1, X) \wedge rec(\varphi_2, X) \\
rec(\varphi_1 \bowtie \varphi_2, X) & = & rec(\varphi_1, X) \vee rec(\varphi_2, X) \\
 & & \\
rec(\rho_a^b(\varphi), X) & = & rec(\varphi, X) \\
rec(\pi_a(\varphi), X) & = & rec(\varphi, X) \\
rec(\beta_a^b(\varphi), X) & = & rec(\varphi, X) \\
rec(\theta(\varphi, g : \mathcal{C} \to \mathcal{D}), X) & = & rec(\varphi, X) \\
rec(\Theta(\varphi, g, \mathcal{C}, \mathcal{D}), X) & = & \bot \\
rec(\mu(Y = \varphi), X) & = & \bot \\
rec(let\ (Y = \varphi)\ in\ \psi, X) & = & (rec(\psi, Y) \wedge rec(\varphi, X)) \vee rec(\psi, X) \quad note\ X \neq Y \\
rec(X, Y) & = & X = Y \\
rec(\emptyset, X) & = & \top \\
rec(|c_1 \to v_1, \ldots, c_n \to v_n|, X) & = & \bot
\end{array}
$$

Lemma 11. *A term φ* syntactically recursive *in the variable X is recursive in X, i.e.* $\big(rec(\varphi, X) = \top\big) \Rightarrow \big(\forall V : [\![\varphi]\!]_{V[X/\emptyset]} = \emptyset\big)$.

Proof Sketch. See full proof in appendix A.3.8.

The proof is simply an induction on the size of φ such that $rec(\varphi, X) = \top$. □

4.3.2 Decomposed fixpoints

Definition 24. *A fixpoint $\mu(X = \varphi \cup \psi)$ is said* decomposed *into (φ, ψ) when φ is constant in X and ψ is recursive in X. φ is the* constant *part and ψ is the* recursive *part.*

4.3.3 Theorem for filtered and joined decomposed fixpoints

We now present specialized versions of Theorem 3 and Theorem 2 for decomposed fixpoints.

These theorems are important because they allow the rewriting of fixpoint in a way that can limit the size of intermediate computation (a filter always reduces the size and a join sometimes reduces the size) while always leaving the recursive part unchanged. In practice, recursive computations are expensive and limiting the size of the recursive term has a great impact on performance.

Theorem 4. *Let $\mu(X = \varphi \cup \psi)$ be a decomposed fixpoint with φ its constant part, ψ its recursive part and let κ be a μ-algebra term, V an environment, and C, D, E sets with:*

1. $\forall m \in [\![\kappa]\!]_V \quad dom(m) = E \cup D$

2. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \quad (D \subseteq C \subseteq dom(m)) \;\wedge\; (dom(m) \cap E = \emptyset)$

3. $\forall p \in perm(\psi, X, C), d \in D \quad p(d) = d$

4. $\forall c \in E \quad canAdd(\psi, X, c) = \top$

5. $sim(\kappa, X) = 0$

we have: $[\![\kappa \bowtie \mu(X = \varphi \cup \psi)]\!]_V = [\![\mu(X = \varphi \bowtie \kappa \cup \psi)]\!]_V$.

Proof Sketch. See full proof in appendix A.3.9.
The idea is simply to adapt the general proof in the case of a decomposed filter. □

Theorem 5. *Let $\mu(X = \varphi \cup \psi)$ be a decomposed fixpoint with φ its constant part and ψ its recursive part, V an environment, C and a filter f with:*

1. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \quad C \subseteq dom(m)$

2. $\forall p \in perm(\psi, X, C) \quad \forall d \in FC(f) \quad p(d) = d$

we have: $[\![\sigma_f(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu(X = \sigma_f(\varphi) \cup \psi)]\!]_V$.

Proof Sketch. See full proof in appendix A.3.10.
The idea is simply to adapt the general proof in the case of a decomposed filter. □

4.3.4 Theorem: Combination of decomposed fixpoints

We now present a theorem that allows us to merge two fixpoints (and by repetitive use, several fixpoints) into to just one fixpoint.

Theorem 6. *Given two decomposed fixpoints $\mu(X = \varphi \cup \psi)$ and $\mu(Y = \xi \cup \kappa)$ and $C_X, E_X, D_X, C_Y, E_Y, D_Y$ with:*

1. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \quad dom(m) = D_X \cup E_X = C_X$

2. $\forall m \in [\![\mu(Y = \kappa \cup \xi)]\!]_V \quad dom(m) = D_Y \cup E_Y = C_Y$

3. $D_Y \subseteq C_X$, $D_X \subseteq C_Y$, $E_X \cap C_Y = \emptyset$, $E_Y \cap C_X = \emptyset$

4. $\forall p \in perm(\psi, X, C_X), \forall c \in D_Y \quad p(c) = c$

5. $\forall p \in perm(\xi, Y, C_Y), \forall c \in D_X \quad p(c) = c$

6. $\forall c \in E_X \quad canAdd(\xi, Y, c)$

7. $\forall c \in E_Y \quad canAdd(\psi, X, c)$

Then we have: $[\![\mu(X = \varphi \cup \psi) \bowtie \mu(Y = \kappa \cup \xi)]\!]_V = [\![\mu(X = let\ (Y = X)\ in\ \varphi \bowtie \kappa \cup \psi \cup \xi)]\!]_V$ *and* $[\![\mu(X = \varphi \cup \psi) \bowtie \mu(Y = \kappa \cup \xi)]\!]_V = [\![\mu(X = \varphi \bowtie replace(\kappa, Y, X) \cup \psi \cup replace(\xi, Y, X))]\!]_V$

Proof Sketch. See full proof in appendix A.3.10.
This proof is an extension of the proof of Theorem 4 but that has to deal with the fact that both fixpoints are interacting with each other. In fact one can see Theorem 4 as a special case of this theorem where $\xi = \emptyset$. □

As explained earlier, fixpoint computations are costly and this theorem allows for a rewriting that reduces this cost in several different ways:

- first, all solutions of the produced fixpoint correspond to the join of two solutions of each fixpoints, therefore the number of intermediate solutions never increases;
- second, the number of intermediate solutions can be drastically reduced when joining two fixpoints since the base solutions (i.e. the non recursive part) are the join of the base solution of the two initial fixpoints;
- finally, a fixpoint has a an important fixed cost as we have to have some bookkeeping to test whether its computation has finished.

4.3.5 Application : decomposed fixpoints and other μ-algebra constructs

Theorem 7. *Given a decomposed fixpoint $\mu(X = \varphi \cup \psi)$ and a, b and C with $a \in C$ such that:*

1. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \; C \subseteq dom(m)$
2. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \; b \notin dom(m)$
3. $\forall p \in perm(\psi, X, C) \;\; p(a) = a$

then

1. $[\![\beta_a^b\,(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu\big(X = \beta_a^b\,(\varphi) \cup \psi\big)]\!]_V$ *when* $canAdd(\psi, X, b)$
2. $[\![\pi_a\,(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu(X = \pi_a\,(\varphi) \cup \psi)]\!]_V$ *when* $canAdd(\psi, X, a)$
3. $[\![\rho_a^b\,(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu\big(X = \rho_a^b\,(\varphi) \cup \psi\big)]\!]_V$ *when when* $canAdd(\psi, X, b) \wedge canAdd(\psi, X, a)$

Proof Sketch. See full proof in appendix A.3.11.

The theorem holds because:

1. values for b will propagate as constant but values for a also thanks to condition 3.
2. $canAdd(\psi, X, a)$ ensures that values for a are not relevant to the computation of ψ
3. $canAdd(\psi, X, a) \wedge canAdd(\psi, X, b)$ ensures that values for a and b are not relevant to the computation of ψ

□

4.4 Typing μ-algebra terms

We now present our type system for μ-algebra. This type system will be used in the next section in the definition of our rewriting rules. For instance, in a μ-algebra term like $\sigma_f(\varphi \bowtie \psi)$ we can sometimes push the filter into φ (i.e. it is equivalent to the term $\sigma_f(\varphi) \bowtie \psi$) but this rewritings depend on φ, on ψ but also on the environment they are evaluated under..

Our type system will allow us to detect situations where rewritings are sure to be correct. This analysis is similar to other analyses in the context of the SPARQL Algebra, see e.g. the definition of $cVars$ and $pVars$ [SML10a] that resembles our C and P, but we have variables and fixpoints which imply major changes.

4.4.1 Types

Our types are pairs (c, p) of sets of columns. Intuitively, a type (c, p) describes a set c of columns that are **c**ertain to be present in all mappings and a set p of columns that appear **p**ossibly. Formally, the pair (c, p) is a valid type for a set of mappings M when for each mapping $m \in M$ we have $c \subseteq dom(m) \subseteq p$.

Note that when (c, p) is the valid type for a set of mappings M then all (c', p') such that $c' \subseteq c$ and $p \subseteq p'$ are also valid types for M.

4.4.2 Types of μ-algebra terms

Given an environment V and a μ-algebra term φ, the type of $[\![\varphi]\!]_V$ depends, obviously, on φ but also on V. During the compilation (where our typing occurs) we don't want to use the exact environment V, otherwise in a term like let $(X = \varphi)$ in ψ we would need to actually compute $[\![\varphi]\!]_V$ in order to get the type of X in $V[X/[\![\varphi]\!]_V]$. Our typing judgments rely on abstract environments.

Definition 25. *An* abstract environment Γ *is a function that associates each variable X with a pair (c, p) of sets. An abstract environment is an* abstraction *of the environment V when for all variable X, $V[X] \neq \emptyset \Rightarrow \Gamma[Y]$ is a valid type for $V[X]$.*

Given a μ-algebra term φ and an abstract environment Γ associating each free variable X of φ with a type $\Gamma(X)$, the computed type for φ is $(C(\varphi, \Gamma), P(\varphi, \Gamma))$ with P and C as defined in figure 4.1.

In the context of SPARQL, the free variables of μ-algebra are Q, N, T and the T_n. The type of Q is $(\{s, p, o, g\}, \{s, p, o, g\})$, the type of N is $(\{s\}, \{s\})$ and the type of T (or T_n for any n) is $(\{s, p, o\}, \{s, p, o\})$ (all these types can be deduced from the type of Q).

For instance, with T typed as in SPARQL then the term $T \cup \rho_s^a\left(\rho_p^b(T)\right)$ is typed $(\{o\}, \{a, b, s, p, o\})$.

Definition 26. *The set $C(\varphi, \Gamma)$ of* certain *columns and the set $P(\varphi, \Gamma)$ of* possible *columns of the term φ under the abstract environment Γ is as defined as in figure 4.1.*

Note that, during the typing of a μ-algebra term φ, once the fixpoints are reached, each of the subterms of φ actually gets a unique type. The type of a subterm φ is $(C(\varphi, \Gamma), P(\varphi, \Gamma))$ but in the next sections we will abuse the notation and simply refer to $C(\varphi)$ and $P(\varphi)$ for subterms when the context makes it clear.

4.4.3 Validity of types

Lemma 12. *Given a term φ and a environment V and its abstraction Γ then $\forall m \in [\![\varphi]\!]_V \quad C(\varphi, \Gamma) \subseteq dom(m) \subseteq P(\varphi, \Gamma)$.*

Proof Sketch. See full proof in appendix A.3.11.

The proof relies on an induction on the size of considered formulas. The only special case is for fixpoints. Our proof does not rely on the actual value of $W_0^{C,\varphi}$: any value would have provided a correct type. The larger $W_0^{C,\varphi}$ we use, the more precise the type will get because a careful examination of $C(\varphi, \Gamma)$ shows that it is increasing in $\Gamma(Y)$ for all Y.

Therefore, for any pair Z_0^1, Z_0^2 of starting sets, with $j \in \{1, 2\}$, $Z_{i+1}^j = C(\varphi, \Gamma[X/(Z_i^j, \emptyset)])$, $Z_\infty^j = lim_{i \to \infty} Z_i^j$ we have $Z_\infty^1 \subseteq Z_0^2$ implies $Z_\infty^1 \subseteq Z_\infty^2$. Since $P(\mu(X = \varphi), \Gamma)$ corresponds

$$
\begin{array}{lcl}
P(\varphi_1 \cup \varphi_2, \Gamma) & = & P(\varphi_1, \Gamma) \cup P(\varphi_2, \Gamma) \\
P(\varphi_1 \backslash\backslash \varphi_2, \Gamma) & = & P(\varphi_1, \Gamma) \\
P(\varphi_1 \setminus \varphi_2, \Gamma) & = & P(\varphi_1, \Gamma) \\
P(\varphi_1 - \varphi_2, \Gamma) & = & P(\varphi_1, \Gamma) \\
P(\varphi_1 \rtimes \varphi_2, \Gamma) & = & P(\varphi_1, \Gamma) \\
P(\varphi_1 \bowtie \varphi_2, \Gamma) & = & P(\varphi_1, \Gamma) \cup P(\varphi_2, \Gamma) \\
\\
P(\rho_a^b(\varphi), \Gamma) & = & P(\varphi, \Gamma)[\{a\} \leftrightarrow \{b\}] \\
P(\pi_a(\varphi), \Gamma) & = & P(\varphi, \Gamma) \setminus \{a\} \\
P(\beta_a^b(\varphi), \Gamma) & = & P(\varphi, \Gamma)[\{a\} \rightarrow \{a, b\}] \\
P(\theta(\varphi, g : \mathcal{C} \rightarrow \mathcal{D}), \Gamma) & = & P(\varphi, \Gamma)[\mathcal{C} \rightarrow \mathcal{D}] \cup P(\varphi, \Gamma) \quad \text{when } \mathcal{C} \not\subseteq C(\varphi, \Gamma) \\
P(\theta(\varphi, g : \mathcal{C} \rightarrow \mathcal{D}), \Gamma) & = & P(\varphi, \Gamma)[\mathcal{C} \rightarrow \mathcal{D}] \quad \text{when } \mathcal{C} \subseteq C(\varphi, \Gamma) \\
P(\Theta(\varphi, g, \mathcal{C}, \mathcal{D}), \Gamma) & = & (P(\varphi, \Gamma) \setminus C) \cup D \\
P(\sigma_{\neg bnd(c)}(\varphi), \Gamma) & = & P(\varphi, \Gamma) \setminus \{c\} \\
P(\sigma_f(\varphi), \Gamma) & = & P(\varphi, \Gamma) \\
P(X, \Gamma) & = & \Gamma(X) \\
P(\emptyset, \Gamma) & = & \emptyset \\
P(|c_1 \rightarrow v_1, \ldots, c_n \rightarrow v_n|, \Gamma) & = & \{c_1, \ldots, c_n\} \\
P(\text{let } (Y = \varphi) \text{ in } \psi, \Gamma) & = & P(\psi, \Gamma[Y/(P(\varphi, \Gamma), \emptyset)]) \\
P(\mu(Y = \varphi), \Gamma) & = & lim_{i \rightarrow \infty} W_i^{P,\varphi}
\end{array}
$$

$$
\begin{array}{lcl}
C(\varphi_1 \cup \varphi_2, \Gamma) & = & C(\varphi_1, \Gamma) \cap C(\varphi_2, \Gamma) \\
C(\varphi_1 \backslash\backslash \varphi_2, \Gamma) & = & C(\varphi_1, \Gamma) \\
C(\varphi_1 \setminus \varphi_2, \Gamma) & = & C(\varphi_1, \Gamma) \\
C(\varphi_1 - \varphi_2, \Gamma) & = & C(\varphi_1, \Gamma) \\
C(\varphi_1 \rtimes \varphi_2, \Gamma) & = & C(\varphi_1, \Gamma) \\
C(\varphi_1 \bowtie \varphi_2, \Gamma) & = & C(\varphi_1, \Gamma) \cup C(\varphi_2, \Gamma) \\
\\
C(\rho_a^b(\varphi), \Gamma) & = & C(\varphi, \Gamma)[\{a\} \leftrightarrow \{b\}] \\
C(\pi_a(\varphi), \Gamma) & = & C(\varphi, \Gamma) \setminus \{a\} \\
C(\beta_a^b(\varphi), \Gamma) & = & C(\varphi, \Gamma)[\{a\} \rightarrow \{a, b\}] \\
C(\theta(\varphi, g : \mathcal{C} \rightarrow \mathcal{D}), \Gamma) & = & C(\varphi, \Gamma)[C \rightarrow C \cup D] \\
C(\sigma_{bnd(c)}(\varphi), \Gamma) & = & C(\varphi, \Gamma) \cup \{c\} \\
C(\sigma_f(\varphi), \Gamma) & = & C(\varphi, \Gamma) \\
C(X, \Gamma) & = & \Gamma(X) \\
C(\emptyset, \Gamma) & = & \emptyset \\
C(|c_1 \rightarrow v_1, \ldots, c_n \rightarrow v_n|, \Gamma) & = & \{c_1, \ldots, c_n\} \\
C(\text{let } (X = \varphi) \text{ in } \psi, \Gamma) & = & C(\psi, \Gamma[X/(C(\varphi, \Gamma), \emptyset)]) \\
C(\mu(X = \varphi), \Gamma) & = & lim_{i \rightarrow \infty} W_i^{C,\varphi}
\end{array}
$$

$$
\text{where } \begin{cases} W_0^{P,\varphi} = & \emptyset \\ W_{i+1}^{P,\varphi} = & P(\varphi, \Gamma[X/(\emptyset, W_i^{P,\varphi})]) \end{cases} \text{ and } \begin{cases} W_0^{C,\varphi} = & P(\mu(Y = \varphi), \Gamma) \\ W_{i+1}^{C,\varphi} = & W_i^{C,\varphi} \cap C(\varphi, \Gamma[X/(W_i^{C,\varphi}, \emptyset)]) \end{cases}.
$$

Figure 4.1: Definition of P and C.

to a valid type, therefore if $[\![\varphi]\!]_V$ is not empty for V abstracted by Γ we have $Z^1\infty \subseteq P(\mu(X = \varphi), \Gamma)$ and thus using $W_0^{C,\varphi} = P(\mu(X = \varphi), \Gamma)$ gives the largest set possible for $C(\mu(X = \varphi), \Gamma)$ and thus the most precise (the most precise four our methods; other methods could find a tightest type). □

4.4.4 Precision of types

Deciding the most precise type (i.e. decide whether a given column is sure to be present or will never be present) is a hard problem. The more precise the analysis, the more rewriting we can do and therefore the more optimization we get. However a perfectly precise analysis is often impossible: if it were feasible we would be able to detect μ-algebra terms corresponding to $\emptyset$ and thus, by translation, the SPARQL terms that always answer nothing. We therefore stick to a syntactical analysis that might sometimes be imprecise.

This syntactical analysis provides approximations that are conservative: if Γ is an abstract environment of V then the domain of a mapping of $[\![\varphi]\!]_V$ will be a subset of $P(\varphi, \Gamma)$ and it will always contain $C(\varphi, \Gamma)$.

Note that the main source of imprecision comes from subterms whose semantics is actually empty, our analysis is therefore relatively precise for most useful terms.

4.4.5 Computation of types

The figure 4.1 gives us a definition of P and C but since it is based on a fixpoint semantics, we need to show that these fixpoint do actually terminate. The termination of the computation is implied by the finiteness of $P(\varphi, \Gamma)$ and $C(\varphi, \Gamma)$ for all φ. Indeed, since μ-algebra terms are finite, a computation of P or C that would not terminate would correspond to a fixpoint with an infinite number of columns. However $C(\mu(X = \varphi), \Gamma)$ terminates if $P(\mu(X = \varphi), \Gamma)$ is finite and $P(\varphi, \Gamma)$ contains, at most: p for $(c, p) \in \Gamma(Y)$ and Y appearing in φ, $\{a, b\}$ for any subformula of the form $\rho_a^b(\xi)$, $\pi_a(\xi)$ or $\beta_a^b(\xi)$ and D for any subformula of the form $\theta(\xi, g : \mathcal{C} \to \mathcal{D})$. Since φ and all of the $\Gamma(Y)$ are finite, $P(\varphi, \Gamma)$ is also finite.

4.5 Normalizing rules for μ-algebra terms

The general idea used to produce those equivalent forms is similar to the optimization of the relational algebra: we have a set of "rewriting rules" that detect patterns in a μ-algebra that could be replaced by some other pattern. We maintain a set of equivalent μ-algebra terms. At the beginning, the set is composed of our initial term (i.e. the μ-algebra term obtained via the SPARQL translation). Then we recursively apply the rules, wherever they are applicable, to each μ-algebra term we obtain until we reach a fixpoint.

With a very general set of rewriting rules, the termination of the above algorithm is not certain. For instance, with only the valid rule $\varphi \to \varphi \cup \emptyset$, our algorithm will never terminate. So we have to make sure that such behaviour does not arise.

Furthermore, even with a terminating set of rules, the running time of our algorithm might prevent effective use on real queries. That is why we split our rules into *normalizing* rules that reduce the number of μ-algebra terms considered and *generating* rules producing new μ-algebra terms. Each time a new μ-algebra term t is produced by a *generating* rule we use the *normalizing* rules on t until no more can be applied.

4.5.1 Simplification of constant terms

An expression is said constant when it does not contain variables (i.e. $\forall X \quad sim(\varphi, X) = 0)$). Constant terms can always be computed at compile-time to a set of l mappings and thus be replaced by an expression of the form $|c_1^1 \to v_1^1, \ldots, c_n^1 \to v_n^1| \cup \cdots \cup |c_1^l \to v_1^l, \ldots, c_n^l \to v_n^l|$ when $l > 0$ or $\emptyset$ when $l = 0$.

4.5.2 Removing $\emptyset$

Formulas containing $\emptyset$ can always be rewritten to simpler formulas that are either $\emptyset$ or formulas with no $\emptyset$ as subformulas.

$$
\begin{array}{lcr}
\emptyset \bowtie \varphi & \to & \emptyset \\
\varphi \bowtie \emptyset & \to & \emptyset \\
\emptyset \setminus \varphi & \to & \emptyset \\
\emptyset \setminus\setminus \varphi & \to & \emptyset \\
\emptyset - \varphi & \to & \emptyset \\
\varphi \setminus \emptyset & \to & \varphi \\
\varphi \setminus\setminus \emptyset & \to & \varphi \\
\varphi - \emptyset & \to & \varphi \\
\emptyset \ltimes \varphi & \to & \emptyset \\
\varphi \rtimes \emptyset & \to & \varphi \\
\emptyset \cup \varphi & \to & \varphi \\
\varphi \cup \emptyset & \to & \varphi \\
\sigma_f(\emptyset) & \to & \emptyset \\
\theta(\emptyset, g : \mathcal{C} \to \mathcal{D}) & \to & \emptyset \\
\Theta\,(\emptyset, g, \mathcal{C}, \mathcal{D}) & \to & \emptyset \\
\rho_a^b\,(\emptyset) & \to & \emptyset \\
\pi_b\,(\emptyset) & \to & \emptyset \\
\beta_a^b\,(\emptyset) & \to & \emptyset \\
\text{let } (X = \emptyset) \text{ in } \psi & \to & replace(\psi, X, \emptyset) \\
\text{let } (X = \varphi) \text{ in } \emptyset & \to & \emptyset \\
\mu(X = \emptyset) & \to & \emptyset
\end{array}
$$

The proof of the equivalence behind these rewritings is straightforward.

4.5.3 Pushing renamings

There are a few operators that we included in our grammar that don't change the way the computation is done: renamings and projections. Renaming has zero effect on the way mappings are computed, and projection only limits the amount of data which needs to be computed. Therefore, we push renaming down the tree and we push projection up the tree (renamings are done as soon as possible and projections at the last moment). Projections are pushed upwards to the innermost μ (or the root of the tree if there is no upward μ), while renamings always appear in front of variables.

In this part, a, b and c are necessarily different. $fresh$ corresponds to a new unique variable name. The notation $\varphi[u/w]$ indicates that every occurrence of u in φ is replaced by w.

$$
\begin{array}{lcr}
\rho_a^a(\varphi) & \rightarrow & \varphi \\
\rho_a^b(\varphi_1 \cup \varphi_2) & \rightarrow & \rho_a^b(\varphi_1) \cup \rho_a^b(\varphi_2) \\
\rho_a^b(\varphi_1 \bowtie \varphi_2) & \rightarrow & \rho_a^b(\varphi_1) \bowtie \rho_a^b(\varphi_2) \\
\rho_a^b(\varphi_1 \ltimes \varphi_2) & \rightarrow & \rho_a^b(\varphi_1) \ltimes \rho_a^b(\varphi_2) \\
\rho_a^b(\varphi_1 \setminus \varphi_2) & \rightarrow & \rho_a^b(\varphi_1) \setminus \rho_a^b(\varphi_2) \\
\rho_a^b(\varphi_1 \setminus\setminus \varphi_2) & \rightarrow & \rho_a^b(\varphi_1) \setminus\setminus \rho_a^b(\varphi_2) \\
\rho_a^b(\varphi_1 - \varphi_2) & \rightarrow & \rho_a^b(\varphi_1) - \rho_a^b(\varphi_2) \\
\rho_a^b(\sigma_f(\varphi)) & \rightarrow & \sigma_{f[a\leftrightarrow b]}(\rho_a^b(\varphi)) \\
\rho_a^b(\theta(\varphi, g : \mathcal{C} \rightarrow \mathcal{D})) & \rightarrow & \theta(\rho_a^b(\varphi), g[a \leftrightarrow b] : \mathcal{C}[a \leftrightarrow b] \rightarrow \mathcal{D}[a \leftrightarrow b]) \\
\rho_a^b(\Theta(\varphi, g, \mathcal{C}, \mathcal{D})) & \rightarrow & \Theta\left(\rho_a^b(\varphi), g[a \leftrightarrow b], \mathcal{C}[a \leftrightarrow b], \mathcal{D}[a \leftrightarrow b]\right) \\
\rho_a^b(\pi_b(\varphi)) & \rightarrow & \pi_{fresh}\left(\rho_a^b\left(\rho_b^{fresh}(\varphi)\right)\right) \\
\rho_a^b(\pi_a(\varphi)) & \rightarrow & \pi_a(\varphi) \\
\rho_a^b(\pi_c(\varphi)) & \rightarrow & \pi_c\left(\rho_a^b(\varphi)\right) \\
\rho_a^b(\text{let } (X = \varphi) \text{ in } \psi) & \rightarrow & \text{let } (X = \varphi) \text{ in } \rho_a^b(\psi) \\
\rho_a^b(\mu(X = \varphi)) & \rightarrow & \mu\left(X = \rho_a^b(\varphi[X/\rho_b^a(X)])\right)
\end{array}
$$

The only unchanged formula $\rho_a^b(\varphi)$ are when φ is a variable or a renaming. Therefore, in a normalized formula, where it is not possible to apply normalizing rules anymore, renamings can only appear in front of a variable, either directly or through a chain of renaming (i.e. $\rho_{a_1}^{b_1}\left(\ldots(\rho_{a_n}^{b_n}(X))\right)$).

4.5.4 Pushing column projections

In the same manner we push toward the top the removal of columns:

$$
\begin{array}{lcr}
\pi_a(\varphi_1) \cup \varphi_2 & \rightarrow & \pi_{fresh}\left(\rho_a^{fresh}(\varphi_1) \cup \varphi_2\right) \\
\pi_a(\varphi_1) \bowtie \varphi_2 & \rightarrow & \pi_{fresh}\left(\rho_a^{fresh}(\varphi_1) \bowtie \varphi_2\right) \\
\pi_a(\varphi_1) \ltimes \varphi_2 & \rightarrow & \pi_{fresh}\left(\rho_a^{fresh}(\varphi_1) \ltimes \varphi_2\right) \\
\pi_a(\varphi_1) \setminus \varphi_2 & \rightarrow & \pi_{fresh}\left(\rho_a^{fresh}(\varphi_1) \setminus \varphi_2\right) \\
\pi_a(\varphi_1) \setminus\setminus \varphi_2 & \rightarrow & \pi_{fresh}\left(\rho_a^{fresh}(\varphi_1) - \varphi_2\right) \\
\varphi_1 \cup \pi_a(\varphi_2) & \rightarrow & \pi_{fresh}\left(\varphi_1 \cup \rho_a^{fresh}(\varphi_2)\right) \\
\varphi_1 \ltimes \pi_a(\varphi_2) & \rightarrow & \pi_{fresh}\left(\varphi_1 \ltimes \rho_a^{fresh}(\varphi_2)\right) \\
\varphi_1 \setminus \pi_a(\varphi_2) & \rightarrow & \pi_{fresh}\left(\varphi_1 \setminus \rho_a^{fresh}(\varphi_2)\right) \\
\varphi_1 \bowtie \pi_a(\varphi_2) & \rightarrow & \pi_{fresh}\left(\varphi_1 \bowtie \rho_a^{fresh}(\varphi_2)\right) \\
\sigma_f(\pi_a(\varphi)) & \rightarrow & \pi_a\left(\sigma_{f[a/fresh}(])\varphi\right) \\
\theta(\pi_a(\varphi), g : \mathcal{C} \rightarrow \mathcal{D}) & \rightarrow & \pi_{fresh}\left(\theta(\rho_a^{fresh}(\varphi), g : \mathcal{C} \rightarrow \mathcal{D})\right) \\
\text{let } (X = \pi_a(\varphi)) \text{ in } \psi & \rightarrow & \text{let } (X = \varphi) \text{ in } \psi[X/\pi_a(X)] \\
\text{let } (X = \varphi) \text{ in } \pi_a(\psi) & \rightarrow & \pi_a(\text{let } (X = \varphi) \text{ in } \psi)
\end{array}
$$

The only subtle rule is $\theta(\pi_a(\varphi), g : \mathcal{C} \rightarrow \mathcal{D}) \rightarrow \pi_{fresh}\left(\theta(\rho_a^{fresh}(\varphi), g : \mathcal{C} \rightarrow \mathcal{D})\right)$. When the column a does not appear as input or output of f then we can rewrite this rule to $\theta(\pi_a(\varphi), g : \mathcal{C} \rightarrow \mathcal{D}) \rightarrow \pi_a(\theta(\varphi, g : \mathcal{C} \rightarrow \mathcal{D}))$ but in the general case we have to be careful and rename a first.

After applying all those rules, the formula can only have column removal operations at the topmost part or in front of fixpoints (i.e. $\mu(X = \pi_{a_1}(\ldots \pi_{a_n}(\varphi))))$.

4.5.5 Commutative operators

We suppose there exists an arbitrary order $<$ on μ-algebra terms and for each commutative operator $+ \in \{ \bowtie, \cup\}$ we replace $A+B$ with $B+A$ when $A > B$. This considerably reduces the number of equivalent plans but it increases the number of rewriting rules.

For instance, for a n-join $\varphi_1 \bowtie \ldots \bowtie \varphi_n \bowtie \varphi_{n+1}$ there are $2^n \times C_n$ equivalent plans using commutativity and associativity ($2^n \times C_n$ is the number of binary trees over n internal nodes or $n+1$ different leaves where C_n is the n-th number of Catalan). If we leave out the commutativity using the order introduced to consider only plans where each join respects the order there are only C_n different plans.

However, this simplification implies to extend the set of rewriting rules, for instance, if we have the rule $\sigma_{filter}(A \bowtie B) \rightarrow \sigma_{filter}(A) \bowtie B$ (if B and f have no common columns) then we also need the symmetrical rule $\sigma_{filter}(B \bowtie A) \rightarrow \sigma_{filter}(A) \bowtie B$.

In the same manner, each time we have a sequence of filters $\sigma_{f_1}(\ldots \sigma_{f_n}(\varphi))$ we reorder the $f_1, \ldots, f_n$ such that $\sigma_{f_i}(\emptyset) < \sigma_{f_j}(\emptyset) \Leftrightarrow i < j$.

4.5.6 Expansion of let binders

Given a let-binder let $(X = \varphi)$ in ψ satisfying the conditions of lemma 4 (i.e. $\forall Y \in def(\psi)\ \ sim(\varphi, Y)$), the rule let $(X = \varphi)$ in $\psi \rightarrow \psi[X/\varphi]$ is valid.

To normalize let-binders we can use the rule to remove the let-binder, but that sometimes results in an exponential increase in the size of the formula. In our normalization process we apply the rule only when X is syntactically present once (or not present) in φ. In other cases, we push the formula at the top using the rules (supposing $X \notin def(\xi)$):

$$
\begin{array}{lcll}
(\text{let } (X = \varphi) \text{ in } \psi) \bowtie \xi & \rightarrow & \text{let } (X = \varphi) \text{ in } (\psi \bowtie \xi) & \\
(\text{let } (X = \varphi) \text{ in } \psi) ⟕ \xi & \rightarrow & \text{let } (X = \varphi) \text{ in } (\psi ⟕ \xi) & \\
\xi ⟖ (\text{let } (X = \varphi) \text{ in } \psi) & \rightarrow & \text{let } (X = \varphi) \text{ in } (\xi ⟖ \psi) & \\
\xi \setminus (\text{let } (X = \varphi) \text{ in } \psi) & \rightarrow & \text{let } (X = \varphi) \text{ in } (\xi \setminus \psi) & \\
(\text{let } (X = \varphi) \text{ in } \psi) \setminus \xi & \rightarrow & \text{let } (X = \varphi) \text{ in } (\psi \setminus \xi) & \\
\xi \setminus\setminus (\text{let } (X = \varphi) \text{ in } \psi) & \rightarrow & \text{let } (X = \varphi) \text{ in } (\xi \setminus\setminus \psi) & \\
(\text{let } (X = \varphi) \text{ in } \psi) \setminus\setminus \xi & \rightarrow & \text{let } (X = \varphi) \text{ in } (\psi \setminus\setminus \xi) & \\
\xi - (\text{let } (X = \varphi) \text{ in } \psi) & \rightarrow & \text{let } (X = \varphi) \text{ in } (\xi - \psi) & \\
(\text{let } (X = \varphi) \text{ in } \psi) - \xi & \rightarrow & \text{let } (X = \varphi) \text{ in } (\psi - \xi) & \\
(\text{let } (X = \varphi) \text{ in } \psi) \cup \xi & \rightarrow & \text{let } (X = \varphi) \text{ in } (\psi \cup \xi) & \\
\sigma_{filter}(\text{let } (X = \varphi) \text{ in } \psi) & \rightarrow & \text{let } (X = \varphi) \text{ in } \sigma_{filter}(\psi) & \\
\beta_a^b(\text{let } (X = \varphi) \text{ in } \psi) & \rightarrow & \text{let } (X = \varphi) \text{ in } \beta_a^b(\psi) & \\
\mu(Y = \text{let } (X = \varphi) \text{ in } \psi)) & \rightarrow & \text{let } (X = \varphi) \text{ in } \mu(Y = \psi) & \text{when } sim(\varphi, Y) = 0
\end{array}
$$

4.5.7 Filter

Our normalizing rules to simplify filters:

$$
\begin{array}{lcr}
\sigma_{a\&\&b}(\varphi) & \rightarrow & \sigma_a(\sigma_b(\varphi)) \\
\sigma_{\neg(a||b)}(\varphi) & \rightarrow & \sigma_{\neg a}(\sigma_{\neg b}(\varphi)) \\
\sigma_{\neg\neg a}(\varphi) & \rightarrow & \sigma_a(\varphi)
\end{array}
$$

The following rules also exist (note that we reversed the direction of the arrows to avoid combinatory explosion):

$$\begin{array}{lcr} \sigma_{\neg a}(\varphi) \cup (\sigma_{\neg b}(\varphi)) & \rightarrow & \sigma_{\neg(a\&\&b)}(\varphi) \\ \sigma_a(\varphi) \cup \sigma_b(\varphi) & \rightarrow & \sigma_{a\|b}(\varphi) \end{array}$$

4.5.8 Removing filters

Filters can sometimes be removed. For instance, in $t = \Big(\text{let } (X = N) \text{ in } \sigma_{bnd(o)}(X) \bowtie \rho_s^n(T_\emptyset)\Big)$, the $\sigma_{bnd(o)}(X)$ can be reduced to $\emptyset$ and then the whole t can also be reduced to $\emptyset$, but in $t' = \Big(\text{let } (X = N \cup \rho_n^i(N)) \text{ in } \sigma_{bnd(i)}(X) \bowtie \rho_s^n(T_\emptyset)\Big)$, $\sigma_{bnd(i)}(X)$ cannot be directly reduced.

Then, depending on ψ, ϵ, and whether c belongs or not to $C(\varphi)$ and $P(\varphi)$, we have the rules:

$c \notin P(\varphi)$	$c \in C(\varphi)$
$\sigma_{bnd(c)}(\varphi) \rightarrow \emptyset$	$\sigma_{bnd(c)}(\varphi) \rightarrow \varphi$
$\sigma_{\neg bnd(c)}(\varphi) \rightarrow \varphi$	$\sigma_{\neg bnd(c)}(\varphi) \rightarrow \emptyset$

Furthermore, in the expression $\sigma_f(\varphi)$ we can compute f' which is f where we replaced the occurrences of the elements of $FC(f) \setminus P(\varphi)$ by $Error$. If f' necessarily evaluates to $\top$ then we have the rule $\sigma_f(\varphi) \rightarrow \varphi$ and if f' necessarily evaluates to $\perp$ or E then we have the rule $\sigma_f(\varphi) \rightarrow \emptyset$.

4.5.9 Simplification rules

Finally we also have the following rules to simplify redundant terms (proof of their validity is rather straightforward and thus omitted):

$$\begin{array}{lcll}
\varphi \cup \varphi & \rightarrow & \varphi & \\
\varphi \bowtie \varphi & \rightarrow & \varphi & \text{when } |P(\varphi,\Gamma) \setminus C(\varphi,\Gamma)| \leq 1 \\
\varphi \ltimes \varphi & \rightarrow & \varphi & \text{when } |P(\varphi,\Gamma) \setminus C(\varphi,\Gamma)| \leq 1 \\
\varphi \setminus \varphi & \rightarrow & \emptyset & \\
\varphi \backslash\backslash \varphi & \rightarrow & \emptyset & \text{when } C(\varphi,\Gamma) \neq \emptyset \\
\varphi - \varphi & \rightarrow & \emptyset & \\
\varphi - \psi & \rightarrow & \varphi & \text{when } C(\varphi) \setminus P(\psi,\Gamma) \neq \emptyset \\
\varphi - \psi & \rightarrow & \varphi \setminus \psi & \text{when } P(\varphi,\Gamma) \setminus C(\varphi,\Gamma) = P(\psi,\Gamma) \setminus C(\psi,\Gamma) = \emptyset \\
\varphi \backslash\backslash \psi & \rightarrow & \varphi \setminus \psi & \text{when } C(\varphi) \cap C(\psi) \neq \emptyset \\
\varphi \bowtie \pi_{c_1}(\pi_{c_2}(\ldots \pi_{c_n}(\varphi))) & \rightarrow & \varphi & \text{when } P(\varphi,\Gamma) \setminus \{c_1,\ldots,c_n\} \subseteq C(\varphi,\Gamma) \\
\mu(X = \varphi) & \rightarrow & \varphi & \text{when } sim(\varphi, X) = 0 \\
\Theta(varphi, g, \mathcal{C}, \mathcal{D}) & \rightarrow & \emptyset & \text{when } \mathcal{C} \not\subseteq P(\varphi,\Gamma)
\end{array}$$

4.6 Producing rules

The rewriting rules we present here are used to produce new terms from a given term. The rule set we introduce needs to create a finite number of terms (in order for the algorithm to stop).

4.6.1 Associativity

Of the several binary operators we have, two of them are associative. These rules combine with the commutative rules which means that for a rule $(\varphi \bowtie \psi) \bowtie \xi$ we have also a rule for $(\psi \bowtie \varphi) \bowtie \xi$, $\xi \bowtie (\psi \bowtie \varphi)$ and $\xi \bowtie (\varphi \bowtie \psi)$. The associativity of the $\bowtie$ operator is of particular importance as it will lead to the several join orders of conjunctive queries.

$$\begin{array}{rcl} \varphi \bowtie (\psi \bowtie \xi) & \leftrightarrow & (\varphi \bowtie \psi) \bowtie \xi \\ \varphi \cup (\psi \cup \xi) & \leftrightarrow & (\varphi \cup \psi) \cup \xi \end{array}$$

4.6.2 Distributivity

Following the same idea as for associativity we can introduce the rule of distributivity from all operators to each other when applicable:

$$\begin{array}{lcrl}
\sigma_{a\|b}(\varphi) & \leftrightarrow & \sigma_a(\varphi) \cup \sigma_b(\varphi) & \\
\varphi \bowtie (\psi \cup \xi) & \leftrightarrow & (\varphi \bowtie \psi) \cup (\varphi \bowtie \xi) & \\
(\psi \cup \xi) \ltimes \varphi & \leftrightarrow & (\psi \ltimes \varphi) \cup (\xi \ltimes \varphi) & \\
(\psi \cup \xi) \setminus \varphi & \leftrightarrow & (\psi \setminus \varphi) \cup (\xi \setminus \varphi) & \\
(\psi \cup \xi) \setminus\setminus \varphi & \leftrightarrow & (\psi \setminus\setminus \varphi) \cup (\xi \setminus\setminus \varphi) & \\
(\psi \cup \xi) - \varphi & \leftrightarrow & (\psi - \varphi) \cup (\xi - \varphi) & \\
\sigma_f(\varphi \cup \psi) & \leftrightarrow & \sigma_f(\varphi) \cup \sigma_f(\psi) & \\
\sigma_f(\varphi \bowtie \psi) & \leftrightarrow & \sigma_f(\varphi) \bowtie \psi & FC(f) \subseteq C(\varphi) \\
\sigma_f(\varphi \bowtie \psi) & \leftrightarrow & \sigma_f(\varphi) \bowtie \sigma_f(\psi) & FC(f) \subseteq C(\varphi) \cap C(\psi) \\
\varphi \bowtie \beta_a^b(\psi) & \leftrightarrow & \beta_a^b(\varphi \bowtie \psi) & a \in C(\psi, \Gamma) \cap C(\varphi, \Gamma) \wedge b \notin P(\psi, \Gamma) \cup P(\varphi, \Gamma) \\
\varphi \bowtie \beta_a^b(\psi) & \leftrightarrow & \sigma_{a=b}(\varphi \bowtie \psi) & a \in C(\psi, \Gamma) \cap C(\varphi, \Gamma) \wedge b \notin P(\psi, \Gamma) \wedge b \in C(\varphi, \Gamma)
\end{array}$$

Since fixpoints are the new feature of our algebra, we will now present rules associated with fixpoints with more details.

4.6.3 Combining fixpoints and joins or filters

In our translation of SPARQL, given the term $PP(John\ knows*\ ?a)$ our algorithm will translate $knows*$ into a fixpoint computation and then filter to keep elements of the relation that have a $John$. Clearly, computing the whole binary relation $knows*$ before filtering is wasteful compared with computing directly the unary relation $knows*\ John$. (Note that in some cases $?a$ might be more constrained than $John$ and in this case our computation should start from $?a$; starting from the fixed part of a PP is not always the best plan.)

In order to generate the various plans, we need rules to move joins and filters inside fixpoints (i.e. $\sigma_f(\mu(X = A)) \rightarrow \mu(X = \sigma_f(A))$). Clearly, such a rule is not always true: given the μ-algebra term $A = \beta_s^o(N) \cup \pi_m(\rho_s^m(X) \bowtie \rho_o^m(T))$, we have $\sigma_{s=John}(\mu(X = A)) \rightarrow \mu(X = \sigma_{s=John}(A))$ but we don't have $\sigma_{o=John}(\mu(X = A)) \rightarrow$
$\mu(X = \sigma_{o=John}(A))$. This is caused by our fixpoint computation that uses the column o to produce new elements and replace it with the o from T.

We now present four rules on fixpoints to combine them with filters or join, supposing they are evaluated under an environment abstracted by Γ:

- Using the Theorem 2 the rule to combine filters with fixpoints is when $\forall c \in FC(f)\ \forall p \in perm(\varphi, X, C(\mu(X = \varphi), \Gamma))\ p(c) = c \wedge FC(f) \subseteq C(\varphi, \Gamma)$ we have that

$$\sigma_f(\mu(X = \varphi)) \rightarrow \mu(X = \sigma_f(\varphi))$$

- Using the Theorem 3 the rule to combine a join with fixpoints is when $sim(\psi, X) = 0$ and $\forall c \in P(\psi)\ \forall p \in perm(\varphi, X, C(\mu(X = \varphi), \Gamma))\quad p(c) = c$ and $\forall c \in P(\mu(X = \varphi), \Gamma) \setminus C(\mu(X = \varphi), \Gamma)\ canAdd(\varphi, X, c)$ we have:

$$\psi \bowtie \mu(X = \varphi) \rightarrow \mu(X = \psi \bowtie \varphi))$$

- Furthermore, when fixpoints can be decomposed into an initial part and a recursive part, we have a specialized version of the rules. If the fixpoint is $\mu(X = \varphi \cup \psi)$, where φ is constant in X and ψ is recursive, then (by Theorem 5) when $\forall p \in perm(\psi, X, C(\mu(X = \varphi \cup \psi), \Gamma))$ $FC(f)\ p(c) = c$ we have that:

$$\sigma_f(\mu(X = \varphi \cup \psi)) \rightarrow \mu(X = \sigma_f(\varphi) \cup \psi)$$

- In the same manner, if the fixpoint is $\xi = \mu(X = \varphi \cup \psi)$ where φ is constant in X and ψ is recursive, then (by Theorem 4) when $sim(\kappa, X) = 0$ and $C(\kappa, \Gamma) = P(\kappa, \Gamma)$ and for each $c \in P(\kappa, \Gamma)$ either $c \in C(\xi, \Gamma) \wedge \forall p \in perm(\psi, X, C(\xi, \Gamma))\ p(c) = c$ or $c \notin P(\xi, \Gamma) \wedge canAdd(\psi, X, c) = \top$ then we have:

$$\kappa \bowtie \mu(X = \varphi \cup \psi) \rightarrow \mu(X = (\kappa \bowtie \varphi) \cup \psi)$$

4.6.4 Reversing fixpoints

Motivation

As we have seen, pushing filters, selections and joins into a fixpoint $\mu(X = \varphi)$ depends on φ. The translation of the regular path expression $knows*$ is $rep(knows*) =$ $\mu(X = \beta_s^o(N) \cup \pi_m(\rho_o^m(X) \bowtie \rho_s^m(\sigma_{p=knows}(T))))$. In this fixpoint, we cannot push the filter $\sigma_{o=John}$ since o is used by X. This fixpoint starts with a set of couples $\beta_s^o(N)$ binding s and o, and at each iteration of the fixpoint, it will build new (s, o) by appending a mapping (m, o) validating $knows$ to a mapping (s, m) already built. However, to compute the same set of mappings, the fixpoint could have started from the same set and recursively appended a mapping (s, m) validating $knows$ to an already-built mapping (m, o) without using o, i.e. the fixpoint $\mu(X = \beta_s^o(N) \cup \pi_m(\rho_s^m(X) \bowtie \rho_o^m(\sigma_{p=knows}(T))))$ in which we can push the filter $\sigma_{o=john}$ which gives $\mu(X = \beta_s^o(\sigma_{s=john}(N)) \cup \pi_m(\rho_s^m(X) \bowtie \rho_o^m(\sigma_{p=knows}(T))))$.

Simple case

Let us consider a fixpoint of the form $\mu(X = \varphi \cup \pi_c(\rho_a^c(\varphi) \bowtie \rho_b^c(X)))$ and let us suppose that $P(\varphi, \Gamma) = C(\varphi, \Gamma) = \{a, b\}$ and a, b and c are all distinct. φ represents a binary relation from a to b and the fixpoint computes its transitive closure. Therefore the role of a and b are symmetrical and this fixpoint computes the same relation as $\mu(X = \varphi \cup \pi_c(\rho_b^c(\varphi) \bowtie \rho_a^c(X)))$.

Simple case extended to n columns

Let us now consider a fixpoint of the form

$$\mu\big(X = \varphi \cup \pi_{c_1}\left(\ldots \pi_{c_n}\left(\rho_{a_1}^{c_1}\left(\ldots \rho_{a_n}^{c_n}(\varphi)\right) \bowtie \rho_{b_1}^{c_1}\left(\ldots \rho_{b_n}^{c_n}(X)\right)\right)\right)\big)$$

and let us suppose that $P(\varphi,\Gamma) = C(\varphi,\Gamma) = \{a_1,\ldots,a_n,b_1,\ldots,b_n\}$ with the (a_i), (b_i) and (c_1) all distinct. φ represents a binary relation from the n-uplet $a_1,\ldots,a_n$ to the n-uplet $b_1,\ldots,b_n$ and the fixpoint computes its transitive closure. Similarly as in the simple case the role of (a_i) and (b_i) are symmetrical and this fixpoint computes the same set as $\mu\big(X = \varphi \cup \pi_{c_1}\left(\ldots \pi_{c_n}\left(\rho_{b_1}^{c_1}\left(\ldots \rho_{b_n}^{c_n}(\varphi)\right) \bowtie \rho_{a_1}^{c_1}\left(\ldots \rho_{a_n}^{c_n}(X)\right)\right)\right)\big)$.

Via an unfolding of a fixpoint over one column

Given a fixpoint expression of the form $\mu\big(X = \psi \cup \pi_b\left(\rho_a^b(X) \bowtie \varphi\right)\big)$ with $P(\varphi,\Gamma) = C(\varphi,\Gamma) = \{a,b\}$, $P(\psi,\Gamma) = C(\psi,\Gamma) = \{a,d\}$ and a,b,d all different.

This fixpoint actually computes the reflexive transitive closure of the relation between a and b induced by φ and then joins the relation between a and b induced by ψ which gives an overall relation between a and d.

From a relation point of view, we can replace the reflexive transitive closure with the union of the transitive closure plus the identity relation. With a fresh c (i.e. $c \notin \{a,b,d\}$) we have:

$\mu\big(X = \psi \cup \pi_b\left(\rho_a^b(X) \bowtie \varphi\right)\big) = \psi \cup \pi_b\left(\rho_a^b(\psi) \bowtie \mu(X = \varphi \cup \pi_c\left(\rho_a^c(X) \bowtie \rho_b^c(\varphi)\right))\right)$

or using the reversion presented earlier:

$\mu\big(X = \psi \cup \pi_b\left(\rho_a^b(X) \bowtie \varphi\right)\big) = \psi \cup \pi_b\left(\rho_a^b(\psi) \bowtie \mu(X = \varphi \cup \pi_c\left(\rho_b^c(X) \bowtie \rho_a^c(\varphi)\right))\right)$

Via the unfolding of a fixpoint over several columns

Just as before but a is $a_1,\ldots,a_n$, b is $b_1,\ldots,b_n$ and d is $d_1,\ldots,d_k$ (note that we do not need to have $k = n$). Given a fixpoint expression of the form $\mu\big(X = \psi \cup \pi_{b_1}\left(\ldots \pi_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\big)$ with $P(\varphi,\Gamma) = C(\varphi,\Gamma) = \{a_1,\ldots,c_n,b_1,\ldots,b_n\}$, $P(\psi,\Gamma) = C(\psi,\Gamma) = \{a_1,\ldots,a_n,d_1,\ldots,d_k\}$ and a_i,b_j,d_l all different.

We have $\mu\big(X = \psi \cup \pi_{b_1}\left(\ldots \pi_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\big) =$

$$\psi \cup \pi_{b_1}\left(\ldots \pi_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(\psi)\right) \bowtie \mu\big(X = \varphi \cup \pi_{c_1}\left(\ldots \pi_{c_n}\left(\rho_{a_1}^{c_1}\left(\ldots \rho_{a_n}^{c_n}(X)\right) \bowtie \rho_{b_1}^{c_1}\left(\ldots \rho_{b_n}^{c_n}(\varphi)\right)\right)\right)\right)\right)$$

or using the reversion presented earlier:

$\mu\big(X = \psi \cup \pi_{b_1}\left(\ldots \pi_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\big) =$

$$\psi \cup \pi_{b_1}\left(\ldots \pi_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(\psi)\right) \bowtie \mu\big(X = \varphi \cup \pi_{c_1}\left(\ldots \pi_{c_n}\left(\rho_{b_1}^{c_1}\left(\ldots \rho_{b_n}^{c_n}(X)\right) \bowtie \rho_{a_1}^{c_1}\left(\ldots \rho_{a_n}^{c_n}(\varphi)\right)\right)\right)\right)\right)$$

4.6.5 Combine fixpoints

Given the Regular Path Expression $locatedIn*/sameAs*$ its translation will imply two fixpoints: one to compute $locatedIn*$ and one to compute $sameAs*$. Each of these fixpoints might have a very large number of solutions and joining them will be costly, it might have been more efficient to compute both in a single fixpoint: $\mu(X = \beta_s^o(N) \cup \pi_m(\rho_s^m(X) \bowtie loc \cup \rho_o^m(X) \bowtie sam))$ where $loc = \rho_o^m(\sigma_{p=locatedIn}(T))$ and $sam = \rho_s^m(\sigma_{p=sameAs}(T))$.

In general, if we have the join of two decomposed fixpoints operating on two separate domains then they can be joined using Theorem 6. Given $\phi_1 = \mu(X = \varphi \cup \psi)$ and $\phi_2 = \mu(Y = \kappa \cup \xi)$ such that

- κ, ξ and φ are constant in X and ψ recursive in X
- κ, φ and ψ are constant in Y and ξ recursive in Y
- $C(\phi_1, \Gamma) = P(\phi_1, \Gamma)$
- $C(\phi_2, \Gamma) = P(\phi_2, \Gamma)$
- for each $c \in P(\phi_1, \Gamma)$ either $c \in P(\phi_2, \Gamma) \wedge \forall p \in perm(\psi, X, C(\phi_1, \Gamma))\ p(c) = c$ or $c \notin P(\phi_2, \Gamma) \wedge canAdd(\xi, X, c)$
- for each $c \in P(\phi_2, \Gamma)$ either $c \in P(\phi_1, \Gamma) \wedge \forall p \in perm(\xi, X, C(\phi_2, \Gamma))\ p(c) = c$ or $c \notin P(\phi_1, \Gamma) \wedge canAdd(\psi, X, c)$

Then we have

$$\mu(X = \varphi \cup \psi) \bowtie \mu(Y = \xi \cup \kappa) \rightarrow \mu(X = \varphi \bowtie \kappa \cup \psi \cup \text{let } (Y = X) \text{ in } \xi)$$

Or

$$\mu(X = \varphi \cup \psi) \bowtie \mu(Y = \xi \cup \kappa) \rightarrow \mu(X = \varphi \bowtie \kappa \cup \psi \cup \xi')$$

where ξ' is ξ where replaced the free occurrences of Y with X.

4.7 Ad-hoc rules

When relying on a specific model like graphs for SPARQL we can add some normalizing ad-hoc rules for rewritings that could be found by our algorithm but would speed up the process or simply would not be found by our rewriting process.

4.7.1 For SPARQL : remove N

Computing N, the whole set of nodes, is often very costly in practice and can often be avoided. As a heuristic it is generally a good idea to remove a N when possible. If we have a term $\sigma_{s=v}(N)$ then it returns $|s \rightarrow v|$ or $\emptyset$. If we are able at compile time to decide in which case we are, we can simplify this. When we have a term $N \bowtie \varphi$ and $s \in C(\varphi)$, if we are sure that, given a mapping m solution of φ, $m(s)$ is a node, then we can rewrite this term to φ.

This rule is equivalent to an application of several of our rules:

$$\begin{aligned}
\rho_p^{p'}\left(\rho_o^{o'}(T)\right) \bowtie N \rightarrow\ & \rho_p^{p'}\left(\rho_o^{o'}(T)\right) \bowtie \left(\rho_s^{o}\left(\pi_p\left(\pi_s(T)\right)\right) \cup \pi_p\left(\pi_o(T)\right)\right) \\
\rightarrow\ & \left(\rho_p^{p'}\left(\rho_o^{o'}(T)\right) \bowtie \rho_s^{o}\left(\pi_p\left(\pi_s(T)\right)\right)\right) \cup \left(\rho_p^{p'}\left(\rho_o^{o'}(T)\right) \bowtie \pi_p\left(\pi_o(T)\right)\right) \\
\rightarrow\ & \left(\rho_p^{p'}\left(\rho_o^{o'}(T)\right) \bowtie \rho_s^{o'}\left(\pi_{p'}\left(\pi_s\left(\rho_p^{p'}\left(\rho_o^{o'}(T)\right)\right)\right)\right)\right) \cup \\
& \left(\rho_p^{p'}\left(\rho_o^{o'}(T)\right) \bowtie \pi_{p'}\left(\pi_{o'}\left(\rho_p^{p'}\left(\rho_o^{o'}(T)\right)\right)\right)\right) \\
\rightarrow\ & \rho_p^{p'}\left(\rho_o^{o'}(T)\right) \cup \rho_p^{p'}\left(\rho_o^{o'}(T)\right) \rightarrow \rho_p^{p'}\left(\rho_o^{o'}(T)\right)
\end{aligned}$$

4.8 Rewriting algorithm

In this section we present our full algorithm to generate equivalent terms to a given μ-algebra term.

4.8.1 Computing properties of μ-algebra terms

As we have seen, rewriting rules depend on several properties: the type environment ($P(\bullet, \Gamma)$ and $C(\bullet, \Gamma)$) but also which variables appear (def), whether the term is constant, linear or recursive (sim and rec) in the various variables, the set of permutations ($perm$). Except for P and C, by using a hash function, all of these operators can be computed very quickly. It is not strictly linear since for terms of the form let $(Y = \varphi)$ in ψ we will need three recursive calls to sim but for all the different X, $sim(\text{let } (Y = \varphi) \text{ in } \psi, X)$ will depend on the result of the same $sim(\psi, Y)$. Our algorithm will produce a lot of terms on which we will compute those functions but these terms are obtained through rewriting rules which means that many subformulae will be shared between terms.

4.8.2 Main algorithm

The main algorithm consists in steps and each step producing new terms from the terms of the preceding step. For each term created at the last step we first produce all term accessible through a rewriting rule applied on a subformula, then we normalize those terms and finally we check whether this normalized term was already created. This algorithm is depicted in algorithms 3 & 2.

From a high level point of view algorithm 2 produces the set of terms accessible with one rewriting rule applied on the term (either at the top or on a subformula). Therefore if we consider the graph whose nodes are μ-algebra terms and vertices are the rewritings, then algorithm 3 is simply a BFS. Any other algorithm to discover nodes could be used. In particular if the number of explored nodes is too big we could use an $A*$ (the distance would be the estimated computational cost of a term) to reduce the search space (but there is no reason for this $A*$ to give the best cost) or limit the BFS to a fixed number of steps.

Algorithm 1 Produce all equivalent by applying one rewriting rule on the top term

```
function applyRules(φ,Γ)
    res ← ∅
    for rule ∈ ruleset do
        if isApplicable(rule, φ, Γ) then
            res ← res ∪ applyRule(rule, φ)
    end for
end function
```

Algorithm 2 Recursively producing terms by applying one rewriting rule on a subterm of the given term

function EXPLORETERM(φ,Γ)
  switch φ **do**
    case $\psi \cup \xi$
      $sub \leftarrow \{\psi' \cup \xi \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\} \cup \{\psi \cup \xi' \mid \xi' \in \text{EXPLORETERM}(\xi, \Gamma)\}$
    case $\psi \backslash\backslash \xi$
      $sub \leftarrow \{\psi' \backslash\backslash \xi \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\} \cup \{\psi \backslash\backslash \xi' \mid \xi' \in \text{EXPLORETERM}(\xi, \Gamma)\}$
    case $\psi - \xi$
      $sub \leftarrow \{\psi' - \xi \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\} \cup \{\psi - \xi' \mid \xi' \in \text{EXPLORETERM}(\xi, \Gamma)\}$
    case $\psi \setminus \xi$
      $sub \leftarrow \{\psi' \setminus \xi \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\} \cup \{\psi \setminus \xi' \mid \xi' \in \text{EXPLORETERM}(\xi, \Gamma)\}$
    case $\psi \bowtie \xi$
      $sub \leftarrow \{\psi' \bowtie \xi \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\} \cup \{\psi \bowtie \xi' \mid \xi' \in \text{EXPLORETERM}(\xi, \Gamma)\}$
    case $\psi \bowtie\!\!\!\!\!\!\times\; \xi$
      $sub \leftarrow \{\psi' \bowtie\!\!\!\!\!\!\times\; \xi \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\} \cup \{\psi \bowtie\!\!\!\!\!\!\times\; \xi' \mid \xi' \in \text{EXPLORETERM}(\xi, \Gamma)\}$
    case $\rho_a^b(\psi)$
      $sub \leftarrow \{\rho_a^b(\psi)' \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\}$
    case $\pi_a(\psi)$
      $sub \leftarrow \{\pi_a(\psi)' \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\}$
    case $\beta_a^b(\psi)$
      $sub \leftarrow \{\beta_a^b(\psi)' \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\}$
    case $\theta(\psi, g : \mathcal{C} \to \mathcal{D})$
      $sub \leftarrow \{\theta(\psi', g : \mathcal{C} \to \mathcal{D}) \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\}$
    case $\Theta(\psi, g, \mathcal{C}, \mathcal{D})$
      $sub \leftarrow \{\Theta(\psi', g, \mathcal{C}, \mathcal{D}) \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\}$
    case $\sigma_f(\psi)$
      $sub \leftarrow \{\sigma_f(\psi') \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma)\}$
    case $\mu(X = \psi)$
      $sub \leftarrow \{\mu(X = \psi') \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma[X/(C(\mu(X = \psi), \Gamma), P(\mu(X = \psi), \Gamma))])\}$
    case let $(X = \xi)$ in ψ
      $sub \leftarrow \{\text{let } (X = \xi) \text{ in } \psi' \mid \psi' \in \text{EXPLORETERM}(\psi, \Gamma[X/(C(\xi, \Gamma), P(\xi, \Gamma))])\} \cup$
      $sub \leftarrow \{\text{let } (X = \xi') \text{ in } \psi \mid \xi \in \text{EXPLORETERM}(\xi, \Gamma)\}$
    case X
      $sub \leftarrow \emptyset$
    case $\emptyset$
      $sub \leftarrow \emptyset$
    case $|c_1 \to v_1, \ldots, c_n \to v_n|$
      $sub \leftarrow \emptyset$
  return $sub \cup \text{APPLYRULES}(\varphi, \Gamma)$
end function

Algorithm 3 Producing all terms with rewriting rules from a given term

```
function GenerateAllEquivalent(term,Γ)
    new_terms ← {Normalize(φ)}
    seen ← new_term
    while new_terms ≠ ∅ do
        discovered ← ∅
        for φ ∈ new_terms do
            for ψ ∈ exploreTerm(φ, Γ) do
                ξ ← Normalize(ψ)
                if ξ ∉ seen then
                    seen ← seen ∪ ξ
                    discovered ← discovered ∪ ξ
            end for
        end for
        new_terms ← discovered
    end while
end function
```

4.9 Example of rewriting

The SPARQL query below that select the pair of $?a$ and $?b$ such that $?a$'s firstname is John, $?b$'s lastname is Doe and $?a, ?b$ is linked with a path composed only of edges labeled `:knows` :

```
SELECT * WHERE {
?a (knows)* ?b .
?a firstname John .
?b lastname Doe .
}
```

Using our rewrite rules we obtain multiples terms that are presented in the table below:

- Term 1 is obtained by the the translation of the SPARQL query to μ-algebra term.
- Term 2 is obtained through the normalization of 1. by pushing the renamings so that they appear only in front of variables.
- Term 3 is obtained from term 2 by combining the fixpoint and the translation of $PP(?a$ `:lastname :Doe`$)$ and then normalizing to simplify $N \bowtie rpe(?a$ `:Doe` $?b)$ into $rpe(?a$ `:Doe` $?b)$.
- In term 2, we cannot combine the fixpoint and the translation of $PP(?b$ `:firstname :John`$)$ since $?b$ is not stable by all the elements of $perm(\varphi, X, \{?a, ?b\})$ (where φ is the fixpoint). We can however reverse the fixpoint in term 2, which gives us the term 4.
- In term 2 we can combine the fixpoint with the constraint on $?a$ but not with the constraint on $?b$. In its reversed, term 4, we can combine the fixpoint with the constraint on $?b$ but not with the constraint on $?a$. After a normalization, the combination between the fixpoint and the constraint on $?b$ gives us term 5.

- In all the terms we have presented, we use the associativity and commutativity of $\bowtie$ to reduce the number of terms presented. In most query evaluators the commutativity is not used ($A \bowtie B$ is executed just like $B \bowtie A$) which is why we use the commutativity to normalize our terms but the associativity is often taken into consideration when choosing a query plan. Indeed $(A \bowtie B) \bowtie C$ corresponds to execute first $A \bowtie B$ then joining with C which might be much more efficient then performing $(A \bowtie C) \bowtie B$. Each of the terms 2 and 4 actually corresponds to three different terms when taking the associativity into account. We note them 2^a, 2^b, 2^{ab}, 4^a, 4^b and 4^{ab} depending on which PP is joined last with the others (a is the constraint on $?a$, b is the constraint on $?b$ and ab the path constraint on both).

We have not yet defined exactly how μ-algebra terms will be executed. However, if we suppose a bottom-up approach, the different but equivalent terms above correspond to the following ways of computing the solutions:

(2^a) corresponds to evaluate each PP individually and then merge the partial results by starting with joining the recursive PP with the constraint on $?b$.

(2^b) corresponds to evaluate each PP individually and then merge the partial results by starting with joining the recursive PP with the constraint on $?a$.

(2^{ab}) corresponds to evaluate each PP individually and then merge the partial results by starting with performing the cartesian product of the two non recursive PP and then join with the recursive PP.

(3) starts by evaluating $?a$ `:lastname :Doe`, then, using a fixpoint, it builds the set of mappings verifying the two PP and, finally, joins with the mappings verifying the third PP.

(4^*) corresponds to (2^*) but where, to build the solution of the PP ($?a$ `:knows`$*$ $?b$), instead of starting from $?a$ and add new $?b$, we start from $?b$ and add new $?a$.

(5) starts from the solution of PP($?b$ `:firstname :John`) and then builds recursively the solution of this PP and the PP $?a$ `:knows`* $?b$. Finally it joins this result with the solution of PP($?a$ `:lastname :Doe`).

It seems clear that, among the possible μ-algebra terms, some seems to correspond to better plans than other. For instance building the solutions of the PP $?a$`:knows`*$?b$ without using the fact that $?a$ and $?b$ are constrained by the other PP implies that we might build a set of solutions quadratic in the number of `:knows` relations to keep only a few (we can have that most $?a$ do not have *John* as firstname and *Doe* as lastname). And yet deciding which μ-algebra terms is the most efficient (e.g. in terms of time to compute the solution) depends on the query executor which is the topic of the next chapter.

Furthermore without knowledge about the queried data it is impossible decide whereas to start from the $?a$ validating the first PP and use the fixpoint to build the solution for $?b$ (such as in (3)) or to start from $?b$. This question will be treated in chapter 6.

1		$(\rho_s^{?a}(\rho_o^b(\mu(X = \beta_s^o(N) \cup \pi_m(\rho_o^m(X) \bowtie \rho_s^m(\pi_p(\sigma_{p=\texttt{:knows}}(T))))))))$
	$\bowtie$	$(\rho_s^{?a}(\pi_o(\sigma_{o=\texttt{:Doe}}(\pi_p(\sigma_{p=\texttt{:lastname}}(T))))))$
	$\bowtie$	$(\rho_s^{?b}(\pi_o(\sigma_{o=\texttt{:John}}(\pi_p(\sigma_{p=\texttt{:firstname}}(T))))))$
2		$(\mu(X = \beta_{?a}^{?b}(\rho_s^{?a}(N)) \cup \pi_m(\rho_{?b}^m(X) \bowtie \pi_p(\sigma_{p=\texttt{:knows}}(\rho_s^m(\rho_o^{?b}(T)))))))$
	$\bowtie$	$(\pi_o(\sigma_{o=\texttt{:Doe}}(\pi_p(\sigma_{p=\texttt{:lastname}}(\rho_s^{?a}(T))))))$
	$\bowtie$	$(\pi_o(\sigma_{o=\texttt{:John}}(\pi_p(\sigma_{p=\texttt{:firstname}}(\rho_s^{?b}(T))))))$
3		$(\pi_o(\sigma_{o=\texttt{:John}}(\pi_p(\sigma_{p=\texttt{:firstname}}(\rho_s^{?b}(T))))))$
	$\bowtie$	$(\mu(X = \beta_{?a}^{?b}(\pi_o(\sigma_{o=\texttt{:Doe}}(\pi_p(\sigma_{p=\texttt{:lastname}}(\rho_s^{?a}(T)))))) \cup \pi_m(\rho_{?b}^m(X) \bowtie \pi_p(\sigma_{p=\texttt{:knows}}(\rho_s^m(\rho_o^{?b}(T)))))))$
4		$(\pi_o(\sigma_{o=\texttt{:John}}(\pi_p(\sigma_{p=\texttt{:firstname}}(\rho_s^{?b}(T))))))$
	$\bowtie$	$(\pi_o(\sigma_{o=\texttt{:Doe}}(\pi_p(\sigma_{p=\texttt{:lastname}}(\rho_s^{?a}(T))))))$
	$\bowtie$	$(\mu(X = \beta_{?a}^{?b}(\rho_s^{?a}(N)) \cup \pi_m(\rho_{?a}^m(X) \bowtie \pi_p(\sigma_{p=\texttt{:knows}}(\rho_s^{?a}(\rho_o^m(T)))))))$
5		$(\pi_o(\sigma_{o=\texttt{:Doe}}(\pi_p(\sigma_{p=\texttt{:lastname}}(\rho_s^{?a}(T))))))$
	$\bowtie$	$(\mu(X = \beta_{?b}^{?a}(\pi_o(\sigma_{o=\texttt{:John}}(\pi_p(\sigma_{p=\texttt{:firstname}}(\rho_s^{?b}(T)))))) \cup \pi_m(\rho_{?a}^m(X) \bowtie \pi_p(\sigma_{p=\texttt{:knows}}(\rho_s^{?a}(\rho_o^m(T)))))))$

Conclusion

In this chapter, after some motivating examples, we have introduced definitions, lemmas and theorems laying the ground for new rewrite rules. We then equipped our μ-algebra with a typing system capturing the shape of the domain of the solutions for a μ-algebra term. The chapter continued with the presentation of our rewrite rules that we decomposed into "normalizing" and "producing" rules. The "normalizing" rules allow us to reduce the search space for terms will the later allow us to discover new terms. Finally, we presented our rewrite algorithm and an example of a term rewritten.

Now that our method can produce numerous equivalent terms, there are two natural questions: how can one evaluate those terms? how to select the most efficient term to be evaluated? These two questions are inherently linked as the efficiency is relative to the evaluation method. We will therefore tackle both questions in the next chapter.

CHAPTER 5

Evaluation of μ-algebra terms

In the last two chapters, we have seen how to transform a SPARQL *query into a μ-algebra term and then how to rewrite a μ-algebra term to obtain multiple terms who are semantically equivalent. The next natural question therefore is: "how to select one of those terms to be evaluated?" which in turn raises the question: "how to evaluate terms?".*

This chapter treats the problem of the evaluation and of the efficient evaluation of a given μ-algebra terms. First we will present a general scheme to evaluate μ-algebra terms using "streams" and "typed streams".

Once we have studied the evaluation of μ-algebra terms, we will be able to extract a cost model for the evaluation time of a given term. In this view, the multiple equivalent terms can be considered as possible query execution plans. As there can be several execution backends, that have different evaluation time, we will refine our cost model using cost rules.

Equipped with an evaluator and a cost model, we can select the best query execution plan (i.e. the best

term as estimated by our cost model) and execute it. This gives us an optimizing query evaluator that we will compare to state of the art SPARQL *query evaluators on queries containing Property Paths in chapter 7.*

We can now assemble the various pieces presented in the last chapters and build SPARQL *evaluator: we translate* SPARQL *queries into the μ-algebra, then we generate several plans, we heuristically select one and evaluate it. We implemented two evaluators based on this method. One is a distributed* SPARQL *query evaluator restricted to a relatively small fragment of* SPARQL *while the second is a single-core evaluator that supports a large fragment of* SPARQL.

5.1 General bottom-up evaluation for μ-algebra terms

In this section we present our general method for a bottom-up evaluation of μ-algebra terms. By *bottom-up* evaluation, we indicate any "*naive*" evaluation recursively following the μ-algebra semantics. It thus designates any evaluation where each subterm is evaluated and thus where all intermediate results are computed. For instance if we have a term $\varphi \bowtie (\psi \cup \kappa)$, a bottom up evaluation will compute the solution to the whole term but also to each of the φ, ψ, κ and to $\psi \cup \kappa$.

As demonstrated in the previous chapter, given a μ-algebra there often are multiple equivalent terms (in the sense of computing the same solutions). There are, therefore, often many different ways to actually compute a given term each subterm has a different bottom-up evaluation and there might even be ways to evaluate terms that do not correspond to the bottom up evaluation of any of its rewritten form.

5.1.1 Stream

Streams are one-way communication channels between a *sender* and a *receiver*. The sender can *send* a message (here messages are mappings) or it can *end* the communication (in which case no more messages can come out of the sender).

In our centralized evaluation prototype, a stream takes the form of a pair of functions $(send, end)$. The function *send* takes a mapping and sends it to the receiver while the function *end* takes no argument and indicates that the communication is now finished, both functions have no return value.

In a distributed evaluation, a stream can take the form of one-to-one queues of messages with a special message to indicate that the queue is closed (but the message in the queue still needs to be treated).

The order of messages is not important for our compilation. Messages can be reordered in a stream as long as the end message is processed after all other messages in the same stream. Furthermore we will need for the termination of fixpoints an operation that wait for all queues to be empty.

The stream mechanism that we presented corresponds to a Dataflow architecture. The main differences with other existing frameworks is that our streams control their own termination with the "end" message. All we will see, to trigger an end message we sometimes need the computing blocks to be aware of the state of this flow in the program. This is generally not permitted by Dataflow frameworks as it is costly. However our programs do not actually need the exact state but simply an over-approximation determining whether a block is active or not and this information can be maintained easily even in a distributed setup.

5.1.2 General compilation

In our compiler, μ-algebra terms are recursively compiled using *streams*. Each term is compiled to a block that has exactly one output stream (that will carry the set of solutions for this term) and it has one input stream for each variable appearing as subformulas plus one special input *start* to bootstrap the evaluation (*start* only carry the bootstrap information).

We compile the terms recursively via the function $compile(\varphi, out)$ that takes a stream and an output stream and returns a pair, the start stream and a list of pair of variable and

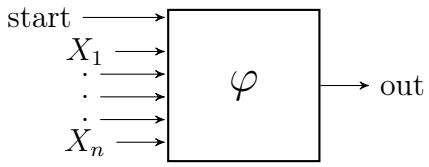


Figure 5.1: Graphical representation of a compiled term φ

input stream attached to that variable. We now present the intuition of what compiled terms do for each operator in our language:

- the compilation of $\emptyset$ is simple: it ends as it starts, does not use any variable or sends mappings;
- the compilation of $|c_1 \to v_1, \ldots, c_n \to v_n|$ is also simple: as it starts, it sends the mapping $\{c_1 \to v_1, \ldots, c_n \to v_n\}$ and ends;
- for a variable X, the compilation is to plug the out stream to the variable X;
- a filter $\sigma_f(\varphi)$ ends when φ ends and otherwise it sends mappings solution of φ passing the filter test;
- drops $(\pi_a(\varphi))$, renames $(\rho_a^b(\varphi))$, column multiply $(\beta_a^b(\varphi))$ and user-defined functions $(\theta(\varphi, g : \mathcal{C} \to \mathcal{D}))$ are compiled in a pretty similar way: the evaluation ends when φ ends, it has the same variables as φ and modifies the mappings it receives before sending them;
- the evaluation of a reduce operation $\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$ works in the following way: it starts the evaluation of φ when it starts and sends mappings only when φ ends. When a mapping m is received from φ, the evaluator decomposes m into $m_{\mathcal{C}}$ and $m_{\bar{\mathcal{C}}}$ such that $m = m_{\mathcal{C}} + m_{\bar{\mathcal{C}}}$ and $dom(m_{\mathcal{C}}) \subseteq \mathcal{C}$, $dom(m_{\bar{\mathcal{C}}}) \cap \mathcal{C} = \emptyset$. When $dom(m_{\mathcal{C}}) = \mathcal{C}$, the mapping $m_{\mathcal{C}}$ is then stored in the hash table using the key $m_{\bar{\mathcal{C}}}$. Once the execution of φ has finished we iterate through the hash table, for each key k associated with the mappings $c_1, \ldots, c_l$ we send $k + g(c_1, \ldots, c_l)$. Once this iteration is finished, the evaluation of $\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$ ends. Note that the evaluation of $\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$ is *blocking*, i.e. a mappings solution of $\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$ will be sent only when φ ends. That is why we need φ to be constant in all recursive variables.

  If we possess additional information about g we can produce an even more efficient code. For instance if g simply performs an addition then we can replace the part where we store $m_{\bar{\mathcal{C}}}$ by incrementing the value associated with the key $m_{\bar{\mathcal{C}}}$. As we explained in the introduction of μ-algebra, the $\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$ corresponds to a *reduce* operation on the solution of φ. With this transformation, the operation we are performing is actually a *combine-Reduce* operation.
- the union $\varphi_1 \cup \varphi_2$ ends when both of them have ended and it sends mappings from either of them;
- the compilation of the various minuses $\varphi_1 \ \lambda \ \varphi_2$ (for $\lambda \in \{\backslash, \backslash\backslash, -\}$) are similar to each others: φ_1 will compute a set of mappings and some of them will be discarded by mappings in s_2.

When the evaluation of $\varphi_1 \ \lambda \ \varphi_2$ starts, we create two sets s_1 and s_2, each s_i will contains some of the mappings solutions of φ_i. When a mapping from φ_1 arrives we check whether it is discarded by a mapping in s_2, if it is not them either we store it in s_1 when φ_2 has not finished its computation and otherwise we send it. When a mapping from φ_2 arrives, we remove the mappings from s_1 that are discarded by this mapping and then if φ_1 has not finished its computation we store it in s_2. When φ_2 finishes we empty s_1 by sending the mappings it contains. When φ_1 finishes, we just empty s_2. When both φ_1 and φ_2 are finished we end the computation of φ. As for $\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$, the minuses are *blocking* in φ_2, thus the need for φ_2 to be constant in all recursive variables.

- the evaluation of a join $\varphi = \varphi_1 \bowtie \varphi_2$ is done in the following way: when the evaluation starts, we create two sets s_1 and s_2. When a mapping m arrives from φ_i then for each compatible mappings $m' \in s_{\bar{i}}$ (with $\bar{2} = 1$ and $\bar{1} = 2$) we send $m + m'$ and if $\varphi_{\bar{i}}$ is not finished we store m in s_i.

  When the evaluation of φ_i finishes we empty $s_{\bar{i}}$ and when both are finished we end the evaluation of φ.

- the evaluation of a left join is somewhat similar but we keep markers to indicate which mappings of s_1 (i.e. sent by φ_1) have been matched: we create s_1 and s_2 as the evaluation starts, when a mapping m arrives from φ_i we add $m + m'$ for all compatible mappings m'; m or m' (depending on which comes from φ_1) is also marked as matched. Then we proceed to store m into s_i unless $\varphi_{\bar{i}}$ has finished. If $\varphi_{\bar{i}}$ has finished, $i = 1$ and m has not been matched with element of s_2 then we either send m.

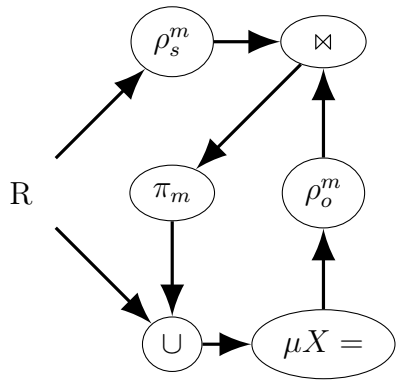
  When φ_2 finishes we empty s_1 and send element of s_1 that are marked as unmatched. When φ_1 finishes, we empty s_2. When both are finished we end the evaluation of φ;

- for the evaluation of a let binder let $(X = \varphi)$ in ψ we need to start φ and ψ as let $(X = \varphi)$ in ψ starts. For each mapping m sent by φ we need to send m to every use of X in ψ and when φ ends, we end each use of X in ψ. The output of ψ is plugged in the output of let $(X = \varphi)$ in ψ;

- finally for the evaluation of $\mu(X = \varphi)$ we start the evaluation of φ as $\mu(X = \varphi)$ starts and we also create a set s of mapping already seen. For each mapping m sent by φ we check that $m \notin s$ and if it is the case, we add m to s, we send m to the output and then we send m back into the uses of X inside φ.

  $\mu(X = \varphi)$ ends when φ ends but the end of φ usually depends on the end of the recursive uses of X. We end X when ending the recursive uses of X would activate the end signal on φ. This can be computed efficiently by keeping track of which nodes are active, when all the nodes upon which φ depend have ended or are waiting on the recursive use of X then $\mu(X = \varphi)$ can send the signal that X has ended and ends itself.

An example Let us consider the term $\mu(X = R \cup \pi_m(\rho_o^m(X) \bowtie))$ that computes the transitive closure of a relation R binding s and o. The graph corresponding to the compilation of that term is depicted below. Note that R is depicted here but as R is a free variable it is just an incoming stream.



5.1.3 Typed streams

The mappings we manipulate in streams are solutions of terms. Since we are able to type terms we are also able to compute for each stream s the sets $C(s), P(s)$ such that all the mapping circulating into the stream s will have their domain included in $P(s)$ and that $C(s)$ is included in this domain.

We use this information to get a tighter representation of mappings and a faster execution:

- We associate to a each element in $P(s)$ a unique integer in $0 \ldots |P(s)|-1$ and mappings are thus represented by n-uplets (with a special value to indicate a missing part).

- Each time we need to get given a compatible mapping m_1 coming from a stream s_1 the set of compatible mappings sent by a stream s_2, we store the mappings of s_2 based on the hash value of $C(s_1) \cap C(s_2)$ and we only need to compare the $(P(s_1) \cap P(s_2)) \setminus (C(s_1) \cap C(s_2))$ part of the domain. In the case where $(P(s_1) \cap P(s_2)) = (C(s_1) \cap C(s_2))$ the running time of the join is linear in the size of both inputs.

  When $D = (P(s_1) \cap P(s_2)) \setminus (C(s_1) \cap C(s_2))$ is very small we can actually get a fast join by enumerating the subsets of D (it is exponential in the size of D thus limited to small D).

  For instance, if $P(s_1) = C(s_1) = \{x, y\} = P(s_2)$ but $C(s_2) = \{x\}$ then we use two hash tables xy_i and x_i for each stream. When a mapping m comes from s_1 we store m in xy_1, look for it in xy_2 store $m' = \{x \to m(x)\}$ in x_1 with a pointer to m and look for m' in x_2. And for a mapping coming from s_2, either it binds x and y and we store it into xy_2 and look for it in xy_1. Or m binds only x and we store m into x_2 and look for a match in x_1.

5.1.4 Special case of distributed systems

In the case of the compilation for a distributed system, the parallelism will not work by affecting to each worker a node of the dataflow program but rather we will instantiate a copy of the dataflow program for each worker. We suppose that our hash function returns elements of the space H and that this space H is partitioned among the workers. Each worker will thus be responsible for a part of the total hash space.

For a stream that links a sender S and a receiver R, by default S will send its mappings to local instance of R with the exception when R needs to perform a hash $h(m_{\mathcal{C}})$ on m in which case S sends the data directly to the instance of R on the worker responsible for $h(m_{\mathcal{C}})$.

For a stream from S to R where R receives only data from the local instance of S, the instance of R ends when the instance of sender ends. If R performs hashes and thus receive data from multiple workers, we need all the instances of S on all the workers to be finished before terminating R.

5.2 Bottom-Up cost model for μ-algebra terms

Now that we have defined the process of the general bottom-up computation we can define a cost-model for the evaluation of μ-algebra terms. This cost model will rely on an auxiliary function.

- the function $estimateSize(\varphi, V)$ will estimate the number of solutions to φ when evaluated in a specific context.
- the function $estimateCostJoin(\varphi_1, \varphi_2, V)$ will estimate the time needed to perform the join of the two sets of mappings $[\![\varphi_1]\!]_V$ and $[\![\varphi_2]\!]_V$. This function is important as the estimation of the running time of a join computation depends on the actual terms being joined (see section 5.1.3).

A perfect estimation for the number of solutions of φ evaluated in the environment V would be to actually compute $|[\![\varphi]\!]_V|$ but this is very often too expensive. In practice we rely on an abstracted version of the environment V. The whole chapter 6 is dedicated to the question of how to build a function $estimateSize$ that is relatively efficient and precise.

In the same manner we will not actually compute the exact time needed to perform $[\![\varphi_1]\!]_V \bowtie [\![\varphi_2]\!]_V$ as it implies to actually compute both sets. The $costJoin$ function will actually depends on the estimation provided by $estimateSize$ for $[\![\varphi_1]\!]_V$, $[\![\varphi_2]\!]_V$ plus the type information of φ_1 and φ_2 to detect cases where the join can be performed using a hash-based system that is linear or a double for-loops which is proportional on both sizes.

Now we can define the cost model. This cost model correspond to a big O number of operations a naive bottum-up evaluator has to perform.

$$
\begin{array}{lcr}
cost(\varphi_1 \cup \varphi_2) & = & cost(\varphi_1) + cost(\varphi_2) + size(\varphi_1 \cup \varphi_2) \\
cost(\varphi_1 \backslash\backslash \varphi_2) & = & cost(\varphi_1) + cost(\varphi_2) + costJoin(\varphi_1, \varphi_2) + size(\varphi_1 \backslash\backslash \varphi_2) \\
cost(\varphi_1 - \varphi_2) & = & cost(\varphi_1) + cost(\varphi_2) + costJoin(\varphi_1, \varphi_2) + size(\varphi_1 - \varphi_2) \\
cost(\varphi_1 \setminus \varphi_2) & = & cost(\varphi_1) + cost(\varphi_2) + costJoin(\varphi_1, \varphi_2) + size(\varphi_1 \setminus \varphi_2) \\
cost(\varphi_1 \ltimes \varphi_2) & = & cost(\varphi_1) + cost(\varphi_2) + costJoin(\varphi_1, \varphi_2) + size(\varphi_1 \ltimes \varphi_2) \\
cost(\varphi_1 \bowtie \varphi_2) & = & cost(\varphi_1) + cost(\varphi_2) + costJoin(\varphi_1, \varphi_2) + size(\varphi_1 \bowtie \varphi_2) \\
cost(\rho_a^b(\varphi)) & = & cost(\varphi) + size(\varphi) \\
cost(\pi_a(\varphi)) & = & cost(\varphi) + size(\varphi) \\
cost(\beta_a^b(\varphi)) & = & cost(\varphi) + size(\varphi) \\
cost(\theta(\varphi, g : \mathcal{C} \to \mathcal{D})) & = & cost(\varphi) + size(\varphi) \\
cost(\sigma_{filter}(\varphi)) & = & cost(\varphi) + size(\varphi) \\
cost(\mu(X = \varphi)) & = & cost(\varphi) + size(\varphi) \\
cost(X) & = & size(X) \\
cost(\emptyset) & = & 1 \\
cost(|c_1 \to v_1, \ldots, c_n \to v_n|) & = & 1
\end{array}
$$

5.2.1 Refining the cost model

The constant hidden in the big O notation in this cost model is very different from operator to operator. Furthermore, in some settings even with a perfect estimation, the time to perform one of the above operator does not really follow a linear model. This is caused by caching effects, swapping, networking phase, etc.

In practical experiments the cost is mainly dominated by joins, IO to the disk and fixpoints (sometimes the three) but the constants vary greatly between single core evaluation where our experiment are dominated by IO (unless there is a very large intermediate result) and distributed computation where joins are expensive (fixpoints would also be expensive but we did not experiment on this as explained below).

5.2.2 Cost rules

In practice, the evaluators we will present are almost bottom-up evaluators but they come with an optimized version of some the operators. For instance when the graph is encoded in μ-algebra as T then our evaluator might have indexed T and perform $\sigma_{p=\texttt{:knows}}(T)$ using this index and thus be much faster than actually scanning the whole T and returning elements to which the predicate is *knows* and clearly sub linear in T.

In practice, we could design a cost model for each implementation of μ-algebra but we prefer to handle all cases by keeping the general model of computation and tweaking it using *cost rules* that are passed to the query optimizer. A *cost rule* is a rule of the form $(pattern, cost(pattern))$. Such a rule recognize a *pattern* that we can evaluate either in the naive way or using a specialized version whose cost estimation is $cost(pattern)$.

Developing on the example of earlier, one such pattern could be $\sigma_{p=\texttt{:knows}}(T) \rightarrow size(\sigma_{p=\texttt{:knows}}(T))$. This pattern is used by a cost model when instead of computing the whole T and then filtering the triples such that $p = \texttt{:knows}$ we can more directly access the result of $\sigma_{p=\texttt{:knows}}(T)$ because we have some index on T.

Note that we do not necessarily have that the cost of a cost rule is less than the cost of the naive evaluation in which case our evaluator will fall back to the naive evaluator. In this sense our evaluators are bottom-up evaluators extended with efficient evaluation of some patterns.

These cost rules might also take into account the fact that some operations can be performed much more efficiently if we look at the underlying model. For instance the variable N representing the set of valid nodes can sometimes not be removed from a formula but N can be implemented much more efficiently than a double pass on the set of all triples whenever the evaluator maintains a set of nodes.

5.2.3 Examples

Let us reuse the terms of section 4.9 and let us suppose that we are querying a graph that has the following properties: there are 10^8 nodes, 10^6 different lastnames, 100 firstnames and the relation `:knows` is symmetric and its transitive closure forms 10^4 connected components. For the sake of simplicity, in our model, all first and last names are equiprobable and the connected components are roughly of the same size (10^4 elements).

All our terms will compute the solutions to the two non recursive PP so let us compare only the difference in cost between various terms:

1. All the terms 2^* and 4^* compute the whole $PP(?a$ `:knows`* $?b)$ which contains $10^8 \times 10^4$ terms which means the cost is at least 10^{12};

2. The term 3 initializes the fixpoint computation with the $?a$ last named `:Doe` (and there are $10^8/10^6 = 10^2$ such persons). The whole fixpoint computation is dominated by the join between the `:knows` relation and X. The fixpoint returns a set of size $10^2 \times 10^4$ that is joined with the set of $?b$ first named `:John`. In total the order of magnitude is 10^6 plus the join between `:knows` and the recursive variable.

3. The term 5 does the same thing with $?b$ first named `:John`. The fixpoint therefore manipulates $10^6 \times 10^4$ elements and the final join is between this result and a set containing 10^2 elements (the `:Does`). In total the order of magnitude is 10^{10} plus the join of `:knows` with the recursive variable which is much more than the cost of term 3.

In this made up scenario we see that the different terms can have very different cost models. Furthermore we see that, using our cost model, the most efficient term is one that has been obtained through a rule pushing a join inside a fixpoint.

5.3 Single core bottom-up evaluation of μ-algebra

Building on the SPARQL to μ-algebra translation presented in section 3.5, on the rewriting rules presented in the chapter 4 and the bottom-up evaluation presented just above we implemented a prototype capable of executing a SPARQL query on a dataset.

While the tools is only based on the techniques presented, it supports only a restricted fragment of SPARQL : triple pattern, property paths, conjunction, union and optional; the aggregation, for instance, are not yet supported. This tool serve as a basis for our experimentation and should be seen as a research prototype rather than a finished SPARQL evaluator.

Our prototype tool is implemented using the OCAML programming language and is available online at the address `https://gitlab.inria.fr/jachiet/musparql/`.

5.4 Towards a distributed μ-algebra evaluator

Considering the ever increasing amount of data produced, we have the goal of developing a distributed μ-algebra evaluator. The general bottom-up evaluation presented in section 5.1 produces query execution plans that are almost entirely parallel: each operator is translated with streams and individual elements in streams can be treated in parallel with either the help of distributed hash tables or with streams equipped with a hashing mechanism.

However writing programs that are features rich, stable, resilient and scalable distributed programs is known to be a major challenge. And the single-machine counterpart of distributed databases often perform an order of magnitude faster (in terms of total CPU consumption). Instead of implementing the approach presented above, we tried available frameworks such as Map Reduce, Spark or Flink for handling the distribution.

5.4.1 Framework for distributed computations

Parallel computation generally refers to the act of using multiple units of computing power (potentially on several machines) to perform a computation. By using the term *distributed*

computation, we put the emphasis on the fact that the computation happen in several machines that can only share information by sending messages. Furthermore, as we are using several machines, some of them might crash during the computation, or might get disconnected from the network therefore the *fault-tolerance* might be important (for long jobs running on many machines). The distributed frameworks take care of running the multiple units of computation, of the resiliency of the system (they detect and sometimes even correct the failures) and handle the communication between individual workers.

5.4.2 Map-Reduce

Map-Reduce is a very simple model of computation. The data is thought of as bags of individual values on which the programmer can use two primitives *map* and *reduce*. This programming model is very simple but arbitrary user-defined functions can be passed as arguments to *map* and *reduce* and this simplicity translates into a resilient and scalable distributed framework.

They are competitive SPARQL query executors that are based on the Map-Reduce framework (such as *CliqueSquare* [GKM+15]) but we only use Map-Reduce for data storage as other frameworks emerged that have a richer set of instructions, more optimization and, as a consequence perform generally better.

5.4.3 Apache Spark

Map-Reduce has several limitations. One such limitation is obviously the simplicity of the programming model but one has to also consider the performance of map-Reduce jobs. In Map-Reduce computations, the resiliency is obtained by storing the results of individual map-Reduce jobs into a resilient storage. In, e.g., Apache Hadoop, this means to store each result thrice to ensure resiliency.

Spark is a distributed computation framework introduced in 2012 [ZCD+12] to overcome these limitations. Spark has several primitives that revolve around the concept of *RDD*s.

Resilient Distributed Datasets

A *Resilient Distributed Datasets* (RDD) represents a typed bag of values. The novelty of the RDD is not only to store the values but also to keep track the *lineage*, i.e. the computation that was performed to obtain those values. Contrarily to Map-Reduce that stores intermediate results into a fault-tolerant disk-based storage, Spark tries to store them into RAM. When a machine is not responsive (either slow, crashed or disconnected from the network), Spark will start from the last checkup point and reconstruct the missing part using this lineage. Obviously when a lot of computation has been performed on the data, reconstructing the missing data might trigger heavy computation but Spark checkpoint by default partitions (until RAM is exhausted) and the programmer can also trigger a disk-based resilient checkup.

Operators with Spark

Spark provides some operations on top of these RDD. The set of such operation extend which each new version of Spark but it contains:

- *flatMap* which corresponds to the map in the map-Reduce programming model;

- *Union* which merges the content of two RDDs of the same type;
- *coGroup* which reduces several RDD $r_1, \ldots, r_n$ along a set of keys and thus returns a RDD that is $n+1$-tuple, for each key k appearing in one the RDD, it contains k followed by n list, the i-th list containing the list of values in the RDD i associated with the key k;
- Spark also allows the programmer to sort the data, control the partitioning, checkpoint, and output of RDD in several ways.

And, as syntactic sugar, Spark also has the following operators:

- *map* and *filter* which are specialized versions of *flatMap* restricting the output size of the function to exactly one and zero or one;
- *join* and *leftJoin* which are specialized version of *coGroup*;
- *ReduceByKey* which corresponds to the reduce of the map-Reduce and can be expressed as a *coGroup*.

Broadcast Variables

When a Spark program performs a computation (e.g., a map) it needs to capture the lineage and then send the data the computation will use (in the case of map, the function of the map). This is similar to what happens in Map-Reduce frameworks where the functions applied also need to be sent to all workers. However, while Map-Reduce jobs can be performed in batch and where the function can be a complete program that is sent (depending on the framework), Spark performs interactively which means the function is only known at running time. Capturing and sending the data thus corresponds to taking the closure of the function and serializing it.

When there are a lot of data stored in the function, taking the closure and serializing the function can take a lot of time and will use a lot of network. To overcome this limitation Spark introduces *broadcast variables*.

Broadcast variables compute the closure of a term and then send the serialization of the closure in a peer-to-peer fashion to all working nodes. This way we allow the programmer to avoid serializing multiple times the same data and the sharing mechanism can be done in parallel of computing tasks.

For instance, if we have a text database and a dictionary of some sort. If the dictionary takes a few hundred megabytes then each filter or map using this dictionary would imply to send the whole dictionary. If the dictionary is stored in a broadcast variable, then it would be distributed more efficiently and Spark does not have to send the dictionary along with each use.

5.4.4 SparqlGX

SparqlGX [GJGL16a] is a distributed sparql query evaluator. SparqlGX origins predate the inception of the μ-algebra but the μ-algebra helped us to extend and characterize the supported fragment. The syntax of the fragment of sparql queries accepted by SparqlGX

is more restricted than the syntax of SPARQL translatable to μ-algebra as it e.g. does not consider Property Paths which were one of the main motive behind the inception of μ-algebra.

SparqlGX uses the Spark framework to manage the distributed computation. However, Spark does not natively support efficient recursion nor general joins or minuses. Thereby we cannot directly translate our work on the μ-algebra over to Spark. Yet, using ideas developed for μ-algebra, I was able to advance SparqlGX in multiple aspects, for instance extending its fragment (based on the *safe* compatible lookup) or optimize more (based on cardinality estimation).

We will not go deep into the details of the architecture of SparqlGX but if the reader is interested we invite her/him to look into the submitted journal version [GJGL17] of our SPARQL paper and we now present two extensions of SparqlGX enabled by our work on μ-algebra.

The two modes of SparqlGX

SparqlGX can operate into two modes: *standalone* or with a *load* phase. When operating in the standalone version, SparqlGX takes a file containing the whole dataset and run the query against this file. The optimization strategy has no knowledge of the data and therefore proceed using an heuristic approach.

In the *load* version, SparqlGX stores the data using the vertical partitioning approach. The main concept behind this idea is to store a triple $s\ p\ o$ by storing the couple s, o in a file named p which allows SparqlGX to process more efficiently queries where no triple pattern has a variable predicate. Furthermore, the load phase allows SparqlGX to capture statistics about the data that can be used for the cardinality estimation phase (as explained in chapter 6). These statistics will allow to choose more efficient plans for terms as we will explain in the next chapter.

5.4.5 Safe joins for SparqlGX

With these extensions, SparqlGX now supports the SELECT fragment of the SPARQL query language with modifiers and where the graph pattern is a query composed of triple patterns, conjunctions, disjunctions and optionals but where conjunctions (resp. optionals) are restricted to *safe* conjunctions (resp. *safe* optionals). The syntax of SPARQL queries accepted in SparqlGX is presented in figure 5.2.

A operator between two μ-algebra terms φ_1, φ_2 (or equivalently between two SPARQL subqueries) is *safe* when $P(\varphi_1) \cap P(\varphi_2) = C(\varphi_1) \cap C(\varphi_2)$ which allows us (as explained in section 5.1.3) to have a fast join algorithm.

```
Query :=
        | TP
        | Query JOIN Query
        | Query OPTIONAL Query
        | Query UNION Query
        | Filter(cond,Query)

TP  :=  UV UV UV

UV  :=
      | ?variable
      | <uri>
      | "litteral"
```

Figure 5.2: Syntax of supported SPARQL queries in SparqlGX.

Conclusion

In this chapter, we have investigated how we can evaluate μ-algebra terms and deduced from this evaluation a cost model for our general evaluation scheme. We have then presented two evaluators that we implemented based on this general compilation scheme. These two evaluators correspond to two different types of applications: SparqlGX handles only a fragment of the sparql languages but run on an efficient distributed platforms, `musparql` on the other hand is a single core evaluator but can handle and optimize complex queries.

In our general optimization scheme for the evaluation of μ-algebra terms we rely on a cost model to guess what is the estimated best QEP. And this cost model itself relies on a cardinality estimation. While a naive cardinality estimation leads to already interesting results, we investigated the use of more complex schemes to assess the number of solutions to a μ-algebra term.

The task of estimating the number of solutions to a given query, even a simple conjunctive query, has been the center of many research projects and yet still is an active research subject. The next chapter is devoted to present the current state of our research, especially in the context of SparqlGX (i.e. the safe fragment).

CHAPTER 6

Cardinality estimation of μ-algebra terms

Our general method of compilation relies on a rewrite systems that produce multiples terms. A cost model is then used to determine the most efficient term but this cost model uses a cardinality estimation.

There exists a variety of query evaluation schemes but, in most of them, estimating the cardinality of intermediate results is key for performance, especially when the computation is distributed and the datasets are very large. For example, it helps in choosing a join order that minimizes the size of intermediate results. It is therefore a well studied subject, especially for the relational model and for SQL *databases. Estimating the cardinality of* SPARQL *queries however, even for the simple conjunctive fragment, raises new challenges.*

In this context, we propose a new cardinality estimation based on statistics about the data. Our cardinality estimation is a worst-case analysis tailored for the conjunctive fragment of SPARQL *and capable of taking advantage of the implicit schema often present in* RDF

datasets (e.g. functional dependencies). This implicit schema is captured by statistics therefore our method does not need for the schema to be explicit nor perfect (our system performs well even if there are a few "violations" of these implicit dependencies).

We implemented our cardinality estimation and used it to optimize the evaluation of queries by SparqlGX*. We benchmark* SparqlGX*: equipped with our cardinality estimation, the query evaluator* SparqlGX *performs better against most queries (sometimes by an order of magnitude) and is only ever slightly slower.*

While our approach is mainly focused on BGP as it is a well established fragment of SPARQL *note that our approach applies to the whole conjunctive fragment of μ-algebra.*

As this section could be read relatively independently from the rest of this thesis, we recall some definition specific for the evaluation of BGP.

Definition 27 (Mapping Collection)**.** *A "mapping collection" is a set of mappings with the additional information of a domain $d_1, \ldots, d_k$ shared by all mappings in the collection (i.e. all the mappings in the collection have $d_1, \ldots, d_k$ as domain).*

The solutions of a TP (s, p, o) on a graph G is the mapping collection whose domain is the domain of (s, p, o) and a mapping m belongs to this mapping collection if $(s', p', o') \in G$ (where $x' = x$ when $x \in \mathcal{V}$ and $x' = m(x)$ otherwise).

Definition 28 (Compatible Mappings)**.** *Two mappings m_1 and m_2 are* compatible *(written $m_1 \sim m_2$) when $m_1(c) = m_2(c)$ for all $c \in dom(m_1) \cap dom(m_2)$.*

Given two compatible mappings m_1 and m_2 we define their sum as the mapping whose domain is $dom(m_1) \cup dom(m_2)$ and $(m_1 + m_2)(c) = m_1(c)$ when $c \in dom(m_1)$ and $m_2(c)$ otherwise.

Definition 29 (Join)**.** *Given two mapping collections A and B, the join of A and B (written $A \bowtie B$) is defined as $(m_A + m_B \mid m_A \sim m_B$ and $(m_A, m_B) \in A \times B)$*

A BGP is a list of TPs $(t_1, \ldots, t_n)$, the solution of a BGP $(t_1, \ldots, t_n)$ is the collection mapping corresponding to the join of solutions of individual TP (the join operation is associative and commutative so all join orders lead to the same mapping collection). In this section, we first tackle the problem of estimating the number of solution for a given BGP on an RDF dataset then briefly explain how we plain to handle the full μ-algebra cardinality estimation.

6.1 Summaries

We will now manipulate projections of mapping collections. If c is in the domain of a mapping collection $M = m_1, \ldots, m_n$, the values $(m_1(c), \ldots, m_n(c))$ form a multiset which is the projection of the mapping collection M on the column c. Our cardinality estimation is based on summaries of projection of mapping collections.

The projections that we manipulate are often very large (as large as the query answer), that is why, in practice, we manipulate multiset summaries. Multiset summaries are a tight representation that over-approximates multisets.

6.1.1 Definitions

The informal intuition behind a multiset summary is to represent it using the most frequent elements and summarize the rest with three numbers.

Given a multiset $M = (m_1, \ldots, m_n)$ the indicator function $\chi_M(x) = |\{i \mid x = m_i\}|$ counts the number of each element m from M such that $m = x$. Given a multiset M represented via its indicator function $\chi : S \to \mathbb{N}$, we will compute a small set $S' \subset S$ and represent M in two parts: the elements of M belonging to S' and the other elements (belonging in $S \setminus S'$). In order to represent the elements of M in S', we simply restrict the indicator function χ to this S' and to represent the multiset E of elements of M in $S \setminus S'$ we use three integers: T the **T**otal number of elements in E, D the number of **D**istinct values in E, and Y the maximal multiplicit**Y** of an element in E.

Definition 30 (Multiset summary)**.** *Formally, a multiset summary corresponds to a quintuple* $< S', \chi', T, D, Y >$ *where* $S' \subset S$, χ' *is a function from* S' *to* $\mathbb{N}$, *and* $(T, D, Y) \in \mathbb{N}^3$. $< S', \chi', T, D, Y >$ *is a summary of the multiset represented by* $\chi_M : S \to \mathbb{N}$ *when:*

- χ' *over-approximates* χ *on* S', *i.e.* $\forall v \in S' \quad \chi(v) \leq \chi'(v)$;
- Y *is an upper bound on* $\chi(v)$ *for* $v \notin S'$, *i.e.* $\forall v \in S \setminus S' \quad \chi(v) \leq Y$;
- T *is an upper bound on the number of elements* ***counted with multiplicity*** *of the multiset not in* S', *i.e.* $\sum_{v \in S \setminus S'} \chi(v) \leq T$;
- D *is an upper bound on the number of* ***distinct*** *elements of the multiset that is not in* S', *i.e.* $|\{v \in S \setminus S' \mid \chi(v) > 0\}| \leq D$.

Definition 31 (Column summary)**.** *The multiset summary* $< S, \chi, T, D, Y >$ *is a* column summary *for the column* c *of the mapping collection* $m_1, \ldots, m_k$ *when* $< S, \chi, T, D, Y >$ *is a summary for the multiset* $(m_1(c), \ldots, m_k(c))$.

Definition 32 (Collection Summary)**.** N, s *is a* collection summary *for the collection mapping over the domain* $c_1, \ldots, c_k$ *if* $N \in \mathbb{N}$ *is greater than the number of mappings in the collection and* s *is a set of pairs* $s = \{(c_1, S_1), \ldots, (c_k, S_k)\}$ *where each* S_i *is a column summary of the column* c_i.

Example of summaries

Let us consider the following RDF dataset of 11 triples:

```
A memberOfTeam 1   A memberOfTeam 2   A memberOfTeam 3
B memberOfTeam 1   C memberOfTeam 1   E memberOfTeam 3
1 teamLeader B     2 teamLeader A     3 teamLeader C
4 teamLeader D     5 teamLeader E
```

There are two predicates: `memberOfTeam` and `teamLeader`. A possible collection summary for the TP ($?s$ `memberOfTeam` $?o$) is $6, \{?s \to < \{A\}, \{A \to 3\}, 3, 3, 1 >; ?o \to < \{1\}, \{1 \to 3\}, 3, 2, 2 >\}$ and for the TP ($?s$ `teamLeader` $?o$) one possible collection summary is $5, \{?s \to < \{B\}, \{B \to 1\}, 4, 4, 1 >; ?o \to < \{1\}, \{1 \to 1\}, 4, 4, 1 >\}$.

With only the information of these summaries we can deduce that, in the dataset, the relation induced by the `memberOfTeam`, is such that the subject `A` might appear several times but –except for this `A`– other team members have only one team. In addition, we know that the team `1` has 3 members and the other teams have less than 2 members.

In the summary of the relation induced by `teamLeader` (i.e. the relation between $?s$ and $?o$ in ($?s$ `teamLeader` $?o$)) we see that this relation is bijective: all the subjects $?s$ and all the objects $?o$ are each present only once.

Therefore if we need to compute the solutions of the BGP:
($?member$ `memberOfTeam` $?team$. $?team$ `teamLeader` $?leader$) we know that there are less than 6 solutions: the relation `teamLeader` is bijective therefore the number of solutions of this BGP is less than the number of solutions for the TP: ($?member$ `memberOfTeam` $?team$).

Simple operations on summaries

We define the following operations on multiset summaries:

- $count(cs, v)$: the function that returns the number of times the value v can appear in a multiset summarized by cs;

$$count(< S, \chi, T, D, Y >, v) = \begin{cases} \chi(v) & v \in S \\ Y & v \notin S \end{cases}$$

- $truncate(cs, n)$: the multiset summary where we enforce that each value appears at most n times. Formally, $truncate(< S, \chi, T, D, Y >) =$
$< S, min(\chi, n), min(T, D \times n), D, min(Y, n) >$
(where $min(\chi, n)(x) = min(n, \chi(x))$);

- $limit(cs, n)$: the multiset summary where we enforce that there are at most n elements. More precisely we have $limit(< S, \chi, T, D, Y >) =$
$< S, min(\chi, n), min(T, n), min(D, n), min(Y, n) >$;

- $size(cs)$: the estimated size of the multiset summarized by cs, $size(< S, \chi, T, D, Y >) = \sum_{x \in S} \chi(x) + T$;

- the sum of two multisets over S represented by χ_1 and χ_2 is defined as $(\chi_1 + \chi_2)(x) = \chi_1(x) + \chi_2(x)$. The sum of two multiset summaries is defined as $< S_1, \chi_1, T_1, D_1, Y_1 >$ $+ < S_2, \chi_2, T_2, D_2, Y_2 >=< S_1 \cup S_2, \chi', T_1 + T_2, D_1 + D_2, Y_1 + Y_2$ with:

$$\chi'(x) = \begin{cases} \chi_1(x) + \chi_2(x) & \text{when } x \in S_1 \cap S_2 \\ \chi_1(x) + Y_2 & \text{when } x \in S_1 \setminus S_2 \\ Y_1 + \chi_2(x) & \text{when } x \in S_2 \setminus S_1 \end{cases}$$

  We have the property that if a multiset m_1 is summarized by s_1 and m_2 by s_2 then $m_1 + m_2$ is summarized by $s_1 + s_2$;

- Given a multiset $\{m_1, \ldots, m_k\}$ with its summary $< S, \chi, T, D, Y >$ we can multiply them by an integer n: $\{m_1, \ldots, m_k\} \times n$ is the multiset containing $k \times n$ elements: n elements m_i for each $1 \leq i \leq k$; this multiset is summarized by $< S, \chi, T, D, Y >$ $\times n =< S, \chi \times n, T \times n, D, Y \times n >$ (with $(\chi \times n)(x) = \chi(x) \times n$);

- Given a collection summary $s = (t, \{(c_1, s_1), \ldots, (c_k, s_k)\})$ we note: $summ(s, c_k) = s_k$ the summary for the column c_k; $cols(s) = \{c_1, \ldots, c_k\}$ the set of columns of the summarized collection and $size(s) = t$ the size of the summary over-approximated by s.

6.1.2 The multiplicative factor

Given two collection summaries s_A (resp. s_B) summarizing two collection mappings $A = \{A_1, \ldots, A_n\}$ (resp. $B = \{B_1, \ldots, B_m\}$) we want to compute a collection summary for the solutions of $A \bowtie B$. In order for this summary to be precise, we will introduce in this section the "multiplicative factor" and in the next section we will show how to use the multiplicative factor to compute a relatively precise summary for $A \bowtie B$.

The mapping collection $A \bowtie B$ can be seen as a cartesian product $A \times B$ where we removed mappings (m_A, m_B) that do not agree on all the common columns of A and B. Each mapping m is thus built using a unique pair $(m_A, m_B) \in A \times B$ (but such a pair does not necessarily

correspond to a mapping of $A \bowtie B$). If we track which mappings of A and B were used to build which mappings of $A \bowtie B$, we can count for each mapping $m_A \in A$ its "multiplicative factor": the number of mappings in $A \bowtie B$ that were built using m_A. The i-th multiplicative factor of A toward B is defined as the i-th greatest multiplicative factor of an element of A (or 0 if $|A| < i$).

As summaries work with over-approximation we will show in this section how to compute, with only the summaries for A and B, a bound for the i-th multiplicative factor $mult(s_A, s_B, i)$, i.e. an over-approximation to $mult(A, B, i)$ for any A and B such that A (resp. B) is summarized by s_A (resp. s_B).

A common column

Let c be a column shared between A and B (note that such a c does not necessarily exist). A mapping $m_A \in A$ can only be combined with mappings $m_B \in B$ to form mappings of $A \bowtie B$ when $m_A(c) = m_B(c)$. Therefore, a mapping $m_A \in A$ can only be used to build, at most, $count(summ(s_B, c), m_A(c))$ mappings of $A \bowtie B$.

Let $cs_A =< S_A, \chi_A, T_A, D_A, Y_A >$ (resp. $cs_B =< S_B, \chi_B,$
$T_B, D_B, Y_B >$) be the column summary for the column c in s_A (resp. in s_B) then for each $v \in S_A \cup S_B$ there are, at most, $count(s_A, v)$ mappings of A and each can be combined with $count(cs_B, v)$ mappings of B. To that we need to add that the, at most, T_A mappings $m \in A$ with $m(c) \notin S_A \cup S_B$ can each be joined with, at most, Y_B elements.

In total this gives us $T_A + \sum_{v \in S_A \cup S_B} count(cs_A, v)$ values that over-approximate the various multiplicative factors of elements of A. By sorting these values, the i-th greatest value gives us a bound on $mult(s_A, s_B, i)$.

Note that the mappings with values for the columns c falling into $S_B \setminus S_A$ might be counted twice: each of the T_A elements produces Y_B mappings of $A \bowtie B$, but for each $v \in S_B \setminus S_A$ we also counted that Y_A mappings produced $\chi_B(x)$ each. However, this is not an actual issue since we combine the several bounds on $mult(s_A, s_B, i)$, in particular, $i > size(s_A)$ implies $mult(s_A, s_B, i) = 0$ and $size(s_A) \leq T_A + \sum_{v \in S_A} count(cs_A, v)$.

General case

There are no more than $size(s_A)$ mappings in A and each can be used in at most $size(s_B)$ mappings of $A \bowtie B$, we have a first bound:

$$mult(s_A, s_B, i) \leq \begin{cases} size(s_B) & \text{when } i \leq size(s_A) \\ 0 & \text{otherwise} \end{cases}$$

Then, we simply use the technique presented earlier on each column shared between the domains of A and B. Each column giving us a new bound on $mult(s_A, s_B, i)$ and we combine all of them: if $mult(s_A, s_B, i) \leq k_1$ and $mult(s_A, s_B, i) \leq k_2$ then $mult(s_A, s_B, i) \leq min(k_1, k_2)$.

An example

Lets us consider the two datasets and their join presented in figure 6.1 both containing a letter, a color and in one we have an ID and in the other a GID. In the joined dataset, we have all of these columns. Notice that valid RDF would require to have URIs but for the sake

of clarity we use numbers, letters and colors that could easily be translated into URIs (e.g. 1 could be translated into <`http://example.com/1`>).

Set 1

letter	color	ID
A	Red	0
B	Red	1
C	Red	2
B	Blue	3
B	Green	4
B	Yellow	5
B	Fuchsia	6

Set 2

letter	color	GID
B	Red	g0
A	Red	g1
A	Red	g2
A	Blue	g3
A	Blue	g4
A	Blue	g5

Set 1 $\bowtie$ Set 2

letter	color	ID	GID
B	Red	1	g0
B	Red	2	g0
A	Red	0	g1
A	Red	0	g2

Figure 6.1: Example datasets.

Thus, a collection summary for the Set 1 is:
$(7, \{\, letter \rightarrow < \{A, B, C\}, \{A \rightarrow 1, B \rightarrow 5, C \rightarrow 1\})\}, 0, 0, 0 >,$
$color \rightarrow < \{Red, Blue\}, \{Red \rightarrow 3, Blue \rightarrow 1\})\}, 2, 2, 1 >,$ And a collection summary for
$ID \rightarrow < \{0, 1\}, \{0 \rightarrow 1, 1 \rightarrow 1\})\}, 4, 4, 1 >\})$
the Set 2 is:
$(6, \{\, letter \rightarrow < \{A, B\}, \{A \rightarrow 5, B \rightarrow 1\})\}, 0, 0, 0 >,$
$color \rightarrow < \{Red, Blue\}, \{Red \rightarrow 3, Blue \rightarrow 3\})\}, 0, 0, 0 >,$
$GID \rightarrow < \{g0, g1\}, \{g0 \rightarrow 1, g1 \rightarrow 1\})\}, 4, 4, 1 >\})$

Now, we can represent graphically the multiplicative factor from Set 1 to Set 2. The i-th value represents the i-th multiplicative factor. In the figure 6.2, the 7×6 grid represents the first bound on the multiplicative factor. There are 7 elements in Set 1, each of which can be multiplied 6 times.

The red part corresponds to the bound given by the column summary for the column letter: A is present once in Set 1 but 5 times in Set 2 so we have one 5 in our multiplicative factors; B is present 5 times in Set 1 but once in Set 2 so we have five 1 in our multiplicative factors C is present once in A but not in Set 2 therefore we have one 0. We sort them and it gives us the red part: 5, 1, 1, 1, 1, 1, 0.

The green part corresponds to the bound given by the column color. *Red* is present thrice in Set 1 and each can be combined with 3 elements from Set 2 and *Blue* is present once but can also be multiplied thrice. Once sorted, it gives us: 3, 3, 3, 3.

The blue part corresponds to the min of all bounds (which is also the intersection of shapes). The total number of elements in the join guessed by our method is the area of this blue part which is 6.

Our method guessed 6 while the actual size of the joined set is only 4 but notice that there exists sets with the same summaries that have a join of size 6. For instance, we could exchange the color of ID 0 and ID 3 in Set 1 to get a join of size 6 (for which our algorithm would be perfect).

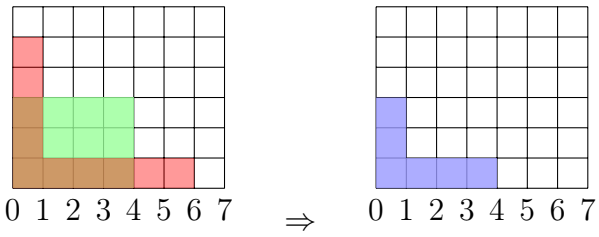


Figure 6.2: Combining multiplicative factors for several columns.

The total operator

Once the multiplicative factors from A to B for all $i \in \mathbb{N}$ (i.e. $mult(s_A, s_B, i)$) have been computed we can compute the total operator.

The function $total(s_A, s_B, n, k)$ gives a bound on the number of mappings of $A \bowtie B$ that can be built using n different mappings of A when we know that each of those mappings has a multiplicative factor bounded by $k \in \mathbb{N} \cup \{\infty\}$:

$$total(s_A, s_B, n, k) = \sum_{i \leq n} min(k, mult(s_A, s_B, i))$$

For instance in the example given above we know that the *total* number of mappings is $total(s_A, s_B, size(s_A), \infty)$. And if we want to now the number of entry *Red* in the column summary of the column color of $Set1 \bowtie Set2$, we know that there are $n = 3$ *Red* in $Set1$ and $k = 3$ in $Set2$. We thus know that there are less than $n \times k = 9$ *Red* in the column *color* of $Set1 \bowtie Set2$ but the function $total(s_A, s_B, 3, 3) = 5$ gives us a tighter bound.

6.1.3 Joining summaries

Given two collection summaries s_A (resp. s_B) summarizing two collections mappings $A = \{A_1, \ldots, A_n\}$ (resp. $B = \{B_1, \ldots, B_m\}$) we want to compute a collection summary for the solutions of $A \bowtie B$.

We suppose that we computed a bound for $total(s_A, s_B, n, k)$. The total number of mappings of $A \bowtie B$ can be bounded by
$min(total(s_A, s_B, size(A), size(B)), total(s_B, s_A, size(B), size(A))$.

Combining summaries for non common columns

Let c be a column in the domain of A but not in the domain of B, let $cs_A =< S_A, \chi_A, T_A, D_A, Y_A >$ be the summary for the column c of A, the summary we compute for the column c of $A \bowtie B$ is $< S_a, \chi', T', D_A, Y' >$ where:

- $T' = total(s_A, s_B, T_A, \infty)$
- $Y' = total(s_A, s_B, Y_A, \infty)$
- $\chi' = total(s_A, s_B, \chi(x), \infty)$

Columns that are in the domain of B but not in the domain of A are treated symmetrically.

Combining summaries for a common column

Let c be a column in the domain of both A and B, let $cs_A =< S_A, \chi_A, T_A, D_A, Y_A >$ (resp. $cs_B =< S_B, \chi_B, T_B, D_B, Y_B >$) be the summary for the column c of A (resp. of B). Each mapping m of A will be used in at most $count(cs_B, m(c))$ mappings of $A \bowtie B$, therefore the $count(cs_A, m(c))$ mappings sharing the value $m(c)$ will be used in, at most, $total(s_A, s_B, count(cs_A, m(c)), count(cs_B, m(c)))$ mappings of $A \bowtie B$. Note that by symmetry between A and B we also have that it will be used in, at most, $total(s_B, s_A, count(cs_B, m(c)), count(cs_A, m(c)))$ mappings of $A \bowtie B$.

Our summary for the column c of $A \bowtie B$ is $< S_a \cup S_b, \chi', T', min(D_A, D_B), Y' >$ where:

- $T' = min(total(s_A, s_B, T_A, Y_B), total(s_B, s_A, T_B, Y_A))$
- $Y' = min(total(s_A, s_B, Y_A, Y_B), total(s_B, s_A, Y_B, Y_A))$
- $\chi'(x) = min(total(s_A, s_B, count(cs_A, x), count(cs_B, x)),$
  $total(s_B, s_A, count(cs_B, x), count(cs_A, x)))$

6.1.4 Computation of the join summaries in practice

When computing a summary representing the join of two given summaries, most of the running time of the algorithm is spent in the computation of the total operator and computing $mult(s_A, s_B, i)$.

We optimize the computation of these using a clever representation. First, we see that the multiple *mult* are obtained as the combination (through a min) of piecewise constant functions and thus is also a piecewise constant function and the number of pieces of $i \rightarrow mult(s_A, s_B, i)$ is linear in the number of pieces defining s_A and s_B (i.e. the total number of distinguished elements in s_A and s_B which is generally very small compared with $size(s_A)$ and $size(s_B)$).

If there are m pieces of the function $i \rightarrow mult(s_A, s_B, i)$ then naively computing $total(s_A, s_B, n, k)$ would be linear in m. However using a tree to compute partial sums we can get down to a logarithmic computation (i.e. $O(ln(m))$) of $total(s_A, s_B, n, k)$. Since there are $O(m) \times c$ calls (where c is the number of columns represented by A and B) to this function this gives us a $O(ln(m) \times m \times c)$ algorithms to compute the summary of the join of two multisets represented by summaries whose size is m and the number of columns is c.

6.2 Computing collection summaries representing the solutions of a single TP

6.2.1 Computing a multiset summary from a multiset

Given a multiset defined by χ on the set S we compute its multiset summary of size K by sorting S by χ decreasing, we extract from this the set S' of the K first elements ($S' \subseteq S$, the set of size K with the biggest χ). Then the computed summary is $< S', \chi, T, D, Y >$ where $T = \sum_{x \in S \setminus S'} \chi(x)$, $D = |\{x \in S \setminus S' \mid \chi(x) > 0\}|$ and $Y = max_{x \in S \setminus S'}(\chi(x))$.

Choosing K allows to set a balance between precision of summaries and the time needed to compute them. In practice we adopt the same constant K for all summaries. We notice

that computing summaries with several thousands of elements performs well in practice. In section 7.5, we report on practical experiments with $K = 3000$.

6.2.2 Gathering statistics

We recall that in the RDF format, the predicate carries the "semantic" relationship. In most datasets there is usually a limited number of different predicates in the datasets and in queries variable predicates are relatively rare [AFMPF11].

During the load phase we compute the list of all predicates P (in one pass over the data) and (in a second pass) we compute for each $p \in P$ (in parallel), a collection summary corresponding to the solution of the TP $(?s\ p\ ?o)$. To do that, we compute the list T_p of triples that have p as a predicate, we then compute a multiset summary o_p for the object of T_p and a multiset summary s_p for the subjects of T_p, the collection summary is $|T_p|, \{(?s, s_p), (?o, o_p)\}$.

Summaries are computed recursively: we start by computing summaries for individual TP and then combine them. Let $(t_s\ t_p\ t_o)$ be a TP, let us show how to compute its associated summary.

6.2.3 Fixed predicate $t_p = p$

Let us consider first, the cases where the predicate is fixed to a value p, depending on whether the subject is fixed (either the variable $?s$ or the value s) and whether the object is fixed (either $?o$ or o) we have four cases (the case $(s, ?o)$ is symmetrical to $(?s, o)$ and thus not treated):

Case $t_s = ?s$ and $t_o = ?o$: the returned summary is simply: $size(s_p), \{(?s, s_p), (?o, o_p)\}$.

Case $t_s = s$ and $t_o = o$: this TP has either 0 or 1 solution and binds 0 columns. The returned summary is $(0, \emptyset)$ when $count(s_p, s) = 0$ or $count(o_p, o) = 0$ and $(1, \emptyset)$ otherwise.

Case $t_s = ?s$ and $t_o = o$: this TP has, at most, $count(o_p, o)$ solutions and only binds the column $?s$. The returned summary is $count(o_p, o), \{(?s, truncate(s_p, 1))\}$.

6.2.4 Variable predicate $t_p = ?p$

Let us note r, the collection summary for the solutions of $(t_s\ t_p\ t_o)$, the idea is to combine the summaries s_i (summary for $(t_s\ p_i\ t_o)$) for each $p_i \in P$. We build r such that $size(r) = \sum_{p_i \in P} size(s_i)$ and for each eventual bounded column c ($c \in \{?s, ?o\}$), then $summ(r, c) = \sum_{p_i \in P} summ(s_i, c)$ and, finally, the summary for the column $?p$, is $< P, \chi_p, 0, 0, 0 >$ where $\chi_p(p_i) = size(s_i)$.

6.2.5 Duplicated variable

It is possible in SPARQL to have a variable that is present twice (or thrice). Since there are only three parts to a triple pattern there is at most one duplicated variable. If t_p is the duplicated variable, we apply the replacement scheme proposed in 6.2.4 but we replace all the duplicated parts (and not only t_p).

If the predicate is not duplicated but we have a duplicated variable then the triple is of the form $(?s\ p\ ?s)$ (if the predicate is variable we first apply the replacement scheme).

Let $T_p, \{(?s, s_p), (?o, o_p)\}$ be the collection summary pre-computed for the TP $(?s\ p\ ?o)$ with $s_p =< S_s, \chi_s, T_s, D_s, Y_s >$. Then we compute the possible number of different values in the intersection of the multiset represented by s_p and o_p: $n = min(n_s, n_o)$ where $n_s = min(D_s, |\{x \in S_o \setminus S_s \mid \chi_s(x) > 0\}|)$ and $n_o = min(D_o, |\{x \in S_s \setminus S_o \mid \chi_o(x) > 0\}|)$ the collection summary is $n, \{?s, < S_s \cap S_o, min(\chi_s, \chi_o, 1), min(n_o, n_s), min(n_o, n_s), 1 >\}$

6.3 Optimization of distributed BGP query plans with an over-estimation

In this section, we showcase how to use a worst-case cardinality estimation to optimize the query plan of a distributed SPARQL query evaluator.

6.3.1 Query plan

We suppose that the evaluator has access to the four following primitives:

- $TP(t)$ takes a TP t and returns the mapping collection solution of t;
- $HashJoin(a, b)$ takes two terms a and b and returns the join of the mapping collections returned by a and b;
- $Broadcast(v, a, b)$ takes two terms a and b, stores the mapping collection returned by a into v and then evaluates b;
- $LocalJoin(a, v)$ returns the join of the mapping collection returned by a and the mapping collection stored into v.

The idea behing the $Broadcast(v, a, b)$ primitive is to first compute once the solution for a, then store it into the variable v that is sent to all computing nodes. This way, during the computation of b, if we come across a $LocalJoin(c, v)$ then each computing node holds the whole mapping collection v and the join can be done locally.

6.3.2 Query plan cost

We now present how to compute the cost of a query plan. We note $sol(a)$ the mapping collection return by evaluating the query plan a and $size(a)$ its size. We do not have access to the actual size of the mapping collection solution of a query plan but all query plans correspond to BGP and we know how to estimate their size: given a BGP, we compute a collection summary s and its estimated cardinality is just $size(s)$. Since our cardinality estimation is a worst case, our query plan cost estimation also constitutes a worst-case analysis.

Our query plan cost analysis is conditioned by three constants:

- `shuffleCost` we suppose that the cost of shuffling a mapping collection has a cost linear in its size with a coefficient `shuffleCost`,

- `broadcastCost` we also suppose that the cost of broadcasting a mapping collection is linear with a coefficient `broadcastCost`, however the *Broadcast* operation breaks when the mapping collection does not fit into RAM, that is why we have:

- `broadcastThreshold` that indicates the maximum size of a mapping collection that we can broadcast.

In our translation, the individual TP are all translated exactly once, we set their cost to 0.

For a $HashJoin(a, b)$ we need to compute a and b then shuffle a (resp. b) so that they are hashed on $dom(sol(a)) \cap dom(sol(b))$, this costs `shuffleCost`$\times size(a)$ (respectively `shuffleCost`$\times size(b)$) but only if a (resp. b) is not already correctly shuffled. We also need to materialize $HashJoin(a, b)$ –even if both components are already shuffled– which costs $size(HashJoin(a, b))$.

For a $Broadcast(v, a, b)$ we need to compute a, send the full set of solutions of a to each workers and then compute b; therefore the cost of $Broadcast(v, a, b)$ is `broadcastCost`$\times size(a)$ plus the cost of computing a and b. However $broadcast(v, a, b)$ can break if a is not small enough to fit into RAM, that is why we impose $size(a) <$ `broadcastThreshold`.

For a $LocalJoin(a, v)$ we need to compute a and join it with v –already computed– it costs $size(LocalJoin(a, v))$ plus the cost of computing a.

6.3.3 Optimizing the query plan

Given a BGP $(t_1, \ldots, t_n)$ the naive translation is to simply join $TP(t_i)$ using at each step a hash-join algorithm:
$HashJoin(TP(t_1), HashJoin(\ldots, TP(t_n)) \ldots)$.

The naive translation is often not the most efficient plan: we might have to materialize large intermediate results (that could be avoided by using a different join order) and it might also yield shuffles that could be avoided. For instance, if t_1 and t_4 bind $?a$, t_2 binds $?a$ and $?b$, t_3 binds $?b$ and $?c$, then the naive translation shuffles t_1 and t_2 on $?a$ then it shuffles $t_1 \bowtie t_2$ and t_3 on $?b$ and finally it shuffles $t_1 \bowtie t_2 \bowtie t_3$ and t_4 on $?a$ whilst the order $(((t_1 \bowtie t_4) \bowtie t_2) \bowtie t_3)$ implies one less (potentially costly) shuffle.

Furthermore, the $HashJoin$ algorithm is not always the most appropriated join algorithm [CNBA15]: when one mapping collection is large enough compared to the other then a broadcast-join (i.e. *Broadcast* the small dataset to all workers then do a $LocalJoin$) avoids a costly shuffle of the large dataset to the price of sending a small mapping collection to all workers.

Finally, when a triple t_i binds only one variable v, has a small number of solutions and at least one of the t_j, with $j \neq i$, contains v in its domain, it might be more efficient to broadcast-filter t_i. Broadcast-filtering t_i consists in computing $Broadcast(v_i, TP(t_i), P)$ where P is a term computing $t'_1 \bowtie \ldots t'_{i-1} \bowtie t'_{i+1} \ldots t'_n$ and $TP(t'_j) = LocalJoin(TP(t_j), v_i)$ if v is in the domain of t_j and $TP(t'_j) = TP(t_j)$ otherwise.

To find the query plan minimizing our cost estimation, we essentially enumerate possible plans: that filter-broadcast or not triples, with all possible join orders and for each join we consider the hash-join and the broadcast-joins. Our algorithm thus enumerates an exponential number of plans. However, with a few simple heuristics to cut down the number of plans and the heavy use of memoization, it performs well in practice as we now illustrate.

6.4 Extensions

We now present extensions to this method that we are considering for the future.

6.4.1 Combine with a cardinality estimation on nodes

For the moment, our method relies on the semantic relationship carried by the predicate. Relying on this relationship works well for some query shapes but it is not very efficient on some other such as star-shaped queries.

A *star-shaped query* is a BGP query in which there is a variable shared by all triple patterns, and this variable is the only variable shared between two TP. For instance let us consider the query `C3` from watdiv shown below.

```
SELECT ?v0 WHERE {
       ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
       ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/friendOf> ?v2 .
       ?v0 <http://purl.org/dc/terms/Location> ?v3 .
       ?v0 <http://xmlns.com/foaf/age> ?v4 .
       ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/gender> ?v5 .
       ?v0 <http://xmlns.com/foaf/givenName> ?v6 .
}
```

Our method guesses that there are at most $\sim$ 2.45 billions solutions while there are actually $\sim$ 42 million solutions to this query. Our prediction is thus far from the actual number (yet for a query with 6 auto joins two orders of magnitude is reasonable and can still be useful for plan generation).

The problem with our method on this query is that our method relies on the semantic relationship carried on the edges. In the above query `C3` our method rightfully guesses e.g. the number of gender ($?v5$), of given names ($?v6$), of age ($?v4$) per $?v0$ (that is why our guess that is not too far off) however our method does not precisely get how many distinct $v0$ have all the properties $likes$, $friendOf$, etc. One way to be much more accurate on such patterns would be to use methods of cardinality estimation based on nodes such as the Characteristics Sets [NM11].

The characteristics sets define a "type" for each node in an RDF graph based on which predicates label the outgoing edges of the node. With p different predicates there can 2^p types but in practice many different nodes have the same type and it suffices to merge very similar types to have a manageable number of types.

Without this merging of types, the cardinality estimation on star shapes query such as `C3` would be exact. Therefore it is not surprising that their cardinality estimation is very precise on such queries. However, as soon as the query is not star shaped, the precision decreases rapidly.

The characteristic set method is not a worst-case analysis for general queries but, even with the merging of similar types, it is a worst-case analysis for star-shaped queries. We plan in the future to use the characteristic set method to improve the accuracy on star shaped subpatterns appearing in queries. Since of the main drawback of our method is the accuracy on such patterns, we believe the global accuracy of our system would be much improved.

6.4.2 Consider subsets of columns for summaries

For the moment, the summaries described above are applied column by column, a collection summary is essentially the combination of the summaries of the multisets that are projection of a column. We could also consider collection summary as the combination of the summaries of the multisets that are projection of a subset of the columns. For a collection that has n columns, this would means 2^n summaries but n is generally very small. And when n is large we could limit ourselves to subsets that appear as a key of a join, or subset of less than k column (which means $\binom{n}{k}$ summaries).

This could also enable us to treat the case of missing values, i.e. collection summaries for collections where the domain of mappings are not all the same.

6.4.3 Cardinality estimation for the full μ-algebra fragment

The method we have presented so far can be easily translated for the conjunctive queries of μ-algebra (i.e. the μ-algebra programs composed of joins, filters and variables). While there are no reason to suppose that most μ-algebra programs are conjunctive, many of the μ-algebra programs we manipulated (and produced via a SPARQL translation) contained conjunctive parts where the optimization (and thus the cardinality estimation) was important.

Using the extension presented above to work on subsets of columns with potentially missing values, we can estimate the cardinality of the union, minuses and joins. The handling of several other operators is also trivial with collection summaries: renaming, duplication, removal of columns can all be performed with the same operations on summaries.

For the UDF operators (aggregation and mapping) we can not get a precise a list of values for the output columns since they are treated as black boxes. But we can say that the number of mappings after the operator is less than before the operator was applied (and exactly the same for mappings) and that untouched columns contain the same multiset of values. In aggregation, we can also say that the n-upplet containing the key column is unique.

The cardinality estimation for fixpoints is an hard and interesting problem. For instance, in SPARQL, the transitive closure on a predicate (such as $?a\ knows^+\ ?b$) can lead to a number of solutions that is anywhere between linear and quadratic in the number of triples with *knows*.

In our prototype we rely on a worst case analysis: we compute the number of distinct values each column can take and simply make the product (since two different mappings need two different value in at least one column). This analysis leads to very far off cardinality estimation in many scenarios but it handles well the cases of transitive closures where of the part of the closure is fixed.

Conclusion

In the present study, we introduced a worst-case cardinality estimation based on a new concept: collection summaries, which are extracted from statistics on the data. We have showed how to compute collection summaries for Basic Graph Patterns and how they can be used to estimate the cardinality of query answers.

With this cardinality estimator, we now have all the pieces for evaluating SPARQL queries using the μ-algebra. The next chapter will now assess the benefits of our approach in comparison with the state of the art.

Part III

Results & conclusion

CHAPTER 7

Comparision of our approach with the state of the art

In the last chapters we have presented all the building blocks necessary to write an efficient SPARQL *query processor. Using these building blocks we have implemented two* SPARQL *query evaluators based on this method. A natural follow-up question is to investigate the benefits of our approach in comparison with the state of the art.*

The results of this comparison will be split into three parts: in the first part, we will demonstrate that, from a theoretical perspective, our μ-algebra considers query execution plans for queries with recursion that other methods cannot consider.

*In a second part we will rediscover this theoretical comparison from an experimental perspective. We will run various query executors (*SQL *based, Datalog engines and native* SPARQL *query executors) and exhibit a query and a large class of graphs on which all query executors (except ours) performs at least quadratically in the number of nodes in the graph while our imple-*

mentation has a linear time complexity. We will then carry on with more queries and more graphs to show that our method can often outperform others in various scenarios.

As we showed in chapter 1 of the preliminaries, SparqlGX *is on par with other state of the art distributed query evaluators. It has a very good average query times but sometimes choose poorly its query execution plans. In the third part of this chapter, we will thus present how the cardinality estimation scheme presented in chapter 6 allows us to chose smarter query execution plans and can drastically improves the performance of* SparqlGX.

7.1 Theoretical comparison of μ-algebra bottom-up evaluation and other approaches on recursive queries

The optimized evaluation of SPARQL is well studied especially for the BGP fragment. The evaluation of recursive queries is also a well studied subject especially outside of the SPARQL language. As we have seen, SPARQL 1.1 introduced Property Paths that can contain recursive parts. The specific question of Property Path is a subject that has been much less tackled than the BGP fragment as noticed by [YGG15, BBC+17].

Complex queries (for instance containing recursion) are not are not always supported by query evaluators and when they are, they also are not well optimized. The gMark benchmark [BBC+17] compared various possible backends for graph databases. Their benchmark included SQL databases, Datalog engines, native graphs databases based on openCypher or SPARQL. With their benchmark, they found out that *"In the presence of recursion, [they] actually observed numerous failures on the majority of the studied systems"*. This observation arrives despite using relatively small graphs (less than 16 000 nodes) and relatively simple queries (union of regular path queries).

In this section, we will examine the various ways of handling those type of Regular Path Queries and will compare SPARQL–tailored query engines along with other query engines. In this section we propose to compare our approach to various lines of work that have tackled the subject from the more ad-hoc, tailored for SPARQL to very general approaches: Reachability joins, Waveguide, Regular Path queries, SQL extended for recursive queries and finally recursive Datalog.

For this section, we introduce a syntactic sugar to explain more precisely the difference between the plans of each formalism. Given two terms φ and ψ, both defining binary relations between the columns s and o, we define as syntactic sugar: $\varphi/\psi = \pi_{mid}\left((\rho_o^{mid}(\varphi)) \bowtie (\rho_s^{mid}(\psi))\right)$.

7.1.1 The relational algebra

The relational algebra introduced by Codd built the foundations of our work. Similarly to our work the relational algebra operates on a "set semantic" (while SQL operates on a "bag" semantic). The relational algebra differs from our work in several points. The two most salient ones being that the relational algebra works on a fixed domain and that it is not equipped with a fixpoint operator.

There exists a line of work (see e.g. [Agr88]) to extend the relational algebra with an operator α representing recursive queries. If φ is a relational algebra term defining a binary relation, $\alpha(\varphi)$ represents the transitive closure of φ. If this operator is sufficient to represent SPARQL, it does not allow for a plan space as large as our μ-algebra. Indeed with $\varphi_1/\varphi_2/\varphi_3$, the α-extended algebra does

not distinguish between the two following plans :

- the plan where we compute $\varphi = \varphi_1/\varphi_2$ before computing the transitive closure, which corresponds to compute $R_0 = \varphi$ then $R_{i+1} = R_i/\varphi$
- the plan where at each step we join with φ_1 or φ_2, i.e. $R_0 = \varphi$, $R_{2i+1} = R_{2i}/\varphi_1$, $R_{2i+2} = R_{2i+1}/\varphi_2$.

Furthermore, when producing a plan for a query such as $?a\ knows*\ bob$ this algebra represents the query as something similar to $\sigma_{o=bob}(\alpha(knows(?a, o)))$ where the $knows*$ is on

its own (without the fixed part *bob*) even though at the execution of the query we probably do not want to compute the whole $knows*$ relation but rather start from $?a = bob$ and progressively discover $?a$ such that $?a\ knows*\ bob$.

As we have seen, several RDF data engines use SQL as a backend for querying. And since SQL supports recursive queries through Common Table Expression (CTE), one way to handle recursive queries would be to just translate those with recursive CTE. However the support of recursive CTE varies very much between vendors:

- some explicitly do not support recursive CTE such as MariaDB / MySQL ;
- some have a partial support (e.g. they do not support cycles) such as Oracle ;
- such some support all type of CTE but consider CTE as optimization barrier (which prevents pushing selections into them) such as PostgreSQL and SQLite ;
- and finally some support CTE with optimization but this optimization is similar to a "Magic Set" approach which does not optimize all queries.

This approach was followed in [YGG13] but as the authors note and the authors of [BBC+17] also note, this method is not very successful as the databases engines are not tailored for this kind of use and most of them do not optimize recursive CTE. Some SQL vendors are therefore capable of expressing recursive queries but not optimize them.

7.1.2 SPARQL based method of optimization

SPARQL-algebra

The building blocks of the SPARQL-algebra are the Property Paths and the Triple Patterns. They can be combined with various types of joins and minuses but there is no recursivity in SPARQL outside of the Property Paths blocks. Which means it is not possible to constrain a recursive Property Path inside the SPARQL-algebra.

For instance if we have $Join(PP(?a\ knows^*\ ?b), PP(?a\ name\ John))$ there is no plan grouping the two PP into something like $PP(?a[name\ John]/knows^*?b)$. In practice an evaluator might chose to evaluate $PP(?a\ name\ John)$ and then choose the mappings solution and build from them to produce the mappings solution of both PP but it cannot represent this plan differently than joining the result of both PP.

The evaluators *ARQ* and *Virtuoso* are both based on the SPARQL algebra. *Virtuoso* has this quadratic behaviour (even though it is very fast) but does not seem to be affected by the order of the query. *ARQ* time complexity seems to depend on the order of the query but the way it is programmed makes *ARQ* unable to handle queries on graphs larger than 3000 nodes and we can therefore not really attest of the quadratic or not behaviour.

Automata

Given a Regular Path Query, it is possible to translate the regular expression into an automaton and use this automaton to evaluate the query.

One way to evaluate the query with an automaton and manipulate tuples (n_1, n_2, q) to indicate that there is a path from n_1 to n_2 labeled by q. At the beginning we either start

with (n, n, q_0) when q_0 is final or with with the sets (n_1, n_2, q_i) for each transition $q_0 \xrightarrow{l_i} q_i$ and $n_1 \xrightarrow{l_i} n_2$ in the graph. Then at each iteration we apply one transition.

It is possible to push further this idea using Brzozowski's derivatives, see e.g. [NS16] where they introduce Vertigo, a system capable of querying a graph with several billions nodes or to use weighted automata, see e.g. [ST09].

Waveguide

Waveguide[YGG15] introduced Waveguide Plans (WGP) allowing the optimized evaluation of PP. WGP mix together α-plans (which are plans based on the relational algebra) and FA-plans (which are based on automata). They show that their method subsumes both approaches, more precisely they show that both approaches force the associativity.

Given the regular expression $a/b/c$ the automata approach will compute the binary relation R recursively with $R_0 = \beta_s^o(N)$ and $R_{i+1} = ((R_i/a)/b)/c$ while the α-approach will compute $R_{i+1} = R_i/(a/b/c)$. In the mean time the WGP will also contain a plan that has the following computation: $R_{i+1} = (R_i/a)/(b/c)$. Notice that our approach also considers these three possible plans.

Given several PP, Waveguide translates each and then joins them. In comparison, our approach translates each PP into a μ-algebra term and joins them but before the optimization of the query. Which means the evaluation of PP can be interleaved. For instance given 3 PP: $?a\ knows*\ ?b$, $?b\ lastname\ Doe$ and $?b\ firstname\ John$ our optimizer proposes the plan that starts by finding the valid nodes $?b$ then finds the $?a$ reachable from these $?b$.

Waveguide being not publicly available, we asked the author to provide us with a copy to include it in our benchmark but we did not receive an answer. Therefore Waveguide is not included in our benchmark.

Reachability Joins

Reachability joins are the basis of the Ferrari system [GBS13] which brought fast Property Paths into RDF-3X. The idea is that given two terms A and B, both defining binary relations, their reachability join (noted $A \bowtie_R B$) is equal to the join of A with the transitive closure of B.

It is equivalent to $A \bowtie \alpha(B)$ but, in this system, B and A have to have one common variable and B corresponds to a set of pre-determined predicates.

The Ferrari system is very fast for star over patterns for which it pre-computed reachability indexes but it is incapable of handling the computation of arbitrary stars (which can appear in SPARQL). Furthermore, from an expressiveness point of view, $A \bowtie_R B = A \bowtie \alpha(B)$ and therefore it suffers the same problem with associativity as α-plans: it cannot express plans such as $\mu(X = \beta_s^o(N) \cup (X/A)/(B/C))$.

7.1.3 Magic Sets for Datalog

A major line of research focuses on tackling recursive queries in Datalog. The optimization and fast execution of Datalog engine on graph data is a challenge due to the expressive power of Datalog and its logic-based form [CGT89]. One classical way to optimize Datalog engines is to use the "Magic Set" technique [BMSU86, SZ86].

The *magic set* optimization [GdM86] is capable of pushing selection but only on fixed terms. This technique uses *adornments*. An adornment defines for each argument of a relation whether it is *free* (f) or *bound* (b). For a n-ary relation, there are thus 2^n adornments.

Intuitively, when we write $R(X, Y)$? in a Datalog solver, we expect to find the couples of (X, Y) that validates R. In practice, there are many cases where e.g. we know X and we want to get the corresponding Y (which would be written e.g. $R(42, Y)$.

A relation R is adorned by $^{v_1 \dots v_n}$ (noted $R^{v_1 \dots v_n}$ when R is used in contexts where the i-th argument is known for the i such that $v_i = b$ and either known or unknown for the i such that $v_i = f$. For instance the use of $R(42, Y)$ will be adorned by bf but it could also be adorned with ff.

Note that each adornment will be valid depending on the use, if we issue two commands $R(42, Y)$?, resp. $R(X, 42)$?, they can be adorned with the bf, resp. fb but the only adornment of R valid for both is ff.

The goal of the "Magic Set" technique is to find adornments for all relations where most variables are *bound* because when variables are bound, we don't have to materialize the full set of solutions to a relation but limit ourselves.

Note for a Datalog program there often exist a variety of possible adornments. One way to adorn (and therefore evaluate) Datalog programs is *left-to-right*. In such a situation, we start from the top. We know the adornment of the output. Then given a conjunctive rule $R(X_1, \dots, X_n) : -R_1(X_1^1, \dots, X_1^{k_1}), \dots, R_\ell(X_\ell^1, \dots, X_\ell^{k_\ell})$ and an adornment for R. We note B the set of bound variable. At first B is the subset of $\{X_1, \dots, X_n\}$ that is bound by the adornment. Then the adornment of R_i is given by which variables of $X_1^1 \dots X_{k_1}^1$ are in B and we add to B all the variables of R_1 and we iterate to get an adornment for each R_i.

Example Let us consider the following example corresponding to the Datalog translation of our example SPARQL query.

SPARQL query

```
  SELECT ?x ?y
{
  ?x :named :axel .
  ?x :knows+ ?y .
}
```

in Datalog (supposing a relation per predicate) the program is:

```
KTrans(X,Y) :- Knows(X,Y)
KTrans(X,Y) :- KTrans(X,Z), Knows(Z,Y)
BGP(X,Y,Z) :- Named(X,Z), R2(X,Y)
Sol(X,Y) :- BGP(X,Y,axel)
```

We can adorn the program :

$$\begin{array}{rcl} KTrans^{bf}(X,Y) & :- & Knows^{bf}(X,Y) \\ KTrans^{bf}(X,Y) & :- & KTrans^{bf}(X,Z), Knows^{fb}(Y,Z) \\ BGP^{ffb}(X,Y,Z) & :- & Named^{fb}(X,Z),\ KTrans^{bf}(X,Y) \\ Sol(X,Y) & :- & BGP^{ffb}(X,Y,axel) \end{array}$$

And obtain the following "magic" program:

$$\begin{array}{rcl} MagicKTrans(X,Y) & :- & MagicKTrans(X,Z), Knows(Y,Z) \\ MagicKTrans(X,Y) & :- & Named(X,axel), Knows(X,Y) \\ MagicBGP(X,Y) & :- & MagicKTrans(X,Y) \\ MagicSol(X,Y) & :- & MagicBGP(X,Y) \end{array}$$

With this rewriting the Datalog evaluation will only manipulates X that are named *axel* in $KTrans$ and thus not compute the set of pairs of people linked by a chain of *knows* but only a subset where the first is *named axel*.

Note that for this adornment to work we need simultaneously to have the Datalog engine to adorn $Named$ before $KTrans$ in the Sol rule and since X is the first variable of $KTrans$, we need $KTrans$ to be the "right" translation of $knows*$ (in opposition to the "left" translation which would be the equivalent $KTransRev(X,Y) : -Knows(X,Y)$, $KTransRev(X,Y) : -Knows(X,Z), KTransRev(Z,Y)$).

In our benchmark we used the well-known *Datalog* developed by *Ramsdell*[1] tool, the *dlv* engine and the *Vlog* system[UJK16].

7.2 An experiment for μ-algebra bottom-up with a recursive query

In this section we translate our theoretical comparison into an experimental one. To that end we compare our prototype using the techniques presented in this paper with other existing RDF or SQL stores in the context of SPARQL with recursive Property Paths and we show that our prototype do deliver the theorized complexity and thus outperforms other approaches.

7.2.1 Query used

Our first query consists in the query to retrieve the pair of nodes x and y such that x is linked to *bob* with an edge *named* and is linked to y through a path composed of edges *knows*. There are several version of this query:

- the SPARQL version of the query is given in the figure 7.1;
- the SQL version of the query had to be slightly adapted for each SQL store but we give the Postgres version in figure 7.2. These hand-translated SQL versions suppose that the data is stored without prefixes with one table "subject-object" for the predicate `<http://example/named>` and one for `<http://example/knows>`) with indexes;
- we also considered a Datalog version of the query presented in figure 7.3. Notice that the Datalog query also contains the graph that is partially omitted for obvious space reasons. In the Datalog version the query is simplified to its bare minimum: we just asks whether there is a path from any node to 42 an the path is either a reflexive, transitive closure or just a transitive closure.

[1] `http://www.ccs.neu.edu/home/ramsdell/tools/Datalog/`

```
SELECT ?x ?y
{
  ?x :named :bob .
  ?x :knows* ?y .
}
```

Figure 7.1: SPARQL version of the main query

```
WITH RECURSIVE knows_star AS
(
      SELECT *
      FROM knows
   UNION ALL
      SELECT
        knows_star.s AS s,
        knows.o AS o
      FROM knows_star, knows
      WHERE knows_star.o=knows.s
)
SELECT * FROM knows_star k, named n
WHERE n.o=name_42' and k.o=n.s;
```

Figure 7.2: SQL version of the main query

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z),edge(Z,Y).
sol :- path(_,42).
sol?
```

```
path(X,X) :- edge(X,_).
path(X,X) :- edge(_,X).
path(X,Y) :- path(X,Z),edge(Z,Y).
sol :- path(_,42).
sol?
```

Figure 7.3: Datalog version of the main query

7.2.2 Graphs

We considered several graphs: a chain of n nodes and a simple loop. In both cases `<http://example/nam` is linked to `<http://example/name_`$i+1$`>` which is represented is figure 7.4 and in the loop we additionally connect the first and last nodes.

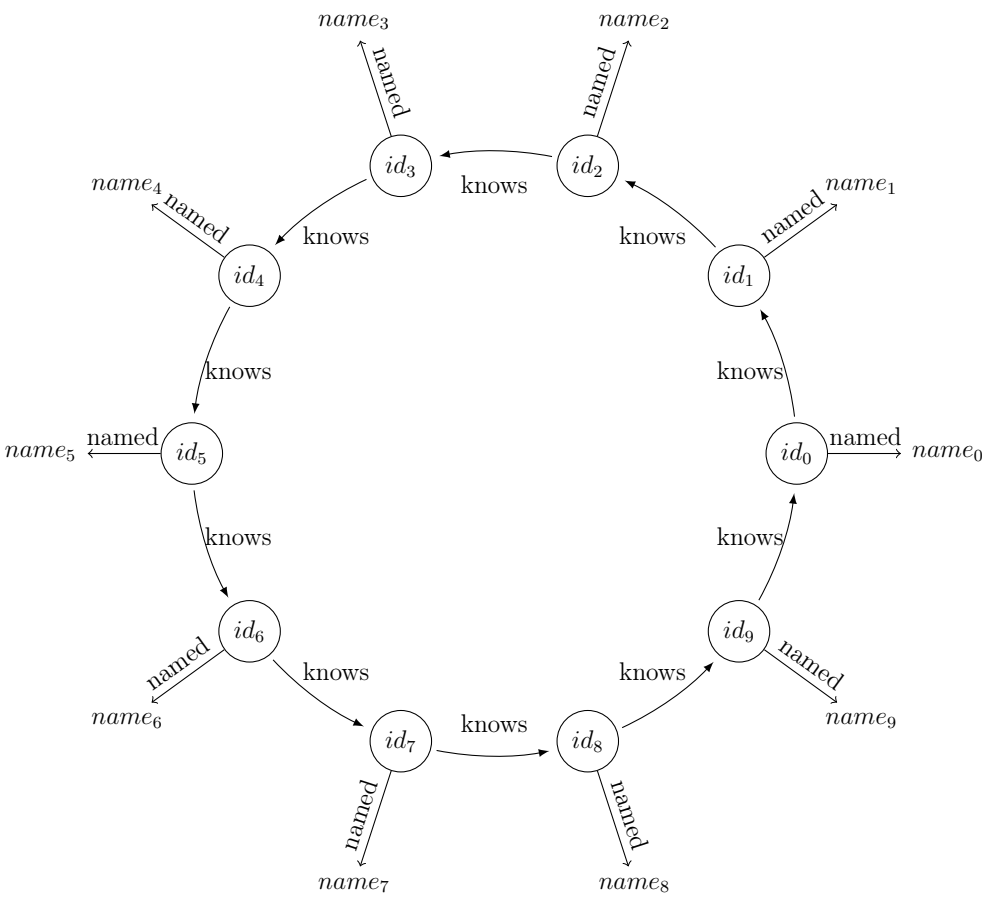


Figure 7.4: A loop graph of size 10 composed of *knows* with for each node an extra information of a *name*

7.2.3 SQL

SQL is based on the relational algebra. Both have been extensively studied either for themselves or in the context of SPARQL query evaluation. However using SQL for the optimization of SPARQL has been not been very successful even without considering PP [EM09, NM11].

Even if they are not supported by all SQL stores, recursive queries do exist in SQL. Vendors introduced in their product various extensions allowing some kind of recursion and in its 99 version, SQL introduced Common Table Expressions (CTE) allowing recursive queries. Recursive CTE allow for a very broad kind of recursive queries, broader than what is allowed in the α-extended. However not all SQL databases support all these features (e.g. MySQL does not even support recursive CTE) and vendors generally consider CTE as "optimization fences". We benchmarked several SQL databases in our benchmark comparison.

Postgres 10.2 & SQLite 3.24 Postgres and SQLite both consider Common Table Expressions (CTE) as optimization barriers. It is thus not surprising that they actually compute the whole knows_star relation before selecting pairs (s, o) such that $s = name\_42$.

Our tests do find this quadratic behavior (see fig 7.5). Even PSQL with indexes on s and o for the table *knows* go above the second barrier for a graph with 1000 nodes.

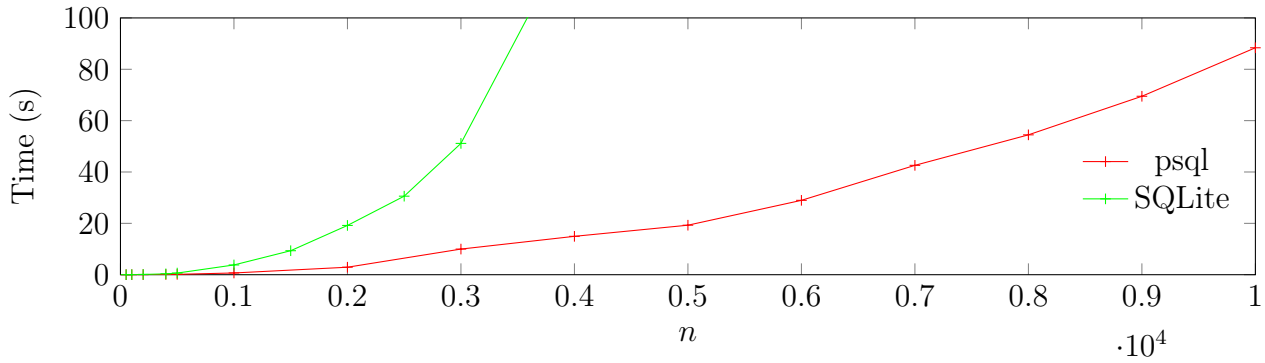


Figure 7.5: Time for Postgresl and SQLite

MySQL As of today, MySQL does not supports CTE and was, therefore not tested. MariaDB (the recent fork of MySQL) supports CTE as beta and does not optimize them.

7.2.4 Native SPARQL

ARQ 3.1.10

Jena is an Apache framework for building semantic web and linked data application. ARQ is the SPARQL query evaluator of Jena and it has full SPARQL 1.1 support.

ARQ does optimize queries but its optimizer does not optimize all SPARQL 1.1. Figure 7.6 reports on the time spent evaluating queries on two queries ARQ_1 and ARQ_2. ARQ_1 is our benchmark query while ARQ_2 is the same query but where we exchanged the order or the PP. As the figure shows ARQ sometimes also has the quadratic behaviour (depending on the order the PP in the input query).

ARQ, even with the "good" ordered query ARQ_1, breaks with a stackoverflow when evaluating paths with around 3000 nodes .

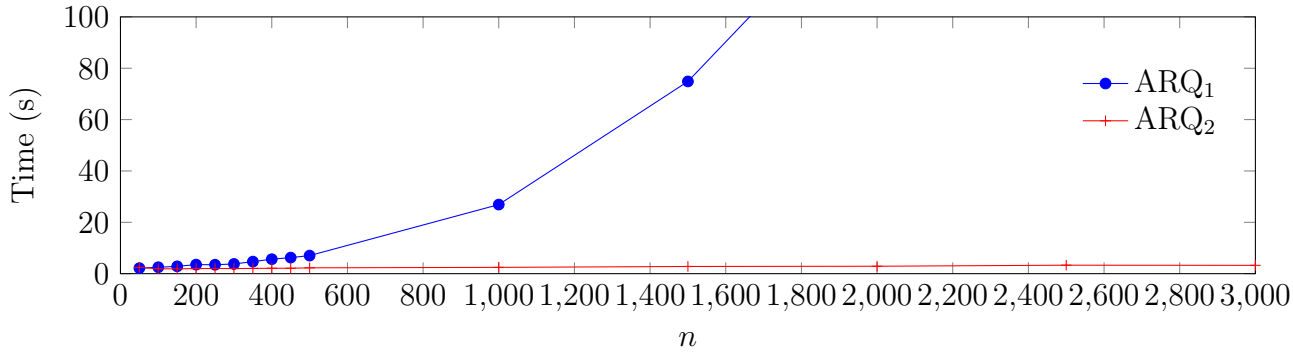


Figure 7.6: Time for ARQ depending on triple order

Virtuoso 7.2.4.2

Virtuoso also optimizes queries and is pretty fast (the fastest beside our prototype) but as we see in figure 7.7 it still exhibits a quadratic behaviour. Virtuoso does not seem to depend on the order of PP in the query.

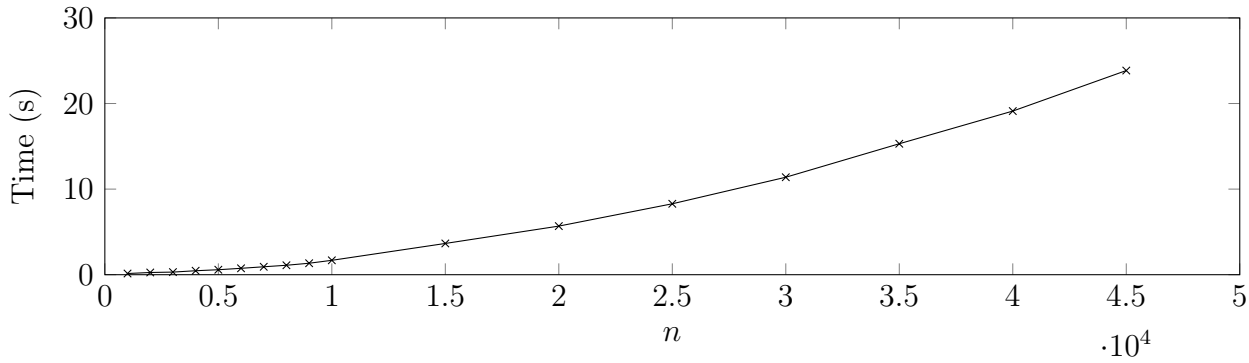


Figure 7.7: Query evaluation time for Virtuoso

7.2.5 Datalog

Curiously none of the tested Datalog solver (Vlog fetched july 2018, Ramsdell 2.6, DLV of december 2012) was able to respond quickly despite a relatively simple derivation path: `sol :- path(_,42) :- path(42,42) :- edge(42,_) :- edge(42,43)` and only DLV with the optimization parameter `-brave` was able to answer quickly in the simplest case (no cycle in the graph).

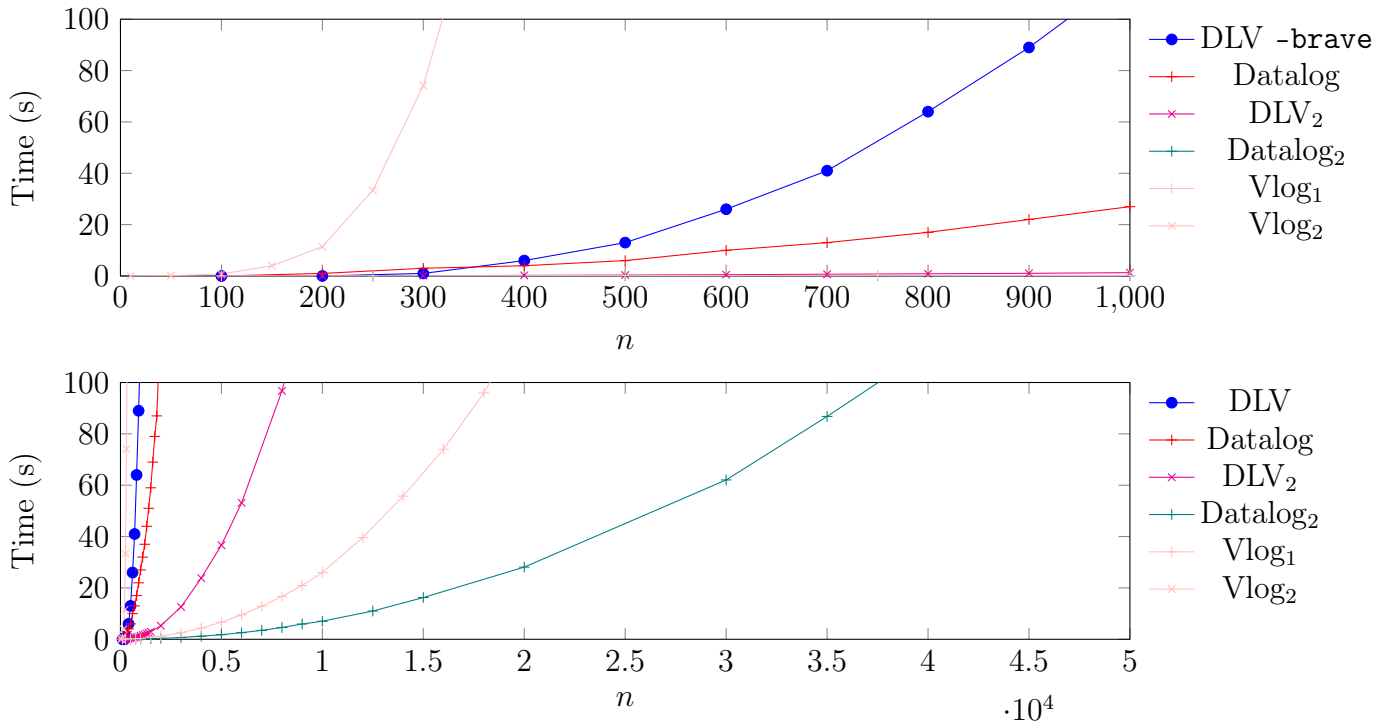


Figure 7.8: Time for Datalog solvers

7.2.6 Our prototype

The query of our benchmark is similar to the example query of section 3.5. Its naive translation is:

$rpe(?a\ knows*\ ?b) \bowtie rpe(?b\ named\ name\_42)$ where
$rpe(?a\ knows*\ ?b) =$
$\rho_s^a\left(\rho_o^b\left(\mu(X = \beta_s^o(N) \cup \pi_m\left(\rho_o^m(X) \bowtie \rho_s^m\left(\pi_p\left(\sigma_{p=knows}(T)\right)\right)\right)\right)\right)$ and $rpe(?b\ named\ name\_42) =$
$\rho_s^{?b}\left(\pi_o\left(\sigma_{o=name\_42}(T)\right)\right)$. After optimization it is: $\mu(X = init \cup \pi_m(K \bowtie \rho_{?a}^m(X)))$ where
$init = \beta_{?b}^{?a}\left(\pi_o\left(\sigma_{o=name\_42}(\rho_s^{?b}(T))\right)\right)$ and
$K = \pi_p\left(\sigma_{p=knows}(\rho_o^m\left(\rho_s^{?a}(T)\right)\right)\right)$.

We verify experimentally that the execution time for our prototype is linear. Our results are presented in figure 7.9 (note that the Y axis goes from 0 to 4×10^6 and not to 10^8 as in other figures and the X axis goes up to 10^6).

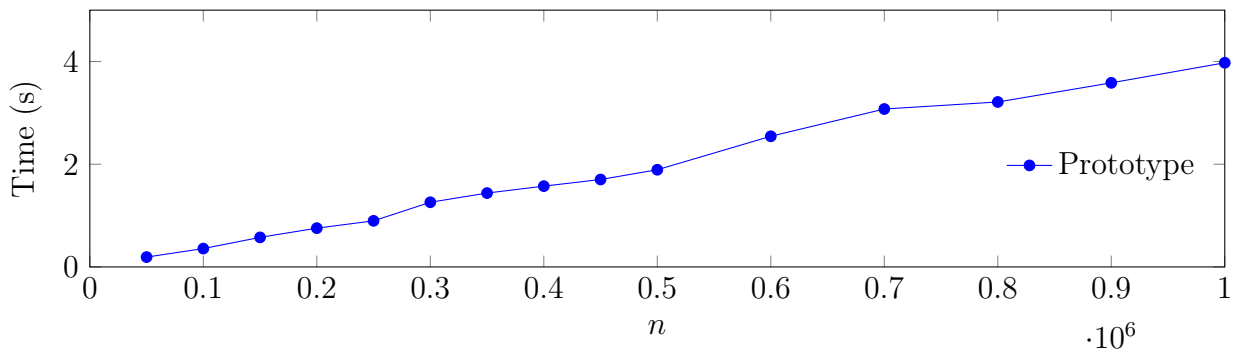


Figure 7.9: Time for our prototype

7.3 Pushing the benchmark further

In order to get a more precise picture of how much our method improves on existing methods, we wanted to use a benchmark making use of linear recursion. The simplest use is the Union of Conjunctive Regular Path Queries. In the TASWEET paper, the benchmark is composed of a single disjunctive RPQ on a single graph, the recent gMark benchmark includes what they call "recursive" queries through a custom extension of SPARQL that allows to repeat Property Paths but only a limited number of times.

That is why we devised an experiment to compare the tools performing the best in our first tests with our prototype based on the μ-RA. We designed our benchmark to have simple queries on large graphs.

Competitors In our second setup, we removed ARQ and SQLite, which performed very poorly in our first setup. We also removed Virtuoso that was the second top performing in our first benchmark (after our prototype), as it is limited to a very specific type of recursive queries (in our benchmark, only 2 of the 10 queries were accepted by virtuoso).

Queries As randomly generated queries tend to have too few or too many solutions, we selected 10 queries containing between one and three fixpoints and presenting diverse behaviours on query engines. These queries are presented in Table 7.1; the number of solutions to our ten queries are presented in table B.1 and in figure 7.10.

Graphs We also generated randomly graphs with varying number of nodes n (with n logarithmically spaced between $n = 100$ and $n = 50 \times 10^6$). In these graphs there were 5 labels $P1$ to $P5$. The edges were generated randomly such that the label Pi corresponds to $2n(1 - i/5) + 20$ edges. This allows for $P1+$ to be very large (roughly n^2)

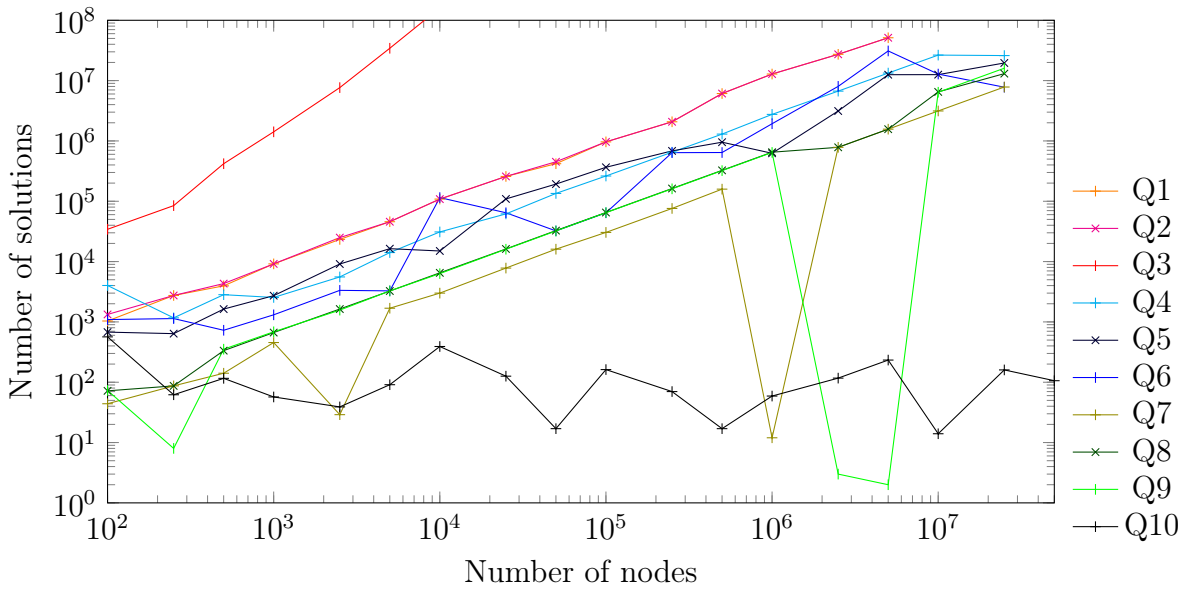


Figure 7.10: Number of solutions for each query and each graph size in our second benchmark.

while $P5$ is always very selective (with 20 edges). In order to ensure that all queries had solutions, for each label Pi, we also added edges $N0 \xrightarrow{Pi} r_1$, $r_2 \xrightarrow{Pi} N0$ and $N0 \xrightarrow{Pi} N0$ (for two random nodes r_1 and r_2).

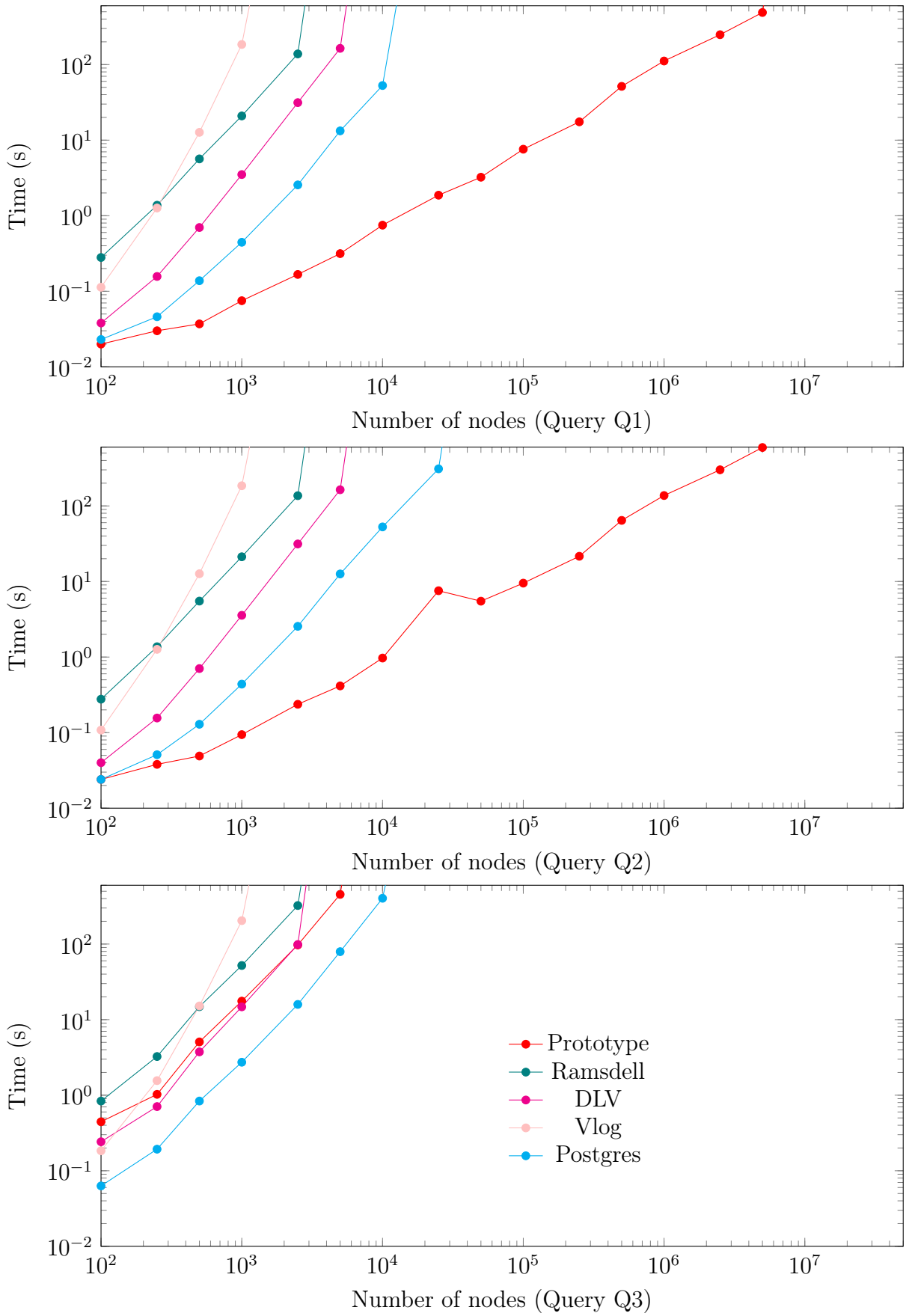
Setup For each of the graphs and each of the queries we measured the query time (it thus does not include the time to load the data). These times include the time to optimize the query. The complete results are shown in Table B.2 of the appendix (times are in milliseconds) and in the plot of figure 7.11. As for the first benchmark, we gave Postgres indexes for a fast access to both sides of edges.

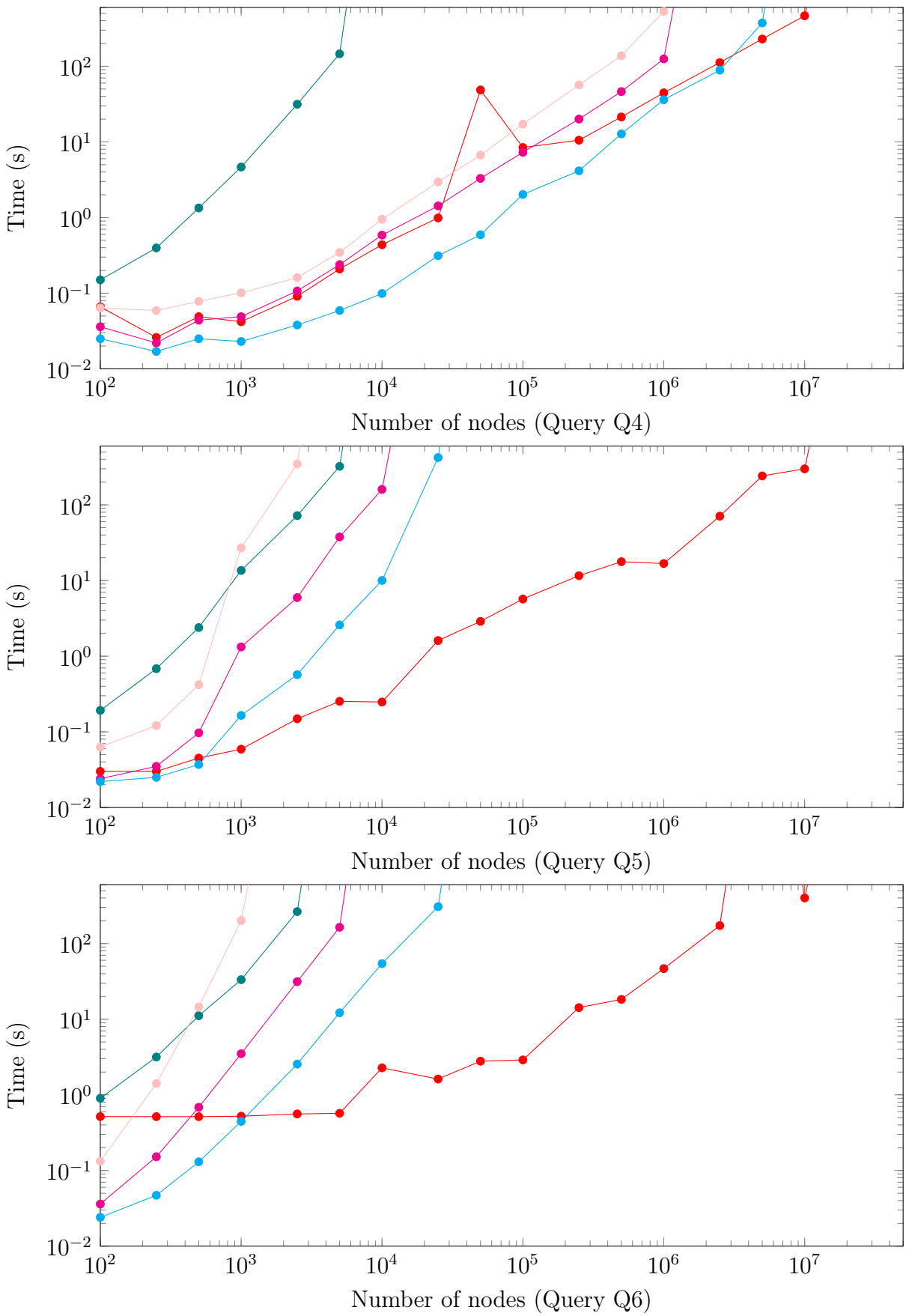
Q1 and Q2 Q1 and Q2 have roughly the same solutions. This is not necessary (the queries are not equivalent) but it is due to the fact that $P1+$ captures almost all pairs of nodes while $P5+$ is very close to $P5$. Postgres has the same behaviour on both queries: by looking at the plan the Postgres optimizer selected, we see that it computes $P1+$ and $P5+$ or $P5$ separately and then joins the results. It is thus not very surprising that both queries take roughly the same time since the computation of $P5+$ is fast (as it is very small).

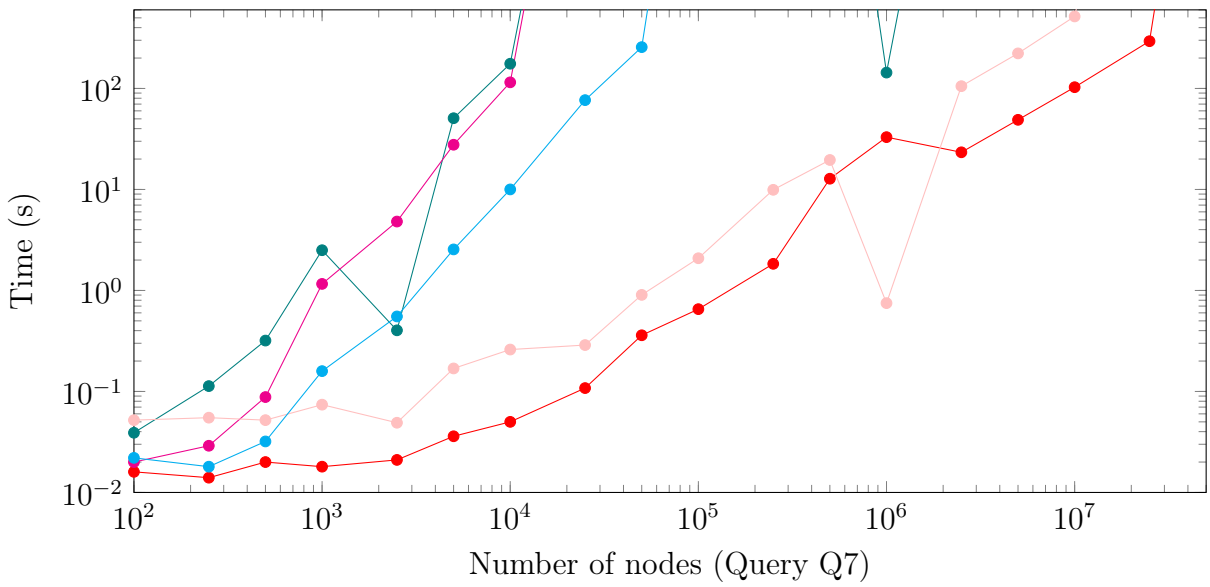
On the opposite, our prototype is slower on $Q2$. The query plan for $Q2$ selected by our prototype is a merged fixpoint starting from $P1/P5$ and appending/preppending $P1$ and $P5$. For $Q1$, our prototype starts from $P1/P5$ and just tries to prepend $P1$ which takes less time, hence the time difference between $Q1$ and $Q2$. However, our prototype still is the *fastest* on both queries as it does not try to materialize the full $P1+$.

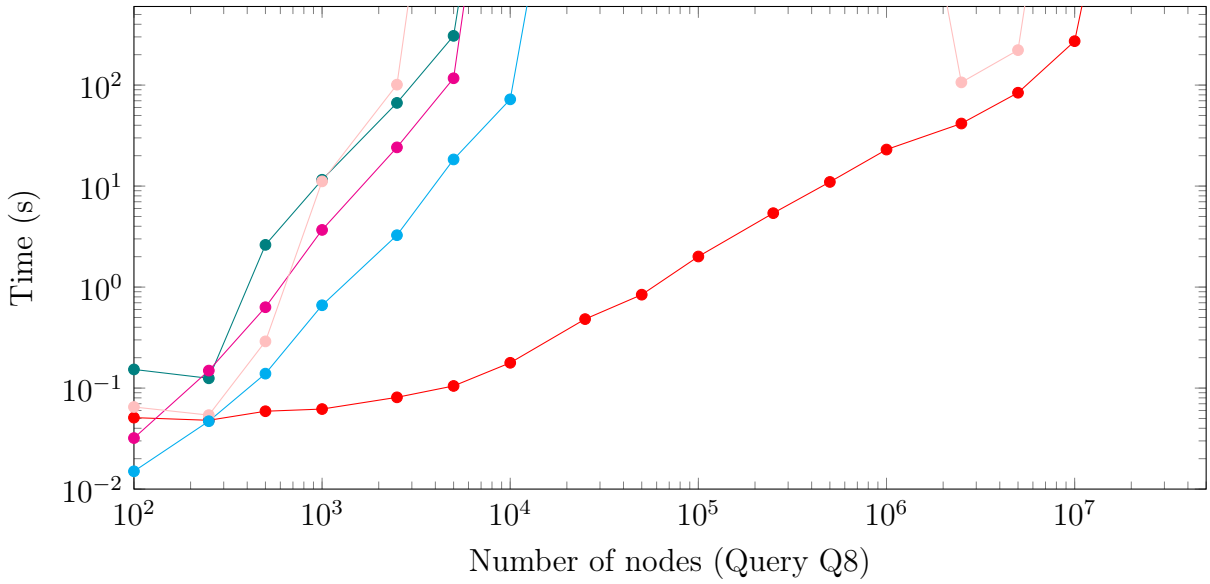
Q3, Q4, Q5 On these queries, our prototype starts from a set of valid triples $?a, ?b, ?c$ then discovers new solutions. These plans are optimal in the sense that all partial solutions are solutions of the query but our prototype is not always the fastest.

Q3 Postgres is the *fastest* on Q3. Q3 has a lot of solutions ($\sim 10^6$ for $n = 1000$ and $\sim 10^8$ for $n = 2500$). Postgres evaluates this query by computing all the transitive closures


Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2} 10^{3} 10^{4} 10^{5} 10^{6} 10^{7}
Number of nodes (Query Q1)
Number of nodes (Query Q2)
Number of nodes (Query Q3)
Prototype
Ramsdell
DLV
Vlog
Postgres


Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2} 10^{3} 10^{4} 10^{5} 10^{6} 10^{7}
Number of nodes (Query Q4)
Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2} 10^{3} 10^{4} 10^{5} 10^{6} 10^{7}
Number of nodes (Query Q5)
Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2} 10^{3} 10^{4} 10^{5} 10^{6} 10^{7}
Number of nodes (Query Q6)


Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
Number of nodes (Query Q7)


Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
Number of nodes (Query Q8)

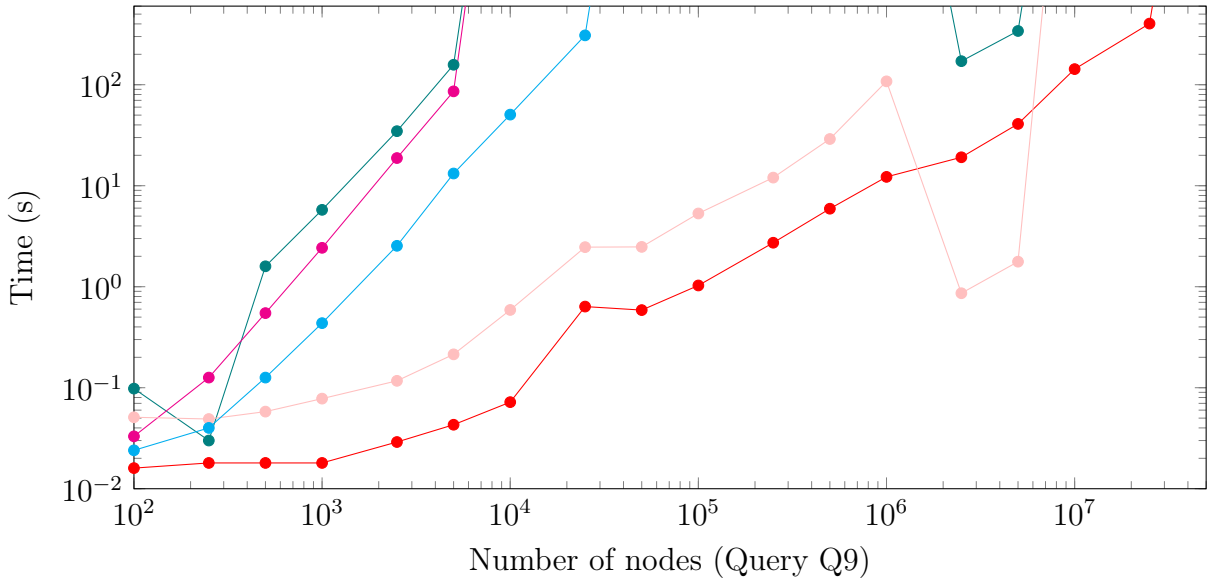

Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
Number of nodes (Query Q9)

Table 7.1: Our queries: Pi are labels, Ni are nodes

Name	Query		
Q1	$?a$	$(P1+)/P5$	$?b$
Q2	$?a$	$(P1+)/(P5+)$	$?b$
Q3	$?a$	$(P1+)/P2$	$?b$
	$?b$	$P3+$	$?c$
Q4	$?a$	$(P4\|P5)+$	$?b$
	$?b$	$P3+$	$?c$
Q5	$?a$	$P2+$	$?b$
	$?a$	$P4+$	$?c$
	$?a$	$P5$	$N0$
Q6	$?a$	$P1+/P2$	$?b$
	$N0$	$P3+$	$?b$
Q7	$N0$	$P1/(P2+)$	$?a$
Q8	$N0$	$(P1+)/(P2+)$	$?a$
Q9	$N0$	$P1/(P1+)$	$?a$
Q10	$?a$	$(P4+)/(P5+)/(P3+)$	$?b$

and then joining them. The plan of Postgres is not more optimal than ours; however, Postgres is very efficient to retrieve data and join huge quantities of partial results. It thus gains on our prototype as long as the individual transitive closures are not much larger than the final result (which is what happens here and not on $Q1$ and $Q2$ with $P1+$).

Q4 On Q4, we see that many of the evaluators perform roughly the same. Despite the fact that Postgres has access to indexes and and is very efficient, our prototype is slightly faster on large graphs our prototype, this can be attributed to the optimality of our plan that starts only from valid triplets.

Q6, Q7, Q8 & Q9 These queries include a constant node ($N0$) from which our prototype will start building solutions. It is therefore not surprising that our prototype outperforms the other systems.

Q6 On Q6, all systems except our prototype seem to materialize $P1+$ which takes a lot of time. Systems fail on $Q6$ as they fail on $Q1$ and $Q2$.

Q7, Q8 & Q9 all these queries are queries where the Datalog engines outperform Postgres. This is due to the triggering of the magic set algorithm that avoids the full materialization of the $P1+$ relation. Despite this use of the magic set, these query engines perform an order of magnitude slower than our prototype except on a few graphs where the number of solutions is very small.

Q10 This query includes three fixpoints on $P3$, $P4$ and $P5$. For each of these predicates Pi, the size of $Pi+$ stays linear in the size of Pi (contrary to e.g. $P1$ for which the size of $P1+$ grows quadratically with the graph size). This is why query engines such as Postgres perform well. Our prototype starts with pairs $(?a, ?b)$ that are solutions to $?a\ P4/P5+\ ?b$ with a fixpoint; then it computes solutions to $?a\ P4/P5+/P3\ ?b$

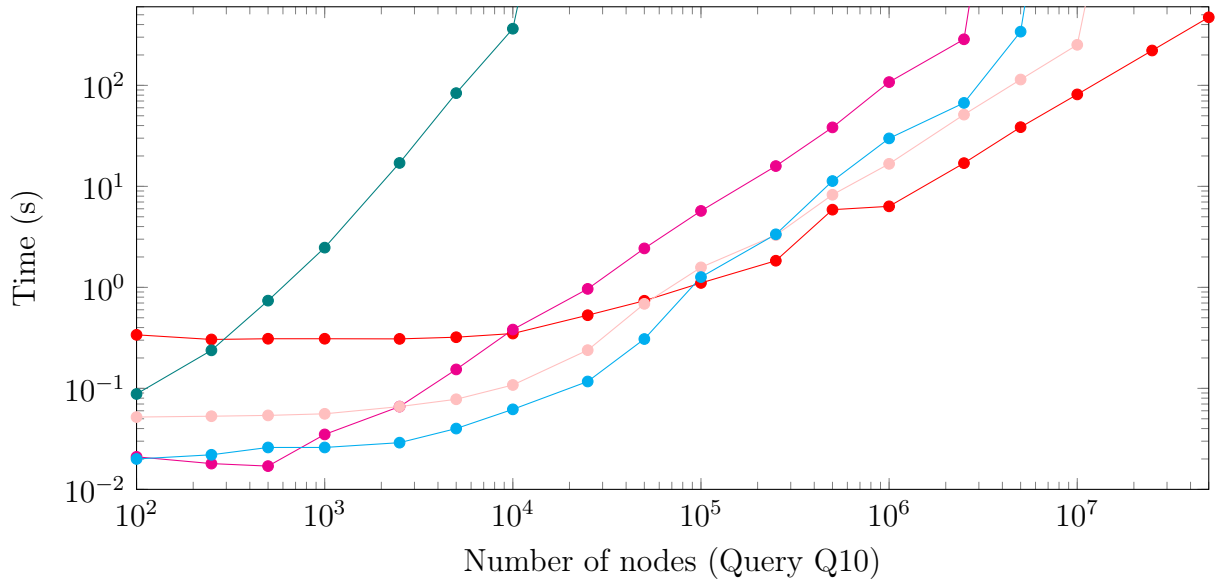


Figure 7.11: Query time (in seconds) as function of the size of the graph for each of our 10 queries.

(which is a valid solution) then, in a second fixpoint, grows this fixpoint by adding a $?a$ or a $?b$.

Synthesis

In all queries, our system performs well, beating the Datalog engine on nearly all the graphs for all the queries. Furthermore, in the majority of the queries we considered, our system outperformed Postgres, often by several orders of magnitude. For the setups where our system is outperformed by Postgres, the difference is less important and can be imputed (in a large part) to the sheer performance of Postgres plan execution and not its plan space. Indeed, in Postgres, we stored triples as integers with indexes on both sides of edges while our prototype stores triples using a simple scheme and plain text files.

Sources to run the benchmark can be found on `https://gitlab.inria.fr/tyrex-public/mu-RA` and sources to our prototype can be found on `https://gitlab.inria.fr/jachiet/musparql`.

7.4 Efficiency of distributed SPARQL query evaluators

The efficiency of query evaluators can be assessed for a wide range of different scenarios and along a wide set of metrics. This efficiency is measured on several runs of a batch of queries on a given dataset. Theses measurements therefore depend on the scenario (the dataset and queries used). The efficiency is often considered using the metric "mean time to answer a query". In this section we will therefore base our comparison on this metric but it is also possible to include other metrics which is something that we investigated further for distributed SPARQL query evaluators in [GJGL17].

Handling simple queries (e.g. BGP) on very large graphs is a subject that has been attempted, CliqueSquare, PigSPARQL, S2RDF and RYA are example of stores capable of

handling large datasets.

We report here on the empirical comparison published in [GJGL17] (in which I participated) comparing several open source and state of the art SPARQL query evaluators. This comparison uses two popular benchmarks for SPARQL: LUBM and WatDiv. These benchmarks contain a set of queries (Q1-14 for LUBM and F1-5, C1-3, L1-5 and F1-5 for WatDiv) along with a dataset generator. The figure 7.16 shows the query time for LUBM10k that is LUBM with a dataset of 1.38 billion triples and for WatDiv1k that is WatDiv with a dataset of 109 millions triples. As we can see on the figure the comparison is mixed: some evaluators perform well on average but poorly on some queries (such as RYA); others perform well but cannot handle very large datasets (such as 4store). As one can see, SparqlGX, which is a SPARQL query evaluator that we developed [GJGL16a], performs well on average. However, as noted by [AGRL17], plans selected using simple statistics by SparqlGX are sometimes suboptimal which will motivate the need for more diverse plans (and thus our algebra) along with better statistics (and thus our cardinality estimation).

4store CliqueSquare CouchBaseRDF CumulusRDF PigSPARQL RDFHive RYA S2RDF SDE SPARQLGX

Evaluator	WatDiv1k	Lubm1k	Lubm10k
SparqlGX	∅	∅	∅
CliqueSquare	F1,2,5 & S2,3,5,6,7 Parser	∅	∅
CouchBaseRDF	C3 Failure	Q2,14 Failure	Pre-processing Failure
RDFHive	∅	Q2 Timeout	Q2 Timeout
RYA	C2,3 Timeout	Q2 Timeout	Q2 Timeout
S2RDF	∅	∅	Pre-processing Failure
SDE	∅	Q2 Timeout	Q2 Timeout

Figure 7.12: Failure Summary for problematic evaluators.

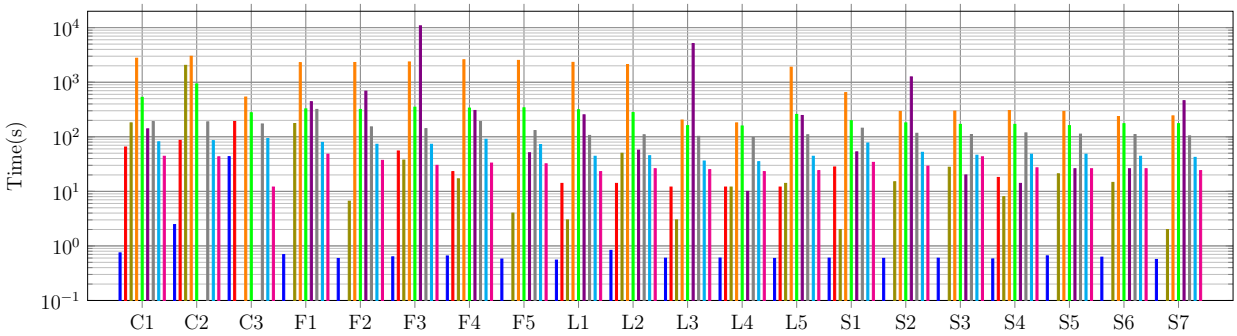


Figure 7.13: Query Response Time with WatDiv1k.

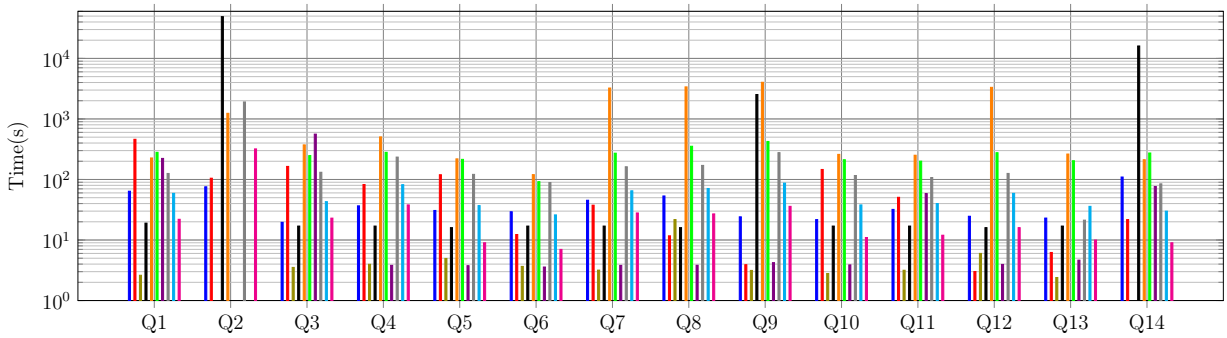


Figure 7.14: Query Response Time with Lubm1k.

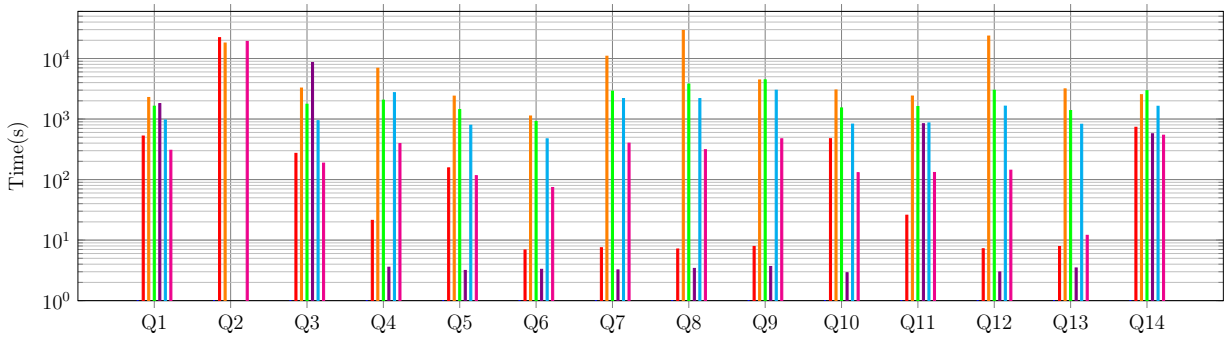


Figure 7.15: Query Response Time with Lubm10k.

Figure 7.16: Loading and response time with various datasets.

7.5 Experimental of SparqlGX with our cardinality estimation

As we have seen in the last section, SparqlGX is a competitive distributed SPARQL query evaluator. We have implemented the cardinality estimation scheme described in chapter 6 into the SparqlGX evaluator and in this section we report on our benchmark testing the impact of our cardinality estimation on the performance. Since SparqlGX already has optimization strategies based on statistics, the SparqlGX-core can be used as a common basis of comparison to benchmark the various optimization modules. The source code of our complete prototype is available online with SparqlGX.

7.5.1 Related Work

Estimating the number of solutions for a query has long been viewed as a key element in the optimization of queries and it is a well-studied problem in the relational world [PSC84]. Various techniques successful in the relational world (e.g. histograms [Ioa03, OR00]) have been less successful for the semantic web [EM09, NM11]. The main reasons for that is the heterogeneous and string nature of RDF [NM11] and the fact that SPARQL queries usually contain a lot of self-joins that are notoriously hard to optimize [PT08].

Various works have tackled the specific issue of cardinality estimation for SPARQL. A first line of work [SSB+08] introduced the "selectivity estimation" now in use in several SPARQL evaluators [ZYW+13]. Variants of this method have been implemented in popular SPARQL query evaluators (e.g. in RDF-3X [NW08]).

A second line of work [KRA17] takes as input an actual schema and produces an optimized query plan based based on information extracted from the schema.

A third line of work [NM11, GN14] tries to derive the implicit schema of an RDF graph by fitting nodes into characteristic sets, or by summarizing [GSMT14] the graph into large entities. These approaches are the closest in spirit to the approach that will present but they tend to focus on finding an implicit schema type for nodes while our approach is more focused on finding an implicit schema for edges.

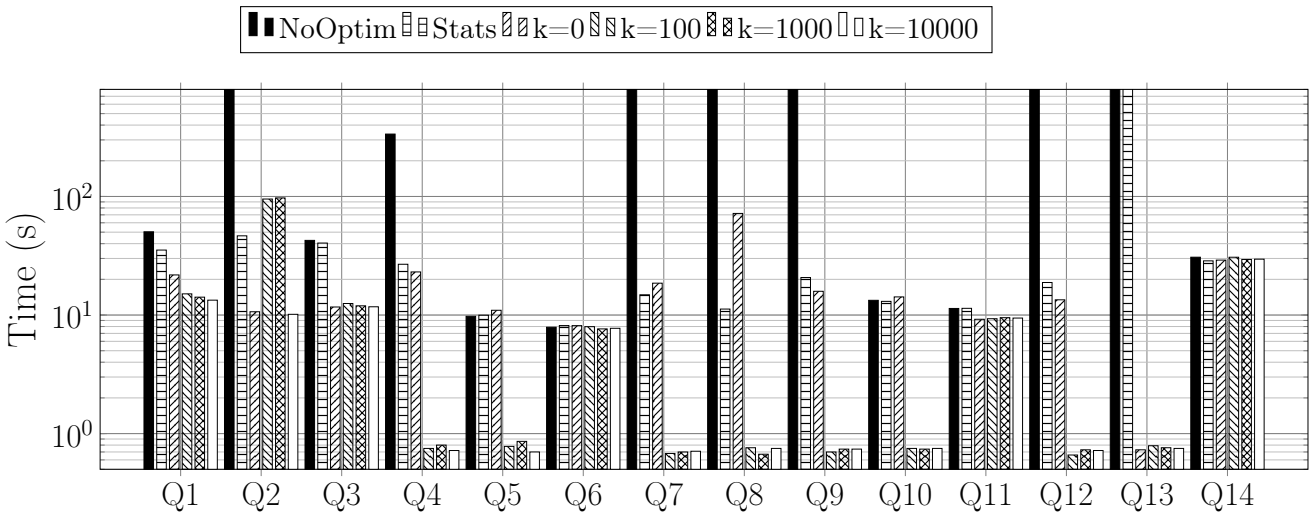
7.5.2 Setup

Our cardinality estimator was tested using various k for the statistic size $k = 0$, $k = 100$, $k = 1\,000$, $k = 10\,000$. We also compared ourselves with the *stats* module of SparqlGX [GJGL16b] that re-organizes TP using statistics about the dataset collected at load time plus general heuristic to obtain a fast query execution plan. We also compared this enhanced SparqlGX with *NoOptim* module of SparqlGX that does not optimize anything.

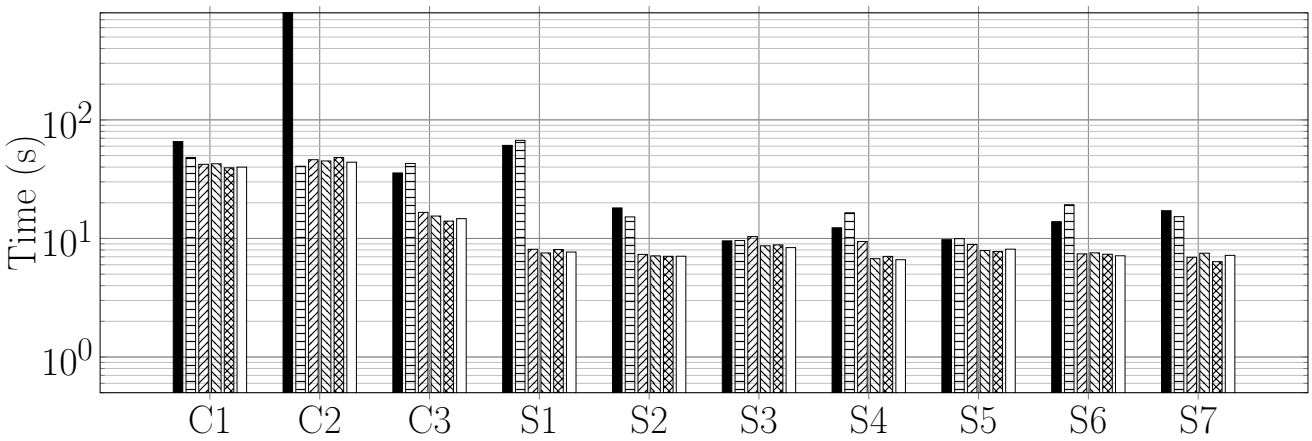
Our prototype uses Apache Spark [ZCD+12] version 2.1.0 in a cluster of two computing nodes running debian each equipped with 20 GB of RAM and 24 cores of computation. The dataset was stored using Hadoop 2.7.3.

7.5.3 Datasets and queries

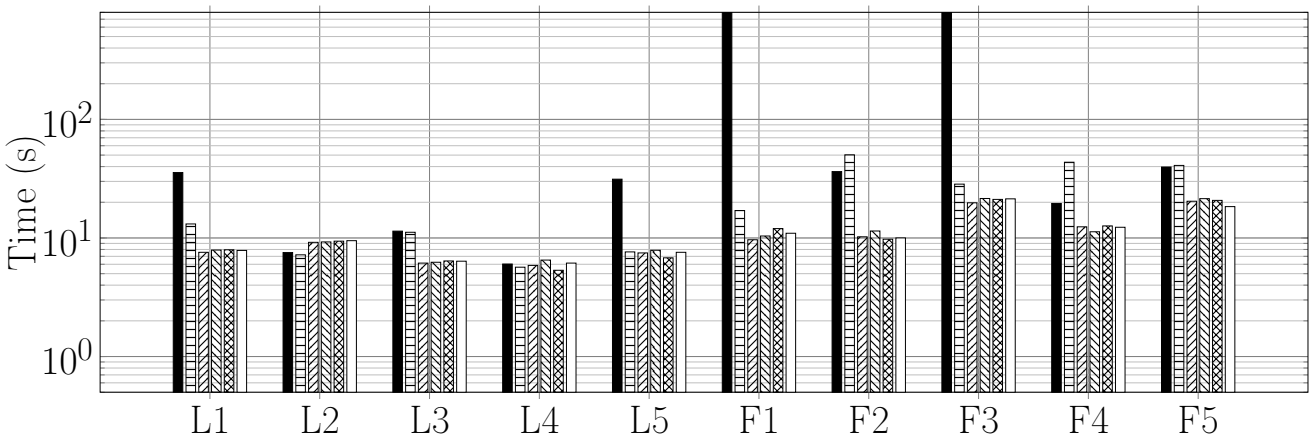
We tested the optimization against the benchmark Lubm [GPH05] 10k that contains 1.38 billions triples and weights 232 GB uncompressed while the WatDiv [AHÖD14] 10k contains



(a) Query response times (**seconds**) Lubm10k



(b) Query response times (**seconds**) WatDiv10k



(c) Query response times (**seconds**) WatDiv10k

Figure 7.17: Time spent to answer Lubm and WatDiv queries

Lubm Query	$Q1$	$Q2$	$Q3$	$Q4$	$Q5$	$Q6$	$Q7$
Stats	35.32	46.57	40.48	26.81	9.97	**8.15**$^+$	14.82
k=0	21.79	**10.66**$^+$	**11.69**$^+$	23.10	10.97	**8.14**$^+$	18.57
k=100	15.09	95.15	**12.51**$^+$	**0.75**$^+$	**0.78**$^+$	**7.98**$^+$	**0.68**$^+$
k=1000	14.15	97.21	**11.96**$^+$	**0.80**$^+$	**0.86**$^+$	**7.64**$^+$	**0.70**$^+$
k=10000	**13.35**$^+$	**10.14**$^+$	**11.74**$^+$	**0.72**$^+$	**0.70**$^+$	**7.73**$^+$	**0.71**$^+$

Lubm Query	$Q8$	$Q9$	$Q10$	$Q11$	$Q12$	$Q13$	$Q14$
Stats	11.23	20.72	13.04	11.39	18.81	$\perp$	**28.64**$^+$
k=0	71.95	15.86	14.22	**9.21**$^+$	13.44	**0.73**$^+$	**29.06**$^+$
k=100	**0.76**$^+$	**0.70**$^+$	**0.75**$^+$	**9.31**$^+$	**0.66**$^+$	**0.79**$^+$	**30.73**$^+$
k=1000	**0.67**$^+$	**0.74**$^+$	**0.74**$^+$	**9.50**$^+$	**0.73**$^+$	**0.76**$^+$	**29.42**$^+$
k=10000	**0.75**$^+$	**0.74**$^+$	**0.75**$^+$	**9.43**$^+$	**0.72**$^+$	**0.75**$^+$	**29.59**$^+$

WatDiv Query	$C1$	$C2$	$C3$	$S1$	$S2$	$S3$	$S4$	$S5$	$S6$	$S7$
Stats	48.21	**40.64**$^+$	42.89	67.18	15.20	9.60	16.46	9.96	19.25	15.27
k=0	42.19	46.13	16.60	**8.10**$^+$	**7.34**$^+$	10.37	9.42	**8.90**$^+$	**7.41**$^+$	**6.94**$^+$
k=100	42.63	45.03	15.45	**7.55**$^+$	**7.13**$^+$	**8.67**$^+$	**6.75**$^+$	**7.91**$^+$	**7.56**$^+$	**7.51**$^+$
k=1000	**39.41**$^+$	48.16	**14.01**$^+$	**8.07**$^+$	**7.07**$^+$	**8.82**$^+$	**7.06**$^+$	**7.76**$^+$	**7.36**$^+$	**6.37**$^+$
k=10000	**39.95**$^+$	43.97	**14.69**$^+$	**7.68**$^+$	**7.08**$^+$	**8.37**$^+$	**6.62**$^+$	**8.12**$^+$	**7.14**$^+$	**7.20**$^+$

WatDiv Query	$L1$	$L2$	$L3$	$L4$	$L5$	$F1$	$F2$	$F3$	$F4$	$F5$
Stats	13.13	**7.21**$^+$	11.17	**5.67**$^+$	**7.63**$^+$	17.04	50.25	28.45	43.54	40.92
k=0	**7.57**$^+$	9.19	**6.14**$^+$	**5.87**$^+$	**7.48**$^+$	**9.66**$^+$	**10.23**$^+$	**19.80**$^+$	**12.42**$^+$	20.41
k=100	**7.91**$^+$	9.26	**6.24**$^+$	**6.51**$^+$	**7.88**$^+$	**10.39**$^+$	**11.45**$^+$	21.53	**11.28**$^+$	21.46
k=1000	**7.95**$^+$	9.40	**6.4**$^+$	**5.34**$^+$	**6.80**$^+$	12.00	**9.76**$^+$	21.15	**12.64**$^+$	20.75
k=10000	**7.85**$^+$	9.49	**6.37**$^+$	**6.14**$^+$	**7.57**$^+$	**10.96**$^+$	**10.01**$^+$	21.38	**12.31**$^+$	**18.39**$^+$

Figure 7.18: Query time results for SPARQLGX against Lubm10k and WatDiv 10k with the "stats" and "summaries" optimizers

1.10 billions triples and weights 149 GB. We also loaded DBpedia [ABK+07] with 2.2 billions triples and weighting 301 GB.

In Lubm, most queries (Q4 to Q13) compute an empty answer unless the dataset is extended with reasoning. We did not extend the dataset with reasoning but we left those queries as it is an interesting use-case of our method to detect empty answers (which it fails to do for instance for Q11 by predicting 20 results or less).

The time spent in evaluating each query of these benchmarks and each query optimizer are shown graphically in figure 7.17 and numerically in figure 7.18. These times do not include the translation time that took less than one second for all queries except for $C2$ and $F4$ with $k = 10\,000$ (for which it took approximately 10s but we have no doubt that this compilation could be further optimized). Notice that the time axis of the figure 7.17 is logarithmic.

7.5.4 Experimentation

Between each query the Spark cluster was stopped and relaunched. The OS did not restart between queries but we flushed the writing cache. All queries ran three times and we interleaved the different methods so that all methods benefit equally from the read cache. We took the best of the three for each method (other metrics would give an advantage to the last tested method). All experiments were stopped after 10 minutes of computation. That "stats" query optimizer of SPARQLGX had to be stopped for the query $Q13$.

The statistics were collected with a double pass on the data during the load phase and the uncompressed size of the collected statistic is 28 MB for WatDiv, 7.8 MB for Lubm and 143 MB for DBpedia, for $k = 10\,000$ (it is roughly linear in the size of k). Compared with the dataset size, the size of statistics is negligible (less than a thousandth) and the computation

time of statistics is also negligible compared with the loading time. However during the translation of a query with n triple patterns we can compute up to 2^n summaries each of size roughly linear in k. Therefore a k larger than 10 000 would imply a long computation time. This problem could be tackled by adapting k dynamically in consideration with the number of triple patterns in the queries and their constant parts.

7.5.5 Results: comparison between different query optimizers

On most queries, the SparqlGX query evaluator goes the fastest when equipped with our *summaries* and $k = 10\,000$. And the query time tends to decrease as k increases. On the opposite there are several queries with a non-empty result where our method vastly outperforms the *stats* module of SparqlGX. Moreover, see figure 7.17, the *NoOptim* module of SparqlGX is always slower –by sometimes two or more orders of magnitude– than our *summaries* with $k = 10\,000$. Nonetheless, when dealing with very simple sparql queries, e.g. Q6 & Q14 of Lubm, involving only one tp, the five tested cases appear to need similar times to answer.

There are two queries for which the *stats* module is faster than our $k = 10\,000$ optimizer: $C2$ and $F4$. Both queries are complex but the problem is with the fact that Spark evaluation adds a constant, non negligible constant time on the evaluation of small broadcast joins. Indeed, broadcast joins imply to collect the data on the driver before sending it and the collect operation in *Apache Spark* is costly: it forbids other operation to run in parallel and a few seconds of delay to wait for all executors to finish where only a fraction of the cluster is working. We try to fix our cost model to take this fixed cost into account but then we were cornered by the limit of precision of our cardinality estimation.

Notice our approach can lead to very sub-optimal plans when the cardinality estimations are very far-off; but in these benchmark queries our query plans are all optimal or near optimal in query evaluation time.

7.5.6 Precision of the cardinality estimations

On the 34 queries of the benchmark there are only 28 for which algorithm prediction was off by less than 10^6 solutions and 15 that are off by less 100.

The queries of WatDiv and Lubm are designed to be relatively complex queries containing large star patterns and it is thus not surprising that our estimations are sometimes far-off by one or two orders of magnitudes (sometimes even more). Furthermore these queries tend to be very selective compared with the size of intermediate sets involved. However we would like to point the following:

- the useful cardinality estimations to optimize the query are not on the final query but on subqueries, and in this benchmark subqueries are often much more precise (as the precision degrades with the number of TP involved);
- we are using overestimation, which means than an error will never trigger a risky plan;
- even on far-off cardinality estimations our cardinality estimation tends to favor good plans; for instance if we have three sets A, B and C and the estimation of A is k times the actual number of elements in A, then the estimated cost of the plans $(A \bowtie B) \bowtie C$

and $(A \bowtie C) \bowtie B$ will be multiplied by k but are generally kept in the same relative order (unless C and B are also far-off);

- if we fall back on another cardinality estimation for the join orders and use the worst case estimation only for broadcast joins then even a factor of 100 is actually not very much. If we allow the broadcast of sets of size up to 10^8 then a error of two order of magnitude –i.e. 100×– effectively allows broadcast for sets of size up to 10^6.

Conclusion

In this chapter, we have evaluated the benefits of our contributions. This evaluation was split along three criteria.

In the first part, we compared the plans that are considered for recursive queries when using our μ-algebra and the plans that can be considered with other query evaluators such as the relational algebra, Datalog and other SPARQL query evaluators. This comparison shows that our μ-algebra considers more plans and among those additional plans are plans that are much more efficient.

In the second, we translate this theoretical into an empirical comparison. We choose a very simple query for which our method considers more plans and we observe that our prototype implementation takes a time linear in the size of the graph to be evaluated while others methods take a quadratic time. This results is not surprising after our theoritical comparison but it validates the comparison experimentally.

Finally in a third part, we compare the performance of SparqlGX equipped with diverse query optimizers. A heuristic one, a statistical one and four variations of the cardinality estimator that we introduced. The results are that our new cardinality estimation allows SparqlGX to choose better plans. Equipped with our estimation, SparqlGX performs faster on most queries, sometimes by an order of magnitude.

CHAPTER 8

Conclusions & Perspectives

8.1 Conclusions

During my PhD, I tackled the question of the efficient evaluation of SPARQL and proposed a new approach based on a new tool: the μ-algebra.

Chapter 1 presents RDF and SPARQL. RDF is a data model describing graphs. SPARQL is a rich language to query RDF graphs. SPARQL query evaluation raises challenges: how to evaluate such a rich query language on graphs that are, by nature, very large. More precisely: how can we find efficient query plans to handle even simple queries but on very large datasets? How can we find efficient plans for complex queries on which today's evaluators tends to fail even on relatively small datasets? Since the relational model was at the center of most of the research in database, before addressing those questions, we needed to dive into the relational model and what it can offer to evaluate SPARQL queries.

Chapter 2 briefly presents the relational model and its query languages. The research on the relational algebra and SQL in particular are a great basis on which we can build query evaluators and optimizers. However, in their current form, relational-based SPARQL query evaluators suffer from inefficiencies and there are multiple reasons at the root of these inefficiencies:

- relational query evaluators do not optimize well some type of queries such as recursive queries;
- relational query languages and SPARQL do not match well semantically;
- and finally some features of SPARQL are hard to evaluate efficiently.

To address these two first points, we have presented our μ-algebra, inspired by the relational algebra but adapted to handle both the recursivity part and the non-relational (missing values) part.

Chapter 3 then presented this μ-algebra. This algebra takes its inspiration from the relational algebra but diverges on some essential points: as it name suggests the relational algebra describes relations while solutions to our algebra are sets of mappings that do not necessarily share the same domain. Furthermore our algebra is equipped with a novel fixpoint

operator μ. For this fixpoint to be well-defined we restricted ourselves to "linear" terms and introduced a translation from a large fragment of the SPARQL–algebra to our μ-algebra. Since our algebra is a variation of the relational algebra and since the relational algebra optimization is based on rewrite rules, the direction we have followed is to investigate the rewrite rules and strategies can be applied for our μ-algebra.

Chapter 4, after some motivating examples, starts by introducing definitions, lemmas and theorems laying the ground work for new rewrite rules. We then equipped our μ-algebra with a typing system capturing the shape of the domain of the solutions for a μ-algebra term. The chapter continued with the presentation of our rewrite rules that we decomposed into "normalizing" rules and "producing". The "normalizing" rules allow us to reduce the search space for terms will the later allow us to discover new terms. Finally, we presented our rewrite algorithm and an example of a term rewritten. Now that our method can produce numerous equivalent terms, there are two natural questions: how can one evaluate those terms? how to select the most efficient term to be evaluated? These two questions are inherently linked as the efficiency is relative to the evaluation method. We therefore tackled both questions in the following chapter.

Chapter 5 investigated how we can evaluate μ-algebra terms and deduced from this evaluation a cost model for our general evaluation scheme. We have then presented two evaluators that we implemented based on this general compilation scheme. These two evaluators correspond to two different types of applications: SparqlGX handles only a fragment of the SPARQL languages but run on an efficient distributed platforms, `musparql` on the other hand is a single core evaluator but can handle and optimize complex queries. In our general optimization scheme for the evaluation of μ-algebra terms, we rely on a cost model to guess what is the estimated best QEP. And this cost model itself relies on a cardinality estimation. While a naive cardinality estimation already leads to interesting results, we investigated the use of more complex schemes to assess the number of solutions to a μ-algebra term. The task of estimating the number of solutions to a given query, even a simple conjunctive query, has been the center of many research projects and yet still is an active research subject.

Chapter 6 introduced a worst-case cardinality estimation based on a new concept: collection summaries, which are extracted from statistics on the data. We showed how to compute these collection summaries for conjunctive queries and how they can be used to estimate the cardinality of query answers. With this cardinality estimator, we had all the pieces for evaluating SPARQL queries using the μ-algebra.

Chapter 7 evaluated the benefits of our contributions. This evaluation is split along three criteria. In the first part, we compared the plans that are considered for recursive queries when using our μ-algebra and the plans that can be considered with other query evaluators such as the relational algebra, Datalog and other SPARQL query evaluators. This comparison showed that our μ-algebra consider more plans and among those additional plans are plans that are much more efficient. In the second, we translated this theoretical into an empirical comparison. We choose a very simple query for which our method considers more plans and we observed that our prototype implementation takes a time linear in the size of the graph to be evaluated while others methods take a quadratic time. This results validated experimentally the theoretical comparison. Finally in a third part, we compared the performance of SparqlGX equipped with diverse query optimizers. A heuristic one, a statistical one and four variations of the cardinality estimator that we introduced. The results are that our new cardinality estimation allows SparqlGX to choose better plans. Equipped with our estimation, SparqlGX performed faster on most queries, sometimes by

an order of magnitude.

8.2 Perspectives

8.2.1 Cardinality estimation

We have many ongoing research ideas to improve the cardinality estimation that are detailed in section 6.4. We could improve the precision of our statistics by combining them with other statistics provided by other ongoing works in the field of cardinality estimation.

Our statistics works relatively well for the conjunctive fragment (which corresponds to the BGP fragment of SPARQL queries). It would be interesting to further investigate the precision of our method on BGP but also on richer fragments. A good precision might be very hard to achieve on the full μ-algebra but a first step would be to assess the precision of our method e.g. on the fragment composed of safe unions and conjunctions which is notoriously hard to optimize.

8.2.2 On the languages covered

For the moment, our system has only be tested for SPARQL. There is current trend of more and more languages and it would be interesting to research how well other query languages can be translated into the μ-algebra and benefits from our approach.

In our work, we expect to see the μ-algebra as a potential target language for several expressive language. One language that we could investigate is OpenCypher which is a graph query language that includes regular path queries and operates on property graphs. Another example of language that we could consider is XPath. XPath allows some restricted form of recursive queries. Our language could be seen as either as a target for XPath or for regular XPath which allows for unrestricted recursive queries.

As we have seen, one very interesting aspect of our μ-algebra is its ability to capture more rewriting that what SQL and Datalog were natively capable of. One direction that we could investigate is to try to apply directly our optimization to Datalog and SQL engines. In fact, even if we do not have results just yet, this direction is already explored.

8.2.3 On the μ-algebra

For the moment μ-algebra is evaluated either in our prototype or, for a small fragment, translated into the distributed framework Spark. However, μ-algebra shares similarities with recursive SQL and one other direction that we could pursue is to investigate the compilation of μ-algebra into this recursive SQL using our intermediate representation as a logical optimization phase leaving the details of the physical optimization to well-tuned SQL engines. In a similar fashion, we could investigate the translation of μ-algebra into Datalog.

μ-algebra possesses naturally a set semantics. Just like SQL that can be seen as a bag semantic version of the relational algebra, we could investigate the potential of μ-algebra equipped with a bag semantic: what rewriting rules continue to hold, etc.. There are many cases where such a set semantics would be more appropriated and we could foresee that the translation of SPARQL would be simpler as this modified μ-algebra would directly have a set semantics.

Finally the μ-algebra is a relatively rich language and as one of the advantages of SPARQL is to enrich data with ontologies, one direction that we could pursue is to rewrite incoming SPARQL queries to take into account the ontology. Thanks to the expressivity of our μ-algebra, we could take into account a broad class of ontologies.

8.2.4 On its execution

For the moment, the evaluation of μ-algebra terms is implemented as a prototype in a single core evaluator. There are several directions of improvements for this evaluator.

The current prototype uses rather naive implementation that we could optimize. For instance the data structures to encode RDF graphs are essentially strings. It would be much more efficient to use an auxiliary data structure to encode URI, blank nodes and litterals not with strings but with pointers or integers and put more indexes on data. Such optimization would not necessarily improve performance in a distributed setup but they would in a single-machine implementation. Furthermore, we could also improve the efficiency by using multiple core to implement the various operators.

As explained in section 5.4, SPARQLGX can be seen as an implementation for a small fragment of μ-algebra in Apache Spark. Apache Spark does not possess an efficient iteration for the moment and Spark model of computation is very far from a stream approach as described in chapter 5. One direction we should investigate is the translation of our prototype in a distributed framework using more a dataflow approach (more similar to the compilation we presented) or equipped with a real fixpoint computation.

Bibliography

[ABE09] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with Regular Expression Patterns (for Querying RDF). *Web Semant.*, 7(2):57–73, April 2009.

[ABJM17] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 111:1–111:15, 2017.

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web*, Lecture Notes in Computer Science, pages 722–735. Springer, Berlin, Heidelberg, 2007. DOI: 10.1007/978-3-540-76298-0_52.

[ABYG+17] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Shadi Ghajar-Khosravi, and Mark H. Chignell. TASWEET: Optimizing Disjunctive Path Queries in Graph Databases. In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Bre\s s, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 470–473. OpenProceedings.org, 2017.

[ACP12] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting Beyond a Yottabyte, or How SPARQL 1.1 Property Paths Will Prevent Adoption of the Standard. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 629–638, New York, NY, USA, 2012. ACM.

[AE13] Faisal Alkhateeb and Jérôme Euzenat. Answering SPARQL queries modulo RDF Schema with paths. *arXiv:1311.3879 [cs]*, November 2013. arXiv: 1311.3879.

[AE14] Faisal Alkhateeb and Jérôme Euzenat. Constrained regular expressions for answering RDF-path queries modulo RDFS. *International Journal of web information systems*, 10(1):24–50, 2014.

[AFMPF11] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An Empirical Study of Real-World SPARQL Queries. *CoRR*, abs/1103.5043, 2011.

[Agr88] R. Agrawal. Alpha: an extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, July 1988.

[AGRL17] Abdullah Abbas, Pierre Genevès, Cécile Roisin, and Nabil Layaïda. Optimising SPARQL Query Evaluation in the Presence of ShEx Constraints. In *BDA 2017 - 33ème conférence sur la " Gestion de Données - Principes, Technologies et Applications "*, pages 1–12, Nancy, France, November 2017.

[AHÖD14] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *The Semantic Web – ISWC 2014*, Lecture Notes in Computer Science, pages 197–212. Springer, Cham, October 2014.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[AMMH07] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, New York, NY, USA, 1979. ACM.

[AV91] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci.*, 43(1):62–124, August 1991.

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[BBC+15] Vinayak Borkar, Yingyi Bu, E. Preston Carman, Jr., Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J. Carey, and Vassilis J. Tsotras. Algebricks: A Data Model-agnostic Compiler Backend for Big Data Languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 422–433, New York, NY, USA, 2015. ACM.

[BBC+17] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. gmark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29:856–869, 2017.

[BEH+10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[BGV05] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-Definedness and Semantic Type-Checking in the Nested Relational Calculus and XQuery. In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005*, number 3363 in Lecture Notes in Computer Science, pages 99–113. Springer Berlin Heidelberg, January 2005. DOI: 10.1007/978-3-540-30570-5_7.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[BV07] Jan Van den Bussche and Stijn Vansummeren. Polymorphic Type Inference for the Named Nested Relational Calculus. *ACM Trans. Comput. Logic*, 9(1), December 2007.

[CDA12] Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed Automatic Incrementalization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 299–310, New York, NY, USA, 2012. ACM.

[CDGLV15] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Node Selection Query Languages for Trees. *arXiv:1509.08979 [cs]*, September 2015. arXiv: 1509.08979.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, March 1989.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[CLF09] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics Preserving SPARQL-to-SQL Translation. *Data Knowl. Eng.*, 68(10):973–1000, October 2009.

[CLW14] James Cheney, Sam Lindley, and Philip Wadler. Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1027–1038, New York, NY, USA, 2014. ACM.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM*

Symposium on Theory of Computing, STOC '77, pages 77–90, New York, NY, USA, 1977. ACM.

[CNBA15] Olivier Curé, Hubert Naacke, Mohamed-Amine Baazizi, and Bernd Amann. On the Evaluation of RDF Distribution Algorithms Implemented over Apache Spark. *arXiv:1507.02321 [cs]*, July 2015. arXiv: 1507.02321.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CRCM14] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. PAXQuery: A Massively Parallel XQuery Processor. In *Proceedings of Workshop on Data Analytics in the Cloud*, DanaC'14, pages 6:1–6:4, New York, NY, USA, 2014. ACM.

[Cyg05] Richard Cyganiak. A relational algebra for SPARQL. 2005.

[DAA12] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. Provenance for SPARQL Queries. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2012*, number 7649 in Lecture Notes in Computer Science, pages 625–640. Springer Berlin Heidelberg, November 2012. DOI: 10.1007/978-3-642-35176-1_39.

[Den74] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, pages 362–376, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DKSU11] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 145–156, New York, NY, USA, 2011. ACM.

[EM09] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. In Tassilo Pellegrini, Sóren Auer, Klaus Tochtermann, and Sebastian Schaffert, editors, *Networked Knowledge - Networked Media*, number 221 in Studies in Computational Intelligence, pages 7–24. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-02184-8_2.

[FS98] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings 14th International Conference on Data Engineering*, pages 14–23, February 1998.

[GBS13] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. Sparqling Kleene: Fast Property Paths in RDF-3x. In *First International Workshop*

on Graph Data Management Experiences and Systems, GRADES '13, pages 14:1–14:7, New York, NY, USA, 2013. ACM.

[GdM86] Georges Gardarin and Christophe de Maindreville. Evaluation of Database Recursive Logic Programs As Recurrent Function Series. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 177–186, New York, NY, USA, 1986. ACM.

[GHS13] Marek Grabowski, Jan Hidders, and Jacek Sroka. Representing MapReduce Optimisations in the Nested Relational Calculus. In Georg Gottlob, Giovanni Grasso, Dan Olteanu, and Christian Schallhart, editors, *Big Data*, number 7968 in Lecture Notes in Computer Science, pages 175–188. Springer Berlin Heidelberg, July 2013. DOI: 10.1007/978-3-642-39467-6_17.

[GJGL16a] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. SPARQLGX: efficient distributed evaluation of SPARQL with apache spark. In *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*, pages 80–87, 2016.

[GJGL16b] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. SPARQLGX in action: Efficient distributed evaluation of SPARQL with apache spark. In *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016.*, 2016.

[GJGL17] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. The SPARQLGX System for Distributed Evaluation of SPARQL Queries. October 2017.

[GKM+15] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. A. Quiané-Ruiz, and S. Zampetakis. CliqueSquare: Flat plans for massively parallel RDF queries. In *2015 IEEE 31st International Conference on Data Engineering*, pages 771–782, April 2015.

[GLSG15] Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. Efficiently Deciding mu-Calculus with Converse over Finite Trees. *ACM Trans. Comput. Logic*, 16(2):16:1–16:41, April 2015.

[GMUW09] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson international edition. Pearson Prentice Hall, 2009.

[GN14] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in sparql queries. In *In Edbt*, pages 439–450, 2014.

[GNC+09] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. VLDB Endow.*, 2(2):1414–1425, August 2009.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, October 2005.

[GSMT14] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 289–300, New York, NY, USA, 2014. ACM.

[HA92] Maurice AW Houtsma and Peter MG Apers. Algebraic optimization of recursive queries. *Data & Knowledge Engineering*, 7(4):299–325, 1992.

[HAK+16] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal*, 25(3):355–380, June 2016.

[HH07] Olaf Hartig and Ralf Heese. The SPARQL Query Graph Model for Query Optimization. In *The Semantic Web: Research and Applications*, pages 564–578. Springer, Berlin, Heidelberg, June 2007.

[HL10] Hai Huang and Chengfei Liu. Selectivity Estimation for SPARQL Graph Pattern. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1115–1116, New York, NY, USA, 2010. ACM.

[HLS09] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered rdf store. In *Scalable Semantic Web Knowledge Base Systems - SSWS2009*, pages 94–109, 2009.

[HSP13] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. *W3C recommendation*, 21(10), 2013.

[Ioa03] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 19–30. VLDB Endowment, 2003.

[IP99] Yannis E. Ioannidis and Viswanath Poosala. Histogram-Based Approximation of Set-Valued Query-Answers. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 174–185, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[JCR11] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic Optimization for MapReduce Programs. *Proc. VLDB Endow.*, 4(6):385–396, March 2011.

[KK16] Mark Kaminski and Egor V. Kostylev. Beyond Well-designed SPARQL. In Wim Martens and Thomas Zeume, editors, *19th International Conference on Database Theory (ICDT 2016)*, volume 48 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:18, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[KKG17] Mark Kaminski, Egor V. Kostylev, and Bernardo Cuenca Grau. Query Nesting, Assignment, and Aggregation in SPARQL 1.1. *ACM Trans. Database Syst.*, 42(3):17:1–17:46, August 2017.

[Koz82] Dexter Kozen. Results on the propositional mu-calculus. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming*, number 140 in Lecture Notes in Computer Science, pages 348–359. Springer Berlin Heidelberg, July 1982. DOI: 10.1007/BFb0012782.

[KRA17] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. Type-based semantic optimization for scalable RDF graph pattern matching. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 785–793, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[KRRV15] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. SPARQL with Property Paths. In Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015*, number 9366 in Lecture Notes in Computer Science, pages 3–18. Springer International Publishing, October 2015. DOI: 10.1007/978-3-319-25007-6_1.

[KRU15] Kostylev, Egor V., Reutter, Juan L., and Ugarte, Martín. CONSTRUCT Queries in SPARQL. 2015.

[LM12] Katja Losemann and Wim Martens. The Complexity of Evaluating Path Expressions in SPARQL. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, pages 101–112, New York, NY, USA, 2012. ACM.

[LM13] Katja Losemann and Wim Martens. The Complexity of Regular Expressions and Property Paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24:1–24:39, December 2013.

[LP77] Michel Lacroix and Alain Pirotte. Domain-oriented relational languages. In *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3*, VLDB '77, pages 370–378. VLDB Endowment, 1977.

[MGL+10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, September 2010.

[Nix] Lyndon Nixon. *A Cost Model for Querying Distributed RDF-Repositories with SPARQL.*

[NM11] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*, pages 984–994, April 2011.

[NS16] Maurizio Nolé and Carlo Sartiani. Regular path queries on massive graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, SSDBM '16, pages 13:1–13:12, New York, NY, USA, 2016. ACM.

[NW08] Thomas Neumann and Gerhard Weikum. RDF-3x: A RISC-style Engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, August 2008.

[OR00] B. John Oommen and Luis Rueda. An Empirical Comparison of Histogram-Like Techniques for Query Optimization. In *ICEIS*, 2000.

[Ord10] Carlos Ordonez. Optimization of Linear Recursive Queries in SQL. *IEEE Transactions on Knowledge and Data Engineering*, 22(2):264–277, February 2010.

[PCR12] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: A scalable rdf triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, pages 4:1–4:8, New York, NY, USA, 2012. ACM.

[PDGQ05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[PSC84] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Rec.*, 14(2):256–276, June 1984.

[PT08] T. Pitoura and P. Triantafillou. Self-join size estimation in large-scale distributed data systems. In *2008 IEEE 24th International Conference on Data Engineering*, pages 764–773, 2008.

[PZO+16] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing SPARQL queries over distributed RDF graphs. *The VLDB Journal*, 25(2):243–268, April 2016.

[PZSHL11] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Georg Lausen. RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In Raúl García-Castro, Dieter Fensel, and Grigoris Antoniou, editors, *The Semantic Web: ESWC 2011 Workshops*, number 7117 in Lecture Notes in Computer Science, pages 50–64. Springer Berlin Heidelberg, May 2011. DOI: 10.1007/978-3-642-25953-1_5.

[RCM14] David Wood Richard Cyganiak and Lanthaler Markus. RDF 1.1 concepts and abstract syntax, February 2014.

[RKA11] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *The Semanic Web: Research and Applications*, Lecture Notes in Computer Science, pages 46–61. Springer, Berlin, Heidelberg, May 2011.

[RMR15a] Mariano Rodríguez-Muro and Martin Rezk. Efficient SPARQL-to-SQL with R2rml mappings. *Web Semantics: Science, Services and Agents on the World Wide Web*, 33:141–169, August 2015.

[RMR15b] Mariano Rodríguez-Muro and Martin Rezk. Efficient SPARQL-to-SQL with R2rml mappings. *Web Semantics: Science, Services and Agents on the World Wide Web*, 33:141–169, August 2015.

[RSV15] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. Recursion in SPARQL. In *The Semantic Web - ISWC 2015*, pages 19–35. Springer, Cham, October 2015.

[SASU13] Anish Das Sarma, Foto N. Afrati, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 277–288, Trento, Italy, 2013. VLDB Endowment.

[Sch07] Thomas Schwentick. Automata for XML—A survey. *Journal of Computer and System Sciences*, 73(3):289–315, May 2007.

[Shi16] Emilio Patrick Shironoshita. Query optimization for SPARQL, February 2016. International Classification G06F17/30; Cooperative Classification G06F17/30436.

[SJMR18] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. Provsql: Provenance and probability management in postgresql. *PVLDB*, 11(12):2034–2037, 2018.

[SML10a] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[SML10b] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[SPZL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. Pigsparql: Mapping sparql to pig latin. In *Proceedings of the International Workshop on Semantic Web Information Management*, SWIM '11, pages 4:1–4:8, New York, NY, USA, 2011. ACM.

[SPZSL16] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2rdf: Rdf querying with sparql on spark. *Proc. VLDB Endow.*, 9(10):804–815, June 2016.

[SSB+08] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[ST09] Maryam Shoaran and Alex Thomo. Fault-tolerant computation of distributed regular path queries. *Theor. Comput. Sci.*, 410(1):62–77, January 2009.

[SVS+13] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[SZ86] Domenico Saccà and Carlo Zaniolo. On the implementation of a simple class of logic queries for databases. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 16–23, New York, NY, USA, 1986. ACM.

[TLX13] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal MapReduce Algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 529–540, New York, NY, USA, 2013. ACM.

[TSF+12] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-based Query Optimisation for SPARQL. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 324–335, New York, NY, USA, 2012. ACM.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[UJK16] Jacopo Urbani, Ceriel JH Jacobs, and Markus Krötzsch. Column-oriented datalog materialization for large knowledge graphs. In *AAAI*, pages 258–264, 2016.

[Var98] Moshe Y. Vardi. Reasoning about the past with two-way automata. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, number 1443 in Lecture Notes in Computer Science, pages 628–641. Springer Berlin Heidelberg, July 1998. DOI: 10.1007/BFb0055090.

[WCM16] M. Wylot and P. Cudré-Mauroux. DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud. *IEEE Transactions on Knowledge and Data Engineering*, 28(3):659–674, March 2016.

[WS03] Hai Wang and Kenneth C. Sevcik. A Multi-dimensional Histogram for Selectivity Estimation and Fast Approximate Query Answering. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '03, pages 328–342, Toronto, Ontario, Canada, 2003. IBM Press.

[XGFS13] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[XRRMC14] Guohui Xiao, Martin Rezk, Mariano Rodríguez-Muro, and Diego Calvanese. Rules and Ontology Based Data Access. In Roman Kontchakov and Marie-Laure Mugnier, editors, *Web Reasoning and Rule Systems*, number 8741 in Lecture Notes in Computer Science, pages 157–172. Springer International Publishing, September 2014. DOI: 10.1007/978-3-319-11113-1_11.

[YGG13] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Evaluation of sparql property paths via recursive sql. *AMW*, 1087, 2013.

[YGG15] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Towards Query Optimization for SPARQL Property Paths. *arXiv:1504.08262 [cs]*, April 2015. arXiv: 1504.08262.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZYW+13] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 265–276, Trento, Italy, 2013. VLDB Endowment.

APPENDIX A

Proofs

A.1 Proofs of chapter 3

Proposition 1. *For all pairs of mappings* (m_1, m_2) *and all filter* f *such that* $dom(m_1) \cap FC(f) = dom(m_2) \cap FC(f)$ *and* $\forall c \in dom(m_1) \cap FC(f)$ $m_1(c) = m_2(c)$ *then* $eval(f)(m_1) = eval(f)(m_2)$ *and thus the definition of* FC *works as expected.*

Proof. $eval(f)(m_1) = eval(f)(m_2)$ when $dom(m_1) \cap FC(f) = dom(m_2) \cap FC(f)$ and $\forall c \in (dom(m_1) \cap FC(f))$ $m_1(c) = m_2(c)$ by induction on the size of f.

- For $f = bnd(c)$, we have $eval(bnd(c))(m_1) = eval(bnd(c)(m_2)$ by definition.
- For $f = test(c_1, \ldots, c_2)$, we have $eval(f)(m_1) = E$ iff there exists i such that $c_i \notin dom(m_1)$ but $c_i \in FC(f)$ implies $c_i \notin dom(m_2)$ and thus $eval(f)(m_2) = E$. Reciprocally $eval(f)(m_2) = E$ implies $eval(f)(m_1) = E$. Now, if all c_i are in the $dom(m_1) \cap FC(f)$ then $eval(f)(m_1) = test(m_1(c_1), \ldots, m_1(c_n)) = test(m_2(c_1), \ldots, m_2(c_n)) = eval(f)(m_2)$.
- For $f \in \{f_1 \wedge f_2, f_1 \vee f_2, \neg f\}$ the result holds by induction.

In any case we have the evaluation of a filter f on a mapping m only depends on the $FC(f)$ part of the domain of m. □

A.1.1 Lemma 1

Lemma 1. *Given a term* φ *valid in the sense of definition 17 and such that* $sim(\varphi, X) = 0$ *then* $[\![\varphi]\!]_V$ *does not depend on* $V(X)$*, i.e.* $\forall S \quad [\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\emptyset]}$.

Proof.
- Clearly this is true for $\emptyset$, $|c_1 \to v_1, \ldots, c_n \to v_n|$, and for variables Y since $sim(X, Y) = 0 \Rightarrow Y \neq X$.
- For binary formulas $\varphi_1 \ \lambda \ \varphi_2$ with $\lambda \in \{\cup, \bowtie, ⋊, \backslash\backslash, \backslash, -\}$, since $sim(\varphi, X) = 0$ we have that $sim(\varphi_1, X) = 0$ and $sim(\varphi_2, X) = 0$ which gives us recursively $[\![\varphi_1]\!]_{V[X/S]} = [\![\varphi_1]\!]_{V[X/\emptyset]}$, $[\![\varphi_2]\!]_{V[X/S]} = [\![\varphi_2]\!]_{V[X/\emptyset]}$ and thus $[\![\varphi_1 \ \lambda \ \varphi_2]\!]_{V[X/S]}$ also does not depend on S.

- For unary operators $\beta_a^b(\varphi_1)$, $\rho_a^b(\varphi_1)$, $\pi_a(\varphi_1)$, $\theta(\varphi, g : \mathcal{C} \to \mathcal{D})$ and $\Theta(\varphi, g, \mathcal{C}, \mathcal{D})$ the same argument as for binary operators applies: $[\![\varphi]\!]_V$ does not depend on $V(X)$. The only difficulties are for let binders and fixpoints.

- Given the term "let $(Y = \varphi)$ in ψ" such that $sim(\text{let } (Y = \varphi) \text{ in } \psi, X) = 0$. We have that $sim(\psi, X) = 0$ which means $[\![\psi]\!]_{V[X/S]} = [\![\psi]\!]_{V[X/\emptyset]}$ and either $sim(\varphi, X) = 0$ or $sim(\psi, Y) = 0$.

  - If $sim(\psi, Y) = 0$ then $[\![\psi]\!]_{V[Y/S']} = [\![\psi]\!]_{V[Y/\emptyset]}$ and thus $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]} = [\![\psi]\!]_{V[Y/[\![\varphi]\!]_V, X/S]} = [\![\psi]\!]_{V[Y/\emptyset, X/S]} = [\![\psi]\!]_{V[Y/\emptyset, X/\emptyset]}$.
  - If $sim(\varphi, X) = 0$ then $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\emptyset]}$ and thus $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]} = [\![\psi]\!]_{V[X/S, Y/[\![\varphi]\!]_{V[X/S]}]} = [\![\psi]\!]_{V[X/S, Y/[\![\varphi]\!]_{V[X/\emptyset]}]} = [\![\psi]\!]_{V[X/\emptyset, Y/[\![\varphi]\!]_{V[X/\emptyset]}]}$.

  In all cases $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]} = [\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\emptyset]}$.

- Finally, given the term $\mu(Y = \varphi)$ such that $sim(\mu(Y = \varphi), X) = 0$. When $X = Y$ the result is clear ($V(X)$ is replaced by U_i) so let us suppose $X \neq Y$.

  We have $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\emptyset]}$. Let us note $U_0^S = \emptyset$, $U_0^\emptyset = \emptyset$ and $U_{i+1}^S = U_i^S \cup [\![\varphi]\!]_{V[Y/U_i^S]}$, $U_{i+1}^\emptyset = U_i^\emptyset \cup [\![\varphi]\!]_{V[Y/U_i^\emptyset]}$.

  Let us prove by induction that $U_i^S = U_i^\emptyset$ for all $i \in \mathbb{N}$. Clearly $U_0^S = U_0^\emptyset$ and by recurrence we have $U_{i+1}^S = U_i^S \cup [\![\varphi]\!]_{V[Y/U_i^S, X/S]} = U_i^S \cup [\![\varphi]\!]_{V[Y/U_i^S, X/\emptyset]}$ but $U_i^S = U_i^\emptyset$ thus we do have $U_{i+1}^S = U_i^\emptyset \cup [\![\varphi]\!]_{V[Y/U_i^\emptyset, X/\emptyset]} = U_{i+1}^\emptyset$. Since this is true for all $i \in \mathbb{N}$ we have $[\![\mu(X = \varphi)]\!]_{V[X/S]} = [\![\mu(X = \varphi)]\!]_{V[X/\emptyset]}$

□

A.1.2 Lemma 2

Lemma 2. *Given a valid term φ we have that $A \subseteq B \Rightarrow [\![\varphi]\!]_{V[X/A]} \subseteq [\![\varphi]\!]_{V[X/B]}$.*

Proof. We prove this lemma by induction on the size of formulas.

- Lemma 1 implies this lemma when $sim(\varphi, X) = 0$.

- For $\varphi_1 \bowtie \varphi_2$ and $A \subseteq B$ we have by induction $[\![\varphi_i]\!]_{V[X/A]} \subseteq [\![\varphi_i]\!]_{V[X/B]}$ and for $S \in \{A, B\}$, $[\![\varphi_1 \bowtie \varphi_2]\!]_{V[X/S]}$ is increasing in both $[\![\varphi_i]\!]_{V[X/S]}$ thus the result holds.

- For $\varphi_1 \ \lambda \ \varphi_2$ with $\lambda \in \{ \bowtie, \backslash\backslash, \backslash, - \}$ we have $sim(\varphi_1 \ \lambda \ \varphi_2, X) = 1 \Rightarrow sim(\varphi_2, X) = 0$ and thus $[\![\varphi_2]\!]_{V[X/A]} = [\![\varphi_2]\!]_{V[X/B]} = [\![\varphi_2]\!]_{V[X/\emptyset]}$. We have that $[\![\varphi_1 \ \lambda \ \varphi_2]\!]_{V[X/S]}$ is increasing in $[\![\varphi_1]\!]_{V[X/S]}$ and by induction we have that $[\![\varphi_1]\!]_{V[X/A]} \subseteq [\![\varphi_1]\!]_{V[X/B]}$ thus the result holds.

- For $\varphi = \lambda(\varphi_1)$ with λ a unary operator, (i.e. $\varphi \in \{\beta_a^b(\varphi_1), \rho_a^b(\varphi_1), \pi_a(\varphi_1), \theta(\varphi_1, g : \mathcal{C} \to \mathcal{D})\}$) the results holds since $[\![\varphi]\!]_{V[X/S]}$ is increasing in $[\![\varphi_1]\!]_{V[X/S]}$.

- For a fixpoint $\mu(Y = \varphi)$, either $sim(\mu(Y = \varphi), X) = 0$ and the result is obvious or $sim(\mu(Y = \varphi), X) = 2$ and the term is not valid.

- For a let $(Y = \varphi)$ in ψ we have several cases depending on whether $sim(\varphi, X) = 0$, $sim(\psi, X) = 0$ and $sim(\psi, Y) = 0$:

- when $sim(\psi, Y) = 0$ we have $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]} = [\![\psi]\!]_{V[X/S,Y/[\![\varphi]\!]_{V[X/S]}]} = [\![\psi]\!]_{V[X/S,Y/\emptyset]}$ and by induction on ψ, $A \subseteq B \Rightarrow [\![\psi]\!]_{V[X/A,Y/\emptyset]} \subseteq [\![\psi]\!]_{V[X/B,Y/\emptyset]}$ and thus the result holds.
- when $sim(\varphi, X) = 0$ we have $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/A]} = [\![\psi]\!]_{V[X/A,Y/[\![\varphi]\!]_{V[X/A]}]} = [\![\psi]\!]_{V[X/A,Y/[\![\varphi]\!]_{V[X/\emptyset]}]}$ and by induction on ψ, $A \subseteq B \Rightarrow [\![\psi]\!]_{V[X/A,Y/W]} \subseteq [\![\psi]\!]_{V[X/B,Y/W]}$ and thus with $W = [\![\varphi]\!]_{V[X/\emptyset]}$ the result holds.
- when $sim(\psi, X) = 0$ we have $sim(\varphi, X) = sim(\psi, Y) = 1$, by lemma 1 we have $[\![\psi]\!]_{V[X/S',Y/S]} = [\![\psi]\!]_{V[X/\emptyset,Y/S]}$ and by induction we have $W \subseteq D \Rightarrow [\![\psi]\!]_{V[X/\emptyset,Y/W]} \subseteq [\![\psi]\!]_{V[X/\emptyset,Y/D]}$. But by induction we also have $A \subseteq B \Rightarrow [\![\varphi]\!]_{V[X/A]} \subseteq [\![\varphi]\!]_{V[X/B]}$ and thus with $W = [\![\varphi]\!]_{V[X/A]}$ and $D = [\![\varphi]\!]_{V[X/B]}$ we have $A \subseteq B \Rightarrow [\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/A]} = [\![\psi]\!]_{V[X/\emptyset,Y/[\![\varphi]\!]_{V[X/A]}]} \subseteq [\![\psi]\!]_{V[X/\emptyset,Y/[\![\varphi]\!]_{V[X/B]}]} = [\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/B]}$.

□

A.1.3 Theorem 1

Theorem 1. *Given a fixpoint expression $\mu(X = \varphi)$ that is valid in the sense of definition 17 and an environment V, the function $f(S) = [\![\varphi]\!]_{V[X \to S]}$ has the following properties:*

1. $\forall W \neq \emptyset \quad f(W) = \bigcup_{w \in W} f(\{w\})$
2. $\forall A, B, C \quad A = B \cup C \Rightarrow f(A) = f(B) \cup f(C)$
3. *f has a least fixpoint P with $P = [\![\mu(X = \varphi)]\!]_V$*

Proof. We will first prove by induction on the size of formula the following property: given a valid term φ, for all S we have $\forall m \in [\![\varphi]\!]_{V[X/S]} \quad \exists w_m \in S \quad m \in [\![\varphi]\!]_{V[X/\{w_m\}]}$.

- Using lemma 1 the property is clearly true for terms φ such that $sim(\varphi, X) = 0$. And the only variable Y with $sim(Y, X) = 1$ is X. For X the property trivially holds (with $w_m = m$).
- For unary operators $\varphi \in \{\beta_a^b(\varphi_1), \rho_a^b(\varphi_1), \pi_a(\varphi_1), \theta(\varphi_1, g : \mathcal{C} \to \mathcal{D})\}$ we have $m \in [\![\varphi]\!]_{V[X/S]}$ implies the existence of $m' \in [\![\varphi]\!]_{V[X/S]}$ such that m is the image of m' through this operator. By the induction hypothesis, for m' there is w such that $m' \in [\![\varphi_1]\!]_{V[X/\{w_{m'}\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w_{m'}\}]}$
- For a join operator $\varphi = \varphi_1 \bowtie \varphi_2$ we have that since $sim(\varphi, X) = sim(\varphi_1, X) + sim(\varphi_2, X)$ and $sim(\varphi, X) = 1$ then either $sim(\varphi_1, X) = 0 \wedge sim(\varphi_2, X) = 1$ or $sim(\varphi_2, X) = 0 \wedge sim(\varphi_1, X) = 1$. In either case $m \in [\![\varphi]\!]_{V[X/S]}$ implies the existence of $m_1 \in [\![\varphi_1]\!]_{V[X/S]}$ and $m_2 \in [\![\varphi_2]\!]_{V[X/S]}$ such that $m = m_1 + m_2$. For the $i \in \{1, 2\}$ such that $sim(\varphi_i) = 1$, there exists w such that $m_i \in [\![\varphi_i]\!]_{V[X/\{w_{m_i}\}]}$. For $j = 3 - i$ we have $sim(\varphi_j, X) = 0$ and thus $[\![\varphi_j]\!]_{V[X/S]} = [\![\varphi_j]\!]_{V[X/\emptyset]} = [\![\varphi_j]\!]_{V[X/\{w_{m_i}\}]}$ which means that in any case $m_1 \in [\![\varphi_1]\!]_{V[X/\{w_{m_i}\}]}$, $m_2 \in [\![\varphi_2]\!]_{V[X/\{w_{m_i}\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w_{m_i}\}]}$.
- For the term $\varphi = \varphi_1 \ \lambda \ \varphi_2$ with $\lambda \in \{ \bowtie, \backslash\backslash, \backslash, - \}$ any mapping $m \in [\![\varphi]\!]_{V[X/S]}$ is built using at least one mapping m_1 from $[\![\varphi_1]\!]_{V[X/S]}$. By induction, we have w such that $m_1 \in [\![\varphi]\!]_{V[X/\{w\}]}$. But $sim(\varphi_2, X) = 0$ thus $[\![\varphi_2]\!]_{V[X/S]} = [\![\varphi_2]\!]_{V[X/\{w\}]}$ and thus $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\{w\}]}$.

- For the term $\varphi = \varphi_1 \cup \varphi_2$, $m \in [\![\varphi]\!]_{V[X/S]}$ implies $m \in [\![\varphi_1]\!]_{V[X/S]}$ or $m \in [\![\varphi_2]\!]_{V[X/S]}$. By induction we have w such that $m \in [\![\varphi_1]\!]_{V[X/\{w\}]}$ or $m \in [\![\varphi_2]\!]_{V[X/\{w\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w\}]}$.
- Given the term $\mu(Y = \varphi)$ we have $sim(\mu(Y = \varphi)) = 0$ (since it cannot be 2) and thus the result.
- Finally for let $(Y = \varphi)$ in ψ and $m \in [\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]}$ we have that $sim(\text{let } (Y = \varphi) \text{ in } \psi)$ 1 that implies
  - either $sim(\psi, X) = 1$ and $sim(\psi, Y) = 0$, in which case $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]} = [\![\psi]\!]_{V[X/S,Y/[\![\varphi]\!]_{V[X/S]}]} = [\![\psi]\!]_{V[X/S,Y/\emptyset]}$ in which case the induction hypothesis applies on ψ and we have w such that $m \in [\![\psi]\!]_{V[X/\{w\},Y/\emptyset]}$ and thus $m \in [\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]}$;
  - or $sim(\psi, X) = 1$ and $sim(\varphi, X) = 0$, in which case $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]} = [\![\psi]\!]_{V[X/S,Y/[\![\varphi]\!]_{V[X/S]}]} = [\![\psi]\!]_{V[X/S,Y/[\![\varphi]\!]_{V[X/\emptyset]}]}$ and by induction on ψ we have w such that $m \in [\![\psi]\!]_{V[X/\{w\},Y/[\![\varphi]\!]_{V[X/\emptyset]}]} = [\![\psi]\!]_{V[X/\{w\},Y/[\![\varphi]\!]_{V[X/\{w\}]}]}$ and thus $m \in [\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X}$
  - or $sim(\psi, X) = 0$, $sim(\psi, Y) \leq 1$ and $sim(\varphi, X) = 1$, in which case $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/S]}$ $[\![\psi]\!]_{V[X/S,Y/[\![\varphi]\!]_{V[X/S]}]} = [\![\psi]\!]_{V[X/\emptyset,Y/[\![\varphi]\!]_{V[X/S]}]}$. By the induction hypothesis on ψ and Y we have that $m \in [\![\psi]\!]_{V[X/\emptyset,Y/[\![\varphi]\!]_{V[X/S]}]}$ implies that there exists $w' \in [\![\varphi]\!]_{V[X/S]}$ such that $[\![\psi]\!]_{V[X/\emptyset,Y/\{w'\}]}$. By reusing the induction hypothesis on $w' \in [\![\varphi]\!]_{V[X/S]}$ we have w such that $w' \in [\![\varphi]\!]_{V[X/\{w\}]}$. Since $\{w'\} \subseteq [\![\varphi]\!]_{V[X/\{w\}]}$, using lemma 2 we have $m \in [\![\psi]\!]_{V[X/\emptyset,Y/[\![\varphi]\!]_{V[X/\{w\}]}]} = [\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\{w\}]}$.

Now, with this property we have $f(W) \subseteq \bigcup_{w \in W} f(\{w\})$ but since $\{w\} \subseteq W$ we also have (lemma 2) $f(\{w\}) \subseteq f(W)$ and thus $f(W) = \bigcup_{w \in W} f(\{w\})$

Points 2. and 3. come easily as consequences of point 1. □

A.2 Relationship between μ-algebra and existing relational variants

A.2.1 Turing Hardness of the μ-algebra

Our language comprises a map operator $\theta(\varphi, g : \mathcal{C} \to \mathcal{D})$ where f is supposedly taken from a set of reasonable expressions. Such expressions should contain at least the function $x \to x+1$ as it is a function present in the SPARQL standard.

Adding a constant With just the function $g : x \to x + 1$ it is easy to add any constant to a column of a mapping. Let $Add(k, a, \varphi)$ be the term that adds k on the column a of the mapping solutions of φ.

To implement this function we can use the following recursive technique:

$Add(0, a, \varphi) = \varphi$

$Add(k + 1, a, \varphi) = \theta(\varphi, g : \{a\} \to \{a\})$

Note that here Add is not directly a μ-algebra term but a macro to generate μ-algebra terms.

Multiplying by 2 Now with this function we can craft a set of mappings representing the integers $\{a \to x, b \to 2 \times x\}$ with the term:

$$Times(2, a, b) = \mu(X = |a \to 0; b \to 0| \cup Add(1, a, Add(2, b, X)))$$

Note that once again $Times(2, a, b)$ is a macro that generates an actual μ-algebra term. Equipped with this set we can now multiply by 2, for instance to multiply a column a in a term φ by two:

$$Multiply(2, a, \varphi) = \rho_b^a \left(\pi_a \left(\varphi \bowtie Times(2, a, b)\right)\right)$$

Conversely, it is also possible to divide and stores the carry in the column c:

$$\begin{aligned} Divide(k, a, c, \varphi) = \pi_b \left(\rho_a^b \left(\varphi\right) \bowtie Times(k, a, b) \bowtie |c \to 0|\right) \\ \cup \pi_b \left(\rho_a^b \left(\varphi\right) \bowtie Add(1, a, Times(k, a, b)) \bowtie |c \to 1|\right) \end{aligned}$$

Adding 0-1 value Let φ be a μ-algebra term with a column a that either contains 0 or 1, we design $BinAdd(\varphi, a, b)$ the term that adds the values contained in the column a to the column b.

$$BinAdd(\varphi, a, b) = \sigma_{a=0}\left(\varphi\right) \cup \sigma_{a=1}\left(Add(1, b, \varphi)\right)$$

Encode the tape of the Turing Machine The tape of a TM is doubly infinite ruban that can store values. It is known that values can be restricted to 0 and 1 without loss of generality. We encore the right side of the tape $r_0, r_1 \ldots$ as the integer as $\sum_i r_i \times 2^i$ that will be stored in the r column of mappings. The left side of the tape $l_0, l_1, \ldots$ will also be encoded as $\sum_i l_i \times 2^i$ and will be stored in the l column.

Encode the transition At each step, a TM reads a value v and has a state q, then, depending on q and v it write a value w, change to a state q and then either moves left or right on the tape.

Given a TM we encode the transitions into two terms T_R (for the right transitions) and T_L (for the left transitions) in the following way: T_S is a union of mappings, each mapping describing a transition. In a mapping m in T_S, we have:

- $m(q)$ that describes the current state,
- $m(v)$ the current value,
- $m(q')$ the new state,
- $m(w)$ the value to write.

If a term X contains the mappings describing TM at some step, the following term computes the same TM after one step if it uses a right transition:

$$Step_R(X) = \pi_w \left(BinAdd(l, w, Multiply(2, l, Divide(2, r, v, \rho_{q'}^q \left(\pi_v \left(\pi_q \left(X \bowtie T_R\right)\right)\right)))\right)$$

Indeed:

- $\rho^{q}_{q'}(\pi_v(\pi_q(X \bowtie T_R)))$ fetch the new state into q and the value w to write.
- $\pi_w(BinAdd(l, w, Multiply(2, l, \cdot)))$ actually writes w
- $Divide(2, r, v, \cdot)$ fetches the new value.

Encoding the left transitions is done in a similar fashion. We suppose that all theses transitions are encoded into the term $Step(X)$ that computes one step of the TM:

$$Step(X) = Step_R(X) \cup Step_L(X)$$

Encoding the computation As we have seen, if the variable X contains mapping representing TM, we can simulate the computation of the next state of the TM by a μ-algebra term. It is also easy to encode the initial state of the TM as the left and right side of the tape are initially empty and therefore can be both represented as 0. The overall term that returns a mapping corresponding to the final step of the TM if it exists or returns an emptyset is:

$$\sigma_{q=q_f}\mu(X = |q \to q_0, v \to 0, l \to 0, m \to 0| \cup Step(X))$$

Note that if φ is linear in some variable X, then in all of our macros M, then $M(\varphi)$ also was (at most) linear in X and therefore the final term is indeed be linear.

A.2.2 The halting problem

We have seen that TM computations can be encoded into μ-algebra, yielding a result only when the TM halts. Since μ-algebra has a negation, we can negate the output above and returns a non-empty set when the TM does not halt which proves that the μ-algebra is strictly more powerful than a TM.

A.2.3 Expressive power of restricted μ-algebra

As we have seen, equipping the map operator with even very simple function leads to the expressive power of a Turing Machine. That is why it is interesting to consider the expressive power of the fragment of our language that does not comprise the map $\theta(\varphi, g : \mathcal{C} \to \mathcal{D})$ nor the aggregation $\Theta(f, g, \mathcal{C}, \mathcal{D})\,\varphi$ operators.

Syntactic sugar Some of our operator can be seen as syntactic sugar such as $\bowtie$ that is an union between a $\bowtie$ and a $\setminus$. From an expressive power point of view we can remove those. As explained in section 4.2.1, it is also possible to get rid of let binders.

Safe unions One of the ways that our terms differ from the relational algebra is because our terms might have "missing values". As we will see in the next chapter it is possible to compute the set of columns that might be defined for any given term (this set also depends on the set of columns in the environment where this term is evaluated). Therefore for each union where the two terms might have a different set of column, we complete those terms so that their type match using a new value $MISSING$ (this is very similar to the translation of SPARQL to SQL that uses NULL).

For the same reason, among the three difference operator $\setminus\setminus$, $\setminus$, $-$ we can restrain ourselves to $-$. Note that terms with only safe unions and $-$ can be exponentially less compact but from an expressive power point of view, it has no effect.

Emptysets As show in chapter 4, terms that contains emptysets are either equivalent to the emptyset or we can simplify them so that they do not contain the emptyset anymore.

Equivalent fragment We all the restriction proposed above, we can consider the following syntax:

$$
\begin{array}{lll}
\varphi & ::= & & \text{formula} \\
 & | & \varphi_1 \cup \varphi_2 & \text{union} \\
 & | & \varphi_1 - \varphi_2 & \text{set minus} \\
 & | & \varphi_1 \bowtie \varphi_2 & \text{join} \\
 & | & \pi_a(\varphi) & \text{column dropping} \\
 & | & \beta_a^b(\varphi) & \text{column multiplying} \\
 & | & \sigma_{filter}(\varphi) & \text{row filtering} \\
 & | & \mu(X = \varphi) & \text{fixpoint} \\
 & | & X & \text{variable} \\
 & | & |c_1 \to v_1, \ldots, c_n \to v_n| & \text{a mapping}
\end{array}
$$

Figure A.1: Grammar of restrictedμ-algebra

We call this fragment in this section restricted-μ and we study the effect of imposing the linearity in our term.

A.2.4 Datalog & μ-algebra expressive powers

In this section we present how to translate various Datalog into μ-algebra. The results presented here are not at the heart of our work and most of them are already known in the literature (with very similar statements and with similar proofs, see e.g. [AV91] or [AHV95] regarding Datalog and the while$^+$ language).

The only novelty of this proof relies in the proof that the linearity of restricted-μ actually reduce the expressive power. However to understand why we need to present a translation from Datalog to μ-algebra and back. We will therefore not rely on formal proofs but we will build some intuition and provide examples.

Datalog with only one IBD We recall in this section that datalog programs can always be transformed to programs that have only one recursive rule and one output rule (this is exercise 14.17 of the alice book [AHV95]).

Step 1: the n-aryfication Given a Datalog program P, we can always modify P so that all rules in P are n-ary for some n. To do that we simply take n to be the maximal arity over all the rules and extend all the rules with a constant c to match this arity.

For instance:

```
Path(1,2).
Access(1).
Access(X) :- Access(Y), Path(X,Y)
```

can be made 3-ary in the following way:

```
Path(1,2,c).
Access(1,c,c).
Access(X,c,c) :- Access(Y,c,c), Path(X,Y,c)
```

Step 2: one rule datalog Given a Datalog program P, we can always modify P so that there is only one recursive rule and one "output" rule in P. The idea is to first convert P into a n-ary program P' (for some n) then creates a unique $n+1$ rules that takes as its first argument the name of the rule. For instance, our running example becomes:

```
Rec(path,1,2,c).
Rec(access,1,c,c).
Rec(access,X,c,c) :- Rec(access,Y,c,c), Rec(path,X,Y,c).
Output(X) :- Rec(access,X,c,c).
```

From a derivation rule to μ-algebra It is a well-known fact that non-recurisve datalog and the relational algebra coincide (see e.g. chapter 14 of the alice book [AHV95]). Given a production $head(\bar{Y}) : -body_1(\bar{X}_1), \ldots, body_k(\bar{X}_k)$ we can translate $body_1(\bar{X}_1), \ldots, body_k(\bar{X}_k)$ using $k-1$ joins between each $body_i$, renames to rename arguments of $body_i$, antiprojections to remove existential variables, and filters for constants. Finally we use joins with constants for the constants of the head and renames for the variables.

For instance, if we translate the Datalog IBD *Rec* into a term *Rec* that has 4 columns (a_1, a_2, a_3 and a_4) the translation of the body

```
Rec(access,X,c,c) :- Rec(access,Y,c,c), Rec(path,X,Y,c).
```

is $\rho^Y_{a2}\left(\pi_{a_1}\left(\pi_{a_3}\left(\pi_{a_4}\left(\sigma_{a1=access\wedge a3=c\wedge a4=c}\left(Rec\right)\right)\right)\right)\right) \bowtie \pi_{a_1}\left(\pi_{a_4}\left(\rho^Y_{a3}\left(\rho^X_{a2}\left(\sigma_{a1=path\wedge a4=c}\left(Rec\right)\right)\right)\right)\right)$
The whole translation is (using *body* to denote the above term):
$\rho^{a2}_X\left(\pi_Y\left(body\right)\right) \bowtie |a3 \rightarrow c| \bowtie |a4 \rightarrow c| \bowtie |a1 \rightarrow access|$

From inflationary Datalog$^\neg$ to μ-algebra Given an inflationary-Datalog$^\neg$ program P that, *w.l.o.g.*, has recursive rule *Rec* and one output rule *Output* we can translate *Rec* to a fixpoint of the form $\mu(Rec = \varphi_1 \cup \ldots \varphi_k)$ where each φ_i corresponds to one derivation of the rule *Rec*. Finally we translate each production of Output into a term ψ (where Rec is replaced by the fixpoint above) and we generate a term that is the union of all these ψ.

Given our initial example we have the term O (here we cut the translation to ease the reading):

$$
\begin{array}{rcr}
B_1 & = & |a_1 \rightarrow path| \bowtie |a_2 \rightarrow 1| \bowtie |a_3 \rightarrow 2| \bowtie |a_4 \rightarrow c| \\
B_2 & = & |a_1 \rightarrow access| \bowtie |a_2 \rightarrow 1| \bowtie |a_3 \rightarrow c| \bowtie |a_4 \rightarrow c| \\
A_1 & = & \rho^Y_{a2}\left(\pi_{a_1}\left(\pi_{a_3}\left(\pi_{a_4}\left(\sigma_{a1=access\wedge a3=c\wedge a4=c}\left(Rec\right)\right)\right)\right)\right) \\
A_2 & = & \rho^Y_{a3}\left(\rho^X_{a2}\left(\pi_{a_1}\left(\pi_{a_4}\left(\sigma_{a1=path\wedge a4=c}\left(Rec\right)\right)\right)\right)\right) \\
B_3 & = & \rho^{a3}_X\left(\pi_Y\left(A_1 \bowtie A_2\right)\right) \\
B_4 & = & \mu(Rec = B_1 \cup B_2 \cup B3) \\
O & = & \pi_{a_1}\left(\pi_{a_3}\left(\pi_{a_4}\left(\sigma_{a_1=access}\left(B\right)_4\right)\right)\right)
\end{array}
$$

The semantics does coincide with inflationary-Datalog$^\neg$ because the formula $B_1 \cup B_2 \cup B_3$ captures the "immediate consequence" of the Datalog program.

From stratified Datalog to μ-algebra In a stratified Datalog program, each rule can be indexed with an integer n such that a negation of a rule indexed by k can only appear in the production rule of a term indexed with $k' > k$.

In the case of a stratified Datalog program, merging all the rules into one will break the stratification. The trick here is to operate stratum by stratum and translate the stratum i into a rule Rec_i. The resulting program will have one rule per stratum.

Just like in the inflationary case, each stratum i can be translated into a unique fixpoint $\mu(X_i = \varphi_i)$. The production rules of the stratum i can only reference to a Rec_j where $j \leq i$. We translate Rec_i into X_i and the Rec_j into $\mu(X_j = \varphi_j)$. Note that each φ_i can contain several occurrences of Rec_j with $j < i$ and that makes the translation exponential but all the fixpoints do are non mutually recursive and positive.

Let us consider the following example (already stratified):

```
Path(...) an EDB

Access_1(0).
Access_1(X) :- Access_1(Y),Path(Y,X)

Access_2(1).
Access_2(X) :- Access_2(Y),Path(Y,X), not Access_1(Y)
```

We translate Path into a term $\mu(X_0 = \varphi_0)$ (despite the fixpoint φ_0 is actually not recursive as Path is an EDB). Then we translate $Access_1$:

$$\mu\big(X_1 = |a_1 \to 0| \cup \rho_{a_2}^{a_1}\left(\pi_{a_1}\left(X_1 \bowtie Path\right)\right)\big)$$

Then we translate $Access_2$ (using $Access_1$ to denote the term above) :

$$\mu\big(X_2 = |a_1 \to 1| \cup \rho_{a_2}^{a_1}\left(\pi_{a_1}\left(X_2 \bowtie Path \triangleright Access_1\right)\right)\big)$$

From linear Datalog to restricted-μ Given a linear Datalog program, we can use the stratified translation. In the resulting term each φ_i is composed of $\psi_1 \cup \ldots \psi_k$ where each of the ψ_j corresponds to a linear production rule and thus contains at most one occurrence of X_i therefore our μ-algebra term is also linear (in addition to be recursive and positive as proven by the stratified translation). All in all, our term does belong to restricted-μ.

From restricted-μ to linear Datalog This direction is actually very simple once we know how to translate a term to a Datalog program, we just need to check that the resulting term is actually linear. To translate terms into Datalog, we work bottom-up associating each subterm φ to a Datalog rule. Datalog rules have columns that are indexed (there is a first column, a second, a third, etc.) while μ-algebra has column names. To handle this discrepancy, we suppose that we have calculated the type of each term (i.e. we compute a set of column names), then we order column names (any total order on the column names can be used).

The only difficulty here is the language of filters, in restricted-μ we actually impose no restriction on the filter conditions; for the translation we suppose that only the equality is used.

We thus recursively create production rules for each Datalog predicate $s_\varphi(\bar{T})$ corresponding the each term φ $s_\varphi(\bar{T})$ (where $\bar{T}$ is the ordered set of columns of the type of φ).

- For $\varphi = \varphi_1 \bowtie \varphi_2$ we create a rule for the join: $s_\varphi(\bar{T}) \leftarrow s_1(\bar{T_1}), s_2(\bar{T_2})$.
- For $\varphi = \varphi_1 \cup \varphi_2$ we have two production rules, one for each φ_i: $s_\varphi(\bar{T}) \leftarrow s_{\varphi_i}(\bar{T_i})$.
- For $\varphi = \varphi_1 \triangleright \varphi_2$ we create the rule $s_\varphi \leftarrow s_{\varphi_1}(\bar{T_1}), \neg s_{\varphi_2}(\bar{T_2})$.
- For $\varphi = \sigma_{a=b}(\varphi')$ we create the rule $s_\varphi(\bar{T_1}, b, \bar{T_2}) \leftarrow s_{\varphi'}(\bar{T_1}, b, \bar{T_2})$ if we suppose that the ordered type of φ' is $\bar{T_1}, a, \bar{T_2}$
- For $\varphi = \pi_p(\varphi')$ we create the rule $s_\varphi(\bar{T_\varphi}) \leftarrow s_{\varphi'}(\bar{T_{\varphi'}})$
- For $\varphi = \beta_a^b(\varphi')$ we create the rule $s_\varphi(\bar{T'}) \leftarrow s_{\varphi'}(\bar{T_{\varphi'}})$ where $\bar{T'}$ is $\bar{T_{\varphi'}}$ where we inserted a a in the place of where b will be stored.
- For $\varphi = \mu(X = \varphi')$ we create the rule $s_X(\bar{T}) \leftarrow s_{\varphi'}(\bar{T_{\varphi'}})$.
- For $\varphi = X$ we create the rule $s_X(\bar{T}) \leftarrow s_{\varphi'}(\bar{T_{\varphi'}})$.

Since the restricted-μ term is linear we can see that each production rule contain at most one subgoal that is recursive with the head.

A.3 Proofs of chapter 4

A.3.1 Lemma 6

Lemma 6. *Let C be such that $C \subseteq dom(w)$, for all $m \in [\![\varphi]\!]_{V[X/\{w\}]} \setminus [\![\varphi]\!]_{V[X/\emptyset]}$ there exists $p \in perm(\varphi, X, C)$ such that $\forall c\ p(c) = \bot \vee (m(c) \neq \bot \wedge m(c) = w(p(c)))$.*

Proof. We prove that by induction on the size of φ. In this proof, we note $s(\varphi, V, X, S) = [\![\varphi]\!]_{V[X/S]} \setminus [\![\varphi]\!]_{V[X/\emptyset]}$. When $\forall c\ p(c) = \bot \vee (m(c) \neq \bot \wedge m(c) = w(p(c)))$ we say that m is a p-image of w.

- When $sim(\varphi, X) = 0$ then $s(\varphi, X, \{w\}) = \emptyset$ (note that this includes but is not limited to $\varphi = \mu(Y = \varphi_1)$, $\varphi = |c_1 \to v_1, \ldots, c_n \to v_n|$, $\varphi = Y$ for $Y \neq X$).
- Given $\varphi = \varphi_1 \cup \varphi_2$ and $m \in s(\varphi, V, X, S)$ we have $m \in s(\varphi_1, V, X, S)$ or $m \in s(\varphi_2, V, X, S)$ and by induction the existence of $p \in perm(\varphi_1, X, C)$ or $p \in perm(\varphi_2, X, C)$. All in all, we have the existence of $p \in perm(\varphi, X, C)$ such that m is the p-image of w.
- Given $\varphi \in \{\varphi_1 \backslash\backslash \varphi_2, \varphi_1 \setminus \varphi_2, \varphi_1 - \varphi_2, \sigma_s(\varphi_1)\}$ we have $m \in s(\varphi, V, X, \{w\})$ which implies $m \in s(\varphi_1)$ and by induction m is a p-image of w for $p \in perm(\varphi_1, X, C) = perm(\varphi, X, C)$.

- Given $\varphi = \varphi_1 \bowtie \varphi_2$ and $m \in s(\varphi, V, X, \{w\})$. By definition $m \in [\![\varphi_1 \bowtie \varphi_2]\!]_{V[X/\{w\}]}$ for $i \in \{1, 2\}$ we have $m_1 \in [\![\varphi_1]\!]_{V[X/\{w\}]}$ and $m_2 \in [\![\varphi_2]\!]_{V[X/\{w\}]} \cup \{\{\}\}$ such that $m = m_1 + m_2$. Since $sim(\varphi_2, X) = 0$ we have $m_1 \in s(\varphi_1, V, X, \{w\})$. By induction we get $p \in perm(\varphi_1, X, C) = perm(\varphi, X, C)$ such that for all c, $p(c) = \bot \vee (m_1(c) \neq \bot \wedge m_1(c) = w(p(c)))$. But $m = m_1 + m_2$ implies $m_1(c) \neq \bot \Rightarrow (m(c) \neq \bot \wedge m(c) = m_1(c))$, by combining both we have the desired property: $p(c) = \bot \vee (m(c) \neq \bot \wedge m(c) = w(p(c)))$.

- Given $\varphi = \varphi_1 \bowtie \varphi_2$ and $m \in s(\varphi, V, X, \{w\})$. We have (m_1, m_2) such that $m = m_1 + m_2$ and exactly one of the $m_i \in s(\varphi_i, X, \{w\})$ (otherwise $sim(\varphi, X) \geq 2$). m_i is a p-image of some $p \in perm(\varphi_i, X, C)$. But $m = m_i + m_2$ implies $m_i(c) \neq \bot \Rightarrow (m(c) \neq \bot \wedge m(c) = m_i(c))$, by combining both we have the desired property: $p(c) = \bot \vee (m(c) \neq \bot \wedge m(c) = w(p(c)))$.

- For $\varphi = \pi_a(\varphi_1)$ we have $m \in s(\varphi, V, X, \{w\})$ implies $m' \in s(\varphi_1, V, X, \{w\})$ such that there exist $p' \in perm(\varphi_1, V, X, \{w\})$ with m' the p'-image of w and $m' = m$ except $m(a) = \bot$. But p with $p(a) = \bot$ and $p = p'$ elsewhere belongs to $perm(\varphi, V, X, \{w\})$ and m is a p-image of w.

- For $\varphi = \rho_a^b(\varphi_1)$ we have $m \in s(\varphi, V, X, \{w\})$ and $m' \in s(\varphi, V, X, \{w\})$ with $m' = m$ except $m'(a) = m(b)$ and $m'(b) = m(a)$. By induction for m' we have $p' \in perm(\varphi_1, V, X, \{w\})$ such that m' is the p'-image of w. By construction we have $p \in perm(\varphi, V, X, \{w\})$ such that m is a p-image of w: for $c \notin \{a, b\}$, $m(c) = m'(c) = w(p'(c)) = w(p(c))$ and $m(a) = m'(b) = w(p'(b)) = w(p(a))$.

- For $\varphi = \rho_a^b(\varphi_1)$ we have $m \in s(\varphi, V, X, \{w\})$ and $m' \in s(\varphi_1, V, X, \{w\})$ with $m' = m$ except $m'(a) = m(b)$ and $m'(b) = m(a)$. By induction on m' we have $p' \in perm(\varphi_1, V, X, \{w\})$ such that m' is the p'-image of w. By construction we have $p \in perm(\varphi, V, X, \{w\})$ such that m is a p-image of w: for $c \notin \{a, b\}$, $m(c) = m'(c) = w(p'(c)) = w(p(c))$ and $m(a) = m'(b) = w(p'(b)) = w(p(a))$.

- For $\varphi = \beta_a^b(\varphi_1)$ we have $m \in s(\varphi, V, X, \{w\})$ and $m' \in s(\varphi_1, V, X, \{w\})$ with $m' = m$ except $m(b) = m(a) = m'(a)$. By induction on m' we have $p' \in perm(\varphi_1, V, X, \{w\})$ such that m' is the p'-image of w. By construction we have $p \in perm(\varphi, V, X, \{w\})$ such that m is a p-image of w: for $c \neq b$, $m(c) = m'(c) = w(p'(c)) = w(p(c))$ and $m(b) = m'(a) = w(p'(a)) = w(p(b))$.

- For $\varphi = \theta(\varphi_1, g : \mathcal{C} \to \mathcal{D})$ we have $m \in s(\varphi, V, X, \{w\})$ and $m' \in s(\varphi_1, V, X, \{w\})$ with $m' = m$ except $m(c) = f(m')(c)$ for $c \in \mathcal{D}$ if $dom(m') \subseteq \mathcal{C}$. By induction on m' we have $p' \in perm(\varphi_1, V, X, \{w\})$ such that m' is the p'-image of w. By construction we have $p \in perm(\varphi, V, X, \{w\})$ such that m is a p-image of w: for $c \notin \mathcal{D}$, $m(c) = m'(c) = w(p'(c)) = w(p(c))$ and for $c \in \mathcal{D}$, $p(c) = \bot$ which thus proves that m is a p-image of w.

- For (let $(Y = \varphi)$ in ψ), we have:
  - Either $sim(\psi, Y) = 0$ and $f(\text{let } (Y = \varphi) \text{ in } \psi, V, X, \{w\}) = f(\psi, V, X, \{w\})$ gives us the results by induction ($perm(\psi, X, C) \subseteq perm(\text{let } (Y = \varphi) \text{ in } \psi)$).
  - Or $sim(\varphi, X) = 0$ but in this case $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\{w\}]} = [\![\psi]\!]_{V[X/\{w\}, Y/[\![\varphi]\!]_{V[X/\emptyset]}]}$ and the result holds by induction on $V' = V[Y/[\![\varphi]\!]_{V[X/\emptyset]}]$ and ψ.

- Or $sim(\psi, X) = 0$ and $sim(\varphi, X) = sim(\psi, Y) = 1$. Let $m \in f(\text{let } (Y = \varphi) \text{ in } \psi, V, X, \{w\})$ there exists w_Y such that $m \in f(\psi, V[X/\emptyset], Y, \{w_Y\})$ with $w_Y \in f(\varphi, V, X, \{w\})$. By induction we have p_1 such that w_Y is a p_1-image of w and $p_1 \in perm(\varphi, X, C)$. With $C = im(p_1)$ we have by induction the existence of p_2 such that m is the p_2 image of w_Y, by composing p_1 and p_2 we obtain the result.

- Finally, for $\varphi = X$, we have $m \in s(\varphi, V, X, \{w\})$ implies $m = w$ and since $C \subseteq dom(w)$ we do have that m is a p-image of w.

□

A.3.2 Lemma 7

Lemma 7. *Given a term φ, an environment V, a variable X and a mapping w, If $c \notin dom(w)$, $\forall Y, sim(\varphi, Y) = 0 \vee (\forall m \in V(Y) c \notin dom(m))$, and $canAdd(\varphi, X, c)$ then we have for all v (with $w(v) = w + \{c \to v\}$):*

1. $\forall m \in [\![\varphi]\!]_{V[X/\{w\}]}\ c \notin dom(m)$;
2. $[\![\varphi]\!]_{V[X/\{w\}]} = [\![\pi_c(\varphi)]\!]_{V[X/\{w(v)\}]}$;
3. $\forall m \in [\![\varphi]\!]_{V[X/\{w(v)\}]}\ \ c \notin dom(m) \vee m(c) = v$.

Proof. Point 1 raises absolutely no challenge: c is not in the domain of mapping in the environment and due to the definition of canAdd, c can only syntactically appear below a $\pi_c(\varphi)$. We will now prove points 2 & 3 by induction on the size of formula we have:

- For φ such that $sim(\varphi, X) = 0$ we have $[\![\varphi]\!]_{V[X/\{w\}]} = [\![\varphi]\!]_{V[X/\{w(v)\}]}$ and point 1 gives us 2 & 3.

- For $\varphi_1 \cup \varphi_2$, $[\![\varphi]\!]_{V[X/\{w\}]} = [\![\varphi_1]\!]_{V[X/\{w\}]} \cup [\![\varphi_2]\!]_{V[X/\{w\}]}$ and $canAdd(\varphi, X, c)$ implies $canAdd(\varphi_1, X, c)$ and $canAdd(\varphi_2, X, c)$ thus by induction $[\![\varphi]\!]_{V[X/\{w\}]} = [\![\pi_c(\varphi_1)]\!]_{V[X/\{w(v)\}]} \cup [\![\pi_c(\varphi_2)]\!]_{V[X/\{w(v)\}]} = [\![\pi_c(\varphi)]\!]_{V[X/\{w(v)\}]}$ and thus points 2 & 3.

- For $\varphi_1 \bowtie \varphi_2$, $[\![\varphi]\!]_{V[X/\{w\}]} = [\![\varphi_1 \bowtie \varphi_2]\!]_{V[X/\{w\}]} = \{a + b \mid a \in [\![\varphi_1]\!]_{V[X/\{w\}]} \wedge b \in [\![\varphi_2]\!]_{V[X/\{w\}]} \wedge a \sim b\}$. But for each pair $(a, b) \in [\![\varphi_1]\!]_{V[X/\{w\}]} \times [\![\varphi_2]\!]_{V[X/\{w\}]}$ we have a pair $(a', b') \in [\![\varphi_1]\!]_{V[X/\{w(v)\}]} \times [\![\varphi_2]\!]_{V[X/\{w(v)\}]}$ with $(a'(x) = a(x) \wedge b'(x) = b(x)) \vee x = c$. Since each of $a'(c)$ and $b'(c)$ are either undefined or equal to the value v and $a \sim b$ we have $a' \sim b'$ and if $c \in dom(a' + b')$ then $(a' + b')(c) = v$.

- For $\varphi_1\ \lambda\ \varphi_2$ with $\lambda \in \{\backslash\backslash, \backslash, \bowtie\}$ we have $sim(\varphi_2, X) = 0$ and by induction $[\![\varphi_1]\!]_{V[X/\{w\}]} = [\![\pi_c(\varphi_1)]\!]_{V[X/\{w(v)\}]}$. For $m \in [\![\varphi_2]\!]_{V[X/\{w(v)\}]}$ we have $c \notin dom(m)$ which implies $[\![\pi_c(\varphi)]\!]_{V[X/\{w(v)\}]} = [\![\pi_c(\varphi_1)\ \lambda\ \varphi_2]\!]_{V[X/\{w(v)\}]}$ (the compatibility will not change nor their shared domain). Note that $canAdd(\varphi_1 - \varphi_2, X, c)$ requires $sim(\varphi_1, X) = 0$ because otherwise c might be defined by φ_1 and change the set of solutions.

- For unary operators $\varphi \in \{\rho_a^b(\varphi)_1, \beta_a^b(\varphi)_1, \pi_a(\varphi)_1, \theta(\varphi_1, g : \mathcal{C} \to \mathcal{D})\sigma_f\varphi_1\}$ we ensure that $c \notin \{a, b\} \cup FC(f) \cup \mathcal{C} \cup \mathcal{D}$ and the result follows easily.

- For a variable, we either have $X \neq Y$ and the result is clear because of the constraint on $AV[Y]$ or $X = Y$ and the result comes from the definition of $w(v)$.

- For a let $(Y = \varphi)$ in ψ we have, as usual, three cases:

  - Either $sim(\psi, Y) = 0$ in which case $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\{w\}]} = [\![\psi]\!]_{V[X/\{w\}]}$ and $[\![\pi_c(\text{let } (Y = \varphi) \text{ in } \psi)]\!]_{V[X/\{w(v)\}]} = [\![\pi_c(\psi)]\!]_{V[X/\{w(v)\}]}$ and by induction $[\![\psi]\!]_{V[X/\{w\}]} = [\![\pi_c(\psi)]\!]_{V[X/\{w(v)\}]}$ and thus the result
  - Either $sim(\varphi, X) = 0$ in which case $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\{w\}]} = [\![\psi]\!]_{V[X/\{w\},Y/[\![\varphi]\!]_{V[X/\{w\}]}]} = [\![\psi]\!]_{V[X/\{w\},Y/[\![\varphi]\!]_{V[X/\emptyset]}]}$ and $[\![\pi_c(\text{let } (Y = \varphi) \text{ in } \psi)]\!]_{V[X/\{w(v)\}]} = [\![\pi_c(\psi)]\!]_{V[X/\{w(v)\},Y/[\![\varphi]\!]_{V[X/\emptyset]}]}$. Using the induction hypothesis on ψ and $V' = V[Y/[\![\varphi]\!]_{V[X/\emptyset]}]$ we have $[\![\pi_c(\psi)]\!]_{V'[X/\{w(v)\}]} = [\![\psi]\!]_{V'[X/\{w\}]}$ and $\forall m \in [\![\pi_c(\psi)]\!]_{V'[X/\{w(v)\}]} c \notin dom(m) \vee m(c) = v$.
  - Finally, when $sim(\psi, X) = 0$ we have: $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\{w\}]} = [\![\psi]\!]_{V[X/\emptyset,Y/[\![\varphi]\!]_{V[X/\{w\}]}]}$ and $[\![\psi]\!]_{V[X/\emptyset,Y/[\![\varphi]\!]_{V[X/\{w\}]}]} = \bigcup_{m_Y \in [\![\varphi]\!]_{V[X/\{w\}]}} [\![\psi]\!]_{V[X/\emptyset,Y/\{m_Y\}]}$.

    Similarly, $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\{w(v)\}]} = \bigcup_{m'_Y \in [\![\varphi]\!]_{V[X/\{w(v)\}]}} [\![\psi]\!]_{V[X/\emptyset,Y/\{m'_Y\}]}$

    By induction on φ, for each m_Y we have m'_Y in $[\![\varphi]\!]_{V[X/\{w(v)\}]}$ and conversely for each $m'_Y \in [\![\varphi]\!]_{V[X/\{w(v)\}]}$ we have $m_Y \in [\![\varphi]\!]_{V[X/\{w\}]}$ with either $m'_Y = m_Y$ or $m'_Y = m_Y + \{c \to v\}$. Since there is a one to one-or-two correspondence between m_Y and m'_Y we only need to prove points 2 & 3 for each m_Y.

    When $m'_Y = m_Y$ then the property is trivial: $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\emptyset,Y/\{m_Y\}]} = [\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_{V[X/\emptyset,Y/\{m'_Y\}]} = [\![\pi_c(\text{let } (Y = \varphi) \text{ in } \psi)]\!]_{V[X/\emptyset,Y/\{m_Y\}]}$.

    When $m'_Y = m_Y + \{c \to v\}$ we apply the induction hypothesis on ψ, $V[X/\emptyset]$, Y and m_Y (c and v unchanged). It gives us that $[\![\psi]\!]_{V[X/\emptyset,Y/\{m_Y\}]} = [\![\pi_c(\psi)]\!]_{V[X/\emptyset,Y/\{m'_Y\}]}$ with $\forall m \in [\![\psi]\!]_{V[X/\emptyset,Y/\{m'_Y\}]}$ $c \notin dom(m) \vee m(c) = v$. Therefore we have points 2 & 3.

□

A.3.3 Lemma 8

Lemma 8. *Let φ be a μ-algebra term, X a variable, c a column and C a set of columns with $c \notin C$ and $canAdd(\varphi, X, c)$ then $perm(\varphi, X, C \cup \{c\}) = \{p \cup \{c \to c\} \mid p \in perm(\varphi, X, C)\}$.*

Proof. We prove this lemma by induction on the size of φ.

- For $\varphi \in \{\varphi_1 \cup \varphi_2, \varphi_1 \backslash\backslash \varphi_2, \varphi_1 \setminus \varphi_2, \varphi_1 \bowtie \varphi_2, \varphi_1 \ltimes \varphi_2\}$ the result is obvious following the definition of *perm*.

- For $\varphi \in \{\rho_a^b(\varphi_1), \pi_a(\varphi_1), \theta(\varphi_1, g : \mathcal{C} \to \mathcal{D}), \sigma_f(\varphi_1)\}$, we have $canAdd(\varphi, X, c)$ which implies that $perm(\varphi, X, C \cup \{c\})$ is the image of $perm(\varphi_1, X, C)$ via a transformation that leave the c image unchanged.

- For $\varphi \in \{\mu(Y = \varphi_1), \emptyset, |c_1 \to v_1, \ldots, c_n \to v_n|\}$, we have $perm(\varphi, X, C \cup \{c\}) = perm(\varphi, X, C) = \emptyset$.

- For $\varphi = X$ we have $perm(X, X, C \cup \{c\}) = \{x \to x \mid x \in C\{c\}\} = \{p \cup \{c \to c\} \mid p \in perm(X, X, C)\}$

- For a let $(Y = \varphi)$ in ψ we have $perm(\text{let } (Y = \varphi) \text{ in } \psi, X, C \cup \{c\}) = perm(\psi, X, C \cup \{c\}) \cup \{p_2 \circ p_1 \mid p_1 \in perm(\varphi, X, C \cup \{c\})\ p_2 \in perm(\psi, Y, im(p_1))\}$. By induction $perm(\psi, X, C) = perm(\psi, X, C \cup \{c\})$ and $perm(\varphi, X, C \cup \{c\}) = \{p \cup \{c \to c\} \mid p \in perm(\varphi, X, C)\}$ which means $\{p_2 \circ p_1 \mid p_1 \in perm(\varphi, X, C \cup \{c\})\ p_2 \in perm(\psi, Y, im(p_1))\} = \{p_2 \circ (p_1 \cup \{c \to c\}) \mid p_1 \in perm(\varphi, X, C)\ p_2 \in perm(\psi, Y, im(p_1) \cup \{c\})\}$.

  By induction $perm(\psi, Y, im(p_1) \cup \{c\}) = \{p \cup \{c \to c\} \mid p_2 \in perm(\psi, Y, im(p_1))\}$ and thus $\{p_2 \circ p_1 \mid p_1 \in perm(\varphi, X, C \cup \{c\})\ p_2 \in perm(\psi, Y, im(p_1))\} =$
  $\{(p_2 \cup \{c \to c\}) \circ (p_1 \cup \{c \to c\}) \mid p_1 \in perm(\varphi, X, C)\ p_2 \in perm(\psi, Y, im(p_1))\} =$
  $\{(p_2 \circ p_1) \cup \{c \to c\} \mid p_1 \in perm(\varphi, X, C)\ p_2 \in perm(\psi, Y, im(p_1))\}$

  All in all, $perm(\text{let } (Y = \varphi) \text{ in } \psi, X, C \cup \{c\}) = \{p \cup \{c \to c\} \mid p \in perm(\psi, X, C)\} \cup \{(p_2 \circ p_1) \cup \{c \to c\} \mid p_1 \in perm(\varphi, X, C)\ p_2 \in perm(\psi, Y, im(p_1))\} = \{p \cup \{c \to c\} \mid p \in perm(\text{let } (Y = \varphi) \text{ in } \psi, X, C)\}$.

□

A.3.4 Theorem 2

Theorem 2. *Let $\mu(X = \varphi)$ be a fixpoint, V an environment, C and a filter f with:*

1. $\forall m \in [\![\mu(X = \varphi)]\!]_V \quad C \subseteq dom(m)$
2. $\forall p \in perm(\varphi, X, C) \quad \forall d \in FC(f) \quad p(d) = d$

we have: $[\![\sigma_f(\mu(X = \varphi))]\!]_V = [\![\mu(X = \sigma_f(\varphi))]\!]_V$.

Proof. We define $U_0 = V_0 = \emptyset$ and $U_{i+1} = [\![\varphi]\!]_{V[X/U_i]}$, $V_{i+1} = [\![\sigma_f(\varphi)]\!]_{V[X/V_i]}$ and we prove that $\{m \mid m \in U_i \wedge f(m) = \top\} = V_i$ by induction on i. For $i = 0$, the result is clear. Let $i \in \mathbb{N}$, we have $V_{i+1} = [\![\varphi]\!]_{V[X/V_i]}$ and $\{m \mid m \in U_{i+1} \wedge f(m) = \top\} = [\![\sigma_f(\varphi)]\!]_{V[X/U_i]}$. Clearly $V_i \subseteq U_i$ and thus $V_{i+1} \subseteq [\![\sigma_f(\varphi)]\!]_{V[X/U_i]}$ (thanks to the monotony of fixpoints). Let $m \in [\![\sigma_f(\varphi)]\!]_{V[X/U_i]}$ there are two cases:

- either $m \in [\![\sigma_f(\varphi)]\!]_{V[X/\emptyset]} = V_1 \subseteq V_{i+1}$
- or we have $w \in U_i$ such that $m \in [\![\sigma_f(\varphi)]\!]_{V[X/\{w\}]} \setminus [\![\sigma_f(\varphi)]\!]_{V[X/\emptyset]}$. By lemma 6 we have $p \in perm(\varphi, X, C)$ such that $\forall d \in FC(f) \quad m(d) = w(p(d)) \vee p(d) = \bot$, but $FC(f) \subseteq D$ and thus $\forall d \in FC(f) \quad m(d) = w(d)$. If $eval(f)(m) = \top$ and $\forall d \in FC(f) \quad w(d) = m(d)$ then $eval(f)(w) = \top$ and thus $w \in V_i$ which means $m \in V_{i+1}$.

In all cases $\{m \mid m \in U_i \wedge f(m) = \top\} = V_i$ and $[\![\sigma_f(\mu(X = \varphi))]\!]_V = \cup_{i \in \mathbb{N}} \{m \mid m \in U_i \wedge f(m) = \top\} = \cup_{i \in \mathbb{N}} V_i = [\![\mu(X = \sigma_f \varphi)]\!]_V$.

□

A.3.5 Lemma 9

Lemma 9. *Let φ and ψ be μ-algebra terms, V an environment and D and C be sets of columns such that:*

1. $\forall m \in [\![\psi]\!]_V \quad dom(m) \subseteq D \subseteq C$

2. $\forall m \in [\![\mu(X = \varphi)]\!]_V \quad D \subseteq C \subseteq dom(m)$

3. $\forall p \in perm(\varphi, X, C), c \in D \quad p(c) = c$

Then we have $[\![\psi \bowtie \mu(X = \varphi)]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$.

Proof. Let $U_0 = V_0 = W_0 = \emptyset$, $U_{i+1} = [\![\varphi]\!]_{V[X/U_i]}$, $V_{i+1} = [\![\varphi \bowtie \psi]\!]_{V[X/V_i]}$ and $W_{i+1} = [\![\varphi \bowtie \psi]\!]_{V[X/U_i]} = \{m_1 + m_2 \mid m_1 \in U_{i+1} \;\; m_2 \in [\![\psi]\!]_{V[X/\emptyset]}\}$.

Let us prove $W_i = V_i$ and $V_i \subseteq U_i$ by induction on i.

For $i = 0$, it is trivial and for $i = 1$ we have $W_1 = [\![\varphi \bowtie \psi]\!]_{V[X/\emptyset]} = V_1$. Now let $i \geq 1$.

$V_i \subseteq U_i \Rightarrow [\![\varphi]\!]_{V[X/V_i]} \subseteq [\![\varphi]\!]_{V[X/U_i]}$ and thanks to the requirement 1 & 2 we have $[\![\psi \bowtie \varphi]\!]_{V[X/A]} \subseteq [\![\varphi]\!]_{V[X/A]}$ for $A \subseteq [\![\mu(X = \varphi)]\!]_V$ and thus $V_{i+1} \subseteq U_{i+1}$.

$W_{i+1} = W_{i+1} \setminus W_1 \cup W_1$ but $W_1 = V_1 \subseteq V_{i+1}$ so let $m \in W_{i+1} \setminus W_1$ and lets prove that $m \in V_{i+1}$.

We have $m_1 \in U_{i+1}$ and $m_2 \in [\![\psi]\!]_V$ such that $m = m_1 + m_2$ and $m_1 \sim m_2$. But $dom(m_2) \subseteq dom(m_1)$ and thus $m = m_1$. Since $m = m_1 \in [\![\varphi]\!]_{V[X/U_i]} \setminus [\![\varphi]\!]_{V[X/\emptyset]}$ we have $u \in U_i$ and $p \in perm(\varphi, X, C)$ such that $m \in [\![\varphi]\!]_{V[X/\{u\}]}$ and $\forall c \in D \quad m(c) = u(c)$. Since $dom(m_2) \subseteq D$ and $m_2 \sim m$ we have $m_2 \sim u$ and thus $u \in W_i$. By induction we have $W_i = V_i$ thus $u \in V_i$. Therefore we have $m \in [\![\varphi]\!]_{V[X/\{u\}]}$ and $m_2 \in [\![\psi]\!]_{V[X/\{u\}]}$ with $m_2 \sim m$ and $m = m + m_2$ thus $m \in [\![\varphi \bowtie \psi]\!]_{V[X/\{u\}]} \subseteq [\![\varphi \bowtie \psi]\!]_{V[X/V_i]} = V_{i+1}$.

□

A.3.6 Lemma 10

Lemma 10. *Let φ and ψ be μ-algebra terms, V an environment, c a column and D and C be sets of columns such that:*

1. $\forall m \in [\![\psi]\!]_V \quad dom(m) \subseteq D \cup \{c\} \subseteq C \cup \{c\}$

2. $\forall m \in [\![\mu(X = \varphi)]\!]_V \quad D \subseteq C \subseteq dom(m) \wedge c \notin dom(m)$

3. $\forall p \in perm(\varphi, X, C), c \in D \quad p(c) = c$

4. $canAdd(\varphi, X, c) = \top$

Then we have $[\![\psi \bowtie \mu(X = \varphi)]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$.

Proof. Let $U_0 = V_0 = W_0 = \emptyset$, $U_{i+1} = [\![\varphi]\!]_{V[X/U_i]}$, $V_{i+1} = [\![\varphi \bowtie \psi]\!]_{V[X/V_i]}$ and $W_{i+1} = [\![\varphi \bowtie \psi]\!]_{V[X/U_i]} = \{m_1 + m_2 \mid m_1 \in U_{i+1} \;\; m_2 \in [\![\psi]\!]_{V[X/\emptyset]}\}$.

For $i = 0$ and $i = 1$ it is clear that $V_i = W_i$. Lets prove that $V_i \subseteq W_i$. Let $m \in V_i$ we have $m_1 \in [\![\varphi]\!]_{V[X/V_i]}, m_2 \in [\![\psi]\!]_{V[X/\emptyset]}$ such that $m = m_1 + m_2$. Given that $m_1 \in [\![\varphi]\!]_{V[X/V_i]}$ we have $v \in V_i$ with $m \in [\![\varphi]\!]_{V[X/\{v\}]}$ and with $V_i = W_i$ we have $u \in U_i$ and $w \in [\![\psi]\!]_{V[X/\emptyset]}$ such that $v = u + w$.

Depending on whether $c \in dom(v)$ we have:

- If $c \notin dom(v)$ then $v = u$ and $m_1 \in [\![\varphi]\!]_{V[X/U_i]}$ and thus $m \in W_{i+1}$.

- If $c \in dom(v)$ then $v = u + \{c \to v(c)\}$. But with lemma 7 we have $[\![\pi_c(\varphi)]\!]_v = [\![\varphi]\!]_{V[X/\{u\}]}$ which means there a $m' \in [\![\varphi]\!]_{V[X/U_i]}$ with either $m' = m_1$ or $m_1 = m' + \{c \to v\}$. In both cases $m = m' + m_2$ and thus $m \in W_{i+1}$.

We now need to prove that $m \in W_{i+1} \Rightarrow m \in V_{i+1}$. If $m \in W_1$ it is clear.

Let $m \in W_{i+1} \setminus W_1$ we have $m_1 \in U_{i+1}$ and $m_2 \in [\![\psi]\!]_{V[X/\emptyset]}$. Thanks to requirement 1 & 2 we have $m = m_1 + \{c \to m_2(v)\}$ or $m = m_1$. Since $m_1 \in [\![\varphi]\!]_{V[X/U_i]} \setminus [\![\varphi]\!]_{V[X/\emptyset]}$ we have $u \in U_i$ and $p \in perm(\varphi, X, C)$ such that $m \in [\![\varphi]\!]_{V[X/\{u\}]}$ and $\forall d \in D$ we have $m_1(d) = u(d)$. Since $dom(m_2) \subseteq D \cup \{c\}$, $c \notin dom(u)$, $m_1 \sim m_2$ and $m_1 \sim u$ we have $u \sim m_2$ and thus $u + m_2 \in V_i$.

Depending on whether $c \in dom(m)$ there are two cases:

- if $u + m_2 = u$ then $u \in V_i$ and $m_1 \in [\![\varphi]\!]_{V[X/\{u\}]} \subseteq [\![\varphi]\!]_{V[X/V_i]}$
- if $u+m_2 = u+\{c \to v\}$ then with lemma 7 we have $[\![\pi_c(\varphi)]\!]_{V[X/\{u+\{c\to v\}\}]} = [\![\varphi]\!]_{V[X/\{u\}]}$ and $\forall \in [\![\varphi]\!]_{V[X/\{u+\{c\to v\}\}]}$ $c \notin dom(m) \vee m(c) = v$ which means either m_1 or $m_1 + m_2$ are in $[\![\varphi]\!]_{V[X/\{u+\{c\to v\}\}]} \subseteq [\![\varphi]\!]_{V[X/V_i]}$.

In all cases we have $m_2 \in [\![\psi]\!]_{V[X/\emptyset]}$ and thus $m_1 + m_2 \in V_{i+1}$.

□

A.3.7 Theorem 3

Theorem 3. *Let $\mu(X = \varphi)$ be a fixpoint and ψ be a μ-algebra term, V an environment, and C, D, E sets with:*

1. $\forall m \in [\![\psi]\!]_V \quad dom(m) = E \cup D$
2. $\forall m \in [\![\mu(X = \varphi)]\!]_V \quad (D \subseteq C \subseteq dom(m)) \ \wedge \ (dom(m) \cap E = \emptyset)$
3. $\forall p \in perm(\varphi, X, C), d \in D \quad p(d) = d$
4. $\forall c \in E \quad canAdd(\varphi, X, c) = \top$
5. $sim(\psi, X) = 0$

we have: $[\![\psi \bowtie \mu(X = \varphi)]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$.

Proof. We prove recursively on the number of columns defined by ψ that $[\![\mu(X = \varphi) \bowtie \psi]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$.

If ψ defines 0 columns, we have either $[\![\psi]\!]_V = \emptyset$ or $[\![\psi]\!]_V = \{\{\}\}$ and in both cases we have $[\![\mu(X = \varphi) \bowtie \psi]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$.

Now, we suppose that ψ defines $n+1$ columns $c_1, \ldots, c_{n+1}$. Since $\{c_1, \ldots, c_{n+1}\}$ can be split into E and D verifying the conditions of the theorem then $\{c_1, \ldots, c_n\}$ can also be split into $E \setminus \{c_{n+1}\}$ and $D \setminus \{c_{n+1}\}$ and using the induction hypothesis we have $[\![\pi_{c_{n+1}}(\psi) \bowtie \mu(X = \varphi)]\!]_V = [\![\mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi)\big)]\!]_V$.

From requirement 1 of the theorem we have that $[\![\psi \bowtie \pi_{c_{n+1}}(\psi)]\!]_V = \{m_1 + m_2 \mid m_1 \in [\![\psi]\!]_V \ m_2 \in [\![\pi_{c_{n+1}}(\psi)]\!]_V \ m_1 \sim m_2\}$ but $\forall m_1 \in [\![\psi]\!]_V, m_2 \in [\![\pi_{c_{n+1}}(\psi)]\!]_V \quad dom(m_1) = dom(m_2) \cup \{c_{n+1}\}$ therefore $m_1 + m_2 = m_1$ and thus $[\![\psi \bowtie \pi_{c_{n+1}}(\psi)]\!]_V = [\![\psi]\!]_V$.

Finally this gives us $[\![\psi \bowtie \mu(X = \varphi)]\!]_V = [\![\big(\psi \bowtie \pi_{c_{n+1}}(\psi)\big) \bowtie \mu(X = \varphi)]\!]_V = [\![\psi \bowtie \mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi)\big)]\!]_V$ and $[\![\mu\big(X = \psi \bowtie \varphi \bowtie \pi_{c_{n+1}}(\psi)\big)]\!]_V = [\![\mu(X = \psi \bowtie \varphi)]\!]_V$ therefore we only need to prove that
$[\![\psi \bowtie \mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi)\big)]\!]_V = [\![\mu\big(X = \psi \bowtie \varphi \bowtie \pi_{c_{n+1}}(\psi)\big)]\!]_V$.

- When $c_{n+1} \in E$

1. Let $m \in [\![\mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi)\big)]\!]_V$ we have $m \in [\![\mu(X = \varphi) \bowtie \pi_{c_{n+1}}(\psi)]\!]_V$ and since $\forall m_1 \in [\![\mu(X = \varphi)]\!]_V \quad c_{n+1} \notin dom(m_1)$ and $\forall m_2 \in [\![\pi_{c_{n+1}}(\psi)]\!]_V \quad c_{n+1} \notin dom(m_2) \wedge dom(m_2) = \{c_1, \ldots, c_n\}$ in all cases $c \notin dom(m) \wedge \{c_1, \ldots, c_n\} \subseteq dom(m)$.
2. We can prove recursively that $\forall p \in perm(\varphi, X, C \cup \{c_1, \ldots, c_n\}), \forall i \quad p(c_i) = c_i$: for all $c_i \in D$ we have that by requirement 3 and we can add each of the $c_i \in E$ via lemma 8.
3. $canAdd(\varphi \bowtie \pi_{c_{n+1}}(\psi), X, c) = canAdd(\varphi, X, c_{n+1}) \wedge (sim(\psi, X) = 0) = \top$.

We are therefore in the conditions of lemma 10, which gives us that $[\![\mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi)\big) \bowtie \psi]\!]_V = [\![\mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi) \bowtie \psi\big)]\!]_V$ and concludes our proof when $c_{n+1} \in E$.

- When $c \in D$

1. Let $m \in [\![\mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi)\big)]\!]_V$ we have $m \in [\![\mu(X = \varphi) \bowtie \pi_{c_{n+1}}(\psi)]\!]_V$ and since $\forall m_1 \in [\![\mu(X = \varphi)]\!]_V \quad c_{n+1} \in dom(m_1)$ and $\forall m_2 \in [\![\pi_{c_{n+1}}(\psi)]\!]_V \quad dom(m_2) = \{c_1, \ldots, c_n\}$ in all cases $\{c_1, \ldots, c_{n+1}\} \subseteq dom(m)$.
2. We can prove recursively that $\forall p \in perm(\varphi, X, C \cup \{c_1, \ldots, c_{n+1}\}), \forall i \quad p(c_i) = c_i$: for all $c_i \in D$ we have that by requirement 3 and we can add each of the $c_i \in E$ via lemma 8.

We are therefore in the conditions of lemma 9, which gives us that $[\![\mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi)\big) \bowtie \psi]\!]_V = [\![\mu\big(X = \varphi \bowtie \pi_{c_{n+1}}(\psi) \bowtie \psi\big)]\!]_V$ and concludes our proof when $c_{n+1} \in D$.

In all cases we have $[\![\mu(X = \varphi) \bowtie \psi]\!]_V = [\![\mu(X = \varphi \bowtie \psi)]\!]_V$. □

A.3.8 lemma 11

Lemma 11. *A term φ* syntactically recursive *in the variable X is recursive in X, i.e.* $\big(rec(\varphi, X) = \top\big) \Rightarrow \big(\forall V : [\![\varphi]\!]_{V[X/\emptyset]} = \emptyset\big)$.

Proof. By induction on the size of φ such that $rec(\varphi, X) = \top$:

- $[\![\varphi_1 \setminus\setminus \varphi_2]\!]_{V[X/\emptyset]} \subseteq [\![\varphi_1]\!]_{V[X/\emptyset]} = \emptyset$
- $[\![\varphi_1 \setminus \varphi_2]\!]_{V[X/\emptyset]} \subseteq [\![\varphi_1]\!]_{V[X/\emptyset]} = \emptyset$
- $[\![\varphi_1 - \varphi_2]\!]_{V[X/\emptyset]} \subseteq [\![\varphi_1]\!]_{V[X/\emptyset]} = \emptyset$
- $[\![\varphi_1]\!]_{V[X/\emptyset]} = \emptyset$ implies $[\![\varphi_1 \ltimes \varphi_2]\!]_{V[X/\emptyset]} = \emptyset$
- $[\![\varphi_1 \cup \varphi_2]\!]_{V[X/\emptyset]} = [\![\varphi_1]\!]_{V[X/\emptyset]} \cup [\![\varphi_2]\!]_{V[X/\emptyset]} = \emptyset$
- $rec(\varphi_1, X) = \top$ or $rec(\varphi_2, X) = \top$ implies $[\![\varphi_1]\!]_{V[X/\emptyset]} = \emptyset$ or $[\![\varphi_2]\!]_{V[X/\emptyset]} = \emptyset$ and either of them implies $[\![\varphi_1 \bowtie \varphi_2]\!]_{V[X/\emptyset]} = \emptyset$
- For $\varphi \in \{\rho_a^b(\varphi_1), \pi_a(\varphi_1), \theta(\varphi_1, g : \mathcal{C} \to \mathcal{D}), \beta_a^b(\varphi_1)\}$ we have $rec(\varphi, X) = rec(\varphi_1, X)$ and $[\![\varphi_1]\!]_{V[X/\emptyset]} = \emptyset$ implies $[\![\varphi]\!]_{V[X/\emptyset]} = \emptyset$.

- For let $(Y = \varphi)$ in ψ we have:
  - either $rec(\psi, X) = \top$ and thus by induction on $V' = V[Y/[\![\varphi]\!]_{V[X/\emptyset]}]$ we have $[\![\psi]\!]_{V'} = \emptyset$ and thus $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_V = [\![\psi]\!]_{V[Y/[\![\varphi]\!]_{V[X/\emptyset]}, X/\emptyset]} = \emptyset$.
  - or $rec(\psi, Y) = rec(\varphi, X) = \top$ and thus by induction on V, X we have $[\![\varphi]\!]_{V[X/\emptyset]} = \emptyset$ and by induction on $V' = V[X/\emptyset], Y$ we have $[\![\psi]\!]_{V'[Y/\emptyset]} = \emptyset$ and thus $[\![\text{let } (Y = \varphi) \text{ in } \psi]\!]_V = [\![\psi]\!]_{V[Y/[\![\varphi]\!]_{V[X/\emptyset]}, X/\emptyset]} = [\![\psi]\!]_{V[Y/\emptyset, X/\emptyset]} = \emptyset$.
- For $\emptyset$, Y, $|c_1 \to v_1, \ldots, c_n \to v_n|$ and $\mu(Y = \varphi)$ the result is clear.

□

A.3.9 Theorem 4

Theorem 4. *Let $\mu(X = \varphi \cup \psi)$ be a decomposed fixpoint with φ its constant part, ψ its recursive part and let κ be a μ-algebra term, V an environment, and C, D, E sets with:*

1. $\forall m \in [\![\kappa]\!]_V \quad dom(m) = E \cup D$
2. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \quad (D \subseteq C \subseteq dom(m)) \ \wedge \ (dom(m) \cap E = \emptyset)$
3. $\forall p \in perm(\psi, X, C), d \in D \quad p(d) = d$
4. $\forall c \in E \quad canAdd(\psi, X, c) = \top$
5. $sim(\kappa, X) = 0$

we have: $[\![\kappa \bowtie \mu(X = \varphi \cup \psi)]\!]_V = [\![\mu(X = \varphi \bowtie \kappa \cup \psi)]\!]_V$.

Proof. First we have by the general theorem3 on join and fixpoint that $[\![\mu(X = \varphi \cup \psi) \bowtie \kappa]\!]_V = [\![\mu(X = (\varphi \cup \psi) \bowtie \kappa)]\!]_V = [\![\mu(X = (\varphi \bowtie \kappa) \cup (\psi \bowtie \kappa))]\!]_V = \bigcup_i U_i$ where $U_0 = \emptyset$ and $U_{i+1} = [\![(\varphi \bowtie \kappa) \cup (\psi \bowtie \kappa)]\!]_{V[X/U_i]}$.

It is clear that $[\![\psi \bowtie \kappa]\!]_{V[X/U_i]} \subseteq [\![\psi]\!]_{V[X/U_i]}$ since the domain of mappings of $[\![\kappa]\!]_V$ is always included in the domain of mappings of $[\![\mu(X = \varphi \cup \psi)]\!]_V$. Let us show that $[\![\psi]\!]_{V[X/U_i]} \subseteq [\![\psi \bowtie \kappa]\!]_{V[X/U_i]}$.

For $i = 0$ we have $[\![\psi]\!]_{V[X/U_i]} = [\![\psi]\!]_{V[X/\emptyset]} = 0$.

For $U_{i+1} = [\![\psi \bowtie \kappa]\!]_{V[X/U_i]} \cup U_1$. But for $m \in [\![\psi]\!]_{V[X/U_i]}$, we have $m \in [\![\psi]\!]_{V[X/\{u\}]}$ for some $u \in U_i$ (Theorem 1). Since $\forall p \in perm(\psi, X, C \cup E), \forall d \in D \cup E \quad p(c) = c$ we have $\forall d \in D \cup E \quad m(d) = u(d)$. But $u \in U_i$ implies $i > 0$ (otherwise $U_0 = \emptyset$) and thus $v \in [\![\varphi \cup \psi]\!]_{V[X/U_{i-1}]}$ and $w \in [\![\kappa]\!]_{V[X/\emptyset]}$ such that $u = v + w$. Therefore $\forall d \in D \cup E \quad m(d) = u(d) = w(d) \vee d \notin dom(w)$ which means $w \sim m$ and $m \in [\![\psi \bowtie \kappa]\!]_{V[X/U_{i+1}]}$. Therefore $[\![\psi]\!]_{V[X/U_i]} \subseteq [\![\psi \bowtie \kappa]\!]_{V[X/U_i]}$.

Since for all $i \in \mathbb{N}$ we have $[\![\psi]\!]_{V[X/U_i]} = [\![\psi \bowtie \kappa]\!]_{V[X/U_i]}$ we have $[\![\sigma_f(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu(X = \sigma_f(\varphi) \cup \psi)]\!]_V$.

□

A.3.10 Theorem 5

Theorem 5. *Let $\mu(X = \varphi \cup \psi)$ be a decomposed fixpoint with φ its constant part and ψ its recursive part, V an environment, C and a filter f with:*

1. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \quad C \subseteq dom(m)$
2. $\forall p \in perm(\psi, X, C) \quad \forall d \in FC(f) \quad p(d) = d$

we have: $[\![\sigma_f(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu(X = \sigma_f(\varphi) \cup \psi)]\!]_V$.

Proof. By the general Theorem 2 on filtered fixpoints we have that: $[\![\sigma_f(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu(X = \sigma_f(\varphi \cup \psi))]\!]_V = [\![\mu(X = \sigma_f(\varphi) \cup \sigma_f(\psi))]\!]_V$. Therefore we only need to show that $[\![\psi]\!]_{V[X/\{w\}]} = [\![\sigma_f(\psi)]\!]_{V[X/\{w\}]}$ for w a mapping such that $C \subseteq dom(w)$ and $eval(f, w) = \top$.

Clearly $[\![\sigma_f(\psi)]\!]_{V[X/\{w\}]} \subseteq [\![\psi]\!]_{V[X/\{w\}]}$, let us show that $[\![\psi]\!]_{V[X/\{w\}]} \subseteq [\![\sigma_f(\psi)]\!]_{V[X/\{w\}]}$.

Let $m \in [\![\psi]\!]_{V[X/\{w\}]}$, by lemma 6 there exists $p \in perm(\psi, X, C)$ such that $\forall c : p(c) = \bot \vee (m(c) \neq \bot \wedge m(c) = w(p(c)))$. Therefore $w(c) = p(c)$ for all $c \in FC(f)$ by condition 2 and thus $eval(f, m) = eval(f, w) = \top$ which proves $m \in [\![\sigma_f(\psi)]\!]_{V[X/\{w\}]}$. □

Theorem 6. *Given two decomposed fixpoints $\mu(X = \varphi \cup \psi)$ and $\mu(Y = \xi \cup \kappa)$ and $C_X, E_X, D_X, C_Y, E_Y, D_Y$ with:*

1. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \quad dom(m) = D_X \cup E_X = C_X$
2. $\forall m \in [\![\mu(Y = \kappa \cup \xi)]\!]_V \quad dom(m) = D_Y \cup E_Y = C_Y$
3. $D_Y \subseteq C_X,\ D_X \subseteq C_Y,\ E_X \cap C_Y = \emptyset,\ E_Y \cap C_X = \emptyset$
4. $\forall p \in perm(\psi, X, C_X), \forall c \in D_Y \quad p(c) = c$
5. $\forall p \in perm(\xi, Y, C_Y), \forall c \in D_X \quad p(c) = c$
6. $\forall c \in E_X \quad canAdd(\xi, Y, c)$
7. $\forall c \in E_Y \quad canAdd(\psi, X, c)$

Then we have: $[\![\mu(X = \varphi \cup \psi) \bowtie \mu(Y = \kappa \cup \xi)]\!]_V = [\![\mu(X = let\ (Y = X)\ in\ \varphi \bowtie \kappa \cup \psi \cup \xi)]\!]_V$ *and* $[\![\mu(X = \varphi \cup \psi) \bowtie \mu(Y = \kappa \cup \xi)]\!]_V = [\![\mu(X = \varphi \bowtie replace(\kappa, Y, X) \cup \psi \cup replace(\xi, Y, X))]\!]_V$

Proof. Let $U_0 = V_0 = \emptyset$, $U_{i+1} = [\![\varphi \cup \psi]\!]_{V[X/U_i]}$, $V_{i+1} = [\![\kappa \cup \xi]\!]_{V[X/V_i]}$ and $W_{i,j} = \{u + v \mid u \in U_i,\ v \in V_j,\ u \sim v\}$ we have $[\![\mu(X = \varphi \cup \psi) \bowtie \mu(X = \varphi \cup \psi)]\!]_V = \bigcup_{(i,j)\in\mathbb{N}^2} W_{i,j}$.

Let us show that $[\![\xi \cup \kappa \bowtie \varphi]\!]_{V[Y/W_{i,j}]} = W_{i,j+1}$ for $(i, j) \in \mathbb{N}*^2$. Clearly $[\![\kappa \bowtie \varphi]\!]_{V[Y/W_{i,j}]} = W_{1,1}$ is included in both side.

For $m \in [\![\xi \cup \kappa \bowtie \varphi]\!]_{V[Y/W_{i,j}]}$ we have either $m \in [\![\varphi \bowtie \kappa]\!]_{V[Y/W_{i,j}]}$ (which is already treated) or $m \in [\![\xi]\!]_{V[Y/W_{i,j}]}$. When $m \in [\![\xi]\!]_{V[Y/W_{i,j}]}$ we have $u \in U_i, v \in V_j$ such that $m \in [\![\xi]\!]_{V[Y/\{u+v\}]}$. By the requirement of the theorem (and in a very similar way to the proof of Theorem 3) we have $\forall c \in E_x \cup D_x \quad m(c) = u(c)$ and thus $m \sim u$. Let $E_X = \{c_1, \ldots, c_n\}$ we have $dom(m) = dom(v) + E_X$ and by repeated application of lemma 7 we have $[\![\pi_{c_1}(\ldots \pi_{c_n}(\xi))]\!]_{V[Y/\{u+v\}]} = \cdots = [\![\pi_{c_1}(\xi)]\!]_{V[Y/v+\{c_1 \to u(c_1)\}]} = [\![\xi]\!]_{V[Y/\{v\}]}$. Therefore if w is the mapping m restrained to the domain C_Y we have $w \in [\![\pi_{c_1}(\ldots \pi_{c_n}(\xi))]\!]_{V[Y/\{u+v\}]}$ implies $w \in [\![\xi]\!]_{V[Y/\{v\}]}$ with $v \in V_j$. Therefore $w \in V_{j+1}$ and $m = w + u$ is in $W_{i,j+1}$.

Let $m \in W_{i,j+1}$, we have $v_j \in V_j$, $v_{j+1} \in V_{j+1}$ and $u_i \in U_i$ such that $v_{j+1} \in [\![\kappa \cup \xi]\!]_{V[Y/\{v_j\}]}$. By requirements of the theorem $\forall c \in D_Y \quad v_{j+1}(c) = v_j(c)$ and thus $v_j \sim u_i$. And by repeated application of lemma 7 we have $(v_{j+1} + u_i) \in [\![\xi \cup \kappa]\!]_{V[Y/\{v_j+u_i\}]}$ and thus $(v_{j+1} + u_i) \in [\![\xi]\!]_{V[Y/W_{i,j}]}$.

In all the case we have $[\![\xi \cup \kappa \bowtie \varphi]\!]_{V[Y/W_{i,j}]} = W_{i,j+1}$. Since $sim(\xi \cup \kappa \bowtie \varphi, X) = 0$ we also have $[\![\xi \cup \kappa \bowtie \varphi]\!]_{V[Y/W_{i,j},X/W_{i,j}]}$ By complete symmetry between ξ and ψ we have $[\![\psi \cup \kappa \bowtie \varphi]\!]_{V[X/W_{i,j},Y/W_{i,j}]} = W_{i+1,j}$.

Let $Z_0 = \emptyset$ and $Z_{i+1} = [\![\varphi \bowtie \kappa \cup \xi \cup \psi]\!]_{V[X/Z_i,Y/Z_i]}$. Let us show that $Z_i = \bigcup_{\substack{(l,m)\in\mathbb{N}*^2 \\ l+m=i+1}} W_{l,m}$

For $i = 0$, $Z_0 = \emptyset$.

For $i = 1$, $Z_1 = [\![\varphi \bowtie \kappa \cup \xi \cup \psi]\!]_{V[X/\emptyset,Y/\emptyset]} = [\![\varphi \bowtie \kappa]\!]_{V[X/\emptyset,Y/\emptyset]} = W_{1,1}$.

Let $i \in \mathbb{N}*$ we have by induction
$Z_{i+1} = [\![\varphi \bowtie \kappa \cup \xi \cup \psi]\!]_{V[X/Z_i,Y/Z_i]} = \bigcup_{\substack{(w,l)\in\mathbb{N}*^2 \\ l+m=i+1}} [\![\varphi \bowtie \kappa \cup \xi \cup \psi]\!]_{V[X/W_{l,m},Y/W_{l,m}]}$.
But given $(l, m) \in \mathbb{N}*^2$ we have $[\![\varphi \bowtie \kappa \cup \xi \cup \psi]\!]_{V[X/W_{l,m},Y/W_{l,m}]} = [\![\varphi \bowtie \kappa \cup \psi]\!]_{V[X/W_{l,m},Y/W_{l,m}]} \cup [\![\varphi \bowtie \kappa \cup \xi]\!]_{V[X/W_{l,m},Y/W_{l,m}]} = W_{l+1,m} \cup W_{l,m+1}$. Therefore $Z_{i+1} = \bigcup_{\substack{(l,m)\in\mathbb{N}*^2 \\ l+m=i+1}} W_{l+1,m} \cup W_{l,m+1} = \bigcup_{\substack{(l,m)\in\mathbb{N}*^2 \\ l+m=i+2}} W_{l,m}$. Finally we have:

$$[\![\varphi \bowtie \kappa \cup \xi \cup \psi]\!]_V = \bigcup_{i\in\mathbb{N}} Z_i = \bigcup_{(l,m)\in\mathbb{N}^2} W_{l,m} = [\![\mu(X = \varphi \cup \psi) \bowtie \mu(X = \varphi \cup \psi)]\!]_V$$

□

A.3.11 Theorem 7

Theorem 7. *Given a decomposed fixpoint $\mu(X = \varphi \cup \psi)$ and a, b and C with $a \in C$ such that:*

1. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \; C \subseteq dom(m)$
2. $\forall m \in [\![\mu(X = \varphi \cup \psi)]\!]_V \; b \notin dom(m)$
3. $\forall p \in perm(\psi, X, C) \quad p(a) = a$

then

1. $[\![\beta_a^b(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu\big(X = \beta_a^b(\varphi) \cup \psi\big)]\!]_V$ *when* $canAdd(\psi, X, b)$
2. $[\![\pi_a(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu(X = \pi_a(\varphi) \cup \psi)]\!]_V$ *when* $canAdd(\psi, X, a)$
3. $[\![\rho_a^b(\mu(X = \varphi \cup \psi))]\!]_V = [\![\mu\big(X = \rho_a^b(\varphi) \cup \psi\big)]\!]_V$ *when when* $canAdd(\psi, X, b) \wedge canAdd(\psi, X, a)$

Proof. The theorem holds because:

1. values for b will propagate as constant but values for a also thanks to condition 3: let $U_0 = \emptyset$, $U_{i+1} = [\![\varphi \cup \psi]\!]_{V[X/U_i]}$, $V_0 = \emptyset$ and $V_{i+1} = [\![\beta_a^b(\varphi) \cup \psi]\!]_{V[X/V_i]}$ we will show that $V_i = \{m + \{b \to m(a)\} \mid m \in U_i\}$. Let us proceed by induction. This is true for $i = 0$. Let $i \in \mathbb{N}$ and let $m \in V_{i+1} = [\![\varphi \cup \psi]\!]_{V[X/V_i]}$ we have either that $m \in [\![\varphi]\!]_{V[X/\emptyset]}$ (which trivially gives us the result) or that $m \in [\![\psi]\!]_{V[X/V_i]}$. In the second case, we have $w \in V_i$ such that $m \in [\![\psi]\!]_{V[X/\{w\}]}$. But by induction we have

$w' \in U_i$ such that $w = w' + \{b \to w'(a)\}$. By lemma 6, $m(a) = w(a)$ and by lemma 7 and lemma 8 we have $m' \in [\![\psi]\!]_{V[X/\{w'\}]}$ with $m = m' + \{a \to m(a)\}$ which proves $V_{i+1} \subseteq \{m+\{b \to m(a)\} \mid m \in U_{i+1}\}$. Now let $m' \in U_{i+1}$ and let $m = m'+\{b \to m'(a)\}$. Either $m' \in [\![\varphi]\!]_{V[X/U_i]}$ (and the result is trivial) or $m' \in [\![\psi]\!]_{V[X/\{w'\}]}$ for some $w' \in U_i$. By induction we have $w \in V_i$ with $w = w' + \{b \to w'(a)\}$ and by lemma 7 we have that $w \in V_i$ implies $m \in V_{i+1}$

2. $canAdd(\psi, X, a)$ ensures that values for a are not relevant to the computation of ψ. More precisely let $U_{i+1} = [\![\varphi \cup \kappa]\!]_{V[X/U_i]}$ and $V_{i+1} = [\![\varphi \cup \pi_a(\kappa)]\!]_{V[X/V_i]}$ with $U_0 = V_0 = \emptyset$. We will show that $V_i = \{\{c \to v \in m \mid c \neq a\} \mid m \in U_i\}$. This is true for $i = 0$. And by lemma 7 this propagate by induction.

3. $canAdd(\psi, X, a) \wedge canAdd(\psi, X, b)$ ensures that values for a and b are not relevant to the computation of ψ: just like the other we can prove that V_i (i.e. the ith step of $\mu\big(X = \varphi \cup \rho_a^b(\kappa)\big)$) is the image of U_i (the i-th step of $\mu(X = \varphi \cup \kappa)$) through a renaming of the column a into b.

□

Lemma 12. *Given a term φ an a environment V and its abstraction Γ then $\forall m \in [\![\varphi]\!]_V \quad C(\varphi, \Gamma) \subseteq dom(m) \subseteq P(\varphi, \Gamma)$.*

Proof. We prove the result by induction on the size of φ.

- Let $m \in [\![\varphi_1 \cup \varphi_2]\!]_V$ we have $m \in \varphi_1$ or $m \in \varphi_2$ and thus $C(\varphi_1, \Gamma) \subseteq dom(m) \subseteq P(\varphi_1, \Gamma)$ or $C(\varphi_2, \Gamma) \subseteq dom(m) \subseteq P(\varphi_2, \Gamma)$ and thus in both cases $C(\varphi_1, \Gamma) \cap C(\varphi_1, \Gamma) \subseteq m \subseteq P(\varphi_1, \Gamma) \cup P(\varphi_1, \Gamma)$.

- Let $m \in [\![\varphi_1 \bowtie \varphi_2]\!]_V$ we have $m_1 \in [\![\varphi_1]\!]_V$ and $m_2 \in [\![\varphi_2]\!]_V$ and $dom(m) = dom(m_1) \cup dom(m_2)$ thus the result.

- For $\varphi \in \{\pi_a(\psi), \rho_a^b(\psi), \beta_a^b(\psi), \sigma_f(\psi)\}$ the result is obvious.

- For the special filters $\varphi = \sigma_{bnd(c)}(\psi)$ and $\varphi = \sigma_{\neg bnd(c)}(\psi)$ the result comes from the semantics of $eval(bnd(c), m)$ that evaluates to true only for mappings binding the column c.

- For $\varphi \in \{\varphi_1 \backslash\backslash \varphi_2, \varphi_1 \setminus \varphi_2, \varphi_1 - \varphi_2\}$ we have $m \in [\![\varphi]\!]_V$ implies $m \in dom(\varphi_1)$ and thus the result.

- For $\varphi = \theta(\psi, g : \mathcal{C} \to \mathcal{D})$ we have for all $m \in [\![\theta(\psi, g : \mathcal{C} \to \mathcal{D})]\!]_V$ a $m' \in [\![\psi]\!]_V$ with either $\mathcal{C} \subseteq m'$ and $dom(m) = dom(m') \cup \mathcal{D}$ or $dom(m) = dom(m')$. That is why $C(\varphi, \Gamma) = C(\psi, \Gamma) \cup \mathcal{D}$ when $C \subseteq C(\psi, \Gamma)$ and $C(\varphi, \Gamma) = C(\psi, \Gamma)$ otherwise (and similarly for P).

- For $\varphi = |c_1 \to v_1, \ldots, c_n \to v_n|$ the result is obvious.

- For $\varphi = \emptyset$ any set will be ok.

- For let $(X = \varphi)$ in ψ we have $[\![\text{let } (X = \varphi) \text{ in } \psi]\!]_V = [\![\psi]\!]_{V[X/[\![\varphi]\!]_V]}$. By induction we have $\forall m \in [\![\varphi]\!]_V \quad C(\varphi, \Gamma) \subseteq dom(m) \subseteq P(\varphi, \Gamma)$ and thus $\Gamma[C(\varphi, \Gamma), P(\varphi, \Gamma)]$ is an abstract environment of $V[X/[\![\varphi]\!]_V]$ and thus the result.

- For $\mu(X = \varphi)$ we have $U_0 = \emptyset$, $U_{i+1} = [\![\varphi]\!]_{V[X/U_i]}$ and $[\![\mu(X = \varphi)]\!]_V = \lim_{i\to\infty} U_i$. We show by induction on i that (W_i^C, W_i^P) is a valid type for U_i. For $i = 0$ it is clear. And if the induction hypothesis holds for $i \in \mathbb{N}$ then $\Gamma[X/V_i]$ is a valid abstraction for the environment $V[X/U_i]$ and by the general induction we have the result.

- Finally for variables the result is obvious by the definition of abstract environment.

$\square$

APPENDIX B

Details of our second benchmark

Table B.1: Number of solutions for each query and each graph

Query	Number of solutions for $n =$																	
	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
Q1	1035	2742	3960	9160	23307	45785	108597	257829	419302	964676	2077707	6104086	12851317	27285381	51323155	59953009	39624142	?
Q2	1330	2742	4316	9160	24969	45786	108597	257829	451550	964678	2077707	6104086	12851317	27285381	51323155	46784404	22998679	?
Q3	34307	84611	419140	1415274	7656802	34385488	160647218	?	?	?	?	?	?	?	?	?	?	?
Q4	4015	1167	2826	2544	5563	14106	31012	61695	134770	262135	650429	1297915	2753082	6692132	13334968	26476752	26006107	?
Q5	677	639	1631	2722	9114	16364	14987	109289	192653	365625	683670	952857	624038	3134311	12531942	12560576	19595646	?
Q6	1093	1136	727	1311	3334	3269	114987	64457	32264	64347	639292	642534	1927697	8025112	31038127	12835432	7774933	?
Q7	44	87	141	455	29	1686	3001	7805	16052	30445	75998	158807	12	783574	1566480	3140131	7830973	?
Q8	72	87	334	666	1634	3252	6504	16190	32622	65052	162602	325348	650274	783574	1566480	6499291	13054450	?
Q9	73	8	353	685	1569	3258	6320	16031	32154	64723	160677	321923	643960	3	2	6417508	16042045	?
Q10	565	62	116	57	39	91	391	126	17	162	70	17	59	117	233	14	160	106

Our prototype																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	20	30	37	75	167	315	750	1869	3231	7589	17468	51508	111490	248563	488896	T	T	T
$Q2$	24	38	49	94	237	415	970	7546	5488	9510	21556	64344	137279	300510	593119	T	T	T
$Q3$	445	1026	5072	17570	97371	454282	T	T	T	T	T	T	T	T	T	T	T	T
$Q4$	66	26	49	42	91	209	437	989	48561	8435	10527	21361	44554	111947	229056	464465	T	T
$Q5$	30	30	45	59	149	253	248	1609	2886	5702	11601	17736	16789	70998	241506	299637	T	T
$Q6$	516	515	515	522	559	570	2275	1618	2786	2892	14230	18266	46627	172920	T	400496	T	T
$Q7$	16	14	20	18	21	36	50	108	360	653	1835	12773	32955	23330	48907	102896	293638	T
$Q8$	51	48	59	62	81	105	178	482	841	2008	5395	10991	22993	41633	84018	272161	T	T
$Q9$	16	18	18	18	29	43	72	636	587	1029	2726	5920	12232	19133	40918	142755	402768	T
$Q10$	339	305	310	310	309	321	349	530	737	1106	1836	5878	6344	16989	38571	81378	221415	471908

Ramsdell Datalog																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	280	1376	5649	20907	138423	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q2$	277	1369	5503	21234	136629	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q3$	836	3250	14843	52035	323688	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q4$	150	397	1339	4655	31442	146118	T	T	T	T	T	T	T	T	T	T	T	T
$Q5$	192	684	2389	13600	72106	323422	T	T	T	T	T	T	T	T	T	T	T	T
$Q6$	901	3163	11094	33286	264580	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q7$	39	113	319	2503	403	50738	175707	T	T	T	T	T	143564	T	T	T	T	T
$Q8$	153	125	2619	11550	66633	307356	T	T	T	T	T	T	T	T	T	T	T	T
$Q9$	98	30	1593	5770	34679	157606	T	T	T	T	T	T	T	171043	340492	T	T	T
$Q10$	88	238	739	2474	17056	83787	363832	T	T	T	T	T	T	T	T	T	T	T

Postgres																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	23	46	138	445	2557	13266	52817	T	T	T	T	T	T	T	T	T	T	T
$Q2$	24	51	129	438	2551	12579	52743	309599	T	T	T	T	T	T	T	T	T	T
$Q3$	63	193	837	2731	15898	79327	403633	T	T	T	T	T	T	T	T	T	T	T
$Q4$	25	17	25	23	38	59	99	313	592	2021	4147	12763	36110	88753	373743	T	T	T
$Q5$	22	25	37	165	570	2589	10031	422135	T	T	T	T	T	T	T	T	T	T
$Q6$	24	47	130	446	2547	12152	54281	307631	T	T	T	T	T	T	T	T	T	T
$Q7$	22	18	32	159	553	2549	10008	76641	256535	T	T	T	T	T	T	T	T	T
$Q8$	15	47	139	661	3264	18359	72266	T	T	T	T	T	T	T	T	T	T	T
$Q9$	24	40	126	436	2541	13211	50576	308115	T	T	T	T	T	T	T	T	T	T
$Q10$	20	22	26	26	29	40	62	117	308	1264	3353	11291	29864	67052	340287	T	T	T

Table B.2: Time in milliseconds to evaluate queries in each query engine, T means timeout (thus > 600000 ms = 10 min)

DLV																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	38	157	698	3506	31378	163923	T	T	T	T	T	T	T	T	T	T	T	T
$Q2$	40	156	704	3568	31387	163717	T	T	T	T	T	T	T	T	T	T	T	T
$Q3$	242	707	3744	14769	98659	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q4$	36	22	44	49	107	239	586	1425	3294	7280	20020	46237	125058	T	T	T	T	T
$Q5$	24	35	97	1325	5949	37692	160384	T	T	T	T	T	T	T	T	T	T	T
$Q6$	36	152	684	3499	31347	163998	T	T	T	T	T	T	T	T	T	T	T	T
$Q7$	20	29	88	1161	4811	27714	114981	T	T	T	T	T	T	T	T	T	T	T
$Q8$	32	149	632	3682	24174	116852	T	T	T	T	T	T	T	T	T	T	T	T
$Q9$	33	126	548	2427	18793	85968	T	T	T	T	T	T	T	T	T	T	T	T
$Q10$	21	18	17	35	66	154	382	966	2431	5714	15888	38375	107760	285948	T	T	T	T

Vlog																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	113	1265	12685	184598	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q2$	108	1265	12664	184995	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q3$	184	1559	15168	204207	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q4$	64	59	78	101	161	345	950	2956	6689	17157	56588	137383	531602	T	T	T	T	T
$Q5$	63	121	420	27047	347451	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q6$	132	1410	14498	201749	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q7$	52	55	52	74	49	169	260	288	904	2084	9917	19587	749	105329	222389	516499	T	T
$Q8$	65	54	290	11139	100941	T	T	T	T	T	T	T	T	106399	221938	T	T	T
$Q9$	51	49	58	78	117	214	589	2465	2479	5311	12032	29011	107995	862	1770	T	T	T
$Q10$	52	53	54	56	66	78	108	239	687	1577	3293	8259	16715	51337	114233	252075	T	T