UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

THE PROBLEM OF INCOMPLETE DATA IN SPARQL

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS, MENCIÓN COMPUTACIÓN

DANIEL RICARDO HERNÁNDEZ HERNÁNDEZ

PROFESOR GUÍA:
CLAUDIO GUTIÉRREZ GALLARDO

MIEMBROS DE LA COMISIÓN:
PABLO BARCELÓ BAEZA
JUAN REUTTER DE LA MAZA
ANDREAS PIERIS

SANTIAGO DE CHILE
2020

Resumen

La información incompleta es uno de los mayores desafíos para la gestión de datos en la Web. Los datos en son incompletos en la Web por varias razones: porque hay datos que se desconocen, porque es ocultada por privacidad, o porque aún no ha sido agregada a los datos, porque aún no se ha integrado con otras fuentes de datos, etc. Una pregunta fundamental en la exploración de datos en la Web es cómo definir la semántica de las consultas sobre datos incompletos.

Esta tesis estudia el problema de información incompleta en SPARQL, el lenguaje de consulta definido por el W3C para los datos de la Web. Nos enfocamos en dos formas básicas de incompletitud en SPARQL: los blancos y variables con valores no asignados (*unbound values*). Su principal metodología es la aplicación de ideas y técnicas conocidas del modelo relacional para enfrentar estos problemas en SPARQL. Con este fin, contrastamos los nodos blancos y los valores no asignados con tres tipos de valores nulos usados en el modelo relacional: valores que se sabe que existen, pero se desconocen; valores que se sabe que no existen; y valores de los que no se sabe si existen.

Las contribuciones de esta tesis son las siguientes. Primero, mostramos que de acuerdo a la especificación de RDF y el uso de los nodos blancos en la base de conocimiento *Wikidata*, estos corresponden a los nulos que denotan valores que existen, pero se desconocen. Sin embargo, las consultas SPARQL entregan respuestas que son incorrectas de acuerdo a la semántica de los nodos blancos como valores que existen pero son desconocidos. Nosotros exploramos la factibilidad de que SPARQL retorne solo resultados correctos (*certain answers*) implementando métodos que son usados con el mismo fin en el modelo relacional. Para entender el impacto que este cambio puede tener en SPARQL, analizamos cómo afecta los ejemplos de consulta de usuarios de *Wikidata*.

En relación con la semántica de los valores no asignados, mostramos que ellos se comportan en algunos casos como valores que no existen y en otros como valores que no se sabe que existen. Proponemos una extensión del álgebra relacional inspirada en las tablas con tuplas posibles de Biskup (*maybe-tuples*). Esta álgebra es el resultado de escoger dos operadores para cada operador del álgebra relacional, donde uno entrega resultados correctos y el otro resultados posibles. Este resultado mejora nuestro entendimiento de SPARQL al hacer explícito que SPARQL es un resultado particular de escoger para cada operador uno de los dos operadores alternativos del álgebra que proponemos.

Uno de los problemas para analizar la información incompleta en SPARQL es la falta de un formalismo para comparar las diferentes propuestas. Nosotros abordamos este problema proponiendo un nuevo formalismo llamado *Nested Datalog*, que extiende Datalog para componer consultas. Extendiendo la codificación de SPARQL en Datalog por Angles y Gutiérrez (después de corregir algunos de sus problemas para consultas que implican información incompleta), mostramos cómo codificar SPARQL usando Nested Datalog, en particular para la cláusula EXISTS y la información incompleta.

En este nuevo formalismo, describimos las tres semánticas propuestas por la comunidad para la cláusula EXISTS (particularmente para los problemas de correlación y substitución), mostramos los supuestos hechos en cada una de ellas y entregamos un menú de estrategias para abordar los problemas de correlación y substitución.

Abstract

Incomplete information poses a major challenge for data management on the Web. Web data may be incomplete for a variety of reasons: the missing information may be unknown to those who created the dataset, it may have been suppressed for privacy reasons, it may not yet have been added to the dataset, there may be a gap left after integrating other datasets, and so forth. A fundamental question for exploiting data on the Web is then how to define the semantics for queries over incomplete datasets.

This thesis investigates incomplete information in SPARQL, the standard query language defined by the W3C for querying data in the Web. We focus the study in two basic forms of incompleteness in SPARQL: blank nodes and unbound variables. The core of the method is to apply ideas and techniques from the relational model to address these problems in SPARQL. To this end, we contrast the blank nodes and the unbound values with three types of null value used in the relational model: unknown values, inapplicable values, and no-information nulls.

As the results of this investigation, the contributions of the thesis are the following. First, we show, according to the RDF specification and the use of blank nodes in Wikidata, that blank nodes correspond to unknown values. However, SPARQL queries can return answers that are uncertain regarding the semantics of blank nodes as unknown values. We explore the feasibility that SPARQL returned certain answers by implementing the methods known from the relational model. To understand the impact this change produced in SPARQL, we analyze how this new semantics would affect the results of user queries in Wikidata.

Regarding the semantics of unbound values, we show that they behave in some cases as inapplicable values and as no-information nulls in other cases. We propose an extension for the relational algebra inspired in Biskup's maybe-tuples, which includes the core algebra of SPARQL. This algebra results from defining two operators for each standard relational algebra operator in order to support null values. One of these two operators returns certain answers and the other possible answers. This result improves our understanding of SPARQL since it makes explicit that SPARQL is a particular result from choosing for each operator one of the two alternative extended operators in the algebra we propose.

One of the current problems to analyze incomplete information in SPARQL is the lack of a unified formalism to compare the different proposals. We address this problem by proposing a new formalism called Nested Datalog that extends nr-Datalog$^\neg$ in order to allow the composition of queries. Extending the codification of SPARQL in nr-Datalog$^\neg$ by Angles and Gutierrez after fixing some of its issues for incomplete information, we show how to codify SPARQL using Nested Datalog, particularly the EXISTS clause and incomplete information.

In this new formalism, we describe the three alternative semantics proposed by the community to address the EXISTS clause (particularly the issue of correlation and substitution) and show what they mean. We study its underlying assumption and present a menu of possible approaches to address the current problems.

Acknowledgments

First, I want to thank my advisor Claudio Gutierrez for his careful guidance and his enthusiasm to talk about any question beyond short-term research. I can see in him, the truly scientific curiosity, critical thinking, and humanism in the widest sense. I would also like to acknowledge the people that contribute to my development as a researcher, especially to Aidan Hogan and Renzo Angles that were always open to discuss new ideas and advice. I am also grateful to Pablo Barcelo, Juan Reutter, and Andreas Pieris, who cautiously reviewed my thesis and helped me to improve it.

My doctoral studies were funded by CONICYT Doctorates scholarship CONICYT-PCHA-21140161, and the ANID – Millennium Science Initiative Program – Code ICN17_002.

Contents

1 Introduction **1**
- 1.1 Preliminary notions . . . 2
- 1.2 Problems, hypothesis and goals . . . 5
- 1.3 Summary of contributions of this thesis . . . 5
- 1.4 Structure of this thesis. . . . 8

I Preliminaries **9**

2 RDF and SPARQL **11**
- 2.1 The RDF data model . . . 11
- 2.2 The SPARQL query language . . . 14

II Incomplete data in SPARQL **19**

3 Blank nodes as unknown values **21**
- 3.1 Naive semantics of RDF . . . 24
- 3.2 RDF graphs as V-tables . . . 25
  - 3.2.1 The V-tables relational algebra . . . 29
  - 3.2.2 The relational algebra of SPARQL . . . 30
- 3.3 Approximating certain answers in SPARQL . . . 34
  - 3.3.1 Certain answers with null values . . . 34
  - 3.3.2 Under- and over-approximations . . . 35
  - 3.3.3 Approximating relational algebra queries . . . 41
  - 3.3.4 SPARQL rewriting strategies . . . 48
- 3.4 Evaluation . . . 51
  - 3.4.1 Evaluation Setting . . . 52
  - 3.4.2 TPC–H experiments . . . 52
  - 3.4.3 Wikidata survey . . . 56
- 3.5 Conclusions . . . 57

4 Unbound values as incomplete data **59**
- 4.1 A brief review of the landscape of null values . . . 60
- 4.2 Where do unbound values came from? . . . 61
- 4.3 Null values in SQL and SPARQL . . . 64
  - 4.3.1 The generalized algebra . . . 64
  - 4.3.2 Mapping SQL and SPARQL operators to the generalized algebra . . . 67

4.4 Conclusions 72

III On the semantics of EXISTS 73

5 The notion of substitution under incomplete data 75
5.1 The problem of substitution in SPARQL 76
5.1.1 An overview of the problems of substitution in SPARQL 77
5.1.2 Existing proposals for the semantics of EXISTS 83
5.1.3 How the existing proposals solve the substituion issues 85
5.2 Environment-binding as a substitution proposal 88
5.3 Free, bound, and range restricted variables 91
5.3.1 Substitution in relational calculus 91
5.3.2 Substitution in SQL 91
5.3.3 Substitution in SPARQL 94
5.4 Conclusions 96

6 Expressing SPARQL in Datalog 97
6.1 An overview of nr-Datalog$^\neg$ 98
6.2 Translation of the safe SPARQL-0 fragment 100
6.3 Fixing the translation of equality atoms in filter-conditions 104
6.4 Fixing the translation of negation in filter-conditions 106
6.5 Translation of the SPARQL-1 fragment 111
6.5.1 SELECTION queries in nr-Datalog$^\neg$ 111
6.5.2 MINUS queries in nr-Datalog$^\neg$ 111
6.5.3 VALUES queries to nr-Datalog$^\neg$ 112
6.5.4 BIND queries in nr-Datalog$^\neg$ 113
6.5.5 The translation of SPARQL-1 fragment to nr-Datalog$^\neg$ 113
6.6 Conclusion 114

7 Nested Datalog 117
7.1 Syntax and Semantics of Nested Datalog 118
7.2 Query atoms and external atoms 123
7.3 The expressive power of Nested nr-Datalog$^\neg$ 123
7.4 Conclusion 127

8 On the Nested Datalog semantics of the EXISTS clause 129
8.1 Two forms of substitution of SPARQL variables 129
8.1.1 Substitution of parameters 130
8.1.2 Substitution of goal variables 133
8.2 Nested Datalog with nulls 139
8.3 Conclusions 141

IV Conclusions 143

9 Conclusions and future work 145

Bibliography **149**

Appendix A The information lattice of mappings with marked nulls **155**

Appendix B SQL and relational algebra **161**
B.1 SQL and the generalized PRU algebra 162
B.2 SQL and the generalized selection 163
B.3 SQL and the generalized join 163
B.4 SQL and the generalized difference 164

Chapter 1

Introduction

In this thesis we study incomplete information in Web query languages. Incomplete information poses a major challenge for data management on the Web. Web data may be incomplete for a variety of reasons: the missing information may be unknown to those who created the dataset, it may have been suppressed for privacy reasons, it may not yet have been added to the dataset, there may be a gap left after integrating other datasets, and so forth. A fundamental question for exploiting data on the Web is then how to define the semantics for queries over incomplete datasets.

To take a literary example, we know that the poem "Beowulf" was written by *somebody*, but nobody knows who. If we fill a table with creations and creators, an option is to simply omit the authorship and use a dash to denote that we do not known who wrote "Beowulf" as is depicted by Table 1.1.

Creation	Creator
Beowulf	–
El Ingenioso hidalgo don Quijote de la Mancha	Miguel de Cervantes Saavedra

Table 1.1: Writings and their authors

Table 1.1 presents the usual form to represent incomplete data in relational databases. In order to see how this data could be represented in RDF [56]—the current standard data model for the Web—, consider how the standard direct mapping from tables to RDF (see [10]) transforms the data in the table above to the labeled digraph depicted in Figure 1.1.

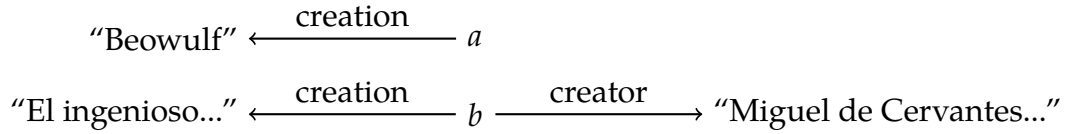


Figure 1.1: Direct mapping of Table 1.1 into an RDF representation.

The graph in Figure 1.1 introduces the nodes a, b, c, and d to denote the identity of each row in Table 1.1. The fields of each row are denoted by the labeled arcs connecting row

identities with field values. The empty field in Table 1.1 (the dash) is codified in the graph by the lack of an arc from node *a* to an author. As is suggested by Figure 1.1, RDF data can be represented as a labeled digraph (called *RDF graph*) where nodes are resources and labels in arcs identify relationships between them. This is the only formalization about RDF we need for this introduction. The standard formalization for RDF graphs will be provided later.

This translation for the empty field in Table 1.1 is motivated by the fact that in SQL null values imply no information about the value in the cell of a table. That is, a null value represents an unknown value or the nonexistence of the value. However, according to our background in the literary example that motivates this introduction, we known that "Beowulf" has an author, but the author is unknown. Thus, we need a particular type of null value, one that implies that the value exists but it is missing. These null values are called *unknown values*. In RDF there are resources called blank nodes that are sometimes used to denote unknown values (see Hogan et al. [42]). A relevant use case for blank nodes to denote unknown values is Wikidata, a collaborative knowledge base [69, 24]. In Figure 1.2 a blank node, denoted _:*b*, is used to refer the author of "Beowulf" in the same way that Wikidata does.

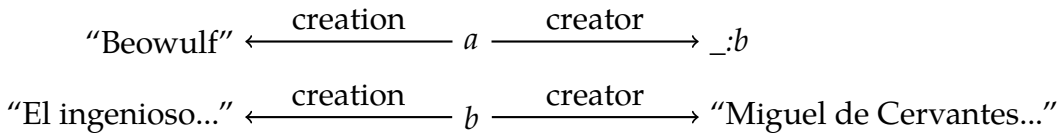


Figure 1.2: Mapping of Table 1.1 using a blank node to denote the author of "Beowulf."

So far, we have informally presented two example RDF graphs that codify incomplete data. One uses the lack of a value (Figure 1.1), and the other uses a blank node (Figure 1.2). If RDF is the standard data model of the Web, SPARQL [36] is the standard query language defined by the World Wide Web Consortium for querying data in the Web. In this thesis we address the problem of incomplete data in SPARQL.

1.1 Preliminary notions

We next present some examples that show how relevant incomplete data is in the design of SPARQL and the particularities of SPARQL compared to SQL. Since we have not yet introduced the syntax and semantics of SPARQL, we will describe queries in natural language.

Unbound values. Consider the query "Get all literary creations and its authors". In the graph of Figure 1.1, this query with the intended semantics of SPARQL does not return "Beowulf" because it has no known author. In order to include also "Belowulf" in the answers, SPARQL allows the retrieving of optional fields. Indeed, in SPARQL we can ask for "all literary creations and, if it is possible, their authors". If an author is not given for a literary creation, then the attribute is said to be unbound.

An *unbound* value indicates here that we do not know the author of "Beowulf" nor if the author exists. In RDF nothing can be inferred about non-present data. This is known as the

open-world assumption. Unbound values act thus as null values in SQL in the sense that they denote that we have no information about a required value. Recall that in SQL null values also denote that there is no information about the value. However, in SQL we can infer negative information from non-present data. For example, if the tuple (Alice, 22222) is not in the table of contacts we can infer that Alice has no phone with number 22222.

Angles and Gutierrez [8] showed that SPARQL, like SQL, can be seen as a language that returns tables with null values denoting the unbound values. However, SPARQL differs from SQL in the way that null values are operated. In this thesis we show that the difference between both languages regarding incomplete information resides in the way that they evaluate the join and the difference in the presence of null values. Let R and S be the tables defined as follows:

$$R = \left[\begin{array}{cc} A & B \\ \hline a & - \\ c & d \end{array}\right], \qquad S = \left[\begin{array}{cc} A & B \\ \hline a & b \\ c & d \end{array}\right].$$

The following table shows the results of the join and difference for these two query languages:

	SQL	SPARQL
$R \bowtie S$	$\left[\begin{array}{cc} A & B \\ \hline c & d \end{array}\right]$	$\left[\begin{array}{cc} A & B \\ \hline a & b \\ c & d \end{array}\right]$
$R - S$	$\left[\begin{array}{cc} A & B \\ \hline a & - \end{array}\right]$	$\left[\begin{array}{cc} A & B \end{array}\right]$

According to SQL and SPARQL the join of both tables includes the tuple (c, d). The difference is that in SPARQL tuple (a, b) is also included in the join, while in SQL it is not included. Intuitively, in SPARQL tuples $(a, -)$ and (a, b) are joined because the null value implies no knowledge about attribute A, so there exists the possibility that both tuples agree. On the contrary, in SQL both tuples are not merged because there is no certainty that both tuples agree. The same reasoning applies for the difference. In SPARQL the tuple $(a, -)$ is not dropped because there is a possible valuation of the null value where it is not dropped whereas in SQL it is dropped because that valuation exists.

This issue of unbound (and null) values is closely related to the open/closed character of the semantics. Unfortunately, there are SPARQL queries that behave contrary to the open-world semantics, and hence to the philosophy of the Web. SPARQL is a language that combines features that are suitable to query data in the Web (open world) with features that come from SQL, designed for databases that assume a closed-world. The question of whether SPARQL is suitable for querying the Web has been widely studied and discussed. Perez et al. [59] identify a notion, namely weak monotonicity, that captures queries that conform to the open-world semantics (i.e., weakly-monotone queries return answers that are still true when more data is considered). Perez et al. [59] defined syntactically a SPARQL fragment, called the *well-designed* queries, that are weakly-monotone. Arenas and Ugarte [12] proved that not all weakly-monotone query is well-designed, and defines a query language with the same expressive power than the fragment of the weakly-monotone queries.

Blank nodes. Now consider the query "who is not the author of 'Beowulf'." According to the standard semantics of SPARQL, the answer of this query over the graph depicted in Figure 1.2 is Miguel de Cervantes. However, according to the information provided by the graph there is no reason to reject the possibility of Cervantes being the author of "Beowulf" (i.e., that the blank node *_:b* references Cervantes). Of course we have good reasons to discard Cervantes as the author of "Beowulf", but these reasons are based on knowledge that is beyond the information provided by the graph. Thus, in this case SPARQL is giving an answer that is not certain in the light of the available facts. Note that this does not conform to the semantics of RDF.

Certain answers. A well-known interpretation for incomplete databases is the one that represents them as sets of complete databases called the models of the database (see [28]). In this formalism, μ is said to be a *certain answer* of a query Q over an incomplete database D if and only if μ is an answer of Q over every model of D. Similarly, μ is said to be a *certain negative answer* of Q over D if and only if μ is not an answer of Q over every model of D. There is a third case, called *maybe answer*, where μ is an answer of Q in some models of D but not an answer of Q in others. Also, μ is said to be a *possible answer* of Q over D if μ is a certain or a maybe answer of Q over D. We are interested in the certain answers of a query because they represent the facts that are true according to the database. The problem is that computing the certain answers of a query is intractable in most query languages. For instance, the data complexity of the standard query evaluation problem in relational algebra is AC^0; on the other hand, the analogous complexity with unknown values is coNP-hard [2]. This complexity limitation leads to finding approximations, i.e., algorithms that compute not all the certain answers but only part of them with less cost.

Relational algebra and Datalog. In this thesis we study SPARQL translating it to Datalog and to relational algebra. These query languages are well studied, have strong foundation in logic and provide techniques to study the problems we addressed in this thesis. Translations from SPARQL to Datalog have been studied with different motivations, e.g., Polleres [60], Schenk [66], Angles and Gutierrez [8], and Polleres and Wallner [61]. We focus in the translation by Angles and Gutierrez because the other translations do not generate Datalog, but expressions that are translatable to Datalog.

The SPARQL EXISTS clause. Since the version 1.1, SPARQL allows expressing the query "get all people x where there exists a department y whose responsible is x". This query can be rewritten as "get all people x where there exists an answers for *'get all departments y whose responsible is x'*." This rewritten query consists of two queries. The first, called the *outer query*, is "get all people x." The second, called the *inner query*, is "get all departments y whose responsible is x." The specification is not clear about how to evaluate the rewritten query [40, 57]. This ambiguity have led to differences in implementations. An essential issue here is the scope of variables. Is variable x referring the same person in the outer and in the inner queries? If is the case that the variable x refers to the same resource in both queries, then it is called a *correlated variable*.

Another problem of the semantics of the SPARQL EXISTS clause is how to correlate unbound variables. Should we accept as correlated a variable x bound in the outer query and unbound in the inner query? For instance, consider the query "get all people x and

their emails y when there exists an answer for the (sub) query *'gets all people z and optionally their emails y'*." Should we to accept a person x if there exists a person z whose email is missing? Since the unbound value can be considered a null value, this question touches the notion of incomplete data.

1.2 Problems, hypothesis and goals

The subject addressed in this thesis is the issue of the handling of incomplete data in SPARQL. The main hypothesis of this work is that the notions and techniques developed to study incomplete data in the relational model can be used to study the issues regarding incomplete data in SPARQL. The theory of relational databases is mature and has a strong foundation in first-order logic. This foundation can provide a clear understanding of the notions involved in the SPARQL design.

In order to address this hypothesis, we worked in the following goals regarding incomplete data in SPARQL:

- G1 Define the notions of blank nodes and unbound values in SPARQL in terms of null values in the literature of relational databases.
- G2 Provide a semantics for SPARQL based on well-known query languages (Datalog and relational algebra), in order to address the problem of incomplete data in SPARQL.

1.3 Summary of contributions of this thesis

The main contributions of this thesis are results obtained in the process to achieve the goals referred above. We observe that all these results are somehow touched upon by problems introduced by the way SPARQL deals with incompleteness. We focus our study in two forms of incompleteness, namely the blank nodes and the unbound values.

Goal G1. To address the first goal, we compare the semantics of blank nodes and unbound values with three types of null values studied in the literature of relational databases, namely *unknown values* (a value that we know exists but is missing), *inapplicable values* (a value that we know does not exist), and *no-information nulls* (we do not know if a value exists). According to the RDF specification [38], unknown values can be viewed as unknown values. However, as we aforementioned (see Section 1.1), SPARQL does not agree with the RDF semantics of blank nodes. Likewise, the semantics of unbound values results contradictory if we consider the RDF or the SPARQL design. For example, in SPARQL we can ask for "all books and optionally its writer". Since RDF follows the open world assumption, if we obtain no writer for a book, we cannot assume that that book has no writer. In other words, in that query the null value behaves as a no-information null. In SPARQL, the query above can be rewritten equivalently as the union of "all books and its writers" (i.e., including only books with writers), and "all books having no writers". In this second query an unbound value is generated for each book whose writer is not indicated in the database. However, according to the RDF semantics this second query must return no books having no writers, because the open world assumption. SPARQL is

thus following the closed world assumption in this second query and the unbound value can be seen as an inapplicable value.

We next summarize how we address the aforementioned problems regarding the semantics of blank nodes and unbound values.

To study the difference among SPARQL and RDF regarding blank nodes we choose a fragment of SPARQL that does not generates unbound values. We observe that in general the evaluation of a query Q in this fragment is not correct under the assumption that blank nodes denote unknown values, that is, may generate answers that are not certain. We thus address the following question: Given a query Q in this fragment, is there a query Q' in SPARQL that is a good approximation of the certain answers of Q? What does it mean "good"? What is the cost paid for the approximation? Is this problem relevant for the existing datasets? The study of these questions lead to the following contributions:

C1 We showed that by translating techniques of the approximation of certain answers used in relational model to SPARQL we can construct an approximation of the certain answers of SPARQL queries in the fragment studied whose data complexity remains in the complexity of the SPARQL fragment studied. Since this fragment corresponds to relational algebra, the data complexity of the evaluation problem of the translated query is A^0.

C2 We show that although generally querying for certain answers incur in a significant cost, in this case query answering is still feasible.

C3 Regarding the question of whether or not querying for certain answers is important, we studied a paradigmatic dataset, Wikidata, that uses blank nodes as unknown values. We analyzed 446 Wikidata queries and found that only a 0.6% of these queries return uncertain answers in practice.

We avoided SPARQL queries with unbound values when studying certain answers because the approximation of certain answers have not been studied in the relational algebra extended with the operators outer union or outer join. We focus thus in the problem of what is the relation among SPARQL and relational algebra extended with unknown nulls. The approximation we used followed the idea behind the Biskup [15] algebra of tables with *maybe tuples*. The algebra he proposes combines in each operation the tuples that represent sure facts with the tuples that represent possible facts. This idea is also used by Guagliardo and Libkin [33] to approximate certain answers in SQL. We adapt their techniques to SPARQL.

We then addressed the problem of what is the relation among unbound values and the null values known from the relational algebra literature. The contributions regarding this question are the following:

C4 We propose an algebra for tables with marked nulls based on the Biskup [15] algebra, called the *generalized relational algebra*. The generalized algebra defines two variants for each operator. The idea behind these variants is that one returns answers that are valid in all models of the tables involved in the operation, and the other returns answers that are valid in some models. That is, one variant returns certain answers and the other variant returns possible answers.

C5 We show that SPARQL and the relational algebra corresponding to SQL are contained in the generalized relational algebra. In fact, they take one variant for each of the relational algebra operators, and they differ only in the join and the difference. SPARQL returns possible answers for the join operator and returns certain answers for the difference. On the contrary, SQL returns certain answers for the join and possible answers for the difference.

The techniques used to address the goals aforementioned are founded in the notion of approximation. The question that arises is how to compare different approximations. We make the following contribution in this regard:

C6 We define the notions of under- and over-approximation among evaluation procedures in SPARQL. These notions define two lattices whose respective top and bottom are the most and least informative evaluation procedures. We can thus take advantage of the properties of lattices (e.g., combine evaluation procedures to get a more accurate one) and define a criterion to determine whether an evaluation procedure is correct. An evaluation procedure is said to be correct if it is an element in one of these lattices.

Goal G2. To address the second goal, we review the translations from the core of SPARQL to relational query languages. In particular we focus in the translation by Angles and Gutierrez [8] from SPARQL to Datalog. This translations has problems in the way that SPARQL deals with incomplete data, particularly with the 3-valued logic used in condition formulas to address unbound values. In fact, the main problem of their translation is that some answers are lost because the unknown truth value is not considered.

C7 We find some problems in the translation by Angles and Gutierrez [8] from SPARQL to Datalog and we propose a solution for them.

C8 We extend the translation by Angles and Gutierrez [8] to include the operators MINUS, BIND and VALUES, that are introduced in the version 1.1 of SPARQL [36].

A relevant SPARQL feature is the nesting of queries inside EXISTS clauses, that we study to address the goal G2. A nested query is evaluated as a Boolean condition over a mapping. The condition is true if the nested query has an answer, and false otherwise. As in other Boolean conditions, the variables in the mapping are substituted in the nested query before the evaluation. However, as we show in this thesis, substitution (particularly in the presence of incomplete information) is a notion that have to be reviewed. This study is relevant not only for SPARQL, but also for other query languages having outer/inner queries, such as SQL and G-core. Regarding to this topic, we made the following contributions:

C9 We report inconsistencies and ambiguities in the semantics of the SPARQL EXISTS clause that motivated a discussion by the community and lead to three proposals to fix the specification, namely *shallow-binding*, *deep-binding*, and *environment-binding*.

C10 We describe the different ways that engines today evaluate the EXISTS clause.

C11 We show how the three proposals described above differ, and that this difference occurs due different ways to deal with unbound values and a different definition of variable scope.

C12 We propose a new formalism, namely, Nested Datalog, that extends Datalog allowing the nesting of queries, and thus provide a simple translation for the SPARQL EXISTS clause.

C13 We show that Nested Datalog has the same expressive power than Datalog.

C14 We show an explicit translation from the SPARQL EXISTS clause to Nested Datalog. Currently it produces queries whose size is exponential in the size of the variables that are substituted in the nested query.

C15 We propose an extension to Nested Datalog (Nested Datalog with null values) to consider unbound values as a special symbol, instead of a constant as is done in the translation by Angles and Gutierrez [8]. We show that the translation of SPARQL to Nested Datalog with Nulls does no produce queries whose size is exponential on the size of the variables that are substituted in the nested query.

1.4 Structure of this thesis.

We organize this thesis in the following parts:

- In Part I we present the preliminaries of this thesis. We include the definitions of the languages RDF and SPARQL.
- In Part II we address the problem of incomplete information in SPARQL (goal G1). This part is divided in two chapters. In Chapter 3 we present a semantics for blank nodes based on the marked nulls that denote values that we know exists but are missing. We thus use the techniques known from the relational model to approximate certain answers in SPARQL. In Chapter 4 we address the problem of formalizing unbound values in terms of null values. We explain why there is no consistent semantics for unbound values that answer this question. Also, we describe the relation among the algebra of Biskup [15], the SPARQL algebra, and SQL.
- In Part III we address the problem of providing a logical foundation for incomplete information in SPARQL. In particular, we focus in the problem of giving a semantics to the EXISTS clause. We divide this part in four chapters. In Chapter 5 we describe the issues arising with specification of the EXISTS clause. In Chapter 6 we show how to give a semantics based on Datalog to SPARQL (excluding the EXISTS clause). We base this chapter in the one proposed by Angles and Gutierrez [8]. We also report the problems with this translation and provide a fixed translation. In Chapter 7 we propose a new formalism called Nested Datalog that extends Datalog in order to allow the composition of queries. In Chapter 8 we show how this formalism can be used to formalize SPARQL, particularly the EXISTS clause.
- In part IV we present the conclusions of the thesis.

Part I

Preliminaries

Chapter 2

RDF and SPARQL

The *Resource Description Framework* (RDF) [56, 37, 38] was proposed by the W3C as the main data model to share data on the Web. The initial idea was to publish RDF documents similarly as previously was done with HTML. RDF documents are essentially a set of claims about resources. Resources are named with Web identifiers (URLs). Data about resources can be retrieved with the Web protocol (HTTP) by sending a request on the resource identifier. With this design, the Web infrastructure for documents serves also to build the Web of Data.

After the standardization of the RDF data model, several query languages were proposed [13] for the RDF data model, including RQL [17], SquishQL [54], NautiLOD [25], and SPARQL [62, 36], among others. SPARQL became a W3C Recommendation in 2008, and it is still considered the standard language for querying RDF data.

This chapter reviews the RDF data model (Section 2.1) and the SPARQL query language (Section 2.2). To simplify the study of SPARQL, we use the algebraic formalization by Perez et al. [59]. Their formalization has some differences with the standard SPARQL algebra, but as Angles and Gutierrez [6], and Kaminski et al. [44] showed, the differences do not imply a different expressive power.

2.1 The RDF data model

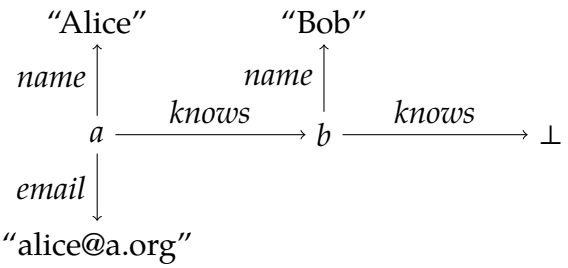
RDF is based on a special type of labeled digraph called RDF graph. The labels of nodes and edges are elements of the union of three pairwise disjoint sets, namely **I**, **B** and **L**, called identifiers, blank nodes and literals, respectively. We will keep this notation along the whole document. The structure of RDF graphs is stated by the following definition.

Definition 2.1 (RDF Syntax)**.** *An RDF graph G is a finite set of triples (s, p, o) in $(\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ where s is called the subject, p is called the predicate, and o is called the object.*

Notation. We denote identifiers as words with one or more letters (e.g., *a*, *b*, *c*, *name*, *email*, and *knows*). We denote literals with quotation marks (e.g., "Alice" and "Bob"), except for literals codifying numbers where we omit the quotation marks (e.g., 1, 2, and 3). We

denote blank nodes using the symbol $\perp$ with a numeric subscript when there is more than one blank node (e.g., $\perp_1$, $\perp_2$, and $\perp_3$).

Example 2.1. *The following figure depicts an RDF graph:*



Each triple (s, p, o) of the RDF graph is depicted as an arc with label p from a node s to a node o. In this graph, elements "Alice", "Bob" *and "alice@a.org" are literals; elements a, b, name, knows, and email are identifiers; and $\perp$ is a blank node.*

Since RDF graphs are sets of RDF triples, we use the set operators $\cup$, $\cap$, and $\setminus$ between RDF graphs, with their standard meaning for sets.

Definition 2.2 (RDF vocabulary)**.** *An RDF vocabulary (or simply a vocabulary) is a finite subset of $\mathbf{I} \cup \mathbf{L}$. Given an RDF graph G, the vocabulary of G, denoted* $\mathrm{voc}(G)$*, is the set of all identifiers and literals occurring in G.*

Example 2.2. *If G is the graph of Example 2.1 then* $\mathrm{voc}(G)$ *is composed of all labels of nodes and arcs occurring in G, except the blank node $\perp$.*

We next present the semantics of RDF graphs as defined by Hogan et al. [42] based in the definition by Hayes [37]. We will omit datatype interpretations, and the use of RDF vocabularies with predefined semantics (e.g., RDFS [16] and OWL [55]) as they are not directly concerned with the subject of this thesis.

Definition 2.3 (Semantics of ground triples [42])**.** *An interpretation $\mathcal{A}$ over a vocabulary V is a tuple (R, P, E, I) such that R is a non-empty set, called the domain, or the universe or the resources of $\mathcal{A}$; P is a set (not necessarily disjoint from or a subset of R) called the properties of $\mathcal{A}$; $E : P \to 2^{R \times R}$ is a mapping that assigns an extension, denoted $p^{\mathcal{A}}$, to each property $p \in P$; and $I : V \to R \cup P$ is the interpretation mapping that assigns a resource or a property to each element of V such that I is the identity for literals.*

Given a vocabulary V, an interpretation $\mathcal{A} = (R, P, E, I)$ over V, and a triple $(s, p, o) \in V^3$, we say that $\mathcal{A} \models (s, p, o)$ if and only if $I(p) \in P$ and $(I(s), I(o)) \in E(I(p))$.

Definition 2.3 provides a semantics for RDF triples without blank nodes. In order to define the semantics of RDF graphs with blank nodes we have to consider a function that maps blank nodes to elements in $\mathbf{I} \cup \mathbf{L}$.

Definition 2.4 (Semantics of an RDF graph)**.** *Let G be an RDF graph, V be a vocabulary such that $V \supseteq \mathrm{voc}(G)$, $\mathcal{A} = (R, P, E, I)$ be an interpretation over V, $v : \mathbf{B} \to R$ be a function, and I_v denote an extension of I that includes $\mathbf{B}$ as part of its domain such that $I_v(b) = v(b)$ for $b \in \mathbf{B}$ and $I_v(x) = I(x)$ for $x \in \mathbf{I} \cup \mathbf{L}$. We say that $\mathcal{A}$ is a model of G, denoted $\mathcal{A} \models G$, if there exists a function v such that for each $(s, p, o) \in G$, it holds that $I(p) \in P$ and $(I_v(s), I_v(o)) \in E(I(p))$.*

Example 2.3. *Let G be the following RDF graph:*

$$a \xrightarrow{\quad knows \quad} b \xrightarrow{\quad knows \quad} \bot$$

Then, interpretations $\mathcal{A}_1$ to $\mathcal{A}_4$ below are models of G.

$$\begin{array}{lllll}
\mathcal{A}_1 = (\ \{1,2,3\}, & \{p\}, & \{p \mapsto \{(1,2),(2,3)\}\}, & \{a \mapsto 1, b \mapsto 2, \bot \mapsto 3, knows \mapsto p\}\ ) \\
\mathcal{A}_2 = (\ \{1\}, & \{p\}, & \{p \mapsto \{(1,1)\}\}, & \{a \mapsto 1, b \mapsto 1, \bot \mapsto 1, knows \mapsto p\}\ ) \\
\mathcal{A}_3 = (\ \{1,2,3\}, & \{p\}, & \{p \mapsto \{(1,2),(2,3),(3,1)\}\}, & \{a \mapsto 1, b \mapsto 2, \bot \mapsto 3, knows \mapsto p\}\ ) \\
\mathcal{A}_4 = (\ \{1,2\}, & \{p,q\}, & \{p \mapsto \{(1,1)\}, q \mapsto \{(1,2)\}\}, & \{a \mapsto 1, b \mapsto 1, \bot \mapsto 1, knows \mapsto p\}\ )
\end{array}$$

The semantics of an RDF graph G is defined in terms of the interpretations of G following a model-theoretic semantics. In this thesis we assume familiarity with the model-theoretic and the proof-theoretic semantics of relational databases. An introduction of both approaches is given by Reiter [65]. The RDF model has some differences with the usual model-theoretic and proof-theoretic semantics given to relational databases:

1. Interpretations of RDF graphs allow predicates to occur as elements in relations. The main reason of this design is that RDF aims to describe not only resources (i.e., the elements of the relations), but also notions as predicates and classes used to describe the resources.

2. Interpretations of an RDF graph are not bounded to a specific "relational" vocabulary as the models of a relation do. In fact, the interpretations of an RDF graph G may contain relations that are not referred by the predicates in G. For instance, in Example 2.3, predicate q of interpretation $\mathcal{A}_4$ does not represent a predicate in the graph G. The RDF model is designed for the Web. In this context we cannot assume that people will get a consensus over a fixed set of predicates to be used to describe the world.

The notions of entailment and equivalence between RDF graphs are defined as usual.

Definition 2.5 (Simple entailment)**.** *An RDF graph G_1 entails an RDF graph G_2, denoted $G_1 \models G_2$, if and only if every interpretation over the vocabulary of $G_1 \cup G_2$ which satisfies G_1 also satisfies G_2. We say that two RDF graphs G_1 and G_2 are (logically) equivalent, denoted $G_1 \equiv G_2$, if and only if $G_1 \models G_2$ and $G_2 \models G_1$.*

A notion that is closely related with entailment and the equivalence of RDF graphs is the notion of map between graphs.

Definition 2.6 (Map between RDF graphs)**.** *A map $h : \mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \to \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$ is a function preserving elements in $\mathbf{I} \cup \mathbf{L}$, i.e., $h(x) = x$ if $x \in \mathbf{I} \cup \mathbf{L}$. Given a graph G and a map h, $h(G)$ denotes the RDF graph $\{(h(s), h(p), h(o)) \mid (s,p,o) \in G\}$. We overloaded the meaning of map to speak of a map $h : G_1 \to G_2$ if and only if G_1 and G_2 are RDF graphs, h is a map, and $h(G_1) \subseteq G_2$.*

Two RDF graphs G_1 and G_2 are said homomorphic equivalent if and only if there are maps $h_1 : G_1 \to G_2$ and $h_2 : G_2 \to G_1$. They are called isomorphic if $h_1(G_1) = G_2$ and $h_2(G_2) = G_1$ (or, as RDF graphs are finite, both maps h_1, h_2 are one-to-one).

So far, we have reviewed the standard notions of entailment and map among RDF graphs. It is well-known that these notions are equivalent (see [19, 38, 35]). This equivalence is formalized by the following theorem:

Theorem 2.1 ([19, 38, 35])**.** *Let G_1 and G_2 be two RDF graphs. Then:*

- *$G_1 \models G_2$ if and only if there is a map $h : G_2 \to G_1$.*
- *G_1 and G_2 are (logically) equivalent if and only if G_1 and G_2 are homomorphic equivalent.*

In this thesis we assumed familiarity with the Reiter [63] *proof-theoretic* semantics of the relational model extended with null values, whose theory codifies three assumptions over a relational database, namely the *closed-domain*, the *unique-name*, and the *closed-world* assumptions. We next discuss the differences among the relational model and the RDF model regarding these assumptions.

1. *No Unique Name Assumption:* As is exemplified by the model $\mathcal{A}_2$ of Example 2.3, different identifiers and blank nodes may refer to the same resource (in this case a, b, and $\perp$ refer to resource 1). The lack of the unique name assumption is motivated by the distributed nature of the Web. Data publishers in different parts of the world may use different identifiers to refer to the same resource.

2. *No Closed World Assumption:* Let G be an RDF graph and (s, p, o) be a triple without blank nodes that is not in G. Then, there exists an interpretation $\mathcal{A}$ of G such that $\mathcal{A} \models (s, p, o)$. This follows because under the semantics provided by Definition 2.4, all RDF graphs are satisfiable. Hence, no negative information can be inferred from RDF graphs, and thus RDF graphs do not follow the closed world assumption in the sense of Reiter [64].

2.2 The SPARQL query language

SPARQL is a query language designed to query RDF graphs. In this section we present the core fragment of the SPARQL language for consideration in this thesis. This fragment is essentially the fragment studied by Perez et al. [59]. We do not include the clauses FROM and GRAPH as they do because those features are not related to the focus of this thesis, but we add the clauses VALUES, BIND and EXISTS.

In order to define the syntax and semantics of SPARQL, we need to introduce some notions. Along with the sets **I**, **B**, and **L**, that are used to define RDF triples, we assume a countable infinite set **V**, called the set of variables, and a set **F**, called the set of functions, that consists in functions of the form $f : (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \{\varnothing\})^n \to \mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \{\varnothing\}$, where sets **V**, **F**, **I**, **B**, and **L** are pairwise disjoint, and $\varnothing$, called the unbound value, is an element that is not in the set $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. The prefix "?" is used to denote variables (e.g., ?x). A *SPARQL mapping* (or simply a *mapping* when no confusion arises) is a partial function $\mu : \mathbf{V} \to \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. We denote by $\mathrm{dom}(\mu)$ the domain of μ, that is, the subset of **V** where the function is defined. Two mappings μ_1 and μ_2 are said to be compatible, denoted $\mu_1 \sim \mu_2$ if and only if for every variable ?x in $\mathrm{dom}(\mu_1) \cap \mathrm{dom}(mu_2)$ it holds $\mu_1(?x) = \mu_2(?x)$. Given two compatible mappings μ_1 and μ_2, the join of μ_1 and μ_2, denoted $\mu_1 \smile \mu_2$, is the mapping with domain $\mathrm{dom}(\mu_1) \cup \mathrm{dom}(\mu_2)$ that is compatible with μ_1 and μ_2. Given a mapping μ and a finite set

$\mathcal{X}$ of variables, we write $\mu_{|\mathcal{X}}$ to denote the mapping μ whose domain is restricted to the set $\mathcal{X}$, that is, $\mu_{|\mathcal{X}} \sim \mu$ and $\text{dom}(\mu_{|\mathcal{X}}) = \text{dom}(\mu) \cap \mathcal{X}$.

Throughout this thesis we sometimes use the translation of sets of mappings to relations used by Cyganiak [21] and Polleres [60], among others. This consists in viewing mappings as tuples under the named perspective. However, as mappings in a set Ω of mappings may have different domains, we need to choose one for the relation associated to Ω. The set of variables $\mathcal{X}$ of the chosen domain must be big enough to contain the domains of all mappings in Ω. Then, for each variable $?x \in \mathcal{X}$, we extend ("fill" in) all mappings μ where $?x \notin \text{dom}(\mu)$ defining $\mu(?x) = \varnothing$. The result of this procedure is a relation in the domain $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \{\varnothing\}$. This procedure is formalized as follows:

Definition 2.7 (Filled mapping)**.** *Given a SPARQL mapping μ with* $\text{dom}(\mu) = \mathcal{X}$ *and a set of variables $\mathcal{Y}$ such that $\mathcal{X} \subseteq \mathcal{Y}$, the filling of μ over $\mathcal{Y}$, denoted* $\text{fill}(\mu, \mathcal{Y})$*, is the mapping such that* $\text{dom}(\text{fill}(\mu, \mathcal{Y})) = \mathcal{Y}$ *and*

$$\text{fill}(\mu, \mathcal{Y})(?x) = \begin{cases} \mu(?x) & \text{if } ?x \in \text{dom}(\mu), \\ \varnothing & \text{otherwise.} \end{cases}$$

Now we are ready to define the SPARQL syntax.

Definition 2.8 (SPARQL syntax)**.** *The set of SPARQL queries (or simply queries) is defined recursively as follows:*

- *An element of* $(\mathbf{I} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V})$ *is a triple pattern. A set of triple patterns is a query—called a basic graph pattern.*
- *If Q_1, Q_2 are queries, then:*
  - $(Q_1$ UNION $Q_2)$ *is a query—called a UNION query.*
  - $(Q_1$ AND $Q_2)$ *is a query—called an AND query.*
  - $(Q_1$ OPT $Q_2)$ *is a query—called an OPT query.*
  - $(Q_1$ MINUS $Q_2)$ *is a query—called a MINUS query.*
- *If Q is a query and $\mathcal{X} \subset \mathbf{V}$ is a finite set of variables, then* (SELECT $\mathcal{X}$ WHERE Q) *is a query—called a SELECT query.*
- *A filter-condition is an expression defined recursively as follows:*
  - *An equality $t_1 = t_2$, where t_1 and t_2 are elements of $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V}$, is a filter-condition.*
  - *If t is an element of $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V}$ then* isBlank(t) *is a filter-condition.*
  - *If* $?x$ *is a variable then* bound$(?x)$ *is a filter-condition.*
  - *If Q is a query then* EXISTS(Q) *is a filter-condition.*
  - *A Boolean combination of filter-conditions (with operators $\wedge$, $\vee$, and $\neg$) is a filter-condition.*

  If Q is a query and φ is a SPARQL buit-in condition (see below), then $(Q$ FILTER $\varphi)$ *is a query—called a FILTER query.*

- *If* $\mathcal{X} \subset \mathbf{V}$ *is a finite set of variables, and* Ω *is a set of SPARQL mappings such that* $\text{dom}(\mu) \subseteq \mathcal{X}$ *for each* $\mu \in \Omega$*, then* (VALUES $\mathcal{X}$ Ω) *is a query—called a VALUES query.*

- *If* Q *is a query,* $f : (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \{\varnothing\})^n \to (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \{\varnothing\})$ *is a function in* $\mathbf{F}$*, and* $?y, ?x_1, \ldots, ?x_n$ *are variables such that* $?y$ *does not occur in* Q *nor in* $\{?x_1, \ldots, ?x_n\}$*, then* (Q BIND $f(?x_1, \ldots, ?x_n)$ AS $?y$) *is a query—called a BIND query.*

We call SPARQL-0 to the fragment of SPARQL that consists of queries composed by the operators AND, UNION, FILTER, and OPTIONAL, and we call SPARQL-1 to the fragment that consists of queries composed by any of the operators described in this definition.[1]

Historically, SPARQL *queries* and *graph patterns* were two different concepts. Polleres [60] defines a SPARQL query as a quadruple (V, P, DS, SM) where V is the result form, P is a graph pattern, DS is a dataset, and SM is a set of solution modifiers. In this thesis we ignore datasets and solution modifiers, so according to Polleres notation, a query is simple a pair (V, P). Furthermore, in this thesis we restrict result forms to the SELECT clause. Thus, we only study SELECT queries.

Note: Since in SPARQL 1.1 graph patterns admit SELECT queries as graph patterns, and in the fragment we are studying we do not considering modifiers, there is no need to consider SPARQL queries and graph patterns as different concepts. Thus, throughout this thesis we will use both terms with the same meaning.

Note on Multisets. Before presenting the semantics of SPARQL we have to discuss *set* and *multiset semantics*. SPARQL specification uses multiset semantics by default (like SQL). Recall that a multiset is a modification of the concept of set, that unlike sets, allows repeated elements. In SPARQL 1.0 a SELECT query (SELECT $\mathcal{X}$ WHERE P), where P is a graph pattern without another SELECT query inside, returns by default multisets. To get sets one has to use (as in SQL) the modifier DISTINCT: (SELECT DISTINCT $\mathcal{X}$ WHERE P). In SPARQL 1.1 the situation more complex. As SPARQL 1.1 admits SELECT queries as graph patterns, queries can be evaluated as a combination of both semantics. This brings many complexities to the theory (see the work of Angles and Gutierrez [9]). In order to focus on incompleteness, that already in the case of set semantics it has enough interest, in this thesis we will restrict to the *set semantics* of SPARQL as is done by Perez et al. [59].

According to Perez et al. [59], the semantics of SPARQL is defined using operations over sets of mappings. These operators are presented in the following definition:

Definition 2.9 (Algebra of mappings [59])**.** *Let* Ω_1 *and* Ω_2 *be two sets of mappings. Then, the operators* $\bowtie$, $\cup$, $-$, *and* $\mathbin{⟕}$ *are defined over sets of mappings as follows:*

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\},\\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\
\Omega_1 - \Omega_2 &= \{\mu_1 \mid \mu_1 \in \Omega_1, \text{ and there does not exist } \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\},\\
\Omega_1 \mathbin{⟕} \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2).
\end{aligned}$$

[1] Intuitively, the SPARQL-0 fragment corresponds to the intersection of set of the operators in this definition with the version 1.0 of the standard [62], whereas the SPARQL-1 with the version 1.1 [36].

Given a query Q we write $\text{var}(Q)$ to denote the set of variables occurring in Q. We use this notation also for filter-conditions, i.e., $\text{var}(\varphi)$ are the variables occurring in the filter-condition φ.

Intuitively, a filter-condition φ drops all mappings μ where $\mu(\varphi)$ is not true. The following definition defines the truth value of $\mu(\varphi)$ for all filter-conditions without the EXISTS clause. The semantics of the EXISTS clause is highly complex, thus it requires its own chapter. We will study it in Chapter 5.

Definition 2.10 (Semantics of SPARQL filter-conditions)**.** *Let μ be a finite mapping from variables to elements in $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$, φ be a SPARQL filter-condition without occurrences of EXISTS clauses, and t_1, t_2 be elements in $\mathbf{V} \cup \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. Let $\nu : \mathbf{V} \cup \mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \to \mathbf{V} \cup \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$ be the function defined as*

$$\nu(t) = \begin{cases} \mu(t) & \text{if } t \in \text{dom}(\mu), \\ \varnothing & \text{if } t \in \mathbf{V} \setminus \text{dom}(\mu), \\ t & \text{if } t \notin \mathbf{V}. \end{cases}$$

The truth value of $\mu(\varphi)$ is defined recursively as follows:

- *If φ is an equality $t_1 = t_2$ then:*
  1. *$\mu(\varphi)$ is error if $\nu(t_1) = \varnothing$ or $\nu(t_1) = \varnothing$.*
  2. *$\mu(\varphi)$ is true if $\nu(t_1) \neq \varnothing$, $\nu(t_1) \neq \varnothing$, and $\nu(t_1) = \nu(t_2)$.*
  3. *$\mu(\varphi)$ is false if $\nu(t_1) \neq \varnothing$, $\nu(t_1) \neq \varnothing$, and $\nu(t_1) \neq \nu(t_2)$.*
- *If φ is an expression of the form* $\text{bound}(?x)$ *then $\mu(\varphi)$ is true if* $?x \in \text{dom}(\mu)$*. Otherwise, $\mu(\varphi)$ is false.*
- *If φ is an expression of the form* $\text{isBlank}(t_1)$ *then:*
  1. *$\mu(\varphi)$ is error if $\nu(t) = \varnothing$.*
  2. *$\mu(\varphi)$ is true if $\nu(t) \neq \varnothing$ and $\nu(t) \in \mathbf{B}$.*
  3. *$\mu(\varphi)$ is false if $\nu(t) \neq \varnothing$ and $\nu(t) \notin \mathbf{B}$.*
- *If φ is a Boolean combination of conditions using operators $\wedge$, $\vee$ and $\neg$, then the truth value of $\mu(\varphi)$ is the given by 3-valued logic presented in Table 2.1.*

We write $\mu \models \varphi$ to mean that φ is true over μ.

Now we are ready to present the semantics of SPARQL.

Definition 2.11 (SPARQL semantics)**.** *Let G be a RDF graph, and Q be a SPARQL query. Then, the evaluation of Q over G, denoted $[\![Q]\!]_G$, is the set of mappings recursively defined as follows:*

- *If Q is a basic graph pattern, then $[\![Q]\!]_G$ is the set of all mappings $\mu|_{\text{var}(Q)}$ such that $\mu(Q') \subseteq G$, where Q' is the result of replacing all blank nodes in Q by fresh variables, and μ is a mapping with $\text{dom}(\mu) = \text{var}(Q')$.*
- *If Q is $(Q_1$ AND $Q_2)$ then $[\![Q]\!]_G = [\![Q_1]\!]_G \bowtie [\![Q_2]\!]_G$.*
- *If Q is $(Q_1$ UNION $Q_2)$ then $[\![Q]\!]_G = [\![Q_1]\!]_G \cup [\![Q_2]\!]_G$.*

Table 2.1: SPARQL Three-valued logic

p	q	$p \wedge q$	$p \vee q$
true	true	true	true
true	error	error	true
true	false	false	true
error	true	error	true
error	error	error	error
error	false	false	error
false	true	false	true
false	error	false	error
false	false	false	false

p	$\neg p$
true	false
error	error
false	true

- *If Q is* (Q_1 MINUS Q_2) *then* $[\![Q]\!]_G = [\![Q_1]\!]_G - [\![Q_2]\!]_G$.
- *If Q is* (Q_1 OPT Q_2) *then* $[\![Q]\!]_G = [\![Q_1]\!]_G \bowtie [\![Q_2]\!]_G$.
- *If Q is* (SELECT $\mathcal{X}$ WHERE Q_1) *then* $[\![Q]\!]_G$ *is the set of mappings* $\mu|_{\mathcal{X}}$ *such that* $\mu \in [\![Q_1]\!]_G$.
- *If Q is* (VALUES $\mathcal{X}$ Ω) *then* $[\![Q]\!]_G$ *is the set of mappings* Ω.
- *If Q is* (Q_1 FILTER φ) *then* $[\![Q]\!]_G$ *is the set of mappings* $\mu \in [\![Q_1]\!]_G$ *such that* $\mu(\varphi)$ *is true (see Definition 2.10).*
- *If Q is* (Q_1 BIND $f(?x_1, \ldots, ?x_n)$ AS ?y) *then let* μ' *be the mapping* $\text{fill}(\mu, \{?x_1, \ldots, ?x_n\})$, *and* y_μ *be the value* $f(\mu'(?x_1), \ldots, \mu'(?x_n))$. *Then,* $[\![Q]\!]_G$ *is the set of SPARQL mappings*

$$\{\mu \in [\![Q_1]\!]_G \mid y_\mu = \varnothing\} \cup \{\mu \cup \{?y \mapsto y_\mu\} \mid \mu \in [\![Q_1]\!]_G \text{ and } y_\mu \neq \varnothing\}.$$

The following definition of $\text{dom}(Q)$ corresponds to the set of variables that in the specification [36, §18.2.1] are called the *in-scope* variables of a query Q. Despite the fact that the output of a SPARQL is not a relation because its mappings can have different domains (e.g., when Q is a union of queries with different variables), most engines use $\text{dom}(Q)$ as the attributes of the "relation" that result of evaluating Q.

Definition 2.12 (Domain of a SPARQL query)**.** *Let (s, p, o) be a triple pattern, Q_1, Q_2 be SPARQL queries, φ be a filter condition, and $\mathcal{X}$ be a set of variables. The domain of a SPARQL query Q, denoted* $\text{dom}(Q)$, *is defined recursively as follows:*

$$\begin{aligned}
\text{dom}((s,p,o)) &= \text{var}((s,p,o)),\\
\text{dom}(Q_1 \text{ AND } Q_2) &= \text{dom}(Q_1) \cup \text{dom}(Q_2),\\
\text{dom}(Q_1 \text{ UNION } Q_2) &= \text{dom}(Q_1) \cup \text{dom}(Q_2),\\
\text{dom}(Q_1 \text{ MINUS } Q_2) &= \text{dom}(Q_1),\\
\text{dom}(Q_1 \text{ FILTER } \varphi) &= \text{dom}(Q_1),\\
\text{dom}(Q_1 \text{ OPT } Q_2) &= \text{dom}(Q_1) \cup \text{dom}(Q_2),\\
\text{dom}(\text{SELECT } \mathcal{X} \ Q_1) &= \mathcal{X},\\
\text{dom}(\text{VALUES } \mathcal{X} \ \Omega) &= \mathcal{X},\\
\text{dom}(Q_1 \text{ BIND}(f(?x_1, \ldots, ?x_n) \text{ AS } ?y)) &= \text{dom}(Q_1) \cup \{?y\}.
\end{aligned}$$

Part II

Incomplete data in SPARQL

Chapter 3

Blank nodes as unknown values

According to Hogan et al. [42, 53], blank nodes are used in RDF data for manifold reasons. They describe a resource for which an IRI is not defined, or to indicate the existence of an *unknown value*. In this chapter we study the use of blank nodes as unknown values and describe the problem of this usage regarding certain answers in SPARQL. The material is based on ideas presented in [34] and developed and presented in [41].

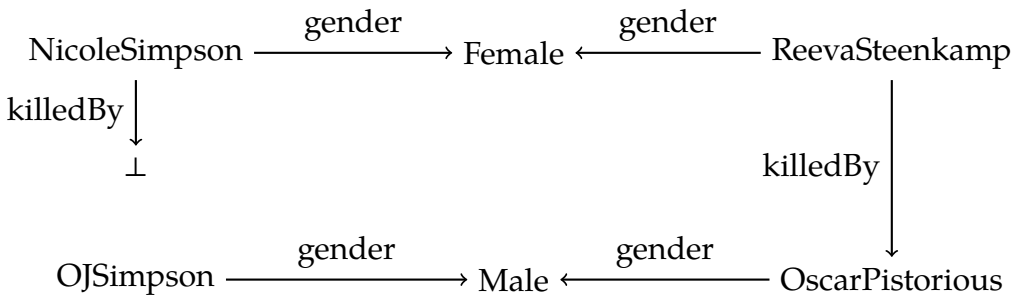
For historical reasons, the semantics of blank nodes in SPARQL do not follow the original interpretation of blank nodes as unknown values defined in the RDF semantics.[1] Our goal (in this chapter) is not try to fix this mismatch, but to address the question of how to offer to RDF data consumers the option to return certain answers to SPARQL queries when

[1] It would be an historical investigation to clear the reasons (if any) why such a mismatch occurs, why SPARQL was designed the way it is. Let us at least inform the reader about some facts that can help understand this development. The Semantic Web ecosystem consists in a family of standards (SPARQL, OWL, SHACL, and others) each one assumes a particular semantics for the underlying RDF syntax. The editors of the standards for each of these pieces had different perspectives as they are defining their tools for different purposes. The RDF semantics was designed with an ambitious goal in mind: to serve as universal language over the Web. Over it, the query language SPARQL designer had the end of querying pieces of RDF data. Their design was inspired by SQL, and thus ended with several incompatibilities with the original RDF semantics. The most prominent are the close world assumption and the unique name assumption. The mismatch described in Example 3.1 corresponds to the unique name assumption mismatch for the particular case of blank nodes. As Reiter shows [63], unknown values can be interpreted as elements that are not affected by the unique name assumption. As we already mentioned, the Semantic Web standards were defined with different purposes in mind. Blank nodes in particular, have thus different uses: 1) RDF publishers use blank nodes to create data structures as lists or reified statements without the overhead of defining "unnecessary" IRIs. This is useful to reduce the publishers work and thus encourage publication in the RDF format. In this context, blank nodes are assumed to be different entities for data consumers. 2) With the OWL language we can define an ontology stating that a resource belongs to a class without requiring an IRI to name that class. This is particularly useful to define complex classes as a composition of several unnamed classes. Ontology consumers do not assume blank nodes refer to different entities unless this can be inferred. 3) RDF datasets make sometimes additional assumptions on what blank nodes represent. For instance, in Wikidata entities named by IRIs are assumed to be different whereas blank nodes are used to denote unknown values. We can guess that SPARQL evaluates $\bot_1 \neq \bot_2$ as true (despite this is not certain regarding the RDF semantics) because the SPARQL designers assume a particular use case for this formula. And last, but not least, the SPARQL specification sometimes contradicts itself regarding the semantics of blank nodes. For instance, blank nodes occurring in triple patterns behave as unknown values that can match any value in the data.

assuming blank nodes do represent unknown values (as for example occurs in the relevant case of Wikidata).

The following example illustrates our motivation.

Example 3.1. *Let G be the following graph taken from the Wikidata [69] knowledge-base:*



and let Q_1 and Q_2 be the following SPARQL queries:

$$Q_1: \quad \text{(SELECT ?x WHERE ((?x, killedBy, ?y) AND (?y, gender, Male)))},$$
$$Q_2: \quad \text{(SELECT ?x WHERE ((?x, killedBy, ?y) MINUS (?y, gender, Male)))},$$

that ask, respectively, for persons ?x *killed by somebody of gender male, and for persons* ?x *killed by somebody whose gender is not male.*

In the graph data shown, according to the RDF semantics, the blank node ($\perp$) denotes that Nicole Brown Simpson (a victim of homicide) has a killer, but that her killer is unknown.

For Q_1, SPARQL returns a single solution: $\{?x \mapsto \text{ReevaSteenkamp}\}$. It is a certain answer regarding the RDF semantics, that is, the only killer that, from the data, we are certain that is male. On the other hand, for Q_2, SPARQL again returns a single solution: $\{?x \mapsto \text{NicoleSimpson}\}$. In this case, this answer is uncertain because we do not know that Nicole Simpson's killer was not male; in fact, $\perp$ could refer to a male in the data.

Example 3.1 is not fictional: it was taken from the Wikidata [69] knowledge-base, which publishes data as RDF and provides a public SPARQL query interface on the Web. Wikidata is the main case of use of blank nodes as existential variables. While the example uses real data, for readability, we use fictitious IRIs. In reality, Wikidata uses opaque identifiers, e.g. `w:Q268018` to represent Nicole Simpson.

The use of blank nodes to represent unknown values in Wikidata agrees to the standard semantics of blank nodes in RDF (see Definition 2.4) in the sense that we cannot infer that, for example, $\perp$ and OJSimpson are different resources in every interpretation of the graph G. That is, we cannot assume that OJSimpson does not killed NicoleSimpson since there is an interpretation of the graph where $\perp$ and OJSimpson refer to the same resource. However, as Example 3.1 shows, SPARQL does not follow the standard existential semantics of blank nodes in RDF.[2]

[2]The Example 3.1 highlights a key problem in the current SPARQL semantics when dealing with unknown values. In the first query only certain answers are returned: answers that hold no matter whom the unknown value(s) refer(s) to. In the second query uncertain answers are returned: answers that may or may not hold depending on whom the unknown value(s) refer(s) to.

The example shown motivates the need to offer certain answers in SPARQL. In this chapter we will address the following questions: should users be offered a *choice* to only return certain answers? Is such a choice important? And what would be its cost?

Regarding cost, query evaluation under certain answer semantics incurs in a significant computational overhead; for example, considering queries expressed in the standard relational algebra, if we consider only "complete databases" without unknown values, the data complexity of the standard query evaluation problem is AC^0; on the other hand, the analogous complexity with unknown values under certain answer semantics leads to coNP-hardness [2].[3]

In terms of need, a recent study of blank nodes suggests that 66% of websites publishing RDF use blank nodes, with the most common use-cases being to represent resources for which no IRI has been defined (e.g., for representing RDF lists), or to represent unknown values [42]. The methods proposed in this thesis specifically target datasets using blank nodes in the second sense; such datasets include Wikidata [69], as illustrated in Example 3.1.

In terms of cost, work by Guagliardo and Libkin [33] offers promising results in terms of the practical feasibility of *approximating* certain answers in the context of relational databases, returning only (but not all) such answers. Performance results suggest that such approximations have reasonable runtimes when compared with standard SQL evaluation. Furthermore, their implementation strategy is based on query rewriting over off-the-shelf query engines, obviating the need to build special-purpose engines, minimizing implementation costs.

In this chapter, we thus tackle the question: should users be given a choice of certain semantics for SPARQL under the assumption that blank nodes occurring in the data represent existential variables? Along these lines, we adapt the methods of Guagliardo and Libkin [33] in order to propose and evaluate (to the best of our knowledge) the first approach that guarantees to return only certain answers for a fragment of SPARQL (capturing precisely the relational algebra) over RDF datasets with existential blank nodes, further developing a set of concrete rewriting strategies for the SPARQL setting. We evaluate our rewriting strategies for two popular SPARQL engines—Virtuoso and Fuseki—offering comparison of performance between base queries and rewritten queries (under various strategies), and a comparison of our SPARQL and previous SQL results [33]. We further analyze Wikidata user queries to see if a certain answer semantics would really affect the answers over real-world queries and data, performing further experiments to ascertain costs in this setting.

Ours is not the first work to explore a certain answer semantics for SPARQL. Ahmetaj et al. [3] define a certain answer semantics for SPARQL, but their focus is on supporting OWL 2 QL entailment for queries based on *well-designed patterns* [58], and in particular on complexity results for query evaluation, containment and equivalence. Arenas and Perez [11] also consider a certain answer semantics for SPARQL towards studying conditions for

[3]Following the footnotes discussion on the design of SPARQL: One can thus hardly blame the design committee of the SPARQL language for having chosen to initially overlook the issue of unknown values (see footnote 1). It not only would have escalated considerably the complexity of query evaluation, but the cost and complexity of correctly implementing the new standard would likewise have increased considerably.

monotonicity: a semantic condition whereby answers will remain valid as further data is added to the system; as such, certain answers in their work are concerned with an *open world semantics*. However, determining if a query is *weakly monotonic*—i.e., monotonic disregarding unbound values—is undecidable. Hence later Arenas and Ugarte [12] proposed a syntactic fragment of SPARQL that closely captures this notion of weak monotonicity. In contrast to such works, we maintain SPARQL's negation features [5] with a closed world semantics. In many contexts, users are interested in writing SPARQL queries regarding resources that the present dataset does/does not contain. A relevant practical use case is, as we discuss later, queries for the Wikidata SPARQL service using difference.

To the best of our knowledge, the work on which we based this chapter [41] is the first to investigate a certain answer semantics for SPARQL under the presence of blank nodes in RDF data interpreted as existential variables.

Organization of this chapter

- In Section 3.1 we present a semantics, called the naive semantics of RDF, that is a simplification of the standard simple entailment semantics described in Section 2.1. The naive semantics allows the application of the results of the literature of incomplete data in the relational model to the RDF model and SPARQL.
- In Section 3.2 we show how under the naive semantics RDF graphs can be seen as tables with marked nulls, called V-tables by Imielinski and Lipski [43]. Also we show how relational algebra queries can be defined over RDF and we present a SPARQL fragment where queries can be expressed in terms of relational algebra queries.
- In Section 3.3 we study how to approximate certain answers in SPARQL by adapting the methods from the relational algebra.
- In Section 3.4 we present an initial cost–benefit analysis of a certain answer semantics for SPARQL. This analysis includes a study of the experimental cost of approximating certain answers using the methods described in the previous section. We address the problem of how the results are changed by using these approximation methods in a real-world setting.
- In Section 3.5 we present the conclusions of this chapter.

3.1 Naive semantics of RDF

The Example 3.1 from Wikidata made some assumptions about RDF semantics. These reflect practical assumptions that Wikidata gives to RDF datasets, that we follow throughout this chapter. These assumptions are the following:

1. *The unique name assumption is made for IRIs and literals.* For instance, in Example 3.1 we assume that Nicole Brown Simpson and Reeva Steenkap are not the same person. However, the unique name assumption is not made for blank nodes. For instance, in Example 3.1 the blank node $\perp$ may refer to any person in the database or even people that do not occur in the database.

2. *The closed-world assumption is made for RDF graphs.* This assumption makes more interesting the problem of query evaluation. For example, consider the query asking for *all people that have not been killed by a male*. Under the open-world assumption this query has no certain answers because for every person a there is an interpretation where a was killed by b and b is a male. Under the closed-world there are cases in which a is an answer of this question. Indeed, if the only fact in the database is "b killed a" then a is an answer since no interpretation states than b is a male. Furthermore, b is also an answer because there is no interpretation where b has been killed. If we extend the database with the fact "$\perp$ has gender male" then a is no longer an answer since $\perp$ can be interpreted as b whereas b is still an answer since there is no interpretation where b is killed. If we extend again the database by including the fact "b killed $\perp$" then b is no longer an answer since b could have killed himself.

The two assumptions described above motivate the semantics for RDF graphs we present next. This semantics, called *naive* in what follows, restricts the standard semantics presented in Definition 2.4 to the interpretations that satisfy these assumptions.

Definition 3.1 (Naive semantics of an RDF graph)**.** *Let G be an RDF graph. $\mathcal{A} = (R, P, E, I)$ is a naive model of G, denoted $\mathcal{A} \models_{\text{naive}} G$, if it is a model of G (see Definition 2.4) where:*

1. *I is bijective (i.e., each name in the vocabulary represents a different element in the domain and each element in the domain has a name in the vocabulary).*
2. *There exists a function $v : \mathbf{B} \to V$ such that:*
   (a) *for each $(s, p, o) \in G$, it holds that $I(p) \in P$ and $(I_v(s), I_v(o)) \in E(I(p))$.*
   (b) *for each predicate $p' \in P$ and pair $(s', o') \in E(p')$ there exists a triple $(s, p, o) \in G$ such that $I_v(s) = s'$, $I(p) = p'$, and $I_v(o) = o'$.*

The first condition of Definition 3.1 restricts interpretations to those that satisfy the unique-name assumption. The injectivity of function I ensures that each element in the vocabulary corresponds to a unique element in the domain. The surjectivity of function I is introduced to state a one to one correspondence among elements in the domain and names in the vocabulary.

The second condition of Definition 3.1 restricts interpretations to those that satisfy the closed-world assumption. Interpretations $\mathcal{A}$ of a graph G already satisfy the existence of a function v that holds condition (a), that is, every triple in G is interpreted by $\mathcal{A}$. The additional condition (b) ensures the closed-world assumption since it restricts the facts of $\mathcal{A}$ to the facts that are interpretations of the triples in G.

3.2 RDF graphs as V-tables

To study the naive semantics of RDF graphs we use the theory of V-tables proposed by Imielinski and Lipski [43]. Informally a V-table is a relational table that can have existential variables (besides constants) in its entries. After the following example, we will give some basic definitions and notation concerning to V-tables.

Example 3.2. *Consider the following V-table:*

Person	Office
Alice	$\perp_1$
Bob	$\perp_2$
Carl	$\perp_1$

The elements Person *and* Office *above the horizontal line are called the attributes of the table, whereas the elements below it are called the values of the table. The values* Alice, Bob, *and* Carl *are called constants because represent distinguishable known elements, whereas the values $\perp_1$, $\perp_2$, and $\perp_3$ are called marked nulls and represent unknown values. The offices of all persons in this example are therefore unknown. However, because the office of* Alice *and* Carl *are denoted with the same marked null, we know that they share office.*

As the example above illustrates, V-tables are defined over three disjoint sets, namely the set of *marked nulls*, the set of *constants*, and the set of *attributes*. To make explicit the relation among V-tables and RDF data and SPARQL formalism, we assume that these sets are defined using the base sets of RDF: the set of marked nulls corresponds to $\mathbf{B}$, the set of constants to $\mathbf{C} = \mathbf{I} \cup \mathbf{L}$, and the set of attributes corresponds to $\mathbf{V}$, the variables in SPARQL.

A *V-table over* a finite set of attributes $\mathcal{X} \subset \mathbf{V}$ is a finite set of functions $\mu : \mathcal{X} \to \mathbf{C}$. Given a V-table R over a set of attributes $\mathcal{X}$, we write $\text{dom}(R)$ to denote the set of attributes $\mathcal{X}$ on which R is defined. The elements of a V-table R are called the tuples or mappings of R. A V-table R can thus be considered a particular type of a set of mappings that are returned by SPARQL queries, where the mappings $\mu \in R$ hold that $\text{dom}(\mu) = \text{dom}(R)$. Recall that, in general, the mappings that a SPARQL query returns do not necessarily have the same domain.

A *database schema* is a pair (T, dom) where T is a finite set, called the set of table names and, $\text{dom} : T \to 2^{\mathbf{V}}$ is a function that associates to each table name a finite set of attributes. A *V-database D* over the database schema (T, dom) is a function that associates each table name $r \in T$ to a V-table over $\text{dom}(r)$, denoted r^D. V-tables and V-databases are said to be *regular* if they have no blank nodes.

The semantics of a regular V-database D is defined as usual by the model that interprets D. This approach, called *model-theoretic semantics* by Reiter [65], goes as follows. Let D be a regular V-database over a database schema $\mathcal{S} = (T, \text{dom})$, and $\mathcal{A} = (R, P, E, I)$ be an interpretation over the vocabulary $\text{adom}(D) \cup T$ where $\text{adom}(D)$ denotes the constants occurring in D. $\mathcal{A}$ is said to be an interpretation of D, denoted $\mathcal{A} \models D$, if the following conditions hold:

1. $I : \text{adom}(D) \cup T \to R \cup P$ is a bijective function where $I(\text{adom}(D)) = R$ and $I(T) = P$.
2. Let $r \in T$ be a table name where $I(r) = p$ and the attributes $\text{dom}(r)$ in lexicographical order are $A_1, \ldots, A_n$. The function E associates each predicate p to an extension $p^{\mathcal{A}} \subset R^n$ where $\{A_1 \mapsto a_1, \ldots, A_n \mapsto a_n\} \in r^D$ if and only if $(I(a_1), \ldots, I(a_n)) \in p^{\mathcal{A}}$.

So far, we presented the semantics of regular V-databases. To describe the semantics of V-databases, we need to introduce some notions.

A *blank node valuation* (or simply a valuation) is a function $h : \mathbf{B} \cup \mathbf{C} \to \mathbf{B} \cup \mathbf{C}$ that preserves the constants (i.e., that only changes blank nodes).

Let h be a valuation. Then:

- Given a tuple μ of a V-table, we write $h(\mu)$ to denote the tuple μ' where $\mu'(?x) = h(\mu(?x))$ for every variable ?x in $\text{dom}(\mu)$.
- Given a V-table R, we write $h(R)$ to denote the V-table consisting in the tuples $\{h(\mu) \mid \mu \in R\}$.
- Given a V-database D over a database schema (T, dom), we write $h(D)$ to denote the V-database where $r^{h(D)} = h(r^D)$ for each table name $r \in T$.
- Given two V-databases D_1 and D_2 over the same database schema, we say that there exists a map $h : D_1 \to D_2$ if and only if h is a valuation such that $D_2 = h(D_1)$.

Now we are ready to present the semantics of V-databases.

Definition 3.2 (Semantics of V-databases)**.** *The semantics of a V-database D is given by the set, denoted* $\text{repr}(D)$*, of regular V-databases D' where there exists a map $h : D \to D'$. We say that an interpretation $\mathcal{A}$ interprets a V-database D if $\mathcal{A} \models D'$ for a database $D' \in \text{repr}(D)$. We say that a V-database D_1 entails a V-database D_2, denoted $D_1 \models D_2$, if each interpretation of D_1 is also an interpretation of D_2.*

Since the interpretations of a V-database D are defined as the interpretations of regular databases in $\text{repr}(D)$, given two databases D_1 and D_2, it holds that $D_1 \models D_2$ if and only if $\text{repr}(D_1) \subseteq \text{repr}(D_2)$.

Likewise Theorem 2.1 does for RDF graphs, Lemma 3.1 states the relation between entailment and maps among V-tables.

Lemma 3.1. *Let R_1 and R_2 be two V-tables over a set of attributes $\mathcal{X}$. Then it holds: $R_1 \models R_2$ if and only if there is a map $h : R_2 \to R_1$.*

Proof. This proof is structured in two parts, one for each direction of the equivalence.

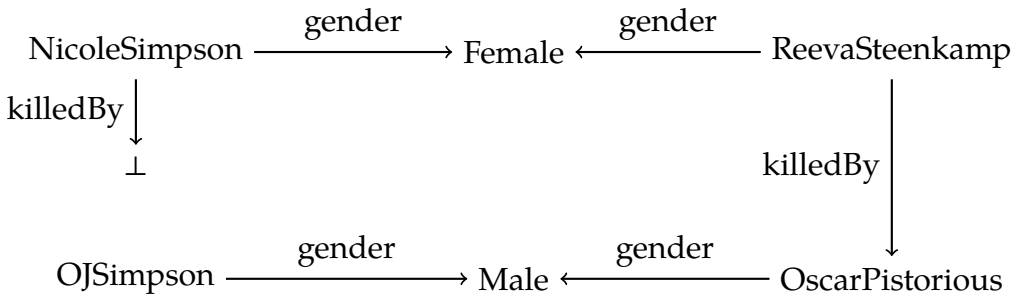
Part 1. Let R_1 and R_2 be two V-tables over a set $\mathcal{X}$ of attributes where there exists a map $h : R_2 \to R_1$. Let $R_3 \in \text{repr}(R_2)$ be a regular V-table. By definition, there exists a map $g : R_3 \to R_2$. By construction, $g \circ f : R_3 \to R_1$ is a map. Hence, $R_1 \models R_2$.

Part 2. Let $\mathcal{R}_\mathcal{X}$ be the set of all V-tables over the set $\mathcal{X}$ of attributes, and $\leq$ be the relation over $\mathcal{R}_\mathcal{X}$ such that $R_a \leq R_b$ if and only if there exists a map $h : R_a \to R_b$. Let $\equiv$ be the equivalency relation defined as $R_a \equiv R_b$ if and only if $R_a \leq R_b$ and $R_b \leq R_a$. Let $\leq$ be the relation over $\mathcal{R}_\mathcal{X}/\equiv$ defined as usual, i.e., $[R_a]_\equiv \leq [R_b]_\equiv$ if and only if $R_a \leq R_b$. Let $\frown$ be the operation over $\mathcal{R}_\mathcal{X}/\equiv$ defined as $[R_a]_\equiv \frown [R_b]_\equiv = \max\{R_c \mid R_c \leq R_a \text{ and } R_c \leq R_b\}$. By construction, $\frown$ is commutative ($[R_a]_\equiv \frown [R_a]_\equiv = [R_b]_\equiv \frown [R_a]_\equiv$), associative ($[R_a]_\equiv \frown ([R_b]_\equiv \frown [R_c]_\equiv) = ([R_a]_\equiv \frown [R_b]_\equiv) \frown [R_c]_\equiv$), idempotent ($[R_a]_\equiv \frown [R_a]_\equiv = [R_a]_\equiv$), and monotone ($[R_a]_\equiv \frown [R_b]_\equiv \leq [R_a]_\equiv$). Given a subset $A = \{R_1, \ldots, R_n\}$ of $\mathcal{R}_\mathcal{X}$, let $\frown A$ denote $[R_1]_\equiv \frown \cdots \frown [R_n]_\equiv$. The properties of $\frown$ already mentioned imply that, for every relation $R \in \mathcal{R}_\mathcal{X}$, the equality $[R]_\equiv = \frown \text{repr}(R)$ holds. Let R_1 and R_2 be two V-tables in $\mathcal{R}_\mathcal{X}$ where $R_1 \models R_2$. It thus holds that $[R_2]_\equiv = \frown(\text{repr}(R_2) \setminus \text{repr}(R_1)) \frown \frown R_1$. Since $\frown$ is monotone, $[R_2]_\equiv \leq [R_1]_\equiv$ holds. Hence, $R_2 \leq R_1$ and thus there exists a map $h : R_2 \to R_1$. □

An RDF graph G can be seen as a V-database D. We next formalize the translation of an RDF graph as V-database.

Definition 3.3. *The V-database of an RDF graph G, denoted* $\mathrm{vdb}(G)$*, is the one composed by a single V-table named* triples*,* $\mathrm{dom}(\text{triples}) = \{?\mathrm{s}, ?\mathrm{p}, ?\mathrm{o}\}$*, and* $\text{triples}^{\mathrm{vdb}(G)} = \{\{?\mathrm{s} \mapsto s, ?\mathrm{p} \mapsto p, ?\mathrm{o} \mapsto o\} \mid (s, p, o) \in G\}$.

Example 3.3. *Let G be the following graph of Example 3.1 that we reproduce here to help the reader:*



The V-database D of the RDF graph G is the one composed by the following V-table:

$$\text{triples}^D = \left[\begin{array}{ccc} ?\mathrm{s} & ?\mathrm{p} & ?\mathrm{o} \\ \hline \text{NicoleSimpson} & \text{gender} & \text{Female} \\ \text{NicoleSimpson} & \text{killedBy} & \bot \\ \text{ReevaSteenkamp} & \text{gender} & \text{Female} \\ \text{ReevaSteenkamp} & \text{killedBy} & \text{OscarPistorious} \\ \text{OJSimpson} & \text{gender} & \text{Male} \\ \text{OscarPistorius} & \text{gender} & \text{Male} \end{array}\right].$$

Similarly, an interpretation of an RDF graph G in the naive semantics can be seen as an interpretation of the V-database $\mathrm{vdb}(G)$ as follows.

Definition 3.4. *Let $\mathcal{A}_G = (R, P, E, I)$ be an interpretation of an RDF graph G. The V-database interpretation corresponding to the interpretation $\mathcal{A}_G$, denoted* $\mathrm{vdb}(\mathcal{A}_G)$*, is the interpretation (R', P', E', I') where $R' = R \cup P$, $P' = \{\text{triples}\}$, $E'(\text{triples}) = \{(s, p, o) \mid p \in P \text{and} (s, o) \in E(p)\}$, and I' is the identity over $R \cup P$.*

So far, we have defined a translation from the RDF data model to the V-database model (including databases and interpretations). We next show some properties of the naive semantics of RDF graphs that correspond to similar properties of V-databases.

Lemma 3.2. *Let G be an RDF graph, and $\mathcal{A}$ an interpretation over the vocabulary $V \supseteq \mathrm{voc}(G)$. Then, $\mathcal{A} \models_{\text{naive}} G$ if and only if* $\mathrm{vdb}(\mathcal{A}) \models \mathrm{vdb}(G)$.

Proof. We prove this lemma in one direction of the equivalence. The inverse direction is omitted since it follows the same argument. Let G be an RDF graph and $\mathcal{A} = (R, P, I, G)$ be an interpretation over a vocabulary V such that $\mathcal{A} \models_{\text{naive}} G$, and $v : \mathbf{B} \to V$ be the function described in Definition 3.1. By Lemma 3.1, $v(\mathrm{vdb}(G)) \models \mathrm{vdb}(G)$. By construction, $\mathrm{vdb}(\mathcal{A}) \models v(\mathrm{vdb}(G))$. Hence, $\mathrm{vdb}(\mathcal{A}) \models \mathrm{vdb}(G)$. □

Likewise Theorem 2.1 does for the simple entailment semantics of RDF graphs, Lemma 3.3 shows the relation between the naive semantics of simple graphs and the maps among them.

Lemma 3.3. *Let G_1 and G_2 be RDF graphs. $G_1 \models_{\text{naive}} G_2$ if and only if there is a map $h : G_2 \to G_1$ where $h(G_2) = G_1$.*

Proof. We prove this lemma in one direction of the equivalence. The inverse direction is omitted since it follows the same argument. Let G_1 and G_2 be RDF graphs such that $G_1 \models_{\text{naive}} G_2$, and $\mathcal{A}$ be an interpretation over the vocabulary $V = \text{voc}(G_1) \cup \text{voc}(G_2)$ such that $\mathcal{A} \models_{\text{naive}} G_1$. By definition, $\mathcal{A} \models_{\text{naive}} G_2$. By Lemma 3.2, $\text{vdb}(\mathcal{A}) \models \text{vdb}(G_1)$ and $\text{vdb}(\mathcal{A}) \models \text{vdb}(G_2)$ hold. Since $\text{vdb}(\mathcal{A})$ is an arbitrary interpretation of the V-database $\text{vdb}(G_1)$, it holds that $\text{vdb}(G_1) \models \text{vdb}(G_2)$. By Lemma 3.1, it holds that there is a map $h : \text{vdb}(G_2) \to \text{vdb}(G_1)$. By construction, $h : G_2 \to G_1$. □

Corollary 3.1. *Let G_1 and G_2 be two simple RDF graphs. It holds then that $G_1 \models_{\text{naive}} G_2$ implies $G_1 \models G_2$.*

Proof. It follows from Theorem 2.1 and Lemma 3.3. □

Since the naive semantics of RDF graphs introduces the unique name assumption over IRIs and the closed world assumption, the certain answers of a query under the naive semantics are not necessarily certain over the simple entailment semantics. For instance, the query "get the male people that have not been killed" over the graph of Example 3.1 has OJSimpson and OscarPistorious as certain answers under the naive semantics. According to the simple entailment semantics (Definition 2.5) these answers are not certain because the world is open. On the contrary, the certain answers under the simple entailment semantics are also certain under the naive semantics (as Corollary 3.1 states), because the interpretations of an RDF graph G under the naive semantics is a subset of the interpretations of G under the simple entailment semantics.

3.2.1 The V-tables relational algebra

We assume familiarity with the operators *join* ($\bowtie$), *selection* (σ_φ), *renaming* ($\rho_{A/B}$), *union* ($\cup$), and *difference* ($-$) of the named perspective of the relational algebra [1]. We also assume familiarity with the *naive evaluation* of relational algebra queries over V-tables, that operate null values as if they were constants [43]. We next present the syntax and semantics of relational algebra of V-tables to summarize the notations used in this thesis.

Definition 3.5 (Relational algebra of V-tables)**.** *The notion of relational algebra query Q over a database schema (T, dom), the attributes of Q, denoted $\text{dom}(Q)$, and the semantics of Q over a database D over (T, dom), denoted $[\![Q]\!]_D$, are defined recursively as follows:*

1. *If Q is a table name $r \in T$ then Q is a relational algebra query, $\text{dom}(Q) = \text{dom}(r)$, and $[\![Q]\!]_D = r^D$.*
2. *If Q is a V-table R then Q is a relational algebra query, $\text{dom}(Q) = \text{dom}(R)$, and $[\![Q]\!]_D = R$.*

3. *If Q is a relational algebra query and $\mathcal{X}$ is a set of attributes where $\mathcal{X} \subseteq \mathrm{dom}(Q)$, then $\pi_{\mathcal{X}}(Q)$ is a relational algebra query called the projection of Q in the attributes $\mathcal{X}$,* $\mathrm{dom}(\pi_{\mathcal{X}}(Q)) = \mathcal{X}$, *and* $[\![Q]\!]_D = \{\mu_{|\mathcal{X}} \mid \mu \in [\![Q]\!]_D\}$.

4. *If Q_1 and Q_2 are relational algebra queries, then $Q_1 \bowtie Q_2$ is a relational algebra query called the join of Q_1 and Q_2,* $\mathrm{dom}(Q_1 \bowtie Q_2) = \mathrm{dom}(Q_1) \cup \mathrm{dom}(Q_2)$, *and*

$$[\![Q_1 \bowtie Q_2]\!]_D = \{\mu_1 \backsim \mu_2 \mid \mu_1 \in [\![Q_1]\!]_D, \mu_1 \in [\![Q_1]\!]_D, \text{ and } \mu_1 \sim \mu_2\},$$

   where $\mu_1 \sim \mu_2$ denotes that μ_1 and μ_2 are compatible (i.e., agree in the common attributes), and $\mu_1 \backsim \mu_2$ denotes the tuple that is compatible with μ_1 and μ_2 and has $\mathrm{dom}(\mu_1) \cup \mathrm{dom}(\mu_2)$ *as domain.*

5. *If Q_1 and Q_2 are relational algebra queries where* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = \mathcal{X}$, *then $Q_1 \cup Q_2$ and $Q_1 - Q_2$ are relational algebra queries,* $\mathrm{dom}(Q_1 \cup Q_2) = \mathrm{dom}(Q_1 - Q_2) = \mathcal{X}$, *and:*

$$[\![Q_1 \cup Q_2]\!]_D = \{\mu \mid \mu \in [\![Q_1]\!]_D \text{ or } \mu \in [\![Q_2]\!]_D\},$$
$$[\![Q_1 - Q_2]\!]_D = \{\mu \mid \mu \in [\![Q_1]\!]_D \text{ and } \mu \notin [\![Q_2]\!]_D\}.$$

6. *A term is an attribute name, a constant or a marked null. An atom is an equality $t_1 = t_2$ where t_1 and t_2 are terms. A selection formula φ is a Boolean combination of atoms (i.e., given two selection formulas φ and ψ, then $\neg\varphi$, $\varphi \wedge \psi$ and $\varphi \vee \psi$ are selection formulas). Given a relational algebra query Q and a selection formula φ where every attribute name occurring in φ is in* $\mathrm{dom}(Q)$, *the expression $\sigma_\varphi(Q)$ is a relational algebra query,* $\mathrm{dom}(\sigma_\varphi(Q)) = \mathrm{dom}(Q)$, *and* $[\![\sigma_\varphi(Q)]\!]_D = \{\mu \in [\![Q]\!]_D \mid \mu(\varphi) \text{ is true}\}$, *where the relation $\mu(\varphi)$ is defined recursively as follows:*

   - *If φ is an atom $t_1 = t_2$, then $\mu(\varphi)$ is true if the result of replacing in φ every variable* ?x *by μ(?x) is an equality where both sides have the same value (i.e., the same constant or the same marked null). Otherwise, $\mu(\varphi)$ is false.*
   - *If φ is non-atomic, then $\mu(\varphi)$ is evaluated according to the standard semantics of the Boolean connectors $\neg$, $\wedge$, and $\vee$ (e.g., $\mu(\psi_1 \wedge \psi_2)$ is true if and only if $\mu(\psi_1)$ and $\mu(\psi_2)$ are true).*

It is necessary to emphasize that the V-tables algebra evaluates selection formulas according to two-valued logic, where marked null values are evaluated as if they were constants. To see this, consider the query $\sigma_{\neg(?x=a)}(R)$. If a mapping $\mu = \{?x \mapsto \bot\}$ is a tuple of relation R, then $\mu(\neg(?x = a))$ is true since $\mu(?x = a)$ is false because $\bot$ and a are different values (constants). An alternative, followed by SQL, is to evaluate null values using three-valued logic (i.e., using *true*, *false* and *unknown* as the truth values). According to the three-valued logic evaluation, $\mu(?x = a)$ is unknown because $\bot$ denotes an unknown value. We therefore cannot know if $\bot$ and a have the same value or different values. By the same argument, according to the three-valued logic evaluation, $\mu(\neg(?x = a))$ is also unknown.

3.2.2 The relational algebra of SPARQL

In this section we present a fragment of SPARQL that precludes the occurrence of unbound values, that is, for every query Q in this fragment and every graph G, it holds that

$\text{dom}(\mu) = \text{dom}(Q)$ for each solution $\mu \in [\![Q]\!]_G$. Since all solution mappings have the same domain, the queries of this fragment can be expressed using the algebra of V-tables.

The syntax of the relational fragment of SPARQL is defined as follows.

Definition 3.6 (Relational fragment of SPARQL)**.** *The syntax of queries in the relational fragment of SPARQL is defined as follows:*

1. *A triple pattern is a query in the relational fragment.*
2. *Given a SPARQL query Q in the relational fragment and a set of variables $\mathcal{X}$ where $\mathcal{X} \subseteq \text{dom}(Q)$, the query* (SELECT $\mathcal{X}$ WHERE Q) *is a query in the relational fragment.*
3. *A query* (VALUES $\mathcal{X}$ Ω) *where* $\text{dom}(\mu) = \mathcal{X}$ *for each* $\mu \in \Omega$ *is a query in the relational fragment.*
4. *Given a SPARQL query Q in the relational fragment and a filter-condition φ where* $\text{var}(\varphi) \subseteq \text{dom}(Q)$ *and φ is a Boolean combination of equality atoms, the query* (Q FILTER φ) *is a query in the relational fragment.*
5. *Given two SPARQL queries Q_1 and Q_2 in the relational fragment, the queries* (Q_1 AND Q_2) *and* (Q_1 MINUS Q_2) *are queries in the relational fragment.*
6. *Given two SPARQL queries Q_1 and Q_2 in the relational fragment where* $\text{dom}(Q_1) = \text{dom}(Q_2)$*, the query* ($Q_1$ UNION Q_2) *is a query in the relational fragment.*

As we already mentioned, all solution mappings of a query in the relational fragment have the same domain. This is formalized as follows.

Lemma 3.4. *Let Q be a SPARQL query in the above relational fragment. Then, for every graph G it holds that* $\text{dom}(\mu) = \text{dom}(Q)$ *for each mapping* $\mu \in [\![Q]\!]_G$.

Proof. It can be shown by induction on the structure of the query Q.

If Q is a triple pattern then, by definition, $\text{dom}(Q)$ is the set of variables occurring in Q, and $\text{dom}(\mu) = \mu$ for every solution μ of Q.

If Q has the form (SELECT $\mathcal{X}$ WHERE P) then, by induction, each solution μ' of query P holds that $\text{dom}(\mu') = \text{dom}(P)$. Since $\mathcal{X} \subseteq \text{dom}(P)$, it holds that for every solution μ of Q there exists a solution μ' of P such that $\mu'|_{\mathcal{X}} = \mu$. Hence, $\text{dom}(\mu) = \text{dom}(\mu'|_{\mathcal{X}}) = \mathcal{X}$. By definition, $\text{dom}(Q) = \mathcal{X}$. Hence, $\text{dom}(\mu) = \text{dom}(Q)$.

If Q has the form (VALUES $\mathcal{X}\Omega$) then, by definition, $\text{dom}(\mu) = \mathcal{X} = \text{dom}(Q)$, for each solution μ of Q.

If Q has the form (P FILTER φ) then each solution μ of Q is also a solution of query P. By induction, $\text{dom}(\mu) = \text{dom}(P)$. Since $\text{dom}(Q) = \text{dom}(P)$, it holds that $\text{dom}(\mu) = \text{dom}(Q)$.

If Q has the form Q_1 AND Q_2 then for each solution μ of Q there exists two solutions μ_1 of Q_1 and μ_2 of Q_2 such that $\mu = \mu_1 \backsim \mu_2$. By induction, $\text{dom}(\mu_1) = \text{dom}(Q_1)$ and $\text{dom}(\mu_2) = \text{dom}(Q_2)$. By definition, $\text{dom}(Q) = \text{dom}(Q_1) \cup \text{dom}(Q_2)$. Hence, $\text{dom}(\mu) = \text{dom}(Q)$.

If Q has the form Q_1 MINUS Q_2 then for each solution μ of Q, μ is also a solution of Q_1. By induction, $\text{dom}(\mu) = \text{dom}(Q_1)$. By definition, $\text{dom}(Q) = \text{dom}(Q_1)$. Hence, $\text{dom}(\mu) = \text{dom}(Q)$.

If Q has the form Q_1 UNION Q_2 then for each solution μ of Q it holds that μ is a solution of Q_1 or μ is a solution of Q_2. Without loss of generality, assume that μ is a solution of Q_1. By induction, $\text{dom}(\mu) = \text{dom}(Q_1)$. By definition, $\text{dom}(Q) = \text{dom}(Q_1)$. Hence, $\text{dom}(\mu) = \text{dom}(Q)$. □

The important property of the SPARQL relational fragment is that its queries are expressible as relational algebra queries. We next present the translation procedure from a SPARQL query Q in the relational fragment to a relational algebra query, denoted $\text{ra}(Q)$. The schema of the relational query $\text{ra}(Q)$ is the relational schema of the V-database that results from translating Q according to Definition 3.3. This translation is mostly based in the translation from SPARQL to relational algebra by Cyganiak [21]. The difference is that we restrict his translation to the relational fragment of SPARQL.

If Q is the query (VALUES $\mathcal{X}$ Ω) in the relational fragment of SPARQL, then $\text{ra}(Q)$ is the V-table R where $\text{dom}(R) = \mathcal{X}$ and the tuples of R are the mappings in Ω. Recall that $\text{dom}(\mu) = \mathcal{X}$ for each $\mu \in \Omega$ because the query Q is in the relational fragment of SPARQL.

The translation of a triple pattern to a relational algebra query is the result from applying the following operators over the relation containing the triples:

1. A constant c in a triple pattern implies a selection with an equality formula among c and the attribute where c occurs. For instance, if the constant c occurs in the subject position of the triple pattern then the operator $\sigma_{?\text{s}=c}$ is be applied.

2. A variable ?x occurring twice in a triple implies a selection with an equality among the positions where ?x occurs. For instance, if ?x occurs in the subject and in the object of a triple pattern, then the operator $\sigma_{?\text{s}=?\text{o}}$ is applied.

3. A variable ?x in a triple pattern implies a renaming of the attribute where ?x occurs by ?x. For instance, if variable ?x occurs in the subject position of the triple pattern then the operator $\rho_{?\text{s}/?\text{x}}$ is applied.

4. The variables occurring in a triple pattern are the attributes in the result. For instance, if the variables of the triple pattern are ?x and ?y then $\pi_{?\text{x}\,?\text{y}}$ is applied.

Example 3.4. *The following are translations for triple patterns:*

$$\begin{aligned}
\text{ra}((?\text{x}, p, ?\text{y})) &= \rho_{?\text{s}/?\text{x}}(\rho_{?\text{o}/Y}(\pi_{?\text{s},?\text{o}}(\sigma_{?\text{p}=p}(\text{triples})))),\\
\text{ra}((?\text{x}, p, o)) &= \rho_{?\text{s}/?\text{x}}(\pi_{?\text{s}}(\sigma_{?\text{p}=p\wedge?\text{o}=o}(\text{triples}))),\\
\text{ra}((?\text{x}, p, ?\text{x})) &= \rho_{?\text{s}/?\text{x}}(\pi_{?\text{s}}(\sigma_{?\text{p}=p\wedge?\text{o}=?\text{s}}(\text{triples}))).
\end{aligned}$$

Let Q, Q_1, Q_2 be queries in the relational fragment of SPARQL, $\mathcal{X}, \mathcal{Y}$ be sets of variables where $\mathcal{X} \subseteq \text{dom}(Q)$ and $\mathcal{Y} = \text{type}(Q_1) \cap \text{type}(Q_2)$, and φ be a Boolean combination of equality formulas. The translation for the rest of operators of the relational fragment of

SPARQL is defined recursively by the following equalities:

$$\begin{aligned}
\text{ra}(\text{SELECT}\,\mathcal{X}\,\text{WHERE}\,Q) &= \pi_{\mathcal{X}}(\text{ra}(Q)),\\
\text{ra}(Q\,\text{FILTER}\,\varphi) &= \sigma_{\varphi}(\text{ra}(Q)),\\
\text{ra}(Q_1\,\text{AND}\,Q_2) &= \text{ra}(Q_1) \bowtie \text{ra}(Q_2),\\
\text{ra}(Q_1\,\text{UNION}\,Q_2) &= \text{ra}(Q_1) \cup \text{ra}(Q_2),\\
\text{ra}(Q_1\,\text{MINUS}\,Q_2) &= \text{ra}(Q_1) \bowtie (\pi_{\mathcal{Y}}(\text{ra}(Q_1)) - \pi_{\mathcal{Y}}(\text{ra}(Q_2))).
\end{aligned}$$

Now we are ready to present the correspondence among SPARQL queries in the relational fragment and the V-tables relational algebra.

Theorem 3.1. *For every SPARQL query Q in the relational fragment and RDF graph G, it holds that $[\![Q]\!]_G = [\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$.*

Proof. It can be shown by induction on the structure of the query Q.

If Q is a triple pattern then, by simple inspection in all possible combinations for constants and variables, it holds that $[\![Q]\!]_G = [\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$.

If Q has the form (SELECT $\mathcal{X}$ WHERE P) then, by induction, μ' is a solution of $[\![P]\!]_G$ if and only if μ' is a solution of $[\![\text{ra}(P)]\!]_{\text{vdb}(G)}$. By definition, for every solution μ of $[\![Q]\!]_G$ there is a solution μ' of $[\![P]\!]_G$ such that $\mu'|_{\mathcal{X}} = \mu$. Hence, μ is a solution of $[\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$. Since this argument can be used in the contrary direction, we conclude that $[\![Q]\!]_G = [\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$.

If Q has the form (VALUES $\mathcal{X}\Omega$) then, by definition $[\![Q]\!]_G = [\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$.

If Q has the form (P FILTER φ) then each solution μ of $[\![Q]\!]_G$ is also a solution of query P. By induction, $[\![P]\!]_G = [\![\text{ra}(P)]\!]_{\text{vdb}(G)}$. Since μ satisfies the condition φ, it holds that μ is a solution of $[\![\text{ra}(P)]\!]_{\text{vdb}(G)}$. Since this argument can be used in the contrary direction, we conclude that $[\![Q]\!]_G = [\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$.

If Q has the form Q_1 AND Q_2 then for each solution μ of $[\![Q]\!]_G$ there exists two solutions μ_1 of $[\![Q_1]\!]_G$ and μ_2 of $[\![Q_2]\!]_G$ such that $\mu = \mu_1 \backsim \mu_2$. By induction, μ_1 is a solution of $[\![\text{ra}(Q_1)]\!]_{\text{vdb}\,G}$ and μ_2 is a solution of $[\![\text{ra}(Q_2)]\!]_{\text{vdb}\,G}$. By definition, μ is a solution of $[\![\text{ra}(Q)]\!]_{\text{vdb}\,G}$. Since this argument can be used in the contrary direction, we conclude that $[\![Q]\!]_G = [\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$.

If Q has the form Q_1 MINUS Q_2 then for each solution μ of $[\![Q]\!]_G$, μ is also a solution of $[\![Q_1]\!]_G$ and there not exist a solution μ_2 of $[\![Q_2]\!]_G$ such that $\mu_1 \sim \mu_2$. By induction, μ is also a solution of $[\![\text{ra}(Q_1)]\!]_{\text{vdb}(G)}$ and there not exist a solution μ_2 of $[\![\text{ra}(Q_2)]\!]_{\text{vdb}(G)}$ such that $\mu_1 \sim \mu_2$. Let $\mathcal{Y}$ be the set $\text{dom}(Q_1) \cap \text{dom}(Q_2)$. Then, $\mu|_{\mathcal{Y}}$ is a solution of the relational algebra expression $\pi_{\mathcal{Y}}(\text{ra}(Q_1)) - \pi_{\mathcal{Y}}(\text{ra}(Q_2))$. Since $\mu \sim \mu_{\mathcal{Y}}$, it holds that μ is a solution of the relational algebra expression $\text{Q}_1 \bowtie (\pi_{\mathcal{Y}}(\text{ra}(Q_1)) - \pi_{\mathcal{Y}}(\text{ra}(Q_2)))$. By definition, μ is then a solution of $[\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$. Since this argument can be used in the contrary direction, we conclude that $[\![Q]\!]_G = [\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$.

If Q has the form Q_1 UNION Q_2 then the set of solutions of $[\![Q]\!]_G$ is the union of the sets $[\![Q_1]\!]_G$ and $[\![Q_2]\!]_G$. By induction, this set is the union of the respective sets $[\![\text{ra}(Q_1)]\!]_{\text{vdb}(G)}$ and $[\![\text{ra}(Q_2)]\!]_{\text{vdb}(G)}$. By definition, this set is $[\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$. Hence, it holds that $[\![Q]\!]_G = [\![\text{ra}(Q)]\!]_{\text{vdb}(G)}$. □

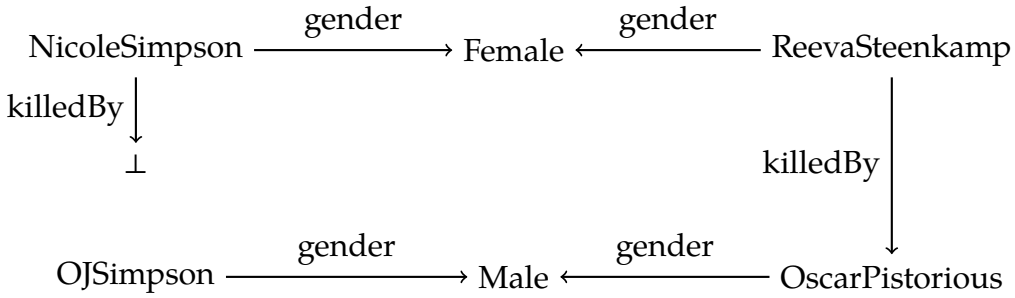
In the remainder of this section we will use relational algebra queries to query RDF graphs viewed as V-databases. That is, given a relational algebra query Q we write $[\![Q]\!]_G$ to denote the result of evaluating $[\![Q]\!]_{\text{vdb}(G)}$.

3.3 Approximating certain answers in SPARQL

So far, we proposed a semantics where the RDF graphs can be understood in terms of V-databases. Recall that a V-database is a database composed by tables with marked nulls called V-tables (see Section 3.2). Also, we identify a SPARQL fragment where queries can be codified as relational algebra queries over V-databases. Our motivation to show this correspondence among the RDF/SPARQL and V-tables/relational algebra is to use the techniques to compute certain answers from the relational algebra to SPARQL. In this section we define conceptually the problem and the techniques that we evaluate experimentally in the next section.

3.3.1 Certain answers with null values

Consider the RDF graph of Example 3.1 that we replicate here:



and the SPARQL query

(SELECT ?x WHERE ((?x, killedBy, ?y) MINUS (?y, gender, Male)))

that asks for people whose killer is not a male. The answer Nicole Simpson—given according to the standard SPARQL semantics—is uncertain because we do not known if the killer was not male; the blank node $\perp$ could refer to a male in the data.

Formally, a tuple μ is said to be a *certain answer* of a relational algebra query Q over a V-database D if for every database $D' \in \text{repr}(D)$ it holds that $\mu \in [\![Q]\!]_{D'}$. The translation of the notion of certain answer from relational algebra to the relational fragment of SPARQL under the naive semantics of RDF graphs is straightforward. Given a SPARQL query Q in the relational fragment and a graph G, a solution mapping μ is a *certain answer* of Q on G if for every graph G' without blank nodes where $G' \models_{\text{naive}} G$ it holds that $\mu \in [\![Q]\!]_{G'}$.

Lipski [50] generalizes the notion of certain answers to incorporate answers having null values. Libkin [48] called *certain answers with nulls* to this notion. Given a V-database D and a relational algebra query Q over the schema of D, a tuple μ is a *certain-answer with nulls* of Q on D if and only if for every map $h : D \to D'$ where $D' \in \text{repr}(D)$, it holds that $h(\mu)$ is a *certain answer* of Q on D'. This notion is defined in the context of SPARQL as follows.

Definition 3.7 (Certain-answers with nulls [48])**.** *Let G be an RDF graph and Q be a SPARQL algebra query in the relational fragment. Then, the set of certain-answers with nulls, denoted* $\text{cert}(Q, G)$*, is the set of mappings defined as follows:*

$$\text{cert}(Q, G) = \{\mu \mid h(\mu) \in [\![Q]\!]_{G'} \text{ for every map } h : G \to G' \text{ where } h(G) = G' \text{ and } G' \text{ has no blank nodes}\}.$$

Example 3.5. *Consider the graph G of Example 3.1 (we replicate above to help the reader) and the SPARQL query* $(\text{NicoleSimpson}, \text{killedBy}, ?\text{x})$ *that asks for the person* ?x *who killed Nicole Simpson. The mapping $\mu = \{?\text{x} \mapsto \bot\}$ is not a certain-answer of Q on G because μ has the blank node $\bot$. On the other hand, μ is a certain-answer with nulls of Q on G because for every map $h : G \to G'$ it holds that the mapping $\mu = \{?\text{x} \mapsto h(\bot)\}$ is a certain-answer of Q on G.*

To understand the difference between these two notions, let $\mathcal{A} = (R, P, E, I)$ be an interpretation of G, and let $h : G \to G'$ be a map where G' has no blank nodes and $\mathcal{A} \models G'$. The logic statement $\mathcal{A} \models \text{killedBy}(\text{NicoleSimpson}, \bot)$ does not hold because, by definition, $\bot \notin \text{dom}(I)$. On the other hand, the statement $\mathcal{A} \models \text{killedBy}(\text{NicoleSimpson}, h(\bot))$ holds because $(\text{NicoleSimpson}, \text{killedBy}, h(\bot)) \in G'$*. The notion of certain-answer with nulls therefore captures some correct information that the notion of certain-answer losses.*

The notion of *possible-answer* is a complementary notion to that of certain-answer. An answer μ is said possible for a query Q on a V-database D if $\mu \in [\![Q]\!]_{D'}$ for some V-database $D' \in \text{repr}(D)$. Unlike certain-answers, admitting null values in possible answers do not capture more meaning. To see this, consider the query Q that ask for the killer of Nicole Simpson. For any person a the mapping $\mu = \{?\text{x} \mapsto a\}$ is a possible answer of Q. The mapping $\mu = \{?\text{x} \mapsto \bot\}$ does not add information to set of possible-answers since it is subsumed by all of them.

We next provide the definition of possible-answers in terms of SPARQL queries and RDF databases.

Definition 3.8 (Possible-answers)**.** *Let G be an RDF graph, and Q be a SPARQL query on the relational fragment. The set of possible-answers, denoted* $\text{poss}(Q, G)$*, is the set of mappings defined as follows:*

$$\text{poss}(Q, G) = \{\mu \mid \mu \in [\![Q]\!]_{G'} \text{ for some map } h : G \to G' \text{ where } h(G) = G' \text{ and } G' \text{ has no blank nodes}\}.$$

3.3.2 Under- and over-approximations

Computing certain- and possible-answers implies a significant computational overhead. If we consider relational algebra, the data complexity of the standard query evaluation problem is AC^0; on the other hand, the analogous complexity with unknown values is coNP-hard [2]. Since in practice we cannot return the exact set of certain- and possible-answers we require approximated evaluation procedures. In this section we formalize what we mean by *approximated*.

If we ask for the certain-answers of a query, we face two types of error. The first type, called *false positive*, consists in returning an answer that is not certain. This error is equivalent to

make a false claim. The second type, called *false negative*, consists in not returning a certain-answer. This error is equivalent to make an omission. False positives are considered worse errors than false negatives [49]. Evaluation procedures with no false positives are said to be *sound*, while evaluation procedures with no false negatives are said to be *complete*. Since sound and complete evaluation procedures for certain answers are intractable, it is widely accepted to define approximations that are always sound but sometimes incomplete [63]. On the other hand, if we ask for the possible-answers of a query, false negatives are considered worse errors than false positives. An approximation of possible-answers must therefore include all possible answers, and sometimes some answers that are impossible.

The question that arises is how good is an approximation. Regarding the accuracy of the approximations for certain-answers, in one extreme we can simply choose an evaluation procedure returning no answers. This choice is trivially correct and without computational cost. However, it gives no useful information. On the other extreme, we can define a procedure that is complete, but it implies in some cases an intractable complexity. In general, we want an evaluation procedure that returns "many" or "enough" answers with the computational resources we have. However, there is no consensus in what we exactly mean by "many" or "enough" answers.

An *evaluation procedure* is a function Eval that receives a SPARQL query Q in the relational fragment, and an RDF graph G, and returns a possibly infinite set Ω of mappings. The functions cert and poss (see definitions 3.7 and 3.8), that compute the certain- and possible-answers, are therefore evaluation procedures. We next present the notions of *under-* and *over-approximation* among evaluation procedures. These notions allow a comparison of evaluation procedures in terms of their accuracy with respect to cert and poss.

The notion of under-approximation is the simplest of the two notions because it expresses the order given by the relation of subset among answer sets. On the other hand, the notion of over-approximation is based on the order of *informativeness* among mappings with the same domain. The notion of informativeness depends on the idea, called *null-substitution principle* by Codd [20], that constants provide more information than null values, so the substitution of a null value by a constant implies more information.

Definition 3.9 (Informativeness of mappings)**.** *Given two mappings μ_1 and μ_2 where $\mathrm{dom}(\mu_1) = \mathrm{dom}(\mu_2)$, the mapping μ_2 is said to be as informative as the mapping μ_1, denoted $\mu_1 \leq \mu_2$, if there exists a map h such that $h(\mu_1) = \mu_2$.*

Grahne [28] shows that the informativeness order of tuples with nulls defines a lattice. The extension of his lattice to tuples with marked nulls is straightforward. We present such lattice using the notation of SPARQL mappings in Appendix A.

Now we present the notions of over- and under-approximation. They define when an approximation is correct in terms of soundness and completeness, and present an order for the informativeness of solutions.

Definition 3.10 (Under- and over-approximation)**.** *Let Eval_1 and Eval_2 be evaluation procedures.*

1. *We say that Eval_1 under-approximates Eval_2, denoted $\mathrm{Eval}_1 \leq_{\mathrm{under}} \mathrm{Eval}_2$, if and only if for every graph G and every SPARQL query Q in the relational fragment, it holds that $\mathrm{Eval}_1(Q, G) \subseteq \mathrm{Eval}_2(Q, G)$.*

2. *We say that* Eval_1 *over-approximates* Eval_2*, denoted* $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_2$*, if and only if for every graph* G*, and every SPARQL query* Q *in the relational fragment, it holds that for every mapping* $\mu_2 \in \text{Eval}_2(Q, G)$ *there exists a mapping* $\mu_1 \in \text{Eval}_1(Q, G)$ *such that* $\mu_1 \leq \mu_2$*.*

We write $\mathcal{E}_{\text{under}}$ *and* $\mathcal{E}_{\text{over}}$ *to denote the set of evaluations procedures over SPARQL queries in the relational fragment that under-approximate* cert *and over-approximate* poss*, respectively.*

Example 3.6. *Let* G *be an RDF graph,* Q *be a SPARQL query in the relational fragment, and* Eval_1 *and* Eval_2 *be two evaluation procedures where:*

$$\text{Eval}_1(Q, G) = \{\{?x \mapsto \bot_1\}\},$$
$$\text{Eval}_2(Q, G) = \{\{?x \mapsto \bot_1\}, \{?x \mapsto \bot_2\}\}.$$

Considering only this query and this graph, it seems that Eval_1 *is strictly less accurate than* Eval_2 *regarding to under-approximations (i.e.,* $\text{Eval}_1 \leq_{\text{under}} \text{Eval}_2$*, but not* $\text{Eval}_2 \leq_{\text{under}} \text{Eval}_1$*). On the other hand* Eval_1 *and* Eval_2 *over-approximate each other (i.e.,* $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_2$ *and* $\text{Eval}_2 \leq_{\text{over}} \text{Eval}_1$*). Intuitively, the mapping* $\{?x \mapsto \bot_2\}$ *adds more information when considering the partial order defined by under-approximations, but not when considering over-approximations.*

Lemma 3.5. $(\mathcal{E}_{\text{under}}, \leq_{\text{under}})$ *is a lattice.*

Proof. To show that $(\mathcal{E}_{\text{under}}, \leq_{\text{under}})$ is a lattice we have to show that $(\mathcal{E}_{\text{under}}, \leq_{\text{under}})$ is a partial order and there exists an algebra $(\mathcal{E}_{\text{under}}, \wedge, \vee)$ where for any pair of evaluation procedures Eval_1 and Eval_2 in $\mathcal{E}_{\text{under}}$ it holds that:

1. $\text{Eval}_1 \vee \text{Eval}_2$—the lattice join—is the minimal evaluation procedure in $\mathcal{E}_{\text{under}}$ such that Eval_1 and Eval_2 under-approximate $\text{Eval}_1 \vee \text{Eval}_2$, and
2. $\text{Eval}_1 \wedge \text{Eval}_2$—the lattice meet—is the maximal evaluation procedure in $\mathcal{E}_{\text{under}}$ such that Eval_1 and Eval_2 are under-approximated by $\text{Eval}_1 \wedge \text{Eval}_2$.

$(\mathcal{E}_{\text{under}}, \leq_{\text{under}})$ is a partial order because the reflexivity, antisymmetry, and transitivity of $(\mathcal{E}_{\text{under}}, \leq_{\text{under}})$ follow from the same properties in $\subseteq$.

Throughout this proof, we will assume that G is an arbitrary graph, and that Q is an SPARQL query in the relational fragment.

The lattice join: Given two evaluation procedures Eval_1 and Eval_2 in $\mathcal{E}_{\text{under}}$, let the join be defined as follows:

$$(\text{Eval}_1 \vee \text{Eval}_2)(Q, G) = \text{Eval}_1(Q, G) \cup \text{Eval}_2(Q, G).$$

Since $\text{Eval}_1 \in \mathcal{E}_{\text{under}}$ and $\text{Eval}_2 \in \mathcal{E}_{\text{under}}$, it holds that:

$$\text{Eval}_1(Q, G) \subseteq \text{cert}(Q, G),$$
$$\text{Eval}_2(Q, G) \subseteq \text{cert}(Q, G).$$

It holds then that:

$$\text{Eval}_1(Q, G) \cup \text{Eval}_2(Q, G) \subseteq \text{cert}(Q, G).$$

Thus, it holds that $\text{Eval}_1 \vee \text{Eval}_2 \in \mathcal{E}_{\text{under}}$. Hence, $\vee$ is closed in $\mathcal{E}_{\text{under}}$.

By simple inspection, it holds that Eval_1 and Eval_2 under-approximate $\text{Eval}_1 \vee \text{Eval}_2$.

Let Eval_1, Eval_2 and Eval_3 be evaluation procedures in $\mathcal{E}_{\text{under}}$ such that Eval_1 and Eval_2 under-approximate Eval_3. By definition, it holds that:

$$\text{Eval}_1(Q, G) \subseteq \text{Eval}_3(Q, G),$$
$$\text{Eval}_2(Q, G) \subseteq \text{Eval}_3(Q, G).$$

It holds then that $\text{Eval}_1(Q, G) \cup \text{Eval}_2(Q, G) \subseteq \text{Eval}_3(Q, G)$. Thus, it holds that $\text{Eval}_1 \vee \text{Eval}_2$ under-approximates Eval_3. Hence, $\text{Eval}_1 \vee \text{Eval}_2$ is the minimal evaluation procedure in $\mathcal{E}_{\text{under}}$ that is under-approximated by Eval_1 and Eval_2.

The lattice meet: Given two evaluation procedures Eval_1 and Eval_2 in $\mathcal{E}_{\text{under}}$, let the meet be defined as follows:

$$(\text{Eval}_1 \wedge \text{Eval}_2)(Q, G) = \text{Eval}_1(Q, G) \cap \text{Eval}_2(Q, G).$$

Let Eval_1, Eval_2 and Eval_3 be evaluation procedures in $\mathcal{E}_{\text{under}}$ such that Eval_1 and Eval_2 are under-approximated by Eval_3. By definition, it holds that:

$$\text{Eval}_1(Q, G) \supseteq \text{Eval}_3(Q, G),$$
$$\text{Eval}_2(Q, G) \supseteq \text{Eval}_3(Q, G).$$

It holds then that $\text{Eval}_1(Q, G) \cap \text{Eval}_2(Q, G) \supseteq \text{Eval}_3(Q, G)$. Thus, it holds that $\text{Eval}_3(Q, G)$ under-approximates $\text{Eval}_1 \wedge \text{Eval}_2$. Hence, $\text{Eval}_1 \wedge \text{Eval}_2$ is the maximal evaluation procedure in $\mathcal{E}_{\text{under}}$ that under-approximates Eval_1 and Eval_2. □

The top of the lattice $(\mathcal{E}_{\text{under}}, \leq_{\text{under}})$ is cert and the bottom is the evaluation $\text{Eval}_\varnothing$ where $\text{Eval}_\varnothing(Q, G) = \varnothing$ for every graph G and query Q over the schema of G.

We next show that one can obtain an interesting lattice from over-approximations. However, unlike under-approximations, over-approximations do not define a partial order over evaluation procedures. To see this, consider the evaluation procedures Eval_1 and Eval_2 where $\text{Eval}_1(Q, G) = \text{Eval}_2(Q, G)$ for every graph G and SPARQL query Q in the relational fragment, except a query Q' and graph G' where $\text{Eval}_1(Q', G') = \{(\bot_1)\}$ and $\text{Eval}_2(Q', G') = \{(\bot_2)\}$. It is not difficult to see that $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_2$ and $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_2$. However, $\text{Eval}_1 \neq \text{Eval}_2$. Hence, since over-approximations do not define antysimetric relations, they do not define partial orders. Theorem 3.2 below states that over-approximations define preorders. We will show later that over-approximations define a lattice, but over equivalence classes of evaluation procedures.

Theorem 3.2. $(\mathcal{E}_{\text{over}}, \leq_{\text{over}})$ *is a preorder.*

Proof. Let Eval be an evaluation procedure in $\mathcal{E}_{\text{over}}$. Given a graph G and a SPARQL query Q in the relational fragment, $\text{Eval} \leq_{\text{over}} \text{Eval}$ because for each mapping $\mu \in \text{Eval}(Q, G)$ it holds that $\mu \leq \mu$. Hence, $(\mathcal{E}_{\text{over}}, \leq_{\text{over}})$ is reflexive.

Let Eval_1, Eval_2, and Eval_3 be evaluation procedures in $\mathcal{E}_{\text{over}}$ where $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_2$ and $\text{Eval}_2 \leq_{\text{over}} \text{Eval}_3$. Let G be a graph, Q be a relational algebra query for graphs, and μ be a mapping where $\mu_3 \in \text{Eval}_3(Q, G)$. Since $\text{Eval}_2 \leq_{\text{over}} \text{Eval}_3$, there exists a mapping $\mu_2 \in \text{Eval}_2(Q, G)$ such that $\mu_2 \leq \mu_3$. Since $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_2$, there exists a mapping $\mu_1 \in \text{Eval}_1(Q, G)$ such that $\mu_1 \leq \mu_2$. Since $\leq$ is transitive (see Appendix A), it holds that $\mu_1 \leq \mu_3$. Thus, $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_3$. Hence, $(\mathcal{E}_{\text{over}}, \leq_{\text{over}})$ is transitive.

Since $(\mathcal{E}_{\text{over}}, \leq_{\text{over}})$ is reflexive and transitive, it is a preorder. □

We require to consider equivalence classes over $\mathcal{E}_{\text{over}}$ to build a partial order from the preorder $(\mathcal{E}_{\text{over}}, \leq_{\text{over}})$. We next define the equivalence relation as usual.

Definition 3.11 (Over-approximation equivalence classes)**.** *Given two evaluation procedures* Eval_1 *and* Eval_2 *in* $\mathcal{E}_{\text{over}}$*, we say that they are equivalent, denoted* $\text{Eval}_1 \cong \text{Eval}_2$*, if and only if* $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_2$ *and* $\text{Eval}_1 \leq_{\text{over}} \text{Eval}_2$.

We next present the lattice for over-approximations.

Theorem 3.3. $(\mathcal{E}_{\text{over}}/\cong, \leq_{\text{over}})$ *is a lattice.*

Proof. To show that $(\mathcal{E}_{\text{over}}/\cong, \leq_{\text{over}})$ is a lattice we have to show that $(\mathcal{E}_{\text{over}}/\cong, \leq_{\text{over}})$ is a partial order and that for any pair $\text{Eval}_1, \text{Eval}_2 \in \mathcal{E}_{\text{over}}/\cong$ they have a join and a meet:

1. $[\text{Eval}_1]_\cong \vee [\text{Eval}_2]_\cong$—the lattice join—is the minimal equivalence class of evaluation procedures in $\mathcal{E}_{\text{over}}/\cong$ such that $[\text{Eval}_1]_\cong$ and $[\text{Eval}_2]_\cong$ over-approximate $[\text{Eval}_1]_\cong \vee [\text{Eval}_2]_\cong$, and
2. $[\text{Eval}_1]_\cong \wedge [\text{Eval}_2]_\cong$—the lattice meet—is the maximal equivalence class of evaluation procedures in $\mathcal{E}_{\text{over}}/\cong$ such that $[\text{Eval}_1]_\cong$ and $[\text{Eval}_2]_\cong$ are over-approximated by $[\text{Eval}_1]_\cong \wedge [\text{Eval}_2]_\cong$.

The lattice $(\mathcal{E}_{\text{over}}/\cong, \leq_{\text{over}})$ is a partial order because the reflexivity and the transitivity follow from the fact that $\mathcal{E}_{\text{over}}/, \leq_{\text{over}}$ is a preorder; and the antysimetry follows from the definition of equivalence.

Throughout this proof, we will assume that G is an arbitrary graph, and Q is a SPARQL query in the relational fragment.

The join: Given two evaluation procedures Eval_1 and Eval_2 in $\mathcal{E}_{\text{over}}$, let the join of them, denoted $\text{Eval}_1 \vee \text{Eval}_2$, be defined as follows:

$$(\text{Eval}_1 \vee \text{Eval}_2)(Q, G) = \{\mu_1 \vee \mu_2 \mid \mu_1 \in \text{Eval}_1(Q, G) \text{ and } \mu_2 \in \text{Eval}_2(Q, G)\},$$

where $\mu_1 \vee \mu_2$ denotes the least informative mapping that is more informative than μ_1 and μ_2 (see Appendix A).

Given two equivalence classes of evaluation procedures $[\text{Eval}_1]_\cong$ and $[\text{Eval}_2]_\cong$ in $\mathcal{E}_{\text{over}}/\cong$, let the join of them, denoted $[\text{Eval}_1]_\cong \vee [\text{Eval}_2]_\cong$, be defined as follows:

$$[\text{Eval}_1]_\cong \vee [\text{Eval}_2]_\cong = [\text{Eval}_1 \vee \text{Eval}_2]_\cong.$$

To see that the equivalence class of evaluation procedures $[\text{Eval}_1 \vee \text{Eval}_2]_{\cong}$ is over-approximated by $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$, let μ be a mapping in $(\text{Eval}_1 \vee \text{Eval}_2)(Q, G)$. By construction, there exists mappings μ_1 and μ_2 where $\mu_1 \in \text{Eval}_1(Q, G)$, $\mu_2 \in \text{Eval}_2(Q, G)$, and $\mu = \mu_1 \vee \mu_2$. Since $\mu_1 \leq \mu$ and $\mu_2 \leq \mu$, it holds that Eval_1 and Eval_2 over-approximate $\text{Eval}_1 \vee \text{Eval}_2$. Hence, also $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$ over-approximate $[\text{Eval}_1]_{\cong} \vee [\text{Eval}_2]_{\cong}$.

To see that $[\text{Eval}_1]_{\cong} \vee [\text{Eval}_2]_{\cong}$ is the minimal equivalence class of evaluation procedures that is over-approximated by $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$, let $[\text{Eval}_3]_{\cong}$ be an equivalence class of evaluation procedures in $\mathcal{E}_{\text{over}}/\cong$ such that $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$ over-approximate $[\text{Eval}_3]_{\cong}$, and μ be a mapping in $\text{Eval}_3(Q, G)$. Then, there exists mappings $\mu_1 \in \text{Eval}_1(Q, G)$ and $\mu_2 \in \text{Eval}_2(Q, G)$ such that $\mu_1 \leq \mu$ and $\mu_2 \leq \mu$. By definition, $\mu_1 \vee \mu_2 \in (\text{Eval}_1 \vee \text{Eval}_2)(Q, G)$. Because $\mu_1 \vee \mu_2$ is the least informative mapping that is more informative than μ_1 and μ_2 (see Appendix A), it holds that $\mu_1 \vee \mu_2 \leq \mu_3$. Hence, $\text{Eval}_1 \vee \text{Eval}_2 \leq_{\text{over}} \text{Eval}_3$, so $[\text{Eval}_1]_{\cong} \vee [\text{Eval}_2]_{\cong} \leq_{\text{over}} [\text{Eval}_3]_{\cong}$.

The fact that the operator $\vee$ is closed in the set $\mathcal{E}_{\text{over}}/\cong$ follows from that $[\text{Eval}_1]_{\cong} \vee [\text{Eval}_2]_{\cong}$ is the minimal equivalence class of evaluation procedures that is over-approximated by $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$. The equivalence classes of evaluation procedures $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$ over-approximate $[\text{poss}]_{\cong}$ because they are in $\mathcal{E}_{\text{over}}/\cong$. Since $[\text{Eval}_1]_{\cong} \vee [\text{Eval}_2]_{\cong}$ is minimal, it follows that $[\text{Eval}_1]_{\cong} \vee [\text{Eval}_2]_{\cong}$ over-approximates $[\text{poss}]_{\cong}$. Hence, $[\text{Eval}_1]_{\cong} \vee [\text{Eval}_2]_{\cong} \in \mathcal{E}_{\text{over}}/\cong$.

The meet: Given two evaluation procedures Eval_1 and Eval_2 in $\mathcal{E}_{\text{over}}$, let the meet of Eval_1 and Eval_2, denoted $\text{Eval}_1 \wedge \text{Eval}_2$, be defined as follows:

$$(\text{Eval}_1 \wedge \text{Eval}_2)(Q, G) = \text{Eval}_1(Q, G) \cup \text{Eval}_2(Q, G).$$

Then, the meet of the equivalence classes of evaluation procedures $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$, denoted $[\text{Eval}_1]_{\cong} \wedge [\text{Eval}_2]_{\cong}$, is defined as follows:

$$[\text{Eval}_1]_{\cong} \wedge [\text{Eval}_2]_{\cong} = [\text{Eval}_1 \wedge \text{Eval}_2]_{\cong}.$$

The fact that the equivalence classes of evaluation procedures $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$ over-approximate $[\text{Eval}_1]_{\cong} \wedge [\text{Eval}_2]_{\cong}$ is direct from the definition. Without loss of generality, let μ be a mapping in $\text{Eval}_1(Q, G)$. By definition, mapping μ is also in $(\text{Eval}_1 \wedge \text{Eval}_2)(Q, G)$, thus $[\text{Eval}_1]_{\cong} \wedge [\text{Eval}_1]_{\cong} \leq_{\text{over}} [\text{Eval}_1]_{\cong}$.

To see that $[\text{Eval}_1]_{\cong} \wedge [\text{Eval}_2]_{\cong}$ is the maximal equivalence class of evaluation procedures that over-approximates $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$, let $[\text{Eval}_3]_{\cong}$ be an equivalence class of evaluation procedures in $\mathcal{E}_{\text{over}}/\cong$ that over-approximates $[\text{Eval}_1]_{\cong}$ and $[\text{Eval}_2]_{\cong}$, and μ be a mapping in $(\text{Eval}_1 \wedge \text{Eval}_2)(Q, G)$. Then, by definition, tuple $\mu \in \text{Eval}_1(Q, G) \cup \text{Eval}_2(Q, G)$. Without loss of generality assume that $\mu \in \text{Eval}_1(Q, G)$. Since $\text{Eval}_3 \leq_{\text{over}} \text{Eval}_1$, there exists a mapping $\mu' \in \text{Eval}_3(Q, G)$ such that $\mu' \leq \mu$. Hence, $[\text{Eval}_1]_{\cong} \wedge [\text{Eval}_2]_{\cong} \leq_{\text{over}} [\text{Eval}_3]_{\cong}$.

The fact that the operator $\wedge$ is closed on $\mathcal{E}_{\text{over}}/\cong$ follows from the transitivity of $\leq_{\text{over}}$. Indeed, $[\text{Eval}_1]_{\cong} \wedge [\text{Eval}_2]_{\cong} \leq_{\text{over}} [\text{Eval}_1]_{\cong}$ and $[\text{Eval}_1]_{\cong} \leq_{\text{over}} [\text{poss}]_{\cong}$ implies $[\text{Eval}_1]_{\cong} \wedge [\text{Eval}_2]_{\cong} \leq_{\text{over}} [\text{poss}]_{\cong}$. □

To conclude the discussion about over-approximations, we will recall the expected properties of their lattice: it has a top and a bottom element.

The top element of $(\mathcal{E}_{\text{over}}/\cong, \leq_{\text{over}})$ is the equivalence class of the evaluation poss, because by definition, every element in $\mathcal{E}_{\text{over}}$ over-approximates poss.

The bottom element of $(\mathcal{E}_{\text{over}}/\cong, \leq_{\text{over}})$ is the equivalence class of the evaluation procedure $\text{Eval}_\varnothing$ where, for every graph G and SPARQL query Q in the relational fragment, it holds that:

1. If $\text{poss}(Q, G)$ is empty, then $\text{Eval}_\varnothing(Q, G) = \varnothing$.
2. otherwise, if $\text{poss}(Q, G)$ has an tuple μ, then $\text{Eval}_\varnothing(Q, G) = \{\mu'\}$ where μ' is the mapping that associates a different fresh blank node to each variable $?x \in \text{dom}(Q)$.

In fact, for every evaluation procedure $\text{Eval}' \in \mathcal{E}_{\text{over}}$ it holds that $\text{Eval}_\varnothing \leq_{\text{over}} \text{Eval}'$. The relation is trivial if $\text{poss}(Q, G)$ is empty. Otherwise, if $\text{poss}(Q, G)$ has an answer μ, the relation follows from that $[\mu']_\cong$ is the bottom of the lattice $(\Omega_\mu/\cong, \leq)$, where Ω_μ is the set of mappings with domain $\text{dom}(Q)$ that are less informative than μ, and $\cong$ is the equivalence relation for the informativeness preorder $\leq$ among mappings (see Appendix A).

Lattices of incomplete data were studied by Grahne [28]. In this section we extended his work by introducing two lattices for approximations. An approximation is more accurate when it is closer to the top of its corresponding lattice. Since the complexity of the standard query evaluation of the top of these lattices is coNP-hardness [2], and the evaluation of the bottom are trivial, defining a practical evaluation procedure in between implies a trade-off between complexity and accuracy. The problem is that there is no objective measure about "how good" is an approximation. The study of the lattice of approximations may serve for this purpose.

3.3.3 Approximating relational algebra queries

In this section we present an under- and an over-approximation for relational algebra queries based on the combination of both types of approximation. The combination of under- and over-approximations was proposed by Biskup [15] when defining tables having maybe tuples, that we will call *M-tables* (for "Maybe") in what follows. Intuitively, an M-table R is a table where each tuple $\mu \in R$ is marked either as *sure* or *maybe*. Biskup's tables also admit nulls to represent values that are unknown. Biskup considered non-marked nulls (i.e., Codd nulls [20]), however in this chapter we assume that M-tables admit marked nulls because they allow expressing blank nodes. As we show herein, this extension is straightforward.

Formally, an *M-table* R over a set of attributes $\mathcal{X}$, is defined by a pair of V-tables, denoted (R_2, R_1) where $R_1 \subseteq R_2$ and $\text{dom}(R_1) = \text{dom}(R_2) = \mathcal{X}$. We write $\text{sure}(R)$ and $\text{poss}(R)$ to denote the first and second components of an m-table R, respectively. We write $\text{dom}(R)$ to denote the set of attributes $\mathcal{X}$ on which the M-table R is defined. Intuitively, $\text{sure}(R)$ denotes the sure tuples of R and $\text{poss}(R)$ to denote all tuples of R (i.e., the tuples that are marked as sure or maybe). Tuples in $\text{poss}(R) \setminus \text{sure}(R)$ are said *maybe-tuples* of R. An *M-database* over a database schema (T, dom) is a function that associates each table name $r \in T$ to a M-table over $\text{dom}(r)$, denoted r^D.

Example 3.7. *Consider that the employees of a company are registered in the following M-table:*

$$\left[\begin{array}{cc} ?person & ?office \\ \hline Alice & 221 \\ Bob & \bot_1 \\ \hdashline Carl & \bot_1 \\ Denis & \bot_2 \end{array}\right]. \tag{3.1}$$

The dashed line divides the sure-tuples (above) from the maybe-tuples (below). The sure tuples are interpreted as in V-tables. The first tuple states that Alice works in office 221, and the second tuple that bob works in an office, but that the office is unknown. The first maybe-tuple states that we do not known if Carl works in the company, but in the case Carl works there, it works in the same office as Bob. The second maybe-tuple states that we do not know if Denis works in the company nor the room where Denis works. Moreover, we know that no more people than the specified as possible by the table (i.e., Alice, Bob, Carl and Denis) work in the company.

Definition 3.12 (Semantics of M-databases)**.** *A regular table R' (i.e., with neither null values nor maybe-tuples) is said to be an instance of an M-table R if there exists a V-table R'' where* $\text{sure}(R) \subseteq R'' \subseteq \text{poss}(R)$*, and there exists a map $h : R'' \to R'$, and* $\text{dom}(R) = \text{dom}(R') = \text{dom}(R'')$*.*

An M-database D' is said to be an instance of an M-database D over a database schema (T, dom) *if D' is an M-table over* (T, dom) *and for every table name $r \in T$ the table $r^{D'}$ is an instance of r^D.*

The semantics of an M-database D is given by the set, denoted $\text{repr}(D)$*, of instances of D. We say that an interpretation $\mathcal{A}$ interprets a M-database D if $\mathcal{A} \models D'$ for a database $D' \in \text{repr}(D)$. We say that an M-database D_1 entails an M-database D_2, denoted $D_1 \models D_2$, if each interpretation of D_1 is also an interpretation of D_2.*

Observe that M-tables generalize V-tables. In fact, a V-table R has the same interpretations that the M-table R' where $R = \text{sure}(R') = \text{poss}(R')$.

Biskup [15] proposed an algebra for M-tables that includes the usual operators of relational algebra in the named perspective: selection, projection, renaming, join, union, and difference. The main property of this algebra is, roughly speaking, that it defines an under- and an over-approximation of the evaluation procedures that compute the certain answers with null values and the possible answers evaluation procedures, respectively. We next present an example to illustrated the idea behind the Biskup algebra and how this property holds.

Example 3.8. *Consider the following M-tables:*

$$R_1 = \left[\begin{array}{cc} ?x & ?y \\ \hline a & b \\ \bot & c \\ \hdashline \end{array}\right], \qquad R_2 = \left[\begin{array}{cc} ?x & ?z \\ \hline a & d \\ e & f \\ \hdashline \end{array}\right], \qquad R_3 = \left[\begin{array}{c} ?z \\ \hline d \\ \hdashline \end{array}\right],$$

where the dashed line separate the sure tuples (above) from the maybe tuples (below). Let Q be the relational algebra query $(R_1 \bowtie R_2) \bowtie R_3$. We start with the first join:

$$R_1 \bowtie R_2 = \left[\begin{array}{ccc} ?x & ?y & ?z \\ \hline a & b & d \\ \hdashline a & c & d \\ e & c & f \end{array}\right].$$

The sure tuple is obtained of joining the tuples where no null values are involved. The two maybe tuples are obtained of instancing the null value as a or c, respectively. These tuples resulting of instancing null values are marked as maybe because they are not true in all instances.

We next evaluate second join:

$$(R_1 \bowtie R_2) \bowtie R_3 = \left[\begin{array}{ccc} ?x & ?y & ?z \\ \hline a & b & d \\ \hdashline a & c & d \end{array}\right].$$

In this example the set $\{(a, b, d)\}$ marked as sure coincides exactly with the certain answers of the query. Similarly, the set $\{(a, b, d), (a, c, d)\}$ coincides with the possible answers of the query. In general there are no coincidences, but approximations.

Gugliardo and Libkin [33] proposed an approximation for certain answers that follows the Biskup idea of combining the answers that are sure with the answers that are possible. They use this approximation to show experimentally the feasibility of computing certain answers for SQL with no much overhead.

Greco et al. [30, 29] showed that the approximation by Guagliardo and Libkin can be improved in terms of accuracy by using conditional tables. Conditional tables are essentially V-tables extended with formulas restricting the interpretations of the null values. For instance, a conditional table can be extended with the formula $\bot_1 \neq \bot_2$ that excludes the interpretations of the table where $\bot_1$ and $\bot_2$ are interpreted as the same element of the domain. The use of conditional tables is out the scope of this thesis.

In this section we present an under- and an over-approximation that is based in the Biskup [15] algebra of M-tables and in the approximation of certain answers by Guagliardo and Libkin [33]. We present these approximations in terms of relational algebra queries of RDF graphs. Recall that every SPARQL query in the relational fragment can be codified as a relational algebra query (see Theorem 3.1) so we can study approximations in this SPARQL fragment by using relational algebra queries. For simplicity, for now on, given a relation algebra query Q and an RDF graph G, we write $[\![Q]\!]_G$ to denote $[\![Q]\!]_{\mathrm{vdb}(G)}$ (recall that $\mathrm{vdb}(G)$ is the graph G codified as a V-table).

The advantage of the Guagliardo an Libkin [33] approximation is that it consists in rewriting a SQL query Q into a SQL query Q' that approximates Q. The relational algebra does not provide an operator to distinguish blanks from constants. Hence to rewrite a relational algebra query, we need a built-in predicate in the form isBlank(?x) in the target algebra, which evaluates to true for a mapping μ if $\mu(?x) \in \mathbf{B}$, or false otherwise. In the remainder of this section we will present two translations of a relational algebra query Q into relational algebra extended with the operator isBlank, denoted $(\cdot)_{\mathrm{under}}$ and $(\cdot)_{\mathrm{over}}$.

Approximated selection

The translation of the under-approximation was developed by Guagliardo and Libkin [33].

Definition 3.13 (Under- and over-approximated filter-conditions [33])**.** *Let φ be a filter-condition whose atoms are equality atoms. Then, the under-approximated (φ_{under}) and the over-approximated (φ_{over}) filter-conditions for φ are defined recursively as follows:*

$$
\begin{aligned}
(?x = ?y)_{\text{under}} &= (?x = ?y),\\
(?x = a)_{\text{under}} &= (?x = a),\\
(?x \neq ?y)_{\text{under}} &= (?x \neq ?y) \wedge \neg\, \text{isBlank}(?x) \wedge \neg\, \text{isBlank}(?y),\\
(?x \neq a)_{\text{under}} &= (?x \neq a) \wedge \neg\, \text{isBlank}(?x),\\
(\psi_1 \vee \psi_2)_{\text{under}} &= (\psi_1)_{\text{under}} \vee (\psi_2)_{\text{under}},\\
(\psi_1 \wedge \psi_2)_{\text{under}} &= (\psi_1)_{\text{under}} \wedge (\psi_2)_{\text{under}},\\
(?x = ?y)_{\text{over}} &= (?x = ?y) \vee \text{isBlank}(?x) \vee \text{isBlank}(?y),\\
(?x = a)_{\text{over}} &= (?x = a) \vee \text{isBlank}(?x),\\
(?x \neq ?y)_{\text{over}} &= (?x \neq ?y),\\
(?x \neq a)_{\text{over}} &= (?x \neq a),\\
(\psi_1 \vee \psi_2)_{\text{over}} &= (\psi_1)_{\text{over}} \vee (\psi_2)_{\text{over}},\\
(\psi_1 \wedge \psi_2)_{\text{over}} &= (\psi_1)_{\text{over}} \wedge (\psi_2)_{\text{over}},
\end{aligned}
$$

where a is a constant, $?x$ and $?y$ are variables, and ψ_1 and ψ_2 are filter-conditions.

According to Guagliardo and Libkin, given a relational algebra query Q and a filter-condition φ with only equalities as atoms, the result of $[\![\sigma_{\varphi_{\text{under}}}(Q)]\!]_G$ under-approximates $\text{cert}(\sigma_\varphi(Q), G)$, for every graph G. Likewise, the result of $[\![\sigma_{\varphi_{\text{over}}}(Q)]\!]_G$ over-approximates $\text{poss}(\sigma_\varphi(Q), G)$. This result justifies the definition of the approximation of the under- and over-approximations as follows.

Definition 3.14 (Approximated translation of the select)**.** *Let Q be a relational algebra query and φ be a filter-condition with only equities as atoms. Then, the under- and over- approximated translations of $\sigma_\varphi(Q)$ are defined as follows:*

$$
\begin{aligned}
(\sigma_\varphi(Q))_{\text{under}} &= \sigma_{\varphi_{\text{under}}}(Q_{\text{under}}),\\
(\sigma_\varphi(Q))_{\text{over}} &= \sigma_{\varphi_{\text{over}}}(Q_{\text{over}}).
\end{aligned}
$$

Approximated join

Based on the ideas of Biskup [15], and Guagliardo and Libkin [33] the approximation of the join operator can be done by expressing them in term of equalities. This translation process is described in the following example.

Example 3.9. *Let $R(?x, ?y)$ and $S(?y, ?z)$ be two V-tables. Since variable $?y$ is the only common attribute between these tables, it holds that:*

$$R \bowtie S \equiv \pi_{?x,?y,?z}(\sigma_{?y=?y_2}(R \bowtie S[?y/?y_2])).$$

Keep in mind that $S[?y/?y_2]$ denotes the renaming in table S of attribute $?y$ as $?y_2$. Since tables R and $S[?y/?y_2]$ have no common attributes, the evaluation of query $R \bowtie S[?y/?y_2]$ is simply a cross

product, where there are no issues when joining unknown values. In fact, the issue is moved to the filter-condition. Since we already have a translation for the filter condition, an over-approximation of $R \bowtie S$ is defined as follows:

$$(R \bowtie S)_{\text{over}} = \pi_{?x,?y,?z}(\sigma_{(?y=?y_2)_{\text{over}}}(R \bowtie S[?y/?y_2])). \tag{3.2}$$

The over-approximation in (3.2) is correct, but it can be improved. To see this, consider the case when ?y *is a blank node $\perp$ in a tuple of R and* $?y_1$ *is a constant c in a tuple of S. In this case the approximation in (3.2) produces a tuple where* ?y *is set to the blank node $\perp$. However, it would be more accurate to return a tuple where* ?y *is set to the constant c, since the join assumed that $\perp = c$.*

To compute the most accurate value from two attributes $?y_1$ and $?y_2$ of the previous example in a simple form, we adapt the SPARQL clause BIND that computes a set of predefined functions **F** to the relational algebra (see Definition 2.8).

Definition 3.15 (Relational algebra BIND operator)**.** *If Q is a relational algebra query over RDF graphs, $f : (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \{\varnothing\})^n \to (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ is a function in* **F***, and* $?y, ?x_1, \ldots, ?x_n$ *are variables such that* ?y *does not occur in Q nor in* $?x_1, \ldots, ?x_n$*, then* $\text{BIND}_{?y:=f(?x_1,\ldots,?x_n)}(Q)$ *is a relational algebra query whose semantics is defined for a graph G as follows:*

$$[\![\text{BIND}_{?y:=f(?x_1,\ldots,?x_n)}(Q)]\!]_G = \{\mu \cup \{?y \mapsto f(\mu(?x_1), \ldots, \mu(?x_n))\} \mid \mu \in [\![Q]\!]_G\}.$$

Observe that Definition 3.15 is restricted to do not have unbound values in the image. This restriction ensures that this operation is closed (i.e., the answers are V-tables).

Definition 3.16 (If-functions)**.** *Let $\varphi(\bar{?x})$ be a filter-condition whose variables are indicated in the parenthesis, f and g be two functions in* **F***, and $\bar{Y}$ and $\bar{Z}$ be two lists of variables whose respective lengths are the arities of f and g. Then, an if-function, denoted* $\text{IF}(\varphi(\bar{?x}), f(\bar{Y}), g(\bar{Z}))$*, is the function that requires instancing the variables $\bar{?x} \cup \bar{Y} \cup \bar{Z}$ to be evaluated. The if-function returns the value computed by function f if the evaluation of φ is true, and the value computed by a function g, if the evaluation of φ is false.*

Example 3.10. *Consider the relation*

$$R = \left[\begin{array}{cc} A & B \\ \hline 1 & 1 \\ 2 & 1 \end{array}\right].$$

Then:

$$\text{BIND}_{C=\text{IF}(A=B,A+B,A-B)}(R) = \left[\begin{array}{ccc} A & B & C \\ \hline 1 & 1 & 2 \\ 2 & 1 & 1 \end{array}\right].$$

Using the BIND operator the over-approximation in (3.2) can be improved as follows:

$$(R \bowtie S)_{\text{over}} = \pi_{?x,?y,?z}(\text{BIND}_{?y:=\text{IF}(\text{isBlank}(?y_1),\ ?y_1,\ ?y_2)}(\sigma_{(?y_1=?y_2)_{\text{over}}}(R[?y/?y_1] \bowtie S[?y/?y_2]))). \tag{3.3}$$

The same reasoning can be applied to get to an under-approximation for the join. However, since $(?y_1 = ?y_2)_{\text{under}} = (?y_1 = ?y_2)$ we can conclude simply that:

$$(R \bowtie S)_{\text{under}} = R \bowtie S. \tag{3.4}$$

The translation described in equations 3.3 and 3.4 is generalized as follows:

Definition 3.17 (Approximated translation of the join)**.** *Let Q be a relational algebra query of the form $R \bowtie S$ where $\text{dom}(R) \cap \text{dom}(S) = \mathcal{X}$, $r : \mathcal{X} \to \mathbf{V}$ and $s : \mathcal{X} \to \mathbf{V}$ be functions that assign a fresh variable to each variable in $\mathcal{X}$. Then, the under- and over-approximations of the join are defined as follows:*

$(R \bowtie S)_{\text{under}} = R_{\text{under}} \bowtie S_{\text{under}}$,
$(R \bowtie S)_{\text{over}} = \pi_{\text{dom}(R)\cup\text{dom}(S)}(\text{BIND}_{\bar{X}:=\text{IF}(r(\bar{X}),s(\bar{X}))}(\sigma_{(\bigwedge_{?x\in\bar{X}} r(?x)=s(?x))_{\text{over}}}((r(R)_{\text{over}} \bowtie (s(S))_{\text{over}})))$,

where $r(R)$ and $s(S)$ denote the results of replacing in R and S each variable $?x \in ?x$ by $r(?x)$ and $s(?x)$, respectively, and $\text{BIND}_{\bar{X}:=\text{IF}(r(\bar{X}),s(\bar{X}))}$ denotes the algebraic operation that assigns to each variable $?x \in \mathcal{X}$ the value $\text{IF}(\text{isBlank}(r(?x)), s(?x), r(?x))$ (i.e., the most informative value given to variable $?x$).

The question that arises is whether we can avoid using the operator BIND to approximate the join operator. The following lemma answers this question.

Lemma 3.6. *Let f and g be two functions such that BIND with these functions is expressible in relational algebra. Then, the operator BIND with if-functions of the form $\text{IF}(\varphi(\bar{X}), f(\bar{Y}), g(\bar{Z}))$ is expressible in relational algebra.*

Proof. Let Q be a relational algebra query, and φ be a selection formula whose free variables are $\bar{X}$. Let $\bar{X} \cup \bar{Y} \cup \bar{Z} \subseteq \text{dom}(Q)$ and V be a variable such that $V \notin \text{dom}(Q)$. Then, the following equivalence shows that the operator BIND with this function is expressible in the relational algebra:

$$\text{BIND}_{V:=\text{IF}(\varphi(\bar{X}),f(\bar{Y}),g(\bar{Z}))}(Q) \equiv \text{BIND}_{V:=f(\bar{Y})}(\sigma_{\varphi(Q)(\bar{X})}) \cup \text{BIND}_{V:=g(\bar{Z})}(\sigma_{\neg\varphi(\bar{X})}(Q)).$$

Since the BIND operator with functions f and g do not add expressive power to the algebra, we conclude that the operator BIND with function $\text{IF}(\varphi(\bar{X}), f(\bar{Y}), g(\bar{Z}))$ does not add expressive power to the algebra. □

Approximated difference

Likewise the approximated join, the translation for the approximated difference can be done by rewriting the checking of the tuples to be joined as a separated filter condition. This process is described in the following example.

Example 3.11. *Let $R(?x)$ and $S(?x)$ be two tables. Then, it holds that:*

$$R - S \equiv R - \Pi_{?x}(\sigma_{?x=?x'}(R \bowtie S[?x/?x']). \tag{3.5}$$

Since tables R and $S[?x/?x']$ have no common attributes, the evaluation of $R \bowtie S[?x/?x']$ is simply a cross product. Likewise the case of the join, we moved the issues to the selection operator. The under-approximation of this query can thus be computed as follows:

$$R - S \equiv R - \Pi_{?x}(\sigma_{(?x=?x')_{\text{over}}}(R \bowtie S[?x/?x']). \tag{3.6}$$

Intuitively, we are dropping from R all elements of Q that are possibly joinable to an element of S. Because we want to compute the elements that are "possible" joinable, we use the over-approximation of the filter-condition.

The same reasoning can be applied to get to an over-approximation for the difference. However, since $(?y_1 = ?y_2)_{\text{under}} = (?y_1 = ?y_2)$ *we can conclude simply that:*

$$(R - S)_{\text{over}} = R - S. \tag{3.7}$$

The translation described in the previous example is generalized as follows:

Definition 3.18 (Approximated translation of the difference)**.** *Let Q be a relational algebra query of the form* $R - S$ *where* $\text{dom}(R) = \text{dom}(S) = \mathcal{X}$, $s : \mathcal{X} \to \mathbf{V}$ *be a function that assigns a fresh variable to each variable in* $\mathcal{X}$*. Then, the under- and over-approximations of the join are defined as follows:*

$$\begin{aligned}(R - S)_{\text{under}} &= R_{\text{under}} - \pi_{\mathcal{X}}(\sigma_{\bigwedge_{?x \in \mathcal{X}}(?x = s(?x))_{\text{over}}}((R)_{\text{under}} \bowtie (s(S))_{\text{over}}),\\ (R - S)_{\text{over}} &= R_{\text{over}} - S_{\text{under}},\end{aligned}$$

where $s(S)$ *denotes the results of replacing in S each attribute* $?x \in \mathcal{X}$ *by* $s(?x)$*.*

The translation to approximate certain- and possible-answers

The following definition describes the summary of the translation for queries Q in relational algebra into queries (with the isBlank Boolean built-in atom as extension) that approximate Q.

Definition 3.19 (Approximations for relational algebra queries queries)**.** *The translations of a relational algebra query Q to its under- and over-approximation, denoted respectively* Q_{under} *and* Q_{over} *are defined recursively as follows. Let* Q, Q_1 *and* Q_2 *be relational algebra queries,* $r : \text{dom}(Q_2) \cap \text{dom}(Q_2) \to \mathbf{V}$ *and* $s : \text{dom}(Q_2) \cap \text{dom}(Q_2) \to \mathbf{V}$ *be functions that assign each variable* $?x \in \text{dom}(Q_2) \cap \text{dom}(Q_2)$ *to a fresh variable, and* $r(Q_1)$ *and* $s(Q_2)$ *denote the results of replacing in* Q_1 *and* Q_2 *each variable* $?x \in \text{dom}(Q_2) \cap \text{dom}(Q_2)$ *by* $r(?x)$ *and* $s(?x)$*, respectively.*

$$\begin{aligned}(p)_{\text{under}} &= p \quad \textit{(if } p \textit{ is a predicate)}\\ (R)_{\text{under}} &= R \quad \textit{(if } R \textit{ is a v-table)}\\ (\sigma_{\varphi}(Q))_{\text{under}} &= \sigma_{\varphi_{\text{under}}}(Q_{\text{under}})\\ (\pi_{\bar{X}}(Q'))_{\text{under}} &= \pi_{\bar{X}}(Q'_{\text{under}})\\ (\rho_{?x/?y}(Q'))_{\text{under}} &= \rho_{?x/?y}(Q'_{\text{under}})\\ (Q_1 \cup Q_2)_{\text{under}} &= (Q_1)_{\text{under}} \cup (Q_2)_{\text{under}}\\ (Q_1 \bowtie Q_2)_{\text{under}} &= (Q_1)_{\text{under}} \bowtie (Q_2)_{\text{under}}\\ (Q_1 - Q_2)_{\text{under}} &= (Q_1)_{\text{under}} - \pi_{\bar{X}}(\sigma_{\bigwedge_{?x \in \bar{X}}(?x = s(?x))_{\text{over}}}((Q_1)_{\text{under}} \bowtie (s(Q_2))_{\text{over}})\\ &\qquad \textit{(where } \bar{X} = \text{dom}(Q_1) = \text{dom}(Q_2)\textit{)}\\ (p)_{\text{over}} &= p \quad \textit{(if } p \textit{ is a predicate)}\\ (R)_{\text{over}} &= R \quad \textit{(if } R \textit{ is a v-table)}\\ (\sigma_{\varphi}(Q))_{\text{over}} &= \sigma_{\varphi_{\text{over}}}(Q_{\text{over}})\end{aligned}$$

$$
\begin{aligned}
(\pi_{\bar{X}}(Q'))_{\text{over}} &= \pi_{\bar{X}}(Q'_{\text{over}}) \\
(\rho_{?x/?y}(Q'))_{\text{over}} &= \rho_{?x/?y}(Q'_{\text{over}}) \\
(Q_1 \cup Q_2)_{\text{over}} &= (Q_1)_{\text{over}} \cup (Q_2)_{\text{over}} \\
(Q_1 \bowtie Q_2)_{\text{over}} &= \pi_{\bar{Y}}(\text{BIND}_{\bar{X}:=\text{IF}(r(\bar{X}),s(\bar{X}))}(\sigma_{(\bigwedge_{?x\in\bar{X}} r(?x)=s(?x))_{\text{over}}}((r(Q_1)_{\text{over}} \bowtie (s(Q_2))_{\text{over}}))) \\
&\qquad \textit{(where } \bar{X} = \text{dom}(Q_1) \cap \text{dom}(Q_2) \textit{ and } \bar{Y} = \text{dom}(Q_1) \cup \text{dom}(Q_2)\textit{)} \\
(Q_1 - Q_2)_{\text{over}} &= (Q_1)_{\text{over}} - (Q_2)_{\text{under}}.
\end{aligned}
$$

Theorem 3.4. *Let* $\text{Eval}_{\text{under}}$ *and* $\text{Eval}_{\text{over}}$ *be the evaluation procedures defined as follows:*

$$
\begin{aligned}
\text{Eval}_{\text{under}}(Q, G) &= [\![Q_{\text{under}}]\!]_G, \\
\text{Eval}_{\text{over}}(Q, G) &= [\![Q_{\text{over}}]\!]_G.
\end{aligned}
$$

Then, $\text{Eval}_{\text{under}}$ *and* $\text{Eval}_{\text{over}}$ *under- and over-approximate the evaluation procedures* cert *and* poss*, respectively.*

Proof. It can be shown by induction on the structure of the query. The proof is straightforward and a proof sketch is provided by Guagliardo and Libkin [33]. □

3.3.4 SPARQL rewriting strategies

We now explore alternatives in SPARQL to express the rewriting of Definition 3.19. We will later study these alternative strategies in terms of the their performance over SPARQL query engines.

To simplify the notations we introduce some notions used by Libkin [49]. We first define the notion of *unification*, which joins tuples with unknown values. We say that μ_1 and μ_2 *unify*, denoted $\mu_1 \Uparrow \mu_2$, if and only if for every common attribute ?x that they share, it holds that $\mu_1(?x) = \mu_2(?x)$ or $\mu_1(?x) \in \mathbf{B}$ or $\mu_2(?x) \in \mathbf{B}$; in other words, $\mu_1 \Uparrow \mu_2$ holds if and only if there is a valuation v such that $v(\mu_1(?x)) = v(\mu_2(?x))$ for every common attribute ?x.

Definition 3.20 (Over-approximated semi-join and anti-semi-join)**.** *Let* R *and* S *be two* V*-tables. Then, the generalized operators semi-join* ($\ltimes_{\Uparrow}$) *and anti-semi-join* ($\overline{\ltimes}_{\Uparrow}$) *are defined as follows:*

$$
\begin{aligned}
R \ltimes_{\Uparrow} S &= \{\mu_1 \in R \mid \exists \mu_2 \in S : \mu_1 \Uparrow \mu_2\}, \\
R \,\overline{\ltimes}_{\Uparrow}\, S &= \{\mu_1 \in R \mid \nexists \mu_2 \in S : \mu_1 \Uparrow \mu_2\}.
\end{aligned}
$$

With these operators we can write the under-approximation of the difference in a simpler form by using the following rules:

$$
\begin{aligned}
(Q_1 - Q_2)_{\text{under}} &= (Q_1)_{\text{under}} \,\overline{\ltimes}_{\Uparrow}\, (Q_2)_{\text{over}} \\
&= (Q_1)_{\text{under}} - ((Q_1)_{\text{under}} \ltimes_{\Uparrow} (Q_2)_{\text{over}}), \\
(Q_1 \ltimes_{\Uparrow} Q_2) &= \pi_{\mathcal{X}}(\sigma_{\bigwedge_{?x\in\mathcal{X}} ?x=s(?x)}(Q_1 \bowtie s(Q_2))),
\end{aligned}
$$

where $\mathcal{X} = \text{dom}(Q_1) = \text{dom}(Q_2)$, $s : \mathcal{X} \to \mathbf{V}$ is a function that maps each variable $?x \in \mathcal{X}$ to a fresh variable, and $s(Q_2)$ denotes the result of renaming in Q_2 each variable $?x \in \mathcal{X}$ as $s(?x)$.

The base case in the SPARQL translation is the graph G, which refers to a ternary relation with fixed attributes S, P and O. The basic unit of querying in SPARQL is a *triple pattern*, e.g., $(?x, p, ?y)$. In RDF, the predicate attribute cannot take blanks, and hence we do not need to consider unification on that attribute directly. A *basic graph pattern* Q in SPARQL is a join over triple patterns $T_1 \bowtie \cdots \bowtie T_k$ where each T_i $(1 \leq i \leq k)$ is a triple pattern.

The most complex case to consider is the difference operator $P - Q$, where certain answers are under-approximated by the unification anti-semijoin $P_{\text{under}} \,\overline{\ltimes}_{\Uparrow}\, Q_{\text{over}}$. The direct application of the translation rules produces complex queries that can be rewritten to a "friendlier" form for SPARQL engines, as now described. First, given a difference $P - Q$ we say that the attribute of the difference $?x$ is *correlated* if $?x$ is shared by P and Q. In the following we will assume that $P - Q$ is a difference with at least a correlated variable and that Q is a basic graph pattern.

CNF/DNF rewritings: In the difference $P - Q$, let $Q = T_1 \bowtie T_2$ (a common case). According to Definition 3.19, the base translation evaluating the required over-approximation of Q is then given as

$$(T_1 \bowtie T_2)_{\text{over}} = \pi_{\bar{Y}}(\text{BIND}_{\bar{X}:=\text{IF}(r(\bar{X}),s(\bar{X}))}(\sigma_{(\bigwedge_{?x\in\bar{X}} r(?x)=s(?x))_{\text{over}}}((r(T_1))_{\text{over}} \bowtie (s(T_2))_{\text{over}}))), \quad (3.8)$$

where $\bar{X} = \text{dom}(T_1) \cap \text{dom}(T_2)$, $\bar{Y} = \text{dom}(T_1) \cup \text{dom}(T_2)$, and r and s are the functions used to rename the shared variables as is specified in Definition 3.19.

In order to simplify the notation, we will write respectively U_1 and U_2 instead of $(r(T_1))_{\text{over}}$ and $(s(T_2))_{\text{over}}$, we will write β to denote the operation $\text{BIND}_{\bar{X}:=\text{IF}(r(\bar{X}),s(\bar{X}))}(\cdot)$ that is needed to return constants over blank nodes, and θ to denote the filter condition in (3.8). With this notation, we will then write the approximation in (3.8) as follows:

$$(T_1 \bowtie T_2)_{\text{over}} = \pi_{\text{var}(Q)}(\beta(\sigma_\theta(U_1 \bowtie U_2))), \quad (3.9)$$

These definitions can be extended naturally (but verbosely) to the case where the basic graph pattern Q is $T_1 \bowtie \cdots \bowtie T_k$. To gain intuition on that extension, let us consider the case where $k = 3$, and $\text{var}(T_1) = \{?x, ?y\}$, $\text{var}(T_2) = \{?x, ?y\}$, and $\text{var}(T_3) = \{?x\}$. Also, assume that variable $?x$ has been replaced by the fresh variables $?x_1$, $?x_2$ and $?x_2$ in the respective triple patterns T_1, T_2 and T_3, and variable $?y$ have been replaced by the fresh variables $?y_1$ and $?y_2$ in the respective triple patterns T_1 and T_2. Then the translation of the basic graph pattern Q has the form:

$$(T_1 \bowtie T_2 \bowtie T_3)_{\text{over}} = \pi_{\{?x,?y\}}(\beta_1(\sigma_{\theta_1}(\pi_{\{?x_4,?y\}}(\beta_2(\sigma_{\theta_2}(U_1 \bowtie U_2))) \bowtie U_3))), \quad (3.10)$$

where:

$\theta_1 = (?x_4 = ?x_2 \vee \text{isBlank}(?x_4) \vee \text{isBlank}(?x_2))$,
$\beta_1 = \text{BIND}_{?x:=\text{IF}(?x_4,?x_2)}(\cdot)$,
$\theta_2 = (?x_1 = ?x_2 \vee \text{isBlank}(?x_1) \vee \text{isBlank}(?x_2)) \wedge (?y_1 = ?y_2 \vee \text{isBlank}(?y_1) \vee \text{isBlank}(?y_2))$,
$\beta_2 = \text{BIND}_{?x_4:=\text{IF}(?x_1,?x_2)}(\text{BIND}_{?y:=\text{IF}(?y_1,?y_2)}(\cdot))$.

The translation in (3.10) follows the order in which triple patterns occur. The expression can be arranged to be independent of the order of triple patterns:

$$(T_1 \bowtie T_2 \bowtie T_3)_{\text{over}} = \pi_{?x,?y}(\beta(\sigma_\theta(U_1 \bowtie U_2 \bowtie U_3))), \tag{3.11}$$

where:

$$\begin{aligned}
\theta &= \theta_{?x_1,?x_2} \wedge \theta_{?x_1,?x_3} \wedge \theta_{?x_2,?x_3} \wedge \theta_{?y_1,?y_2},\\
\theta_{?x_1,?x_2} &= (?x_1 = ?x_2 \vee \text{isBlank}(?x_1) \vee \text{isBlank}(?x_2))\\
\theta_{?x_1,?x_3} &= (?x_1 = ?x_3 \vee \text{isBlank}(?x_1) \vee \text{isBlank}(?x_3))\\
\theta_{?x_2,?x_3} &= (?x_2 = ?x_3 \vee \text{isBlank}(?x_2) \vee \text{isBlank}(?x_3))\\
\theta_{?y_1,?y_2} &= (?y_1 = ?y_2 \vee \text{isBlank}(?y_1) \vee \text{isBlank}(?y_2)),\\
\beta &= \text{BIND}_{?x:=\text{IF}(\text{IF}(?x_1,?x_2),?x_3)}(\text{BIND}_{?y:=\text{IF}(?y_1,?y_2)}(\cdot)).
\end{aligned}$$

The translation in (3.11) implies taking the Cartesian product of all triple patterns, filtering by a conjunction of unification conditions, and then selecting constants over blanks. The advantage is that the Cartesian product $U_1 \bowtie U_2 \bowtie U_3$ is a basic graph pattern that can be computed in an SPARQL engine without concerns regarding to incomplete data. It is expected that computing the whole basic graph pattern is better than computing several basic graph patterns of size two, as is done in (3.10). The disadvantage of the translation in (3.11) with respect to the translation in (3.10) is the size of the filter conditions. If a variable ?x occurs in all triple patterns, then in translation (3.10) an unification condition $\theta_{?x_i,?x_{i+1}}$ is required for each pair of consecutive triple patterns T_i, T_{i+1} in the basic graph pattern Q, while in translation (3.11) an unification condition $\theta_{?x_i,?x_j}$ is required for each pair triple patterns T_i, T_j where variable ?x occurs. Thus, the size of filter-conditions goes from linear in (3.10) to quadratic in (3.11). The experimental comparison between the both strategies is out of the scope of this thesis. In the remaining of this section we focus on the strategy exemplified in (3.11).

The filter-condition θ in (3.11) is in conjunctive normal form (CNF): $\theta_1 \wedge \cdots \wedge \theta_n$ where for $1 \leq i \leq n$, each term θ_i is a disjunctive clause. An alternative solution is to rewrite the unification condition to its equivalent disjunctive normal form (DNF) $\phi_1 \vee \cdots \vee \phi_m$ per a standard conversion. The result is potentially exponential in size; though this does not affect the data complexity, it may have a significant effect on performance in practice. However, this DNF conversion leads to further rewritings that may lead to better performance. First, we can express disjunctions using union ($\cup$) or using disjunctive ($\vee$) selection conditions. Second, since this expression falls on the right-hand side of an anti-semijoin operator, we can also express it as a sequence of such operators. Thus, for the translation of $(P - Q)_{\text{under}}$ into $P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} Q_{\text{over}}$, we can consider:

$$P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} Q_{\text{over}} = P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} \sigma_{\bigwedge_{1\leq j\leq m} \theta_j}(Q')\,, \tag{CNF}$$

$$P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} Q_{\text{over}} = P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} \sigma_{\bigvee_{1\leq j\leq m} \phi_j}(Q')\,, \tag{DNF$_1$}$$

$$P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} Q_{\text{over}} = P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} \bigcup_{1\leq j\leq m} \sigma_{\phi_j}(Q')\,, \tag{DNF$_2$}$$

$$P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} Q_{\text{over}} = P_{\text{under}} \mathbin{\overline{\ltimes}}_{\Uparrow} \sigma_{\phi_1}(Q') \cdots \mathbin{\overline{\ltimes}}_{\Uparrow} \sigma_{\phi_m}(Q')\,. \tag{DNF$_3$}$$

where Q' denotes the rewriting of join variables $\bar{X}$ in Q to produce Cartesian products on all join patterns and the subsequent application of $\beta_{\bar{X},?\bar{x}_1,\ldots,?\bar{x}_k}$ to perform unification over those variables. Note, however, that in the cases of DNF_2 and DNF_3, some terms in the disjunction will *not* require a Cartesian product; for example, when we rewrite $P-(T_1 \bowtie T_2)$ to DNF, a disjunctive term on the right of the anti-semijoin will be $(T_1 \bowtie T_2)$ itself (the others will cover the case that join variables in T_1 or T_2 are bound to blanks). This suggests that these options *may* be more efficient despite a *potential* exponential blow-up.

Removing explicit unification: Given a base query of the form $P - Q$, if the join variables of Q do not correlate with P, we do not need to perform unification on them. Consider a query

$$(?\text{x}, p, a) - ((?\text{x}, p, ?\text{y}) \bowtie (?\text{y}, p, b)).$$

This can be rewritten to

$$(?\text{x}, p, a) \mathbin{\overline{\ltimes}}_{\Uparrow} (\text{BIND}_{?\text{y}:=\text{IF(isBlank(?y}_2\text{),?y}_1\text{,?y}_2\text{)}}(\sigma_{\theta_{\Uparrow}}((?\text{x}, p, ?\text{y}_1) \bowtie (?\text{y}_2, p, b)))).$$

However since ?y does not appear on the left of the difference, we can simplify to

$$(?\text{x}, p, a) \mathbin{\overline{\ltimes}}_{\Uparrow} (\sigma_{\theta_{\Uparrow}}((?\text{x}, p, ?\text{y}_1) \bowtie (?\text{y}_2, p, b))).$$

Converting anti-semijoins to difference: Given a base query of the form $P - Q$, we can consider cases where the correlating variable(s) of P and Q may or may not yield blanks on either side. In particular, if Q returns a tuple with blanks for all correlating variables, then the entire difference $P - Q$ must be empty. On the other hand, if P returns a tuple with blanks for all correlating variables and Q is non-empty, then that tuple is removed from P. Finally, in cases where we know that the correlating variable(s) of P and Q cannot yield blanks[4], we can convert the anti-semijoin to standard difference. These ideas yield possible optimizations when we know more about which attributes can yield nulls.

3.4 Evaluation

Our evaluation presents an initial cost–benefit analysis of a certain answer semantics for SPARQL by addressing the following research questions:

RQ1: How do the proposed SPARQL query rewriting strategies compare in terms of performance with the base query, with themselves, with similar results in an SQL setting, and for different SPARQL implementations?

RQ2: Does a certain answer semantics significantly change query results in a real-world setting?

[4]In standard relational settings, this might be if the correlating variables is a primary key of a table, for example. In RDF, we may detect such a case for subjects or objects of a given property that do not give blanks in a given dataset, for example.

3.4.1 Evaluation Setting

In this section, we describe the SPARQL query engines selected, the machines and configurations, as well as the datasets and queries used. Supporting material can be found online:

`https://users.dcc.uchile.cl/~dhernand/revisiting-blanks`

Engines and machines: The query rewriting strategy allows certain answers to be approximated on current SPARQL implementations. We test with two popular engines, with the added benefit of being able to cross-check that the solutions generated by both produce the same answers: Virtuoso (v.7.2.4.2) and Fuseki (v.2.6.0). The machine used is an AMD Opteron Processor 4122, 24GB of RAM, and a single 240 GB Kingston SUV400S SSD disk; Virtuoso is set with `NumberOfBuffers` = 1360000 and with `MaxDirtyBuffers` = 1000000; Fuseki is initialized with 12GB of Java heap space.

Rewriting strategies: We consider various strategies: $[\texttt{B}|\texttt{CNF}|\texttt{DNF}_{1,\ldots,3}]$ where `B` denotes base queries, `CNF` queries in conjunctive normal form, and `DNF` queries in disjunctive normal form; we denote these variations as Γ in the following. By $[\Gamma^{\nexists}|\Gamma^{-}]$ we denote queries that use either `FNE` ($\nexists$) or `MINUS` ($-$) in SPARQL. By $[\Gamma|\Gamma^{*}]$ queries that rather than use `isBlank` to check if a node is blank or not, in case an engine cannot form an index lookup to satisfy such a condition, we also try adding a triple $(\bot, a, \text{Blank})$ to the data for each blank $\bot$ and a triple pattern to check for that triple in the query (denoted Γ^{*}); this does not apply to base queries. In total, this leads to 18 possible combinations. Rather than present results for all, we will highlight certain configurations in the results.

3.4.2 TPC–H experiments

To address RQ1, we follow the experimental design of Guagliardo & Libkin [33] who provide experiments for PostgreSQL using the TPC-H benchmark. Their results compare the performance of approximations for certain answers with respect to four queries with negations. For this, they modified the TPC-H generator to produce nulls in non-primary-key columns with varying probabilities (1–5%) to generate more/less unknown values. They also use scale factors of 1, 3, 6, and 10, corresponding to PostgreSQL databases of size 1GB, 3GB, 6GB and 10GB, respectively. We follow their setting as closely as possible to later facilitate comparison. We wrote a conversion tool (similar to the Direct Mapping [10]) to represent TPC-H data as RDF, and convert the TPC-H SQL queries to SPARQL.

Unifications: We first evaluate the proposed rewriting strategies of unifications in the difference operator for SPARQL. The base format of the queries used is $P - (Q \bowtie R)$ where each P, Q, and R is a triple pattern. We then generate between 1,000 and 10,000 triples matching each triple pattern to perform tests at various scales. For the data matching the join variable on Q and R, we generate blanks with a rate of 1, 2, 4 and 8%. These experiments allow us to estimate the costs of unifications in difference without other query operators interfering.

Figure 3.1 presents performance results. For clarity, we present only a selection of configurations: CNF is equivalent to DNF_1 in this case and we only show the aforementioned

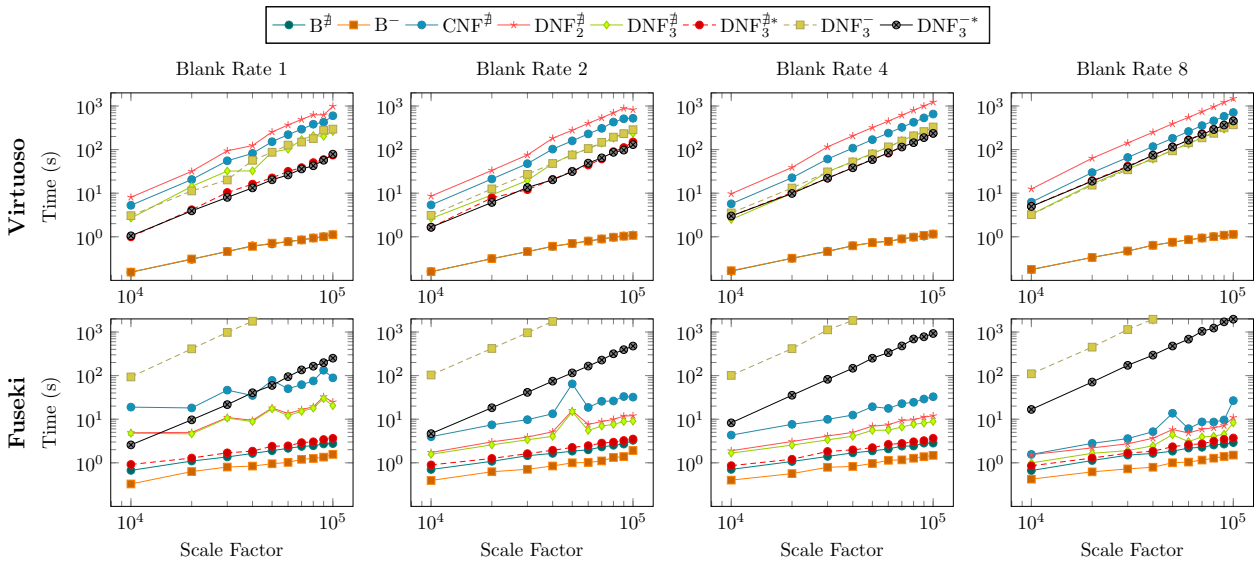


Figure 3.1: Unification results for Virtuoso and Fuseki, varying scales and blank rates

$[\cdot^{\sharp}/\cdot^{*}]$ variations for the base query and DNF_3 (other variations performed analogously). The first row pertains to Virtuoso while the second pertains to Fuseki. All eight sub-plots are presented with log–log axes (base 10) at the same scale permitting direct comparison across plots (comparing horizontally across engines and comparing vertically across blank rates). The y-axis maximum represents a timeout of 25 minutes (reached in some cases by Fuseki).

The base queries whose performances are depicted in Figure 3.1 are the following:

$$B^{-} = (\text{SELECT ?x WHERE}((?x, q, a) \text{ MINUS } ((?x, p_x, ?z) \text{ AND } (?y, p_y, ?z)))),$$
$$B^{\sharp} = (\text{SELECT ?x WHERE}((?x, q, a) \text{ FILTER } (\neg \text{ EXISTS } ((?x, p_x, ?z) \text{ AND } (?y, p_y, ?z))))).$$

The $\text{CNF}^{\sharp}$ variant of these queries is the following:

$$\begin{aligned}
&(\text{SELECT ?x}\\
&\quad \text{WHERE } ((?x, q, a)\\
&\qquad \text{FILTER } (\neg \text{ EXISTS } (((?x, p_x, ?z_1) \text{ AND } (?y, p_y, ?z_2))\\
&\qquad\qquad \text{FILTER } (?z_1 = ?z_2 \vee \text{isBlank}(?z_1) \vee \text{isBlank}(?z_2))))),
\end{aligned}$$

and the $\text{DNF}_3^{\sharp *}$ variant is

$$\begin{aligned}
&(\text{SELECT ?x}\\
&\quad \text{WHERE } ((((?x, q, a)\\
&\qquad \text{FILTER } (\neg \text{ EXISTS } (((?x, p_x, ?z) \text{ AND } (?y, p_y, ?z)))))\\
&\qquad \text{FILTER } (\neg \text{ EXISTS } (((?x, p_x, ?z_1) \text{ AND } (?y, p_y, ?z_2))\\
&\qquad\qquad \text{FILTER } (\text{isBlank}(?z_1)))))\\
&\qquad \text{FILTER } (\neg \text{ EXISTS } (((?x, p_x, ?z_1) \text{ AND } (?y, p_y, ?z_2))\\
&\qquad\qquad \text{FILTER } (\text{isBlank}(?z_2))))).
\end{aligned}$$

(RQ1) The performance of the rewritten queries is (as could be expected) worse than the two base queries for all blank rates, scale factors and engines. In the case of Virtuoso, the base queries generally run in under one second; however, the fastest rewritten queries take at least a second and there is at least an order of magnitude difference between the base query and the fastest rewritten query. Looking at Fuseki, the fastest base query is slower than Virtuoso, but does generally tend to execute within one second (except at the larger scales). However, we see a number of rewriting strategies in the case of Fuseki where the difference is within half-an-order of magnitude of the fastest base case. Otherwise, we see that the choice of strategy is generally not sensitive to the blank rates considered (i.e., lines generally maintain the same ordering across plots), nor is it sensitive to scale (i.e., lines do not generally cross within plots).

Queries: The previous experiments looked at "atomic" unifications. We now run the four TPC-H queries used by Guagliardo and Libkin [33] considering a blank rate of 5%, four scale factors, and two engines. We employ a timeout of 10 minutes. We also choose one base query ($B^{\sharp}$) to be compared against the rewritten queries for approximating certain answers. Fuseki repeatedly times out for these experiments hence here we rather focus on the results of Virtuoso.

In Table 3.1, we present a comparison of the performance results for Virtuoso's fastest rewritten query and the results as presented by Guagliardo and Libkin [33]. More specifically, for a blank rate of 5%, the table shows the range of relative performance between the base query and the best rewritten query execution for that query; since Guagliardo and Libkin do not present absolute runtimes, our comparison is limited to relative performance. Note that due to differences in how SPARQL and SQL treat inequalities over nulls/blanks, Q_3 did not need rewriting for Virtuoso. For Q_2 in PostgreSQL, the actual results drop below the presented numeric precision, returning almost instantaneously for PostgreSQL once a null is found (which confirms that the results are empty).

(RQ1) We see that for Q_1, Virtuoso performs better in relative performance than PostgreSQL, for Q_2 PostgreSQL performs (much) better, for Q_3 there is little difference, while for Q_4 Virtuoso initially performs better than PostgreSQL but then at **SF**$\geq$**3**, Virtuoso begins to throw an error stating that an internal limit of 2,097,151 results has been reached (we could not resolve this). Aside from this latter issue, these results show that Virtuoso with our rewriting strategies is competitive with PostgreSQL under SQL-based rewritings for relative performance between base and rewritten queries. Furthermore, unlike in the previous experiments, we observe that in the case of Q_1 and Q_2, Virtuoso is now sometimes faster for the rewritten queries than the base queries: by removing uncertain answers, the number of intermediary solutions to be processed is reduced.

(RQ2) We observe three of the four base queries returning uncertain answers in SPARQL that do not hold under some valuations: for Q_1, 59% of answers are uncertain; for Q_2, all answers are uncertain; whilst for Q_4, 7% of answers are uncertain; we further highlight that these results are present for a blank rate of 5%. These results suggest that for queries with negation, evaluation under standard SPARQL semantics may in some cases return a significant ratio of uncertain/unsound answers even for modest levels of blanks in the dataset; this is to be expected given that, e.g., even a single blank tuple returned from the right-side of a difference can render all results uncertain (as per Q_2),

Table 3.1: Ranges of average relative performance for scale factor (SF) 1, 3, 6 and 10 on a fixed blank rate of 5%.

Q.	SF=1	SF=3	SF=6	SF=10
		VIRTUOSO		
Q_1	0.95–0.96	0.95–0.96	0.97–0.99	0.94–0.95
Q_2	0.76–1.07	0.73–0.99	0.89–1.06	0.55–0.77
Q_3	1.00–1.00	1.00–1.00	1.00–1.00	1.00–1.00
Q_4	1.55–1.56	error	error	error
		POSTGRESQL (G&L [33])		
Q_1	1.01–1.03	0.99–1.01	0.98–1.01	1.00–1.02
Q_2	0.00–0.00	0.00–0.00	0.00–0.00	0.00–0.00
Q_3	1.01–1.04	1.01–1.04	0.99–1.02	1.00–1.06
Q_4	1.75–1.86	1.80–1.93	2.05–2.25	3.54–3.89

Table 3.2: Numbers of Wikidata use-case queries (from a total of 446) that could be affected by a certain answer semantics

Feature	A	B	C	D
`MINUS`	13	9	9	2
`FILTER NOT EXISTS`	23	15	10	1
`OPTIONAL` w/ `!BOUND`	5	1	0	0
`!=`	7	5	3	0
TOTAL	47	29	21	3

Table 3.3: Query execution times (ms) for the three Wikidata queries with uncertain answers, and ratio of uncertain answers to total answers

	Local (Virtuoso)			**Public** (Blazegraph)			**Uncertain/Total**
	T_1	T_2	T_2/T_1	T_1	T_2	T_2/T_1	
Q_1	144464	142989	0.99	53012	TO	–	20/5487
Q_2	7038	521	0.07	1013	2045	2.02	42/42
Q_3	1266326	1419269	1.12	TO	TO	–	12/27221

3.4.3 Wikidata survey

Since the previous experiments are based on a synthetic benchmark converted from a relational setting, we analyzed the user-contributed SPARQL queries on the Wikidata Query Service, which offers a more native Semantic Web setting.[5] As previously described, Wikidata uses blanks to represent unknown values; our goal now is to determine whether or not a choice of certain answer semantics could impact a current, real-world setting.

(RQ2) We first inspect the 446 queries to see which could potentially be affected by a certain answer semantics. We provide a summary in Table 3.2 according to the query features that may cause uncertain answers, with columns helping to indicate why queries with such features do not give uncertain answers in this context: **A** applies no assumptions, counting queries using the pertinent feature; **B** counts the queries that could still give uncertain answers knowing that Wikidata only uses blanks in a single object position; **C** counts the queries that could give uncertain answers further knowing which predicates have blanks; finally, **D** counts the queries whose solutions do change under certain answers. Hence, we see that 10.5% of the queries contain features that could cause uncertain answers, 6.5% of queries could cause uncertain answers even though Wikidata only uses blanks in a single object position, 4.7% of queries could cause uncertain answers knowing which predicates have blank values and which do not, while finally 0.6% of queries actually return uncertain answers.

We provide some statistics on the three Wikidata queries generating uncertain answers in Table 3.3. First for performance, we run the original query (T_1) and a rewritten version for certain answers (T_2) over both a local Virtuoso index of Wikidata, as well as the public Wikidata Query Service (running Blazegraph). (RQ1) While the performance of the first query is comparable under both standard and certain answer semantics for Virtuoso, the latter times out on Blazegraph. On the other hand, the second query is faster on Virtuoso for certain answer semantics, possibly because it is anticipated that all answers will be discarded. This is not the case for Blazegraph, where the rewritten query takes twice the time. In Virtuoso Q_3 takes slightly longer in the rewritten query. Blazegraph times out in all runs of Q_3. We also look at the ratio of uncertain answers the queries would return. (RQ2) The ratio for Q_1 and Q_3 are relatively low, but on the other hand, for Q_2, the ratio is 100%: all answers are uncertain.

[5] https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples

3.5 Conclusions

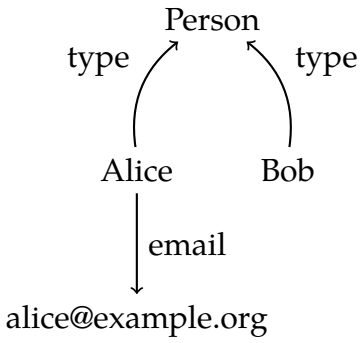
In this chapter, we studied the semantics of SPARQL with respect to RDF graphs that use blank nodes as existential variables encoding unknown values. In particular, we have investigated the feasibility of approximating certain answers in SPARQL, proposing various rewriting strategies. Our initial results suggest that querying for certain/possible answers generally does incur in a significant cost, but that at least for Virtuoso, query answering is still feasible (and in some cases faster than under standard semantics). We showed that the relative performance results for Virtuoso under certain answer semantics are competitive with results published for PostgreSQL. In general, we saw that although some queries are executed faster under certain semantics with current SPARQL implementations, for other queries there can be a significant performance cost. It is important to highlight, however, that experiments were run using off-the-shelf SPARQL implementations; dedicated SPARQL implementations for approximating certain answers may further improve on the performance observed here.

Regarding the question of whether or not offering users a choice of certain answer semantics is important, we analyzed 446 Wikidata queries. Although 10.5% use negation and inequality features that could cause uncertain answers in principle, only 0.6% of these queries return uncertain answers in practice. However, Wikidata only uses unique blanks (acting similarly to unmarked nulls) in the object position. That is, the Wikidata dataset does not offer conclusive practical motivations—considering the pros and cons—to introduce certain answer features to SPARQL engines. It would be interesting to do similar studies for other datasets using existential blanks (we are not immediately aware of such a dataset having a set of SPARQL queries to analyze).

Chapter 4

Unbound values as incomplete data

Let us begin with an example to give the intuition of what unbound values are in SPARQL. Let G be the graph



and let Q be the query

$$((?x, \text{type}, \text{Person}) \text{ OPT } (?x, \text{email}, ?y)).$$

Intuitively, query Q ask for people, and optionally their emails if they are available. The answer to this query is then the following:

$$[\![Q]\!]_G = \left[\begin{array}{cc} ?x & ?y \\ \hline \text{Alice} & \text{alice@example.org} \\ \text{Bob} & - \end{array}\right].$$

The dash in Bob's email denotes an *unbound value*.

As is stated by Lemma 3.4, solutions μ of a SPARQL query Q in the relational fragment hold $\text{dom}(\mu) = \text{dom}(Q)$. But in general, answers of a SPARQL query can have different domains, i.e., $\text{dom}(\mu) \subseteq \text{dom}(Q)$. Intuitively an *unbound value* denotes a wildcard to indicate the absence of a value in a the solution μ and thus, can be considered incomplete data. Unbound values allow to express SPARQL solutions as full fledged rows under a regular relational schema, as is shown in the example above.

In this chapter we study unbound values in SPARQL and their rationale, by mapping the SPARQL algebra to an extension of the relational algebra that supports unbound

values. This extended version of relational algebra, called the *generalized relational algebra*, is the focus of this chapter. It is one where its operators simulate the under and over approximation operators that are implicit in the Biskup M-tables described in Chapter 3. For example, the join will have two versions. Recall that the result of the join of two M-tables defines two sets of tuples, namely the sure and possible tuples. Thus, we can define two alternative versions for the join operator, where one returns the sure tuples and the other the possible tuples. Additionally, it turns out that it is possible to define an SQL fragment that simulates this extended algebra. Thus, by mapping the corresponding fragments of SPARQL and SQL to the generalized relational algebra, we contribute to the understanding of the differences in the design of both query languages.

To study the semantics of unbound values in SPARQL, in this chapter we consider the SPARQL-1 fragment without the BIND operator. We exclude the BIND operator because arbitrary functions can express operations that are not explainable by an interpretation of unbound values as one of the null types we considered in this thesis, namely unknown values, inapplicable values, and no-information nulls.

Organization of this chapter

- In Section 4.1 we briefly review the landscape of null values to help the reader better understand the place that unbound values have in it.
- In Section 4.2 we show the origin of unbound values in SPARQL. They are not present explicitly in the database, but are generated by operators in the query language.
- In Section 4.3 we propose a generalized relational algebra inspired by the Biskup [15] algebra of tables with maybe tuples in order to explain the behavior of null values in SQL and SPARQL. Furthermore, the core of SQL and SPARQL can be seen as different subsets of the generalized relational algebra. Each operator of SQL and SPARQL is mapped to an operator in the generalized relational algebra. We conclude that SQL and SPARQL differ only in two operators, namely the join and the difference.
- We conclude in Section 4.4.

4.1 A brief review of the landscape of null values

As we discussed in the introduction of this thesis, a null value may have different meanings. According to Biskup [14] *"In ANSI [1975], p. IV-28/29, there is a list of 13 'manifestations of null' in stored database relations (the 14th manifestation results from processing nulls)."* In Chapter 3 we studied blank nodes under a semantics where blank nodes denote values that exist but are unknown. In SQL a null value can also denote a value that does not exist, or represents the lack of information about the value. These three semantics can be ordered in terms of informativeness as is depicted in Figure 4.1 below.

We next informally summarize the semantics of these three types of null value. As an example, consider that Alice is applying to a public benefit. Many things are asked by the form, and she has to provide them to the best of her knowledge. However, in some cases her knowledge is not enough. For example, she may be asked for the appraisal of the apartment where she is living. Since she does not know the value, she cannot fill in that

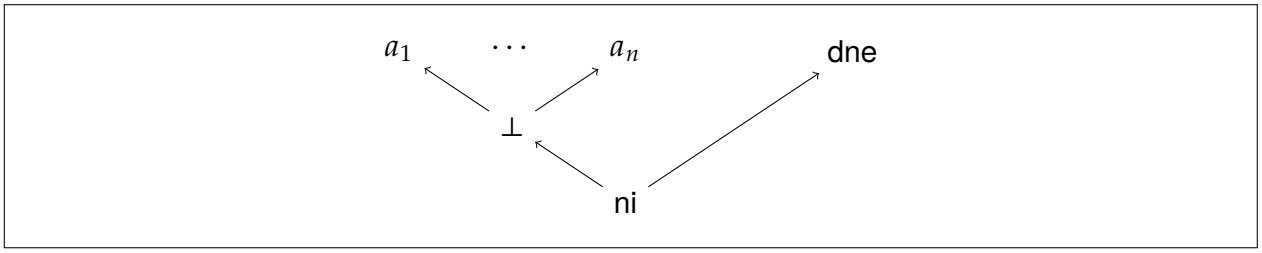


Figure 4.1: Ordering of null values according to Lerat and Lipski [45] and to Libkin [47]. Above is more informative than below. Values $a_1, \ldots, a_n$ are the constants; ni denotes that we do not known if the value exists; $\bot$ denotes that we known that the value exists though is unknown; and dne that the value does not exist.

field. The form asks for her surname, however, in Chile most people have two surnames, one for each parent. But Alice comes from a country where people have a single surname, so one of the required surnames must be left blank.

Unknown values. In the motivation of this section Alice's apartment has an appraisal, but she does not known the value. This type of null value is called *unknown value* or *existential null value*, because it is a value that there exists but is unknown. These are the types of null values that have been studied most. Some studies on unknown values include, among many, those by Codd [20], Biskup [14, 15], Maier [52], Imielinski and Lipski [43], Reiter [63], and Grahne [28].

Inapplicable values. In the motivation of this section Alice's apartment does not have a second surname. The blank in the field for the second surname denotes then the nonexistence or non-applicability of the value. This type of null value is called *inapplicable null*, *non-applicable null*, or *non-existing null*. This type of null value is studied by Vassiliou [68], and Lerat and Lipski [45].

No information nulls. No information nulls are a combination of the unknown and inapplicable value. If a field in a table is left blank without explaining why it was not filled in, then we will not be able to determine if either the value exists though is unknown, or if it is inapplicable. This type of null value is studied by Zaniolo [70], and by Lerat and Lipski [45].

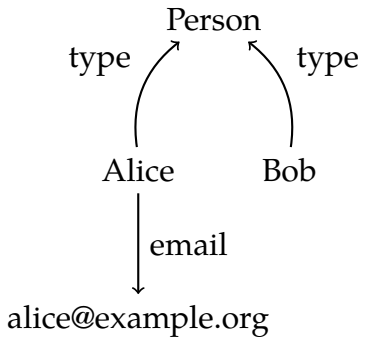
4.2 Where do unbound values came from?

Unbound values are not present in the RDF data model—as is the case of with blank nodes. Instead, they appear when querying RDF graphs with SPARQL. Unbound values are introduced into the language by the UNION of queries of different schemas (in what follows the *outer* UNION), the OPTIONAL operator, the VALUES operator, or the BIND operator. Operators outer UNION and OPTIONAL correspond respectively to the operators *outer union* and *left outer join* of the relational algebra that were studied by Galindo-Legaria [27]. As shown in Chapter 3, we cannot produce unbound values in the SPARQL fragment consisting of the operators FILTER, AND, MINUS, and UNION restricted to queries of the

same schema (in what follows the *regular* UNION). In fact, this fragment can be expressed using standard relational algebra (see Theorem 3.1).

Though the OPTIONAL operator can be formally expressed by a combination of AND, MINUS, and the outer UNION, the unbound values that they produce have a different semantics, depending on what operators produce them. To see this, consider the following example.

Example 4.1. *Let G be the graph at the beginning of this chapter that we show here again to easy reading:*



and let Q be the query

$$((?x, \text{type}, \text{Person}) \text{ OPT } (?x, \text{email}, ?y)).$$

Intuitively, query Q ask for people, and optionally their emails if they are available. The answer to this query is then the following:

$$[\![Q]\!]_G = \left[\begin{array}{cc} ?x & ?y \\ \hline \text{Alice} & \text{alice@example.org} \\ \text{Bob} & - \end{array}\right].$$

The dash in Bob's email denotes an unbound value, and can be interpreted with the meaning that Bob's email is not available.

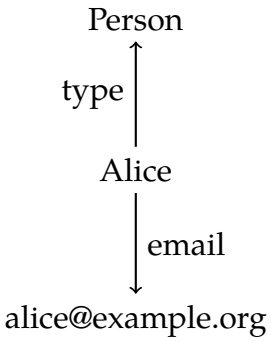
The semantics of the unbound node in Example 4.1 depends on if we either follow the open or the closed world semantics for the RDF graph.

Under the open world semantics, the unbound node means a lack of information about Bob's email. We cannot ensure that Bob has or does not have an email. To see this, consider the graph G' resulting from adding the triple $(\text{Bob}, \text{email}, \text{bob@email})$ to G. In the evaluation of Q over G' the unbound value is filled by a known email. Since G entails G', interpretations of G' are also interpretations of G. Thus, the unbound node can be understood as a no-information null value, because there are interpretations where the value exists and interpretations where the value does not exist.

On the contrary, under the closed world assumption we can infer that Bob does not have an email, so the null value can be understood as an inapplicable null value.

The interpretation of the null values generated by the outer UNION are more involved. The following example presents the complications of the outer UNION regarding the meaning of the unbound values it produces.

Example 4.2. *Consider the graph G depicted as follows:*



Let Q be the query

$$((?x, \text{type}, \text{Person}) \text{ AND } (?x, \text{email}, ?y)) \text{ UNION } (?x, \text{type}, \text{Person}).$$

Intuitively, query Q asks for people and their emails, but also asks for people without considering their emails. The answer to this query is then the following:

$$[\![Q]\!]_G = \left[\begin{array}{ll} ?x & ?y \\ \hline \text{Alice} & \text{alice@example.org} \\ \text{Alice} & - \end{array}\right].$$

In Example 4.2 the semantics of the unbound value is confusing. The two sides of the UNION correspond to queries that have no incomplete data. One can thus argue that the unbound value is added in a contrived way. To continue reasoning about the semantics of this unbound value, let us turn a blind eye on the meaning of the operation UNION and let us address the question of whether the unbound value has the meaning of one of the three types of null values described in the introduction of this section (i.e., unknown-value, inapplicable value, or no-information null value).

If the unbound value means an inapplicable value, then the two tuples in the evaluation output are contradictory. One of them is stating that Alice has an email, while the other that she has not.

If the unbound value means a no-information null value, then we have two options, either a value exists or not. If it does not exist a value, then for the same reasoning as in the previous case, both tuples are contradictory. Thus, if we restrict the result to be consistent, the only option is that the unbound value means an unknown value. However, in this case the information provided by the second tuple is redundant, since it is subsumed by the first tuple. This sort of redundancy has been studied in several works. Zaniolo [70, 71] noticed these redundancies for relations with no-information nulls and defines the notion of *minimal representation* as the result of eliminating each tuple that is subsumed by another one. Galindo-Legaria [27] writes $R!$ to denote the result of removing tuples of a table R that are subsumed by another tuple in R. Arenas and Ugarte [12] define a similar operator in SPARQL, denoted NS, to remove subsumed mappings in a set of mappings. Extending SPARQL with the NS operator allows them to capture syntactically the fragment of weakly-monotone queries.

4.3 Null values in SQL and SPARQL

Recall that the core of both, SQL and SPARQL, can be described using relational algebra when no null values are allowed. The translation from SQL to relational algebra is well known. The translation from the relational SPARQL fragment to relational algebra is given in Section 3.2.2. Table 4.1 summarizes these translations.

Algebra	SQL	SPARQL
$R \cup S$	R `UNION` S	(R UNION S)
$\pi_{\bar{Y}}(R)$	`SELECT` $\bar{Y}$ `FROM` R	(SELECT $\bar{Y}$ WHERE R)
$\rho_{A/B}(R)$	`SELECT` A `AS` B, $\bar{X} \setminus \{A\}$ `FROM` R	(SELECT $((\bar{X} \setminus \{A\}) \cup \{B\})$ WHERE (R BIND(A AS B)))
$\sigma_C(R)$	`SELECT` $\bar{X}$ `FROM` R `WHERE` C	(R FILTER C)
$R \bowtie S$	`SELECT * FROM` R `NATURAL JOIN` S	(R AND S)

$R - S$	`SELECT * FROM` R
	`WHERE NOT EXISTS (`
	`SELECT *`
	`FROM` S`)`
	`WHERE` $R.X_1 = S.X_1$ `AND`
	$\ldots$
	$R.X_n = S.X_n$)
		(R MINUS S)

Table 4.1: Translations between relational algebra and SQL and SPARQL queries with no null values (i.e., with no SQL nulls nor SPARQL unbound values).

In this section we address the question of how relational algebra can be extended with null values to preserve the aforementioned translations of SPARQL and SQL to relational algebra, and thus contribute to the understanding of the design of both query languages. As we already mentioned, we define an algebra, based on the Biskup algebra of M-tables (see Section 3.3.3), where operators are generalized in two alternative forms. Intuitively, one alternative returns the elements that are sure tuples in the Biskup algebra, and the other alternative returns the possible tuples. In this section we show that to express the core of both, SQL and SPARQL, in this generalized algebra it is sufficient choosing one of the generalized operators described to replace the standard operator used for the translation that does not consider null values. For instance, for SQL we replace the join operator by the alternative that returns the sure answers in the translation, and for SPARQL we replace the join operator by the alternative that returns the possible answers. We can therefore describe the join in each query language in terms of certain and possible answers. In the case of SQL, a null value and a constant are not joinable because there exists the possibility that the null represents a value distinct to the constant. On the other hand, in SPARQL an unbound value is always compatible with a constant. In the remainder of this chapter we formalize and study the design of SPARQL unbound values and SQL null values regarding possible and certain answers.

4.3.1 The generalized algebra

A *∅-table* R is a table that admits the symbol $\varnothing$ in cell values to denote nulls. Unlike V-tables (see Section 3.2), nulls in $\varnothing$-tables are unmarked. Formally, the tuples of a $\varnothing$-table are mappings $\mu : \mathrm{dom}(R) \to \mathbf{C} \cup \{\varnothing\}$, called $\varnothing$-mappings, where $\mathbf{C}$ denotes the set of

constants and $\varnothing$ denotes the null value. If ?x $\notin$ dom(R) we say that ?x is unbound in μ. Otherwise, if ?x $\in$ dom(R) and μ(?x) = $\varnothing$, we said that ?x is bound to a null value in μ.

We next define an algebra that generalizes relational algebra for $\varnothing$-tables. We call this algebra *the generalized algebra*. This algebra is based on the Biskup algebra for tables with maybe tuples we briefly referred to in Section 3.3.3.

To define the generalized algebra we need some notions.

Definition 4.1 (Certainly and possibly compatible $\varnothing$-mappings)**.** *Two $\varnothing$-mappings μ_1 and μ_2 are said to be certainly compatible, denoted $\mu_1 \sim_{\text{under}} \mu_2$, if for every common variable* ?x *it holds that μ_1(?x) = μ_2(?x), μ_1(?x) $\neq \varnothing$, and μ_2(?x) $\neq \varnothing$. Similarly, two $\varnothing$-mappings μ_1 and μ_2 are said to be possibly compatible, denoted $\mu_1 \sim_{\text{over}} \mu_2$, if for every common variable* ?x *it holds that μ_1(?x) = μ_2(?x), μ_1(?x) = $\varnothing$, or μ_2(?x) = $\varnothing$.*

Polleres [60] called *bravely-compatible mappings* to possibly compatible mappings, and *strictly-compatible mappings* to certainly compatible mappings. He described a third type of compatible mappings, called *cautiously compatible*, that are out the scope of this thesis. We restrict our study to these forms of compatibility because they resemble the modals *certainly* and *possibly* we used when approximating certain answers.

Intuitively, two $\varnothing$-mappings are certainly compatible if all their instances (instancing the nulls) are compatible. Observe that the instance of a $\varnothing$-mapping is a standard SPARQL mapping, and recall that two mappings μ_1 and μ_2 are said to be compatible, denoted $\mu_1 \sim \mu_2$ if and only if for every common variable ?x holds μ_1(?x) = μ_2(?x). Two $\varnothing$-mappings μ_1 and μ_2 must therefore have no null values in their common variables to be certainly compatible, because null values can be instanced to different values. Similarly, two $\varnothing$-mappings are possibly compatible if there exists an instance that makes them compatible.

The compatibility of $\varnothing$-mappings determines the pairs of mappings that can be joined. The join of two $\varnothing$-mappings has also to be generalized. Since $\mu_1 \sim_{\text{under}} \mu_2$ implies that $\mu_1 \sim_{\text{over}} \mu_2$, it suffices defining the join for $\varnothing$-mappings that are possibly compatible.

Definition 4.2 (Join of compatible $\varnothing$-mappings)**.** *Let μ_1 and μ_2 be two possibly compatible $\varnothing$-mappings. The join of μ_1 and μ_2, denoted $\mu_1 \backsim \mu_2$, is the $\varnothing$-mapping where* dom($\mu_1 \backsim \mu_2$) = dom(μ_1) $\cup$ dom(μ_2) *and for every variable* ?x $\in$ dom($\mu_1 \backsim \mu_2$) *it holds that:*

$$(\mu_1 \backsim \mu_2)(?x) = \begin{cases} \mu_1(?x) & \text{if } ?x \in \text{dom}(\mu_1) \setminus \text{dom}(\mu_2), \\ \mu_2(?x) & \text{if } ?x \in \text{dom}(\mu_2) \setminus \text{dom}(\mu_1), \\ \mu_1(?x) & \text{if } ?x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \text{ and } \mu_1(?x) \neq \varnothing, \\ \mu_2(?x) & \text{if } ?x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \text{ and } \mu_2(?x) \neq \varnothing, \\ \varnothing & \text{otherwise.} \end{cases}$$

Selection formulas of relational algebra do not admit null values. As usual, the generalization to include null values uses 3-valued logic instead of 2-valued logic as relational algebra does.

Definition 4.3 (Selection formulas)**.** *A selection formula, recursively defined, is an equality $t_1 = t_2$ where $t_1, t_2 \in \mathbf{C} \cup \mathbf{V} \cup \{\varnothing\}$, or a Boolean combination of selection formulas (with the*

connectives $\wedge, \neg$ *and* $\vee$*). Let* φ *be a selection formula,* μ *be a* $\varnothing$*-mapping where* $\mathrm{var}(\varphi) \subseteq \mathrm{dom}(\mu)$*, and* $\mu' : \mathbf{C} \cup \mathbf{V} \cup \{\varnothing\} \to \mathbf{C} \cup \mathbf{V} \cup \{\varnothing\}$ *be the function defined as:*

$$\mu'(x) = \begin{cases} \mu(x) & \text{if } x \in \mathrm{dom}(\mu), \\ x & \text{otherwise.} \end{cases}$$

The truth value of φ *in* μ*, denoted* $\mu(\varphi)$*, is defined recursively as follows:*

1. *If* φ *has the form* $t_1 = t_2$ *then:*
   (a) $\mu(\varphi)$ *is true if* $\mu'(t_1) = \mu'(t_2)$, $\mu'(t_1) \neq \varnothing$ *and* $\mu'(t_1) \neq \varnothing$;
   (b) $\mu(\varphi)$ *is false if* $\mu'(t_1) \neq \mu'(t_2)$, $\mu'(t_1) \neq \varnothing$ *and* $\mu'(t_1) \neq \varnothing$;
   (c) $\mu(\varphi)$ *is unknown, otherwise.*
2. *If* φ *is a Boolean combination of selection formulas then the truth value of* $\mu(\varphi)$ *is defined using 3-valued logic as usual.*

Now we are ready to define the generalized algebra.

Definition 4.4 (Generalized algebra)**.** *Let* R*,* R_1 *and* R_2 *be* $\varnothing$*-tables,* $\mathcal{X}$ *be a set of variables where* $\mathcal{X} \subseteq \mathrm{dom}(R)$*,* $?\mathrm{x}, ?\mathrm{y}$ *be variables where* $?\mathrm{x} \in \mathrm{dom}(R)$ *and* $?\mathrm{y} \notin \mathrm{dom}(R)$*, and* φ *be a selection formula. The generalized algebra is defined recursively as follows:*

1. $\pi_{\mathcal{X}}(R) = \{\mu|_{\mathcal{X}} \mid \mu \in R\}$;
2. $\rho_{?\mathrm{x}/?\mathrm{y}}(R)$ *is the table resulting from replacing attribute* $?\mathrm{x}$ *by attribute* $?\mathrm{y}$ *in table* R;
3. $R_1 \cup R_2 = \{\mu \mid \mu \in R_1 \text{ or } \mu \in R_2\}$;
4. $\sigma_{\text{under } \varphi}(R) = \{\mu \in R \mid \mu(\varphi) \text{ is true}\}$;
5. $\sigma_{\text{over } \varphi}(R) = \{\mu \in R \mid \mu(\varphi) \text{ is true or unknown}\}$;
6. $R_1 \bowtie_{\text{under}} R_2 = \{\mu_1 \smile \mu_2 \mid \mu_1 \in R_1, \mu_2 \in R_2, \text{ and } \mu_1 \sim_{\text{under}} \mu_2\}$;
7. $R_1 \bowtie_{\text{over}} R_2 = \{\mu_1 \smile \mu_2 \mid \mu_1 \in R_1, \mu_2 \in R_2, \text{ and } \mu_1 \sim_{\text{over}} \mu_2\}$;
8. $R_1 -_{\text{under}} R_2 = \{\mu_1 \in R_1 \mid \text{for all } \mu_2 \in R_2 \text{ does not } \mu_1 \sim_{\text{over}} \mu_2\}$;
9. $R_1 -_{\text{over}} R_2 = \{\mu_1 \in R_1 \mid \text{for all } \mu_2 \in R_2 \text{ does not } \mu_1 \sim_{\text{under}} \mu_2\}$.

We next explain how the generalized algebra is based on the Biskup algebra for tables with maybe tuples (see Section 3.3.3). In the previous chapter, we defined M-tables as pairs of V-tables because we view blank nodes as marked nulls. Now we define m-tables (lower case m)[1] as pairs of $\varnothing$-tables because in this chapter we are studying unbound values. An *m-table* M is a pair (M_1, M_2) where M_1 and M_2 are $\varnothing$-tables and $M_1 \subseteq M_2$. We write $\mathrm{sure}(M)$ and $\mathrm{poss}(M)$ to denote the first and second components of an m-table M, respectively. Given a $\varnothing$-table R we write R^m to denote the m-table where $\mathrm{sure}(R^m) = \mathrm{poss}(R^m) = R$. The tuples $\mu \in \mathrm{poss}(M) \setminus \mathrm{sure}(M)$ are said to be the maybe-tuples of the m-table M.

[1]We use upper and lower case to distinguish the tables that are extended with marked nulls (M-tables) from the tables that are extended with unbound values (m-tables).

It is not difficult to see that for each operator $\circ$ of the Biskup algebra for m-tables we define two operators $\circ_{\text{under}}$ and $\circ_{\text{over}}$ such that the following relations hold:

$$\left.\begin{array}{r} \circ_{\text{under}}(R) = \text{sure}(\circ(R^m)) \\ \circ_{\text{over}}(R) = \text{poss}(\circ(R^m)) \end{array}\right\} \text{if } \circ \text{ is unary,} \tag{4.1}$$

$$\left.\begin{array}{r} R_1 \circ_{\text{under}} R_2 = \text{sure}(R_1^m \circ R_2^m) \\ R_1 \circ_{\text{over}} R_2 = \text{poss}(R_1^m \circ R_2^m) \end{array}\right\} \text{if } \circ \text{ is binary.} \tag{4.2}$$

We omit the subscripts "under" and "over" in Definition 4.4 for operators $\circ$ where $\circ_{\text{under}}$ and $\circ_{\text{over}}$ are equivalent. Formulas 4.1 and 4.2 are true for all operators of the generalized algebra except for the operators $\sigma_{\text{under}\,\varphi}$ and $\sigma_{\text{over}\,\varphi}$. The truth of these equalities follow from simple inspection and the fact that we translate the definitions from the Biskup [15] algebra to Definition 4.4 following these equalities. The difference regarding the operators $\sigma_{\text{under}\,\varphi}$ and $\sigma_{\text{over}\,\varphi}$ is due to the fact that Biskup considered only atomic equalities as selection formulas φ, while we considered also Boolean combinations of selection formulas. Unfortunately, as the following example shows, our alternative selection operators produce results that are less informative than the Biskup ones.

Example 4.3. *Let R be the $\varnothing$-table with a single tuple $\{?x \mapsto \varnothing\}$ and Q be the query $\sigma_{?x=a}(R)$. According to the generalized algebra the following equalities hold:*

$$\sigma_{\text{under } ?x=a}(R) = \{\},$$
$$\sigma_{\text{over } ?x=a}(R) = \{\{?x \mapsto \varnothing\}\}.$$

On the other hand, according to Biskup a selection of tuples in the m-table R^m that pass a formula $?x = a$ is evaluated as a join among R^m and the m-table consisting in a single sure tuple $\{?x \mapsto a\}$. The result is thus:

$$\text{sure}(\sigma_{?x=a}(R^m)) = \{\},$$
$$\text{poss}(\sigma_{?x=a}(R^m)) = \{\{?x \mapsto a\}\}.$$

The result obtained in this example by the Biskup algebra is strictly more informative than the result obtained with the generalized algebra.

4.3.2 Mapping SQL and SPARQL operators to the generalized algebra

Table 4.2 presents the generalized operator that is used to extend the relational algebra to support null values according to the languages SQL and SPARQL. They agree in all operators except join and difference.

To show the correspondence among SQL and the generalized algebra operators we present the results given by example queries in popular implementations in Appendix B. These correspondences are not conclusive. One could try to prove the correspondence by using a formal semantics of SQL as e.g., the one provided by Guagliardo and Libkin [32] in order to provide a more conclusive correspondence. However, we do not include such validation here because we concentrate in SPARQL.

Regarding the correspondence among SPARQL and the generalized algebra operators, notice that SPARQL operators can return set of mappings with different domains, while the

Standard Algebra	Generalized Algebra	
	SQL	SPARQL
Union	$\cup$	$\cup$
Projection	π	π
Renaming	ρ	ρ
Selection	σ_{under}	σ_{under}
Join	$\bowtie_{\text{under}}$	$\bowtie_{\text{over}}$
Difference	$-_{\text{over}}$	$-_{\text{under}}$

Table 4.2: The generalized relational algebra operators for SQL and SPARQL.

$$\begin{array}{ccc} \Omega_1, \ldots, \Omega_n & \xrightarrow{\text{fill}(\cdot, \mathcal{Z})} & R_1, \ldots, R_n \\ \downarrow Q & & \downarrow Q' \\ \Omega & \xrightarrow{\text{fill}(\cdot, \mathcal{Z})} & R \end{array}$$

Figure 4.2: Diagram for SPARQL and the generalized relational algebra. $\Omega, \Omega_1, \ldots, \Omega_n$ are sets of SPARQL mappings, and $R, R_1, \ldots, R_n$ are the corresponding $\varnothing$-tables that result from padding the unbound values in mappings with null values to conform tables with a set of attributes $\mathcal{Z}$ that is big enough to include all the variables referred in the queries. Query Q' is said to *correspond* to a query Q if the diagram is commutative.

generalized algebra operators return tables. To make these two types of output comparable we require a map among them. This map is provided by the function fill that fills unbound variables with null values (see Definition 2.7). *A generalized relational algebra query Q' is said to correspond to a SPARQL query Q* if the diagram in Figure 4.2 is commutative.

In general, SPARQL queries cannot be translated to the generalized relational algebra by defining a correspondence among SPARQL operators and generalized relational algebra operators. For instance, the union operator in the SPARQL query (Q_1 UNION Q_2) cannot be directly interpreted as a union operator in the generalized relational algebra operator because unlike relational algebra, SPARQL allows the union of queries that return tuples with different domains. To translate SPARQL queries to the generalized relational algebra we present a normalization that generate queries, called *tabular*.

Definition 4.5. *Tabular SPARQL queries are defined recursively as follows:*

1. *A triple pattern is a tabular SPARQL query.*
2. *A query of the form* VALUES $\mathcal{X}\Omega$ *is tabular.*
3. *If Q_1 and Q_2 are tabular SPARQL queries, then* (Q_1 AND Q_2) *is a tabular SPARQL query.*
4. *If Q_1 and Q_2 are tabular SPARQL queries with* $\text{dom}(Q_1) = \text{dom}(Q_2)$, *then* ($Q_1$ UNION Q_2) *and* (Q_1 MINUS Q_2) *are tabular SPARQL queries.*
5. *If Q_1 is a tabular SPARQL query and $\mathcal{X}$ is a finite set of variables, then* (SELECT $\mathcal{X}$ WHERE Q_1) *is a tabular SPARQL query.*

6. *If Q_1 is a tabular SPARQL query and φ is a filter-condition whose variables are in* $\mathrm{dom}(Q_1)$, *then* $(Q_1$ FILTER $\varphi)$ *is a tabular SPARQL query.*

Lemma 4.1. *For every SPARQL-1 query without the BIND operator there exists an equivalent tabular SPARQL query.*

Proof. It can be shown by induction in the structure of the query.

1. If Q is a triple pattern or has the form (VALUES $\mathcal{X}\Omega$) then by definition Q is tabular. These are the base cases of the induction.
2. If Q has the form $(Q_1$ AND $Q_2)$ then, by induction, there exist the tabular queries Q_1' and Q_2' that are equivalent with Q_1 and Q_2. By definition, the query $Q' = (Q_1$ AND $Q_2)$ is tabular. Since Q' and Q are equivalent, the inductive hypothesis holds for this case.
3. If Q has the form $(Q_1$ UNION $Q_2)$ then, by induction, there exist the tabular queries Q_1' and Q_2' that are equivalent with Q_1 and Q_2. Let $\mathcal{X}$ be the set of variables $\mathrm{dom}(Q_1) \cup \mathrm{dom}(Q_2)$, and Q' be the query

   $$((\text{SELECT } \mathcal{X} \text{ WHERE } Q_1) \text{ UNION } (\text{SELECT } \mathcal{X} \text{ WHERE } Q_1)).$$

   By definition, Q' is tabular. Since Q' and Q are equivalent, the inductive hypothesis holds for this case.
4. If Q has the form $(Q_1$ MINUS $Q_2)$, then the inductive hypothesis holds by the same argument as the used in the previous case.
5. If Q has the form (SELECT $\mathcal{X}$ WHERE Q_1) then, by induction, there exists a tabular query Q_1' equivalent to Q_1. By definition the query $Q' =$ (SELECT $\mathcal{X}$ WHERE Q_1') is tabular. Since Q' and Q are equivalent, the inductive hypothesis holds for this case.
6. If Q has the form $(Q_1$ FILTER $\varphi)$ then, by induction, there exists a tabular query Q_1' that is equivalent to Q_1. By definition, $Q' = (Q_1'$ FILTER $\varphi)$ is tabular. Since Q' and Q are equivalent, the inductive hypothesis holds for this case.

□

Lemma 4.1 shows that every SPARQL query can be normalized as a tabular SPARQL query. Moreover, since the proof of the lemma is inductive, a normalization algorithm is provided. We next show that tabular queries can be expressed using the generalized relational algebra. Recall that in the previous chapter, given a SPARQL query Q in the relational fragment, we defined the relational algebra query $\mathrm{ra}(Q)$ equivalent to Q. We next extend this translation to translate an arbitrary SPARQL query Q into a *generalized relational algebra query*, denoted $\mathrm{gra}(Q)$.

Definition 4.6. *Let Q be the set of tabular SPARQL queries. Then the generalized relational algebra is the set of* $\mathrm{gra}(Q)$ *together with the relational operations defined recursively as follows:*

1. *If Q is a triple pattern then* $\mathrm{gra}(Q) = \mathrm{ra}(Q)$.
2. *If Q has the form* (VALUES $\mathcal{X}\Omega$) *then* $\mathrm{gra}(Q) = R_\Omega$, *where R_Ω is the $\varnothing$-table* $\mathrm{fill}(\Omega, \mathcal{X})$.
3. *If Q has the form* $(Q_1$ AND $Q_2)$ *then* $\mathrm{gra}(Q) = \mathrm{gra}(Q_1) \bowtie \mathrm{gra}(Q_2)$.

4. *If Q has the form* (Q_1 UNION Q_2) *then* $\text{gra}(Q) = \text{gra}(Q_1) \cup \text{gra}(Q_2)$.

5. *If Q has the form* (Q_1 MINUS Q_2) *then* $\text{gra}(Q) = \text{gra}(Q_1) -_{\text{under}} \text{gra}(Q_2)$.

6. *If Q has the form* (SELECT $\mathcal{X}$ WHERE Q_1) *then* $\text{gra}(Q) = \text{fill}(\{\mu_\emptyset\}, \mathcal{Y}) \bowtie \pi_{\mathcal{Z}} \text{gra}(Q_1)$, *where $\mu_\emptyset$ denotes the solution mapping with empty domain, $\mathcal{Y} = \mathcal{X} \setminus \text{dom}(Q_1)$, and $\mathcal{Z} = \mathcal{X} \cap \text{dom}(Q_1)$.*

7. *If Q has the form* (Q_1 FILTER φ) *then* $\text{gra}(Q) = \sigma_\varphi(\text{gra}(Q_1))$.

So far, we have described the generalized relational algebra, and a translation from SPARQL to the generalized relational algebra. To define the results of a SPARQL query in terms of the results of a generalized algebra query, we need a function unfill that removes the null values introduced by the generalized relational algebra.

Definition 4.7. *Giving a mapping admitting null values as values μ, we write* $\text{unfill}(\mu)$ *to denote the mapping $\mu|_{\mathcal{X}}$ where $\mathcal{X}$ is the set of variables $\{?x \in \text{dom}(\mu) \mid \mu(?x) \neq \varnothing\}$. Given a set of mappings Ω admitting null values, we write* $\text{unfill}(\Omega)$ *to denote the set of mappings $\{\text{unfill}(\mu) \mid \mu \in \Omega\}$.*

The following theorem gives a semantics for tabular SPARQL queries trough the generalized relational algebra.

Theorem 4.1. *Let Q be a tabular SPARQL query, G be an RDF graph, and $D = \text{vdb}(G)$ be the relational database that codifies the graph G. Then, $[\![Q]\!]_G = \text{unfill}([\![\text{gra}(Q)]\!]_D)$.*

Proof. Let Q be a query, G be an RDF graph, and D be the $\varnothing$-database for G. The equality $[\![Q]\!]_G = \text{unfill}([\![\text{gra}(Q)]\!]_D)$ is equivalent to the equality $[\![\text{gra}(Q)]\!]_D = \text{fill}([\![Q]\!]_G, \text{dom}(Q))$, so we will show that this second equation holds by induction in the structure of Q.

1. If Q is a triple pattern then $[\![\text{gra}(Q)]\!]_D = \text{fill}([\![Q]\!]_G, \text{dom}(Q))$ follows from Theorem 3.1 because no unbound values are involved.

2. If Q has the form (VALUES $\mathcal{X}\Omega$) then, by construction, it holds that $[\![\text{gra}(Q)]\!]_D = \text{fill}([\![Q]\!]_G, \text{dom}(Q))$.

3. If Q has the form (Q_1 AND Q_2) then it holds that

$$
\begin{aligned}
&\text{fill}([\![(Q_1 \text{ AND } Q_2)]\!]_G, \text{dom}(Q)) \\
&\quad = \text{fill}(\{\mu_1 \smile \mu_2 \mid \mu_1 \in [\![Q_1]\!]_G, \mu_2 \in [\![Q_2]\!]_G, \mu_1 \sim \mu_2\}, \text{dom}(Q)) \\
&\quad = \{\mu_1' \smile \mu_2' \mid \mu_1' \in \text{fill}([\![Q_1]\!]_G, \text{dom}(Q_1)), \mu_2' \in \text{fill}([\![Q_2]\!]_G, \text{dom}(Q_2)), \mu_1' \sim_{\text{over}} \mu_2'\} \\
&\quad = \text{fill}([\![Q_1]\!]_G, \text{dom}(Q_1)) \bowtie_{\text{over}} \text{fill}([\![Q_2]\!]_G, \text{dom}(Q_2)) \\
&\quad = [\![\text{gra}(Q_1)]\!]_D \bowtie_{\text{over}} [\![\text{gra}(Q_2)]\!]_D \\
&\quad = [\![\text{gra}(Q_1) \bowtie_{\text{over}} \text{gra}(Q_2)]\!]_D \\
&\quad = [\![\text{gra}(Q)]\!]_D.
\end{aligned}
$$

4. If Q has the form $(Q_1 \text{ UNION } Q_2)$ then it holds that

$$\begin{aligned}
&\text{fill}(\llbracket(Q_1 \text{ UNION } Q_2)\rrbracket_G, \text{dom}(Q)) \\
&\quad = \text{fill}(\llbracket(Q_1\rrbracket_G \cup \llbracket Q_2\rrbracket_G, \text{dom}(Q)) \\
&\quad = \text{fill}(\llbracket Q_1\rrbracket_G, \text{dom}(Q)) \cup \text{fill}(\llbracket Q_2\rrbracket_G, \text{dom}(Q)) \\
&\quad = \llbracket\text{gra}(Q_1)\rrbracket_D \cup \llbracket\text{gra}(Q_2)\rrbracket_D \\
&\quad = \llbracket\text{gra}(Q_1) \cup \text{gra}(Q_2)\rrbracket_D \\
&\quad = \llbracket\text{gra}(Q)\rrbracket_D.
\end{aligned}$$

5. If Q has the form $(Q_1 \text{ MINUS } Q_2)$ then it holds that

$$\begin{aligned}
&\text{fill}(\llbracket(Q_1 \text{ MINUS } Q_2)\rrbracket_G, \text{dom}(Q)) \\
&\quad = \text{fill}(\{\mu_1 \mid \mu_1 \in \llbracket Q_1\rrbracket_G, \nexists \mu_2 \in \llbracket Q_2\rrbracket_G, \mu_1 \sim \mu_2\}, \text{dom}(Q)) \\
&\quad = \{\mu_1' \mid \mu_1' \in \text{fill}(\llbracket Q_1\rrbracket_G, \text{dom}(Q_1)), \nexists \mu_2' \in \text{fill}(\llbracket Q_2\rrbracket_G, \text{dom}(Q_2)), \mu_1' \sim_{\text{over}} \mu_2'\} \\
&\quad = \text{fill}(\llbracket Q_1\rrbracket_G, \text{dom}(Q_1)) -_{\text{under}} \text{fill}(\llbracket Q_2\rrbracket_G, \text{dom}(Q_2)) \\
&\quad = \llbracket\text{gra}(Q_1)\rrbracket_D -_{\text{under}} \llbracket\text{gra}(Q_2)\rrbracket_D \\
&\quad = \llbracket\text{gra}(Q_1) -_{\text{under}} \text{gra}(Q_2)\rrbracket_D \\
&\quad = \llbracket\text{gra}(Q)\rrbracket_D.
\end{aligned}$$

6. If Q has the form $(\text{SELECT } \mathcal{X} \text{ WHERE } Q_1)$ then let $\mathcal{Y}$ be the set of variables $\mathcal{X} \setminus \text{dom}(Q_1)$, and $\mathcal{Z}$ be the set of variables $\mathcal{X} \cap \text{dom}(Q_1)$. Then, it holds that

$$\begin{aligned}
&\text{fill}(\llbracket(\text{SELECT } \mathcal{X} \text{ WHERE } Q_1)\rrbracket_G, \text{dom}(Q)) \\
&\quad = \text{fill}(\llbracket(\text{VALUES } \mathcal{Y}\{\mu_\emptyset\}) \text{ AND } (\text{SELECT } \mathcal{Z} \text{ WHERE } Q_1)\rrbracket_G, \text{dom}(Q)) \\
&\quad = \text{fill}(\llbracket(\text{VALUES } \mathcal{Y}\{\mu_\emptyset\})\rrbracket_G, \mathcal{Y}) \bowtie \text{fill}(\llbracket(\text{SELECT } \mathcal{Z} \text{ WHERE } Q_1)\rrbracket_G, \mathcal{Z}) \\
&\quad = \llbracket\text{fill}(\{\mu_\emptyset\}, \mathcal{Y})\rrbracket_D \bowtie \llbracket\pi_{\mathcal{Z}} \text{gra}(Q_1)\rrbracket_D \\
&\quad = \llbracket\text{fill}(\{\mu_\emptyset\}, \mathcal{Y}) \bowtie \pi_{\mathcal{Z}} \text{gra}(Q_1)\rrbracket_D \\
&\quad = \llbracket\text{gra}(\text{SELECT } \mathcal{X} \text{ WHERE } Q_1)\rrbracket_D \\
&\quad = \llbracket\text{gra}(Q)\rrbracket_D.
\end{aligned}$$

7. If Q has the form $(Q_1 \text{ FILTER } \varphi)$ then it holds that

$$\begin{aligned}
&\text{fill}(\llbracket(Q_1 \text{ FILTER } \varphi)\rrbracket_G, \text{dom}(Q)) \\
&\quad = \text{fill}(\{\mu \in \llbracket Q_1\rrbracket_G \mid \mu(\varphi) = \text{true}\}, \text{dom}(Q)) \\
&\quad = \{\mu \in \text{fill}(\llbracket Q_1\rrbracket_G, \text{dom}(Q)) \mid \mu(\varphi) = \text{true}\} \\
&\quad = \{\mu \in \llbracket\text{gra}(Q_1)\rrbracket_D, \text{dom}(Q)) \mid \mu(\varphi) = \text{true}\} \\
&\quad = \llbracket\sigma_\varphi(\text{gra}(Q_1))\rrbracket_D \\
&\quad = \llbracket\text{gra}(Q_1 \text{ FILTER } \varphi)\rrbracket_D \\
&\quad = \llbracket\text{gra}(Q)\rrbracket_D.
\end{aligned}$$

□

The previous theorem is restricted to tabular queries. Since every SPARQL query in the SPARQL-1 fragment that does not have a BIND operator can be expressed as a tabular SPARQL query (see Lemma 4.1), this theorem can be generalized for SPARQL-1 queries without a the BIND operator.

Corollary 4.1. *For every SPARQL-1 query Q without the BIND operator, and RDF graph G, there exists a relational algebra query Q' such that for every RDF graph G, it holds that* $[\![Q]\!]_G = \text{unfill}([\![Q']\!]_{\text{vdb}(G)})$.

Proof. It follows from Lemma 4.1 and Theorem 4.1. □

4.4 Conclusions

In this chapter, we have studied the semantics of unbound values in SPARQL. In particular, we addressed the question of whether unbound values can be understood by using the classical classification of null values into unknown values, inapplicable values, and no-information nulls.

In order to formalize our approach, we introduced a *generalized relational algebra*. The generalized algebra consists in defining two variants for each operator. The first, denoted with the subscript "under", returns answers that are valid for all models. The second, denoted with the subscript "over", returns answers that are valid for some models.

With this machinery, we were able to understand the rationale behind the unbound values in SPARQL and how this rationale is related with the one of SQL. The picture goes as follows: shows that SPARQL is characterized by a join that computes the answers that are valid in some interpretation of the joined tables, whereas SQL is characterized by a join that computes certain answers. This characterization is inverted for the difference. The SPARQL difference computes certain answers whereas the SQL difference computes possible answers.

As an aside, let us note that generalized relational algebra can be viewed therefore an extension of relational algebra by introducing the modals *necessary* and *possible* of modal logic. Since there is a natural translation from the standard relational algebra to first order logic, it arises the question of whether we can translate generalized algebra queries to first order modal logic [26]. We do not address this question in this thesis, however, we conjecture that a positive answer to this question could lead to an elegant semantics for SPARQL and SQL based on first order modal logic.

Part III

On the semantics of EXISTS

Chapter 5

The notion of substitution under incomplete data

Several query languages as SQL, SPARQL, and recently G-Core [4], allow us to define a query, called the *inner query* or the *subquery*, inside another query, called the *outer query*. Usually, they are evaluated once for each answer of the outer query. Attributes X that occur in both, the outer query and the inner query, are said to be *correlated* if X has to be substituted in the inner query by the value of X in the outer-query each time that the inner query is evaluated.

Let us present an example to show how this works in SPARQL:

Example 5.1. *Consider the following SQL query:*

```
1 SELECT orders.id
2 FROM orders
3 WHERE NOT EXISTS (
4    SELECT *
5    FROM items
6    WHERE orders.id = items.order_id
7 )
```

Intuitively, this query asks for orders having no items. Lines 1–3 codify the outer query, and lines 4–6 codify the inner query. The inner and the outer query each separately are not complete queries. They do not compile, and thus they cannot be evaluated. The outer query is not complete until an inner query inside the parenthesis of the `NOT EXISTS` *clause is present. The inner query is not complete since the attribute* `orders.id` *is not defined in it. The attribute* `orders.id` *is here the correlated variable, because it has to be substituted by the value provided by the outer query each time the inner query is evaluated.*

In SPARQL two forms of nesting are usually referred as subqueries:

1. when a query occurs inside a WHERE clause as a graph pattern—called a subselect—to be combined with other graph patterns with the operations AND, MINUS, UNION, and OPT; and

2. when a query occurs inside an EXISTS clause.

In this chapter we focus on the study of the EXISTS clause in SPARQL. The EXISTS clause occurs inside the FILTER clause as an atomic Boolean condition. The EXISTS clause contains a query, called the *inner query*, that is evaluated against a mapping that is a solution of another query, called the *outer query*. The FILTER clause takes answers from the outer and uses them to evaluate the filter-condition, and thus, evaluate the inner query.

The semantics of the EXISTS clause has been a matter of discussions in the Web Semantic community. We [40], and Patel-Schneider and Martin [57] reported several issues with the semantics the EXISTS clause. Most of the discussion boils down to the subtleties behind the notion of substitution.

In this chapter we study the notion of substitution (in the EXIST clause) in SPARQL and its issues. We present three proposals made by the community to approach the semantics of the SPARQL EXISTS operator. We conclude illustrating the relation among the different semantics of substitution and the notion of range-restricted variables of predicate calculus.

Organization of this chapter

- In Section 5.1 we describe the problem of substitution in SPARQL. We observe several issues regarding ambiguities and contradictions in the specification, and the different forms in which substitution is implemented. In particular, we describe three proposals, called shallow-binding, deep-binding and environment-binding.

- The environment-binding proposal differs from the other two in that it is based on the notion of environment (a mapping that associates variables with their current values) instead of the notion of substitution (the transformation of a query by substituting variables by values in it). In Section 5.2 we show that the environment-binding proposal can be formalized equivalently using the notion of substitution.

- In Section 5.3 we show that to define a semantics for substitution we have to decide what variables are replaceable. Also, we show the relation of the replaceable variables in query languages as SQL and SPARQL with the notions of free, bound, and range restricted variables in first-order logic.

- We conclude this chapter in Section 5.4.

5.1 The problem of substitution in SPARQL

The semantics of the EXISTS clause in SPARQL is strongly related to the form how values in correlated queries are treated. This amounts to the notions of correlation and substitution of values (e.g. the values of the outer query into the inner query to be evaluated).

As we show throughout this section, in the current SPARQL specification the notions of correlation and substitution are not well defined. This known problem (as we discuss below) triggers several ambiguities and contradictions in the semantics, and different semantics in implementations as Fuseki, Blazegraph, Virtuoso and rdf4j. In this section we summarize the issues of substitution in SPARQL and describe three existing proposals to solve these issues.

5.1.1 An overview of the problems of substitution in SPARQL

The anomalies in the W3C Specification of SPARQL 1.1 were detected early and reported no later than 2014, when two erratas were registered.[1][2] We published in 2016 a technical report [40] where we analyzed the reported issues and other we found in the specification. This report gained some attention and a W3C Community group (called the EXISTS CG in what follows) was created to address the issue.[3] In this section we present the problems of substitution in SPARQL we reported in [40].

The W3C SPARQL Specification [36] gives two definitions that are relevant for the semantics of the EXISTS clause we are discussing:

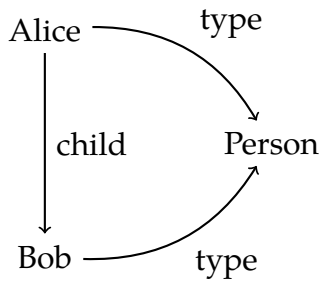
Definition 5.1 (Standard substitution, W3C Spec., §18.6)**.** *Let μ be a solution mapping and P be a graph pattern.* substitute(P, μ) *is the graph pattern formed by replacing, for each X in* dom(μ)*, every occurrence of a variable X in P by $\mu(x)$.*

Definition 5.2 (Evaluation of Exists, W3C Spec. §18.6)**.** *Let μ be the current solution mapping for a filter and P a graph pattern: The value* EXISTS(P)*, given G, is true if and only if* Eval(G, substitute(P, μ)) *is a non-empty sequence.*

We have adapted Definition 5.2 to follow the notation used in this thesis. The original definition writes $D(G)$ where we simply write G. We simplify the notation since we do not consider datasets composed from multiple RDF graphs as the specification does.

Before presenting the issues of substitution in SPARQL, we present an example where the semantics is neither ambiguous nor contradictory. The following example help understanding the rationale behind Definitions 5.1 and 5.2.

Example 5.2. *Let G be the graph depicted as follows:*



[1] https://www.w3.org/2013/sparql-errata#errata-query-8
[2] https://www.w3.org/2013/sparql-errata#errata-query-10
[3] https://www.w3.org/community/sparql-exists/

Let Q be the following SPARQL query:

$$\underbrace{(?x, \text{type}, \text{Person})}_{Q_1} \text{ FILTER } \underbrace{(\neg \text{ EXISTS } (?x, \text{child}, ?y))}_{Q_2}.$$

Intuitively, query Q asks for people having no children. Query Q_1 is the outer query, and query Q_2 is the inner query.

The answers of the outer query on G query are:

$$\left[\begin{array}{c} ?x \\ \hline \text{Alice} \\ \text{Bob} \end{array}\right].$$

To evaluate the inner query we have to consider each mapping that is an answer of the outer query to apply the substitution.

For the first answer (Alice)*, the result of substituting the variable* ?x *by Alice is the triple pattern* (Alice, child, ?y) *whose results on G are:*

$$\left[\begin{array}{c} ?y \\ \hline \text{Bob} \end{array}\right].$$

Thus, the ¬EXISTS *clause is evaluated as false, so the tuple* (Alice) *of the outer query is discarded.*

For the second answer to the outer query (Bob)*, the result of substituting the variable* ?x *by Bob is the triple pattern* (Bob, child, ?y)*. This triple pattern return no answers when evaluated on G. Thus, this tuple of the outer query is not discarded.*

Hence, the evaluation of query Q on graph G returns the following answers:

$$\left[\begin{array}{c} ?x \\ \hline \text{Bob} \end{array}\right].$$

Though definitions 5.1 and 5.2 seem clear and simple, they have several issues and even contradictory pieces we next describe. The issues we describe herein were reported in [40] and [57].

Issue 1: Top-down or bottom-up evaluation?

Definitions 5.1 and 5.2 present a top-down evaluation for inner queries. An inner query cannot be evaluated until the outer query is evaluated. In a previous report [40] we showed that the substitution notion contradicts the statement "Due to the bottom-up nature of SPARQL query evaluation, the subqueries are evaluated logically first, and the results are projected up to the outer query" of the specification [36].

Example 5.3. *Let G be the graph depicted in Example 5.2, and Q be the following SPARQL query:*

(?x, type, Person) FILTER (¬ EXISTS ((?z, child, ?y) FILTER (?z = ?x))).

Intuitively, this query aims to be equivalent to the query in Example 5.2. However, this query depends on the evaluation being done either top-down or bottom-up.

If query Q is evaluated top-down, then variable ?x *must be substituted in the inner query. For instance, for the solution where* ?x *is Alice, the inner query becomes the query*

$$(?z, \text{child}, ?y) \text{ FILTER } (?z = \text{Alice})).$$

This query returns:

$$\left[\begin{array}{cc} ?y & ?z \\ \hline \text{Bob} & \text{Alice} \end{array}\right].$$

Thus, this solution where ?x *is Alice is dropped from the solutions of Q.*

On the contrary, if query Q is evaluated bottom-up, then the inner query is not modified. Thus, the inner query is:

$$(?z, \text{child}, ?y) \text{ FILTER } (?z = ?x).$$

Like in the previous case, the triple pattern (?z, child, ?y) *returns:*

$$\left[\begin{array}{cc} ?y & ?z \\ \hline \text{Bob} & \text{Alice} \end{array}\right].$$

However, the filter-condition ?z = ?x *returns unknown because variable* ?x *is unbound when the filter-condition is evaluated. Hence, the solution is dropped. Since the inner query returns no answers, all answers of the outer-query are accepted. In particular, the solution where* ?x *is Alice is accepted as a solution of query Q.*

Hence, this example shows that the top-down an bottom-up evaluations of query Q produce different answers.

Issue 2: Substitution can break the grammar

The SPARQL specification states that if a variable ?x is in the domain of the current mapping μ, then every occurrence of ?x must be replaced by $\mu(?x)$. However, in some cases we cannot replace a variable ?x by a constant, because the place where the variable ?x occurs admits only variables. We reported this issue in the Jena's mailing list.[4] This issue was included in the SPARQL 1.1 errata.[5]

Example 5.4. *Variable* ?x *cannot be substituted by a constant because it would break the SPARQL grammar:*

1. (SELECT ?x WHERE Q)
2. (Q BIND (F AS ?x))
3. (VALUES {?x}{{?x $\mapsto$ a}})

For instance, if we substitute ?x *by a constant b in the last query produces the query* (VALUES $\{b\}\{b \mapsto a\}$) *that has no sense because* $\{b \mapsto a\}$ *is not a valid RDF mapping.*

[4] `http://mail-archives.apache.org/mod_mbox/jena-users/201412.mbox/<1419368102.2824.10.camel@ruil.local>`

[5] `https://www.w3.org/2013/sparql-errata#errata-query-10`

Issue 3: MINUS behaves different if solution domains are disjoint

The SPARQL specification has a different semantics for the difference operator that the one used in this thesis. Patel-Scheneider and Martin [57] noticed that according to the standard semantics the difference operator produces counterintuitive results when combined with EXISTS. Given two sets of mappings Ω_1 and Ω_2, according to the standard a query Ω_1 MINUS Ω_2 is translated as the algebraic operation Minus(Ω_1, Ω_2), that is defined by the SPARQL specification [36] as follows:

$$\text{Minus}(\Omega_1, \Omega_2) = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 : \mu_1 \nsim \mu_2, \text{ or } \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \varnothing\}.$$

Example 5.5 ([57])**.** *Consider the following query*

$$\{?\text{x} \mapsto a\} \text{ FILTER (EXISTS((?x}, p, o) \text{ MINUS (?x}, p, o))).$$

To check if the mapping $\{?\text{x} \mapsto a\}$ *is an answer of the query we have to check if the inner query has answers. However, according to the standard we first have to substitute* ?x *by* a *in the inner query. The result of the substitution is the query:*

$$(a, p, o) \text{ MINUS } (a, p, o).$$

According to the W3C semantics of the difference, since both sides of the MINUS *operator share no variables, the result is* (a, p, o) *if the graph where the query is evaluated has that triple. This is conterintuitive since both sides of the difference are the same.*

Issue 4: The scope of variables is unclear

Definitions 5.1 and 5.2 state that if a variable ?x is in the current mapping μ then all occurrences of variable ?x in the inner query must be replaced by $\mu(?\text{x})$. This statement implies that the scope of variable ?x includes all its occurrences in the inner query. However, Section 12 of the Specification states the contrary: "Note that only variables projected out of the subquery will be visible, or in scope, to the outer query".

In the specification the word "visible" is used to indicate the variables in the inner query that can be substituted by the values carried by the current mapping from the outer query. Visible variables are also called connected variables in the EXISTS CG's report [31]. In the introduction of this chapter we used the term of correlated variables to refer to connected variables.

This contradictory definition of the scope of variables in SPARQL leads to differences among implementations. We show in a report [40] that the queries described in the following example have different answers in popular implementations.

Example 5.6. *Let* $G = \{(a, p, a), (a, q, a)\}$*, and consider the following queries:*

$$\begin{aligned}
Q_1: &\quad \{?\text{x} \mapsto a\} \text{ FILTER } (\neg \text{ EXISTS } ((?\text{y}, p, ?\text{z}) \text{ FILTER } ?\text{z} = ?\text{x})),\\
Q_2: &\quad \{?\text{x} \mapsto a\} \text{ FILTER } (\neg \text{ EXISTS } ((\text{SELECT } ?\text{y WHERE}(?\text{y}, p, ?\text{x})))),\\
Q_3: &\quad \{?\text{x} \mapsto b\} \text{ FILTER } (\text{EXISTS}(((?\text{y}, p, ?\text{z}) \text{ MINUS } (?\text{z}, q, ?\text{x})))).
\end{aligned}$$

If in the queries above we assume that variable ?x *in the inner query is correlated, then the solutions are:*

$$[\![Q_1]\!]_G = \varnothing,$$
$$[\![Q_2]\!]_G = \varnothing,$$
$$[\![Q_3]\!]_G = \varnothing.$$

On the contrary, if we assume that in the queries above variable ?x *is not correlated, then the solutions are:*

$$[\![Q_1]\!]_G = \{\{?x \mapsto a\}\},$$
$$[\![Q_2]\!]_G = \{\{?x \mapsto a\}\},$$
$$[\![Q_3]\!]_G = \{\{?x \mapsto b\}\}.$$

Issue 5: Blank nodes are variables in basic graph patterns

According to Patel-Schneider and Martin [57], if we substitute a variable ?x occurring in a basic graph pattern by a blank node, then the blank node will act as an existential variable. This can produce counterintuitive results.

Example 5.7. *Let* $G = \{(\bot, p, o), (a, q, o)\}$ *be an RDF graph and Q be the following query:*

$$(?x, p, o) \text{ FILTER (EXISTS}(?x, q, o))$$

The evaluation of the outer query produces the solution mapping $\{?x \mapsto \bot\}$*. According the W3C semantics the inner query is thus substituted as* $(\bot, q, o)$*. Since* $\bot$ *acts as an existential variable, pattern* $(\bot, q, o)$ *matches the triple* (a, q, o)*, so there exists a solution for the inner query, and the mapping* $\{?x \mapsto \bot\}$ *is an answer to the whole query.*

According to Patel-Schneider and Martin [57] and also the EXISTS CG [31], the inner query must return no answer since $\bot$ and a represent different identities. This assumption is arguable, because a blank node represents an existential value according the RDF semantics (see Chapter 3). However, to the best of our knowledge SPARQL engines do not return the mapping $\{?x \mapsto \bot\}$ as an answer to the query in Example 5.7.

Issue 6: Substitution of unbound variables

The presence of unbound values in the answers of the outer and inner queries complicates the substitution in SPARQL. Informally, the substitution of a variable ?x in an inner query by the value provided by current mapping μ is understood as restricting the value of ?x in the inner query to $\mu(?x)$. However it is not clear how to apply such restriction if ?x is unbound in the inner query or if it is unbound in the outer query.

The following example expresses the sort of problems that can be found in the evaluation of EXISTS subqueries in SPARQL.

Example 5.8. *Let Q be the following SPARQL query:*

((?id, a, Person) OPT (?id, corpMail, ?email)) FILTER (EXISTS(
((?id, a, Person) OPT (?id, privMail, ?email)) FILTER (?email = *.*com*)))

This query roughly asks for the ID of persons and optionally their corporate email, subject to some conditions given by the expression in the FILTER EXISTS.

Some of the most popular implementations of SPARQL do not agree on how to evaluate this query. For example, consider the following database of persons

id	1	2	3	4	5	6
corpMail	**.com*	**.net*	**.com*	**.net*		
privMail	**.net*	**.com*			**.com*	**.net*

Fuseki, Blazegraph, Virtuoso and rdf4j give almost all of different results.[6] *Fuseki and Blazegraph give*

$$\left[\begin{array}{cc} \text{?id} & \text{?email} \\ \hline 1 & *.com \\ 3 & *.com \\ 5 & - \end{array}\right],$$

Virtuoso gives

$$\left[\begin{array}{cc} \text{?id} & \text{?email} \\ \hline 1 & *.com \\ 3 & *.com \end{array}\right],$$

and finally rdf4j gives

$$\left[\begin{array}{cc} \text{?id} & \text{?email} \\ \hline 3 & *.com \\ 5 & - \end{array}\right].$$

In order to simplify the discussion of this example we will denote mappings of the form $\{?\text{id} \mapsto a, ?\text{email} \mapsto b\}$ as tuples (a, b).

The main question that a semantics must answer is how to assign the variables in order to evaluate the inner and outer expressions. Let us attempt to understand why not all systems agree in showing up person 1. With this person the outer query produces the tuple $\nu = (1, *.com)$.

If we evaluate first the inner query then the variable ?email is bound to $*.net$ for person 1, thus the filter ?email = $*.com$ fails, and so the whole expression inside the first filter fails, hence person 1 is not shown in the output.

Now, if the inner query is evaluated after binding the variable ?email to $*.com$, then the optional part of the inner query does not match and thus the inner query returns the tuple $\mu = (1, -)$. Does this tuple pass the filter of the inner query? This depends on what is the value we assign to variable ?email in the filter condition ?email = $*.com$. If we take the value from the tuple μ then this condition fails, hence person 1 is not shown in the output. On the contrary, if the value is taken from the mapping ν produced in outer query, where the email is bound to $*.com$, then the filter passes, hence person 1 is outputted.

[6] The engines studied in this example are Fuseki 2.5.0, Blazegraph 2.1.1, Virtuoso 7.2.4.2, and rdf4j 2.2.1.

Example 5.8 shows that unbound values introduce an ambiguity in the source of a value. The value of variable ?email in the filter condition ?email = $*.com$ can be taken from two sources, namely the variable `?email` in the outer pattern (contributed by the mapping ν) and the `?email` in the inner pattern (contributed by the mapping μ).

5.1.2 Existing proposals for the semantics of EXISTS

As we already said, to solve the aforementioned issues of the EXISTS semantics, a W3C Community group (called the EXISTS CG from now on) was created.[7] At the moment the EXISTS CG was created, there were already three proposals of semantics for the EXISTS clause, namely *shallow-binding*, *deep-binding*, and *environment-binding*. The EXISTS CG focused in the shallow- and deep-binding proposals but did not reach a consensus and published a report describing both semantics [31].[8] These two proposals can be viewed as two extremes of a range of possible semantics. The existence of a third proposal, the environment-binding, illustrates the fact that it was possible to define multiple alternative semantics between these two extremes. This multiplicity motivates the approach we follow in this thesis: Instead of proposing an additional semantics for the EXISTS clause, we study the problem of substitution that is at the core of them. In this section we describe these three aforementioned proposals because they serve as examples for our study of the substitution problem.

For the sake of readability, in what follows we will omit the keyword VALUES when using mappings to substitute variables. For instance, given a mapping μ, we write μ AND Q instead of the verbose expression (VALUES $\text{dom}(\mu)$ $\{\mu\}$) AND Q.

Deep-binding proposal

The deep-binding semantics was proposed by Seaborne in the EXISTS CG mailing list [67] and summarized in EXISTS CG report [31]. The main idea of this proposal is to restrict the values of a variable at the point where it is bound. According to Seaborne, the replacement must not be done by substituting a variable, but by restricting the values that the variable binds. This is not done in all places where a variable occurs, but in the leafs of the syntactical tree of a query, i.e., at the level of basic graph patterns.

Definition 5.3. *Given a mapping μ and a query Q, the result of substituting μ in Q according to the deep-binding substitution is the query resulting from:*

1. *renaming consistently by fresh variables all variables in Q that are not in* $\text{dom}(Q)$*; and then*
2. *substituting in Q each basic graph pattern P by the graph pattern* $(\mu|_{\text{dom}(P)}$ AND $P)$.

Example 5.9. *If Q is the query* $(?x, p, ?z)$ OPT $(?x, q, ?y)$ *and* $\mu = \{?x \mapsto a, ?y \mapsto b\}$*, then the result of substituting μ in Q is the query*

$$(\{?x \mapsto a\} \text{ AND } (?x, p, ?z)) \text{ OPT } (\{?x \mapsto a, ?y \mapsto b\} \text{ AND } (?x, q, ?y)).$$

[7] `https://www.w3.org/community/sparql-exists/`

[8] The reason of why the EXISTS CG ignored the environment-binding proposal were not technical, but practical: the authors of this proposal did not participate in the EXISTS CG as the authors of the shallow- and deep-binding proposals did.

Example 5.10. *If Q is the query* $(?x, p, ?z)$ MINUS $(?x, q, ?y)$ *and* $\mu = \{?x \mapsto a, ?y \mapsto b\}$*, then the result of substituting μ in Q is the query*

$$(\{?x \mapsto a\} \text{ AND } (?x, p, ?z)) \text{ MINUS } (\{?x \mapsto a\} \text{ AND } (?x, q, ?u)).$$

Observe that, unlike the previous example, in this case variable ?y *is renamed as* ?u *because* $?y \notin \text{dom}(Q)$.

Shallow-binding proposal

The shallow-binding semantics was proposed by Patel-Schneider and Martin [57] to fix problems in the specification. Its aim was to avoid results that, according to the authors are counter intuitive.

In the shallow-binding semantics no substitution is done in the inner query. Instead, the inner query is evaluated independently of the outer query. To check if a mapping μ from the outer query satisfies the EXISTS clause, it suffices to check if μ is compatible with at least a solution mapping of the inner query. Following this idea, the substitution is defined as follows:

Definition 5.4. *Given a mapping μ and a query Q, the result of substituting μ in Q according to the shallow-binding substitution is the query* $(\mu \text{ AND } Q)$.

Example 5.11. *If Q is the query* $(?x, p, ?z)$ MINUS $(?x, q, ?y)$ *and* $\mu = \{?x \mapsto a, ?y \mapsto b\}$*, then the result of substituting μ in Q is the query*

$$(\{?x \mapsto a, ?y \mapsto b\} \text{ AND } ((?x, p, ?z) \text{ MINUS } (?x, q, ?y))).$$

Environment-binding proposal

Kaminski et al. [44] proposed a semantics for the EXISTS clause that aims to follow the spirit of the current specification. The semantics they proposed is based on the notion of environment variables. According to this semantics, each query Q is evaluated in an RDF graph G and in a SPARQL mapping ν. The result of this evaluation is denoted $[\![Q]\!]_{G,\nu}$. The mapping ν is called the evaluation environment. The semantics of queries is defined as follows:[9]

Definition 5.5 (SPARQL semantics with environment [44])**.** *Given an RDF graph G, and SPARQL mapping ν, the semantics of a query Q is defined recursively as follows:*

1. *If Q is a basic graph pattern, then:*

$$[\![Q]\!]_{G,\nu} = \{\nu \cup \mu \mid \mu \in [\![Q]\!]_G \text{ and } \mu \sim \nu\}.$$

2. *If Q is combination of queries Q_1 and Q_2, then:*

$$\begin{aligned} [\![Q_1 \text{ UNION } Q_2]\!]_{G,\nu} &= [\![Q_1]\!]_{G,\nu} \cup [\![Q_2]\!]_{G,\nu}, \\ [\![Q_1 \text{ AND } Q_2]\!]_{G,\nu} &= [\![Q_1]\!]_{G,\nu} \bowtie [\![Q_2]\!]_{G,\nu}, \\ [\![Q_1 \text{ MINUS } Q_2]\!]_{G,\nu} &= [\![Q_1]\!]_{G,\nu} - [\![Q_2]\!]_{G,\nu}. \end{aligned}$$

[9] We fixed the definition of the semantics for the SELECT clause by Kaminski et al. [44]. In their definition solution mappings of an expression (SELECT X WHERE P) were not restricted to the domain X. The important point from their definition is that the environment is restricted when evaluating the query P.

3. *If Q has the form* (SELECT $\mathcal{X}$ WHERE P) *then:*

$$[\![Q]\!]_{G,\nu} = \{\mu|_X \mid \mu \in [\![P]\!]_{G,\nu|_X}\}.$$

4. *If Q is has the form P* FILTER φ *then the answers of Q are all solutions of $[\![P]\!]_{G,\nu}$ such that the truth value of φ against $\mu \backsim \nu$ and G, denoted $(\mu \backsim \nu)(\varphi, G)$, is true. This truth value defined recursively as follows:*

   (a) *If φ is a clause of the form* EXISTS(P) *then the truth value of $(\mu \backsim \nu)(\varphi, G)$ is true if and only if $[\![P]\!]_{G,\mu\backsim\nu}$ has an answer.*

   (b) *If φ has the form $t_1 = t_2$,* bound(X), *or* isBlank(t) *then $(\mu \backsim \nu)(\varphi, G) = (\mu \backsim \nu)(\varphi)$ where the right side denotes the truth value stated by Definition 2.10.*

   (c) *If φ has is a Boolean combination of filter-conditions then the truth value of $(\mu \backsim \nu)(\varphi, G)$ is the usual for 3-valued logic (see Table 2.1).*

The proposal by Kaminski et al. [44] described in Definition 5.5, called *environment-binding proposal* in what follows, does not consider the VALUES nor the BIND operators. However, it can be easily extended to cover them.

The deep-binding and the environment-binding proposals, join the current mapping to the basic graph patterns of the inner query. Unlike the deep-binding proposal, the environment-binding proposal also substitutes variables in filter-conditions.

Example 5.12. *Let $G = \{(a, p, b)\}$ be an RDF graph, and Q be the following query:*

$$\{?x \mapsto a, ?z \mapsto b\} \text{ FILTER (EXISTS(((?x, } p, ?y) \text{ FILTER } (?x = c \vee ?y = ?z)))).$$

Then, according to the deep-binding proposal the inner query is instantiated by the values produced in the outer query before being evaluated. This substitution produces the following query:

$$(\{?x \mapsto a\} \text{ AND } (?x, p, ?y)) \text{ FILTER } (?x = c \vee ?y = ?z). \tag{5.1}$$

The result of evaluating the graph pattern on the left side of the filter in query (5.1) has as its unique answer the mapping $\{?x \mapsto a, ?z \mapsto b\}$. Thus, the filter condition on the right side of the filter in (5.1) is evaluated against this mapping. $?x = c$ evaluates as false and $?y = ?z$ evaluates as unknown because $?z$ is unbound. Then this solution is dropped and the whole query Q has no answers in G.

According to the environment-binding proposal, the inner query is evaluated against the environment $\nu = \{?x = a, ?z = b\}$. Then, $[\![(?x, p, ?y)]\!]_{G,\nu}$ has a unique solution $\mu = \{?x \mapsto a, ?z \mapsto b\}$. The evaluation of the filter-condition $?x = c \vee ?y = ?z$ is done against the mapping $\nu \backsim \mu = \{?x \mapsto a, ?y \mapsto b, ?z \mapsto b\}$. Since $?y = ?z$ is true against $\nu \backsim \mu$, then μ is a solution of the inner query, and thus, unlike the deep-binding proposal, the evaluation of Q on G has an answer.

5.1.3 How the existing proposals solve the substituion issues

In this section we describe how each proposal described in Section 5.1.2 solves the aforementioned problems of substitution in SPARQL.

Issue 1: Top-down or bottom-up evaluation?

The shallow-binding proposal follows the bottom-up evaluation because the inner query is evaluated without substitution. On the contrary, the deep-binding and the environment-binding proposals follow the top-down evaluation.

Issue 2: Substitution can break the grammar

Since the three proposals (shallow-binding, deep-binding and environment-binding) do not replace variables by constants, but restrict the values in the inner query to agree with the values provided by the outer-query, they are not affected by this issue as the current specification is. Thus, the three proposals fix the current specification regarding this issue.

Issue 3: MINUS behaves different if solution domains are disjoint

The three proposals solved this issue. Since there is no replacement of variables by constants, the domain of answers is not reduced by the substitution, and thus the minuend and the subtrahend do not lose the common variables.

Issue 4: The scope of variables is unclear

Consider the queries

$$
\begin{aligned}
Q_1: &\quad \{?x \mapsto a\} \text{ FILTER } (\neg \text{ EXISTS } ((?y, p, ?z) \text{ FILTER } ?z = ?x)),\\
Q_2: &\quad \{?x \mapsto a\} \text{ FILTER } (\neg \text{ EXISTS } ((\text{SELECT } ?y \text{ WHERE}(?y, p, ?x)))),\\
Q_3: &\quad \{?x \mapsto b\} \text{ FILTER } (\text{EXISTS}(((?y, p, ?z) \text{ MINUS } (?z, q, ?x))))
\end{aligned}
$$

from Example 5.6.

According the shallow-binding proposal the variable ?x in these queries is not correlated because it does not occur in the answers of the inner query. Since the shallow-binding proposal asks for solutions of the inner query that are compatible with the current solution from the outer query, there is no way to connect variable ?x in the outer and inner queries.

The deep-binding proposal renames variables that are assumed to not be "visible" from the outer query. This renaming is done before doing the substitution in order to avoid the substitution of variables that are not connected. To determine what variables must be renamed, the deep-binding proposal [31] uses the notion of *in-scope*, described in the SPARQL specification [36]. In-scope variables are the variables that can occur in the output of a query (see Definition 2.12). Hence, according the deep-binding proposal the occurrences of ?x in the inner query of queries Q_1, Q_2 and Q_3 are not visible from the outer query, so they have to be renamed before applying the substitution.

The unique form to define local variables according to the environment-binding proposal is using a SELECT clause. All variables that are not listed in the SELECT are considered "local" (i.e., not visible from the outer query). The scope of variables is implemented by restricting the environment when a SELECT clause is evaluated in an inner query (see Definition 5.5). The environment-binding proposal follows this design due the instructions

given in the SPARQL 1.1 errata.[10] Under this semantics, the variable ?x is correlated in queries Q_1 and Q_3 but not in query Q_2.

Issue 5: Blank nodes are variables in basic graph patterns

The three proposals do not produce the counterintuitive answers—that according to Patel-Schneider and Martin [57] are returned when variables are substituted by blank nodes—because do not substitute variables by constants in a basic graph pattern.

Issue 6: Substitution of unbound variables

Let Q be the following SPARQL query of Example 5.8:

((?id, a, Person) OPT (?id, corpMail, ?email)) FILTER (EXISTS(
((?id, a, Person) OPT (?id, privMail, ?email)) FILTER (?email = *.com)))

and consider the following database of persons

id	1	2	3	4	5	6
corpMail	*.com	*.net	*.com	*.net		
privMail	*.net	*.com			*.com	*.net

Let us describe how each proposal resolves the question of whether person 1 is an answer of the query. In order to simplify the discussion we will denote mappings of the form $\{?\text{id} \mapsto a, ?\text{email} \mapsto b\}$ as tuples (a, b). With person 1 the outer query produces the tuple $\nu = (1, *.com)$.

According to the shallow-binding proposal, the inner query is evaluated first, producing for person 1 the tuple $\mu = (1, *.net)$. Since tuples ν and μ are incompatible, person 1 is not returned.

According to the deep-binding proposal, the variables in the mapping ν are substituted in the inner query producing the following query as the result of the substitution:

(({?id ↦ 1} AND (?id, a, person)) OPT
({?id ↦ 1, ?email ↦ *.com} AND (?id, privMail, ?email)))
FILTER ?email = *.com

Since the triple pattern (?id, privMail, ?email) has no solution compatible with the mapping $\{?\text{id} \mapsto 1, ?\text{email} \mapsto *.com\}$, this query produces the tuple $(1, -)$ that fails the filter ?email = $*.com$. Hence, person 1 is not returned.

According to the environment-binding proposal, the answers of the triple patterns (?id, a, Person) and (?id, privMail, ?email) have to be compatible with the mapping ν produced in the outer join. This result in the value ?id bound to 1 in the first triple pattern and no bindings in the second triple pattern. According to the environment-binding proposal (see Definition 5.5), given a basic graph pattern P, a graph G, and a mapping ν, the result of evaluating $[\![P]\!]_{G,\nu}$ is the set $\{\mu \cup \nu \mid \mu \in [\![P]\!]_G, \text{ and } \nu \sim \mu\}$. Intuitively, for each biding produced of matching the pattern against the graph, the result is extended with

[10] https://www.w3.org/2013/sparql-errata#errata-query-8

the bindings in the environment. Since in our example the evaluation of the first triple pattern against the graph is the mapping $\{?\text{id} \mapsto 1\}$, the result of evaluating that triple pattern against the environment ν is the tuple $\mu = (1, -) \backsim \nu = (1, *.com)$. Hence, the filter condition ?email = $*.com$ returns true, and the person 1 is returned by the query.

Let us to analyze why the deep-binding and the environment-binding proposals differ. In the case of the deep-binding the mapping $\nu = (1, *.com)$ is used to restrict the variable values in the inner query, but not in its filter-condition. Indeed, the variable values in the filter condition are given by the tuple $\mu = (1, -)$ that is less informative than tuple ν. On the contrary, the environment-binding proposal defines a clear stratification in the environments where each level increases the informativeness of the levels above. Kaminski et al. [44] proved that when a SPARQL query Q is evaluated against an environment ν and a graph G, then for each solution $\mu \in [\![Q]\!]_{g,\nu}$ it holds that $\nu \subseteq \mu$. This fact guaranties the aforementioned monotony in the information.

5.2 Environment-binding as a substitution proposal

The environment-binding proposal by Kaminski et al. [44] differs from the other proposals in that it does not generates a query result of the substitution. However, it also can be understood in terms of substitutions. In this section we will show how the environment-binding proposal can be understood in terms of substitutions.

Definition 5.6 (Environment-binding substitution)**.** *Given a SPARQL mapping ν, and a SPARQL query Q, the semantics of the resulting SPARQL query that results of the substitution* $\text{subs}(Q, \nu)$ *is defined recursively as follows:*

1. *If Q is a basic graph pattern, then:*

$$\text{subs}(Q, \nu) = (\nu \text{ AND } Q).$$

2. *If Q is combination of queries Q_1 and Q_2, then:*

$$\begin{aligned}
\text{subs}(Q_1 \text{ UNION } Q_2,\ \nu) &= (\text{subs}(Q_1, \nu) \text{ UNION } \text{subs}(Q_2, \nu)),\\
\text{subs}(Q_1 \text{ AND } Q_2,\ \nu) &= (\text{subs}(Q_1, \nu) \text{ AND } \text{subs}(Q_2, \nu)),\\
\text{subs}(Q_1 \text{ MINUS } Q_2,\ \nu) &= (\text{subs}(Q_1, \nu) \text{ MINUS } \text{subs}(Q_2, \nu)).
\end{aligned}$$

3. *If Q has the form* (SELECT $\mathcal{X}$ WHERE P) *then:*

$$\text{subs}(Q, \nu) = (\text{SELECT } \mathcal{X} \text{ WHERE } \text{subs}(P, \nu|_{\mathcal{X}})).$$

4. *If Q is has the form P* FILTER φ*, then;*

$$\text{subs}(Q, \nu) = (\text{subs}(P, \nu) \text{ FILTER } \text{subs}(\varphi, \nu)),$$

where the substitution of a filter-condition is defined recursively as follows:

   (a) *If φ is an equality formula $t_1 = t_2$ then*

$$\text{subs}(t_1 = t_2, \nu) = \nu(t_1 = t_2).$$

(b) If φ is a Boolean combination of formulas, then:

$$\begin{aligned}\text{subs}(\varphi_1 \wedge \varphi_2,\ \nu) &= \text{subs}(\varphi_1, \nu) \wedge \text{subs}(\varphi_2, \nu),\\ \text{subs}(\varphi_1 \vee \varphi_2,\ \nu) &= \text{subs}(\varphi_1, \nu) \vee \text{subs}(\varphi_2, \nu),\\ \text{subs}(\neg\varphi_1,\ \nu) &= \neg\, \text{subs}(\varphi_1, \nu).\end{aligned}$$

(c) If φ is a clause of the form EXISTS(P) *then:*

$$\text{subs}(\varphi, \nu) = \text{EXISTS}(\text{subs}(P, \nu)).$$

Example 5.13. *Let $\nu = \{?x \mapsto a, ?z \mapsto b\}$ be a SPARQL mapping, and Q be the following SPARQL query:*

$$((?x, p, ?y)\ \text{FILTER}\ (?x = c \vee ?y = ?z)).$$

The query Q is the inner query of the query in Example 5.12 and mapping ν is the mapping provided by the outer query. The result of subs(Q, ν) *then can be used to revisit Example 5.12, where the environment-binding proposal is compared with the deep-binding proposal. The following are the results of the substitution* subs(Q, ν) *according to these two proposals:*

$$((\nu|_{?x,?z}\ \text{AND}\ (?x, p, ?y))\ \text{FILTER}\ (a = c \vee ?y = b)), \qquad \text{(environment-binding)}$$
$$((\nu|_{?x}\ \text{AND}\ (?x, p, ?y))\ \text{FILTER}\ (?x = c \vee ?y = ?z)). \qquad \text{(deep-binding)}$$

Example 5.13 shows the two differences between both proposals:

1. In the environment-binding proposal the mapping ν is operated without removing variable ?z, as is done in the deep-biding proposal. This ensures that solutions μ of the pattern $(\nu|_{?x,?z}$ AND $(?x, p, ?y))$ satisfy $\nu \subseteq \mu$. In fact, Kaminski et al. [44] proved that for every query Q, database D, mapping ν, and mapping $\mu \in [\![Q]\!]_{D,\nu}$, it holds that $\nu \subseteq \mu$. Thus it is unnecessary to replace ν in the filter-condition $?x = c \vee ?y = ?z$ of the example, since subs(subs(C, μ), ν) = subs(C, μ) for every condition C.
2. Unlike the environment-binding proposal, in the deep-binding proposal variable ?z is not considered correlated. This is due to the fact that in the deep-binding proposal ν is restricted to the variables of the basic graph pattern where the substitution is applied.

Theorem 5.1. *The semantics of SPARQL according to the environment-binding proposal in Definitions 5.5 and 5.6 are equivalent.*

Proof. It can be shown by induction on the structure of the query. Let Q be an SPARQL query, G be an RDF graph, and ν be an RDF mapping. Let $\text{Eval}_A(Q, G, \nu)$ denote the result of evaluating $[\![Q]\!]_{G,\nu}$ according the semantics stated in Definition 5.5, and $\text{Eval}_B(Q, G, \nu)$ denote the result of evaluating $[\![\text{subs}(Q, \nu)]\!]_G$ according to the semantics stated in Definition 5.6.

1. If Q is a basic graph pattern P, then:

$$\begin{aligned}\text{Eval}_A(Q, G, \nu) &= \{\mu \backsim \nu \mid \mu \in [\![Q]\!]_G, \nu \sim \mu\},\\ \text{Eval}_B(Q, G, \nu) &= [\![\nu\ \text{AND}\ Q]\!]_G.\end{aligned}$$

By simple inspection, it holds that $\text{Eval}_A(Q, G, \nu) = \text{Eval}_B(Q, G, \nu)$.

2. If Q has the form (Q_1 UNION Q_2)

$$\begin{aligned}
\text{Eval}_A(Q, G, \nu) &= \text{Eval}_A(Q_1, G, \nu) \cup \text{Eval}_A(Q_2, G, \nu), \\
\text{Eval}_B(Q, G, \nu) &= [\![\text{subs}(Q, \nu)]\!]_G \\
&= [\![\text{subs}(Q_2, \nu) \text{ UNION subs}(Q_2, \nu)]\!]_G \\
&= [\![\text{subs}(Q_2, \nu)]\!]_G \cup [\![\text{subs}(Q_2, \nu)]\!]_G \\
&= \text{Eval}_B(Q_1, G, \nu) \cup \text{Eval}_B(Q_2, G, \nu),
\end{aligned}$$

   By induction, both semantics return the same answers.

3. If Q has the form (Q_1 AND Q_2) then the equivalence can be shown by the same arguments than for the UNION operator, replacing UNION by AND, and $\cup$ by $\bowtie$.

4. If Q has the form (Q_1 MINUS Q_2) then the equivalence can be shown by the same arguments than for the UNION operator, replacing UNION by MINUS, and $\cup$ by $-$.

5. If Q has the form (SELECT $\mathcal{X}$ WHERE Q_1) then

$$\begin{aligned}
\text{Eval}_A(Q, G, \nu) &= \{\mu|_{\mathcal{X}} \mid \mu \in [\![P]\!]_{G,\nu|_{\mathcal{X}}}\} \\
&= \{\mu|_{\mathcal{X}} \mid \mu \in \text{Eval}_A(P, G, \nu|_{\mathcal{X}})\}, \\
\text{Eval}_B(Q, G, \nu) &= [\![(\text{SELECT } \mathcal{X} \text{ WHERE subs}(P, \nu|_{\mathcal{X}}))]\!]_G \\
&= \{\mu|_{\mathcal{X}} \mid \mu \in \text{Eval}_B(P, G, \nu|_{\mathcal{X}})\}.
\end{aligned}$$

   By induction, both semantics return the same answers.

6. If Q has the form (Q_1 FILTER φ) then

$$\begin{aligned}
\text{Eval}_A(Q, G, \nu) &= \{\mu \in \text{Eval}_A(Q_1, G, \nu) \mid (\mu \backsim \nu)(\varphi, G) = \text{true}\}, \\
\text{Eval}_B(Q, G, \nu) &= [\![(\text{subs}(Q_1, \nu) \text{ FILTER subs}(\varphi, \nu))]\!]_G \\
&= \{\mu \in \text{Eval}_B(Q_1, G, \nu) \mid \mu(\text{subs}(\varphi, \nu)) = \text{true}\}.
\end{aligned}$$

   Since by induction $\text{Eval}_A(Q_1, G, \nu) = \text{Eval}_B(Q_1, G, \nu)$, to finish the proof we have to prove that $(\mu \backsim \nu)(\varphi, G) = \text{true}$ if and only if $\mu(\text{subs}(\varphi, \nu)) = \text{true}$. This can be shown by induction on the structure of the filter-condition φ.

   (a) If φ has the form EXISTS(P) then the formula is true according to the first semantics if and only if $\text{Eval}_A(P, G, \nu)$ has an answer whereas it is true according to the second semantics if and only if $\text{Eval}_B(P, G, \nu)$ has an answer. Since by induction $\text{Eval}_A(P, G, \nu) = \text{Eval}_B(P, G, \nu)$, it holds that the truth value of φ is the same in both semantics.

   (b) If φ has the form $t_1 = t_2$, bound(?x), or isBlank(t) then both semantics are equivalent because assign the same values to the variables occurring in φ.

   (c) If φ is a Boolean combination of filter-conditions then the truth value in both semantics is the same, because by induction is a combination of filter-conditions with the same truth values.

□

5.3 Free, bound, and range restricted variables

The variable scope in SPARQL is a source of differences in SPARQL implementations of the EXISTS clause. In this section we will contrast the variable scope in SPARQL with the variable scope in the relational calculus and SQL. The problem with the variable scope in SPARQL was briefly described in Section 5.1.1. This section describes this problem in detail.

5.3.1 Substitution in relational calculus

Substitution in first-order logic is based on the precise notion of free variable. The variables that can be substituted are called *free*, and the variables that are not free are called *bound*. Free variables are determined syntactically and the semantics of formulas relies on the definition of free variables.

Example 5.14. *Consider the following statement:*

$$\mathcal{A} \models_v q(x, y) \wedge \exists x\ p(x, y), \tag{5.2}$$

where v is the valuation $\{x \mapsto a, y \mapsto b\}$. This statement means than structure $\mathcal{A}$ models the result of subsituting variables x and y in the formula. However, not all occurrences of these variables can be substituted. In fact, variable x cannot be substituted in formula $\exists x p(x, y)$, because x is not free. By applying substitution we can infer that:

$$\mathcal{A} \models q(a, b) \wedge \exists x\ p(x, b). \tag{5.3}$$

Let Q be the first-order query

$$(r(x, y),\ q(x, y) \wedge \exists x\ p(x, y)). \tag{5.4}$$

The answers of Q against $\mathcal{A}$ are all facts of the form $r(v(x), v(y))$ where v is a valuation that makes the statement in (5.2) true. Intuitively, the free variables x and y carry the values from the query to the answers.

Example 5.14 illustrates that to compute the answers of a relational calculus query we need a precise notion of substitution. This notion is built upon the concept of free and bound variables. A variable can occur in both forms, in a formula. All free occurrences are substituted, and bound occurrences are not substituted.

5.3.2 Substitution in SQL

In SQL there are three places where a subquery can occur:

1. as a relation in the `FROM` clause;
2. as a relation with a single attribute in clauses `IN`, `SOME`, and `ALL`; and
3. as a relation in an `EXISTS` clause.

Despite the fact that this thesis does not focus on SQL, we review SQL subqueries formed by clauses `IN` and `EXISTS` to show aspects of substitution that are also present in the SPARQL EXISTS clause.

Example 5.15. *The following SQL query contains a subquery nested in a IN clause:*

```
1 SELECT orders.id
2 FROM orders
3 WHERE orders.id NOT IN (
4    SELECT items.order_id
5    FROM items
6 )
```

Like the query of Example 5.1, this query asks for orders having no items. Lines 1–3 codify the outer query, and lines 4–5 codify the inner query. Like the query of Example 5.1, the outer is not a complete query, but the inner is complete since it does not contain a correlated attribute defined in the outer query.

In the query of Example 5.15 there is no correlated attribute. Instead the value of attribute `order.id` in the outer query must not be one of the values associated to the attribute `items.order_id` in the inner query. Intuitively, there is a form of correlation between these two attributes.

SQL IN clauses can also include correlated attributes. The following example shows a query where correlation occurs.

Example 5.16. *Consider the following SQL query selecting orders without items attached to the order less than 31 days after the creation of the order:*

```
1 SELECT orders.id
2 FROM orders
3 WHERE orders.id NOT IN (
4   SELECT items.order_id FROM items
5   WHERE items.created_at - orders.created_at < 31
6 )
```

In this example the inner query must be evaluated once for each tuple of the outer query. In fact, the subquery is open because attribute `orders.created_at` *must be substituted before the evaluation of the inner query.*

Likewise relational calculus, SQL divides attributes in those that can be replaced and those that cannot be replaced. However, this notion does not coincide exactly with the variables that are free and bound in first-order formulas. As we discover, whether a variable is range restricted is also necessary to identify what variables are replaceable. To compare the two languages we will show how SQL queries can be expressed as first-order formulas.

Example 5.17. *Consider the SQL query taken from Example 5.1:*

```
1 SELECT orders.id
2 FROM orders
3 WHERE NOT EXISTS (
4    SELECT *
5    FROM items
6    WHERE orders.id != items.order_id
7 )
```

In the nested query the attribute `orders.id` *is replaceable. Furthermore, it must be substituted before evaluating the inner query by the value from the outer query. On the contrary, the attribute* `items.order_id` *is not replaceable.*

When there is no nesting, a SQL query of the form `SELECT` $\bar{X}$ `WHERE` C can be translated as a predicate calculus query of the form $(q(\bar{x}), \varphi(\bar{y}))$, where $\bar{X}$ is the list of attributes that the SQL query returns, $\bar{x}$ is the list of first-order variables corresponding to the attributes of the SQL query, φ is the first-order formula corresponding to the clause C, $\bar{y}$ are the free variables of φ, and $\bar{x} \subseteq \bar{y}$. For instance, we codify the inner query of Example 5.17 as the following first-order query:

$$(q(\bar{x}),\ \text{items}(\bar{x}) \wedge x \neq y), \tag{5.5}$$

where variables $\bar{x}$ correspond to the attributes of table `items`, x is a variable in $\bar{x}$, and variable y represents the attribute `items.order_id`.

In the formula of the query in (5.5) variables x and y are said to be free (according to the standard definition of free variable in first-order logic). This implies that both variables can be replaced in first-order logic. However, SQL states in a different way what variables can be replaced. In this case, only variable y (corresponding to an attribute defined in the outer query) can be replaced according to the SQL semantics.

Observe that the query in (5.5) is not safe because the range of variable y is not restricted. One may think therefore that in an SQL query Q the replaceable attributes correspond to the variables that are not range restricted in the corresponding relational calculus query for Q. However, in Example 5.17 we added the inequality $(x \neq y)$ intentionally to make the variable y not range restricted. If we had used an equality $x = y$ the variable y would be range restricted, and still replaceable in the corresponding SQL query. The notion of range restricted variables is thus an approximation to the criteria needed to determine what variables are replaceable in SQL. To precisely characterize the replaceable variables in SQL we require limiting the variables that are assumed to be range restricted. To do this, we can translate the SQL equality not as the standard equality $x = y$ in logic, but as a Datalog built-in $E(x, y)$ that evaluates true if x and y are equal (see Ceri et al. [18]). With this convention we can obtain a query of the form $(q(\bar{x}),\ \text{items}(\bar{x}) \wedge E(x, y))$ where y is not range restricted since $E(x, y)$ is not a regular predicate atom, but a built-in. However, built-ins are not part of the standard first-order logic. We thus require an alternative definition of range restricted variables. We next provide that definition based on a normal form, called *Safe Range Normal Form (SRNF)*, and described by Abiteboul et al. [1].

Definition 5.7 (Safe queries with the built-in equality)**.** *The function* rr *defined recursively by rules below is the one such that receives a query φ in SRNF (see Abiteboul et al. [1]) and returns a subset of the variables occurring in φ or $\bot$. In this definition we assume that for each value T in the range of* rr*, T operated with $\bot$ gives $\bot$ (e.g., $\bot \cup T = \bot$).*

1. *If φ is a predicate atom then* $\text{rr}(\varphi) = \text{var}(\varphi)$.
2. *If φ an equality atom, then* $\text{rr}(\varphi) = \varnothing$.
3. *If φ is a conjunction $\psi_1 \wedge \psi_2$ where ψ_1 and ψ_2, then* $\text{rr}(\varphi) = \text{rr}(\psi_1) \cup \text{rr}(\psi_2)$.
4. *If φ is a disjunction $\psi_1 \vee \psi_2$ then* $\text{rr}(\varphi) = \text{rr}(\psi_1) \cap \text{rr}(\psi_2)$.

5. *If* φ *is a negation* $\neg\psi$ *then* $\mathrm{rr}(\varphi) = \varnothing$.

6. *If* φ *is* $\exists\bar{X}\psi$ *then*

$$\mathrm{rr}(\varphi) = \begin{cases} \mathrm{rr}(\psi) \setminus \bar{X} & \text{if } \bar{X} \subseteq \mathrm{rr}(\psi), \\ \bot & \text{otherwise.} \end{cases}$$

We say that a query ψ *is safe if and only if* $\mathrm{rr}(\mathrm{SRNF}(\psi)) = \mathrm{free}(\psi)$. *If* $\mathrm{rr}(\mathrm{SRNF}(\psi)) \neq \bot$, *then we say that a variable* X *in* $\mathrm{free}(\psi) \setminus \mathrm{rr}(\mathrm{SRNF}(\psi))$ *is not range restricted in* ψ.

The problem of whether a variable is range restricted in a first-order formula is undecidable. Hence, like the definition described by Abiteboul et al. [1], Definition 5.7 gets an approximate response to the problem.

Lemma 5.1. *The algorithm described in Definition 5.7 to determine the range restricted variables in a first-order formula is sound but not complete.*

Proof. It follows from the fact that Definition 5.7 is based on the one described by Abiteboul et al. [1]. The difference is that our definition does not take advantage of the transitivity of the equality to allow more variables be considered range restricted. The variables that are labeled as range restricted by our algorithm is thus a subset of the variables labeled range restricted by their algorithm. □

5.3.3 Substitution in SPARQL

There is no consensus in what variables are replaceable in SPARQL. Each of the three proposals we discuss in this thesis uses a different criteria. An example of how different interpretations of the standard are, are the works of Kostylev et al. [44] and Angles and Gutierrez [7]. The former called free variables to the variables $\mathcal{X}$ in the query (SELECT $\mathcal{X}$ WHERE P). The latter assume exactly the contrary, that is, free variables are those that do not occur in $\mathcal{X}$. In this thesis we do not analyze the interpretation of Angles and Gutierrez, because it is not completely specified and because it differs in this point with all the three proposals.

The variables that are replaceable depend on the semantics used for the substitution. For the sake of the simplicity, we next present a translation from SPARQL queries to first-order formulas restricted to a reduced set of SPARQL operators.

Definition 5.8 (Translation from SPARQL to first-order queries)**.** *A SPARQL query* Q *is said to be an AFV-query if and only if:*

1. Q *is conformed by the operators* VALUES, AND *and* FILTER *with an equality or an inequality as filter condition.*

2. *If a expression* $(Q_1$ FILTER $\theta)$ *occurs in* Q *and a variable* ?x *occurs in* θ *but not in* Q_1, *then all occurrences of* ?x *in* Q *are in* θ. *Intuitively, this second condition guarantees that the name of variables coincide with their scope.*

Let Ω *be a set of mappings* $\mathcal{X}$ *be a set of variables where* $\mathrm{dom}(\mu) \subseteq \mathcal{X}$ *for every mapping* $\mu \in \Omega$, Q_1 *and* Q_2 *be two AFV-queries,* θ *be an equality or an inequality. Assume that every SPARQL variable* ?x *is associated to a first-order variable* x, *and let* $\delta(\theta)$ *denote the result of replacing each SPARQL variable in* θ *by its corresponding first-order variable. Assume a predicate* $\mathrm{equal}(\cdot,\cdot)$ *to*

denote the equality. The first-order query that corresponds to an AFV-query Q, denoted $\delta(Q)$, is defined recursively as follows:

$$\begin{aligned}\delta(\text{VALUES } \mathcal{X}\ \Omega) &= \bigvee_{\mu\in\Omega} \bigwedge_{?x\in\text{dom}(\mu)} \text{equal}(x, \mu(?x)),\\ \delta(Q_1 \text{ AND } Q_2) &= \delta(Q_1) \wedge \delta(Q_2),\\ \delta(Q_1 \text{ FILTER } \theta) &= \delta(Q_1) \wedge \delta(\theta).\end{aligned}$$

Observe that in the translation above we model the equality with the predicate equal$(\cdot,\cdot)$ instead of using the equality relation "=". We do that in order to ensure that variables that are bound in a VALUES clause are range restricted according to Definition 5.7.

Now we are ready to discuss the relation of the variables that are replaceable in a SPARQL query Q and the variables that are range-restricted in the first-order query $\delta(Q)$.

Example 5.18. *Let Q be the following SPARQL query:*

$$\{?x \mapsto a, ?y \mapsto b\} \text{ FILTER } ((?x, p, ?z) \text{ FILTER } ?z \neq ?y).$$

The translation of the inner query is the following first-order query:

$$(q(x, z), \text{triple}(x, p, z) \wedge z \neq y). \tag{5.6}$$

Variables ?x *and* ?z *are range restricted in (5.6) according to Definition 5.7, and variable* ?y *is not.*

To evaluate Q we have to substitute the mapping $\nu = \{?x \mapsto a, ?y \mapsto b\}$ in the inner query. The substitution depends on the semantics followed. We have therefore three alternative queries resulting from the substitution according the three proposals we study herein:

shallow-binding:	$\nu\|_{?x,?y}$ AND $((?x, p, ?z)$ FILTER $?z \neq ?u)$,	(5.7)
deep-binding:	$(\nu\|_{?x}$ AND $(?x, p, ?z))$ FILTER $?z \neq ?y$,	(5.8)
environment-binding:	$(\nu\|_{?x,?y}$ AND $(?x, p, ?z))$ FILTER $?z \neq b$.	(5.9)

Observe that ?y *is replaced by* ?u *in (5.7) to get an AFV-query. This replacement does not change the semantics of the query. As these queries are AFV-queries, we can apply the translation δ on them. The translation produces the following first-order queries:*

$$\text{equal}(x, a) \wedge \text{equal}(y, b) \wedge (\text{triples}(x, p, z) \wedge z \neq u), \tag{5.10}$$
$$(\text{equal}(x, a) \wedge \text{triples}(x, p, z)) \wedge z \neq y, \tag{5.11}$$
$$(\text{equal}(x, a) \wedge \text{equal}(y, b) \wedge \text{triples}(x, p, z)) \wedge z \neq b. \tag{5.12}$$

Queries (5.7) and (5.10) are not equivalent. In fact, the filter-condition in (5.7) returns always error because ?y *is unbound in it. On the contrary, in (5.10) variable y is non-range restricted.*

Despite the fact that translation δ does not produce equivalent queries, it provides some intuition on the relation among the replaceable variables in a SPARQL inner query Q and the variables that are range-restricted in $\delta(Q)$. According to the shallow-binding and deep-binding proposals, only variables that are free and range restricted in (5.6) are replaceable (i.e., x and z). On the other hand, according to the environment-binding proposal, all free variables can be replaced (i.e., x, y, and z).

Language	Free variables	
	Range restricted	Non-range restricted
SQL		X
SPARQL (shallow-binding)	X	
SPARQL (deep-binding)	X	
SPARQL (environment-binding)	X	X

Table 5.1: Replaceable variables in queries that are nested EXISTS clauses according to SQL and three alternative semantics of SPARQL. Free variables are divided in range restricted and non-range restricted.

5.4 Conclusions

The findings of what variables are replaceable in relational calculus, SQL and SPARQL are summarized in Table 5.1. However, in the case of SPARQL the results of Example 5.18 are far from being conclusive. This is because that, even in a reduced fragment, unbound values make it difficult to define a simple and intuitive translation from SPARQL to predicate calculus that produce equivalent queries. In fact, translations from SPARQL to Datalog are possible [60, 8, 66] but complicated by the processing of unbound values.

The lesson we learned from SQL is that (at least in this example) replaceable variables do not correspond to free variables of relational calculus, but to variables that are not range restricted according to Definition 5.7. The motivation of the SQL design is to make unambiguous the source of the value for each attribute. In SQL the value of each attribute is given in a specific FROM clause. On the contrary, in SPARQL the source of the value of a variable can be ambiguous. For instance, this ambiguity occurs in the last occurrence of variable y in the first-order query (5.6). The variable y is substituted by b in (5.12) but y also is restricted to be equal to b by the atom equal(y, b). Recall that in Example 5.8 we showed that the question of what is the source of the value of a variable gets relevant when some of these sources can produce unbound values.

Chapter 6

Expressing SPARQL in Datalog

In the previous Chapter we presented a translation from SPARQL to first-order queries in order to explore the relation of replaceable and range restricted variables. However, the translation proposed in Definition 5.8 is not completely faithful due to the peculiar behavior of unbound variables. In this chapter, in order to use a logical foundation to study SPARQL, we use another approach, namely the translation from SPARQL to Datalog by Angles and Gutierrez [8] adapted to the SPARQL fragment we study herein (see Definition 2.8). The clause EXISTS, due to the particularities it poses, is addressed in the next chapter.

Translations from SPARQL to Datalog have been studied by several authors. Polleres [60] proved that the SPARQL fragment including the operators AND, UNION, FILTER, OPT, and MINUS is expressible in Answer Set Programming (an extension of Datalog), by providing an equivalent Answer Set Programming query for each query in that SPARQL fragment. Schenk [66] provides a translation of queries in the SPARQL fragment including the operators AND, UNION, FILTER, and OPT to Datalog in order to study the complexity of SPARQL. Actually, Schenk translations does not directly generate Datalog, but first order formulas that can be translated to Datalog using the Lloyd-Torop transformation [51]. Angles and Gutierrez [8] define a translation from the same SPARQL fragment as Schenk to Datalog in order to study the expressive power of SPARQL. Actually, these three translations consider also the operator GRAPH that we do not consider in this thesis.

In this thesis, we use the translation by Angles and Gutierrez because it directly transforms SPARQL queries to the least expressive language, whereas to generate Datalog, the other two translations require a transformation afterwards. The election of a particular translation does not affect the result our analysis of the SPARQL EXISTS operator because, as we show in the end of this chapter, these three translations produce mostly the same Datalog programs.

The goal of this chapter is to extend, improve and fix errors in the the translation by Angles and Gutierrez [8]. We identify families of queries where the translation did not work and fix them. We also extend their translation with the operators MINUS, VALUES and BIND, that they did not consider because it was made before the introduction of these operators to the standard.

Throughout this chapter we use the names SPARQL-0 and SPARQL-1 to denote the fragments described in Definition 2.8. SPARQL-0 consists of queries that include the operators AND, UNION, FILTER, and OPT whereas SPARQL-1 consists of queries that include also the operators SELECT, MINUS, VALUES, and BIND.

Organization of this chapter

- In Section 6.1, we present an overview of nr-Datalog$^\neg$.
- In Section 6.2, we present the translation by Angles and Gutierrez [8] for AUFO queries. Their translation is in general correct, except two problems that we describe and fix in the following two sections.
- In Section 6.3, we describe and fix the problem of the translation by Angles and Gutierrez [8] with equality atoms in filter-conditions.
- In Section 6.4, we describe and fix the problem of the translation by Angles and Gutierrez [8] with negation in filter-conditions.
- In Section 6.5, we extend the fixed translation to AUSFODVB queries.
- We conclude this chapter in Section 6.6.

6.1 An overview of nr-Datalog$^\neg$

In this section we provide a short description of non-recursive Datalog with safe negation (nr-Datalog$^\neg$ in what follows). We follow the formalization presented by Levene and Loizou [46]. More details about nr-Datalog$^\neg$ can be found in the works of Abiteboul et al. [1], and Ceri et al. [18].

The nr-Datalog$^\neg$ syntax is defined assuming three disjoint sets: *variables*, *constants* and *predicate names*.[1] A *term* is either a variable or a constant. An *atom* is either an expression $p(t_1, \ldots, t_n)$ where p is a predicate name and each t_i is a term, called predicate atom, or an expression $t_1 = t_2$ where t_1 and t_2 are terms, called equality atom. A *literal* is either an atom (i.e. a *positive literal* A) or the negation of an atom (i.e. a *negative literal* $\neg A$). We write $t_1 \neq t_2$ as an abbreviation for $\neg(t_1 = t_2)$.

A *rule* is an expression $L_{n+1} \leftarrow L_1, \ldots, L_n$ where L_{n+1} is a positive predicate atom with no constants called the head, and $L_1, \ldots, L_n$ is a set of literals called the *body*. A *program* Π is a finite set of rules. A *database* D is a finite multiset of atoms without variables.

A variable X occurs positively in a rule R if and only if X occurs in a positive predicate atom in the body of R, in a positive equality atom of the form $X = c$ where c is a constant, or in a positive predicate atom of the form $X = Y$ where Y is variable occurring positively in R. A rule R is said to be *safe* if all the variables occurring in R occur positively in R. A program Π is *safe* if all the rules of Π are safe.

[1] The syntax of SPARQL and nr-Datalog$^\neg$ are defined using arbitrary disjoint sets. To simplify the translation from SPARQL to nr-Datalog$^\neg$ we assume that the set of variables is $\mathbf{V}$ (i.e., the set of SPARQL variables), and the set of constants is $\mathbf{B} \cup \mathbf{I} \cup \mathbf{L}$ (i.e., the set of constants in SPARQL).

The *dependency graph* of a program Π is a digraph (N, E) where the set of nodes N is the set of predicates that occur in the literals of Π, and there is an arc (p_1, p_2) in E if there is a rule in Π whose body contains predicate p_1, and whose head contains predicate p_2. A program is said to be *non-recursive* if its dependency graph is acyclic.

In what follows, restrict programs to safe and non-recursive Datalog programs, that, among other good properties, in their evaluation always return a finite set of facts [1, 18].

Programs are defined over databases having a fixed schema, called the *vocabulary*. A vocabulary τ is a pair (P, α) where P is a finite set of predicate names and α is a function defining an arity for each predicate name. An atom is extensional if its predicate name is in P, otherwise is called intensional. A database over vocabulary (P, α) is a multiset of facts $p(a_1, \ldots, a_n)$ where p is extensional and $\alpha(p) = n$. A query over a vocabulary τ is a pair (L, Π) where L is a positive predicate atom, called *the goal*, and Π is a program where the head of every rule is an intensional atom.

We assume a proof-theoretic semantics for nr-Datalog$^\neg$ programs.

A *substitution* is a partial function θ from variables to constants. Hence, a nr-Datalog$^\neg$ answer is a set of substitutions with the same domain.

Let D be a database and Π a program. The *derivation trees* of Π with respect to D, denoted $\mathrm{dt}(\Pi, D)$, are defined recursively as follows:

1. For each fact $f \in D$, there is a derivation tree in $\mathrm{dt}(\Pi, D)$ having f as unique node.
2. Let $R \in \Pi$ be a rule of the form $L_{n+1} \leftarrow L_1, \ldots, L_m, L_{m+1}, \ldots, L_n$ where $L_1, \ldots, L_m$ are positive predicate literals, and $L_{m+1}, \ldots, L_n$ are not positive predicate literals, θ be a substitution such that the domain of θ is the set of variables occurring in R, and $(t_1, \ldots, t_m)$ be a tuple of derivations trees in $\mathrm{dt}(\Pi, D)$ such that conditions hold for $1 \leq i \leq n$:
   (a) If L_i is a positive predicate atom then the root of t_i is $\theta(L_i)$.
   (b) If L_i is a negative predicate atom then there is no derivation tree $t_i \in \mathrm{dt}(\Pi, D)$ such that the root of t_i is $\theta(L_i)$.
   (c) If L_i is a positive equality atom then $\theta(L_i)$ is an equality of the form $a = a$ where a is a constant.
   (d) If L_i is a negative equality atom then $\theta(L_i)$ is an inequality of the form $a \neq b$ where a and b are different constants.

   If the conditions above hold then there is a derivation tree $t \in \mathrm{dt}(\Pi, D)$ consisting of:
   - Root $\theta(L_{n+1})$;
   - Ordered edges $\theta(L_{n+1}) \rightarrow r_i$ with label R, where for $1 \leq i \leq m$, r_i is the root of t_i and for $j > m$, $r_j = \theta(L_j)$.
3. No more derivations trees are in $\mathrm{dt}(\Pi, D)$.

Let Π be a program, D a database and f a fact. A tree $t \in \mathrm{dt}(\Pi, D)$ is said to be a proof for f if the root of t is f.

Given a nr-Datalog$^\neg$ query $Q = (L, \Pi)$ over a vocabulary τ, and a database D over τ, we write $[\![Q]\!]_D$ to denote the set of substitutions θ such that $\mathrm{dom}(\theta)$ is the set of variables occurring in L and there is a proof for the fact $\theta(L)$ in $\mathrm{dt}(P, D)$.

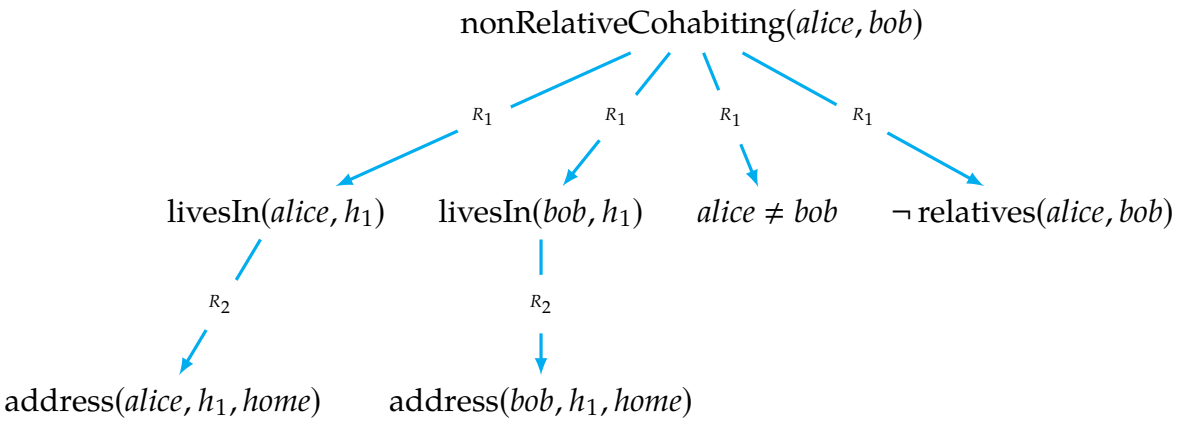
Example 6.1. *Let D be the database consisting in the following facts:*

$$\begin{array}{lll} \mathrm{address}(alice, h_1, home), & \mathrm{address}(alice, h_2, work), & \mathrm{address}(bob, h_1, home), \\ \mathrm{address}(carl, h_1, home), & \mathrm{address}(daisy, h_3, home), & \mathrm{relative}(alice, carl), \end{array}$$

and Π be the program that computes non-relative cohabiting people and consists in the following rules:

$$R_1 : \mathrm{nonRelativeCoahabiting}(X, Y) \leftarrow \mathrm{livesIn}(X, Z), \mathrm{livesIn}(Y, Z), X \neq Y, \neg\, \mathrm{relative}(X, Y)$$
$$R_2 : \mathrm{livesIn}(X, Y) \leftarrow \mathrm{address}(X, Y, home)$$

The proof for the fact $\mathrm{nonRelativeCohabiting}(alice, bob)$ *in* $\mathrm{dt}(\Pi, D)$ *is the derivation tree depicted as follows:*



6.2 Translation of the safe SPARQL-0 fragment

The fragment of SPARQL consisting of the operators AND, UNION, FILTER, and OPT (SPARQL-0 in short), is at the core of SPARQL. In what follows we present its translation to Datalog. The "safe" adjective refers to the allowed scope of the variables inside filter fragments.

According Perez et al. [59], given a query (Q FILTER φ), it is desirable for reasons of simplicity for the user and cleanness of the semantics, that the scope of the variables inside φ should be the expression Q which φ filters. This notion is referred as *safe FILTER* and defined by Angles and Gutierrez [8] as follows:

Definition 6.1 (Safe SPARQL-0 query [8])**.** *A query Q in the SPARQL-0 fragment is safe if for each filter query (R FILTER φ) occurring in Q it holds that* $\mathrm{var}(\varphi) \subseteq \mathrm{var}(R)$.

Note on unsafe SPARQL queries. Angles and Gutierrez [8] showed that the SPARQL-0 fragment (that is, including unsafe queries) is expressible in nr-Datalog$^\neg$ by providing first, a translation from unsafe queries to safe queries, and then, from safe

SPARQL-0 queries to nr-Datalog$^\neg$. Observe that in Definition 6.1 var(R) denotes the variables that occur in the query, and because the fragment does not include the operators SELECT nor MINUS, var(R) coincides with dom(R) (see Definition 2.12). By introducing the SELECT operator into the SPARQL algebra, the translation of unsafe to safe queries results straightforward. Given an unsafe SPARQL query (R FILTER φ), the query ((SELECT dom(R) $\cup$ var(φ) R) FILTER φ) is safe and equivalent to the previous one.

In this section we present their translation from safe SPARQL-0 queries to nr-Datalog$^\neg$. The translation of SPARQL to nr-Datalog$^\neg$ requires first translating each SPARQL Dataset (i.e., RDF graph) into a set of facts. The following definition describes this translation.

Definition 6.2 (RDF dataset as nr-Datalog$^\neg$ facts [8])**.** *Given an RDF graph G,* facts(G) *is the set with*

- *a fact* triple(s, p, o) *for each triple* $(s, p, o) \in G$*, and*
- *a fact* bn($\perp$) *for each blank node* $\perp$ *occurring in* G*.*

In order to express SPARQL using nr-Datalog$^\neg$ it is needed to translate the output of SPARQL queries —set of mappings— into the output of relational algebra queries — relations. A natural translation consists in viewing mappings as tuples under the named perspective. The domain of a mapping μ is thus dom(μ). The difference between set of mappings and relations is that mappings have no unique domain, but tuples in a relation do have. Thus, mappings must be filled with the unbound value $\varnothing$ using the method described in Definition 2.7.

Angles and Gutierrez [8] uses a special predicate atom comp(X, Y, Z) to codify the compatibility between values Y and Z. Since unbound values are filled with the special constant $\varnothing$, two terms are said to be compatible if they are the same constant or if one of them is $\varnothing$. The program defining the compatibility is the following:

$$\text{comp}(X, X, X) \leftarrow \text{adom}(X) \tag{6.1}$$
$$\text{comp}(X, X, Y) \leftarrow \text{adom}(X), Y = \varnothing \tag{6.2}$$
$$\text{comp}(X, Y, X) \leftarrow \text{adom}(X), Y = \varnothing \tag{6.3}$$
$$\text{comp}(Y, Y, Y) \leftarrow Y = \varnothing \tag{6.4}$$

The predicate atom adom(a) codifies the fact that constant a is in the active domain of the database. A graph database is modeled with a single ternary relation with predicate triple so the active domain of the database is obtained by the following program:

$$\text{adom}(X) \leftarrow \text{triple}(X, Y, Z) \tag{6.5}$$
$$\text{adom}(X) \leftarrow \text{triple}(Y, X, Z) \tag{6.6}$$
$$\text{adom}(X) \leftarrow \text{triple}(Y, Z, X) \tag{6.7}$$

Definition 6.3 (Compatibility program [8])**.** *We write Π_{comp} to denote the nr-Datalog$^\neg$ program with the rules specified in (6.1) to (6.7). Thus, Π_{comp} its the program to check the compatibility.*

We next present the translation of the SPARQL-0 fragment to nr-Datalog$^\neg$.

Definition 6.4 (SPARQL-0 to nr-Datalog$^\neg$ translation [8])**.** *In the definition of each rule we will assume the following conventions:*

(i) *Given a set of SPARQL variables $\mathcal{X}$ and a predicate p, then $p(\bar{X})$ denotes the predicate atom with predicate p and where $\bar{X}$ is the list of the Datalog variables that corresponds to the variables in $\mathcal{X}$ in lexicographic order.*

(ii) *Q, R, S are SPARQL-0 queries. In all rules, the query Q is the query whose translation is being defined. In some cases the query Q is defined using queries R and S as parameters (i.e., R and S occur in Q).*

(iii) *$\bar{X}$, $\bar{Y}$, $\bar{Z}$, and $\bar{W}$ are lists of Datalog variables corresponding to the sets of SPARQL variables where $\mathcal{X} = \mathrm{dom}(Q)$, $\mathcal{Y} = \mathrm{dom}(R)$, $\mathcal{Z} = \mathrm{dom}(S)$, and $\mathcal{W} = \mathcal{Y} \cap \mathcal{Z}$. These sets of variables are recursively defined in Definition 2.12.*

(iv) *The result of translating Q, R, and S are respectively $(q(\bar{X}), \Pi_Q)$, $(r(\bar{Y}), \Pi_R)$, and $(s(\bar{Z}), \Pi_S)$. The rules we present here will be of the form $Q \mapsto \Pi_Q$ where $q(\bar{X})$ is the head of at least one of the rules of Π_Q. The values of the nr-Datalog$^\neg$ queries $(r(\bar{Y}), \Pi_R)$ and $(s(\bar{Z}), \Pi_S)$ are given recursively.*

(v) *Given a finite set of variables $\bar{V} \subset \mathbf{V}$ a predicate of the form $p(\bar{V})$ denotes the predicate atom resulting from writing between the parenthesis the variables in the set $\bar{V}$ in lexicographic order. This assumption is needed because attributes in SPARQL mappings have no order, while values in predicate atoms have.*

(vi) *$h_Y : \bar{W} \to V_Y$ and $h_Z : \bar{W} \to V_Z$ are two bijective functions preserving the lexicographic order, where V_Y and V_Z are two disjoint sets of fresh variables. In other words, these functions rename common variables between queries R and S by distinct fresh variables.*

(vii) *We write $\mathrm{Comp}(\bar{W})$ to denote the list of literals of the form $\mathrm{comp}(X, h_Y(X), h_Z(X))$ for each variable $X \in \bar{W}$. For instance, if $\bar{W} = \{W_1, W_2\}$ then*

$$\mathrm{Comp}(\bar{W}) = \mathrm{comp}(W_1, h_Y(W_1), h_Z(W_1)), \mathrm{comp}(W_2, h_Y(W_2), h_Z(W_2)).$$

(viii) *Given a partial function $h : \mathbf{V} \to \mathbf{V}$ and a set of variables $\bar{V} \subset \mathbf{V}$ then $h(\bar{V})$ denotes the result of replacing in $\bar{V}$ each variable $X \in \mathrm{dom}(h)$ by $h(X)$.*

The translation of a SPARQL-0 query Q into a nr-Datalog$^\neg$ query is defined recursively as follows:

1. Triple Patterns: *Let $Q = (t_1, t_2, t_3)$ be a triple pattern without blank nodes. Then:*

$$(t_1, t_2, t_3) \quad \mapsto \quad q(\bar{X}) \leftarrow \mathrm{triple}(t_1, t_2, t_3).$$

   The case when a triple pattern has blank nodes is not needed because a triple pattern is always inside a basic graph pattern, and the rule that translates basic graph patterns replaces blank nodes by variables before translating each triple pattern.

2. Basic Graph Patterns: *Let Q be the basic graph pattern $\{T_1, \ldots, T_n\}$, and $\{T'_1, \ldots, T'_n\}$ be the result of replacing consistently every blank node occurring in Q by a fresh variable. Then, the translation of Q to nr-Datalog$^\neg$ is done with the following rule:*

$$\{T_1, \ldots, T_n\} \quad \mapsto \quad q(\bar{X}) \leftarrow t_1(\bar{X}_1), \ldots, t_n(\bar{X}_n) \ ; \ \Pi_1 \ ; \ \ldots \ ; \ \Pi_n$$

   where the nr-Datalog$^\neg$ query $(t_i(\bar{X}_i), \Pi_i)$ is the translation of T'_i, for $1 \leq i \leq n$. Observe that $\bar{X}$ does not include the fresh variables that replaced the blank nodes occurring in the basic graph pattern, so those variables are excluded from $\mathrm{dom}(Q)$.

3. Filter: *Let $Q = (R \text{ FILTER } \varphi)$ be an SPARQL-0 query where φ is a built-in condition. Then, the translation of Q is given by the following rules:*

$$\begin{aligned}
(R \text{ FILTER } t_1 = t_2) &\mapsto q(\bar{X}) \leftarrow r(\bar{Y}), t_1 = t_2 \ ; \ \Pi_R \\
(R \text{ FILTER isBlank}(t)) &\mapsto q(\bar{X}) \leftarrow r(\bar{Y}), \text{bn}(t) \ ; \ \Pi_R \\
(R \text{ FILTER bound}(X)) &\mapsto q(\bar{X}) \leftarrow r(\bar{Y}), X \neq \varnothing \ ; \ \Pi_R \\
(R \text{ FILTER } \neg\varphi) &\mapsto q(\bar{X}) \leftarrow r(\bar{Y}), \neg r'(\bar{Y}) \ ; \ \Pi_R \ ; \ \Pi'_R \\
(R \text{ FILTER } \varphi_1 \wedge \varphi_2) &\mapsto q(\bar{X}) \leftarrow r_1(\bar{X}), r_2(\bar{X}) \ ; \ \Pi_1 \ ; \ \Pi_2 \\
(R \text{ FILTER } \varphi_1 \vee \varphi_2) &\mapsto q(\bar{X}) \leftarrow r_1(\bar{X}) \ ; \ q(\bar{X}) \leftarrow r_2(\bar{X}) \ ; \ \Pi_1 \ ; \ \Pi_2
\end{aligned}$$

where the nr-Datalog$^\neg$ queries $(r'(\bar{Y}), \Pi'_R)$, $(r_1(\bar{Y}), \Pi_1)$, and $(r_2(\bar{Y}), \Pi_2)$ are the respective translations for the SPARQL queries $(R \text{ FILTER } \varphi)$, $(R \text{ FILTER } \varphi_1)$, and $(R \text{ FILTER } \varphi_2)$.

4. And: *If Q is an AND query then Q is translated using the following rule:*

$$(R \text{ AND } S) \quad \mapsto \quad q(\bar{X}) \leftarrow r(h_Y(\bar{Y})), s(h_Z(\bar{Z})), \text{Comp}(\bar{W}) \ ; \ \Pi_R; \ \Pi_S \ ; \ \Pi_{\text{comp}}$$

The main idea of this translation is to rename all common variables X in predicate atoms $r(\bar{Y})$ and $s(\bar{Z})$ using functions h_Y and h_Z to then check the compatibility of the values that were assigned to variable X. Without that renaming, the values assigned to X in both predicate atoms would be forced to be equal, instead of compatible as occurs in the AND operator.

5. Union: *If Q is a UNION query then Q is translated using the following rule:*

$$\begin{aligned}
(R \text{ UNION } S) \quad \mapsto \quad & q(\bar{X}) \leftarrow r(\bar{Y}), X_1^r = \varnothing, \ldots, X_n^r = \varnothing \\
& q(\bar{X}) \leftarrow s(\bar{Z}), X_1^s = \varnothing, \ldots, X_m^s = \varnothing \\
& \Pi_R; \ \Pi_S
\end{aligned}$$

where $\{X_1^r, \ldots, X_n^r\} = \bar{X} \setminus \bar{Y}$ and $\{X_1^s, \ldots, X_m^s\} = \bar{X} \setminus \bar{Z}$. Values $\varnothing$ are added to fill in non common attributes of tuples.

6. Optional: *If Q is an OPT query then it is translated using the following rule:*

$$\begin{aligned}
(R \text{ OPT } S) \quad \mapsto \quad & q(\bar{X}) \leftarrow r(h_Y(\bar{Y})), s(h_z(\bar{Z})), \text{Comp}(\bar{W}) \\
& q(\bar{X}) \leftarrow r(h_Y(\bar{Y})), \neg t(h_Z(\bar{W})), \text{Comp}(\bar{W}), X_1 = \varnothing, \ldots, X_m = \varnothing \\
& t(\bar{W}) \leftarrow s(\bar{Z}) \\
& \Pi_R \ ; \ \Pi_S \ ; \ \Pi_{\text{comp}}
\end{aligned}$$

where $\{X_1, \ldots, X_m\} = \bar{Z} \setminus \bar{W}$ and t is a fresh predicate introduced to ensure safeness. As in the translation of operator AND, compatibility is checked. Values $\varnothing$ are added to fill attributes in $\text{dom}(S) \setminus \text{dom}(R)$ in solution mappings of R with no compatible solution mappings in S.

Angles and Gutierrez [8] define the translation of a SPARQL query Q as the nr-Datalog$^\neg$ query $(q(\bar{X}), \Pi)$ where the program Π includes both the program that computes the query (described in Definition 6.4) and the database corresponding to the RDF graph. Unlike Angles and Gutierrez, we choose to maintain the code of the query separate from the database where the query is evaluated. This design allows a simpler checking of whether the translation of a query is correct, independently of the database given.

Given a SPARQL query Q whose translation is the nr-Datalog$^\neg$ query $q(\bar{X})$, the evaluation of Q is a set of mappings and the evaluation of $q(\bar{X})$ is a set of facts. Also, the database where Q is evaluated is an RDF graph, and the database where $q(\bar{X})$ is evaluated is a set of facts. In order to compare the semantics of SPARQL queries and nr-Datalog$^\neg$ queries, Definition 6.5 below defines the answers of a nr-Datalog$^\neg$ query over an RDF graph in terms of Datalog mappings.

Definition 6.5 (Evaluation of a nr-Datalog$^\neg$ query on an RDF graph)**.** *Given a nr-Datalog$^\neg$ query $q(\bar{X})$ and an RDF graph G, we write $[\![q(\bar{X})]\!]_G$ to denote the set of facts $[\![q(\bar{X})]\!]_{D_G}$ where D_G is the Datalog database correspoding to the graph G (see Definition 6.2).*

By an abuse of notation, given a nr-Datalog$^\neg$ query $q(\bar{X})$, an RDF graph G, and a SPARQL mapping μ where $\mathrm{dom}(\mu) \subseteq \bar{X}$*, we say that μ is a solution of $[\![q(\bar{X})]\!]_G$, denoted $\mu \in [\![q(\bar{X})]\!]_G$, to mean that the fact* $\mathrm{fill}(\mu, \mathcal{X})(q(\bar{X}))$ *is a solution of $[\![q(\bar{X})]\!]_G$. Recall that* $\mathrm{fill}(\mu, \bar{X})$ *is the mapping that results from extending the domain of μ to $\bar{X}$ by adding the symbol $\varnothing$ (see Definition 2.7).*

Now we can define when a translation from SPARQL to nr-Datalog$^\neg$ is correct.

Definition 6.6 (Correct translation from SPARQL to nr-Datalog$^\neg$)**.** *A translation function δ from SPARQL to nr-Datalog$^\neg$ is correct if and only if for all SPARQL query Q the following properties hold:*

1. *$\delta(Q)$ is a nr-Datalog$^\neg$ query of the form $q(\bar{X})$ where $\bar{X} = \mathrm{dom}(Q)$.*
2. *For all SPARQL mapping μ with* $\mathrm{dom}(\mu) \subseteq \bar{X}$*, and all RDF graphs G, $\mu \in [\![Q]\!]_G$ if and only if $\mu \in [\![q(\bar{X})]\!]_G$.*

Thus, we have the following theorem:

Theorem 6.1 (Angles and Gutierrez [8])**.** *The translation described in Definition 6.4, is correct for all queries in the safe SPARQL-0 fragment restricted to queries having neither equality atoms nor negations inside FILTER conditions.*

Proof. The proof is given by Angles and Gutierrez [8] using induction in the structure of queries in the SPARQL-0 fragment. □

Observe that the translation described in Theorem 6.1 explicitly excludes equality formulas and negations in FILTER conditions. In Section 6.3 and Section 6.4, we address the problems translation in Definition 6.4 has with equality formulas and negation respectively, and extend the translation to include them.

6.3 Fixing the translation of equality atoms in filter-conditions

The problem of the Angles and Gutierrez [8] translation (see Definition 6.4) with equality formulas in built-in SPARQL conditions is illustrated in the following example.

Example 6.2. *Let $G = \{(a, p, b)\}$ be an RDF graph, and Q be the SPARQL query*

$$(R \text{ FILTER } Y = Z).$$

where R is the following SPARQL query:

$$(((X, p, b) \text{ OPT } (X, q, Y)) \text{ OPT } (X, r, Z)).$$

Then,

$$[\![R]\!]_G = \{\{X \mapsto a\}\},$$
$$[\![Q]\!]_G = \{\}.$$

Solution $\{X \mapsto a\}$ is dropped by the filter-condition because $Y = Z$ raises an error when Y or Z are unbound.

According to Definition 6.4, the translation of Q to datalog is the query $(q(X, Y, Z), \Pi_q)$ where Π_Q the following nr-Datalog$^\neg$ program:

$$q(X, Y, Z) \leftarrow r(X, Y, Z), Y = Z \ ; \ \Pi_R,$$

where $(r(X, Y, Z), \Pi_R)$ is the translation of query R.

If we assume that translation of R is correct, then:

$$[\![(r(X, Y, Z), \Pi_R)]\!]_G = \{r(a, \varnothing, \varnothing)\},$$
$$[\![(q(X, Y, Z), \Pi_Q)]\!]_G = \{r(a, \varnothing, \varnothing)\}.$$

Here, the fact $q(a, \varnothing, \varnothing)$ is not dropped because $\varnothing = \varnothing$ is true according to the semantics of equality formulas in nr-Datalog$^\neg$.

The problem described in Example 6.2 is solved by ensuring that terms in the equality are bound. This idea is formalized in the following Definition.

Definition 6.7 (Translation of FILTER queries with equality formulas)**.** *Given a SPARQL query Q of the form (R FILTER $X = Y$), the translation of Q is given by the following rule:*

$$(R \text{ FILTER } X = Y) \quad \mapsto \quad q(\bar{X}) \leftarrow r(\bar{X}), X = Y, X \neq \varnothing, Y \neq \varnothing \ ; \ \Pi_R$$

where $\bar{X} = \text{dom}(Q) = \text{dom}(R)$ and $(r(\bar{X}), \Pi_R)$ is the translation of query R.

The correctness of Definition 6.7 is stated in the following theorem.

Theorem 6.2. *The translation resulting from replacing in Definition 6.4 the rule for equality atoms in filter-conditions by the rule provided by Definition 6.7, is correct for all queries in the safe SPARQL-0 fragment restricted to queries having no negations inside FILTER conditions.*

Proof. We will prove this theorem by induction on the structure of queries, assuming that all other rules of the translation are correct.

Let $Q = (R \text{ FILTER } X = Y)$ be a safe SPARQL query and G be a graph pattern. According to Definition 6.7 the translation of query Q is a nr-Datalog$^\neg$ query of the form $q(\bar{X})$ over the following program:

$$q(\bar{X}) \leftarrow r(\bar{X}), X = Y, X \neq \varnothing, Y \neq \varnothing \ ; \ \Pi_R$$

where $\bar{X} = \text{dom}(Q) = \text{dom}(R)$ and $(r(\bar{X}), \Pi_R)$ is the translation of the SPARQL query R.

Let μ be a SPARQL mapping, and μ' be fill$(\mu, \bar{X})$. By induction, $\mu \in [\![R]\!]_G$ if and only if $\mu'(r(\bar{X})) \in [\![r(\bar{X})]\!]_G$. According to Definition 2.10 $\mu \models t_1 = t_2$ if and only if $\mu(t_1 = t_2)$ has the form $a = a$ where a is a constant. This condition is equivalent to that Datalog literals $\mu'(t_1 = t_2)$, $\mu'(t_1 \neq \varnothing)$, and $\mu'(t_2 \neq \varnothing)$ be true. Thus, $\mu \in [\![Q]\!]_G$ if and only if $\mu'(q(\bar{X})) \in [\![q(\bar{X})]\!]_G$. Hence, $\mu \in [\![Q]\!]_G$ if and only if $\mu \in [\![q(\bar{X})]\!]_G$. □

6.4 Fixing the translation of negation in filter-conditions

In this section we address the problem with negation in filter-conditions that the translation by Angles and Gutierrez [8] has. The following example shows that their translation fails when a filter-condition is evaluated as error.

Example 6.3. *Let R be the SPARQL query*

$$((s, p, X) \text{ OPT } (X, q, Y)),$$

Q be the query (R FILTER ?x $\neq$?y), and $G = \{(s, p, a), (a, q, b), (s, p, c)\}$ be an RDF graph. Then:

$$[\![R]\!]_G = \{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto c\}\},$$
$$[\![Q]\!]_G = \{\{X \mapsto a, Y \mapsto b\}\}.$$

The solution mapping $\mu = \{X \mapsto c\}$ of the query R is discarded in the query Q because $X \neq Y$ is evaluated as error since variable Y is unbound in μ.

According to the translation by Angles and Gutierrez [8], the SPARQL query Q is translated as the nr-Datalog$^\neg$ query $q(X, Y)$ over the following program:

$$q(X, Y) \leftarrow r(X, Y), \neg s(X, Y)$$
$$s(X, Y) \leftarrow r(X, Y), X = Y$$
$$\Pi_R$$

where the nr-Datalog$^\neg$ query $(r(X, Y), \Pi_R)$ is the translation of the SPARQL query R. Since the solutions of the nr-Datalog$^\neg$ query $r(X, Y)$ are the equivalent to the solutions of the SPARQL query R, then:

$$[\![r(X, Y)]\!]_G = \{\{q(a, b), q(c, \varnothing)\},$$
$$[\![s(X, Y)]\!]_G = \{\},$$
$$[\![q(X, Y)]\!]_G = \{\{q(a, b), q(c, \varnothing)\}.$$

Hence, the translation does not work because it returns the fact $q(c, \varnothing)$ that does not correspond to a solution in $[\![Q]\!]_G$.

The main idea to fix the translation of negation in filter-conditions is to separate the solution mappings of a query in three disjoint sets, namely the solutions that make the condition true, false or error. Formally, given a SPARQL query R, a filter-condition φ, and a graph G, these sets are:

$$\Omega_{true} = \{\mu \in [\![R]\!]_G \mid \mu(\varphi) \text{ is true}\}, \tag{6.8}$$
$$\Omega_{false} = \{\mu \in [\![R]\!]_G \mid \mu(\varphi) \text{ is false}\}, \tag{6.9}$$
$$\Omega_{error} = \{\mu \in [\![R]\!]_G \mid \mu(\varphi) \text{ is error}\}. \tag{6.10}$$

Then, it holds:

$$[\![(R \text{ FILTER } \neg\varphi)]\!]_G = [\![R]\!]_G \setminus (\Omega_{true} \cup \Omega_{error}), \tag{6.11}$$

that is, are all solutions mappings of the filtered query, except the mappings where the filter condition $\neg\varphi$ is not evaluated as true (i.e., where φ is evaluated as true of error).

On the contrary, the translation by Angles and Guttierrez [8] computes $[\![R]\!]_G \setminus \Omega_{true}$. To see this, recall that they translate the SPARQL query $(R \text{ FILTER } \neg\varphi)$ to a query $q(\bar{X})$ over the rules:

$$q(\bar{X}) \leftarrow r(\bar{X}), \neg s(\bar{X}) \ ; \ \Pi_R \tag{6.12}$$

where the nr-Datalog$^\neg$ query $(s(\bar{X}), \Pi_R)$ is the translation of the SPARQL query $(R \text{ FILTER } \varphi)$. Thus, Datalog rules in (6.12), correspond to the set difference $[\![R]\!]_G \setminus [\![(R \text{ FILTER } \varphi)]\!]_G$. Since, $\Omega_{true} = [\![(R \text{ FILTER } \varphi)]\!]_G$ the translation by Angles and Gutierrez [8] computes $[\![R]\!]_G \setminus \Omega_{true}$.

The problem with the translation by Angles and Gutierrez [8] is that it misses one of the three truth values that filter-conditions return, the error value. In order to fix the translation of negations in filter-conditions we will express Ω_{error} in nr-Datalog$^\neg$.

Definition 6.8 (Datalog query for mappings raising errors)**.** *Given a predicate p, a SPARQL query R equivalent to a nr-Datalog$^\neg$ query $(r(\bar{X}), \Pi_R)$, and a filter-condition φ, we will write* $\text{Error}^p_\varphi(R)$ *to denote the nr-Datalog$^\neg$ program defined recursively as follows:*

1. *If φ has the form $t_1 = t_2$ or* $\text{isBlank}(t)$ *then* $\text{Error}^p_\varphi(R)$ *is the program*

$$\begin{aligned} &p(\bar{X}) \leftarrow r(\bar{X}), X_1 = \varnothing \\ &\qquad\vdots \\ &p(\bar{X}) \leftarrow r(\bar{X}), X_n = \varnothing \\ &\Pi_R \end{aligned}$$

   where $X_1, \ldots, X_n$ are the variables occurring in φ.

2. *If φ has the form* $\text{bound}(X)$ *then* $\text{Error}^p_\varphi(R)$ *is the program*

$$\begin{aligned} &p(\bar{X}) \leftarrow \mathit{false} \\ &\Pi_R. \end{aligned}$$

   (We do not get errors in this case.)

3. *If φ has the form $\neg\varphi'$ then* $\text{Error}^p_\varphi(R)$ *is the program* $\text{Error}^p_{\varphi'}(R)$.

4. *If φ has the form $\varphi_1 \wedge \varphi_2$ then* $\text{Error}^p_\varphi(R)$ *is the following program:*

$$\begin{aligned} &p(\bar{X}) \leftarrow p_1(\bar{X}) \\ &p(\bar{X}) \leftarrow p_2(\bar{X}) \\ &\Pi_R \ ; \ \text{Error}^{p_1}_{\varphi_1}(R); \ \ \text{Error}^{p_2}_{\varphi_2}(R) \end{aligned}$$

5. *If φ has the form $\varphi_1 \vee \varphi_2$ then* $\text{Error}^p_\varphi(R)$ *is the following program:*

$$p(\bar{X}) \leftarrow p_1(\bar{X}), p_2(\bar{X})$$
$$\Pi_R \ ; \ \text{Error}^{p_1}_{\varphi_1}(R); \ \ \text{Error}^{p_2}_{\varphi_2}(R)$$

The nr-Datalog$^\neg$ program $\text{Error}^p_\varphi(R)$ is defined with the intention to return all facts $p(\bar{c})$ corresponding to the solution mappings of query R such that $\mu(\varphi)$ is error. The correctness of the construction of $\text{Error}^p_\varphi(R)$ is stated by the following theorem.

Theorem 6.3. *Let R be a SPARQL query equivalent to a nr-Datalog$^\neg$ query $(r(\bar{X}), \Pi_R)$, φ be a filter-condition, G be an RDF graph, μ be a solution mapping in $[\![R]\!]_G$, and Ω_{error} be the set*

$$\{\mu \in [\![R]\!]_G \mid \mu(\varphi) \textit{ is error}\}.$$

Then, $\mu \in \Omega_{error}$ if and only if $\mu \in [\![(p(\bar{X}), \text{Error}^p_\varphi(R))]\!]_G$ where $\text{Error}^p_\varphi(R))$ is the nr-Datalog$^\neg$ program defined using the query $(r(\bar{X}), \Pi_R)$ (recall that the definition of $\text{Error}^p_\varphi(R))$ depends on the existence of a translation for the SPARQL query R to nr-Datalog$^\neg$).

Proof. We will prove this theorem by induction in the structure of filter-conditions.

Let R be a SPARQL query, $\bar{X}$ be $\text{dom}(R)$, φ be a filter-condition, G be an RDF graph, μ be a mapping in $[\![R]\!]_G$, μ' be $\text{fill}(\mu\bar{X})$, and p be a fresh predicate.

1. If φ has the form $t_1 = t_2$ or $\text{isBlank}(t)$ then $\text{Error}^p_\varphi(R)$ is the program

$$p(\bar{X}) \leftarrow r(\bar{X}), X_1 = \varnothing$$
$$\vdots$$
$$p(\bar{X}) \leftarrow r(\bar{X}), X_n = \varnothing$$
$$\Pi_R$$

   where $X_1, \ldots, X_n$ are the variables occurring in φ. In this case $\mu(\varphi)$ is error iff $\mu(X_i)$ is unbound for one of the variables X_i occurring in φ. This is equivalent to say that $\mu'(X_i) = \varnothing$. Hence, in this case the theorem holds.

2. If φ has the form $\text{bound}(X)$ then $\text{Error}^p_\varphi(R)$ is the program

$$p(\bar{X}) \leftarrow \text{false}$$
$$\Pi_R.$$

   Since is not possible to generate an error in this case, the theorem holds.

3. If φ has the form $\neg\varphi'$ then $\text{Error}^p_\varphi(R)$ is the program $\text{Error}^p_{\varphi'}(R)$. Since the negation of error is error, and the negation of a truth value distinct of error is not error, then in this case the theorem holds.

4. If φ has the form $\varphi_1 \wedge \varphi_2$ then $\text{Error}^p_\varphi(R)$ is the following program:

$$p(\bar{X}) \leftarrow p_1(\bar{X})$$
$$p(\bar{X}) \leftarrow p_2(\bar{X})$$
$$\Pi_R \ ; \ \text{Error}^{p_1}_{\varphi_1}(R); \ \ \text{Error}^{p_2}_{\varphi_2}(R)$$

By the definition of the connective $\wedge$, $\mu(\varphi_1 \wedge \varphi_2)$ is error iff $\mu(\varphi_1)$ or $\mu(\varphi_2)$ are errors. By induction, $\mu(\varphi_1 \wedge \varphi_2)$ is error if and only if $\mu'(p_1(\bar{X})) \in [\![(p_1(\bar{X}), \text{Error}^{p_1}_{\varphi_1}(R))]\!]_G$ or $\mu'(p_2(\bar{X})) \in [\![(p_2(\bar{X}), \text{Error}^{p_2}_{\varphi_2}(R))]\!]_G$. Thus, $\mu(\varphi_1 \wedge \varphi_2)$ is error if and only if $\mu'(p(\bar{X})) \in [\![(p(\bar{X}), \text{Error}^{p}_{\varphi}(R))]\!]_G$. Hence, in this case the theorem holds.

5. If φ has the form $\varphi_1 \vee \varphi_2$ then $\text{Error}^{p}_{\varphi}(R)$ is the following program:

$$p(\bar{X}) \leftarrow p_1(\bar{X}), p_2(\bar{X})$$
$$\Pi_R \ ; \ \text{Error}^{p_1}_{\varphi_1}(R); \ \ \text{Error}^{p_2}_{\varphi_2}(R)$$

By the definition of the connective $\vee$, $\mu(\varphi_1 \vee \varphi_2)$ is error iff $\mu(\varphi_1)$ and $\mu(\varphi_2)$ are error. By induction, $\mu(\varphi_1 \vee \varphi_2)$ is error if and only if $\mu'(p_1(\bar{X})) \in [\![(p_1(\bar{X}), \text{Error}^{p_1}_{\varphi_1}(R))]\!]_G$ and $\mu'(p_2(\bar{X})) \in [\![(p_2(\bar{X}), \text{Error}^{p_2}_{\varphi_2}(R))]\!]_G$. Thus, $\mu(\varphi_1 \vee \varphi_2)$ is error if and only if $\mu'(p(\bar{X})) \in [\![(p(\bar{X}), \text{Error}^{p}_{\varphi}(R))]\!]_G$. Hence, in this case the theorem holds.

Since the theorem holds in all cases, it is proved. □

So far, given a SPARQL query R, a filter-condition φ, and a graph G, we have that

$$[\![(R \text{ FILTER } \neg\varphi)]\!]_G = [\![R]\!]_G \setminus (\Omega_{true} \cup \Omega_{error})$$

where Ω_{true} and Ω_{error} are defined respectively in (6.8) and (6.10). Also, we have a nr-Datalog$^\neg$ codification for the queries that return Ω_{true} and Ω_{error}. Thus, we are ready to present a correct translation for the query $(R \text{ FILTER } \neg\varphi)$.

Definition 6.9 (Translation of negation in filter-conditions)**.** *Let R be a SPARQL query and φ be a filter-condition. Then the translation of the SPARQL query $(R \text{ FILTER } \varphi)$ is the nr-Datalog$^\neg$ query $(q(\bar{X}), \Pi_Q)$ where Π_Q is the following program:*

$$\begin{array}{lll} (R \text{ FILTER } \neg\varphi) & \mapsto & q(\bar{X}) \leftarrow r(\bar{X}), \neg s(\bar{X}) \\ & & s(\bar{X}) \leftarrow t(\bar{X}), \\ & & s(\bar{X}) \leftarrow p(\bar{X}) \\ & & \Pi_T \ ; \ \text{Error}^{p}_{\varphi}(R) \end{array}$$

where $(t(\bar{X}), \Pi_T)$ is the translation of the query $(R \text{ FILTER } \varphi)$.

Example 6.4. *Let Q be the SPARQL query of Example 6.3, that we repeated here:*

$$(R \text{ FILTER } X \neq Y),$$

where R is the query defined as follows:

$$((s, p, X) \text{ OPT } (X, q, Y)).$$

Let G be the RDF graph of Example 6.3, that is composed by the following triples:

$$\{(s, p, a), (a, q, b), (s, p, c)\}.$$

Then, according to Definition 6.9, query Q is translated as the nr-Datalog$^\neg$ query $q(X, Y)$ over the following program:

$$
\begin{aligned}
&q(X, Y) \leftarrow r(X, Y), \neg s(X, Y)\\
&s(X, Y) \leftarrow t(X, Y),\\
&s(X, Y) \leftarrow p(X, Y)\\
&\Pi_T \ ; \ \mathrm{Error}^p_\varphi(R)
\end{aligned}
$$

where $(t(\bar{X}), \Pi_T)$ is the translation of the query (R FILTER $X = Y$).

Then:

$$
\begin{aligned}
&[\![r(X, Y)]\!]_G = \{r(a, b), r(c, \varnothing)\},\\
&[\![t(X, Y)]\!]_G = \{\},\\
&[\![p(X, Y)]\!]_G = \{p(c, \varnothing)\},\\
&[\![s(X, Y)]\!]_G = \{s(c, \varnothing)\},\\
&[\![q(X, Y)]\!]_G = \{q(a, b)\}.
\end{aligned}
$$

Thus, the result of the SPARQL query Q coincides with the result of the nr-Datalog$^\neg$ query $q(X, Y)$.

The correctness of the translation proposed in Definition 6.9 is stated in the following theorem.

Theorem 6.4. *The translation resulting from replacing in Definition 6.4 the rule for negation in filter-conditions by the rule provided by Definition 6.9, is correct for all queries in the safe SPARQL-0 fragment.*

Proof. We will prove this theorem by induction on the structure of queries, assuming that all other rules of the translation are correct.

Let $Q = (R \text{ FILTER } \neg\varphi)$ be a safe SPARQL query and G be a graph pattern. According to Definition 6.9 the translation of the SPARQL query Q is a nr-Datalog$^\neg$ query of the form $q(\bar{X})$ over the following program:

$$
\begin{aligned}
(R \text{ FILTER } \neg\varphi) \quad \mapsto \quad & q(\bar{X}) \leftarrow r(\bar{X}), \neg s(\bar{X})\\
& s(\bar{X}) \leftarrow t(\bar{X}),\\
& s(\bar{X}) \leftarrow p(\bar{X})\\
& \Pi_T \ ; \ \mathrm{Error}^p_\varphi(R)
\end{aligned}
$$

where $(t(\bar{X}), \Pi_T)$ is the translation of the query (R FILTER φ).

Let μ be an SPARQL mapping in $[\![R]\!]_G$. Then, by induction, $\mu(\varphi)$ is true if and only if $\mu \in [\![t(\bar{X})]\!]_G$. By Theorem 6.3, $\mu(\varphi)$ is error if and only if $\mu \in [\![\mathrm{Error}^p_\varphi(R)]\!]_G$. Thus, $\mu \in [\![s(\bar{X})]\!]_G$ if and only if $\mu(\varphi)$ is true or error. Hence, $\mu \in [\![Q]\!]_G$ if and only if $\mu \in [\![q(\bar{X})]\!]_G$. □

As conclusion, in this section we described and fixed the problem of the translation by Angles and Gutierrez [8] for SPARQL queries of the form (R FILTER $\neg\varphi$).

Theorem 6.4 shows that the aforementioned translation of SPARQL-0 queries is correct. Since the translation of a query of the form (R FILTER φ) is based on the fact that we know a translation for query R without requiring R being in the SPARQL-0 fragment, this translation for filter-conditions is also valid for the extensions of the SPARQL-0 fragment presented in the next section.

6.5 Translation of the SPARQL-1 fragment

So far, we have a translation for SPARQL queries in the safe SPARQL-0 fragment to nr-Datalog$^\neg$ queries. In this section we will extend that translation to include the operators SELECT, MINUS, VALUES and BIND. We call SPARQL-1 to the resulting fragment, by the capital letters of the operators included. The SPARQL-1 fragment includes all SPARQL queries we can express in the core fragment we study in this thesis (see Definition 2.8), except queries including the EXISTS clause inside filter-conditions. As we already stated, the translation of the EXISTS operator to nr-Datalog$^\neg$ will be studied in Chapter 7.

6.5.1 SELECTION queries in nr-Datalog$^\neg$

The operator SELECTION was not considered in the translations from SPARQL to nr-Datalog$^\neg$ by neither Polleres [60], Schenk [66], nor Angles and Gutierrez [8], because SELECT was not included as an operator of graph patterns in SPARQL 1.0. The possibility of including SELECT queries inside other queries is present in SPARQL only since version 1.1.

The translation of a SELECT query (SELECT $\bar{X}$ R) to a nr-Datalog$^\neg$ query is straightforward. It requires producing facts of the form $q(\bar{X})$ from the solutions of R. This implies discarding values from variables in dom(R) that are not in $\bar{X}$ and setting variables in $\bar{X}$ that are not in dom(R) with the value $\varnothing$.

Definition 6.10 (Translation of SELECT queries to nr-Datalog$^\neg$)**.** *Let Q be a SELECT query in the safe SPARQL-1 fragment. Q is then translated as a nr-Datalog$^\neg$ query of the form $(q(\bar{X}), \Pi_Q)$ using the following rule:*

$$(\text{SELECT } \bar{X} \text{ WHERE } R) \quad \mapsto \quad q(\bar{X}) \leftarrow r(\bar{Y}), X_1 = \varnothing, \ldots, X_n = \varnothing;\ \Pi_R$$

where $(r(\bar{Y}), \Pi_R)$ is the translation of the SPARQL R to nr-Datalog$^\neg$, and $\bar{X} \setminus \bar{Y} = \{X_1, \ldots, X_n\}$.

Example 6.5. *The translation of the query* (SELECT$\{X, Y\}$WHERE(X, p, o)) *is the query $q(X, Y)$ over the program:*

$$q(X, Y) \leftarrow r(X), Y = \varnothing$$
$$r(X) \leftarrow \text{triple}(X, p, o)$$

6.5.2 MINUS queries in nr-Datalog$^\neg$

The translation of MINUS is implicit from the translation of the operator OPT by Angles and Gutierrez [8].

Example 6.6. *Let Q be the SPARQL query $((X, p, o)\,\text{OPT}\,(X, q, Y))$. According to Definition 6.4 the translation of Q is then the nr-Datalog$^\neg$ query $q(X, Y)$ over the program Π defined as follows:*

$$
\begin{aligned}
q(X, Y) &\leftarrow r(X_1), s(X_2, Y), \text{comp}(X, X_1, X_2)\\
q(X, Y) &\leftarrow r(X_1), \neg t(X_2), \text{comp}(X, X_1, X_2), Y = \varnothing\\
t(X) &\leftarrow s(X, Y)\\
r(X) &\leftarrow \text{triple}(X, p, o)\\
s(X, Y) &\leftarrow \text{triple}(X, q, Y)\\
&\Pi_{\text{comp}}
\end{aligned}
$$

The program Π is the union of the programs Π_1 and Π_2 defined as follows:

$$
\Pi_1 = \begin{cases}
q(X, Y) \leftarrow r(X_1), s(X_2, Y), \text{comp}(X, X_1, X_2)\\
r(X) \leftarrow \text{triple}(X, p, o)\\
s(X, Y) \leftarrow \text{triple}(X, q, Y)\\
\Pi_{\text{comp}}
\end{cases}
$$

$$
\Pi_2 = \begin{cases}
q(X, Y) \leftarrow r(X_1), \neg t(X_2), \text{comp}(X, X_1, X_2), Y = \varnothing\\
t(X) \leftarrow s(X, Y)\\
r(X) \leftarrow \text{triple}(X, p, o)\\
s(X, Y) \leftarrow \text{triple}(X, q, Y)\\
\Pi_{\text{comp}}
\end{cases}
$$

Because the equivalence

$$Q \equiv ((X, p, o)\,\text{AND}\,(X, q, Y))\,\text{UNION}\,((X, p, o)\,\text{MINUS}\,(X, q, Y))$$

and that $(q(X, Y), \Pi_1)$ is the translation of the SPARQL query $((X, p, o)\,\text{AND}\,(X, q, Y))$ it results natural that the translation of $((X, p, o)\,\text{MINUS}\,(X, q, Y))$ be $(q(X, Y), \Pi_2)$.

The observation made in the previous example is generalized in the following definition to construct the translation of MINUS to nr-Datalog$^\neg$.

Definition 6.11 (Translation of MINUS queries to nr-Datalog$^\neg$)**.** *Let Q be a MINUS query in the safe SPARQL-1 fragment. Q is then translated as a nr-Datalog$^\neg$ query of the form $(q(\bar{X}), \Pi_Q)$ using the following rule:*

$$
\begin{aligned}
(R\,\text{MINUS}\,S) \quad \mapsto \quad & q(\bar{X}) \leftarrow r(h_Y(\bar{Y})), \neg t(h_Z(\bar{W})), \text{Comp}(\bar{W})\\
& t(\bar{W}) \leftarrow s(\bar{Z})\\
& \Pi_R\ ;\ \Pi_S
\end{aligned}
$$

where $(r(\bar{Y}), \Pi_R)$ and $(s(\bar{Z}), \Pi_S)$ are the respective translations of the SPARQL queries R and S, t is a fresh predicate to ensure safeness, $\bar{W} = \bar{X} \cap \bar{Y}$, functions $h_Y : \bar{W} \to \mathbf{V}$ and $h_z : \bar{W} \to \mathbf{V}$ rename variables in $\bar{W}$ to fresh variables (where ranges of functions h_Y and h_Z are disjoint), and $\text{Comp}(\bar{W})$ is the set of literals described in Definition 6.4.

6.5.3 VALUES queries to nr-Datalog$^\neg$

The VALUES operator is introduced in version 1.1 of SPARQL to define inline data in SPARQL queries. Its translation to nr-Datalog$^\neg$ is as follows.

Definition 6.12 (Translation of VALUES queries to nr-Datalog$^\neg$)**.** *Let Q be a VALUES query in the safe SPARQL-1 fragment. The translation of Q is then a query of the form $q(\bar{X})$ over a program having a rule of the form*

$$q(\bar{X}) \leftarrow X_1 = \mu'(X_1), \ldots, X_n = \mu'(X_n)$$

where $\mu' = \text{fill}(\mu, \bar{X})$, $\bar{X} = \{X_1, \ldots, X_n\}$, for each mapping $\mu \in \Omega$.

Example 6.7. *Let Q be the VALUES query* (VALUES $\{X, Y\}$ $\{\{X \mapsto a\}, \{Y \mapsto b\}\}$)*. The translation of Q is then the nr-Datalog$^\neg$ query $q(X, Y)$ over the following program:*

$$q(X, Y) \leftarrow X = a, Y = \varnothing$$
$$q(X, Y) \leftarrow X = \varnothing, Y = b$$

6.5.4 BIND queries in nr-Datalog$^\neg$

SPARQL 1.1 introduces the possibility of assigning a value of a expression into a variable of a mapping. This feature exceeds the expressive power of pure nr-Datalog$^\neg$, but can be expressed in it if *built-ins* are allowed.

Definition 6.13 (Translation of BIND queries to nr-Datalog$^\neg$)**.** *Let $\bar{V}$ be a set of variables, $f : \mathbf{C}_\varnothing^{|\bar{V}|} \to \mathbf{C}_\varnothing$ be a function for which there is a built-in, and Z be a variable that is not in* $\text{dom}(R)$ *nor in $\bar{V}$. The translation of BIND queries is then defined as follows:*

$$(R \text{ BIND}(f(\bar{V}) \text{ AS } Z)) \quad \mapsto \quad q(\bar{X}) \leftarrow r(\bar{Y}), X_1 = \varnothing, \ldots, X_n = \varnothing, Z = f(\bar{V})$$

where $f(\bar{V})$ denotes the built-in that implements the function f, i.e., where given a substitution θ, $\theta(Z = f(\bar{V}))$ if and only if $\theta(Z) = \theta(f(\bar{V}))$.

6.5.5 The translation of SPARQL-1 fragment to nr-Datalog$^\neg$

Now we can state and prove that the translation of the whole SPARQL-1 fragment to nr-Datalog$^\neg$ is correct.

Theorem 6.5. *The translation of SPARQL queries in the safe SPARQL-1 fragment to nr-Datalog$^\neg$ given by the definitions 6.4 (except equality atoms and negations in filter-conditions), 6.7, 6.9, 6.10, 6.11, 6.12, and 6.13 to nr-Datalog$^\neg$ is correct.*

Proof. This theorem is proved by induction in the structure of queries. The correctness of the translation is already proved for rules in all the referred definitions, except the definitions corresponding to the translation for the "new" operators SELECT, MINUS, VALUES and BIND that are presented in this section (i.e., the definitions 6.10, 6.11, 6.12, and 6.13). Thus, it suffices proving the correctness of the translation for these operators because each operator behaves independently of others and so the recursion proof could be done only over the new ones whose proof is missing.

– SELECT: Let Q be a SPARQL query of the form (SELECT $\bar{X}$ WHERE R) in the SPARQL-1 fragment, G be an RDF graph, and the nr-Datalog$^\neg$ queries $q(\bar{X})$ and $r(\bar{Y})$ over the program

$$q(\bar{X}) \leftarrow r(\bar{Y}), X_1 = \varnothing, \ldots, X_n = \varnothing \; ; \; \Pi_R$$

where $X_1, \ldots, X_n$ are the variables of $\bar{X}$ that do not occur in R, be the respective translations of the SPARQL queries Q and R. Let $\mu \in [\![Q]\!]_G$. By definition, if $\mu \in [\![Q]\!]_G$ then there exists a mapping $\mu' \in [\![R]\!]_G$ such that $\mu = \mu'|_{\bar{X}}$. By induction, $\mu' \in [\![r(\bar{Y})]\!]_G$. That is, $\text{fill}(\mu, \bar{Y})(r(\bar{Y})) \in [\![r(Y)]\!]_G$. By the first rule of the program of query $q(\bar{X})$ we have that $\text{fill}(\mu', \bar{Y} \cup \bar{X})(q(\bar{X})) \in [\![q(X)]\!]_G$. Since

$$\begin{aligned}\text{fill}(\mu', \bar{Y} \cup \bar{X})(q(\bar{X})) &= \text{fill}(\mu', \bar{X})(q(\bar{X})) \\ &= \text{fill}(\mu, \bar{X})(q(\bar{X}))\end{aligned}$$

then $\text{fill}(\mu, \bar{X})(q(\bar{X})) \in [\![q(X)]\!]_G$. Thus, $\mu \in [\![q(\bar{X})]\!]_G$. Since this argument is valid in both directions, the translation is correct for SELECT queries.

– MINUS: Let Q be a SPARQL query of the form (R MINUS S) in the SPARQL-1 fragment, G be a graph, and the nr-Datalog$^\neg$ queries $q(\bar{X})$, $r(\bar{Y})$, and $s(\bar{Z})$ over the program

$$\begin{aligned}&q(\bar{X}) \leftarrow r(h_Y(\bar{Y})), \neg t(h_Z(\bar{W})), \text{Comp}(\bar{W}) \\ &t(\bar{W}) \leftarrow s(\bar{Y}) \\ &\Pi_R \,;\, \Pi_S\end{aligned}$$

be the respective translations of the SPARQL queries Q, R, and S, and h_Y, h_z, $\bar{W}$, and $\text{Comp}(\bar{W})$ are the structures described in Definition 6.11. Let $\mu_1 \in [\![Q]\!]_G$. By definition, if $\mu_1 \in [\![Q]\!]_G$ then $\mu_1 \in [\![R]\!]_G$ and a mapping $\mu_2 \in [\![S]\!]_G$ such that $\mu_1 \sim \mu_2$ there not exists. By induction $\mu_1 \in [\![r(\bar{Y})]\!]_G$ and there not exists a mapping $\mu_2 \in [\![s(\bar{X})]\!]_G$ such that $\mu_1 \sim \mu_2$. According to the definition of $\text{dom}(\cdot)$, it holds $\bar{X} = \bar{Y}$. Thus, by the first rule of the program of the query $q(\bar{X})$, it holds that $\mu_1 \in [\![q(\bar{X})]\!]_G$. Since this argument is valid in both directions, the translation is correct for MINUS queries.

– VALUES: Let Q be a SPARQL query of the form (VALUES $\bar{X}$ Ω), G be an RDF graph, and the nr-Datalog$^\neg$ query $q(\bar{X})$ be the translation of the SPARQL query Q. Let $\mu \in [\![Q]\!]_G$. By definition, $\mu \in \Omega$. By construction there is a rule in the program of query $q(\bar{X})$ such that $\mu \in [\![q(\bar{X})]\!]_G$. Since this argument is valid in both directions, the translation is correct for VALUES queries.

– BIND: Let Q be a SPARQL query of the form (R BIND($f(\bar{V})$ AS $\bar{Z}$)) in the SPARQL-1 fragment, G be an RDF graph, and the nr-Datalog$^\neg$ queries $q(\bar{X})$ and $r(\bar{Y})$ be the respective translations of the SPARQL queries Q and R. Let $\mu \in [\![Q]\!]_G$. By definition, $\mu|_{\bar{Y}} \in [\![R]\!]_G$. By induction, $\mu|_{\bar{Y}} \in [\![r(\bar{Y})]\!]_G$. Let $\theta = \text{fill}(\mu, \bar{X})$. By Definition 6.13, $\theta(Z = f(\bar{V}))$. Thus, $\theta(q(\bar{X})) \in [\![q(X)]\!]_G$. Thus, $\mu \in [\![q(X)]\!]_G$. Since this argument is valid in both directions, the translation is correct for BIND queries.

□

6.6 Conclusion

In this Chapter we described and fixed the issues found in the translation from SPARQL to nr-Datalog$^\neg$ by Angles and Gutierrez [8]. These issues are related to the fact that

SPARQL assigns three truth values to filter-conditions. The specification calls "error" to the third truth value, instead of "unknown" as usual. This is because SPARQL also allows expressions that produce errors (for instance, if a number is added to a date). However, in the cases we address in this thesis, the only possibility to get errors is when variables are unbound. As we showed in Chapter 4, unbound variables can be interpreted as null values, so under that semantics errors are got when the truth value of a selection formula is unknown.

It is interesting to note that to fix the translation by Angles and Gutierrez [8] we use a technique that is inspired in the one used to approximate certain answers in Chapter 3. In fact, we defined a query that returns the mappings where the evaluation raises error. Likewise, to approximate certain answers we rewrite filter-conditions to capture the cases where values are null.

Recall that in the previous Chapter we used a naive translation from a reduced fragment of SPARQL to first-order logic to study the problem of substitution. However that translation did not produce equivalent queries. To analyze the problem of substitution using a logic based language, we thus required a translation that produces equivalent queries, as the translation from SPARQL to nr-Datalog$^\neg$ by Angles and Gutierrez [8]. However, we found some issues in their translation we had to fix. In the next chapter we return to the study of substitution in SPARQL by translating it to nr-Datalog$^\neg$.

Chapter 7

Nested Datalog

In the previous chapter we presented a translation from SPARQL queries without occurrences of the EXISTS clause to nr-Datalog$^\neg$. The clause EXISTS essentially amounts to composition of queries. In this chapter we propose an extended variant for Datalog designed to allow query composition, that we call Nested Datalog. The motivation of this new formalism is to capture the fundamental aspects of the notion of nesting in query languages, and by means of it study nesting in SPARQL by using the composition of queries in this extended version of Datalog.

Nested Datalog introduces two main ideas: programs are composed of blocks (i.e. a program can have "nested" programs inside it), and blocks can be "called" using a special type of atoms, called "query atoms". Query atoms can have free variables that are used as parameters when Block programs are called from other programs. Query atoms are similar to what are called *external atoms* by Eiter et al. [22] and denoted with the prefix #. We follow their notation and semantics.

The main issues arousing from introducing these new type of atoms are the scope of variables. In standard Datalog the scope of each variable is the rule where the variable occurs. In Nested Datalog the scope of a variable can range beyond the rule where the call is made, i.e., the rule where the query atoms belongs to. This brings several issues regarding the semantics of substitution of variables and, as we will show, these issues reflect precisely the issues discussed about substitution in SPARQL.

Organization of this chapter

- In Section 7.1 we present the syntax and the semantics of Nested Datalog.
- In Section 7.2 we describe the relation among Nested Datalog and the external atoms defined by Eiter et al. [22, 23].
- In Section 7.3 we show that Nested nr-Datalog$^\neg$ and nr-Datalog$^\neg$ have the same expressive power.
- We conclude the chapter in Section 7.4.

7.1 Syntax and Semantics of Nested Datalog

Nested Datalog is an extension of Datalog designed to support composition of queries. The intuitive idea is to allow call to subprogramas (blocks) inside Datalog programs in order to simulate subqueries in standard query languages.

The extension from Datalog to Nested Datalog is done by allowing a special type of atoms, called *query atoms* that will represent blocks (subprograms). In this regard, Nested Datalog rules extend Datalog rules by admitting *query atoms* in the body of their rules.

Definition 7.1 (Nested Datalog Syntax)**.** *A Nested Datalog program* Π *is a finite set of pairs* $\{(\pi_1, \bar{X}_1), \ldots, (\pi_n, \bar{X}_n)\}$*, called the blocks of the program, where for each* $1 \leq i \leq n$*,* π_i *is a set of Nested Datalog rules, and* $\bar{X}_i$ *is a finite set of variables. We write* $\pi_i[\bar{X}_i]$ *to denote the block* $(\pi_i, \bar{X}_i)$*. The set of intensional predicates of each program* π_j *is disjoint with the intensional predicates of other programs* $\pi_k \in \Pi$*.*

A query atom *is an intensional predicate* p *of a block* $\pi_i[\bar{X}_i] \in \Pi$ *instantiated with a list of terms* $\bar{t}$ *plus the set of variables* $\bar{X}_i$ *of the block (called the parameters of the query atom). It will be denoted (following HEX programs notation)* $\#p[\bar{X}_i](\bar{t})$*. Brackets "[]" are omitted when a query atom has no parameters. We say that a query atom* $\#p_j[\bar{X}_j](\bar{t}_j)$ *is nested in a query atom* $\#p_k[\bar{X}_k](\bar{t}_k)$ *if* $\#p_j[\bar{X}_j](\bar{t}_j)$ *occurs in a rule of the block* $\pi_k[\bar{X}_k]$ *where the intensional predicate* p_k *belongs to.*

Example 7.1. *The following is simple Nested Datalog program. Consider the following rules:*

$$p(X) \leftarrow q(X, Y) \qquad (\pi_1)$$
$$q(X, Y) \leftarrow r(X, Y), \neg \#s[X](a, Y)$$

$$s(Z, Y) \leftarrow \#t(X, Y), \#u(Y, Z) \qquad (\pi_2[X])$$

$$t(X, Y) \leftarrow v(X, Y) \qquad (\pi_3)$$

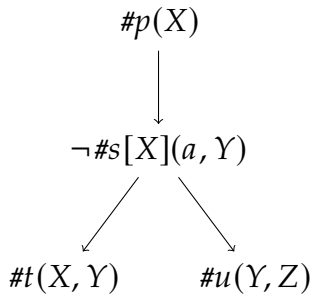
$$u(X, Y) \leftarrow w(X, Y) \qquad (\pi_4)$$

Extra vertical space between groups of rules is added only to denote that they belong to each of the four different blocks (indicated in parenthesis on the right). The set $\{\pi_1, \pi_2[X], \pi_3, \pi_4\}$ *is a Nested Datalog program. The list of terms of a query atom, e.g.* (a, Y) *in* $\#s[X](a, Y)$ *or* (Y, Z) *in* $\#u(Y, Z)$*, is intended to denote that the "subqueries are called" with these values. On the other hand, the parameters inside* $[\cdot]$ *are intended to denote "global variables": e.g.* $[X]$ *in* $\#s[X](a, Y)$ *indicates that in the whole subprogram* s*, the variable* X *represent the same value, e.g.* X *occurring in* $\#t(X, Y))$ *represents the same as the one in* $q(X, Y)$*. The parameters* $[\cdot]$ *were necessary because recall that variables in a rule in standard datalog are "local" variables, e.g. the* X *in* $p(X) \leftarrow q(X, Y)$ *is "local" (inside the rule), e.g. could be renamed.*

Recall that a Datalog query looks like $(p(\bar{t}), \pi)$ where the predicate atom $p(\bar{t})$ is the goal and π is the program. Since each intensional predicate belongs to a unique block of a Nested Datalog program, a query atom $\#p[\bar{X}](\bar{t})$ is simply a traditional $(p(\bar{t}), \pi)$ plus a set of parameters $\bar{X}$. We call Nested Datalog queries (or simply queries) to query atoms. We borrowed the notation of external atoms used in the context of HEX programs [22, 23] to denote query atoms.

Definition 7.2 (Nesting-graph and stratified queries)**.** *The* nesting-graph *of a Nested Datalog query Q is the directed multigraph whose nodes are the queries occurring in Q, including Q, and there is an edge (Q_i, Q_j) if and only if query Q_j is nested in query Q_i. A query Q is said to be stratified if and only if its nesting-graph is a tree, each node of the nesting-graph belongs to a different program, and each query atom occurs only once in the Nested Datalog query. A node of the nesting graph of a query Q is said negative (and denoted with $\neg$ as prefix) if it occurs in a negative literal in Q.*

Example 7.2. *The following figure depicts the nesting-graph of the query of Example 7.1.*



This query is stratified because its nesting-graph is a tree and each node belongs to a different program.

In the following we consider only stratified Nested Datalog. These types of queries are enough to model relevant features of nesting in query languages.[1] In these queries each block is associated with exactly one node of the nesting-graph, and the nesting-graph is a tree. Thus, queries and blocks are structured in "levels of nesting". We can say that one query is nested in another if the nested query is "below" in the nesting-graph. The following definition presents the terminology that we will use along this thesis.

Definition 7.3 (Inner, outer and nested queries)**.** *Let queries Q_i and Q_j be nodes of a nesting-tree. If node Q_i is the parent node of Q_j, then we say that query Q_j is an* inner query *of query Q_i (at level 1), or that Q_i is an* outer query *of Q_j, or that Q_j is* nested *in Q_i. We say Q_j is an* inner query *of query Q_i at level k, if node Q_i is k levels above node Q_j in the nesting-tree. The same definitions that apply to queries apply to blocks, e.g., the block π_j of query Q_j is an* inner block *of block π_i of query Q_i if Q_j is an inner query of Q_i. We say that query Q_j is* nested *in a rule R if Q_j occurs in the body of R. We say that a rule R* belongs *to a query Q_i if R belongs to the block of Q_i.*

Scope of variables. As we suggested in Example 7.1, the scope of variables in Nested Datalog is different than in Standard Datalog. In Standard Datalog the scope of a variable X is the rule R where X occurs. Indeed, in the interpretation of rule R as a logical formula, variable X is universally quantified. On the other hand, in Nested Datalog, the scope of X in a rule R will be the rule R *unless* R belongs to a block $\pi[\bar{X}]$ or has a query atom $\#p[\bar{X}](\bar{t})$ where $X \in \bar{X}$, in which case the scope of X extends beyond the rule R.

[1] Furthermore, a Nested Datalog query where a block is pointed to by two or more query atoms, can be normalized by duplicating the pointed block and renaming consistently each intensional predicate in it by fresh predicates.

Example 7.3. *Consider the following Nested Datalog program:*

$$p(X) \leftarrow q(X), \#r[X](Z) \qquad (\pi_1)$$

$$r(Y) \leftarrow s(Y, Z), Z \neq X \qquad (\pi_2[X])$$

The scope of variable X in the first rule also includes the second rule, because the second rule belongs to the block $\#\pi_2[X]$ with parameter X. On the contrary, the scope of variable Z in the second rule is that same rule, because Z is not a parameter of the block containing it.

The intuition behind the notion of *parameter*, i.e. those variables inside $[\cdots]$, is to indicate "global" variables. This intuition leads to formalizing the notion of substitution in blocks.

Definition 7.4 (Substitution)**.** *Given a block $\pi[\bar{X}]$ and a substitution θ with $\mathrm{dom}(\theta) = \bar{X}$, the result of substituting the parameters of block π with θ is the block, denoted $\pi[\theta]$, with no (variable) parameters, and consisting in the rules that result from adding the equality atom $X = \theta(X)$ to each rule of block π, for each parameter $X \in \bar{X}$.*

Example 7.4. *Consider the block $\pi[X]$ consisting of the rule $r(Y) \leftarrow s(Y, Z), Z \neq X$, and θ be the substitution $\{X/a\}$. The result of substituting the parameters of block $\pi[X]$ with θ is the block $\pi[\theta]$ with no parameters consisting of the rule $r(Y) \leftarrow s(Y, Z), Z \neq X, X = a$. Intuitively, the equality atom $X = a$ added to each rule of the inner-query ensures that the variable X is restricted to the value a.*

Now we are ready to present the semantics of Nested Datalog. We assume a proof-theoretic semantics for Nested Datalog programs.

Definition 7.5 (Semantics of Nested Datalog)**.** *Let D be a database and Π be a Nested Datalog program. The* derivation trees *of Π with respect to D, denoted $\mathrm{dt}(\Pi, D)$, are defined recursively as follows:*

1. *For each fact $f \in D$ and block with no parameters $\pi \in \Pi$ there is a derivation tree in $\mathrm{dt}(\Pi, D)$ having (f, π) as unique node.*
2. *Let $\pi \in \Pi$ be a block with no parameters, and $R \in \pi$ be a rule of the form $L_{n+1} \leftarrow L_1, \ldots, L_n$, and θ be a substitution such that the domain of θ is the set of variables occurring in R such that one the following conditions hold, and r_i be the derivation tree defined as follows, for $1 \leq i \leq n$:*
   (a) *L_i is a positive predicate atom and there is a derivation tree $t_i \in \mathrm{dt}(\Pi, D)$ whose root is $(\theta(L_i), \pi)$. In this case, $r_i = t_i$.*
   (b) *L_i is a negative predicate atom and there is no derivation tree $t_i \in \mathrm{dt}(\Pi, D)$ such that the root of t_i is $(\theta(L_i), \pi)$. In this case, r_i is a derivation tree with a single node $(\theta(L_i), \pi)$.*
   (c) *L_i is a positive equality atom and $\theta(L_i)$ is an equality of the form $a = a$ where a is a constant. In this case, r_i is a derivation tree with a single node $(a = a, \pi)$.*
   (d) *L_i is a negative equality atom and $\theta(L_i)$ is an inequality of the form $a \neq b$ where a and b are different constants. In this case, r_i is a derivation tree with a single node $(a \neq b, \pi)$.*
   (e) *L_i is a positive query atom $\#p[\bar{X}](\bar{t})$, predicate p belongs to block $\pi'[\bar{X}]$, and there is a derivation tree $t_i \in \mathrm{dt}(\Pi \cup \{\pi'[\theta]\}, D)$ where the root of t_i is $(\theta(p(\bar{t})), \pi'[\theta])$. In this case, $r_i = t_i$.*

(f) *L_i is a negative query atom $\neg\#p[\bar{X}](\bar{t})$, predicate p belongs to block $\pi'[\bar{X}]$, and there is no derivation tree $t_i \in \mathrm{dt}(\Pi \cup \{\pi'[\theta]\}, D)$ where the root of t_i is $(\theta(p(\bar{t})), \pi'[\theta])$. In this case, r_i is a derivation tree with a single node $(\theta(L_i), \pi'[\theta])$.*

If the conditions above hold then there is a derivation tree $t \in \mathrm{dt}(\Pi, D)$ consisting of:

- *Root $\theta(L_{n+1})$;*
- *Ordered edges $\theta(L_{n+1}) \to r_i$ with label R, for $1 \leq i \leq n$.*

3. *No more derivations trees are in* $\mathrm{dt}(\Pi, D)$.

Like in standard Datalog, a Nested Datalog query is a pair $Q = (p(\bar{X}), \Pi)$ where $p(\bar{X})$ is a positive literal called the goal of the query, and Π is a Nested Datalog program. Given a database D, a substitution whose domain are the variables in $p(\bar{X})$ is said to be an answer of Q in D, denoted $\theta \in [\![Q]\!]_D$, if there exists a derivation tree $t \in \mathrm{dt}(\Pi, D)$ such that the root of t has the form $(\theta(p(\bar{X})), \pi)$ where π is the block where the predicate of p belongs to.

Example 7.5. *Consider the program $\Pi = \{\pi_1, \pi_2[X]\}$ defined by the following rules:*

$$p(X, Z) \leftarrow r(X), \#q[X](Y), s(Y, Z) \qquad (\pi_1)$$

$$\begin{aligned} q(Y) &\leftarrow t(Y), \neg u(Y) \\ u(Y) &\leftarrow w(Y, W), W \neq X \end{aligned} \qquad (\pi_2)$$

Let us illustrate some steps of the evaluation of the query $\#p(X, Z)$ against the database $D = \{r(a), s(b, c), \dots\}$. A possible solution amounts to check if there is a derivation tree for the literal $\#q[X](Y)$ with $\theta = \{X/a, Y/b, Z/c\}$, in order to know if $p(a, c)$ is an answer of query $\#p(X, Z)$. This means, finding a derivation tree $t \in \mathrm{dt}(\Pi \cup \{\pi_2[\theta]\}, D)$ whose root is $(q(b), \pi_2[\theta])$ and $\pi_2[\theta]$ is the following program:

$$\begin{aligned} q'(Y) &\leftarrow t'(Y), \neg u(Y), X = a \\ u'(Y) &\leftarrow w(Y, W), W \neq X, X = a \end{aligned} \qquad (\pi_2[\theta])$$

Safeness and non-recursiveness. As with Standard Datalog, in this thesis we will consider *safe* an *non-recursive* stratified Nested Datalog queries, called Nested nr-Datalog$^\neg$ queries. These notions are extended to Nested Datalog as follows.

Definition 7.6 (Safeness)**.** *Let $(\#q(\bar{t}), \Pi)$ be a stratified Nested Datalog query, and R be a rule in a block $\pi[\bar{X}]$ of program Π. Assume an order in the literals of rule R given by the literal subscripts $1, \dots, n+1$:*

$$L_{n+1} \leftarrow L_1, \dots, L_n$$

Then, a variable X is safe *at literal L_i of rule R, if and only if one of the following conditions holds:*

1. *$i > 1$ and X is safe at literal L_{i-1}.*
2. *$X \in \bar{X}$ (i.e., is a parameter of the block).*
3. *Literal L_i has the form $X = c$ where c is a constant.*

4. *Literal L_i has the form $X = Y$ where Y is a variable and Y is safe at literal L_{i-1}.*

5. *Literal L_i has the form $p[\bar{Y}](\bar{t})$ and $X \in \bar{t} \setminus \bar{Y}$.*

The rule R is safe if there is an order of its literals under which all variables occurring in the rule are safe.

A query Q is safe if all its rules are safe.

Intuitively, the definition of safe rule in Nested Datalog extends the definition of safe rules in Datalog by allowing unsafe variables (according to the standard Datalog criteria) be safe if they are parameters of the query to where the rule belongs to.

Example 7.6. *Consider the following Nested Datalog program.*

$$p(X) \leftarrow q(X, Y), \#r[X](Y) \tag{7.1}$$

$$r(Y) \leftarrow s(Y, Z), Z \neq X \tag{7.2}$$

$$s(Y, Z) \leftarrow \#t[Y](Z), \#u[Z](Y) \tag{7.3}$$

$$t(X) \leftarrow q(X, Y) \tag{7.4}$$

$$u(Y) \leftarrow q(Y, Z), X \neq Y \tag{7.5}$$

The rule in (7.1) is safe, because variables X and Y are safe at the first-literal of its body. The rule in (7.2) is safe because the unique variable that could be unsafe, variable X, is a parameter of the query block to where this rule belongs. The rule in (7.3) is unsafe because whatever the order of literals $\#t[Y](Z)$ and $\#u[Z](Y)$, one of the variables is unsafe. The rule in (7.4) is unsafe because variable X is unsafe since X is not safe in a previous literal nor X is a parameter of the query where this rule belongs. Hence, query $\#p(X)$ is unsafe because it has an unsafe rule.

The rule in (7.3) exemplifies an interesting case of unsafe rule. Recall that in Datalog a variable Y is safe in a rule R if Y occurs in a positive predicate atom in R or in an equality atom of the form $Y = t$, where t is a constant or a safe variable in R. In Nested Datalog that condition is not enough. Although variable Y occurs in a positive query atom $\#u[Z](Y)$, variable Y is unsafe in rule (7.3). The problem is that, atom $\#u[Z](Y)$ cannot be evaluated before the range of variable Z is restricted. Variable Z is restricted by the literal $\#t[Y](Z)$, that cannot be evaluated before the literal $\#u[Z](Y)$. The rule is unsafe because there is a deadlock between both literals (i.e. there is no way of evaluating it).

In order to define the notion of non-recursive Nested Datalog query we have to define first the notion of dependency graph of a Nested Datalog program. Recall that a Nested Datalog program Π is a set $\{\pi_1[\bar{X}_1], \ldots, \pi_n[\bar{X}_n]\}$ where each π_j is a set of rules admitting query atoms. These rules can relate predicates occurring in different blocks.

Definition 7.7 (Dependency graph)**.** *Let Π be a Nested Datalog program. Then, the dependency graph of Π is a graph whose nodes are the predicates in Π and has an edge (p, q) if and only if there is a rule where p occurs in a head and q occurs in the body either as a predicate atom or as a query atom.*

Definition 7.8 (Non-recursiveness)**.** *A Datalog program Π is said to be non-recursive if the dependency graph of Π is acyclic.*

7.2 Query atoms and external atoms

We mentioned that we borrowed the notation of *external atoms* used in the context of HEX programs [22, 23] to denote query atoms. We now explain the relation between these two notions that justifies our notation.

HEX programs are nonmonotonic logic programs extended with *higher-order atoms* as well as *external atoms*, and interpreted using the answer-set semantics. In this context, an external atom has the form $\#p[y_1, \ldots, y_m](x_1, \ldots, x_n)$ where $y_1, \ldots, y_m$ and $x_1, \ldots, x_n$ are two lists of terms (called input and output list, respectively), and $\#p$ is an *external predicate*. The set of external predicates is assumed disjoint with the set of regular predicate names and terms. The character # is used as a prefix to identify external predicate names. The semantics of an external atom with an external predicate $\#p$ is given by a $(m + n + 1)$-ary Boolean function f assigning to each tuple $(D, b_1, \ldots, b_m, a_1, \ldots, a_n)$ either true or false, where m and n are respectively the lengths of the input and output lists, D is a database, and $b_1, \ldots, b_m, a_1, \ldots, a_n$ is a list of constants. Thus, given a replacement θ, and a database D the truth value of an external atom $\#p[y_1, \ldots, y_m](x_1, \ldots, x_n)$ against replacement θ and database D is $f(D, \theta(y_1), \ldots, \theta(y_m), \theta(x_1), \ldots, \theta(x_n))$, where the domain of θ must include all variables occurring in the external atom and for each constant a is assumed that $\theta(a) = a$.

As the following definition shows, query atoms $\#p[\bar{X}](\bar{t})$ in Nested Datalog can be understood as external atoms of the program π_j where the query atom $\#p[\bar{X}](\bar{t})$ belongs.

Definition 7.9 (HEX Query of a Nested Datalog Query)**.** *Let $\bar{X} = \{X_1, \ldots, X_m\}$ be a set of variables, $\#q[\bar{X}](\bar{t})$ be a query atom occurring in a Nested Datalog program Π, and π be the block of Π where the intensional predicate q occurs. Then, $\#q[\bar{X}](\bar{t})$ can be considered an external atom whose semantics is given by the Boolean function f defined as follows:*

$$f(D, x_1, \ldots, x_m, y_1, \ldots, y_n) = \begin{cases} \textit{true} & \textit{there is a derivation three } t \in \mathrm{dt}(\Pi \cup \{\pi[\theta]\}, D) \\ & \textit{with root } (q(y_1, \ldots, y_n), \pi[\theta]), \\ \textit{false} & \textit{otherwise,} \end{cases}$$

where θ is the replacement $\{X/x_1, \ldots, X/x_m\}$.

7.3 The expressive power of Nested nr-Datalog$^\neg$

In this section we show that Nested nr-Datalog$^\neg$ has the same expressive power as nr-Datalog$^\neg$ by providing a translation among both query languages.

To compare the expressive power of both query languages we extend Datalog databases with a predicate adom$(\cdot)$ that when translating a query Q to a query Q' using a database D, codifies the active domain of D and the constants in query Q. Formally, given a query Q, a Datalog database D is extended with a fact adom(c) for each constant c occurring in Q or in D.

To show that Nested nr-Datalog$^\neg$ and nr-Datalog$^\neg$ have the same expressive power, we require some preliminary definitions.

Recall that the root of each Nested Datalog derivation tree T is a pair $(p(\bar{b}), \pi)$ where $p(\bar{b})$ is a fact, and π is a block (see Definition 7.5). When there is no ambiguity about the block (e.g., when the query has a single block) we will denote the root of tree simply as $p(\bar{b})$. From now on, we call *derived fact* of a block (or query) to a fact that occurs in the root of a derivation tree of that block (or query). Similarly, we call derived fact of a rule R to a fact that is a derived fact of the block where the rule belongs to, and the root is connected to its children with edges labeled with rule R. Observe that if a fact $p(\bar{c})$ is derived from a rule R, then rule R has head $p(\bar{Y})$ and there exists a substitution θ such that $p(\bar{c}) = \theta(p(\bar{Y}))$. We write $\theta(\bar{Y}) = \bar{c}$ to denote that $\bar{Y}$ and $\bar{c}$ are respectively a tuple of variables $(Y_1, \ldots, Y_n)$ and a tuple of constants $(c_1, \ldots, c_n)$, of the same size n, such that $\theta(Y_i) = c_i$, for $1 \leq i \leq n$. The equality $p(\bar{c}) = \theta(p(\bar{Y}))$ implies thus the equality $\theta(\bar{Y}) = \bar{c}$.

Let $\pi[\bar{X}]$ be a block of a Nested nr-Datalog$^\neg$ query Q having no query atoms in the body of the rules of block $\pi[\bar{X}]$. Then, the *de-parametrization* of block $\pi[\bar{X}]$ is the block that consists of the rules R' resulting from applying the following modifications to each rule R of block $\pi[\bar{X}]$:

1. Replace every intensional predicate atom $q(\bar{t})$ in the rule R by an intensional predicate atom of the form $q(\bar{X}, \bar{t})$.
2. Add the extensional atom $\text{adom}(X)$ to the body of R, for each variable $X \in \bar{X}$.

Intuitively, the de-parametrization of a block $\pi[\bar{X}]$ is a block that emulates $\pi[\bar{X}]$ without using parameters. This intuition is expressed in Lemma 7.1.

Lemma 7.1. *Let $\bar{X}$ be the tuple of variables, $\bar{a}$ be a tuple of constants, θ be a substitution that maps each variable $X \in \bar{X}$ to a constant $a \in \bar{a}$ (i.e., $\theta(\bar{X}) = \bar{a}$), and $\pi[\bar{X}]$ be a block of a Nested Datalog query Q having no query atoms in the body of the rules of block $\pi[\bar{X}]$. Then, the intensional fact $p(\bar{b})$ is a derived fact of block $\pi[\theta]$ if and only if $p(\bar{a}, \bar{b})$ is a derived fact of the de-parametrization of block $\pi[\bar{X}]$.*

Proof. For every derivation tree $T \in \text{dt}(\pi[\theta])$ we write T' to denote the derivation tree resulting from the following modifications on tree T:

1. Each intensional fact $q(\bar{b})$ in a node of the derivation tree T is replaced by an intensional fact of the form $q(\bar{a}, \bar{b})$.
2. Each rule R labeling an edge in T is replaced by the rule R' resulting from the de-parametrization of block $\pi[\bar{X}]$.

We next show that for each derived fact $p(\bar{a})$ of block $\pi[\theta]$ there exists a derived fact $p(\bar{a}, \bar{b})$ of the de-parametrization of block $\pi[\bar{X}]$ by induction in the height h of the derivation tree T of block $\pi[\theta]$ whose root is $p(\bar{b})$ (h is the maximum numbers of edges from the root to a leaf of the tree).

Case $h = 1$. In this case rules R and R' of derivation trees T and T' do not include intensional predicate atoms in the body. Rules R and R' have thus the following form:

$$\begin{aligned} R :=&\quad p(\bar{Z}) \leftarrow L_1, \ldots, L_n \\ R' :=&\quad p(\bar{X}, \bar{Z}) \leftarrow L_1, \ldots, L_n, \bar{X} = \bar{a} \end{aligned}$$

where $\bar{X} = \bar{a}$ denote the sets of literals of the form $X = a$ where $X \in \bar{X}$ and $a \in \bar{a}$.

Without loss of generality, assume a substitution θ_1 such that $p(\bar{b})$ is a derived fact of rule R and $\theta_1(\bar{Y}) = \bar{b}$. Then, $p(\bar{b})$ is the root of T, and for each literal L_i in the body of rule R the root of T has as child the node $\theta_1(L_i)$ and as edge label the rule R.

Let $\theta_2 = \theta_1 \cup \theta$. By construction, $\theta_2(p(\bar{X}, \bar{Z})) = p(\bar{a}, \bar{b})$ is the root of T', and the children are the nodes $\theta_2(L_i)$ and $\theta_2(X_1 = a)$. Hence, T' is a derivation tree of the de-parametrization of block $\pi[\bar{X}]$, and $p(\bar{a}, \bar{b})$ is thus a derived fact of the de-parametrization of block $\pi[\bar{X}]$.

Case $h > 1$. Like in the previous case, rules R and R' have the following form:

$$\begin{aligned} R :=&\quad p(\bar{Z}) \leftarrow L_1, \ldots, L_{m-1}, L_m, \ldots, L_n \\ R' :=&\quad p(\bar{X}, \bar{Z}) \leftarrow L_1, \ldots, L_{m-1}, L'_m, \ldots, L'_n, \theta(\bar{X}) = \bar{a} \end{aligned}$$

The difference with the base case $h = 1$, is that now literals $L_m, \ldots, L_n$ and $L'_m, \ldots, L'_n$ have intensional predicates.

Like in the previous case, assume a substitution θ_1 such that rule R derives $p(\bar{b})$, and the substitution $\theta_2 = \theta_1 \cup \theta$. As in the previous case, we have that $\theta_2(p(\bar{X}, \bar{b})) = p(\bar{a}, \bar{b})$ is the root of T' and for each literal L_i in R' it holds that $\theta_2(L_i)$ is a child of the root of T', except for the additional literals $L'_m, \ldots, L'_n$ that have intensional predicates.

By construction, for $m \leq k \leq n$, the literal L_k is the (positive or negative) predicate atom of the form $q(\bar{t})$ whereas L'_k is the (positive or negative) predicate atom of the form $q(\bar{X}, \bar{t})$.

If L_k is positive, then there is a derivation tree of block $\pi[\theta]$ with root $\theta_1(q(\bar{t}))$. Then, by induction, there is a derivation tree with root $\theta_2(q(\bar{X}, \bar{t}))$ of the de-parametrization of block $\pi[\bar{X}]$. Hence, T' is a derivation tree of the de-parametrization of block $\pi[\bar{X}]$, and $p(\bar{a}, \bar{b})$ is thus a derived fact of the de-parametrization of block $\pi[\bar{X}]$.

If L_k is negative, then there is no derivation tree of block $\pi[\theta]$ with root $\theta_1(q(\bar{t}))$. Then, by induction, there is no derivation tree with root $\theta_2(q(\bar{X}, \bar{t}))$ of the de-parametrization of block $\pi[\bar{X}]$. Hence, T' is a derivation tree of the de-parametrization of block $\pi[\bar{X}]$, and $p(\bar{a}, \bar{b})$ is thus a derived fact of the de-parametrization of block $\pi[\bar{X}]$.

So far, we have proved that if $p(\bar{b})$ is a derived fact of block $\pi[\theta]$ then $p(\bar{a}, \bar{b})$ is a derived fact of the de-parametrization of block $\pi[\bar{X}]$. The contrary implication can be shown using the same argument. □

Now we can present the promised result.

Theorem 7.1. *Nested nr-Datalog$^\neg$ has the same expressive power as nr-Datalog$^\neg$.*

Proof. The fact that nr-Datalog$^\neg$ is included in Nested nr-Datalog$^\neg$ follows from the fact that every nr-Datalog$^\neg$ is equivalent to a Nested nr-Datalog$^\neg$ query consisting in a single block with no parameters. The reverse implication, that Nested nr-Datalog$^\neg$ is included in nr-Datalog$^\neg$, can be shown by induction in the number of blocks n of the Nested nr-Datalog$^\neg$ query.

Case $n = 1$. Let Q be a Nested nr-Datalog$^\neg$ query with a single block π_1. Since Q is safe, π_1 has no parameters and is thus equivalent to a nr-Datalog$^\neg$ query.

Case $n > 1$. Let Q be a Nested nr-Datalog$^\neg$ query with $n > 1$ blocks. Since Q is stratified, there exist two blocks $\pi_i[\bar{X}]$ and $\pi_j[\bar{Y}]$ in the query such that block $\pi_i[\bar{X}]$ has no nested block, and block $\pi_i[\bar{X}]$ is nested in $\pi_j[\bar{Y}]$. Then, there is a rule R in $\pi_j[\bar{Y}]$ with a query atom of the form $\#p[\bar{X}](\bar{t})$ where p is an intensional predicate of block $\pi_i[\bar{X}]$.

We next construct a query Q' that is equivalent to Q and consists of $n-1$ blocks. Then, by induction, query Q' is expressible in nr-Datalog$^\neg$, so query Q is also expressible in nr-Datalog$^\neg$.

Let Q' be the query that results of the following modifications in query Q:

1. Delete block $\pi_i[\bar{X}]$.

2. Replace in block $\pi_j[\bar{Y}]$ the (positive or negative) query atom $\#p[\bar{X}](\bar{t})$ by the (positive or negative) predicate atom $p(\bar{X}, \bar{t})$.

3. Add all rules of the deparametrization of block $\pi_i[\bar{X}]$ to block $\pi_j[\bar{Y}]$.

By construction queries Q and Q' have the same number of rules. Indeed, for each rule R' in block $\pi_k[\bar{Y}]$ there is a rule R that belongs either to π_i or π_j such that R' is the result of translating R. Given a rule R in query Q we write R' to denote the rule of query Q' that corresponds to R.

We next show that for each derivation tree T of query Q there exists a derivation tree T' of query Q' with the same root as T. For each derivation tree T of query Q, there are three options:

1. There is no node in T with the predicate name p (i.e., the block $\pi_i[\bar{X}]$ is not involved in the derivation). In this case let T' be T. Since the block $\pi_i[\bar{X}]$ is not involved, T' is a derivation tree of Q' because T' uses the rules that are shared by both queries.

2. The query atom $\#p[\bar{X}](\bar{t})$ is positive in query Q and there is a node of the form $(p(\bar{b}), \pi_i[\theta])$ in tree T. Let $\bar{a}$ be the tuple $\theta(\bar{X})$. By definition, the subtree T_s with root is $(p(\bar{b}), \pi_i[\theta])$ is also a derivation tree of query $\pi_i[\theta]$. By Lemma 7.1, the fact $p(\bar{a}, \bar{b})$ is a derived fact of the desparametrization of block $\pi_i[\bar{X}]$. Hence, there exists a derivation tree T'_s of Q' with root $(p(\bar{a}, \bar{b}), \pi_j[\bar{Y}])$. By construction the derivation T' that results from replacing T_s by T'_s is a derivation tree of Q'.

3. The query atom $\#p[\bar{X}](\bar{t})$ is negative in query Q and there is a node of the form $(\neg(p(\bar{b}), \pi_i[\theta])$ in tree T. By the same argument that in the previous case, given $\theta(\bar{X}) = \bar{a}$, we know that $p(\bar{b})$ is not a derived fact of Q and $p(\bar{a}, \bar{b})$ is not a derived fact of Q'. Hence, the derivation tree T' resulting from replacing node $(\neg p(\bar{b}), \pi_i[\theta])$ by node $(\neg p(\bar{a}, \bar{b}), \pi_j[\bar{Y}])$ is a derivation tree of query Q'.

So far, we have proved that if T is a derivation tree of Q then T' is a derivation tree of Q'. The proof of the contrary, that if T' is a derivation tree of Q' then T is a derivation tree of Q follows the same argument. □

7.4 Conclusion

In this chapter we proposed a formalism to allow the composition of Datalog queries. This formalism extends the standard Datalog with a new type of atom, called query atom, that can be formalized using the notion external atoms by Eiter et al. [23]. We show that Nested Datalog has the same expressive power than standard Datalog. This result implicates that by defining a translation from SPARQL queries with the EXISTS clause to Nested Datalog we also define a translation to the standard Datalog.

Chapter 8

On the Nested Datalog semantics of the EXISTS clause

In this chapter we address the problem of translating SPARQL with the EXISTS clause to Nested Datalog. This translation is an extension of the one from SPARQL to Datalog presented in Chapter 6. As we show in the previous chapter, Nested Datalog and Datalog have the same expressive power. The composition of queries is thus not needed to express the EXISTS clause in Datalog. However, as we show in this section, it helps to provide a more intuitive translation.

Recall that we have several proposals for the semantics of the EXISTS clause. Thus, instead of defining a translation for a single semantics of EXISTS, in this chapter we compare how the proposals behave when translated to Nested Datalog.

Throughout this chapter we use upper case letter for SPARQL variables as in Datalog in order to simplify the translation among both languages.

Organization of this chapter

- In Section 7.1 we describe two forms of substitutions of SPARQL variables. Each of these forms is translated as a different form of substitution in Nested Datalog.
- In Section 8.2 we show that the translation of the EXISTS clause gets queries whose size is exponential in the number of variables to substitute. We thus propose a variant of Nested Datalog extended to deal with null values.
- We conclude the chapter in Section 8.3.

8.1 Two forms of substitution of SPARQL variables

Query atoms are introduced to provide an explanation for substitution in languages as SQL, SPARQL and G-Core. Because this thesis is focused on SPARQL, this section illustrates how Nested Datalog can be used to model the SPARQL EXISTS clause. We identify two forms of substitution. The first involves attributes that are not part of answers of the inner query, and that in Nested Datalog are parameters of the query atom. The second involves

attributes that are part of the answers of the inner query, and that in Nested Datalog occur as variables in the goal of the query.

8.1.1 Substitution of parameters

In a Nested Datalog query $\#q[\bar{X}](\bar{t})$ the parameters $\bar{X}$ are substituted in the block that the query refers to. Parameters can be used to represent correlated variables inside filter-conditions.

Example 8.1. *Consider the SPARQL query:*

$$P \text{ FILTER (EXISTS}(\underbrace{(X, p, Y) \text{ FILTER } (X \neq Z)}_{Q})). \tag{8.1}$$

The EXISTS clause in (8.1) has to be evaluated before substituting the inner query Q with a mapping ν. According to the translation by Angles and Gutierrez [8], fixed with the modifications proposed in Chapter 6, the inner query Q in (8.1) corresponds to the query $\#q(X, Y)$ with the Datalog program defined as follows:

$$\begin{aligned}
&q(X, Y) \leftarrow r(X, Y), \neg s(X, Y) \\
&r(X, Y) \leftarrow \text{triple}(X, p, Y) \\
&s(X, Y) \leftarrow t(X, Y) \\
&s(X, Y) \leftarrow p(X, Y) \\
&t(X, Y) \leftarrow r(X, Y), X = Z, X \neq \varnothing, Z \neq \varnothing \\
&p(X, Y) \leftarrow r(X, Y), X = \varnothing \\
&p(X, Y) \leftarrow r(X, Y), Z = \varnothing
\end{aligned} \tag{8.2}$$

Assume $\nu = \{Z \mapsto a\}$. There are two options to define the semantics of the EXISTS clause in (8.1): either Z is replaceable with a value from the outer query P or it is not replaceable. We can model that Z is replaceable by assuming that Z is a parameter of the query (i.e., we consider $\#q[Z](X, Y)$ instead of $\#q(X, Y)$ when translating the whole query). In this case the result of substituting Z by a in the inner query is the Datalog program defined as follows:

$$\begin{aligned}
&q(X, Y) \leftarrow r(X, Y), \neg s(X, Y), Z = a \\
&r(X, Y) \leftarrow \text{triple}(X, p, Y), Z = a \\
&s(X, Y) \leftarrow t(X, Y), Z = a \\
&s(X, Y) \leftarrow p(X, Y), Z = a \\
&t(X, Y) \leftarrow r(X, Y), X = Z, X \neq \varnothing, Z \neq \varnothing, Z = a \\
&p(X, Y) \leftarrow r(X, Y), X = \varnothing, Z = a \\
&p(X, Y) \leftarrow r(X, Y), Z = \varnothing, Z = a
\end{aligned} \tag{8.3}$$

For the sake of the readability intensional predicates were not renamed as is stated by Definition 7.5.

The last rule of program in (8.3) can be removed without changing the semantics of the program because literals $Z = \varnothing$ and $Z = a$ are contradictory (i.e., this rule cannot be used to infer new facts).

Likewise, the literal $Z \neq \varnothing$ can be removed from the fifth rule of the program in (8.3), since it is subsumed by the equality atom $Z = a$. That is, the fifth rule can be rewritten equivalently as:

$$t(X, Y) \leftarrow r(X, p, Y), X = Z, X \neq \varnothing, Z = a \tag{8.4}$$

Since Z does not occur in the head of this rule, it can be rewritten equivalently as follows:

$$t(X, Y) \leftarrow r(X, p, Y), X = a \tag{8.5}$$

Hence, the program in (8.3) is equivalent to the following Datalog program:

$$\begin{aligned} q(X, Y) &\leftarrow r(X, Y), \neg s(X, Y) \\ r(X, Y) &\leftarrow \text{triple}(X, p, Y) \\ s(X, Y) &\leftarrow t(X, Y) \\ s(X, Y) &\leftarrow p(X, Y) \\ t(X, Y) &\leftarrow r(X, Y), X = a \\ p(X, Y) &\leftarrow r(X, Y), X = \varnothing \end{aligned} \tag{8.6}$$

Let us find the semantics of EXISTS *in that agrees with the program 8.6 we obtained using the Nested Datalog substitution. According to the environment-binding proposal (see Definition 5.6), the result of substituting v in the inner query in (8.1) is the following SPARQL query:*

$$(\{X \mapsto a\} \text{ AND } (X, p, Y)) \text{ FILTER } (X \neq a). \tag{8.7}$$

The translation of the SPARQL query in (8.7) is the Datalog query $q(X, Y, Z)$ whose program is the following:

$$\begin{aligned} q(X, Y, Z) &\leftarrow r(X, Y, Z), \neg s(X, Y) \\ r(X, Y, Z) &\leftarrow w(X, Y), Z = a \\ w(X, Y) &\leftarrow \text{triple}(X, p, Y) \\ s(X, Y) &\leftarrow t(X, Y, Z) \\ s(X, Y) &\leftarrow p(X, Y, Z) \\ t(X, Y, Z) &\leftarrow r(X, Y, Z), X = a \\ p(X, Y, Z) &\leftarrow r(X, Y, Z), X = \varnothing \end{aligned} \tag{8.8}$$

It is not difficult to see that program in 8.6 can infer a fact of the form $p(c_1, c_2)$ if and only if there exists a constant c_3 such that the program in 8.8 can infer the fact of the form $p(c_1, c_2, c_3)$. Hence, in this case the semantics of substitution in Nested Datalog coincides with substitution in SPARQL according to the environment-binding proposal.

Example 8.1 shows that the environment-binding proposal can, in some form, be explained by the use of the (fixed) translation by Angles and Gutierrez [8] to model nested queries. However, it requires a different formalization because in the rules produced by their translation all variables are local to rules, and can thus be renamed without changing the semantics of rules. In other words, Datalog does not provide a method to indicate what variables occurring in different rules are the same variable. Instead, Nested Datalog allows expressing this extended scope of variables. The following two examples show how to use Nested Datalog to enrich the translation by Angles and Gutierrez [8] in order to express the EXISTS clause.

Example 8.2. *Consider the SPARQL query:*

$$\underbrace{(X, p, Y)}_{P} \text{ FILTER(EXISTS}(\underbrace{(X, q, Y) \text{ AND (SELECT } X \text{ WHERE } (X, r, Y))}_{Q})). \tag{8.9}$$

According to the translation by Angles and Gutierrez [8] the inner query Q of this SPARQL query is translated as the Datalog query $q(X, Y)$ over the following program:

$$\begin{aligned} q(X, Y) &\leftarrow r(X_1, Y), s(X_2), \text{comp}(X, X_1, X_2) \\ r(X, Y) &\leftarrow \text{triple}(X, q, Y) \\ s(X) &\leftarrow t(X, Y) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \end{aligned} \tag{8.10}$$

However, their translation does not provide information on whether the occurrences of variable Y in this program can be considered the same variable or not. Nested Datalog can express this relation. If one wants both occurrences of Y to be the same variable, then we can translate the inner query Q as the Nested Datalog query $\#q(X, Y)$ over the program consisting in the two following blocks:

$$\begin{aligned} q(X, Y) &\leftarrow r(X_1, Y), \#s[Y](X_2), \text{comp}(X, X_1, X_2) \\ r(X, Y) &\leftarrow \text{triple}(X, q, Y) \end{aligned} \tag{8.11}$$

$$\begin{aligned} s(X) &\leftarrow t(X, Y) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \end{aligned} \tag{8.12}$$

In this case, the variable Y in the block (8.11) is the same that variable than Y in the block (8.12) because the scope of Y is the scope of the parameter Y in the query atom $\#s[Y](X_2)$.

On the contrary, if we want all occurrences of Y to denote different variables, we can translate the inner query Q as the Nested Datalog query $\#q(X, Y)$ over the program consisting in the following two blocks:

$$\begin{aligned} q(X, Y) &\leftarrow r(X_1, Y), \#s(X_2), \text{comp}(X, X_1, X_2) \\ r(X, Y) &\leftarrow \text{triple}(X, q, Y) \end{aligned} \tag{8.13}$$

$$\begin{aligned} s(X) &\leftarrow t(X, Y) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \end{aligned} \tag{8.14}$$

In this case we can rename each occurrence of Y without changing the semantics of the query.

Example 8.2 shows how Nested Datalog enriches the scope of variables in Datalog by allowing variables to be scoped beyond rules. At the same time, Nested Datalog allows us to express that variables are local by enclosing them in a nested query. In Example 8.2 the three proposals (shallow-binding, deep-binding and environment-binding) agree in that variable Y in the inner query is local. It is relevant to note that not all engines agree on that [40]. The following example shows a case where the three proposals do not agree.

Example 8.3. *Consider the SPARQL query:*

$$\underbrace{(X, p, Y)}_{P} \text{ FILTER(EXISTS}(\underbrace{(X, q, Z) \text{ MINUS } (X, r, Y)}_{Q})). \tag{8.15}$$

According to the translation by Angles and Gutierrez [8] the inner query Q is translated as the query $q(X, Z)$ over the following program:

$$\begin{aligned} q(X, Z) &\leftarrow r(X_1, Z), \neg s(X_2), \text{comp}(X, X_1, X_2) \\ r(X, Z) &\leftarrow \text{triple}(X, q, Z) \\ s(X) &\leftarrow t(X, Y) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \end{aligned} \tag{8.16}$$

As in Example 8.2 this translation does not indicate if variable Y in the filter condition of the SPARQL query in (8.15) is or is not replaceable with the value of Y in the outer query P. In the Datalog program in (8.16) the variable Y is not replaceable because its scope is the rule where it occurs. The formalism of Nested Datalog can be used to overcome this Datalog limitation. Indeed, to codify that variable Y is replaceable, the inner query Q can be translated as the Nested Datalog query $\#q(X, Z)$ over the program composed by the following two blocks:

$$\begin{aligned} q(X, Z) &\leftarrow r(X_1, Z), \neg \#s[Y](X_2), \text{comp}(X, X_1, X_2) \\ r(X, Z) &\leftarrow \text{triple}(X, q, Z) \end{aligned} \tag{8.17}$$

$$\begin{aligned} s(X) &\leftarrow t(X, Y) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \end{aligned} \tag{8.18}$$

On the contrary, if variable Y is not replaceable, then the inner query Q can be translated as the Nested Datalog query $\#q(X, Z)$ over the program consisting in the following two blocks:

$$\begin{aligned} q(X, Z) &\leftarrow r(X_1, Z), \neg \#s(X_2), \text{comp}(X, X_1, X_2) \\ r(X, Z) &\leftarrow \text{triple}(X, q, Z) \end{aligned} \tag{8.19}$$

$$\begin{aligned} s(X) &\leftarrow t(X, Y) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \end{aligned} \tag{8.20}$$

Example 8.3 describe alternatives that need be considered to translate the alternative semantics for EXISTS we discussed in this thesis. Indeed, according to the shallow-binding and deep-binding proposals, the variable Y of this example is not visible from the outer query. On the contrary, according to the environment-binding proposal the variable Y is visible from the outer query.

8.1.2 Substitution of goal variables

We showed that Nested Datalog queries can be used to indicate what variables are visible from an outer query. This includes the variables that occur as parameters of a query atom. Now we discuss what happens with variables that occur as arguments of a query atom (i.e., in the parenthesis).

Example 8.4. *Consider the following SPARQL query:*

$$P \text{ FILTER (EXISTS}(\underbrace{(X, r, Y) \text{ MINUS } (X, s, Z)}_{Q})), \tag{8.21}$$

and assume the mapping $\nu = \{Y \mapsto a\}$ from the outer query P. According to the shallow-binding and deep-binding proposals, the inner query is substituted respectively as follows:

$$\{Z \mapsto a\} \text{ AND } ((X, r, Y) \text{ MINUS } (X, s, Z)) \qquad \textit{shallow-binding}, \tag{8.22}$$

$$(\{Z \mapsto a\} \text{ AND } (X, r, Y)) \text{ MINUS } (X, s, Z) \qquad \textit{deep-binding}. \tag{8.23}$$

The translations of the SPARQL queries Q, (8.22) and (8.23) to Datalog according to the translation by Angles and Gutierrez [8], are Datalog queries with the head $q(X, Y)$ over the following respective programs:

$$\begin{aligned} q(X, Y) &\leftarrow t(X_1, Y), \neg u(X_2), \text{comp}(X, X_1, X_2) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \\ u(X) &\leftarrow v(X, s, Z) \end{aligned} \qquad Q, \tag{8.24}$$

$$\begin{aligned} q(X, Y) &\leftarrow w(X, Y), Y = a \\ w(X, Y) &\leftarrow t(X_1, Y), \neg u(X_2), \text{comp}(X, X_1, X_2) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \\ u(X) &\leftarrow v(X, s, Z) \end{aligned} \qquad \textit{shallow-binding}, \tag{8.25}$$

$$\begin{aligned} q(X, Y) &\leftarrow t(X_1, Y), \neg u(X_2), \text{comp}(X, X_1, X_2) \\ t(X, Y) &\leftarrow w(X, Y), Y = a \\ w(X, Y) &\leftarrow \text{triple}(X, r, Y) \\ u(X) &\leftarrow v(X, s, Z) \end{aligned} \qquad \textit{deep-binding}. \tag{8.26}$$

The Datalog programs (8.25) and (8.25) represent two different places where the variable Y can be replaced by the value a. The Datalog program (8.24) does not codify the place where Y is substituted. Nested Datalog allows to overcome this limitation. For instance, the SPARQL query Q can be translated to Nested Datalog as a query $\#q[Y](X, Z)$ over the following two alternative programs:

$$\begin{aligned} q(X, Y) &\leftarrow t(X_1, Y), \neg u(X_2), \text{comp}(X, X_1, X_2) \\ t(X, Z) &\leftarrow \text{triple}(X, r, Z) \\ u(X) &\leftarrow v(X, s, Z) \end{aligned} \qquad \textit{shallow-binding}, \tag{8.27}$$

$$\begin{aligned} q(X, Z) &\leftarrow t(X_1, Z), \neg u(X_2), \text{comp}(X, X_1, X_2) \\ t(X, Y) &\leftarrow \text{triple}(X, r, Y) \\ u(X) &\leftarrow v(X, s, Z) \end{aligned} \qquad \textit{deep-binding}, \tag{8.28}$$

The programs (8.27) and (8.28) are similar to program (8.24), except that the variable Y is renamed differently in the first and the second rules of both programs. It is not difficult to see that the programs that result from substituting Y in programs (8.27) and (8.28) are equivalent to programs (8.25) and (8.26). Hence, Nested Datalog allows expressing both semantics for EXISTS in a simple and intuitive way.

Observe that all variables named by Y in the program (8.24) of Example 8.4 are somehow connected. Despite all of them being universally quantified, and thus we can consistently

rename the variables, the value used to instantiate them each time a fact is inferred has to be the same. Connection between variables can be defined syntactically by the positions in which a variable occurs. In [39] we called this notion *logical connection* between variables. This notion is formalized as follows:

Definition 8.1 (Logical connection)**.** *Let Q be a Nested Datalog query whose variables where renamed to avoid using the same name in different scopes. Then, the relation of being logically connected defined in the set of variables of Q, is the minimal partial order such that given two variables X and Y occurring respectively in rules R_X and R_Y, X is logically connected with Y, denoted $X \rightsquigarrow Y$, if one of the following conditions holds:*

1. *R_X and R_Y are in the same block, there is a predicate atom $p(\bar{t})$ in the body of R_X, X is in the i-th position of tuple $\bar{t}$, the head of R_Y is $p(\bar{s})$, and Y is in the i-th position of tuple $\bar{s}$.*
2. *There is a query atom $\#p[\bar{Z}](\bar{t})$ in the body of R_X, X is in the i-th position of tuple $\bar{t}$, R_Y is a rule of block of the query atom $\#p[\bar{Z}](\bar{t})$, whose head is $p(\bar{s})$, and Y is in the i-th position of $\bar{s}$.*
3. *There is a variable Z in the rule R_X where X occurs such that the atom $X = Z$ occurs in R_X and $Z \rightsquigarrow Y$.*

The following example shows the relevance of logically connected variables regarding substitution in SPARQL.

Example 8.5. *Consider the following Nested Datalog program Π consisting in two blocks:*

$$p(X) \leftarrow t(X), \#q(X) \tag{8.29}$$

$$\begin{aligned} q(Y) &\leftarrow r(Z, Y) \\ r(U, V) &\leftarrow s(U, V) \end{aligned} \tag{8.30}$$

To evaluate the query $\#p(X)$ over this program we can first find a substitution θ that makes true the literal $t(X)$. Assume that $\theta(X) = c$. We can then find solutions for the query atom $\#q(X)$ where X takes the value c. To solve the query atom we need no substitution. However, the evaluation of the query atom $\#q(X)$ could be optimized if we constraint X to the value c inside the block (8.30). Instead of evaluating the block (8.30), we can alternatively evaluate the blocks (8.31) and (8.32) defined as follows:

$$\begin{aligned} q(Y) &\leftarrow r(Z, Y), Y = c \\ r(U, V) &\leftarrow s(U, V) \end{aligned} \tag{8.31}$$

$$\begin{aligned} q(Y) &\leftarrow r(Z, Y) \\ r(U, V) &\leftarrow s(U, V), V = c \end{aligned} \tag{8.32}$$

The evaluation of query $\#q(X)$ is in programs (8.31) and (8.32) returns the same answers because variables X, Y and V are logically connected, more precisely $X \rightsquigarrow Y$ and $Y \rightsquigarrow V$. The logical connection between these variables ensures that the semantics of the inner program does not change after restricting also the connected variables.

Shadow-binding and deep-binding proposals for the substitution in SPARQL can be viewed as alternatives to substitute logically connected variables. The variable Y that occurs in the first and second rules of the program (8.24) in Example 8.4 are logically

connected. The example shows that one proposal substitutes variable Y in the first rule, while the other proposal substitutes variable Y in the second rule.

The following theorem shows how to substitute variables that are logically connected without changing the semantics of the query.

Theorem 8.1. *Let Π be a Nested Datalog program, and R be an arbitrary rule occurring in Π. Without loss of generality, the literals in the rule R can be organized as follows:*

$$L_{n+1} \leftarrow L_1, \ldots, L_m, X = a, \ldots, L_n$$

where literal L_m has either a predicate atom $q(\bar{t})$ or a query atom $\#q[\bar{Y}](\bar{t})$ where variable X occurs in the tuple $\bar{t}$. Then, Π is equivalent to the program that results from:

1. *removing the literal $X = a$ from rule R,*
2. *adding a literal $Y = a$ for each rule R' where the predicate of the head is q, variable Y occurs in the head, and $X \rightsquigarrow Y$, and*
3. *leaving the other rules unmodified.*

Proof. Let Π be a Nested Datalog program. That includes the following rules:

$$\begin{aligned} L_{n+1} &\leftarrow L_1, \ldots, L_{m-1}, q(t_1, \ldots, t_k), t_j = a, \ldots, L_n \\ q(t_1^1, \ldots, t_k^1) &\leftarrow B_1 \\ &\vdots \\ q(t_1^r, \ldots, t_k^r) &\leftarrow B_r \end{aligned}$$

where $1 \leq j \leq k$, no other rule of Π has predicate q in the head, and $B_1, \ldots, B_r$ denote sets of literals. Let Π' be the program that results from replacing the rules above with the following rules:

$$\begin{aligned} L_{n+1} &\leftarrow L_1, \ldots, L_{m-1}, q(t_1, \ldots, t_k), \ldots, L_n \\ q(t_1^1, \ldots, t_k^1) &\leftarrow B_1, t_j^1 = a \\ &\vdots \\ q(t_1^r, \ldots, t_k^r) &\leftarrow B_r, t_j^r = a \end{aligned}$$

Let R and R' denote the first rules of the respective programs Π and Π' in the order they are presented above. To prove that programs Π and Π' are equivalent we have to demonstrate that a substitution θ makes true all literals in rule R if and only if θ makes true all literals in rule R'. Since, the literals of R' are subsumed by the literals of R, it holds that if θ makes true all literals in R then θ makes true all literals in R'. To prove the converse, we have to show that if θ makes true all literals in R' it also makes true the equality atom $t_j = a$. Since θ makes true the literal $q(t_1, \ldots, t_k)$ the fact $q(\theta(t_1), \ldots, \theta(t_k))$ must be inferred by one of the rules of Π' of the form:

$$q(t_1^1, \ldots, t_k^1) \leftarrow B_1, t_j^1 = a$$

then according to the equality atom $t_j^1 = a$ it holds that $\theta(t_j) = a$. Thus, θ makes true the literal $t_j = a$. Hence, we proved the converse.

The proof for the query atom follows the same argument. □

Intuitively, Theorem 8.1 states that substitutions can be moved across logically connected variables, even when rules of different blocks are involved. If one needs to move substitutions from the top rule of a program to the bottom, it suffices applying the transformation described by Theorem 8.1 as much times as it is needed.

Example 8.6. *The following Nested Datalog programs are equivalent:*

$p(X) \leftarrow q(X), r(X), X = a$	$p(X) \leftarrow q(X), r(X)$	$p(X) \leftarrow q(X), r(X)$
$q(Y) \leftarrow v(Y)$	$q(Y) \leftarrow v(Y), Y = a$	$q(Y) \leftarrow v(Y)$
$r(Z) \leftarrow s(Z)$	$r(Z) \leftarrow s(Z)$	$r(Z) \leftarrow s(Z), Z = a$
$r(U) \leftarrow \#t(U)$	$r(U) \leftarrow \#t(U)$	$r(U) \leftarrow \#t(U)$
$t(V) \leftarrow u(V)$	$t(V) \leftarrow u(V)$	$t(V) \leftarrow u(V), V = a$

In fact, these three Nested Datalog programs show different places where variable X can be restricted to take the value a. All the substitutions done are done in variables that are logically connected. The equivalence between the three programs follows from Theorem 8.1.

Example 8.4 showed that the shallow-binding and the deep-binding proposals can be seen in Nested Datalog as alternative places where to substitute logically connected variables. Theorem 8.1 shows that the place where a variable is substituted does not affect the semantics of the query. However, it is well known that shallow-binding and deep-binding proposals can produce different results (see the examples in Section 5.1).

We will show in the following section that extending Nested Datalog with null values provides a better explanation for substitution in SPARQL, and in particular shows that the differences between both proposals is due to the presence of unbound values in the substitution. The following example shows one of the complications due to unbound values.

Example 8.7. *Consider the following SPARQL query:*

$$P \text{ FILTER } (\text{EXISTS}(\underbrace{((X, u, Y) \text{ OPT } (Y, v, Z)) \text{ AND } (X, w, Z)}_{Q})), \tag{8.33}$$

and assume the mapping $\nu = \{Z \mapsto a\}$ from the outer query P. According the translation by Angles and Gutierrez [8], the SPARQL query Q is translated to the Datalog query $p(X, Y, Z)$ over the following program:

$$\begin{aligned}
p(X, Y, Z) &\leftarrow q(X_1, Y, Z_1), r(X_2, Z_2), \text{comp}(X, X_1, X_2), \text{comp}(Z, Z_1, Z_2) \qquad (8.34)\\
q(X, Y, Z) &\leftarrow s(X, Y_1), t(Y_2, Z), \text{comp}(Y, Y_1, Y_2)\\
q(X, Y, Z) &\leftarrow s(X, Y_1), \neg u(Y_2), \text{comp}(Y, Y_1, Y_2)\\
s(X, Y) &\leftarrow \text{triple}(X, u, Y)\\
t(Y, Z) &\leftarrow \text{triple}(Y, v, Z)\\
u(y) &\leftarrow t(Y, Z)\\
r(X, Y) &\leftarrow \text{triple}(X, w, Z)
\end{aligned}$$

If we evaluate the inner query according to the deep-binding proposal, then variable Z has to be substituted in the triple patterns of the SPARQL query. In the Datalog version of the query this means substituting Z in the rules where the predicate triple *includes the variable Z. Of course this depends on the name of variables we use in the Datalog program. Here we are naming the variables in the Datalog program as are named in the parts of the SPARQL query corresponding to each rule. However, this naming of the variables is arbitrary because in Datalog variables can be renamed consistently without changing the semantics of the program since they are scoped to rules. If we preserve this naming of the variables, the substitution according to the deep-binding proposal produces the following program:*

$$
\begin{aligned}
p(X,Y,Z) &\leftarrow q(X_1,Y,Z_1), r(X_2,Z_2), \mathrm{comp}(X,X_1,X_2), \mathrm{comp}(Z,Z_1,Z_2) \\
q(X,Y,Z) &\leftarrow s(X,Y_1), t(Y_2,Z), \mathrm{comp}(Y,Y_1,Y_2) \\
q(X,Y,Z) &\leftarrow s(X,Y_1), \neg u(Y_2), \mathrm{comp}(Y,Y_1,Y_2) \\
s(X,Y) &\leftarrow \mathrm{triple}(X,u,Y) \\
t(Y,Z) &\leftarrow \mathrm{triple}(Y,v,Z), Z = a \\
u(y) &\leftarrow t(Y,Z) \\
r(X,Z) &\leftarrow \mathrm{triple}(X,w,Z), Z = a
\end{aligned}
\tag{8.35}
$$

On the contrary, if we assume the shallow-binding proposal, then we have to substitute Z in the first rule. This substitution thus produces the following program:

$$
\begin{aligned}
p(X,Y,Z) &\leftarrow q(X_1,Y,Z_1), r(X_2,Z_2), \mathrm{comp}(X,X_1,X_2), \mathrm{comp}(Z,Z_1,Z_2), Z = a \\
q(X,Y,Z) &\leftarrow s(X,Y_1), t(Y_2,Z), \mathrm{comp}(Y,Y_1,Y_2) \\
q(X,Y,Z) &\leftarrow s(X,Y_1), \neg u(Y_2), \mathrm{comp}(Y,Y_1,Y_2) \\
s(X,Y) &\leftarrow \mathrm{triple}(X,u,Y) \\
t(Y,Z) &\leftarrow \mathrm{triple}(Y,v,Z) \\
u(y) &\leftarrow t(Y,Z) \\
r(X,Z) &\leftarrow \mathrm{triple}(X,w,Z)
\end{aligned}
\tag{8.36}
$$

Programs (8.35) and (8.36) are not equivalent. To see this, consider the RDF graph

$$G = \{(b,u,c),(c,v,b),(b,w,a)\}.$$

The facts $q(b,c,\varnothing)$ and $r(b,a)$ are inferred from program (8.35) in the graph G. Since $\varnothing$ and a are compatible, also fact $p(b,c,a)$ is inferred. Hence, query $p(X,Y,Z)$ has an answer in program (8.35). On the contrary, the facts $q(b,c,b)$ and $r(b,a)$ are inferred from program (8.36) in the graph G. Since b and a are incompatible, query $p(X,Y,Z)$ has no answers in program (8.36).

This difference does not contradict Theorem 8.1 because variables Z occurring in the predicate triple *are not logically connected with the variable Z in the first rule. The introduction of the predicate atom* $\mathrm{comp}(Z,Z_1,Z_2)$ *precludes the logical connection between variable Z and variables Z_1 and Z_2. In fact, they cannot be logically connected since they can refer to different values. This is the case when Z_1 refers to an unbound value and Z refers to a constant a.*

Intuitively, the literal $\mathrm{comp}(Z,Z_1,Z_2)$ implies some connection of variable Z with variables Z_1 and Z_2. However that connection is not strong enough to guarantee that we can move substitutions across connected variables without changing the semantics of the program as

is stated by Theorem 8.1 for logical connections. If the literal $\text{comp}(Z, Z_1, Z_2)$ were replaced by literals $Z = Z_1$ and $Z = Z_2$ then variable Z in the first rule would be logically connected with variable Z in the predicate triple, and thus both programs would be equivalent.

Informally, the predicate $\text{comp}(Z, Z_1, Z_2)$ is combining the information about Z that comes from two different sources. In some way, an unbound represents incomplete knowledge about the value of Z, and the predicate $\text{comp}(Z, Z_1, Z_2)$ combines the information, even where both sources are not equally informative.

8.2 Nested Datalog with nulls

This section presents Nested Datalog with nulls, as an extension to Nested Datalog that gives a special semantics to null values. Recall that the translation by Angles and Gutierrez [8] to Datalog reserves a constant, denoted $\varnothing$, to represent the SPARQL unbound variables. However, the symbol $\varnothing$ has no special semantics. Instead, the rules emulate the behavior of unbound variables.

The introduction of null values in Nested Datalog is motivated by the need to provide a simpler translation from SPARQL to Datalog. One could argue that there is no need to consider null values in Datalog since the translation by Angles and Gutierrez [8] does not assume a special semantics for the $\varnothing$ symbol. However, to translate the EXISTS clause there is an exponential blow-up that is illustrated in the following Example.

Example 8.8. *Consider the following SPARQL query*

$$P \text{ FILTER (EXISTS}(\underbrace{(X, p, Z) \text{ MINUS } (X, q, Y)}_{R})), \tag{8.37}$$

where P is a query with $\text{dom}(P) = \{X, Y\}$, and let the environment-binding proposal be the semantics of this query. A first attempt to translate this query to a Nested Datalog query according the assumed semantics, is one of the form $\#q(X, Y)$ whose program Π consists is the following two blocks:

$$q(X, Y) \leftarrow p(X, Y), \#r[Y](X, Z) \tag{8.38}$$

$$\begin{aligned} r(X, Z) &\leftarrow s(X_1, Z), \neg t(X_2), \text{comp}(X, X_1, X_2) \\ s(X, Z) &\leftarrow \text{triple}(X, p, Z) \\ t(X) &\leftarrow u(X, Y) \\ u(X, Y) &\leftarrow \text{triple}(X, q, Y) \end{aligned} \tag{8.39}$$

The Nested Datalog queries $\#q(X, Y)$, $\#p(X, Y)$, and $\#r[Y](X, Z)$ over the program Π codify the respective SPARQL queries (8.37), P and R.

Despite the fact that the Nested Datalog block (8.39) is the translation of the SPARQL query R according to the translation by Angles and Gutierrez [8], the translation does not work when the predicate atom $p(X, Y)$ has an answer where variable Y is unbound. In SPARQL, if Y is unbound then Y does not have to be substituted in the inner query R. A translation that considers both cases

for the value of variable Y (bound and unbound) is the Nested Datalog query over the following program:

$$
\begin{aligned}
q(X, Y) &\leftarrow p(X, Y), Y \neq \varnothing, \#r_1[Y](X, Z) \\
q(X, Y) &\leftarrow p(X, Y), Y = \varnothing, \#r_2(X, Z)
\end{aligned}
\tag{8.40}
$$

$$
\begin{aligned}
r_1(X, Z) &\leftarrow s_1(X_1, Z), \neg t_1(X_2), \mathrm{comp}(X, X_1, X_2) \\
s_1(X, Z) &\leftarrow \mathrm{triple}(X, p, Z) \\
t_1(X) &\leftarrow u_1(X, Y) \\
u_1(X, Y) &\leftarrow \mathrm{triple}(X, q, Y)
\end{aligned}
\tag{8.41}
$$

$$
\begin{aligned}
r_2(X, Z) &\leftarrow s_2(X_2, Z), \neg t_2(X_2), \mathrm{comp}(X, X_2, X_2) \\
s_2(X, Z) &\leftarrow \mathrm{triple}(X, p, Z) \\
t_2(X) &\leftarrow u_2(X, Y) \\
u_2(X, Y) &\leftarrow \mathrm{triple}(X, q, Y)
\end{aligned}
\tag{8.42}
$$

The equality atoms $Y \neq \varnothing$ and $Y = \varnothing$ filter solutions where Y is and is not substituted. To substitute Y the query atom must include Y as a parameter. In general several variables that can be unbound. This generates the exponential blow-up in the Nested Datalog query resulting of the translation of a SPARQL query.

We next define the syntax of Nested Datalog extended with null values.

Definition 8.2 (Syntax of Nested Datalog with nulls)**.** *The syntax of Nested Datalog with nulls is the same of Nested Datalog. That is, a Nested Datalog with nulls query is a Nested Datalog query.*

The main difference of Nested Datalog with and without nulls is the form in what substitution is performed.

Definition 8.3 (Semantics of Nested Datalog with null values)**.** *The semantics of Nested Datalog with null values is the same as for Nested Datalog, except that substitution does not add an equality atom $X = c$ to substitute variable X by constant c in a rule R, but performs the following modifications in R:*

1. *replace all occurrences of X in the body of R by a fresh variable Y, and*
2. *add the literal* $\mathrm{comp}(X, Y, c)$ *to the body of R, where* comp *is the predicate that compute the compatibility according to the compatibility program (see Definition 6.3).*

Example 8.9. *Consider the SPARQL query*

$$
P \text{ FILTER (EXISTS}((X, p, Z) \text{ MINUS } (X, q, Y))) \tag{8.43}
$$

from Example 8.8, where P is a query with $\mathrm{dom}(P) = \{X, Y\}$*, and let the environment-binding proposal be the semantics of this query. Example 8.8 showed that this query can produce an exponential blow-up. Nested Datalog with null avoids the exponential blow-up by translating this*

query as the Nested Datalog query with nulls $\#q(X, Y)$ over the following program:

$$q(X, Y) \leftarrow p(X, Y), \#r[Y](X, Z) \tag{8.44}$$

$$\begin{aligned} r(X, Z) &\leftarrow s(X_1, Z), \neg t(X_2), \text{comp}(X, X_1, X_2) \\ s(X, Z) &\leftarrow \text{triple}(X, p, Z) \\ t(X) &\leftarrow u(X, Y) \\ u(X, Y) &\leftarrow \text{triple}(X, q, Y) \end{aligned} \tag{8.45}$$

If the predicate atom $p(X, Y)$ is true with a substitution θ where $\theta(Y)$ is a constant c, then we have to evaluate the inner query after substituting in it the parameter Y by c. The result is the query $r(X, Z)$ over the following program:

$$\begin{aligned} r(X, Z) &\leftarrow s(X_1, Z), \neg t(X_2), \text{comp}(X, X_1, X_2) \\ s(X, Z) &\leftarrow \text{triple}(X, p, Z) \\ t(X) &\leftarrow u(X, Y) \\ u(X, Y) &\leftarrow \text{triple}(X, q, Y'), \text{comp}(Y, Y', c) \end{aligned} \tag{8.46}$$

On the contrary, if $\theta(Y)$ is unbound, then the result of substituting the parameter Y is the query $r(X, Z)$ over the following program:

$$\begin{aligned} r(X, Z) &\leftarrow s(X_1, Z), \neg t(X_2), \text{comp}(X, X_1, X_2) \\ s(X, Z) &\leftarrow \text{triple}(X, p, Z) \\ t(X) &\leftarrow u(X, Y) \\ u(X, Y) &\leftarrow \text{triple}(X, q, Y'), \text{comp}(Y, Y', \varnothing) \end{aligned} \tag{8.47}$$

This last program is equivalent to the one that does not substitute Y, because the literal $\text{comp}(Y, Y', \varnothing)$ *ensures that $Y = Y'$. Hence, the semantics of Nested Datalog with nulls permits the expression of both alternatives described in Example 8.8, and thus eliminates the exponential blow-up.*

So far, we showed several examples that suggest that Nested Datalog offers a simpler codification of SPARQL queries with EXISTS clauses than Datalog, and that Nested Datalog with nulls offers a codification of SPARQL queries with EXISTS clauses that is even simpler than that provided by Nested Datalog. Of course, these examples are not conclusive to prove the convenience of Nested Datalog with nulls. This requires a complete translation from SPARQL to Nested Datalog and Nested Datalog with nulls. The definition of this translation is out of the scope of this thesis.

8.3 Conclusions

The translation of the EXISTS clause to Datalog is possible but not clean, because it requires rules that have no direct counterpart in Datalog. The main complication is that in SPARQL variables with different scopes can have the same name. Despite variables are naming different resources, two proposals (deep-binding and environment-binding) allow replacing them as if they were the same variable. This problem does not affect Datalog because

variables are scoped to rules. To bridge this gap we use Nested Datalog. However, this extension was not enough to define an intuitive translation for Datalog. We thus proposed another extension, consisting in introducing null values as a special type of value. We see two interesting avenues in this regard: to study the properties of Nested Datalog with the extension to deal with null values, and to formalize the three proposals for the semantics of EXISTS studied herein using a translation to that Datalog extension.

Part IV

Conclusions

Chapter 9

Conclusions and future work

We are now at the end of the effort to understand the problem of incomplete data in SPARQL. Let us sketch what lessons we have learned and what are the paths to follow for the future.

The thesis addressed the problem of incomplete data in SPARQL. The idea that incomplete data plays a central role in SPARQL is not new. Indeed, in an earlier work trying to formalize SPARQL, Cyganiak [21] claimed that the treatment of null values is the major problem encountered when trying to specify the semantics of SPARQL using standard relational algebra. To address this problem we reviewed the problem of incomplete data in the relational model. In particular, we reviewed two forms to incorporate incompleteness to relations: null values (unknown values, inapplicable values, and no-information values) and maybe-tuples. We described then the query languages for databases with these types of incompleteness and the techniques to approximate certain answers in databases with unknown values.

The hypothesis we presented in this thesis was that we can apply relational model techniques dealing with incomplete data to the problems found in SPARQL. This hypothesis resulted valid in some cases. We showed the feasibility of applying the known techniques to the approximation of certain answers in SPARQL when considering blank nodes as null values. However, the application of these techniques for the case of unbound values was elusive, because SPARQL operators were not aligned with a known formalism. Hence, the problem of formalizing the semantics of unbound values with relational model techniques remains open for the practical and theoretical concerns.

We ended the thesis with the study of the semantics of the EXISTS clause. We showed that several of the issues arising from this clause were related to the notion of incomplete data. However, due to the complicate design of the notion of substitution in SPARQL, we chose to study first this notion in an alternative formalism without the nuances of SPARQL and then to address the real case.

The thesis left several new understandings that we will summarize as follows.

A first lesson arising from this thesis is that the SPARQL operators have too many nuances that complicate the design and implementation of substitution. To address this problem,

it is better first to address the fundamental problem of substitution in a cleaner query language and then see how it can be implemented in SPARQL. We did this by proposing Nested Datalog, an extended version of Datalog that allows the composition of queries. We develop this formalisms and derived the main properties regarding querying and show how they relate to SPARQL and SQL. However, we did not focus in the systematization of this relation nor their properties. This task—by its scope—is an interesting future work per se.

Second. This thesis shows that the SPARQL query language can be improved by allowing the approximation of certain answers. Indeed, we showed experimentally that these approximations can be done without incurring in much extra cost, and that they provide a better integration with the RDF semantics. However, there are two questions to be addressed before implementing the approximations of certain answers in SPARQL. The first is to have a workable notion of what a good approximation is, and the second is determining how relevant are certain answers for the Semantic Web. We think that we still need more knowledge about the notion of possible approximations and their costs in order to answer the first question. Regarding the second problem, again it seems that there is still more knowledge needed. We surveyed the Wikidata example queries, and found not enough queries for which it would be relevant to fix the SPARQL semantics. Thus, the question if this problem is really relevant in current practice remains open.

In the third place, we learned that the semantics of SPARQL still needs precisions in order to address issues like the one we studied in this thesis. Throughout this thesis we cited several efforts to formalize SPARQL. Some of these efforts report the ambiguities of the specification and others interpret the specification in forms that differ from other works. For example, Kaminski et al. [44] claimed that their semantics for the EXISTS clause is compatible with the specification in the cases where there is no ambiguity, and also works for the cases where the specification is ambiguous. However, their formalization—called environment-binding in this thesis— does not agree with the interpretation that an editor of the specification, Andy Seaborne, gives to the EXISTS clause—called deep-binding. Other efforts provide valuable knowledge about SPARQL, but their proposals are not free of mistakes induced by the complexities of SPARQL. This is the case of the translation from SPARQL to Datalog by Angles and Gutierrez [8], that we improved in this thesis in order to fix the cases in which their translation does not work. These examples show how complex SPARQL is.

Finally, another conclusion from this thesis has to do with the use of new formalisms to address the semantics of (incompleteness in) SPARQL. The relation among the generalized relational algebra, the algebra of maybe tables by Biskup [15], and the approximation by Guagliardo and Libkin [33] shows how relevant is the notion of modal logic to the problem of incomplete data. Grahne [28] described this relation by introducing modal structures in his study of incomplete information. According to Grahne, over a modal structure a first-order formula φ has two additional semantics—namely $\Diamond\varphi$, the possible answers of φ; and $\Box\varphi$, the necessary answers of φ. These formulas are the subset of the first-order modal logic formulas, that admit a unique modality in the begining of the formula. The generalized relational algebra combines operators that are somewhat modal, that its, combine several modalities in the same query. Hence, we see two interesting future works in this regard: 1) to state a correspondence between the generalized relational algebra and

first-order modal logic, and 2) to extend Datalog with modalities to express the generalized relational algebra.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 0-201-53771-0. URL: http://webdam.inria.fr/Alice/.

[2] Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. "On the Representation and Querying of Sets of Possible Worlds". In: *Theor. Comput. Sci.* 78.1 (1991), pp. 158–187. DOI: 10.1016/0304-3975(51)90007-2. URL: https://doi.org/10.1016/0304-3975(51)90007-2.

[3] Shqiponja Ahmetaj, Wolfgang Fischl, Reinhard Pichler, Mantas Simkus, and Sebastian Skritek. "Towards Reconciling SPARQL and Certain Answers". In: *World Wide Web (WWW)*. 2015, pp. 23–33. DOI: 10.1145/2736277.2741636. URL: http://doi.acm.org/10.1145/2736277.2741636.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. "G-CORE: A Core for Future Graph Query Languages". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1421–1432. DOI: 10.1145/3183713.3190654. URL: https://doi.org/10.1145/3183713.3190654.

[5] Renzo Angles and Claudio Gutierrez. "The Multiset Semantics of SPARQL Patterns". In: *International Semantic Web Conference (ISWC)*. Springer, 2016, pp. 20–36. DOI: 10.1007/978-3-319-46523-4_2. URL: http://dx.doi.org/10.1007/978-3-319-46523-4_2.

[6] Renzo Angles and Claudio Gutiérrez. "Negation in SPARQL". In: *AMW*. Vol. 1644. CEUR Workshop Proceedings. CEUR-WS.org, 2016.

[7] Renzo Angles and Claudio Gutiérrez. "Subqueries in SPARQL". In: *AMW*. Vol. 749. CEUR Workshop Proceedings. CEUR-WS.org, 2011.

[8] Renzo Angles and Claudio Gutiérrez. "The Expressive Power of SPARQL". In: *International Semantic Web Conference*. Vol. 5318. Lecture Notes in Computer Science. Springer, 2008, pp. 114–129.

[9] Renzo Angles and Claudio Gutiérrez. "The Multiset Semantics of SPARQL Patterns". In: *International Semantic Web Conference (1)*. Vol. 9981. Lecture Notes in Computer Science. 2016, pp. 20–36.

[10] Marcelo Arenas, Alexandre Bertails, Eric Prud'hommeaux, and Juan Sequeda. *A Direct Mapping of Relational Data to RDF*. W3C Recommendation. 2012.

[11] Marcelo Arenas and Jorge Pérez. "Querying semantic web data with SPARQL". In: *Principles of Database Systems (PODS)*. ACM, 2011, pp. 305–316. DOI: 10.1145/1989284.1989312. URL: http://doi.acm.org/10.1145/1989284.1989312.

[12] Marcelo Arenas and Martn Ugarte. "Designing a Query Language for RDF: Marrying Open and Closed Worlds". In: *Principles of Database Systems (PODS)*. ACM, 2016, pp. 225–236. DOI: 10.1145/2902251.2902298. URL: http://doi.acm.org/10.1145/2902251.2902298.

[13] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. "Web and Semantic Web Query Languages: A Survey". In: *Reasoning Web, First International Summer School 2005, Msida, Malta, July 25-29, 2005, Tutorial Lectures*. Ed. by Norbert Eisinger and Jan Maluszynski. Vol. 3564. Lecture Notes in Computer Science. Springer, 2005, pp. 35–133. DOI: 10.1007/11526988_3. URL: https://doi.org/10.1007/11526988%5C_3.

[14] Joachim Biskup. "A Formal Approach to Null Values in Database Relations". In: *Advances in Data Base Theory: Volume 1*. Boston, MA: Springer US, 1981, pp. 299–341. ISBN: 978-1-4615-8297-7. DOI: 10.1007/978-1-4615-8297-7_11. URL: http://dx.doi.org/10.1007/978-1-4615-8297-7_11.

[15] Joachim Biskup. "A Foundation of Codd's Relational Maybe-Operations". In: *ACM Trans. Database Syst.* 8.4 (1983), pp. 608–636. DOI: 10.1145/319996.320014. URL: http://doi.acm.org/10.1145/319996.320014.

[16] Dan Brickley and R.V. Guha. *RDF Schema 1.1*. W3C Recommendation. Feb. 2014.

[17] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema". In: *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*. Ed. by Ian Horrocks and James A. Hendler. Vol. 2342. Lecture Notes in Computer Science. Springer, 2002, pp. 54–68. DOI: 10.1007/3-540-48005-6_7. URL: https://doi.org/10.1007/3-540-48005-6%5C_7.

[18] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Surveys in computer science. Springer, 1990.

[19] Ashok K. Chandra and Philip M. Merlin. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*. Ed. by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. ACM, 1977, pp. 77–90. DOI: 10.1145/800105.803397. URL: https://doi.org/10.1145/800105.803397.

[20] E. F. Codd. "Extending the Database Relational Model to Capture More Meaning". In: *ACM Trans. Database Syst.* 4.4 (Dec. 1979), pp. 397–434. ISSN: 0362-5915. DOI: 10.1145/320107.320109. URL: http://doi.acm.org/10.1145/320107.320109.

[21] Richard Cyganiak. "A relational algebra for SPARQL". In: *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170* (2005), p. 35.

[22] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. "A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming". In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, 2005, pp. 90–96. URL: http://ijcai.org/Proceedings/05/Papers/1353.pdf.

[23] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. "Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning". In: *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings*. Ed. by York Sure and John Domingue. Vol. 4011. Lecture Notes in Computer Science.

Springer, 2006, pp. 273–287. DOI: 10.1007/11762256_22. URL: https://doi.org/10.1007/11762256%5C_22.

[24] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandecic. "Introducing Wikidata to the Linked Data Web". In: *International Semantic Web Conference (ISWC)*. 2014, pp. 50–65. DOI: 10.1007/978-3-319-11964-9_4. URL: http://dx.doi.org/10.1007/978-3-319-11964-9_4.

[25] Valeria Fionda, Giuseppe Pirrò, and Claudio Gutiérrez. "NautiLOD: A Formal Language for the Web of Data Graph". In: *TWEB* 9.1 (2015), 5:1–5:43. DOI: 10.1145/2697393. URL: https://doi.org/10.1145/2697393.

[26] Melvin Fitting and Richard L Mendelsohn. *First-order modal logic*. Springer Science & Business Media, 1998.

[27] César A. Galindo-Legaria. "Outerjoins as Disjunctions". In: *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994.* Ed. by Richard T. Snodgrass and Marianne Winslett. ACM Press, 1994, pp. 348–358. DOI: 10.1145/191839.191908. URL: https://doi.org/10.1145/191839.191908.

[28] Gösta Grahne. *The problem of incomplete information in relational databases*. Vol. 554. Springer Science & Business Media, 1991.

[29] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Approximation algorithms for querying incomplete databases". In: *Inf. Syst.* 86 (2019), pp. 28–45.

[30] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Computing Approximate Certain Answers over Incomplete Databases". In: *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management, Montevideo, Uruguay, June 5–9* (2017). (to appear).

[31] SPARQL Maintenance (EXISTS) Community Group. *SPARQL EXISTS report*. Draft. Apr. 2019. URL: https://w3c.github.io/sparql-exists/docs/sparql-exists.html.

[32] Paolo Guagliardo and Leonid Libkin. "A Formal Semantics of SQL Queries, Its Validation, and Applications". In: *PVLDB* 11.1 (2017), pp. 27–39. DOI: 10.14778/3151113.3151116. URL: http://www.vldb.org/pvldb/vol11/p27-guagliardo.pdf.

[33] Paolo Guagliardo and Leonid Libkin. "Making SQL Queries Correct on Incomplete Databases: A Feasibility Study". In: *Principles of Database Systems (PODS)*. ACM, 2016, pp. 211–223. DOI: 10.1145/2902251.2902297. URL: http://doi.acm.org/10.1145/2902251.2902297.

[34] Claudio Gutierrez, Daniel Hernandez, Aidan Hogan, and Axel Polleres. "Certain Answers for SPARQL?" In: *Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016*. Ed. by Reinhard Pichler and Altigran Soares da Silva. Vol. 1644. CEUR Workshop Proceedings. CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1644/paper13.pdf.

[35] Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Perez. "Foundations of Semantic Web databases". In: *J. Comput. Syst. Sci.* 77.3 (2011), pp. 520–541. DOI: 10.1016/j.jcss.2010.04.009. URL: http://dx.doi.org/10.1016/j.jcss.2010.04.009.

[36] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. *SPARQL 1.1 Query Language*. W3C Recommendation. Mar. 2013.

[37] Patrick Hayes. *RDF Semantics*. W3C Recommendation. Feb. 2004.

[38] Patrick Hayes and Peter F. Patel-Schneider. *RDF 1.1 Semantics*. W3C Recommendation. Feb. 2014.

[39] Daniel Hernández, Claudio Gutierrez, and Renzo Angles. "The Problem of Correlation and Substitution in SPARQL - Extended Version". In: *CoRR* abs/1801.04387 (2018). arXiv: 1801.04387. URL: http://arxiv.org/abs/1801.04387.

[40] Daniel Hernández, Claudio Gutiérrez, and Renzo Angles. "Correlation and Substitution in SPARQL". In: *CoRR* abs/1606.01441 (2016).

[41] Daniel Hernández, Claudio Gutiérrez, and Aidan Hogan. "Certain Answers in SPARQL with Blank Nodes". In: *International Semantic Web Conference*. Vol. 5318. Lecture Notes in Computer Science. Springer, 2018, pp. 114–129.

[42] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. "Everything you always wanted to know about blank nodes". In: *J. Web Sem.* 27 (2014), pp. 42–69. DOI: 10.1016/j.websem.2014.06.004. URL: http://dx.doi.org/10.1016/j.websem.2014.06.004.

[43] Tomasz Imielinski and Witold Lipski Jr. "Incomplete Information in Relational Databases". In: *J. ACM* 31.4 (1984), pp. 761–791. DOI: 10.1145/1634.1886. URL: http://doi.acm.org/10.1145/1634.1886.

[44] Mark Kaminski, Egor V. Kostylev, and Bernardo Cuenca Grau. "Query Nesting, Assignment, and Aggregation in SPARQL 1.1". In: *ACM Trans. Database Syst.* 42.3 (2017), 17:1–17:46.

[45] Nadine Lerat and Witold Lipski Jr. "Nonapplicable Nulls". In: *Theor. Comput. Sci.* 46.3 (1986), pp. 67–82. DOI: 10.1016/0304-3975(86)90022-8. URL: https://doi.org/10.1016/0304-3975(86)90022-8.

[46] Mark Levene and George Loizou. *A guided tour of relational databases and beyond*. Springer, 1999.

[47] Leonid Libkin. "Aspects of Partial Information in Databases". PhD dissertation. University of Pennsylvania, 1994.

[48] Leonid Libkin. "SQL's Three-Valued Logic and Certain Answers". In: *International Conference on Database Theory (ICDT)*. 2015, pp. 94–109. DOI: 10.4230/LIPIcs.ICDT.2015.94. URL: http://dx.doi.org/10.4230/LIPIcs.ICDT.2015.94.

[49] Leonid Libkin. "SQL's Three-Valued Logic and Certain Answers". In: *ACM Trans. Database Syst.* 41.1 (2016), 1:1–1:28. DOI: 10.1145/2877206. URL: http://doi.acm.org/10.1145/2877206.

[50] Witold Lipski Jr. "On Relational Algebra with Marked Nulls Preliminary Version". In: *Principles of Database Systems (PODS)*. Waterloo, Ontario, Canada: ACM, 1984, pp. 201–203. ISBN: 0-89791-128-8. DOI: 10.1145/588011.588040. URL: http://doi.acm.org/10.1145/588011.588040.

[51] John W. Lloyd and Rodney W. Topor. "Making Prolog more Expressive". In: *J. Log. Program.* 1.3 (1984), pp. 225–240.

[52] David Maier. "Null Values Partial Information and Database Semantics". In: *The Theory of Relational Databases* (1983), pp. 371–438.

[53] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. "On Blank Nodes". In: *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*. Ed. by Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist. Vol. 7031. Lecture Notes in Computer Science. Springer, 2011,

pp. 421–437. DOI: 10.1007/978-3-642-25073-6_27. URL: https://doi.org/10.1007/978-3-642-25073-6%5C_27.

[54] Libby Miller, Andy Seaborne, and Alberto Reggiori. "Three Implementations of SquishQL, a Simple RDF Query Language". In: *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*. Ed. by Ian Horrocks and James A. Hendler. Vol. 2342. Lecture Notes in Computer Science. Springer, 2002, pp. 423–435. DOI: 10.1007/3-540-48005-6_36. URL: https://doi.org/10.1007/3-540-48005-6%5C_36.

[55] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)*. W3C Recommendation. Dec. 2012.

[56] Ralph R. Swick Ora Lassila. *Resource Description Framework (RDF) Model and Syntax Specification*. W3C Recommendation. Feb. 1999.

[57] Peter F. Patel-Schneider and David Martin. "EXISTStential Aspects of SPARQL". In: *International Semantic Web Conference (Posters & Demos)*. Vol. 1690. CEUR Workshop Proceedings. CEUR-WS.org, 2016.

[58] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3 (2009), 16:1–16:45. DOI: 10.1145/1567274.1567278. URL: http://doi.acm.org/10.1145/1567274.1567278.

[59] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. "Semantics and complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3 (2009), 16:1–16:45.

[60] Axel Polleres. "From SPARQL to rules (and back)". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. Ed. by Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy. ACM, 2007, pp. 787–796. DOI: 10.1145/1242572.1242679. URL: https://doi.org/10.1145/1242572.1242679.

[61] Axel Polleres and Johannes Peter Wallner. "On the relation between SPARQL1.1 and Answer Set Programming". In: *Journal of Applied Non-Classical Logics* 23.1-2 (2013), pp. 159–212.

[62] Eric Prud'hommeaux and Andy Seaborne. *PARQL Query Language for RDF*. W3C Recommendation. Jan. 2008.

[63] Raymond Reiter. "A Sound and Sometimes Complete Query Evaluation Algorithm for Relational Databases with Null Values". In: *J. ACM* 33.2 (Apr. 1986), pp. 349–370. ISSN: 0004-5411. DOI: 10.1145/5383.5388. URL: http://doi.acm.org/10.1145/5383.5388.

[64] Raymond Reiter. "On Closed World Data Bases". In: *Logic and Data Bases*. Advances in Data Base Theory. New York: Plemum Press, 1977, pp. 55–76.

[65] Raymond Reiter. "Towards a Logical Reconstruction of Relational Database Theory". In: *On Conceptual Modelling (Intervale)*. 1982, pp. 191–233.

[66] Simon Schenk. "A SPARQL Semantics Based on Datalog". In: *KI*. Vol. 4667. Lecture Notes in Computer Science. Springer, 2007, pp. 160–174.

[67] Andy Seaborne. *Proposal: deep binding injection*. 2016. URL: https://lists.w3.org/Archives/Public/public-sparql-exists/2016Sep/0024.html (visited on 09/29/2016).

[68] Yannis Vassiliou. "Null Values in Data Base Management: A Denotational Semantics Approach". In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*. Ed. by Philip A.

Bernstein. ACM, 1979, pp. 162–169. DOI: 10.1145/582095.582123. URL: https://doi.org/10.1145/582095.582123.

[69] Denny Vrandecic and Markus Krötzsch. "Wikidata: a free collaborative knowledgebase". In: *Commun. ACM* 57.10 (2014), pp. 78–85. DOI: 10.1145/2629489. URL: http://doi.acm.org/10.1145/2629489.

[70] Carlo Zaniolo. "Database Relations with Null Values". In: *Proceedings of the ACM Symposium on Principles of Database Systems, March 29-31, 1982, Los Angeles, California, USA*. Ed. by Jeffrey D. Ullman and Alfred V. Aho. ACM, 1982, pp. 27–33. DOI: 10.1145/588111.588117. URL: https://doi.org/10.1145/588111.588117.

[71] Carlo Zaniolo. "Database Relations with Null Values". In: *J. Comput. Syst. Sci.* 28.1 (1984), pp. 142–166. DOI: 10.1016/0022-0000(84)90080-1. URL: http://dx.doi.org/10.1016/0022-0000(84)90080-1.

Appendix A

The information lattice of mappings with marked nulls

Lattices over incomplete data were developed, among others, by Grahne [28] for incomplete relations that are described by sets of models and by Zaniolo [71] for relations with nulls that denote no information about the attributes. In this section we will present a lattice for mappings over a common domain that admit marked nulls. To the best of our knowledge, the lattice we will present in this section has not been studied before.

In this appendix we assume familiarity with the Reiter [63] proof-theoretical semantics formalization of tables with marked nulls. His formalization is equivalent to the model-theoretical semantics formalization we used to describe tables with marked nulls in this thesis. We use his formalization in this section because it simplifies some proofs.

Definition A.1 (Constants and marked nulls)**.** *In this chapter we assume two disjoint sets* $\mathbf{C}$ *and* $\mathbf{B}$*, called the set of constants and the set of marked nulls. In terms of the RDF sets (see Definition 2.1),* $\mathbf{C} = \mathbf{I} \cup \mathbf{L}$ *and* $\mathbf{B}$ *is the set of blank nodes.*

Definition A.2 (Value replacements)**.** *Given a function* $h : \mathbf{C} \cup \mathbf{B} \to \mathbf{C} \cup \mathbf{B}$*, and a mapping* μ*, we write* $h(\mu)$ *to denote the mapping where* $\mu(?\mathrm{x})$ *is replaced by* $h(\mu(?\mathrm{x}))$ *for each attribute* $?\mathrm{x}$ *of* μ*.*

Definition A.3 (Informativeness of a mapping)**.** *Let* μ_1 *and* μ_2 *be two mappings where* $\mathrm{dom}(\mu_1) = \mathrm{dom}(\mu_2)$*. Then,* μ_1 *is said to be less or equally informative than* μ_2*, denoted* $\mu_1 \leq \mu_2$*, if and only if there exists a function* $h : \mathbf{C} \cup \mathbf{B} \to \mathbf{C} \cup \mathbf{B}$ *such that* $h(c) = c$ *for every constant* $c \in \mathbf{C}$*, and* $h(\mu_1) = \mu_2$*. Given two mappings* μ_1 *and* μ_2*, we say that they are equivalent, denoted* $\mu_1 \cong \mu_2$*, if and only if* $\mu_1 \leq \mu_2$ *and* $\mu_2 \leq \mu_1$*. We write* $[\mu]_{\cong}$ *to denote the set of mappings that are equivalent to* μ*; this set is also called the equivalence class of* μ *over* $\cong$*. We will say that* $[\mu_1]_{\cong} \leq [\mu_2]_{\cong}$ *if the elements of* $[\mu_1]_{\cong}$ *are equally or less informative than the elements of* $[\mu_2]_{\cong}$*.*

Definition A.4 (Mappings below a mapping)**.** *Given a set of attributes* $\bar{X}$*, we write* $(\mathbf{C} \cup \mathbf{B})^{\bar{X}}$ *to denote the set of mappings* μ *where* $\mathrm{dom}(\mu) = \bar{X}$ *and* $\mu(X) \in \mathbf{C} \cup \mathbf{B}$ *for each attribute* $X \in \bar{X}$*. Given a mapping* μ*, we write* $(\mathbf{C} \cup \mathbf{B})_{\mu}$ *to denote the set of mappings* $s \in (\mathbf{C} \cup \mathbf{B})^{\mathrm{dom}(\mu)}$ *where* $s \leq \mu$*.*

Observe that $((\mathbf{C} \cup \mathbf{B})^{\bar{X}}, \leq)$ is not a partial order because the antisymmetry does not hold. Indeed, let $(\perp_1)$ and $(\perp_2)$ be two different mappings. Then, $(\perp_1) \leq (\perp_2)$, $(\perp_2) \leq (\perp_1)$

and $(\bot_1) \neq (\bot_2)$. To construct a lattice we need a partial order. It is well known that a partial order can be build using classes of equivalence over elements of a preorder. Throughout this thesis, the order of elements is extended in the obvious way for the classes of equivalence, that is, given two elements a and b in a preorder $(A, \leq)$ then, for the elements $[a]_{\cong}$ and $[b]_{\cong}$ in $A/\cong$, it holds that $[a]_{\cong} \leq [b]_{\cong}$ if and only if $a \leq b$. We use this technique in the following theorem.

Theorem A.1. *Given a set of attributes $\bar{X}$, the pair $((\mathbf{C} \cup \mathbf{B})^{\bar{X}}/\cong, \leq)$ is a partial order.*

Proof. To see that $((\mathbf{C} \cup \mathbf{B})^{\bar{X}}/\cong, \leq)$ is a partial order, we have to prove the reflexibility, antisymmetry, and transitivity of $\leq$. The relation $\leq$ is reflexible because $\mathrm{Id}(\mu) = \mu$ for the the identity function Id over $\mathbf{C} \cup \mathbf{B}$. The relation $\leq$ is antisymmetric because if $[\mu_1]_{\cong} \leq [\mu_2]_{\cong}$ and $[\mu_2]_{\cong} \leq [\mu_1]_{\cong}$ then $\mu_1 \cong \mu_2$, so $[\mu_1]_{\cong} = [\mu_2]_{\cong}$. The relation $\leq$ is transitive because if $\mu_1 \leq \mu_2$ and $\mu_2 \leq \mu_3$ then there exists functions $h_1 : \mathbf{C} \cup \mathbf{B} \to \mathbf{C} \cup \mathbf{B}$ and $h_2 : \mathbf{C} \cup \mathbf{B} \to \mathbf{C} \cup \mathbf{B}$ such that $h_1(\mu_1) = \mu_2$, $h_2(\mu_2) = \mu_3$; thus $(h_2 \circ h_1)(\mu_1) = \mu_3$, so $\mu_1 \leq \mu_3$. □

Theorem A.2. *Let $\bar{X}$ be a set of attributes, and μ be a complete mapping (i.e., a mapping without null values) over the schema $\bar{X}$. Then, the pair $((\mathbf{C} \cup \mathbf{B})_{\mu}/\cong, \leq)$ is a lattice.*

Proof. To show that the partial order $((\mathbf{C} \cup \mathbf{B})_{\mu}/\cong, \leq)$ is a lattice we have to show that there exists an algebra $((\mathbf{C} \cup \mathbf{B})_{\mu}/\cong, \wedge, \vee)$ where for any $[\mu_1]_{\cong}, [\mu_2]_{\cong} \in ((\mathbf{C} \cup \mathbf{B})_{\mu}/\cong, \leq)$ the following properties hold:

1. $[\mu_1]_{\cong} \vee [\mu_2]_{\cong}$—the join—is the minimal element in $((\mathbf{C} \cup \mathbf{B})_{\mu}/\cong, \leq)$ such that $[\mu_1]_{\cong} \leq [\mu_1]_{\cong} \vee [\mu_2]_{\cong}$ and $[\mu_2]_{\cong} \leq [\mu_1]_{\cong} \vee [\mu_2]_{\cong}$; and
2. $[\mu_1]_{\cong} \wedge [\mu_2]_{\cong}$—the meet—is the maximal element in $((\mathbf{C} \cup \mathbf{B})_{\mu}/\cong, \leq)$ such that $[\mu_1]_{\cong} \wedge [\mu_2]_{\cong} \leq [\mu_1]_{\cong}$ and $[\mu_1]_{\cong} \wedge [\mu_2]_{\cong} \leq [\mu_2]_{\cong}$.

The join: Given two mappings μ_1 and μ_2 in $(\mathbf{C} \cup \mathbf{B})_{\mu}$, let $\mu_1 \vee \mu_2$ be the mapping defined by the following procedure:

1. Let μ_0 be a mapping whose domain is $\bar{X}$ and that ranges over sets of constants or marked null values, whose initial value is defined as follows:

   $$\mu_0(X) = \{\mu_1(X), \mu_2(X)\}.$$

   Intuitively, each set in the range of μ_0 represents a set of values in $\mathbf{C} \cup \mathbf{B}$ that are equal if μ_1 and μ_2 represent the same mapping.

2. Apply the following update until a fixed point is reached: If there are two attributes X and Y in $\bar{X}$ where $\mu_k(X) \cap \mu_k(Y) \neq \varnothing$ then let μ_{k+1} be the mapping that agrees with μ_k except in attributes X and Y where $\mu_{k+1}(X) = \mu_{k+1}(Y) = \mu_k(X) \cup \mu_k(X)$. Intuitively, this update applies the transitivity of the equality relation assumed in the elements of sets $\mu_k(X)$ and $\mu_k(Y)$.

3. After reaching a fixed point μ, the join between mappings μ_1 and μ_2 is determined for each attribute $X \in \bar{X}$ follows:

(a) If $\mu(X)$ is a set S having no constants then $(\mu_1 \vee \mu_2)(X)$ is a fresh marked null $\perp_S$, and consistently $(\mu_1 \vee \mu_2)(Y) = \perp_S$ for every other attribute $Y \in \bar{X}$ where $\mu(Y) = S$.

(b) If $\mu(X)$ is a set S having a constant c, then $(\mu_1 \vee \mu_2)(X) = c$.

Also, let $E_\mu(\mu_1, \mu_2)$, $E_1(\mu_1, \mu_2)$, $E_2(\mu_1, \mu_2)$, and $E_{12}(\mu_1, \mu_2)$ denote the sets of equalities defined as follows:

$$\begin{aligned} E_\mu &= \{\mu(X) = (\mu_1 \vee \mu_2)(X) \mid X \in \bar{X}\}, \\ E_1 &= \{\mu_1(X) = (\mu_1 \vee \mu_2)(X) \mid X \in \bar{X}\}, \\ E_2 &= \{\mu_2(X) = (\mu_1 \vee \mu_2)(X) \mid X \in \bar{X}\}, \\ E_{12} &= \{\mu_1(X) = \mu_2(X) \mid X \in \bar{X}\}. \end{aligned}$$

Then, given two elements $[\mu_1]_\cong$ and $[\mu_2]_\cong$ in $(\mathbf{C} \cup \mathbf{B})_\mu/\cong$, the join of them, denoted $[\mu_1]_\cong \vee [\mu_2]_\cong$, is defined as follows:

$$[\mu_1]_\cong \vee [\mu_2]_\cong = [\mu_1 \vee \mu_2]_\cong.$$

Let Σ_μ, Σ_1, Σ_2 and Σ_{12} be the respective Reither's theories of the relations containing only mapping μ, mapping μ_1, mapping μ_2 and mapping $\mu_1 \leq \mu_2$. Thus, the partial order $\leq$ corresponds to the entailment over theories.

By construction, it holds that:

$$E_1 \wedge \Sigma_{12} \models \Sigma_1, \tag{A.1}$$
$$E_2 \wedge \Sigma_{12} \models \Sigma_2, \tag{A.2}$$
$$E_\mu \wedge \Sigma_\mu \models \Sigma_{12}. \tag{A.3}$$

The statements in (A.1), (A.1), and (A.1) imply respectively that $\mu_1 \leq \mu_1 \vee \mu_2$, $\mu_1 \leq \mu_1 \vee \mu_2$ and $\mu_1 \vee \mu_2 \leq \mu$. Thus, the join is more informative than the elements joined and the join operation is closed on $(\mathbf{C} \cup \mathbf{B})_\mu$.

Since the theories Σ_1 and Σ_2 implies that there exists a unique mapping of schema $\bar{X}$, the following statement holds:

$$\Sigma_1 \wedge \Sigma_2 \models E_{12}. \tag{A.4}$$

Then, it holds that:

$$E_1 \wedge E_2 \wedge \Sigma_1 \wedge \Sigma_2 \models \Sigma_{12}. \tag{A.5}$$

To see that $\mu_1 \vee \mu_2$ is the minimum mapping such that $\mu_1 \leq \mu_1 \vee \mu_2$ and $\mu_2 \leq \mu_1 \vee \mu_2$, assume that there is another mapping $\mu_3 \in (\mathbf{C} \cup \mathbf{B})_\mu$ where $\mu_1 \leq \mu_3$ and $\mu_2 \leq \mu_3$. Then, it holds that:

$$E_{13} \wedge \Sigma_3 \models \Sigma_1, \tag{A.6}$$
$$E_{23} \wedge \Sigma_3 \models \Sigma_2, \tag{A.7}$$

where E_{13} and E_{23} are the sets of equalities defined as follows:

$$E_{13} = \{\mu_1(X) = (\mu_1 \vee \mu_2)(X) \mid X \in \bar{X}\},$$
$$E_{23} = \{\mu_2(X) = (\mu_1 \vee \mu_2)(X) \mid X \in \bar{X}\}.$$

Then, by applying the conjunction among both sides of the statements in (A.6) and (A.7), it holds that:

$$E_{13} \wedge E_{23} \wedge \Sigma_3 \models \Sigma_1 \wedge \Sigma_2. \tag{A.8}$$

By introducing the formulas $E_1 \wedge E_2$ in both sides in (A.8), it holds that:

$$E_1 \wedge E_2 \wedge E_{13} \wedge E_{23} \wedge \Sigma_3 \models E_1 \wedge E_2 \wedge \Sigma_1 \wedge \Sigma_2. \tag{A.9}$$

By the statements in (A.5) and (A.9) it holds that:

$$E_1 \wedge E_2 \wedge E_{13} \wedge E_{23} \wedge \Sigma_3 \models E_{12} \tag{A.10}$$

The statement in (A.10) implies that $\mu_1 \vee \mu_2 \leq \mu_3$. Hence, $\mu_1 \vee \mu_2$ is the minimal mapping that is more informative than μ_1 and μ_2.

The meet: Let $u : (\mathbf{C} \cup \mathbf{B})^2 \to \mathbf{C}$ be the function defined as follows:

$$u(x, y) = \begin{cases} \text{a constant } c & \text{if } x = y = c, \\ \text{a fresh null} & \text{otherwise.} \end{cases}$$

Let μ_1 and μ_2 be two mappings in $(\mathbf{C} \cup \mathbf{B})_\mu$. Then, $\mu_1 \wedge \mu_2 = u(\mu_1(X), \mu_2(X))$ for each attribute X in the schema of mappings μ_1 and μ_2, and $[\mu_1]_\cong \wedge [\mu_2]_\cong$ be $[\mu_1 \wedge \mu_2]_\cong$.

Let μ_1, μ_2, and μ_3 be three mappings in $(\mathbf{C} \cup \mathbf{B})_\mu$ where $\mu_3 \leq \mu_1$ and $\mu_3 \leq \mu_2$. Let h_1 and h_2 be the functions whose domain are the values in the mapping $\mu_1 \wedge \mu_2$ and for every attribute $X \in \bar{X}$ the following conditions hold:

$$h_1(z) = x \qquad \text{if } \mu_1(X) = x \text{ and } (\mu_1 \wedge \mu_2)(X) = z,$$
$$h_2(z) = y \qquad \text{if } \mu_2(X) = y \text{ and } (\mu_1 \wedge \mu_2)(X) = z.$$

By construction, μ_1 is the result of replacing each value z in the mapping $\mu_1 \wedge \mu_2$ by $h_1(z)$. Thus, $\mu_1 \wedge \mu_2 \leq \mu_1$. Indeed, if $\mu_1(X)$ is a constant c then $(\mu_1 \wedge \mu_2)(X)$ is the constant c or a null value. If $\mu_1(X)$ is a null value, then $(\mu_1 \wedge \mu_2)(X)$ is a null value. In both cases μ_1 is more informative than $\mu_1 \wedge \mu_2$. The same argument applies to show that $\mu_1 \wedge \mu_2 \leq \mu_2$. Furthermore, since $\mu_1 \wedge \mu_2 \leq \mu_1$, by transitivity, it holds that $\mu_1 \wedge \mu_2 \leq \mu$. Hence, $\wedge$ is closed in $(\mathbf{C} \cup \mathbf{B})_\mu$.

Let h_3 be the function whose domain are the values in mapping μ_3 and holds the following condition for each attribute $X \in \bar{X}$:

$$h_3(z) = u(x, y) \qquad \text{if} \quad \mu_3(X) = z, \mu_1(X) = x, \text{ and } \mu_2(X) = y.$$

Then, by construction, $h_3(\mu_3) = \mu_1 \wedge \mu_2$. To see that $\mu_3 \leq \mu_1 \wedge \mu_2$ we have to prove that $\mu_3(X)$ is less or equally informative than $(\mu_1 \wedge \mu_2)(X)$, for each attribute $X \in \bar{X}$. If $\mu_3(X)$

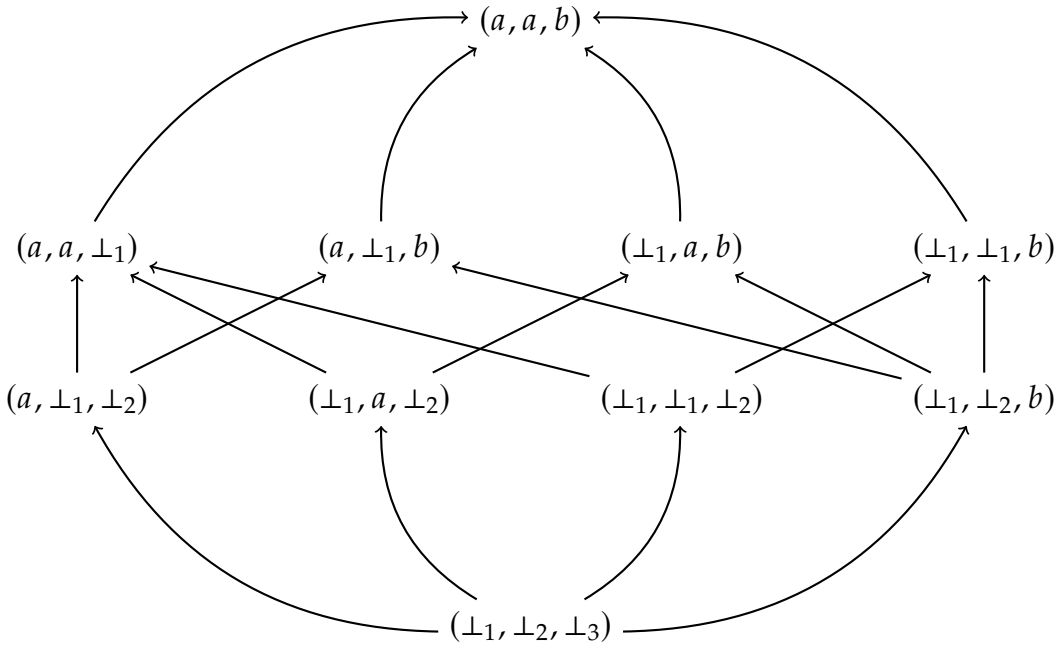


Figure A.1: Lattice of $(\mathbf{C}^{+}_{(a,a,b)}/\cong, \leq)$

is a constant c, then $\mu_1(X) = c$ and $\mu_2(X) = c$, because $\mu_3 \leq \mu_1$ and $\mu_3 \leq \mu_2$. Then, by definition, it holds that $(\mu_1 \wedge \mu_2)(X) = c$. Thus, $\mu_3(X)$ is equally informative than $(\mu_1 \wedge \mu_2)(X)$. Otherwise, if $\mu_3(X)$ is a null value, then $\mu_3(X)$ is less or equally informative than $(\mu_1 \wedge \mu_2)(X)$. Thus, $\mu_3 \leq \mu_1 \wedge \mu_2$. Hence, $\mu_1 \wedge \mu_2$ is the most informative mapping that is subsumed by μ_1 and μ_2.

By simple inspection, the operator $\wedge$ is idempotent, commutative, and associative. □

Observe that, given a mapping μ without nulls, by construction the top of the lattice $((\mathbf{C} \cup \mathbf{B})_{\mu}/\cong, \leq)$ is the class $[\mu]_{\cong}$. It is not difficult to see that the bottom is the mapping s where $s(X)$ is a different null value for each attribute $X \in \bar{X}$. Figure A.1 depicts a the lattice associated to the mapping (a, a, b).

Appendix B

SQL and relational algebra

In the previous section we showed several operators that generalize the operators of relational algebra. In this section we will show how these operators are related to SQL. That is, if SQL queries can be described in terms of these operators. Then, we will show the well-known fact that SQL does not ensure certain-answers and use the relation of SQL and the generalized algebraic operators to provide an intuition on what is happening with SQL.

Since SQL uses unmarked nulls, we have to consider the generalized algebra for unknown values. Generalized operators section were defined in the previous section for the case of databases with marked nulls. It is not difficult to adapt these results to the case of unmarked nulls. Moreover, the selection operator for unmarked null values was defined for both cases in order to provide the intuition of how the corresponding operators for unmarked nulls may be defined.

Throughout this section we present several correspondences between the generalized relational algebra and SQL. These correspondences are justified by experiments. We run queries in three open source engines, namely SQLite3, PostgreSQL and MariaDB, and then compare the results with the returned by the generalized algebra. Of course, this method does not ensure that both query languages are equivalent, but it is useful to explore correspondences when the specification of a query language is not clear or too complex for the study. According Guagliardo and Libkin [32], this is the case of SQL. To validate the formal semantics for SQL that they proposed, they generate random queries and compare the result of the queries against two engines, namely PostgreSQL and Oracle. In this section we follow the experimental strategy of Guagliardo and Libking to find an equivalence between the generalized relational algebra and SQL. However, we do not use the same randomly generalized queries that they use. Instead, we use simple generic queries manually created.

A natural extension of the work presented in this section is repeating the experimental validation by Guagliardo and Libkin [32], and proving theoretically the equivalence of the generalized relational algebra with the formal semantics they proposed. These extensions are left out the scope of this thesis since we are focused on SPARQL.

B.1 SQL and the generalized PRU algebra

The generalized operators projection, renaming and union correspond exactly to SQL. Table B.1 shows the correspondence between these generalized algebraic operators and SQL.

Algebra	SQL
$\pi_{X_1,\ldots,X_n}(R)$	`SELECT DISTINCT` $X_1,\ldots,X_n$ `FROM` R
$\rho_{X/Y}(R)$	`SELECT` X `AS` Y `FROM` R
$R \cup S$	`SELECT * FROM` R `UNION SELECT * FROM` S

Table B.1: Correspondence between PRU algebra and SQL.

To check the correspondences presented in Table B.1 we realized a simple experiment where table R is instantiated as follows:

$$R = \left[\begin{array}{cc} A & B \\ \hline 1 & 2 \\ \bot & 3 \\ 4 & \bot \\ \bot & \bot \end{array}\right], \qquad S = \left[\begin{array}{cc} A & B \\ \hline 1 & 2 \\ 3 & 4 \\ \bot & \bot \end{array}\right],$$

and the three queries in Table B.1 are instantiated as is presented in Table B.2. These three queries were checked in the three aforementioned engines, and all results agree with correspondences presented in Table B.2.

Algebra	SQL
$\pi_A(R)$	`SELECT DISTINCT` A `FROM` R
$\rho_{A/C}(R)$	`SELECT` A `AS` C `FROM` R
$R \cup S$	`SELECT * FROM` R `UNION SELECT * FROM` S

Table B.2: Queries to check the correspondence between PRU algebra and SQL.

The keyword DISTINCT is added to the SQL query in the first row of Table B.1 in order to avoid repeated elements. Recall that the generalized algebra assumes that tables are sets of tuples, while SQL tables admit the repetition of tuples. Regarding to repeated tuples and null values, it is interesting that in this case unmarked nulls is not the particular case of marked nulls when all marked nulls are different, as occur when considering the model-based database semantics of unknown values. In fact, if we replace the unmarked null by marked nulls, the result of query $\pi_A(R)$ has the tuples $\{(1), (\bot_1), (4), (\bot_2)\}$, while the result of this query with unmarked nulls is $\{(1), (\bot), (4)\}$.

Observe that the SQL UNION in the last query is the set union. In SQL the union that admit repeated elements is denoted ALL UNION. Likewise the case of the projection, in the set union unmarked nulls are not a particular case of marked nulls.

B.2 SQL and the generalized selection

Consider the following SQL query

```
SELECT A, B
FROM R
WHERE R.A = R.B
```

and let R be the following table:

$$\left[\begin{array}{cc} A & B \\ \hline a & a \\ a & b \\ a & \bot \\ \bot & b \\ \bot & \bot \end{array}\right]$$

Then the output of this SQL query is the following table:

$$\left[\begin{array}{cc} A & B \\ \hline a & a \end{array}\right]$$

If table R has not had null values, then this SQL query would have been equivalent to the relational algebra query $\sigma_{A=B}(R)$. However, in presence of nulls this query has to be evaluated with a generalized operator. The results of this query are equivalent to the generalized algebraic query $\sigma_{\text{under } A=B}(R)$. Thus, this experiment validates the correspondence presented in Table B.3. Intuitively, SQL tries to return only certain answers by passing only tuples where the condition is evaluated as true.

Algebra	SQL
$\sigma_{\text{under } \varphi}(R)$	`SELECT * FROM` R `WHERE` φ

Table B.3: Correspondence between selection formulas in the generalized algebra and in SQL.

B.3 SQL and the generalized join

Consider the following SQL query

```
SELECT A, B
FROM R NATURAL JOIN S
```

and let R and S be the following tables:

$$R = \left[\begin{array}{c} A \\ \hline a \\ b \\ \bot \end{array}\right] \qquad S = \left[\begin{array}{cc} A & B \\ \hline a & c \\ \bot & d \end{array}\right]$$

Then, the output of this query is the following table:

$$\left[\begin{array}{cc} A & B \\ \hline a & b \end{array}\right]$$

If tables R and S have not had null values, then this SQL query would have been equivalent to the relational algebra query $R \bowtie S$. However, in presence of nulls this query has to be evaluated with a generalized operator. The result of this query is equivalent to the one obtained when evaluating the query $R \bowtie_{\text{under}} S$. Thus, this experiment validates the correspondence presented in Table B.4. Intuitively, SQL assumes that cells where there is a null value are not joinable.

Algebra	SQL
$R \bowtie_{\text{under}} S$	`SELECT * FROM R NATURAL JOIN S`

Table B.4: Correspondence between the join in the generalized algebra and in SQL.

B.4 SQL and the generalized difference

Consider the following SQL query

```
SELECT A
FROM R
WHERE NOT EXISTS (SELECT S.A
                  FROM S
                  WHERE S.A = R.A)
```

where R and S are the following tables:

$$R = \left[\begin{array}{c} A \\ \hline a \\ b \\ \bot \end{array}\right] \qquad\qquad S = \left[\begin{array}{c} A \\ \hline b \\ \bot \end{array}\right]$$

Then, the output of this SQL query is the following table:

$$R = \left[\begin{array}{c} A \\ \hline a \\ \bot \end{array}\right]$$

If tables R and S have not had null values, then this SQL query would have been equivalent to the relational algebra query $R - S$. However, in presence of nulls this query has to be evaluated with a generalized operator. The result of this query is equivalent to the one obtained when evaluating the query $R -_{\text{over}} S$. Thus, this experiment validates the correspondence presented in Table B.5. Intuitively, SQL discards tuples of R that are surely discarded.

Algebra	SQL
$R -_{\text{over}} S$	SELECT * FROM R WHERE NOT EXISTS (SELECT * FROM S WHERE R.X_1 = S.X_1 AND ... AND X_n = S.X_n)

Table B.5: Correspondence between the difference in the generalized algebra and in SQL.

Another form of difference in SQL is the one provided by the operator EXCEPT. For instance, consider the following SQL query:

```
SELECT A FROM R
EXCEPT
SELECT A FROM S
```

If R and S are the same relations as the previous query, then result of this second SQL query is $\{(a)\}$. EXCEPTS corresponds to the set difference generalized with null values. This result does not correspond to the under-approximated difference nor the over-approximated difference.