Albert-Ludwigs-Universität Freiburg
Technische Fakultät, Institut für Informatik

Foundations of SPARQL Query Optimization

Dissertation zur Erlangung des Doktorgrades (Dr. rer. nat.)

von

Michael Schmidt
geboren am 11.10.1980 in Neunkirchen

Dekan	:	**Prof. Dr. Hans Zappe**
Referenten	:	**Prof. Dr. Georg Lausen** **Prof. Dr. Christoph Koch**
Datum der Promotion	:	**22.12.2009**

Freiburg im Breisgau, 7. Januar 2010 Michael Schmidt

Zusammenfassung

Die SPARQL Anfragesprache wurde vom W3C als Standardsprache zur Extraktion von Daten aus RDF Datenbanken vorgeschlagen, einem Datenformat das speziell zur maschinenlesbaren Repräsentation von Informationen im Semantischen Web entwickelt wurde. Die vorliegende Arbeit beschäftigt sich mit unterschiedlichen Fragestellungen, die im direkten Zusammenhang mit der effizienten Auswertung von SPARQL Anfragen stehen. Ein erster wichtiger Beitrag der Arbeit ist eine vollständige Komplexitätsanalyse für alle Fragmente der SPARQL Anfragesprache, welche Operatorkonstellationen aufzeigt, die für die Komplexität der Auswertung hauptverantwortlich sind. Ein zentrales Ergebnis in diesem Zusammenhang ist, dass schon der SPARQL Operator OPTIONAL allein, der zur optionalen Extraktion von Komponenten aus RDF Graphen genutzt werden kann, für die PSPACE-Vollständigkeit der Anfragesprache verantwortlich ist. Motiviert durch die Komplexitätsergebnisse wird im Folgenden die algebraische Optimierung von SPARQL Anfragen untersucht. Als zentrales Hilfsmittel in dieser Analyse entwickeln wir die Konzepte der *möglichen* und *sicheren Variablen* in SPARQL-Anfragen, welche obere und untere Grenzen für die in SPARQL Ergebnis-Mappings gebundenden Variablen festlegen und somit der Besonderheit ungebundener Variablen in SPARQL Mappings gerecht werden. Diese Konzepte ermöglichen es, Äquivalenzen über SPARQL Algebra Ausdrücken präzise und kompakt zu spezifizieren und mit ihrer Hilfe entwickeln wir eine Vielzahl von Umformungsregeln über SPARQL Algebra, einerseits mit dem Ziel etablierte Optimierungstechniken aus der Relationalen Algebra, wie z.B. Filter und Projection Pushing, in den Kontext von SPARQL zu übertragen, andererseits um SPARQL-spezifische Transformations-Schemata zu unterstützen. Unser Ansatz zur algebraischen Optimierung wird schließlich durch einen Ansatz zur semantischen Optimierung von SPARQL Anfragen ergänzt, der auf dem klassischen Chase-Algorithmus zur semantischen Optimierung von Konjunktiven Anfragen aufbaut. Unsere theoretischen Ergebnisse bezüglich algebraischer und semantischer Optimierung sind von direkter praktischer Bedeutung, da sie die Grundlage zur Entwicklung von kostenbasierten Optimierungsansätzen für SPARQL bilden. Abschließend stellen wir einen sprachspezifischen Benchmark für die SPARQL Anfragesprache, genannt *SP²Bench*, vor, der es ermöglicht die Performanz sowie Stärken und Schwächen von SPARQL Interpretern in einem umfassenden und anwendungsunabhängigen Szenario zu ermitteln.

Abstract

We study fundamental aspects related to the efficient evaluation of SPARQL, a prominent query language for the RDF data format that has been developed for the encoding of machine-readable information in the Semantic Web. Our key contributions include (i) a complete complexity analysis for all operator fragments of the SPARQL query language, which identifies operator constellations that make query evaluation hard and – as a central result – shows that the SPARQL OPTIONAL operator alone, which allows for the optional selection of components in RDF graphs, is responsible for the PSPACE-completeness of the SPARQL evaluation problem; (ii) the novel concepts of *possible* and *certain variables* in SPARQL queries, which constitute upper and lower bounds for variables that might be bound in SPARQL result mappings, account for the specifics of the SPARQL query language, and allow to state equivalences over SPARQL expressions in a precise and compact way; (iii) a comprehensive analysis of equivalences over SPARQL algebra, including both the investigation of rewriting rules like filter and projection pushing that are well-known from relational algebra optimization as well as SPARQL-specific rewriting schemes; (iv) an approach to the semantic optimization of SPARQL queries, built on top of the classical chase algorithm; (v) a language-specific benchmark suite for SPARQL, called SP^2Bench, which allows to assess the performance of SPARQL implementations in a comprehensive, application-independent setting. Although theoretical in nature, our results on algebraic and semantic query optimization for SPARQL are of immediate practical interest and facilitate the development of cost-based SPARQL optimization schemes.

Acknowledgments

First of all, I wish to thank professor *Georg Lausen* for the opportunity to write my thesis at Freiburg University. He gave me the freedom I needed and contributed to this thesis with many valuable ideas and helpful feedback. In equal parts, my thanks goes to professor *Christoph Koch*, my former supervisor at Saarland University and second referee of the thesis. He aroused my interest in databases, always had an open ear for me, and introduced me into the world of science. I also am grateful to the DFG and the Graduiertenkolleg *Mathematical Logic and Applications*, which supported me during the last years. Further, I wish to thank professor *Andreas Podelski* and professor *Bernhard Nebel* for their willingness to serve as examiners in my defense.

A special thanks goes to all my colleagues from Freiburg University and former colleagues from Saarland University, namely *Liaquat Ali*, *Lyublena Antova*, *Wei Fang*, *Christina Fries*, *Harald Hiss*, *Matthias Ihle*, *Norbert Küchlin*, *Elisabeth Lott*, *Stefanie Scherzinger*, *Kai Simon*, *Dan Olteanu*, *Florian Schmedding*, *Philip Sorst*, *Martin Weber*, and notably my two work roommates *Thomas Hornung* und *Michael Meier* for the memorable time I spent at work. Beyond many interesting discussions and fruitful cooperation, they permanently offered me a pleasant and motivating working environment.

I am also grateful to all external people who spent their time in proof-reading the thesis, namely *Christoph Pinkel*, *Stefan Schmidt*, and *Markus Thiele*. Their feedback and proposals sustainably improved the quality of the thesis.

Last but not least, I wish to thank my mother *Martina Schmidt*, my father *Gerhard Schmidt*, the rest of my family, my girl-friend *Marie Leinen*, and all my personal friends with all my heart. They accompanied me for many years and offered me mental and personal support at all times. In the end, they gave me the motivation for all the time I spent in my work and in this thesis.

Michael Schmidt

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

Jen: "*Ever heard about the Semantic Web?*"
Roy: "*Yes, but that has nothing to do with databases, has it? By the way, these Semantic Web guys have their own community, so I guess there is nothing we could do for them...*"
Moss: "*Well, Semantic Web means machine-readable information, so there is definitely some data. And I'm pretty sure these guys appreciate expertise from the database community!*"

In May 2001, Tim Berners-Lee described his vision of a "new form of Web content that is meaningful to computers" and would "unleash a revolution of new possibilities" in an article called *The Semantic Web* [BLHL01]. Central to the idea of the Semantic Web is the encoding of information in a machine-readable format, in coexistence with human-readable HTML pages which are almost impossible to "understand" and interpret by computers. The vision was that the semantic annotation of Web resources would allow computers to automatically reason about the underlying data and enable them to make reliable and logically founded conclusions. At a global scale, the Semantic Web thus can be understood as a large knowledge base that links information from different sources together, eases the access to information, and ultimately improves search and knowledge discovery in the Internet.

In their effort to push the vision of the Semantic Web, the World Wide Web Consortium (W3C) founded working groups to systematically work out and standardize its technical foundations [sww]. It soon became clear that the realization of the Semantic Web is an ambitious project that has touching points with numerous areas of Computer Science. One important subgoal, for instance, is the design of languages that offer logically founded reasoning mechanisms. This topic is closely related to description logics [BCM$^+$03], which have been studied in the field of Artificial Intelligence for many years. Another major challenge, and the one we are going to deal with in the following, is the storage and manipulation of data in the Semantic Web. Related problems have – although in the context of different data formats – been extensively studied in database research. Tackling the issue of efficient data processing in the Semantic Web, in this thesis we investigate the challenges that arise in the context of the standard Semantic Web data format RDF [rdfa] and,

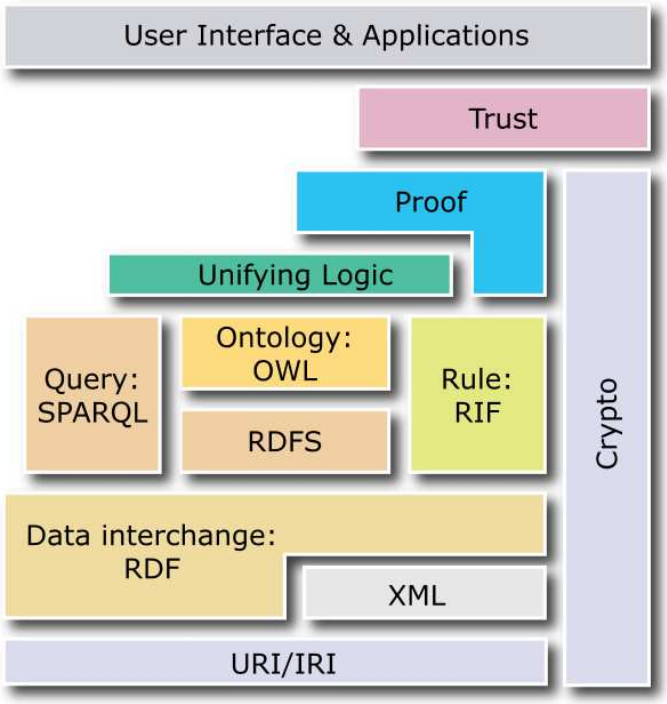

Figure 1.1.: Semantic Web layer cake (taken from [W3Ca]).

in particular, the SPARQL query language for RDF [spac], with the goal to bring together classical database research and the novel ideas behind the Semantic Web.

The diagram in Figure 1.1, known as the *Semantic Web layer cake*, summarizes the current state of the W3C efforts.[1] At the very bottom of the layer cake are the concepts of Uniform Resource Identifiers (URIs) [w3cc; w3cd] and Internationalized Resource Identifiers (IRIs) [w3cb], which provide ways to construct names for globally unique physical or logical resources. Abstracting from the details, URIs are ASCII strings consisting of a namespace and an identifier part.[2] For instance, the URI `http://my.namespace.com#Article1` is composed of namespace `http://my.namespace.com#` and identifier `Article1`. It may be used as a globally unique identifier for a specific article. For readability reasons, we shall use shortened versions of URIs, so-called prefixed URIs, e.g. writing `myns:Article1` for the above URI, where `myns` is a shortcut for the namespace `http://my.namespace.com#`.

The second layer in the Semantic Web cake, built on top of the first layer, contains the Resource Description Framework (RDF) [rdfa], which can be used for data encoding and exchange. Problems like storage, manipulation, and access to RDF

[1] There are different proposals for the architecture of the Semantic Web and different versions of the layer cake, see [GdMB08] for a discussion. We use the latest W3C proposal from [W3Ca].

[2] IRIs are generalized, internationalized versions of URIs, which differ only in technical details. In the remainder of this thesis, we will restrict our discussion to URIs.

data are closely related to the field of databases in nature and therefore will be subject to investigation in this thesis. Conceptionally, RDF databases are collections of so-called triples of knowledge, where each knowledge triple in the database is of the form (*subject*, *predicate*, *object*) and models the binary relation *predicate* between the *subject* and the *object*. While structurally homogeneous, the triple format offers great flexibility and allows to establish statements about and connections between resources (which are encoded as URIs). Consider for instance the RDF database

$$
\begin{aligned}
D := \{&(\texttt{myns:Article1}, \texttt{rdf:type}, \texttt{myns:Article}),\\
&(\texttt{myns:Article1}, \texttt{dc:title}, \textit{“The Semantic Web”}),\\
&(\texttt{myns:Article1}, \texttt{myns:journal}, \texttt{myns:Journal1}),\\
&(\texttt{myns:Article2}, \texttt{rdf:type}, \texttt{myns:Article}),\\
&(\texttt{myns:Article2}, \texttt{dc:title}, \textit{“WWW: Past, Present, and Future”}),\\
&(\texttt{myns:Article2}, \texttt{dcterms:issued}, \textit{“1996”}),\\
&(\texttt{myns:Journal1}, \texttt{rdf:type}, \texttt{myns:Journal}),\\
&(\texttt{myns:Journal1}, \texttt{dc:title}, \textit{“Scientific American”}),\\
&(\texttt{myns:Person1}, \texttt{rdf:type}, \texttt{foaf:Person}),\\
&(\texttt{myns:Person1}, \texttt{foaf:name}, \textit{“Tim Berners-Lee”}),\\
&(\texttt{myns:Article1}, \texttt{dc:creator}, \texttt{myns:Person1}),\\
&(\texttt{myns:Article2}, \texttt{dc:creator}, \texttt{myns:Person1})\},
\end{aligned}
$$

which contains twelve RDF triples in total. To give an informal description, there are two articles (typed using `myns:Article`) in the database, one titled *“The Semantic Web”*, the other one titled *“WWW: Past, Present, and Future”*. The articles are encoded as URIs `myns:Article1` and `myns:Article2`, while their titles are represented as simple string values, so-called literals. The first article appeared in a journal represented by URI `myns:Journal1` that is called *“Scientific American”*. For the second article, the journal is not specified, but instead the year *“1996”* in which the article has been issued is given. Finally, the last four triples encode that there is a person (encoded by URI `Person1`) with name *“Tim Berners-Lee”* that has written (`dc:creator`) the two articles `myns:Article1` and `myns:Article2`.

The first interesting thing to observe here is that the RDF database integrates vocabulary from different namespaces, i.e. the standard RDF namespace `rdf`, a user-defined namespace `myns`, as well as the namespaces `foaf`, `dc`, and `dcterms`. The standard namespace `rdf` provides some basic vocabulary with predefined semantics, such as `rdf:type` used for typing URIs. Next, `foaf` is a public namespace that provides domain-specific vocabulary designed to describe persons in a uniform way. Similar in idea, the namespaces `dc` and `dcterms` provide predefined vocabularies for describing bibliographic entities. The freedom to mix vocabulary from different namespaces accounts for the global nature of the RDF data format. On the one hand, this allows to connect entities from different RDF databases, on the other hand it makes it possible to develop standardized, domain-specific vocabulary collections that can be utilized by developers when creating databases for the respective domain.

A second interesting aspect of RDF that is illustrated in the above examples is the flexibility of the data format: although both `myns:Article1` and `myns:Article2` are of the same type `myns:Article`, they are structurally different: one of them comes with a title and a journal specification, while for the other one the title and the year of publication are provided. In fact, in RDF there is no a-priori restriction on the structure of entities, which makes the data format an excellent candidate for describing semi-structured and unstructured data. In this regard, RDF sharply contrasts with the relational data model, which relies on a fixed schema.

The RDF standard [rdfa] taken alone comprises only syntax specifications and a minimalistic collection of predefined vocabulary. It is extended by the RDF Schema specification (RDFS) [rdfc], settled in the third layer of the Semantic Web cake. RDFS provides additional, general-purpose vocabulary with predefined semantics. For instance, the URIs `rdfs:domain` and `rdfs:range` can be used for domain and range assignments, while `rdfs:subClassOf` and `rdfs:subPropertyOf` express subclass and subproperty relationships.[3] Making the intended meaning of this vocabulary explicit, the RDFS specification is complemented by a formal semantics [rdfc], which implements a lightweight reasoning mechanism. Assume for instance that we add triple (`myns:Article`,`rdfs:subClassOf`,`myns:Publication`) to database D from before, stating that each article also is a publication. The RDFS semantics then implies two fresh triples, i.e. (`myns:Article1`,`rdf:type`,`myns:Publication`) and (`myns:Article2`,`rdf:type`,`myns:Publication`), which are implicit in D.

There are three technologies that coexist with RDFS in the third layer: the Web Ontology Language (OWL) [owl], the Rule Interchange Format (RIF) [rifa; rifb], and the SPARQL query language for RDF [spac]. First, OWL is a data description language which, in the style of description logics [BCM+03], allows to set up user-defined vocabularies with custom semantics, making it possible to implement extended (and possibly domain-specific) reasoning mechanisms beyond the predefined semantics of the RDFS vocabulary. Second, RIF is an ongoing W3C effort to standardize a core rule language to exchange rules between different rule systems. Third, the SPARQL query language offers support to extract data from RDF(S) and OWL databases. SPARQL will be the central topic in the course of this thesis.

The basic construct of SPARQL queries are so-called triple patterns, which – when evaluating the query against some RDF database – are matched against the RDF triples in the input database. As an example, consider the following SPARQL query.

```
SELECT ?article
WHERE {
  ?article rdf:type myns:Article .
  ?article dcterms:issued ?year }
```

The query contains the two triple patterns `?article rdf:type myns:Article`

[3] `rdfs` is the default shortcut for namespace `http://www.w3.org/2000/01/rdf-schema#`.

and `?article dcterms:issued ?year`; structurally, triple patterns are RDF triples with variables in some positions (we distinguish variables by a question mark prefix). During query evaluation, these patterns are matched against the triples in the input database and variables inside the patterns are bound to the respective components of matching triples. For instance, the evaluation of the first pattern against sample database D from before has two solutions, namely (a1) the mapping that binds variable `?article` to `myns:Article1` and (a2) the mapping that binds `?article` to `myns:Article2`. Similarly, the evaluation of the second triple pattern yields a single solution mapping, in which (b) variable `?article` is bound to `myns:Article2` and `?year` is bound to the literal *"1996"*. Next, we can observe that in our sample query the two patterns are connected through operator "`.`", which can be understood as a join operation. This operator merges exactly those mappings that agree on all shared variables, so-called *compatible* mappings. It is easy to see that in our example the mappings (a1) and (b) are incompatible, because they disagree on variable `?article`, while the mappings (a2) and (b) agree on the single shared variable `?article` and therefore are compatible. Consequently, the evaluation of the join operation yields a single result mapping, obtained by merging the mappings (a2) and (b), i.e. we obtain the mapping in which `?article` is bound to `myns:Article2` and `?year` to *"1996"* (mapping (a2) is a "subset" of mapping (b), so merging these mappings gives us a mapping identical to (b)). In the final evaluation step, the SELECT clause in the query projects variable `?article`, so the evaluation result of the query on document D is the mapping that binds variable `?article` to `myns:Article2`. To give an informal description of the semantics, the query extracts all articles for which the year of publication (i.e., predicate `dcterms:issued`) is present in the database.

In addition to the triple patterns and operator "`.`" sketched in the running example, SPARQL provides the operators FILTER, OPTIONAL, and UNION, which can be used to compose more expressive queries. Before discussing SPARQL in more detail, though, we shortly sketch the remaining layers of the Semantic Web cake from Figure 1.1, to complete the big picture. The main challenge in the fourth layer is to define a unifying logic that integrates the reasoning mechanisms of underlying layers into one logical framework and allows to create proofs of deductions that can be verified by users and software agents. As a classical example [KM01], such proof mechanisms could be used to automatically derive user access rights to Web pages based on RDF data that is associated with the page. As also shown in Figure 1.1, the technologies in layers one to four are complemented by encryption technology (such as digital signatures, which allow to verify the origin of information).

The fifth layer of the Semantic Web then covers the issue of trust [Siz07; Har09], pursuing the goal to establish mechanisms that build and propagate trust at global scale. We refer the interested reader to [OH08] for a discussion of research challenges in this context. Finally, on top of the technologies and mechanisms in layers one through five, user interfaces and applications can be built that rely on the content of the Semantic Web and exploit technologies from underlying layers.

1.1. Outline and Contributions

Having introduced the global context and the basic ideas behind data management in the Semantic Web, we now summarize the work that will be presented in this thesis and highlight the major contributions. As should be clear from the previous discussion, our focus will be on the RDF data format and the SPARQL query language. We tackle these technologies from a database perspective and work out parallels between SPARQL and classical query languages like SQL, but also discuss particularities of the RDF data format and the SPARQL query language. In summary, we investigate four major aspects, all of which are closely related to the efficient processing of SPARQL queries over RDF databases:

1. The complexity of SPARQL evaluation.
2. Algebraic optimization of SPARQL queries.
3. Constraints for RDF, SPARQL as a constraint language, and semantic SPARQL query optimization in the presence of constraints for RDF.
4. Benchmarking of SPARQL engines.

Before motivating these topics, we will give some more background on the SPARQL query language. As discussed earlier, the central constructs in SPARQL are (i) triple patterns of the form (*subject*, *predicate*, *object*), which may contain variables in their positions, and (ii) the operators "." (denoted as AND in the following), FILTER, UNION, and OPTIONAL. In addition, there is a SELECT clause that allows to project a subset of the variables used in the query. The semantics of SPARQL evaluation now proceeds as follows. First, it maps all the operators occurring in the SPARQL query to algebraic operators, i.e. transforms the query into a SPARQL algebra expression. This algebra expression is then evaluated on the RDF input database.

The algebraic operators that are defined in SPARQL resemble the algebraic operators defined in relational algebra; in particular, operator AND is mapped to an algebraic join, FILTER is mapped to an algebraic selection operator, UNION is mapped to a union operator, OPTIONAL is mapped to a left outer join (which allows for the optional padding of information), and SELECT is mapped to a projection operator. As opposed to the operators in relational algebra, which are defined on top relations with fixed schema, the algebraic SPARQL operators are defined over so called mapping sets, obtained when evaluating triple patterns (cf. the running example before). In contrast to the fixed schema in relational algebra, the "schema" of mappings in SPARQL algebra is loose in the sense that such mappings may bind an arbitrary set of variables. This means that in the general case we cannot give guarantees about which variables are bound or unbound in mappings that are obtained during query evaluation. In fact, this turns out to be a significant difference which – although relational and SPARQL algebra define operators that are similar in idea – implies some fundamental conceptional differences between the two algebras. With these considerations in mind, we are now ready to discuss the contributions of the thesis.

1.1.1. Complexity of SPARQL Evaluation

The first central topic in this thesis is a thorough complexity analysis for the SPARQL query language. We argue that the study of the complexity of a language is beneficial in several respects. First, it deepens the understanding of the individual operators and their interrelations. Second, it allows to separate subsets of the language that can be evaluated efficiently from more complex (and therefore typically more expressive) subsets of the language, which may be of interest when building applications that use (fragments of) the SPARQL query language. Last but not least, our complexity analysis relates the SPARQL query language and specific fragments to traditional query languages, which allows to carry over results and optimization techniques developed in the context of the traditional language.

As usual in the study of query languages, we take the (combined) complexity of the Evaluation problem as a yardstick for our investigation: given a candidate solution mapping μ, a query Q, and a data set D as input, we are interested in the complexity of checking whether μ is contained in the result of evaluating Q on D. Previous investigations of SPARQL complexity in [PAG06a] have shown that full SPARQL is PSpace-complete, which is bad news from a complexity point of view. The study in [PAG06a], though, leaves several questions unanswered. To give only one example, it is not clear which operator constellation is ultimately responsible for the PSpace-hardness of the language. We therefore re-investigate the complexity of SPARQL and complement previous investigations by important new findings:

Contribution 1 As a key result of our complexity study, we show that the main source of complexity in SPARQL is operator Optional, which taken alone already makes the Evaluation problem for SPARQL PSpace-complete (in combined complexity). This result considerably refines previous investigations from [PAG06a], where it was shown that the combination of the three operators Optional, And, and Union makes the evaluation problem for SPARQL PSpace-complete.

In addition to the previous finding, we present complexity results for fragments of SPARQL without the complex Optional operator. Our investigation is complete in the sense that we derive tight complexity bounds for **all** possible operator combinations, showing that the complexity of SPARQL query evaluation is either PSpace-complete (i.e., for all fragments involving operator Optional), NP-complete (i.e., for Optional-free fragments where operator And co-occurs with Union or projection), or is in PTime (for the remaining fragments). Among other things, this shows that adding (removing) the Filter operator to (from) a fragment of the SPARQL query language in no case affects the complexity of query evaluation. □

1.1.2. Algebraic Optimization of SPARQL Queries

Having established the theoretical background, we turn towards an investigation of algebraic SPARQL query optimization. Motivated by the success of algebraic

rewriting rules in the context of relational algebra (see e.g. [Hal75; SC75; Tod75]), we investigate algebraic rewriting rules for SPARQL algebra. The goal of this study is twofold. Firstly, we are interested in whether (and how) established optimization techniques proposed for relational algebra, such as filter and projection pushing, carry over to SPARQL algebra. Secondly, going beyond the adaption of existing techniques, we study SPARQL-specific optimization schemes. It turns out that, given the discrepancies between relational algebra and SPARQL algebra discussed earlier in this section, the adaption of rewriting techniques from relational algebra is a non-trivial task and requires new concepts and techniques to properly cope with the loose schema in result mappings. Our major results are the following.

Contribution 2 As a general approach to deal with the loose structure of result mappings, we develop the concepts of *possible* and *certain* variables. Possible and certain variables are derived from the input query at compile time and constitute upper and lower bounds for variables that appear in result mappings, independently from the input document. In this line, they give guarantees about variables that are definitely bound and variables that are definitely not bound in result mappings, thus imposing structural constraints on mapping sets. Ultimately, these concepts allow us to state equivalences over SPARQL algebra in a clean and compact way.

Building upon the notions of possible and certain variables, we then summarize existing and develop new equivalences over SPARQL algebra. In total, we present about forty equivalences, of which almost three-fourths are new. When interpreted as rewriting rules, these equivalences allow to implement valuable rewriting techniques that are known from the relational context in SPARQL algebra, in particular filter and projection pushing. In addition, we study general-purpose rewriting rules and SPARQL-specific optimization schemes, like the rewriting of queries involving negation which – in SPARQL algebra expressions – is typically encountered in form of a characteristic constellation of the left outer join and the selection operator.

We finally investigate the differences between bag and set semantics in the context of SPARQL algebra equivalences. While previous theoretical investigations of SPARQL typically build on the set-based SPARQL semantics proposed in [PAG06a], the official W3C SPARQL specification [spac] recommends a bag semantics. We show that almost all rewriting rules that we propose in our study hold under both set and bag semantics and therefore are of immediate practical relevance. □

1.1.3. Constraints for RDF and Semantic SPARQL Optimization

Integrity constraints such as keys, foreign keys, or join dependencies, have been an important topic in classical database research from the beginning [Cod70; Cod71; Cod74; HM75; Che76; Cod79]. They impose structural restrictions that must hold on every database instance, thus restricting the state space of the database. It is well-known that such constraints are vital to schema design, normalization, and lastly

help to avoid insert, update, and deletion anomalies in relational databases [Cod71; Cod74; Che76; AHV]. Beyond that, they are valuable input to query optimizers and may help to find more efficient evaluation plans [Kin81; BV84; CGM90]. In response to their central role in relational databases, we investigate constraints for RDF and study the role of the SPARQL query language in such a setting. Our major findings and results in this context can be summarized as follows.

Contribution 3 We propose a collection of integrity constraints for the RDF data format, including constraint types like functional and inclusion dependencies, or cardinality constraints, which are well-known from relational databases and XML. In addition, we discuss SPARQL-specific constraints that capture the semantics of the RDFS vocabulary. Our formalization in first-order logic shows that many natural RDF(S) constraints fall into constraint classes that have previously been studied in the context of relational databases and XML [Fag77; Fag82; BV84; DT01; DT05].

Furthermore, we investigate the capabilities of SPARQL as a constraint language. As a major result, we show that SPARQL – with only minor extensions – is an excellent candidate for dealing with constraints, i.e. can be used to check if RDF constraints that are specified in first-order logic hold over some input database. This result motivates extensions of SPARQL by constructs to specify user-defined constraints, e.g. in the style of Create Assertion statements in SQL.

From a practical point of view, we complement the study of algebraic query optimization with an approach to semantic SPARQL query optimization. The idea behind semantic query optimization is, given a query and a set of integrity constraints as input, to find more efficient queries that are equivalent on every database instance that satisfies the integrity constraints. We present a system that allows to find minimal equivalent rewritings of SPARQL And-only queries (or, alternatively, can be applied to optimize And-only blocks in more complex queries) and propose a rule-based approach for optimizing more complex SPARQL queries. □

1.1.4. Benchmarking of SPARQL Engines

As a final contribution, we propose a benchmark for the SPARQL query language. The central motivation for this work was the observation that in the recent years a variety of optimization approaches for SPARQL have been made, but a comprehensive, SPARQL-specific benchmark platform was missing, which made it almost impossible to compare the performance of SPARQL optimization schemes and implementations. Addressing this issue, our fourth major contribution is the following.

Contribution 4 We present *SP²Bench*, a publicly available benchmark suite designed to test the performance of SPARQL engines. Unlike other benchmarks that have been proposed in the context of the Semantic Web (such as [GPH05; AMMH; BS09]), *SP²Bench* is not motivated by a single use-case, but instead is highly

SPARQL-specific and covers a variety of challenges that engines may face when processing RDF(S) data with SPARQL. This way, it allows to assess the performance of implementations in a comprehensive and application-independent setting.

SP²Bench comes with a data generator that allows to create arbitrarily large bibliographic databases in RDF format. The design of the data generator relies on an in-depth study of the well-known DBLP database [Ley] and, in response, the generated data mirrors vital social-world relationships encountered in real-world DBLP data. The data generator is complemented by a set of 17 benchmark queries, specifically designed to test characteristic SPARQL operator constellations and RDF access patterns over the generated documents. We discuss design decisions, data generator implementation, and the specific challenges of the benchmark queries. □

1.2. Structure of the Thesis

We start with globally relevant preliminaries in Chapter 2, including mathematical conventions and a formalization of RDF(S) and SPARQL. Next, Chapters 3 to 6 contain the main content and all the contributions of this thesis. The ordering of these chapters sequentially follows the list of contributions discussed in Section 1.1: Chapter 3 contains the complexity analysis for the SPARQL query language, Chapter 4 presents our results on algebraic SPARQL optimization, Chapter 5 treats the subject of constraints for RDF and semantic SPARQL query optimization, and Chapter 6 covers the SP²Bench SPARQL performance benchmark. We wrap up with a short discussion of future work and some concluding remarks in Chapter 7.

How to Read This Thesis. The preliminaries in Chapter 2 establish technical background and mathematical notation that are relevant for all subsequent chapters of the thesis. In addition to this global preliminaries section, the individual chapters may contain their own, local preliminaries, where we introduce notation and concepts that are relevant for the respective chapter only. With some minor exceptions, the chapters are self-contained and do not significantly depend on each other. Whenever we use notation or concepts in some chapter that were not introduced in the global or its local preliminaries section, we will explicitly point to the position where they were defined. Therefore, the reader who is familiar with the global preliminaries may study the chapters in the thesis in random order. We recommend, however, to read the chapters sequently, to follow the logical ordering of the topics. In particular, the complexity results right at the beginning in Chapter 3 establish a deep understanding for the expressiveness of the operators and facilitate the understanding of the algebraic and semantic rewriting approaches in Chapters 4 and 5. Furthermore, the semantic optimization approach for SPARQL presented in Chapter 5 builds upon ideas and proofs discussed before in Chapter 4. Finally, the SPARQL benchmark in Chapter 6 takes common optimization strategies for SPARQL into account and therefore picks up several ideas that were discussed in the two preceding chapters.

Structure of Chapters. Each chapter starts with an introduction that motivates its specific content, makes the reader familiar with the topic, and surveys its contributions in more detail. The introduction is followed by an optional, chapter-specific preliminaries section, the content (which typically splits up into several sections), and a chapter-specific discussion of related work as well as a short conclusion.

1.3. Summary of Publications

Parts of the work that is discussed in the course of this thesis have been published at major database venues and as technical reports. In the following we summarize publications that are related to this thesis and sketch their relations to this work.

Most of the author's publications have been done in the area of SPARQL processing and benchmarking [LMS08; SHK+08; SHLP08; SML08; SHLP09; SHM+09]. In [LMS08] we discussed the prospects of integrity constraints for RDFS. We proposed a set of constraints for RDF(S) and showed that these constraints can be both modeled with user-defined RDF(S) vocabulary and checked with the SPARQL query language. Moreover, we motivated constraint-based optimization of SPARQL by example. Revisiting the latter idea, in the technical report [SML08] we worked out a schematic approach to constraint-based SPARQL query optimization. Many of the above-mentioned ideas and results have found entrance in Chapter 5, where we study constraints for RDF(S) and semantic optimization of SPARQL.

In addition to the semantic query optimization approach, in [SML08], we presented a complexity analysis for SPARQL and discussed algebraic SPARQL query optimization. An extended version of the complexity analysis can be found in Chapter 3. Similarly, the study of algebraic query optimization in Chapter 4 extends the results on algebraic SPARQL optimization that were presented in [SML08].

The discussion of the SP^{2}Bench SPARQL benchmark in Chapter 6 relates to our work presented in [SHK+08; SHLP09; SHM+09] (and the technical report [SHLP08]). The latter chapter essentially follows the benchmark descriptions from [SHLP09; SHM+09]. In addition to the material presented in this thesis, the interested reader may find preliminary SP^{2}Bench benchmark results for selected SPARQL engines and state-of-the-art SPARQL evaluation approaches in [SHK+08; SHLP09].

Finally, we published work concerning the classical chase algorithm (see [MSL09d; MSL09a] and the associated technical reports [MSL09c; MSL09b]).[4] Tackling the issue of the possible non-termination of the chase, the latter publications present novel sufficient termination for the chase algorithm, which strictly generalize previously known termination conditions. Although we will not discuss these termination conditions in the course of this thesis, we want to point out that they are beneficial in the context of our semantic query optimization approach for SPARQL, as they

[4]This work will be published in the thesis of Michael Meier and therefore is not discussed here.

extend the applicability of our semantic optimization scheme (which builds upon the chase procedure). We will resume this discussion later in Chapter 5.

Chapter 2.

Preliminaries

Jen: "*Sounds like the Semantic Web is huge...*"
Roy: "*Yes, and the W3C specs include lots of boring details...*"
Moss: "*You're both right. I agree that RDF is a nice data format, but do we really need a query language for it? Why don't we just use SQL queries to extract data from RDF graphs?*"
Roy: "*That's a reasonable question. So let's first motivate SPARQL and then define a compact fragment of the language!*"

As shortly discussed in the Introduction, RDF databases are collections of triples of the form (*subject*,*predicate*,*object*). Due to their homogeneous structure, they are often represented as labeled directed graphs, where each triple defines an edge from the *subject* node to the *object* node under label *predicate*. Figure 2.2 exemplarily plots the graph representation of the RDF database from the Introduction, where we indicate URI-type subject and object nodes by ellipses and distinguish string literals by quotation marks. As we will see later, predicate positions of triples (and hence the edges in the graph) always carry URIs. To give an example, the edge from URI `myns:Person1` to literal *"Tim Berners-Lee"* is labeled with URI `foaf:name`, so it represents the triple (`myns:Person1`,`foaf:name`,*"Tim Berners-Lee"*). Given this close connection between RDF databases and labeled directed graphs, in the following we shall use the terms *RDF database* and *RDF graph* interchangeably.

RDF as a graph-structured data model differs conceptionally from other data models, such as the relational model [Cod70] or tree-structured XML data [w3ce]. As an immediate consequence, traditional database query languages like SQL [CB74] (for the relational model), or XQuery [xqu] and XPath [xpa] (for XML) are not directly applicable in the context of RDF data. One straightforward approach that allows to fall back on, for instance, the SQL query language for RDF data extraction is the mapping of RDF databases into the relational model. An obvious implementation of this approach would be the creation of a single relational table `Triples`(`subject`,`predicate`,`object`) [MACP02; AMMH07] that stores all RDF triples of the input RDF database. Following this scheme, the RDF graph from Figure 2.2 could be mapped to the relational instance shown in Figure 2.1.

Triples	subject	predicate	object
	myns:Article1	rdf:type	myns:Article
	myns:Article1	dc:title	*"The Semantic Web"*
	myns:Article1	myns:journal	myns:Journal1
	myns:Article2	rdf:type	myns:Article
	myns:Article2	dc:title	*"WWW: Past, Present, and Future"*
	myns:Article2	dcterms:issued	*"1996"*
	myns:Journal1	rdf:type	myns:Journal
	myns:Journal1	dc:title	*"Scientific American"*
	myns:Person1	rdf:type	foaf:Person
	myns:Person1	foaf:name	*"Tim Berners-Lee"*
	myns:Article1	dc:creator	myns:Person1
	myns:Article2	dc:creator	myns:Person1

Figure 2.1.: Relational instance storing the RDF database from the Introduction.

This mapping now facilitates data extraction and manipulation using SQL. Assume for instance that we want to extract all articles written by *"Tim Berners-Lee"*, including the respective years of publication (and only articles for which the year of publication is given). This request can be expressed by the following SQL query.

```
SELECT T1.subject AS article,
       T2.object  AS year
  FROM Triples T1, Triples T2, Triples T3, Triples T4
 WHERE T1.predicate='rdf:type' AND T1.object='myns:Article'
   AND T2.predicate='dcterms:issued'
   AND T3.predicate='dc:creator'
   AND T4.predicate='foaf:name' AND T4.object='"Tim Berners-Lee"'
   AND T1.subject=T2.subject AND T2.subject=T3.subject
   AND T3.object=T4.subject
```

We observe that the query accesses table `Triples` four times, through `T1`-`T4`. According to the restrictions in the WHERE-clause, (i) `T1` matches all RDF triples of type `myns:Article`, (ii) `T2` matches triples with predicate `dcterms:issued`, (iii) `T3` matches all URIs that have been created (`dc:creator`) by someone, and (iv) `T4` matches URIs with name (`foaf:name`) *"Tim Berners-Lee"*. `T1`-`T3` are then joined on the `subject` column, enforcing that the article URIs of the respective tuples are the same; in addition, there is a join `T3.object=T4.subject`, which asserts that the URI of the person in the object position of `T3` coincides with the URI of *"Tim-Berners Lee"* from `T4`. When evaluated against the `Triples` table from Figure 2.1, the query gives the desired result, namely one tuple (`myns:Article2`, *"1996"*).

Concerning the SQL version of the query, one might argue that the query is somewhat "unintuitive" to read. Moreover, we observe that already this simple request

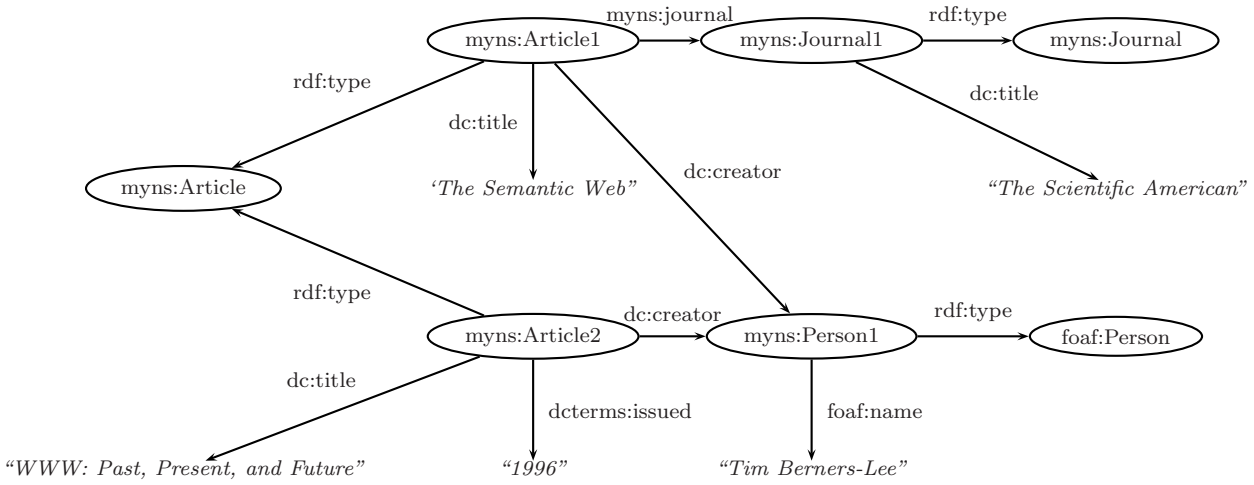

Figure 2.2.: RDF graph representation of the database from Figure 2.1.

involves three self-joins over the `Triples` table. As discussed in [AMMH07; SHK+08], such self-joins occur frequently when specifying SQL queries over relational triple tables for RDF data and it has been observed that they impose severe challenges to the query evaluator [AMMH07; SHK+08; SGK+08], in particular for large RDF databases, which often contain millions or even billions of RDF triples.[1] Similar problems have been identified for more sophisticated relational RDF storage schemes, such as Vertical Partitioning proposed in [TCK05; AMMH07] (cf. the experimental study of RDF data management approaches in [SGK+08; SHK+08]).

Generalizing from the previous example, the main problem of using traditional query languages for RDF data processing is that they do not fit the graph nature of the RDF data model, which justifies the need for a specific RDF query language. In particular, it would be desirable to have a query language that allows to specify graph patterns that in some sense mirror the structure of the RDF input graph.

Addressing this demand, over the last years different graph query languages (e.g, [SOO99; HS08]) and specific RDF query languages, such as RQL [KACP02], SeRQL [ope], and TRIPLE [SD01] have been proposed (see [HBEV04] for a comparison of different RDF query languages). In its effort to standardize the most common and useful constructs of these graph languages, the W3C worked out the SPARQL Protocol and RDF Query Language [spac], called SPARQL for short.

In response to the desiderata discussed above, SPARQL comes with a powerful graph matching facility. The request from before, for instance, can be naturally expressed by the graph pattern in Figure 2.3, where variables are prefixed by

[1] To give a concrete example, the RDF version of the U.S. Census database [Tau] contains more than one billion RDF triples. Other large-scale RDF repositories are listed at [lin].

question marks. The idea is that, during query evaluation, this graph pattern is embedded into the RDF input graph, thereby binding variables in the graph pattern to matching components of the input graph. Coming back to our example, the pattern in Figure 2.3 matches all entities of type `myns:Article` written by *"Tim Berners-Lee"* and including the year of publications. When evaluated against the RDF graph in Figure 2.2, the only possible embedding is to bind variable `?article` to `myns:Article2`, `?person` to `myns:Person1`, and `?year` to literal *"1996"*. Hence, the evaluation of this graph pattern yields the same result as the previous SQL query over the `Triples` table, extended by a binding for variable `?person`.

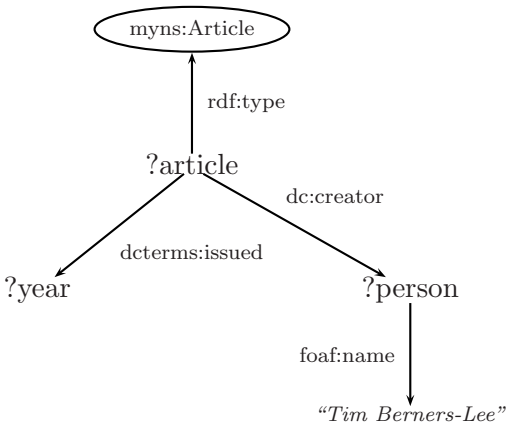

Figure 2.3.: RDF graph pattern.

The SPARQL query language now provides a syntax to express such graph patterns in textual form and to project the desired variables. The pattern in Figure 2.3 for instance, can be expressed by a conjunction of four so-called basic graph patterns, one for each edge in the pattern. The resulting SPARQL query looks as follows.

```
SELECT ?article ?year
WHERE {
  ?article rdf:type myns:Article .
  ?article dcterms:issued ?year .
  ?article dc:creator ?person .
  ?person  foaf:name "Tim Berners-Lee" }
```

The four basic SPARQL graph patterns `?article rdf:type myns:Article`, ..., `?person foaf:name "Tim Berners-Lee"` are connected through the (standard) connection method ".", together forming the graph pattern depicted in Figure 2.3. Note that variables shared between patterns collapse to a single node, as it is the case for variable `?article` (used in the first three basic graph patterns) and variable `?person` (used in the last two patterns). Hence, from an algebraic point of view the connection method "." can be understood as a join operation, which enforces that shared variables are bound to the same node. Finally, the SELECT expression projects the desired variables `?article` and `?year`. Arguably, the SPARQL query above is notationally simpler and easier to understand than its SQL counterpart.

In addition to the features illustrated in our example, the SPARQL query language provides a set of operators, namely FILTER, UNION, and OPTIONAL, which can be used to compose more expressive queries. To give a short description, the FILTER operator allows to enforce additional conditions that must hold for results (in the style of the selection operator in relational algebra), such as the equality of two

variables. Next, operator UNION computes the set-theoretical union of the evaluation result of two graph patterns. Finally, OPTIONAL allows for the optional selection of components in graph patterns. Assume for example that we specify the triple pattern `?article dcterms:issued ?year` in the query above as optional. Then the resulting query would also extract the entity `myns:Article1`, for which the year specification is missing in the database (and variable `?year` would remain unbound).

Having motivated the need for a specific RDF query language, we will now formally introduce the RDF data format and the SPARQL query language. We focus on the formalization of RDF and SPARQL core fragments, leaving out technical and implementation-specific details, such as data types in RDF and the associated type system for SPARQL. Still, our formalization covers the most important features of the official W3C RDF(S) [rdfa; rdfc; rdfd] and SPARQL [spac] specifications and provides a clean semantics for SPARQL query evaluation. It forms the basis for the subsequent investigation of SPARQL complexity and optimization in Chapters 3-5.

Structure. We survey mathematical and notational conventions in Section 2.1. In the following, Section 2.2 introduces the RDF data format, including a definition of a formal model for RDF, as well as a discussion of the predefined RDF(S) vocabulary and RDF(S) reasoning, which will play an important role in our discussion of semantic query optimization for SPARQL in Chapter 5. Next, in Section 2.3 we establish a formal model for the SPARQL query language, comprising the most important operators and concepts of the language. We provide two (slightly different) semantics for our SPARQL fragment, namely a set-based semantics and a bag semantics (also called multi-set semantics), and relate them to the official W3C SPARQL Recommendation [spac]. The preliminaries chapter concludes with a discussion of related work concerning RDF and SPARQL formalizations in Section 2.4.

2.1. Notational and Mathematical Conventions

Environments and Fonts. We use standard environments for definitions, examples, equations, propositions, lemmas, theorems, corollaries, figures, and tables. For the ease of access, all environments are numbered relative to the chapter in which they have been defined, according to the numbering scheme <chapter_id>.<number>, where <number> is a separate, increasing ID for each environment type.

When introducing important terms and definitions, we always use *italic font*. In contrast, **bold font** is used for emphasizing important terms in textual descriptions or, alternatively, to indicate the beginning of a new logical paragraph (see for instance the paragraph **"Mathematical Conventions."** on the next page).

SPARQL and SQL operators or keywords appearing in free text are distinguished by capitalized font (e.g., AND, SELECT); full SQL or SPARQL queries are enclosed in separate boxes and can be identified by typewriter font on gray-shadowed background (see e.g. the example query at the beginning of this chapter).

Mathematical Conventions. We assume that the set of natural numbers $\mathbb{N}$ does not include element 0 and put $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. We use the notation $i \in [n]$ as a shortcut for $i \in \{1, \ldots, n\}$. As usual, $|S|$ stands for the cardinality of set S.

Occasionally, we shall write *iff* for *if and only if* (or, alternatively, *exactly if*). Further, we use the symbol ":=" for defining and introducing assignments, the equality symbol "=" for deductive computation, and "≡" to express logical equivalence.

2.2. The RDF Data Format

The Resource Description Framework has been developed as part of the W3C Semantic Web initiative. As stated in the W3C Primer [rdfb], it is "particularly intended for representing metadata about Web resources", but also to "represent information about things that can be identified on the Web, even when they cannot be directly retrieved from the Web". With these goals in mind, RDF provides mechanisms to describe arbitrary physical or logical resources in a machine-readable way.

In the following, we give a compact formalization of RDF, closely following the notation introduced in [PAG06a]. We start with the definition of three disjoint infinite sets, namely the set of *URIs* U, the set of *blank nodes* B, and the set of *literals* L. They form the basis for describing data using RDF and can be understood as follows.

- The set U of URIs [w3cc; w3cd] comprises an infinite number of strings, where each URI identifies a logical or physical resource. Technically, URIs are ASCII strings that consist of a namespace and an identifier part. For instance, the URI `http://my.namespace.com#Article1` is composed of namespace `http://my.namespace.com#` and identifier `Article1`. It may represent an article that has been published in some conference. For readability, we typically denote URIs in prefix notation, e.g. writing `myns:Article1` for the above URI where `myns` is a shortcut for the namespace `http://my.namespace.com`. A list of the namespace prefixes used in this thesis can be found in Appendix A.
- Elements from B, so-called blank nodes, can be seen as distinct anonymous resources, i.e. physical or logical entities that are not assigned a fixed URI. Blank nodes may be used to identify entities for which the URI is unknown, or in cases where the entity is not prominent enough to be assigned a fixed URI. For instance, blank nodes are typically used as parent nodes to a grouping of data. With this scope, they have an existential character, stating that a certain entity or grouping exists without explicitly fixing the URI. By convention, we distinguish blank nodes from URIs by the namespace prefix "`_`", e.g. we use strings like `_:b1` or `_:grouping1` to denote blank node identifiers.
- The set L constitutes literals, such as strings, integers, or boolean values. As we shall see later, literals are used to describe properties of URIs and blank nodes. The RDF specification [rdfa] distinguishes between plain literals,

such as *"23"*, and typed literals such as *"23"^^xsd:integer*.[2] We will mainly ignore datatypes, as they raise only implementation-specific issues. Literals are distinguished from URIs and blank nodes by quotation marks and italic font.

We abbreviate the union of sets B, L, and U by concatenating their identifiers, e.g. writing BLU as a shortcut for $B \cup L \cup U$. Furthermore, when talking about *resources* in the context of RDF, we always refer to elements of the set BU.

Formalization. The central idea of the RDF data format is to describe resources through triples (s,p,o), where each triple provides the value o of a specific property p for resource s. This value could be either a literal, e.g. when providing the name or age of a person, or a resources again, which allows to establish connections between entities, such as the friendship relation between persons. Following this idea, we define RDF triples and RDF databases over the sets U, B, and L as follows.

Definition 2.1 (RDF Triple) An *RDF triple* is a triple of the form

$$(s, p, o) \in BU \times U \times BLU$$

We call the first component s of an RDF triple its *subject*, the second component p its *predicate*, and the third component o its *object*. We shall refer to the URI in predicate position as *property*. □

Definition 2.2 (RDF Database) An *RDF database*, also called *RDF graph* or *RDF document*, is a finite set of RDF triples. We write $dom(D)$ to denote the subset of elements from BLU that appear in D. □

It is worth mentioning that, from a logical point of view, RDF databases can be understood as sets of binary relations, where each distinct property p in the database (i.e., each URI occurring in predicate position) implicitly defines a relation that contains the (*subject*,*object*)-pairs that stand in relation p. We illustrate our formalization of RDF and the latter observation by means of a small example.

Example 2.1 The sample graph in Figure 2.2 represents the RDF database D defined in the Introduction. In this graph representation, each edge $s \xrightarrow{p} o$ stands for a triple (s, p, o). As a triple with a literal in object position consider for instance (`myns:Article2`,`dcterms:issued`,*"1996"*). Triples with resources in object position, such as (`myns:Article2`,`dc:creator`,`myns:Person1`), encode relationships between resources. The graph does not contain blank nodes. When switching to the binary relation view of the RDF database, we identify the six relations `dc:title`, `myns:journal`, `rdf:type`, `foaf:name`, `dcterms:issued`, and `dc:creator`. Relation `rdf:type` for instance stores all type relations that hold between RDF resources, namely (`myns:Article1`,`myns:Article`), (`myns:Article2`,`myns:Article`), (`myns:Person1`,`foaf:Person`), and (`myns:Journal1`,`myns:Journal`). □

[2] The `xsd` namespace `http://www.w3.org/2001/XMLSchema#` provides a set of predefined base types like `xsd:integer`, `xsd:string`, or `xsd:boolean`.

2.2.1. Predefined RDF and RDFS Vocabulary

The W3C specifications [rdfa; rdfe; rdfc] introduce two standard namespaces: the RDF namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#` (prefix `rdf`) and the RDF Schema namespace `http://www.w3.org/2000/01/rdf-schema#` (prefix `rdfs`). These namespaces comprise a set of URIs with predefined meaning, called *standard vocabulary* in the following. The standard vocabulary supports basic RDF modeling tasks, such as typing, expressing subclass relationships, or creating bags or sequences of resources. From a logical point of view, the vocabulary splits up into so-called *classes* (i.e., URIs that are used for typing resources) and *properties* (i.e., URIs that are used as properties). Complementarily to the concept of classes, we will use the term *object* to denote URIs that are instances of some class.[3]

We summarize the most important standard vocabulary in the following listing.

- The predefined URI `rdf:type` can be used for typing entities. We illustrated the use of this property before in our running example (cf. Figure 2.2).
- The two classes `rdfs:Class` and `rdf:Property` can be used to assign a logical type to URIs. For instance, the triple (`myns:Article`,`rdf:type`,`rdfs:Class`) states that URI `myns:Article` will be utilized as a class. Similarly, the RDF triple (`dc:creator`,`rdf:type`,`rdf:Property`) indicates that URI `dc:creator` is used as a property. By convention, we distinguish classes from properties by capitalizing the first letter of the identifier part (cf. `Article` vs. `creator`).
- The URIs `rdfs:domain` and `rdfs:range` can be used to specify the domain and range of properties. As an extension of our running example, we might introduce the two RDF triples (`dc:creator`,`rdfs:domain`,`myns:Article`) and (`dc:creator`,`rdfs:range`,`foaf:Person`), to indicate that `dc:creator` associates `myns:Article`-typed objects with `foaf:Person`-typed objects.
- `rdfs:subClassOf` and `rdfs:subPropertyOf` are used to describe subclass and subproperty relationships between classes and properties, respectively. The triple (`myns:Article`,`rdfs:subClassOf`,`myns:Publication`) for instance encodes that `myns:Article` is a subclass of `myns:Publication`. Similarly, we could define the triple (`dc:creator`,`rdfs:subPropertyOf`,`myns:relatedTo`), stating that property `dc:creator` specializes property `myns:relatedTo`.
- The `rdf` namespace provides vocabulary for creating so-called *containers*. There exist three predefined container classes in the `rdf` namespace, namely `rdf:Bag` for modeling sets, `rdf:Seq` for modeling ordered lists or sequences, and `rdf:Alt` for modeling alternatives. In addition, there is an infinite set of so-called *container membership properties* `rdf:_1`, `rdf:_2`, `rdf:_3`, ..., used to enumerate the members of a container object. Figure 2.4(a) exemplarily

[3] Although we use the terms *class*, *object*, and *property* to facilitate the subsequent discussion, strictly speaking there is no separation between them in RDF. For instance, a URI might represent both a class and a property at the same time.

shows an RDF sequence containing the first three prime numbers. The sequence itself is modeled as a blank node `_:primes` of type `rdf:Seq`. The three prime numbers "2", "3", and "5" are associated using the first three container membership properties, respectively. Note that RDF containers are "open", i.e. there is no vocabulary to explicitly fix the number of container objects.

- Complementarily to the container classes, RDF provides so-called *collections*, which differ from containers in that they are closed. Collections are represented as lists of type `rdf:List`, where properties `rdf:first` and `rdf:rest` define the head and tail of the list and URI `rdf:nil` can be used to close the list. Figure 2.4(b) depicts the RDF collection version of the prime number scenario from Figure 2.4(a). The blank node `_:primes1to3` represents the full collection and is typed accordingly with `rdf:List`. We observe that the first collection member, prime number *"2"*, is associated using property `rdf:first`. The remaining members are stored in the separate `rdf:List` object `_:primes2to3`, which can be understood as the tail of the list and is connected through property `rdf:rest`. This second list refers to the second member, prime number *"3"*, and links to a third sublist, identified through `_:primes3to3`. The third list then links to element *"5"* and is closed using the predefined URI `rdf:nil`.
- Further, the RDFS standard comprises a set of properties with predefined meaning. In particular, (i) `rdfs:seeAlso` allows to link to background information of a certain resource, (ii) `rdfs:isDefinedBy` is a subproperty of `rdfs:seeAlso` and allows to refer to an external definition[4] of a resource, (iii) `rdfs:label` is used to assign a (typically human-readable) label to a resource, and (iv) `rdfs:comment` is a reserved keyword for adding comments.
- Finally, the RDF standard comprises so-called reification vocabulary, which allows to make statements about triples. To give an example, using the reification vocabulary one could provide background information such as the person who created a triple or the date when a triple was inserted into the database. Reification is not of particular interest in this thesis, so we will not go into more detail here, but instead refer the interested reader to [rdfc; rdfd].

Taken together, all these predefined URIs form a valuable basis for creating RDF documents. The `rdfs` vocabulary, which allows to express relationships between entities (such as `rdfs:subClassOf`, `rdfs:subPropertyOf`) and to fix property domains and ranges (using `rdfs:domain`, `rdfs:range`), is particularly useful to develop domain-specific vocabulary collections and ontologies (we implicitly used ontologies like FOAF [foa] and Dublin Core [dub] in the RDF database from Figure 2.2). The utilization of such predefined vocabulary collections facilitates RDF database design and increases the interoperability between RDF repositories in the Semantic Web.

[4]The exact kind and format of this definition is not further specified by the RDFS standard.

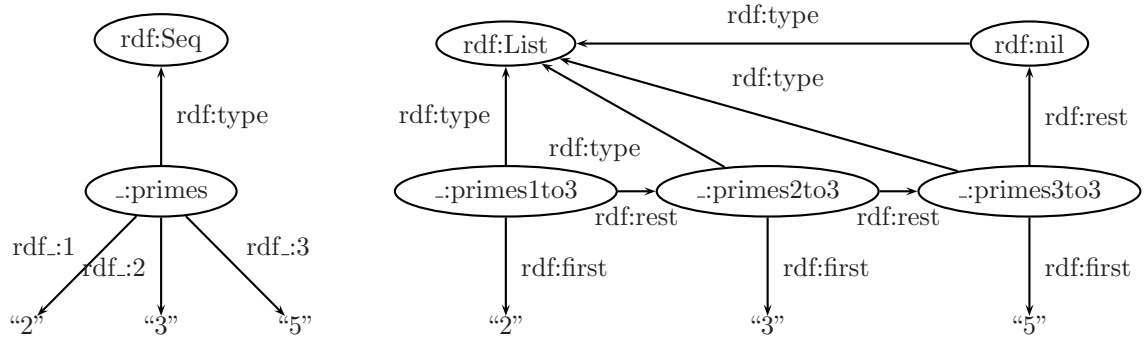

Figure 2.4.: (a) RDF sequence; (b) RDF list.

2.2.2. RDF(S) Semantics

In the previous section we introduced the RDF and RDFS standard vocabulary and described its meaning informally. Now, we turn towards a formal definition of its semantics. We start our discussion with a motivating example:

Example 2.2 Let us consider the RDF database D' below.

$D' := \{$ (myns:Article1,rdf:type,myns:ScientificArticle),
(myns:Article1,dc:creator,myns:Person1),
(myns:ScientificArticle,rdf:type,rdfs:Class),
(myns:Article,rdf:type,rdfs:Class),
(myns:Publication,rdf:type,rdfs:Class),
(myns:ScientificArticle,rdfs:subClassOf,myns:Article),
(myns:Article,rdfs:subClassOf,myns:Publication),
(dc:creator,rdfs:range,foaf:Person) $\}$

The database defines a resource myns:Article1 of type myns:ScientificArticle, which has been created by an entity referred to as myns:Person1 (yet the type of myns:Person1 is not specified). Further, some background schema knowledge is modeled. In particular, we know that myns:ScientificArticle is a subclass of myns:Article, which itself is a subclass of myns:Publication again. Finally, the range of property dc:creator is fixed to objects of type foaf:Person.

Let us now interpret the URIs in D' as real-world entities and discuss their semantics informally. We may derive a couple of new facts here. For instance, it is reasonable to assume that relation myns:subClassOf is transitive, which implies a new triple (myns:ScientificArticle,rdfs:subClassOf,myns:Publication). Given the resulting class hierarchy, the myns:ScientificArticle-typed object myns:Article1 should also be understood as an article (i.e., myns:Article) and – going one step further in the hierarchy – a publication (i.e., myns:Publication). This could be expressed by two additional triples (myns:Article1,rdfs:type,myns:Article) and

(I) Subclass

$$\frac{(\texttt{a}, \texttt{type}, \texttt{Class})}{(\texttt{a}, \texttt{sc}, \texttt{a})}$$

$$\frac{(\texttt{a}, \texttt{sc}, \texttt{b}) \qquad (\texttt{b}, \texttt{sc}, \texttt{c})}{(\texttt{a}, \texttt{sc}, \texttt{c})}$$

$$\frac{(\texttt{a}, \texttt{sc}, \texttt{b}) \qquad (\texttt{x}, \texttt{type}, \texttt{a})}{(\texttt{x}, \texttt{type}, \texttt{b})}$$

(II) Subproperty

$$\frac{(\texttt{a}, \texttt{type}, \texttt{Property})}{(\texttt{a}, \texttt{sp}, \texttt{a})}$$

$$\frac{(\texttt{a}, \texttt{sp}, \texttt{b}) \qquad (\texttt{b}, \texttt{sp}, \texttt{c})}{(\texttt{a}, \texttt{sp}, \texttt{c})}$$

$$\frac{(\texttt{a}, \texttt{sp}, \texttt{b}) \qquad (\texttt{x}, \texttt{a}, \texttt{y})}{(\texttt{x}, \texttt{b}, \texttt{y})}$$

(III) Typing

$$\frac{(\texttt{a}, \texttt{domain}, \texttt{c}) \qquad (\texttt{x}, \texttt{a}, \texttt{y})}{(\texttt{x}, \texttt{type}, \texttt{c})}$$

$$\frac{(\texttt{a}, \texttt{range}, \texttt{c}) \qquad (\texttt{x}, \texttt{a}, \texttt{y})}{(\texttt{y}, \texttt{type}, \texttt{c})}$$

Figure 2.5.: Selected RDFS inference rules.

(`myns:Article1`,`rdfs:type`,`myns:Publication`), respectively. As another example, the range specification `foaf:Person` for `dc:creator` implies that `myns:Person1`, which occurs as object in a triple with predicate `dc:creator`, is of type `foaf:Person`. This could be expressed by the triple (`myns:Person1`,`rdf:type`,`myns:Person`). □

In fact, the official RDF(S) semantics definition [rdfd] introduces a reasoning mechanism that allows to automatically derive new facts from RDF(S) knowledge bases, in the style of the previous example. The formal definition of the semantics that is proposed by the W3C is model theoretic, building on the notion of so-called *interpretations*. Informally speaking, the interpretation of an RDF(S) database contains all the triples that are logically implied according to the semantics.

In [GHM04] it was shown that the model theoretic approach proposed by the W3C can be equally expressed by a set of inference rules. From a practical perspective, these rules are more useful than the model theoretic definition: they can be successively applied (until a fixpoint is reached) to compute the logically implied database, whereas the model theoretic semantics is not constructive by nature. Figure 2.5 provides a subset of the RDFS reasoning rules introduced in [GHM04]. We use the shortcuts `sc` for `rdfs:subClassOf`, `sp` for `rdfs:subPropertyOf` and omit the namespace prefixes for `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:Class`, and `rdfs:Property`, simply writing `type`, `domain`, `Class`, and `Property` instead.

The rules in group (I) define the semantics of property `rdfs:subClassOf`. As usual, we understand the upper part of the inference rule as *premise* and the lower part as *consequence*. According to the first two rules in group (I), the subproperty relationship is reflexive and transitive, respectively. Finally, the third rule transfers subclass reasoning to the object level: whenever an object x is of some type a and it is known that class a is a subclass of b, then it follows that x also is of type b.

Next, group (II) covers the semantics of `rdfs:subPropertyOf`, which is quite similar in idea to the semantics of property `rdfs:subClassOf`. In particular, subproperty relationships also are reflexive and transitive, as stated by the first and

second rule in this group, respectively. Akin to the last rule in group (I), the last rule in group (II) transfers the transitivity of the relationship to the object level: under the premise that a is a subproperty of b and there is a resource x that stands in relation a with some y, it follows that resource x stands in relation b with y.

Finally, the two remaining rules in group (III) treat the semantics of properties `rdfs:domain` and `rdfs:range`. They allow to derive the respective types for entities that are used in subject and object positions of triples having property a whenever the RDF database contains a domain or range specification for a.

Example 2.3 When applying the inference rules from Figure 2.5 to database D' from Example 2.2 until a fixpoint is reached, we obtain the database

$$\begin{aligned} D'' := D' \cup \{ & (\texttt{myns:ScientificArticle},\texttt{rdfs:subClassOf},\texttt{myns:ScientificArticle}), \\ & (\texttt{myns:Article},\texttt{rdfs:subClassOf},\texttt{myns:Article}), \\ & (\texttt{myns:Publication},\texttt{rdfs:subClassOf},\texttt{myns:Publication}), \\ & (\texttt{myns:ScientificArticle},\texttt{rdfs:subClassOf},\texttt{myns:Publication}), \\ & (\texttt{myns:Article1},\texttt{rdf:type},\texttt{myns:Article}), \\ & (\texttt{myns:Article1},\texttt{rdf:type},\texttt{myns:Publication}), \\ & (\texttt{myns:Person1},\texttt{rdf:type},\texttt{myns:Person}) \}. \end{aligned}$$

We can observe that the implied database D'' includes all the triples that have been informally derived in the previous discussion in Example 2.2. □

It should be noted that the list of inference rules shown in Figure 2.5 is not complete. In particular, deduction rules concerning the semantics of blank nodes and rules for RDF-internal constructs, such as containers, are not included. These rules, however, are not important for the remainder of this thesis, so we refer the interested reader to [rdfd; GHM04] for a complete discussion of RDF(S) inference.

We wrap up with the remark that the SPARQL semantics that will be presented in the following section – just like the official W3C SPARQL specification – disregards the issue of RDFS reasoning. This means that SPARQL operates on the RDF graph as is, without inferring new triples. Whenever reasoning is desired, it is assumed to be carried out by a separate, underlying layer. This decision, which keeps the SPARQL query language independent from the reasoning process, brings several advantages: it results in a clean and compact semantics for SPARQL that does not interfere with reasoning rules, makes the SPARQL query language resistant to possible changes in the RDF(S) reasoning process, and allows to use SPARQL without modifications on top of other reasoning mechanisms, such as OWL [owl]. The RDF(S) semantics therefore will not play a role in our investigation of SPARQL complexity and optimization, yet when discussing semantic optimization of SPARQL queries in Chapter 5 we will come back to the issue of RDF(S) inference.

2.2.3. Serialization

There have been different proposals on how to serialize RDF(S) data. Probably the most prominent serialization format is RDF/XML [rdfe], which allows to encode RDF databases as XML trees. The basic idea behind RDF/XML is to split the RDF graph into small, tree-structured chunks, which are then described in XML with the help of predefined tags and attributes. The RDF/XML format was primarily designed to be processed by computers and we will not use it in this thesis.

In addition to RDF/XML, several triple-oriented serialization formats have been proposed. Arguably the simplest of them is N-Triples [ntr], which merely defines a syntax for the basic elements of RDF (i.e., URIs, blank nodes, and literals) and allows to encode an RDF database by listing its triples one by one. More advanced triple-based serialization formats for RDF are Turtle [tur] and Notation 3 [not] (also known as N3). Both languages extend the N-Triples format, e.g. by providing syntactic constructs that allow to encode RDF triples and collections of triples (such as sets of triples sharing the same subject) in a more compact way. We conclude with the remark that the sample RDF databases used in this thesis are rather small, so there is no urgent need to introduce simplifying notations; rather, we will fall back on our mathematical model of RDF databases (cf. Definition 2.2) and either describe them as sets of triples or visualize them using the graph representation.

2.3. The SPARQL Query Language

We now turn towards SPARQL and introduce a formal syntax (Section 2.3.1) and semantics (Sections 2.3.2 and 2.3.3) for a fragment of the query language. This fragment covers all SPARQL operators found in the official W3C Recommendation [spac], but abstracts from technical and implementation-specific details, such as data types and the typing system, which allows us to investigate the complexity and optimization of SPARQL using an expressive core fragment of the language.

In our formalization, we will introduce two alternative semantics for SPARQL evaluation, namely a *set semantics* and a *bag semantics* (also referred to as *multi-set semantics*). The set semantics is simpler and therefore will form the basis for our discussion of SPARQL complexity in Chapter 3 (yet all complexity results carry over to the bag semantics), while – in the course of our study of algebraic SPARQL optimization in Chapter 4 – we will study set and bag semantics in parallel, to work out practically relevant differences. As a final step in our introduction to SPARQL we will then relate our formalization to the official W3C SPARQL Recommendation (Section 2.3.4) and discuss limitations of the language (Section 2.3.5).

We point out that our formalization of SPARQL in Sections 2.3.1 and 2.3.2 was inspired by previous investigations of SPARQL (cf. [PAG06a; AG08a]); the bag semantics in Section 2.3.3 closely follows ideas found in the official W3C Recom-

mendation [spac] and the bag semantics proposed in [PAG06b]. A historical note on the SPARQL formalization process can be found in the related work, Section 2.4.

2.3.1. An Abstract Syntax for SPARQL

Let V be a set of variables disjoint from BLU. As a notational convenience, we distinguish variables by a leading question mark symbol, e.g. strings like $?x$ and $?name$ denote variable names. We start with an abstract syntax for filter conditions.

Definition 2.3 (Filter Condition) Let $?x, ?y \in V$ be variables and $c, d \in LU$. We define *filter conditions* inductively as follows.

- The expressions $?x = c$, $?x = ?y$, and $c = d$ are filter conditions.
- The expression $bound(?x)$ (abbreviated as $bnd(?x)$) is a filter condition.
- If R_1 and R_2 are filter conditions,
  then $\neg R_1$, $R_1 \wedge R_2$, and $R_1 \vee R_2$ are filter conditions. □

By $vars(R)$ we denote the set of variables occurring in filter expression R. Using the previous definition, we are ready to introduce an abstract syntax for expressions (where we abbreviate the operator OPTIONAL as OPT):

Definition 2.4 (SPARQL Expression) A *SPARQL expression* is an expression that is built inductively according to the following rules.

- A so-called *triple pattern* $t \in UV \times UV \times LUV$ is an expression.
- If Q is an expression and R is a filter condition,
  then Q FILTER R is an expression.
- If Q_1, Q_2 are expressions,
  then Q_1 UNION Q_2, Q_1 OPT Q_2, and Q_1 AND Q_2 are expressions. □

Note that we do not allow for blank nodes in triple patterns, so SPARQL expressions (and also SPARQL queries, which will be defined subsequently) by definition never contain blank nodes, yet the official W3C SPARQL Recommendation [spac] allows for blank nodes in triple patterns. We will further elaborate on this issue Section 2.3.4 when relating our SPARQL fragment to the W3C proposal for SPARQL.

The official W3C Recommendation defines four different types of queries, namely SELECT, ASK, CONSTRUCT, and DESCRIBE queries. In the formalization we will restrict ourselves on SPARQL SELECT and ASK queries. While SELECT queries extract the set of all result mappings, ASK queries are boolean queries that return *true* if there are some results, and *false* otherwise.[5] We next define the syntax of

[5]Unlike the official W3C SPARQL Recommendation, which proposes *"yes"* and *"no"* as answers to ASK queries, we use the standard boolean predicates *"true"* and *"false"* in our formalization.

SELECT and ASK queries (their semantics will be fixed in Section 2.3.2). The remaining two query forms, CONSTRUCT and DESCRIBE, will be informally discussed later in Section 2.3.4; they are not of particular interest in this thesis.

Definition 2.5 (SPARQL SELECT Query) Let Q be a SPARQL expression and let $S \subset V$ be a finite set of variables. A *SPARQL* SELECT *query* is an expression of the form $\text{SELECT}_S(Q)$. □

Definition 2.6 (SPARQL ASK Query) Let Q be a SPARQL expression. We call an expression of the form $\text{ASK}(Q)$ *SPARQL* ASK *query*. □

In the remainder of the thesis we will mostly deal with SPARQL SELECT queries. Therefore, we usually denote them as *SPARQL queries*, or simply *queries*. As a notational simplification, we omit braces for the variable set appearing in the subscript of the SELECT operator, e.g. writing $\text{SELECT}_{?x,?y}(Q)$ instead of $\text{SELECT}_{\{?x,?y\}}(Q)$. Further, in the context of abstract syntax SPARQL expressions and queries, we typeset URIs in italic font and often omit the prefixes of URIs.

Example 2.4 The query

$$
\begin{aligned}
Q_1 := \text{SELECT}_{?name,?email}(& \\
& (((?person, name, ?name) \text{ AND } (?person, age, ?age)) \\
& \text{ FILTER } (?age = \text{“30”})) \\
& \text{OPT } (?person, email, ?email))
\end{aligned}
$$

is a valid SPARQL SELECT query, where *name*, *age*, and *email* denote URIs. □

2.3.2. A Set-based Semantics for SPARQL

Having established an abstract syntax for SPARQL, we now provide a set-based semantics for its evaluation. We start with the definition of so-called *mappings*, which are used to express variable-to-graph bindings within in the evaluation process:

Definition 2.7 (Mapping and Mapping Universe) A *mapping* is a partial function $\mu : V \rightarrow BLU$ from a subset of variables V to RDF terms BLU. The domain of a mapping μ, written $dom(\mu)$, is defined as the subset of V for which μ is defined. By $\mathcal{M}$ we denote the universe of all mappings. □

We next define the central notion of *compatibility* between mappings. Informally speaking, two mappings are compatible if they do not contain contradicting variable bindings, i.e. if shared variables always map to the same value in both mappings:

Definition 2.8 (Compatibility of Mappings) Given two mappings μ_1, μ_2, we say μ_1 is *compatible* with μ_2 iff $\mu_1(?x) = \mu_2(?x)$ for all $?x \in dom(\mu_1) \cap dom(\mu_2)$. We write $\mu_1 \sim \mu_2$ if μ_1 and μ_2 are compatible, and $\mu_1 \not\sim \mu_2$ otherwise. □

We overload function *vars* (introduced before for filter expressions) and denote by $vars(t)$ all variables in triple pattern t. Further, we write $\mu(t)$ to denote the triple pattern obtained when replacing all variables $?x \in dom(\mu) \cap vars(t)$ in t by $\mu(?x)$.

Example 2.5 Let

$$\mu_1 := \{?person \mapsto P1, ?name \mapsto \text{“Pete”}\},$$
$$\mu_2 := \{?person \mapsto P2, ?name \mapsto \text{“John”}\},$$
$$\mu_3 := \{?person \mapsto P1, ?email \mapsto \text{“pete@tld.com”}\}$$

be three mappings. Then it holds that $dom(\mu_1) = dom(\mu_2) = \{?person, ?name\}$ and $dom(\mu_3) = \{?person, ?email\}$. Further, $\mu_1 \sim \mu_3$, but $\mu_1 \not\sim \mu_2$ and $\mu_2 \not\sim \mu_3$ (they disagree on variable *?person*). Given triple pattern $t_1 := (?person, name, ?name)$ we have $vars(t_1) = \{?person, ?name\}$ and, for instance, $\mu_1(t_1) = (P1, name, \text{“Pete”})$. □

The next prerequisite for our definition of SPARQL semantics is the notion of *satisfaction* of a filter condition with respect to some mapping:[6]

Definition 2.9 (Filter Semantics) Given a mapping μ, filter conditions R, R_1, R_2, variables $?x$, $?y$, and $c, d \in LU$, we say that μ *satisfies* R, written as $\mu \models R$, if and only if one of the following conditions holds.

- R is of the form $bnd(?x)$ and $?x \in dom(\mu)$.
- R is of the form $c = d$ and it holds that c and d are equal.
- R is of the form $?x = c$, $?x \in dom(\mu)$, and $\mu(?x) = c$.
- R is of the form $?x = ?y$, $\{?x, ?y\} \subseteq dom(\mu)$, and $\mu(?x) = \mu(?y)$.
- R is of the form $\neg R_1$ and it is not the case that $\mu \models R_1$.
- R is of the form $R_1 \vee R_2$ and $\mu \models R_1$ or $\mu \models R_2$.
- R is of the form $R_1 \wedge R_2$ and $\mu \models R_1$ and $\mu \models R_2$. □

The solution of evaluating a SPARQL expression or query on some document D is described by a set of mappings, where each single mapping represents a possible answer in form of a binding of query variables to the subset of elements from set BLU that occur in D. Following the approach proposed in [PAG06a], we define the semantics of our SPARQL fragment using a compact algebra over such mapping sets. We introduce the required algebraic operators in the subsequent definition.

[6] The W3C proposes a three-valued semantics for filters (*true*, *false*, plus *error*). Here, we adopt the two-valued approach (using only *true* and *false*) from [PAG09]. We want to emphasize that all results stated in the thesis also hold under the three-valued semantics.

Definition 2.10 (SPARQL Set Algebra) Let Ω, Ω_l, Ω_r be mapping sets, R denote a filter condition, and $S \subset V$ be a finite set of variables. We define the algebraic operations *join* $\bowtie$, *union* $\cup$, *minus* $\setminus$, *left outer join* $⟕$, *projection* π, and *selection* σ (also called *filter*) over mapping sets as follows.

$$
\begin{aligned}
\Omega_l \bowtie \Omega_r &:= \{\mu_l \cup \mu_r \mid \mu_l \in \Omega_l, \mu_r \in \Omega_r : \mu_l \sim \mu_r\} \\
\Omega_l \cup \Omega_r &:= \{\mu \mid \mu \in \Omega_l \text{ or } \mu \in \Omega_r\} \\
\Omega_l \setminus \Omega_r &:= \{\mu_l \in \Omega_l \mid \text{for all } \mu_r \in \Omega_r : \mu_l \not\sim \mu_r\} \\
\Omega_l ⟕ \Omega_r &:= (\Omega_l \bowtie \Omega_r) \ \cup \ (\Omega_l \setminus \Omega_r) \\
\pi_S(\Omega) &:= \{\mu_1 \mid \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega \wedge dom(\mu_1) \subseteq S \wedge dom(\mu_2) \cap S = \emptyset\} \\
\sigma_R(\Omega) &:= \{\mu \in \Omega \mid \mu \models R\}
\end{aligned}
$$

We refer to the algebra defined by the above operations as *SPARQL set algebra.*□

We shall illustrate and discuss the algebraic operations using the mapping sets

$$
\begin{aligned}
&\Omega_1 := \{\{?person \mapsto P1, ?name \mapsto \text{“Joe”}\}, \{?person \mapsto P2, ?name \mapsto \text{“John”}\}\}, \\
&\Omega_2 := \{\{?person \mapsto P1, ?email \mapsto \text{“joe@tld.com”}\}\}.
\end{aligned}
$$

Operator $\bowtie$ is a join operation that combines compatible mappings from two mapping sets. To show the effect of the join operation let us consider the SPARQL set algebra expression $\Omega_1 \bowtie \Omega_2$. According to the semantics of the join operation, the result is the union of the first mapping from Ω_1 and the (only) mapping in Ω_2, because these mappings are compatible; the second mapping in Ω_1 is incompatible with every mapping in Ω_2 and hence does not generate a result. Consequently, we have $\Omega_1 \bowtie \Omega_2 = \{\{?person \mapsto P1, ?name \mapsto \text{“Joe”}, ?email \mapsto \text{“joe@tld.com”}\}\}$.

The union operator $\cup$ returns the mapping set that contains all mappings from the first or the second set, i.e. $\Omega_1 \cup \Omega_2 = \{\{?person \mapsto P1, ?name \mapsto \text{“Joe”}\}, \{?person \mapsto P2, ?name \mapsto \text{“John”}\}, \{?person \mapsto P1, ?email \mapsto \text{“joe@tld.com”}\}\}$.

Next, the minus operator $\setminus$ retains all mappings from the left side set for which no compatible mappings in the right side set exists. For our example sets Ω_1, Ω_2, we observe that for the mapping $\{?person \mapsto P2, ?name \mapsto \text{“John”}\}$ there is no compatible mapping in Ω_2, whereas the first mapping in Ω_1 is compatible with the only mapping in Ω_2. Hence, we have $\Omega_1 \setminus \Omega_2 = \{\{?person \mapsto P2, ?name \mapsto \text{“John”}\}\}$.

Probably the most interesting operator is the left outer join $⟕$, which is defined as $\Omega_l ⟕ \Omega_r := (\Omega_l \bowtie \Omega_r) \cup (\Omega_l \setminus \Omega_r)$. The idea is as follows. Given a mapping μ from Ω_l, we distinguish two cases: (i) if there are compatible mappings in Ω_r, then the left side of the union $\Omega_l \bowtie \Omega_r$ applies and μ is joined with all compatible mappings from Ω_r; otherwise, (ii) if there is no compatible mapping in Ω_r, then μ itself is part of the result, according to the right side of the union. To give an example, we have

$$\begin{aligned}\Omega_1 \mathbin{⟕} \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\\ &= \{\{?person \mapsto P1, ?name \mapsto \text{“Joe”}, ?email \mapsto \text{“joe@tld.com”}\}\} \cup\\ &\quad \{\{?person \mapsto P2, ?name \mapsto \text{“John”}\}\}\\ &= \{\{?person \mapsto P1, ?name \mapsto \text{“Joe”}, ?email \mapsto \text{“joe@tld.com”}\},\\ &\quad \{?person \mapsto P2, ?name \mapsto \text{“John”}\}\}.\end{aligned}$$

As can be seen, the left outer join operation allows additional (compatible) information to be added if it exists (e.g., mapping $\{?person \mapsto P1, ?name \mapsto \text{“Joe”}\}$ from Ω_1 is extended by a binding for variable *?email*), but does not discard mappings for which no such additional information is present (which is the case for mapping $\{?person \mapsto P2, ?name \mapsto \text{“John”}\}$ in our example). As we will see soon, the left outer join operation will be utilized to define the semantics of operator Opt.

We conclude our discussion of the algebraic operators with the projection and selection operators π and σ. Operator π is a straightforward projection on a set of variables, e.g. the expression $\pi_{?name}(\Omega_1) = \{\{?name \mapsto \text{“Joe”}\}, \{?name \mapsto \text{“John”}\}\}$ restricts all mappings in Ω_1 to variable *?name*. Finally, operator σ implements a filter in the style of relational algebra selection. For instance, the algebra expression $\sigma_{?name=\text{“Joe”}}(\Omega_1)$ selects all mappings from Ω_1 in which variable *?name* is bound to value *“Joe”*, thus we have $\sigma_{?name=\text{“Joe”}}(\Omega_1) = \{\{?person \mapsto P1, ?name \mapsto \text{“Joe”}\}\}$.

Having explained the algebraic operators, we are now ready to define the semantics of SPARQL expressions, Select queries, and Ask queries (cf. Definitions 2.4, 2.5, and 2.6, respectively). We follow the compositional, set-based approach proposed in [PAG06a] and define a function $[\![.]\!]_D$ that maps an abstract syntax SPARQL expression or query into an algebra expression, which is then evaluated according to the semantics of SPARQL set algebra introduced before in Definition 2.10:

Definition 2.11 (SPARQL Semantics) Let t be a triple pattern, D be an RDF database, Q_1, Q_2 denote SPARQL expressions, R be a filter condition, and $S \subset V$ be a finite set of variables. We define the SPARQL semantics inductively as follows.

$$\begin{aligned}[\![t]\!]_D &:= \{\mu \mid dom(\mu) = vars(t) \text{ and } \mu(t) \in D\}\\ [\![Q_1 \text{ And } Q_2]\!]_D &:= [\![Q_1]\!]_D \bowtie [\![Q_2]\!]_D\\ [\![Q_1 \text{ Opt } Q_2]\!]_D &:= [\![Q_1]\!]_D \mathbin{⟕} [\![Q_2]\!]_D\\ [\![Q_1 \text{ Union } Q_2]\!]_D &:= [\![Q_1]\!]_D \cup [\![Q_2]\!]_D\\ [\![Q_1 \text{ Filter } R]\!]_D &:= \sigma_R([\![Q_1]\!]_D)\\ [\![\text{Select}_S(Q_1)]\!]_D &:= \pi_S([\![Q_1]\!]_D)\\ [\![\text{Ask}(Q_1)]\!]_D &:= \neg(\emptyset = [\![Q_1]\!]_D)\end{aligned}$$ □

We illustrate the evaluation of SPARQL queries in the following example.

Example 2.6 Consider SPARQL query Q_1 from Example 2.4, which retrieves the names of all 30-year-old persons and, optionally (i.e., if specified), their email address. We discuss the evaluation of Q_1 on the RDF database

$$\begin{aligned} D := \{&(P1, name, \text{“}Joe\text{”}), (P1, age, \text{“}30\text{”}), (P1, email, \text{“}joe@tld.com\text{”})), \\ &(P2, name, \text{“}John\text{”}), (P2, age, \text{“}29\text{”}), (P2, email, \text{“}john@tld.com\text{”}), \\ &(P3, name, \text{“}Pete\text{”}), (P3, age, \text{“}30\text{”})\}. \end{aligned}$$

In a first step, we apply the semantics $[\![.]\!]_D$ to expression Q_1, which transforms operators And, Opt, Select, and Filter into their algebraic counterparts:

$$\begin{aligned} [\![Q_1]\!]_D = \ & \pi_{?name,?email}(\\ & \sigma_{?age=\text{“}30\text{”}}([\![(?person, name, ?name)]\!]_D \bowtie [\![(?person, age, ?age)]\!]_D) \\ & \bowtie [\![(?person, email, ?email)]\!]_D). \end{aligned}$$

Next, we evaluate the basic triple patterns of this algebra expression, using the topmost rule from Definition 2.11. We obtain the following intermediate results.

$$\begin{aligned} [\![(?person, name, ?name)]\!]_D = \{&\{?person \mapsto P1, ?name \mapsto \text{“}Joe\text{”}\}, \\ &\{?person \mapsto P2, ?name \mapsto \text{“}John\text{”}\}, \\ &\{?person \mapsto P3, ?name \mapsto \text{“}Pete\text{”}\}\} \\ [\![(?person, age, ?age)]\!]_D = \{&\{?person \mapsto P1, ?age \mapsto \text{“}30\text{”}\}, \\ &\{?person \mapsto P2, ?age \mapsto \text{“}29\text{”}\}, \\ &\{?person \mapsto P3, ?age \mapsto \text{“}30\text{”}\}\} \\ [\![(?person, email, ?email)]\!]_D = \{&\{?person \mapsto P1, ?email \mapsto \text{“}joe@tld.com\text{”}\}, \\ &\{?person \mapsto P2, ?email \mapsto \text{“}john@tld.com\text{”}\}\} \end{aligned}$$

We are now ready to evaluate the algebra expression $[\![Q_1]\!]_D$ from above bottom-up:

$$\begin{aligned} \Omega_1 &:= [\![(?person, name, ?name)]\!]_D \bowtie [\![(?person, age, ?age)]\!]_D \\ &= \{\{?person \mapsto P1, ?name \mapsto \text{“}Joe\text{”}, ?age \mapsto \text{“}30\text{”}\}, \\ &\quad \{?person \mapsto P2, ?name \mapsto \text{“}John\text{”}, ?age \mapsto \text{“}29\text{”}\}, \\ &\quad \{?person \mapsto P3, ?name \mapsto \text{“}Pete\text{”}, ?age \mapsto \text{“}30\text{”}\}\}, \end{aligned}$$

$$\begin{aligned} \Omega_2 &:= \sigma_{?age=\text{“}30\text{”}}(\Omega_1) \\ &= \{\{?person \mapsto P1, ?name \mapsto \text{“}Joe\text{”}, ?age \mapsto \text{“}30\text{”}\}, \\ &\quad \{?person \mapsto P3, ?name \mapsto \text{“}Pete\text{”}, ?age \mapsto \text{“}30\text{”}\}\}, \end{aligned}$$

$$\begin{aligned} \Omega_3 &:= \Omega_2 \bowtie [\![(?person, email, ?email)]\!]_D \\ &= (\Omega_2 \bowtie [\![(?person, email, ?email)]\!]_D) \cup (\Omega_2 \setminus [\![(?person, email, ?email)]\!]_D) \\ &= \{\{?person \mapsto P1, ?name \mapsto \text{“}Joe\text{”}, ?age \mapsto \text{“}30\text{”}, ?email \mapsto \text{“}joe@tld.com\text{”}\}\} \\ &\quad \cup \{\{?person \mapsto P3, ?name \mapsto \text{“}Pete\text{”}, ?age \mapsto \text{“}30\text{”}\}\} \\ &= \{\{?person \mapsto P1, ?name \mapsto \text{“}Joe\text{”}, ?age \mapsto \text{“}30\text{”}, ?email \mapsto \text{“}joe@tld.com\text{”}\}, \\ &\quad \{?person \mapsto P3, ?name \mapsto \text{“}Pete\text{”}, ?age \mapsto \text{“}30\text{”}\}\}, \end{aligned}$$

$$\begin{aligned} \Omega &:= \pi_{?name,?email}(\Omega_3) \\ &= \{\{?name \mapsto \text{“}Joe\text{”}, ?email \mapsto \text{“}joe@tld.com\text{”}\}, \{?name \mapsto \text{“}Pete\text{”}\}\}. \end{aligned}$$

Observe that variable *?email* is not bound in the second mapping of the final result Ω, which shows that query variables are not necessarily bound in result mappings. Such unbound variables are crucial for our discussion of SPARQL complexity and algebraic optimization. We will come back to this observation in later chapters.□

In the remainder of this thesis, we will always fully parenthesize expressions, except for the case of AND- and UNION-expressions; as we will see in Chapter 4, the latter operators are associative, for instance Q_1 AND (Q_2 AND Q_3) is equivalent to (Q_1 AND Q_2) AND Q_3, so we shall write Q_1 AND Q_2 AND Q_3 in this case.

2.3.3. From Set to Bag Semantics

Given the set semantics defined in the previous subsection, we now define an alternative bag semantics for SPARQL. It differs in that identical mappings might appear multiple times in (intermediate) query results. We point out that the bag semantics defined in this subsection corresponds to the approach taken by the W3C [spac].[7]

The central idea of the subsequent bag semantics definition is to switch from sets of mappings to *multi-sets of mappings*. Therefore, in the remainder of this thesis we will use the terms multi-set semantics and bag semantics interchangeably. We start our discussion with a formal definition of mapping multi-sets:

Definition 2.12 (Mapping Multi-set) A *mapping multi-set* is a tuple (Ω,m), where $\Omega \subset \mathcal{M}$ is a mapping set and $m : \mathcal{M} \mapsto \mathbb{N}_0$ is a total function such that $m(\mu) \geq 1$ for all $\mu \in \Omega$ and $m(\mu) = 0$ for all $\mu \notin \Omega$. Given $\mu \in \Omega$, we refer to $m(\mu)$ as the *multiplicity* of μ in Ω, saying that μ *occurs* $m(\mu)$ *times in* Ω. □

We note that, for a mapping multi-set (Ω, m), the associated function m implicitly defines Ω (i.e. Ω contains exactly those mappings μ for which $m(\mu) \geq 1$ holds), so strictly speaking the specification of Ω is redundant. The reason for making Ω explicit is to clarify the connection between sets and multi-sets and to facilitate subsequent definitions and proofs. We illustrate Definition 2.12 in the following example.

Example 2.7 Let $\mu_1 := \{?x \mapsto a\}$ and $\mu_2 := \{?y \mapsto b\}$ be mappings. Then (Ω,m) with $\Omega := \{\mu_1, \mu_2\}$ and $m(\mu_1) := 2$, $m(\mu_2) := 1$, and $m(\mu) := 0$ for all $\mu \in \mathcal{M} \setminus \Omega$ is a mapping multi-set, in which μ_1 occurs twice and μ_2 a single time. □

Whenever the multiplicity equals to one for all mappings that are contained in some mapping multi-set, it can be understood as a simple mapping set:

[7] As discussed in [AG08a], there are some minor differences on how the W3C maps SPARQL syntax into SPARQL algebra. These differences, however, do not compromise the expressiveness. We also emphasize that the core evaluation phase (i.e. the result computation for a fixed SPARQL bag algebra expression over some document) is identical to the W3C approach.

Definition 2.13 (Equivalence of Mapping Sets and Multi-sets) Let Ω be a mapping set and (Ω^+, m^+) a mapping multi-set. We say that Ω *equals* (Ω^+, m^+), written as $\Omega \cong (\Omega^+, m^+)$, iff it holds that $\Omega = \Omega^+$ and $m^+(\mu) = 1$ for all $\mu \in \Omega^+$.□

The bag semantics is now defined using adapted versions of the algebraic operations from Definition 2.10, modified to operate on top of multi-sets and to take the multiplicity of the set elements into account. Implementing this idea, we overload the algebraic operations from Definition 2.10 as follows.

Definition 2.14 (SPARQL Bag Algebra) Let $M := (\Omega, m)$, $M_l := (\Omega_l, m_l)$, $M_r := (\Omega_r, m_r)$ be mapping multi-sets, R denote a filter condition, and $S \subset V$ be a finite set of variables. We define the operations *join* $\bowtie$, *union* $\cup$, *minus* $\setminus$, *left outer join* $⟕$, *projection* π, and *selection* σ over mapping multi-sets:

$M_l \bowtie M_r := (\Omega', m')$, where
$\Omega' := \{\mu_l \cup \mu_r \mid \mu_l \in \Omega_l, \mu_r \in \Omega_r : \mu_l \sim \mu_r\}$ and for all $\mu \in \mathcal{M}$ we set
$m'(\mu) := \sum_{(\mu_l,\mu_r)\in\{(\mu_l^*,\mu_r^*)\in\Omega_l\times\Omega_r \mid \mu_l^*\cup\mu_r^*=\mu\}} (m_l(\mu_l) * m_r(\mu_r))$.

$M_l \cup M_r := (\Omega', m')$, where
$\Omega' := \{\mu \mid \mu \in \Omega_l \text{ or } \mu \in \Omega_r\}$ and for all $\mu \in \mathcal{M}$ we set
$m'(\mu) := m_l(\mu) + m_r(\mu)$.

$M_l \setminus M_r := (\Omega', m')$, where
$\Omega' := \{\mu_l \in \Omega_l \mid \text{for all } \mu_r \in \Omega_r : \mu_l \not\sim \mu_r\}$, and for all $\mu \in \mathcal{M}$ we set
$m'(\mu) := \begin{cases} m_l(\mu) & \text{if } \mu \in \Omega', \\ 0 & \text{otherwise.} \end{cases}$

$M_l ⟕ M_r := (M_l \bowtie M_r) \cup (M_l \setminus M_r)$

$\pi_S(M) := (\Omega', m')$, where
$\Omega' := \{\mu_1 \mid \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega \wedge dom(\mu_1) \subseteq S \wedge dom(\mu_2) \cap S = \emptyset\}$
and for all $\mu \in \mathcal{M}$ we set $m'(\mu) := \sum_{\mu_+\in\{\mu_+^*\in\Omega \mid \pi_S(\{\mu_+^*\})=\{\mu\}\}} m(\mu_+)$.

$\sigma_R(M) := (\Omega', m')$, where
$\Omega' := \{\mu \in \Omega \mid \mu \models R\}$ and for all $\mu \in \mathcal{M}$ we set
$m'(\mu) := \begin{cases} m(\mu) & \text{if } \mu \in \Omega', \\ 0 & \text{otherwise.} \end{cases}$

We refer to the above algebra as *SPARQL bag algebra.* □

The above definition exactly corresponds to Definition 2.10 w.r.t. the mappings that are contained in the result set (i.e., the definition of Ω' in each rule mirrors the definition of SPARQL set algebra); it differs, however, in that it additionally fixes the multiplicities for generated set members (cf. function m' in each rule). To give an example, consider the computation of m' in the definition of the $\bowtie$ operator. For each $\mu \in \mathcal{M}$ we define its multiplicity by summing up the multiplicities of all (compatible)

decompositions $\mu_l \in \Omega_l$, $\mu_r \in \Omega_r$ that generate μ when merged together, thereby taking their multiplicities in Ω_l and Ω_r into account. Observe that, if $\mu \notin \Omega'$ then there exists no such decomposition and consequently $m'(\mu)$ is zero by definition.

As an interesting observation, for the minus operation $M_l \setminus M_r$ we set the resulting multiplicity $m'(\mu)$ either to $m_l(\mu)$ (i.e., we take over the multiplicity $m_l(\mu)$ of the mapping μ in M_l if there is no compatible mapping in Ω_r) or set it to 0 (whenever there is at least one compatible mapping in Ω_r). We emphasize that this is exactly the strategy proposed by the W3C [spac]. This strategy, however, contrasts with the standard definition of the difference operator of relational algebra under bag semantics, which forms the basis for SQL implementations. There, the difference between two relational algebra expressions, say $S - R$, is computed by subtracting, for each tuple appearing in S, its multiplicities in S and R (see e.g. [GMUW00]). More precisely, if tuple t appears m times in S and n times in R, then t is contained in the result of $S - R$ if $m > n$ and its multiplicity in $S - R$ is defined as $m - n$.

We finally point out that Definition 2.14 is correct in the sense that the algebraic operations always return multi-sets that are valid according to Definition 2.12, i.e. whenever an operator generates a multi-set (Ω^+, m^+) then $m^+(\mu) \geq 1$ for all $\mu \in \Omega^+$ and $m^+(\mu) = 0$ for all $\mu \in \mathcal{M} \setminus \Omega^+$. This property can be easily proven by a case-by-case discussion of the algebraic operations. We omit the formal proof.

Based on the bag algebra for SPARQL, we can now formally define the bag semantics, similar in style to the set semantics introduced in Definition 2.11:

Definition 2.15 (SPARQL Bag Semantics) Let t be a triple pattern, D be an RDF database, Q_1, Q_2 denote SPARQL expressions, R a filter condition, and $S \subset V$ be a finite set of variables. We define the bag semantics inductively as follows.

$$
\begin{aligned}
&[\![t]\!]_D^+ := (\Omega, m), \text{ where } \Omega := \{\mu \mid \mathit{dom}(\mu) = \mathit{vars}(t) \text{ and } \mu(t) \in D\} \\
&\qquad\qquad \text{and } m(\mu) := 1 \text{ for all } \mu \in \Omega,\ m(\mu) := 0 \text{ otherwise.} \\
&[\![Q_1 \text{ AND } Q_2]\!]_D^+ \quad := [\![Q_1]\!]_D^+ \bowtie [\![Q_2]\!]_D^+ \\
&[\![Q_1 \text{ OPT } Q_2]\!]_D^+ \quad := [\![Q_1]\!]_D^+ \mathbin{⟕} [\![Q_2]\!]_D^+ \\
&[\![Q_1 \text{ UNION } Q_2]\!]_D^+ := [\![Q_1]\!]_D^+ \cup [\![Q_2]\!]_D^+ \\
&[\![Q_1 \text{ FILTER } R]\!]_D^+ := \sigma_R([\![Q_1]\!]_D^+) \\
&[\![\text{SELECT}_S(Q_1)]\!]_D^+ := \pi_S([\![Q_1]\!]_D^+) \\
&[\![\text{ASK}(Q_1)]\!]_D^+ \quad := \neg(\emptyset \cong [\![Q_1]\!]_D^+)
\end{aligned}
$$

□

The definition is identical to Definition 2.11, except for the case of triple pattern evaluation. In particular, we represent the result of evaluating a triple pattern as a multi-set (instead of a set), where we associate multiplicity 1 to each result mapping. Hence, when evaluating a SPARQL expression bottom-up using bag semantics, algebraic operations will always be interpreted as multi-set operations.

The next definition allows us to compare the results obtained from evaluating SPARQL expressions and queries with the different semantics.

Definition 2.16 (Coincidence of Semantics) Let Q be a SPARQL expression or query and D be an RDF document. We say that *the set and bag semantics coincide for Q on D* iff $[\![Q]\!]_D \cong [\![Q]\!]_D^+$. □

In general, the two semantics do not coincide, as witnessed by the example below.

Example 2.8 Consider the SPARQL expression $Q := (?x, c, c)$ UNION $(c, c, ?x)$ and document $D := \{(c, c, c)\}$. We observe that $[\![Q]\!]_D = \{\mu\}$ with $\mu := \{?x \mapsto c\}$ and $[\![Q]\!]_D^+ = (\{\mu\}, m)$ with $m(\mu) := 2$, so the semantics do **not** coincide for Q on D. □

2.3.4. Official W3C SPARQL Syntax and Semantics

The abstract syntax model and the corresponding semantics for SPARQL presented in Sections 2.3.1-2.3.3 form the basis for our theoretical investigations, such as the SPARQL complexity analysis in Chapter 3 and algebraic optimization rules in Chapter 4. Yet, when discussing more practical aspects of the SPARQL query language such as the SP^{2}Bench benchmark in Chapter 6, we will fall back on the official SPARQL syntax and semantics. In this section we therefore relate our formal model to the official W3C specification and discuss both similarities and differences.

Syntax. We start with a survey of syntactic differences between our formal model and the W3C specification. Let us open the discussion with the official W3C SPARQL syntax for the (abstract syntax) query from Example 2.4:

```
SELECT ?name ?email
WHERE {
  ?person name ?name.
  ?person age ?age
  FILTER (?age="30")
  OPTIONAL { ?person email ?email } }
```

First note that operator AND is abbreviated as ".". Further, the keyword OPTIONAL is used in place of the shortcut OPT. The naming of operators SELECT, UNION (not shown in the query), and FILTER remains unchanged. As a minor syntactic difference, triple patterns are not parenthesized and the body of the query is enclosed into a separate block, defined by the keyword WHERE. It is worth mentioning that, in analogy to our definition of queries (cf. Definition 2.5), the SELECT-operator appears always (and only) at the top-level; in particular, the SPARQL specification, just like our formalization, does not allow for nested subqueries.

A major conceptual difference between both syntaxes is that operators in the official W3C syntax are not denoted in the style of mathematical binary operations,

but using a more declarative syntax. Accounting for this difference, the W3C specification provides a set of parsing and grouping rules, defining how to transform the syntax expression into an algebra expression. The W3C algebra itself is similar to the SPARQL bag algebra introduced in Definition 2.14.[8] We do not formally restate the complete W3C syntax and semantics here, but instead will explain the meaning of queries informally whenever using W3C syntax, to rule out ambiguities.

Set vs. Bag Semantics. We presented two different semantics, namely the set-based version $[\![.]\!]_D$ and the bag semantics $[\![.]\!]_D^+$ The W3C SPARQL specification follows the second approach, thus is close to the bag semantics introduced in Definition 2.15. As we shall see in Chapter 3, the issue of bag vs. set semantics does not play a role from a complexity point of view. In our study of SPARQL algebra optimization in Chapter 4, however, we will study set and bag semantics separately and highlight differences that are of immediate practical relevance.

Solution Modifiers. Our SPARQL formalization focuses on a core fragment of the SPARQL query language, i.e. it implements only a subset of the features defined in the official W3C SPARQL specification. One construct that is included in the official recommendation but was not considered in our formalization are the so-called *solution modifiers*, which can be used to manipulate the extracted result set. We informally describe the existing solution modifiers in the listing below.

- The DISTINCT modifier filters away duplicate mappings in the extracted result set. Note that this modifier makes only sense in combination with bag semantics, because under set semantics no duplicate solutions exist at all. We will formalize and investigate the DISTINCT modifier later in Section 4.4.2.
- While DISTINCT eliminates duplicate solutions, the REDUCED modifier offers flexibility to the query optimizer in that it **permits** to eliminate duplicates. The decision on whether to eliminate duplicates or not thus can be made on a case-by-case basis, e.g. by comparing the estimated costs for both alternatives. Like DISTINCT, the REDUCED modifier will be investigated in Section 4.4.2.
- ORDER BY allows to sort the extracted result according to some variables, either in descending or ascending order. The sort order can be fixed using the keywords DESC and ASC (where ASC is used as default sort order).
- The solution modifiers LIMIT and OFFSET can be used to fix the number of results that are returned and the first result mapping that should be output, respectively. They are particularly useful when combined with modifier ORDER BY. As an example query that uses such a modifier constellation, consider $Q11$ in Section 6.4.3. The body of the query extracts electronic editions of publications. According to the solution modifiers, the result is sorted by variable *?ee* in ascending lexicographical order. Modifier "LIMIT 10" enforces that only (up to) 10 results are returned, while "OFFSET 50" asserts that the output contains electronic editions starting from the 51^{th} result element.

[8] Similar means there are only insignificant differences, e.g. concerning the typing system.

Note that solution modifiers are always applied **after** the result has been computed, i.e. their evaluation requires a postprocessing of the query evaluation result. Hence, they are not part of the algebraic evaluation process, but rather constitute operations that are carried out separately from the core evaluation phase.[9] We intentionally omit these modifiers in our formalization, whose goal was to identify a compact fragment that covers the basic, core evaluation process of SPARQL. When discussing algebraic SPARQL query optimization later in this thesis, we will come back to the DISTINCT and REDUCED query forms (cf. Section 4.4.2).

Query Forms. We mentioned before in Section 2.3.1 that the official SPARQL standard introduces four different types of queries, so-called *query forms*. In the following listing, we discuss these four query forms informally.

- SELECT queries compute all possible variable mappings; they constitute the de facto standard query form and have been formally introduced in Definition 2.5.
- SPARQL ASK queries are boolean queries that return *yes/no* (in our formalization *true/false*, respectively). We introduced them in Definition 2.6.
- The DESCRIBE query form extracts additional information related to the result mappings and returns a description in form of a new RDF graph. The SPARQL specification provides only a vague definition for the semantics of DESCRIBE queries, stating that "the DESCRIBE form takes each of the resources identified in a solution [...] and assembles a single RDF graph by taking a 'description' which can come from any information available including the target RDF dataset. The description is determined by the query service." [spac]
- SPARQL CONSTRUCT queries transform the result mapping into a new RDF graph, according to rules that can be specified in the CONSTRUCT clause.

Like for the solution modifiers, we observe that the query form implementation for DESCRIBE and CONSTRUCT is independent from the core evaluation process, but again requires a postprocessing of the extracted result. Therefore, these two query forms are not integrated in our SPARQL formalization. Instead, we focus on the SELECT query form, which can be seen as the natural counterpart of standard SQL queries. In some cases, we will also investigate ASK queries, which are particularly interesting from an optimization perspective of view (see e.g. Section 4.4.1).

Additional Remarks. There are some more implementation-specific features that should be listed for completeness. First, our definition of filter conditions (see Definition 2.3) does not cover the full W3C standard. The latter comprises a broader set of functions, such as $isURI(?x)$, $isLiteral(?x)$, or $isBlank(?x)$ (used to check if variable $?x$ is bound to elements of the respective set), and supports all the standard relational operators (i.e., $<$, $\leq$, $=$, $\neq$, $\geq$, $>$). Complementarily, the W3C semantics includes a type system which is built on top of RDF datatypes (cf. Section 2.2)

[9] Though, in practical scenarios it could make sense to integrate them into the core evaluation process, to improve system performance.

and comes into play when evaluating relational operators, e.g. operator "<" is interpreted as numerical comparison when comparing integer-typed literals, but implements a lexicographical comparison for string values. Arguably, such issues are implementation-specific and therefore were abstracted away in our formalization.

Another difference is that the official SPARQL specification uses an extended version of triple patterns (cf. Definition 2.4), where blank nodes may be used in the subject and object position of triples. In contrast to URIs and literals, these blank nodes are not interpreted as identifying nodes that match same-named blank nodes in the RDF graph, but instead can be understood as variables that match arbitrary nodes and whose bindings do not appear in the final result. Abstracting from the details (i.e., scoping issues), they can simply be replaced by variables that are projected away in the SELECT clause, without changing the semantics of the query. In particular, they do not add expressiveness, so we decided to ignore them in our formalization, to keep the semantics as simple as possible.

Another interesting feature of SPARQL is the possibility to query multiple RDF graphs at a time. This feature accounts for the global nature of the RDF data format, which was designed to link information from different sources together. In particular, SPARQL provides keywords to specify the default graph and a set of so-called named graphs to be accessed during query evaluation. In the body of the SPARQL query, one can associate graph patterns with RDF graphs, which ultimately allows to combine information from different graphs.

Finally, we should note that SPARQL comes with a protocol, which allows to convey SPARQL queries from a client to a SPARQL query processor. This protocol empowers the intended use of SPARQL as a query language in the Semantic Web. We refer the interested reader to [spab] for the official protocol specification.

2.3.5. Limitations of SPARQL

Given that SPARQL is a comparably young technology, the current W3C specification [spac] still has a couple of limitations, which become obvious when comparing SPARQL to established query languages such as SQL or XQuery. The following list surveys important features and constructs that are (to date) missing in SPARQL.

- **Aggregation**: The current specification does not support aggregation functions, such as summing up numeric values, counting, or average computation.
- **Updates**: While the SPARQL standard supports data extraction from RDF graphs, constructs for inserting new triples into RDF graphs and manipulating existing graphs (in the style of SQL INSERT and UPDATE clauses) are missing.
- **Path Expressions**: SPARQL does not support the specification of (constrained) path expressions, e.g. using a single SPARQL query it is impossible to compute the transitive closure of a graph or to extract all nodes that

are reachable from a fixed node. This deficiency has repeatedly been identified in previous publications and different proposals for incorporating path expressions into the language have been made [KJ07; PAG08; ABE09]. The interested reader will find a short discussion of these approaches in Section 2.4.

- **Views:** In traditional query languages like SQL, logical views over the data play an important role. They are crucial to both database design and access management. SPARQL currently does not support the specification of logical views over the data; note, however, that materialized views over the data can be extracted from the input graph using the CONSTRUCT query form.
- **Support for Constraints:** Mechanism to assert and check for integrity constraints in the RDF database are not covered in the current SPARQL specification. In SQL, such integrity constraints are implicitly derived from primary and foreign key specifications established in the schema design phase. Beyond that, it is possible to enforce user-defined constraints using the CREATE ASSERTION statement. Tackling the issue of integrity constraints, we will investigate general capabilities of SPARQL as a constraint language in Section 5.3.

We conclude with the remark that the W3C SPARQL working group is currently working on a new version of SPARQL [spaa] (there is also a Wiki page that contains proposals for future SPARQL extensions[10]). Based on these documents, it can be expected that at least some of the above features will be part of the coming release.

2.4. Related Work

Formalization of RDF(S) and Semantics. The reasoning mechanisms implemented in RDF(S) have their foundations in early logic-based languages for object-oriented data, such as F-logic [KLW95] and description logics [BCM+03]. An early model theoretic formalization of RDF(S) semantics was provided in [Mar04]. Going one step further, foundations and advanced aspects of RDFS databases were studied in [GHM04]. The latter work covers issues like RDFS reasoning and presents a set of inference rules that implement the core of the model theoretic W3C RDF semantics definition [rdfd]. In addition, it studies problems like complexity of entailment (i.e., the question if an RDF(S) graph logically implies another graph), normal forms for RDF(S), and query answering on top of implied RDF(S) databases. Similar in style, a datalog-style query language for RDF, called RDFLog, was proposed in [BFL+08].

In [MPG07] a minimal deductive system for RDFS is presented. The authors identify a fragment of RDFS, called ρdf and show that entailment can be decided efficiently this fragment. Although RDFS reasoning is not a central topic in this thesis, this work gives valuable insights into the RDFS reasoning process.

[10] See `http://esw.w3.org/topic/SPARQL/Extensions`.

Formalization of SPARQL. In the early W3C Working Drafts for SPARQL, the semantics of SPARQL evaluation was primarily defined through test cases and declaration of the expected query results.[11] At that time, several proposals to defining its semantics were made by the research community. One notable line of research in this context is the specification of mappings from SPARQL to relational algebra or SQL, which implicitly fix the semantics of SPARQL [Cyg05; CLJF06]. An early (yet unfinished) approach to an algebraic formalization of SPARQL was given in [FT05]. In 2006 then, the publication [PAG06a] (which was further refined in [PAG06b]) presented two different set-based semantics for a core fragment of the SPARQL query language, both building upon a compact algebra over mapping sets. The first semantics is compositional and differs from the second, operational semantics in some exceptional cases (such as the issue of evaluating nested OPT expressions). Inspired by the latter work, the W3C started to work out a formal semantics for the full SPARQL query language. On March 26, 2007, it released an improved version of previous Working Drafts, which included a formal SPARQL semantics that closely followed the compositional approach presented in [PAG06a; PAG06b].[12] As a difference to the set-based semantics in [PAG06a], though, the W3C proposed a bag semantics. This bag semantics was carried over throughout the standardization process and has found entrance in the current W3C SPARQL Recommendation [spac]. Note that, as indicated in Section 2.3, our formalization of SPARQL also follows the compositional approach from [PAG06a]. Consequently, the bag semantics that we formalized in Definition 2.15 mirrors the current W3C semantics.

Extensions of SPARQL. We recently studied the perspectives of SPARQL as a constraint language in [LMS08]. The work shows that SPARQL – with some minor extensions – can be used to express a large class of constraints and to extract constraints from RDF graphs when these are specified using a predefined vocabulary for encoding constraints. We will review and extend some of these ideas in Section 5.3.

Aggregation functions for SPARQL were proposed in [PSS07]. The latter work defines an extension of SPARQL, called SPARQL++, which embeds standard aggregate functions in CONSTRUCT and FILTER clauses. The motivation for this extension was to express schema mappings through SPARQL CONSTRUCT queries. It is also worth mentioning that some existing SPARQL engines, e.g. ARQ [arq] and Virtuoso [Bla07], already have implemented their own strategies to aggregation.

Path expressions for SPARQL have been identified as an important feature in several research contributions [KJ07; PAG08; ABE09]. The idea that is common to all these approaches is to extend SPARQL by constructs that allow to express relations between nodes that go beyond what can be expressed by simple basic graph patterns, such as e.g. transitively connected nodes. It is natural to assume that

[11] See for instance the W3C Working Draft for the SPARQL Query Language from October 4, 2006 at `http://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004/`.

[12] See `http://www.w3.org/TR/2007/WD-rdf-sparql-query-20070326/`.

querying for (constrained) paths is an important feature in the context of a graph-structured data model like RDF. The approach in [KJ07] uses so-called regular path patterns, akin to regular expressions, to express complex path relationships between nodes in RDF graphs. These regular path patterns are used to extend SPARQL to a dialect called SPARQLeR. In [PAG08] a SPARQL extension called nSPARQL is proposed, driven by the idea of navigating through the RDF graph using a set of predefined axes, very much in the style of the XPath axes for navigating through XML documents. Another reasonable approach is the PSPARQL [ABE09] query language. It relies on an extended version of RDF, called PRDF, where graph edges (i.e., predicates in RDF triples) may carry regular expression patterns as labels. The PSPARQL query language is then defined over such PRDF patterns.

Chapter 3.

Complexity of SPARQL Evaluation

Jen: "*How to get started, guys?*"
Roy: "*Well, first we should try to understand the basics of SPARQL.*"
Jen: "*Oh, I'm afraid I know what you're talking about...*"
Moss: "*... the complexity of SPARQL evaluation, right?*"
Roy: "*Yes, exactly. Don't panic, I'm pretty sure it's gonna be fun!*"

In this chapter we study the complexity of SPARQL query evaluation, with the goal to establish a deep understanding of the operators, their complexity, their interaction, and their expressiveness. As is customary in the context of query languages, we take the decision version of the Evaluation problem as a yardstick: given a mapping μ, an RDF database D, and a SPARQL expression or query Q as input, we are interested in the complexity of deciding whether μ is contained in the result of evaluating Q on D. We study this problem for different fragments of the SPARQL query language, to separate operators and operator constellations that can be evaluated efficiently from more complex (and hence, more expressive) constellations.

In the problem statement above, we left the semantics used for our study of the Evaluation problem unspecified. In fact, we are interested in the complexity of SPARQL query evaluation for both the set semantics introduced in Section 2.3.2 and the bag semantics from Section 2.3.3. As an important result, however, we will show in Section 3.2.1 that the semantics do not differ w.r.t. their evaluation complexity. This observation allows us to restrict our discussion to the (arguably simpler) set semantics, while all complexity results immediately carry over to the bag semantics and, more importantly, also apply to the official W3C SPARQL semantics.

It should be noted that, according to our definition of the Evaluation problem, we investigate the *combined complexity* of SPARQL evaluation, where both the database and the query are part of the input. The study of combined complexity is particularly useful to understand the basic expressive power of the operators and their interrelations, which is in line with our goal to investigate general capabilities and expressiveness of SPARQL and fragments of the query language. We refer the interested reader to [PAG06a] for results on so-called *data complexity* [Var82], which differs in that the size of the query is fixed (i.e. the query is not part of the input).

Coming back to the motivation for the work in this chapter, we argue that a comprehensive complexity analysis is important for several reasons. As argued before, one important aspect is that the investigation of different SPARQL fragments separates subsets of the language that can be evaluated efficiently from complex fragments. The knowledge of the evaluation complexity may be of immediate practical interest when building applications that use (fragments of) SPARQL. For instance, upper complexity bounds implicitly restrict the expressiveness and show that certain problems or task cannot be solved with the fragment under consideration. The evaluation problem for SPARQL expressions built using only operators And, Filter, and triple patterns, for example, is known to be in PTime (cf. [PAG06a]), so under the common assumption that PTime $\neq$ NP one cannot encode NP-hard problems using only these two operators; yet, as we will show later in Section 3.2.5, when extending the latter fragment by projection the evaluation problem becomes NP-complete and one gains the power to model NP-hard problems.

Last but not least, our complexity study relates the SPARQL query language and specific fragments to traditional query languages. To give a concrete example, there has been a corpus of research on conjunctive queries (see [AHV] for a survey of important results) and it is well-known that the evaluation problem for such queries is NP-complete [CM77]. In our complexity study, we will identify SPARQL operator combinations for which the evaluation problem falls into the same complexity class, so the problem of evaluating these queries and the problem of conjunctive query evaluation are mutually reducible. This close connection makes it possible to transfer known results and optimization techniques from the area of conjunctive query optimization into the context of SPARQL. For instance, we will exploit translations from NP-complete SPARQL fragments to conjunctive queries in Chapter 5, which allows us to exploit the chase procedure [MMS79; BV84], originally designed for conjunctive query rewriting under data dependencies, in the context of SPARQL.

We conclude our introduction with the remark that the analysis of SPARQL complexity is not new: the preliminary investigation of the combined complexity of SPARQL in [PAG06a] shows that the evaluation problem for full SPARQL expressions (i.e., the fragment introduced in Definition 2.4) is PSpace-complete. Further, [PAG06a] provides selected results for fragments of the language with lower complexity (we will summarize them in Section 3.2). As a consequent enhancement of this initial analysis, we systematically explore the complexity of **all** expression and query fragments, where fragment means a class of expressions or queries that can be built using a fixed subset of the SPARQL operators. Our exhaustive study gives us valuable insights that go beyond the initial findings presented in [PAG06a]. One central result is that the Evaluation problem for operator Opt alone (i.e., SPARQL expressions built using only Opt and triple patterns) is already PSpace-hard. We further show that this high complexity is caused by an unlimited nesting of Opt expressions. In practice, however, it is reasonable to assume that the nesting depth is fixed, and for this case we prove lower complexity bounds. Still, as a key insight,

operator Opt is by far the most complicated construct in SPARQL. This observation suggests that special care in query optimization should be taken in queries containing operator Opt and will serve as a guideline for our study of SPARQL optimization and benchmarking in subsequent chapters of this thesis.

The main contributions and findings of this section can be summarized as follows.

- We show that the complexity of the Evaluation problem does not change when switching from set to bag semantics and vice versa. This allows us to focus on the simpler set semantics, while all results immediately carry over to the bag semantics and hence to the official W3C SPARQL semantics.
- The main source of complexity in Opt-free expression fragments is the combination of operators And and Union. More precisely, these two operator combinations make the Evaluation problem NP-complete. The Filter operator in no case affects the complexity of the Evaluation problem.
- In [PAG06a] it was shown that the evaluation problem for SPARQL expressions (i.e. expressions built using And, Union, Opt, Filter, and triple patterns) is PSpace-complete. We considerably refine this analysis and show that **all** fragments involving operator Opt are PSpace-complete. In particular, this result already holds for queries built using only operator Opt, which reveals that Opt is by far the most complex operator in the SPARQL query language.
- Motivated by the previous finding, we present a syntactic restriction for expressions involving operator Opt that lowers the complexity of the evaluation problem: when fixing the nesting depth of Opt subexpressions, we obtain tight complexity bounds in the polynomial hierarchy. Spoken the other way around, this result clarifies that only the unrestricted nesting of Opt expressions is responsible for the PSpace-completeness of the query language.
- Going one step further, we also study the complexity of SPARQL queries, obtained from expressions by adding top-level projection in the form of a Select-clause (cf. Definition 2.5). We show that projection does not increase the evaluation complexity for NP- and PSpace-complete fragments (hence, the complexity of fragments including Opt remains the same), but raises the complexity from PTime to NP for And-only expressions. The latter result reveals the close connection between And-only queries and conjunctive queries.

Structure. We will shortly revisit relevant concepts and definitions from complexity theory in Section 3.1, before presenting an elaborate complexity analysis for the SPARQL query language in Section 3.2. First, in Section 3.2.1, we show that the evaluation problem has the same complexity for both set and bag semantics. Subsequently, we investigate the complexity of Opt-free expression fragments (Section 3.2.2), expression fragments involving operator Opt (Sections 3.2.3 and 3.2.4), and SPARQL queries (Section 3.2.5). For the convenience of the reader, we summarize all gathered complexity results in Section 3.2.6. We finally wrap up with a discussion of related work in Section 3.3 and a short conclusion in Section 3.4.

3.1. Preliminaries: Complexity Theory

Before starting our discussion of SPARQL complexity we introduce some background and conventions from complexity theory. As usual, we denote by PTime the complexity class comprising decision problems that can be decided by a deterministic Turing Machine (TM) in polynomial time, by NP the class of problems that can be decided by a non-deterministic TM in polynomial time, and by PSpace the class of problems that can be decided by a deterministic TM within polynomial space bounds. It is common knowledge that the inclusion hierarchy

$$\text{PTime} \subseteq \text{NP} \subseteq \text{PSpace} \tag{3.1}$$

holds, and it is usually conjectured that both inclusion relations are strict.

3.1.1. The Polynomial Hierarchy

Given a complexity class C we denote by coC the set of decision problems whose complement can be decided by a TM in class C. Given complexity classes C_1 and C_2, the class $\text{C}_1^{\text{C}_2}$ captures all problems that can be decided by a TM M_1 in class C_1 enhanced by an oracle machine M_2 for solving problems in C_2. Informally speaking, M_1 may consult M_2 to obtain a *yes*/*no*-answer for a problem in C_2 in a single step. We refer the interested reader to [AB07] for a formal discussion of oracle machines.

Given the previous notation, we are now in the position to define the *polynomial hierarchy*, a sequence of complexity classes that was introduced in [Sto76]:

Definition 3.1 (Polynomial Hierarchy [Sto76]) The *polynomial hierarchy* is the sequence of classes Σ_i^P and Π_i^P for $i \in \mathbb{N}_0$, inductively defined as

$$\Sigma_0^P = \Pi_0^P := \text{PTime}, \quad \Sigma_{n+1}^P := \text{NP}^{\Sigma_n^P}, \text{ and } \quad \Pi_{n+1}^P := \text{coNP}^{\Sigma_n^P}. \qquad \square$$

Note that, by definition, $\Sigma_i^P = co\Pi_i^P$ holds. Further, it is known that $\Sigma_i^P \subseteq \Pi_{i+1}^P$ and $\Pi_i^P \subseteq \Sigma_{i+1}^P$. In addition, the following two inclusion hierarchies are known.

$$\begin{aligned} \text{PTime} = \Sigma_0^P \subseteq \text{NP} = \Sigma_1^P \subseteq \Sigma_2^P \subseteq \Sigma_3^P \subseteq \cdots \subseteq \text{PSpace} \\ \text{PTime} = \Pi_0^P \subseteq \text{coNP} = \Pi_1^P \subseteq \Pi_2^P \subseteq \Pi_3^P \subseteq \cdots \subseteq \text{PSpace} \end{aligned} \tag{3.2}$$

3.1.2. Complete Problems

We consider completeness only with respect to polynomial-time many-one reductions. QBF, the tautology test for quantified boolean formulas, is known to be PSpace-complete [AB07]. Variants of QBF with restricted quantifier alternation

are complete for classes Π_i^P or Σ_i^P, depending on the number i of quantifier alternations and the question whether the first quantifier is $\forall$ or $\exists$, respectively (cf. [Pap94; AB07]). Finally, the NP-completeness of the SetCover problem and the 3Sat problem is folklore (see e.g. [Pap94] for a study of these two problems). We will provide a precise definition of all problems before using them in reductions.

3.2. Complexity of SPARQL

We introduce the SPARQL operator shortcuts $\mathcal{A}$:= And, $\mathcal{F}$:= Filter, $\mathcal{O}$:= Opt, and $\mathcal{U}$:= Union. For notational convenience, we denote the class of SPARQL expressions that can be constructed using a set of operators (plus triple patterns) by concatenating the respective operator shortcuts. To give an example, the class $\mathcal{AU}$ comprises all SPARQL expressions that can be constructed using only operators And, Union, and triple patterns. By $\mathcal{E}$ we denote the full class of SPARQL expressions (according to Definition 2.4), i.e. we define $\mathcal{E} := \mathcal{AFOU}$. In the remainder of this chapter, we will use the terms *class* and *fragment* interchangeably.

In the subsequent complexity study, we follow the approach from [PAG06a] and take the complexity of the Evaluation problem as a reference:

Evaluation: given a mapping μ, a document D, and a SPARQL expression or a SPARQL query Q as input: is $\mu \in [\![Q]\!]_D$?

The following theorem summarizes all previous results on the combined complexity of SPARQL fragments established in [PAG06a], rephrased in our notation. We refer the interested reader to the original work for the proofs of these results.

Theorem 3.1 (see [PAG06a]) The Evaluation problem is

1. in PTime for class $\mathcal{AF}$ (membership in PTime for $\mathcal{A}$ and $\mathcal{F}$ follows directly),
2. NP-complete for class $\mathcal{AFU}$, and
3. PSpace-complete for classes $\mathcal{AOU}$ and $\mathcal{E}$. □

The theorem (and hence, the study in [PAG06a]) leaves several questions about the complexity of SPARQL unanswered. First, all results in Theorem 3.1 have been established for the set semantics $[\![.]\!]_D$, so it is not immediately clear if these results carry over to the official W3C SPARQL specification, which uses a bag semantics. Second, concerning Theorem 3.1(2) it is an open question if the NP-completeness stems from the combination of all three operators And, Filter, Union or if it is already obtained when combining either And or Filter with Union. Third, the theorem does not identify the minimal operator constellation that causes PSpace-hardness, i.e. there are no results for subclasses of $\mathcal{AOU}$ that include operator Opt.

Finally, the results are restricted to SPARQL expressions and do not cover the complexity of queries, which have an additional SELECT operator on top. In the official W3C specification, however, the SELECT-clause is mandatory, so these fragments are of immediate practical interest. We will now systematically explore these issues.

3.2.1. Set vs. Bag Semantics

The definition of the EVALUATION problem in Section 3.2 relies on set semantics for query evaluation. Our first task is to show that all complexity results obtained for set semantics immediately carry over to bag semantics. The corresponding problem for bag semantics, denoted by EVALUATION$^+$, can be phrased as follows.

EVALUATION$^+$: given a mapping μ, document D, and SPARQL expression or SPARQL query Q as input: let $[\![Q]\!]_D^+ := (\Omega^+, m^+)$, is $\mu \in \Omega^+$?[1]

Note that we do not explicitly enforce that $m^+(\mu) \geq 1$, because this condition is implicitly given by our definition of multi-sets (cf. Definition 2.12), i.e. whenever a mapping appears in Ω^+, then its multiplicity is at least one. The following lemma shows that the bag semantics differs from the set semantics at most in the multiplicity associated to each mapping (which might be greater than one).

Lemma 3.1 Let Q be a SPARQL query or expression, D be an RDF database, and μ be a mapping. Let $\Omega := [\![Q]\!]_D$ and $(\Omega^+, m^+) := [\![Q]\!]_D^+$. Then $\mu \in \Omega \Leftrightarrow \mu \in \Omega^+$. □

Proof of Lemma 3.1

We prove the lemma by induction on the structure of Q. To simplify the notation, we shall write $\mu \in [\![Q]\!]_D^+$ if and only if $\mu \in \Omega^+$ for $[\![Q]\!]_D^+ := (\Omega^+, m^+)$. Intuitively, this short notation is justified by the property that $m^+(\mu) \geq 1$ for each $\mu \in \Omega^+$. It is crucial to note that the SPARQL bag algebra operators introduced in Definition 2.10 maintain this property, i.e. whenever an algebraic operation generates a mapping μ, then the multiplicity that is associated with μ is at least one.

The induction hypothesis for the proof of Lemma 3.1 is $\mu \in [\![Q]\!]_D \Leftrightarrow \mu \in [\![Q]\!]_D^+$. For the basic case, let us assume that $Q := t$ is a triple pattern. Let $\Omega := [\![t]\!]_D$ and $(\Omega^+, m^+) := [\![t]\!]_D^+$ be the results obtained when evaluating Q on D using set and bag semantics, respectively. From Definitions 2.10 and 2.14 it follows immediately that $\Omega = \Omega^+$, so it trivially holds that $\mu \in [\![Q]\!]_D \Leftrightarrow \mu \in [\![Q]\!]_D^+$, which completes the basic case. We therefore may assume that the hypothesis holds for every expression.

Coming to the induction step, we distinguish five cases. (1) Let $Q := P_1$ AND P_2. $\Rightarrow$: Let $\mu \in [\![P_1 \text{ AND } P_2]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$. Then, by definition of operator $\bowtie$, there are $\mu_1 \in [\![P_1]\!]_D$, $\mu_2 \in [\![P_2]\!]_D$ s.t. $\mu_1 \sim \mu_2$ and $\mu_1 \cup \mu_2 = \mu$. By application of the

[1] An alternative version of the evaluation problem under bag semantics encountered in literature is to ask whether $\mu \in \Omega$ and $m(\mu) = c$ for some c. Here, we disregard the multiplicity of μ.

induction hypothesis, we have that $\mu_1 \in [\![P_1]\!]_D^+$ and $\mu_2 \in [\![P_2]\!]_D^+$, and consequently $\mu = \mu_1 \cup \mu_2 \in [\![P_1]\!]_D^+ \bowtie [\![P_2]\!]_D^+ = [\![P_1 \text{ AND } P_2]\!]_D^+$. Direction "$\Leftarrow$" is analogical. We omit the proof for case (2) $Q := P_1$ UNION P_2, which is similar to case (1). Next, (3) let $Q := P_1$ OPT P_2. We exemplarily discuss direction "$\Rightarrow$", the opposite direction is similar. Let $\mu \in [\![P_1 \text{ OPT } P_2]\!]_D = ([\![P_1]\!]_D \bowtie [\![P_2]\!]_D) \cup ([\![P_1]\!]_D \setminus [\![P_2]\!]_D)$. Then μ is generated (i) by the subexpression $[\![P_1]\!]_D \bowtie [\![P_2]\!]_D$ or (ii) by $[\![P_1]\!]_D \setminus [\![P_2]\!]_D$. The argumentation for (i) is identical to case (1), i.e. we can show that μ is then generated by $[\![P_1 \text{ OPT } P_2]\!]_D^+ = ([\![P_1]\!]_D^+ \bowtie [\![P_2]\!]_D^+) \cup ([\![P_1]\!]_D^+ \setminus [\![P_2]\!]_D^+)$, namely by the left side of the union. For case (ii), we argue that $\mu \in [\![P_1]\!]_D \setminus [\![P_2]\!]_D$ implies $\mu \in [\![P_1]\!]_D^+ \setminus [\![P_2]\!]_D^+$. So let us assume that $\mu \in [\![P_1]\!]_D \setminus [\![P_2]\!]_D$. Then $\mu \in [\![P_1]\!]_D$ and there is no compatible mapping $\mu' \sim \mu$ in $[\![P_2]\!]_D$. We have $\mu \in [\![P_1]\!]_D^+$ by induction hypothesis. Assume for the sake of contradiction that there is a compatible mapping $\mu' \sim \mu$ in $[\![P_2]\!]_D^+$. Then, again by induction hypothesis, we have that $\mu' \in [\![P_2]\!]_D$, which contradicts to the assumption that there is no compatible mapping to μ in $[\![P_2]\!]_D$. This completes case (3). Finally, cases (4) $Q := \text{SELECT}_S(P)$ and (5) $Q := P$ FILTER R are easily obtained by application of the induction hypothesis. □

It follows as a corollary from the lemma above that the set and bag semantics do not differ w.r.t. the complexity of the SPARQL EVALUATION problem:

Corollary 3.1 Let μ be a mapping, D be an RDF document, and Q be a SPARQL expression or query. Then $\text{EVALUATION}(\mu, D, Q) \Leftrightarrow \text{EVALUATION}^+(\mu, D, Q)$. □

This result allows us to use the simpler set semantics for our study of SPARQL complexity, while all results immediately carry over to bag semantics. We point out that the W3C SPARQL Recommendation relies on a bag semantics very close to ours in Definition 2.15 (cf. the discussion in Section 2.3.4) and therefore all complexity results that we derive in this chapter also apply to the official W3C semantics.

3.2.2. Complexity of OPTIONAL-free Expressions

Our first goal is to establish a precise characterization of the UNION operator. As also noted in [PAG06a], the design of this operator was subject to controversy in the SPARQL working group.[2] In response, we aim to understand the operator and its relation to others beyond the known NP-completeness result for class $\mathcal{AFU}$. The next theorem gives the results for all OPT-free fragments not covered by Theorem 3.1.

Theorem 3.2 The EVALUATION problem is

1. in PTIME for classes $\mathcal{U}$ and $\mathcal{FU}$, and
2. NP-complete for class $\mathcal{AU}$. □

[2]For further details on this issue see the discussion of disjunction in Section 6.1 in the W3C mailing archive `http://www.w3.org/TR/2005/WD-rdf-sparql-query-20050217/`.

Proof of Theorem 3.2

Theorem 3.2(1): We provide a PTime-algorithm that solves the Evaluation problem for fragment $\mathcal{FU}$. It is defined inductively on the structure of the input expression P and returns *true* if $\mu \in [\![P]\!]_D$, *false* otherwise. We distinguish three cases. (a) If $P := t$ is a triple pattern, we return *true* if and only if $\mu \in [\![t]\!]_D$. (b) If $P := P_1$ Union P_2 we (recursively) check if $\mu \in [\![P_1]\!]_D \vee \mu \in [\![P_2]\!]_D$ holds. (c) If $P := P_1$ Filter R for some filter condition R, we return *true* if and only if $\mu \in [\![P_1]\!]_D \wedge R \models \mu$. It is easy to see that the above algorithm runs in polynomial time. Its correctness follows from the definition of the algebraic operators $\cup$ and σ.

Theorem 3.2(2): In order to prove that the Evaluation problem for fragment $\mathcal{AU}$ is NP-complete we have to show NP-membership and NP-hardness.

(Membership) Let P be a SPARQL expression composed of operators And, Union, and triple patterns, D a document, and μ a mapping. We provide an NP-algorithm that returns *true* if $\mu \in [\![P]\!]_D$, and *false* otherwise. Our algorithm is defined on the structure of P: (a) if $P := t$ is a triple pattern then return *true* if $\mu \in [\![t]\!]_D$, *false* otherwise; (b) if $P := P_1$ Union P_2, return the truth value of $\mu \in [\![P_1]\!]_D \vee \mu \in [\![P_2]\!]_D$; finally, (c) if $P := P_1$ And P_2, then guess a decomposition $\mu = \mu_1 \cup \mu_2$ and return the truth value of $\mu_1 \in [\![P_1]\!]_D \wedge \mu_2 \in [\![P_2]\!]_D$. The correctness of the algorithm follows from the definition of the algebraic operators $\bowtie$ and $\cup$. It is easy to see that it can be implemented by a non-deterministic TM that runs in polynomial time.

(Hardness) We reduce the SetCover problem to the Evaluation problem for SPARQL. SetCover is known to be NP-complete, so the reduction gives us the desired hardness result. The SetCover problem is defined as follows.

SetCover: Let $U := \{u_1, \ldots, u_k\}$ be a universe, $S_1, \ldots S_n \subseteq U$ be sets over U, and let l be positive integer: is there a set $I \subseteq \{1, \ldots, n\}$ of size $\lvert I \rvert \leq l$ such that $\bigcup_{i \in I} S_i = U$?

We use the fixed database $D := \{(c, c, c)\}$ for our encoding and represent each set $S_i := \{x_1, x_2, \ldots, x_m\}$ by a SPARQL expression of the form

$$P_{S_i} := (c, c, ?X_1) \text{ And } \ldots \text{ And } (c, c, ?X_m).$$

Next, we define the expression $P_S := P_{S_1}$ Union $\ldots$ Union P_{S_n} as an encoding for the set $S = \{S_1, \ldots, S_n\}$ of all S_i. Finally we define the SPARQL expression

$$P := \underbrace{P_S \text{ And } \ldots \text{ And } P_S}_{P_S \text{ appears exactly l times}}.$$

The intuition of the encoding is as follows. P_S encodes all subsets S_i. A set element, say x, is represented by the presence of a binding from variable $?X$ to value c. The encoding of P allows us to "merge" (at most) l arbitrary sets S_i. It is straightforward

to show that SetCover is true iff $\mu := \{?U_1 \mapsto c, \ldots, ?U_k \mapsto c\} \in [\![P]\!]_D$, i.e. if the complete universe U can be obtained by merging these sets. □

Theorems 3.1 and 3.2 taken together clarify that the source of complexity in Opt-free fragments is the combination of And and Union. In particular, adding or removing Filter expressions in no case increases or decreases the complexity.

3.2.3. Complexity of Expression Classes Including OPTIONAL

We next investigate the complexity of operator Opt and its interaction with other operators. The PSpace-completeness results for classes $\mathcal{AOU}$ and $\mathcal{E} := \mathcal{AFOU}$ in Theorem 3.1 give only partial answers to these questions. The following theorem refines the above results, showing that already class $\mathcal{AO}$ is PSpace-complete:

Theorem 3.3 Evaluation is PSpace-complete for class $\mathcal{AO}$. □

Proof of Theorem 3.3

We reduce QBF, a prototypical PSpace-complete problem, to the SPARQL Evaluation problem for class $\mathcal{AO}$. The hardness part of the proof is in parts inspired by the proof of Theorem 3.1(3), which has been formally proven in [PAG06a]: there, QBF was encoded using operators And, Opt, and Union. Here, we encode the problem using only And and Opt, which turns out to be considerably harder. More precisely, our contribution is to show how to encode a **quantifier-free** boolean formula using And and Opt, while the encoding of the surrounding quantifier sequence remains the same. Membership in PSpace, and hence PSpace-completeness, then follows directly from the PSpace-membership of the fragment $\mathcal{AOU} \supset \mathcal{AO}$ (cf. Theorem 3.1(3)). Formally, QBF is defined as follows.[3]

> QBF: given a quantified boolean formula $\varphi := \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_m \exists y_m \psi$ as input, where ψ is a quantifier-free formula in conjunctive normal form (CNF): is the formula φ valid?

Let us start the discussion with a quantified boolean formula

$$\varphi := \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_m \exists y_m \psi$$

and assume that the inner formula ψ of the quantified formula is in conjunctive normal form, i.e. $\psi := C_1 \wedge \cdots \wedge C_n$ where the C_i ($i \in [n]$) are disjunctions of literals[4]. By V_ψ we denote the set of (boolean) variables in ψ and by V_{C_i} the set of variables in clause C_i. For our encoding, we use the polynomial-size database

[3] Like the proof in [PAG06a], we assume that the inner formula of the quantified formula is in CNF. It is known that also this variant of the QBF problem is PSpace-complete.

[4] A literal is either a boolean variable x or a negated boolean variable $\neg x$.

$$D := \{(a, \mathit{false}, 0), (a, \mathit{true}, 1), (a, \mathit{tv}, 0), (a, \mathit{tv}, 1)\} \cup \{(a, \mathit{var}_i, v) \mid v \in V_{C_i}\} \cup \{(a, v, v) \mid v \in V_\psi\},$$

where the second and the third part of the union set up triples for the variables in each C_i and V_ψ, respectively. For instance, if $V_{C_1} = V_\psi = \{x\}$, the second and the third part of the union would generate the triples (a, var_1, x) and (a, x, x), respectively, where x is understood as a URI representing the boolean variable x.

For each clause $C_i := v_1 \vee \cdots \vee v_j \vee \neg v_{j+1} \vee \cdots \vee \neg v_k$, where $v_1, \ldots, v_j$ are positive and $v_{j+1}, \ldots, v_k$ are negated variables, we define a separate SPARQL expression

$$\begin{aligned} P_{C_i} := (\ldots((\ldots((a, \mathit{var}_i, ?\mathit{var}_i) \\ &\textsc{Opt}\ ((a, v_1, ?\mathit{var}_i)\ \textsc{And}\ (a, \mathit{true}, ?V_1))) \\ &\ldots \\ &\quad \textsc{Opt}\ ((a, v_j, ?\mathit{var}_i)\ \textsc{And}\ (a, \mathit{true}, ?V_j))) \\ &\quad\ \textsc{Opt}\ ((a, v_{j+1}, ?\mathit{var}_i)\ \textsc{And}\ (a, \mathit{false}, ?V_{j+1}))) \\ &\quad\quad \ldots \\ &\quad\quad\ \textsc{Opt}\ ((a, v_k, ?\mathit{var}_i)\ \textsc{And}\ (a, \mathit{false}, ?V_k))), \end{aligned}$$

where $v_1, \ldots, v_k$ stand for the URIs that are associated with the respective variables according to D. We then encode formula ψ as $P_\psi := P_{C_1}\ \textsc{And}\ \ldots\ \textsc{And}\ P_{C_n}$.

It is straightforward to verify that ψ is satisfiable iff there is a mapping $\mu \in [\![P_\psi]\!]_D$. Even more, each mapping $\mu \in [\![P_\psi]\!]_D$ represents a set of truth assignments, where each assignment ρ_μ is obtained as follows: for each $v_i \in V_\psi$ we set $\rho_\mu(v_i) := \mu(?V_i)$ if $?V_i \in \mathit{dom}(\mu)$, or define either $\rho_\mu(v_i) := 0$ or $\rho_\mu(v_i) := 1$ if $?V_i \notin \mathit{dom}(\mu)$; vice versa, for each truth assignment ρ that satisfies ψ there is $\mu \in [\![P_\psi]\!]_D$ that defines ρ according to the construction rule for ρ_μ above. Note that the definition of ρ_μ accounts for the fact that some $?V_i$ may be unbound in μ; in such a case, the value of the variable is not relevant to obtain a satisfying truth assignment and we can randomly choose a value for the corresponding boolean variable v_i.

Given P_ψ, we can encode the quantifier-sequence using a series of nested Opt statements as shown in [PAG06a]. To make the proof self-contained, we shortly summarize this construction. We use SPARQL variables $?X_1, \ldots, ?X_m$ and $?Y_1, \ldots, ?Y_m$ to represent variables $x_1, \ldots x_m$ and $y_1, \ldots, y_m$, respectively. In addition, we use fresh variables $?A_0, \ldots ?A_m$, $?B_0, \ldots ?B_m$, and operators And, Opt to encode the quantifier sequence $\forall x_1 \exists y_1 \ldots \forall x_m \exists y_m$. For each $i \in [m]$ we define P_i and Q_i as

$$\begin{aligned} P_i &:= ((a, \mathit{tv}, ?X_1)\ \textsc{And}\ \ldots\ \textsc{And}\ (a, \mathit{tv}, ?X_i)\ \textsc{And} \\ &\qquad (a, \mathit{tv}, ?Y_1)\ \textsc{And}\ \ldots\ \textsc{And}\ (a, \mathit{tv}, ?Y_{i-1})\ \textsc{And} \\ &\qquad (a, \mathit{false}, ?A_{i-1})\ \textsc{And}\ (a, \mathit{true}, ?A_i)), \\ Q_i &:= ((a, \mathit{tv}, ?X_1)\ \textsc{And}\ \ldots\ \textsc{And}\ (a, \mathit{tv}, ?X_i)\ \textsc{And} \\ &\qquad (a, \mathit{tv}, ?Y_1)\ \textsc{And}\ \ldots\ \textsc{And}\ (a, \mathit{tv}, ?Y_i)\ \textsc{And} \\ &\qquad (a, \mathit{false}, ?B_{i-1})\ \textsc{And}\ (a, \mathit{true}, ?B_i)). \end{aligned}$$

Using these expressions, we encode the quantified boolean formula φ as

$$\begin{aligned} P_\varphi := (a, \mathit{true}, ?B_0) \ \textsc{Opt}\ (P_1 \ \textsc{Opt}\ (Q_1 \\ \textsc{Opt}\ (P_2 \ \textsc{Opt}\ (Q_2 \\ \dots \\ \textsc{Opt}\ (P_m \ \textsc{Opt}\ (Q_m \ \textsc{And}\ P_\psi)) \dots))))). \end{aligned}$$

It can be shown that $\mu := \{?B_0 \mapsto 1\} \in [\![P_\varphi]\!]_D$ if and only if φ is valid, which completes the reduction. We do not restate this technical part of the proof here, but refer the interested reader to the proof of Theorem 3 in [PAG06a] for details. □

To clarify the previous encoding of QBF, let us shortly sketch a small example that illustrates the construction for a fixed quantified boolean formula.

Example 3.1 We show how to encode the quantified boolean formula

$$\begin{aligned} \varphi &:= \forall x_1 \exists y_1 (x_1 \Leftrightarrow y_1) \\ &= \forall x_1 \exists y_1 ((x_1 \vee \neg y_1) \wedge (\neg x_1 \vee y_1)), \end{aligned}$$

where $\psi := ((x_1 \vee \neg y_1) \wedge (\neg x_1 \vee y_1))$ is in CNF. First, observe that formula φ is a tautology. The variables in ψ are $V_\psi := \{x_1, y_1\}$; further, we have $C_1 := x_1 \vee \neg y_1$, $C_2 := \neg x_1 \vee y_1$, and $V_{C_1} = V_{C_2} := \{x_1, y_1\}$. Strictly following the construction described in the proof of Theorem 3.3, we set up the database

$$\begin{aligned} D := \{&(a, \mathit{false}, 0), (a, \mathit{true}, 1), (a, \mathit{tv}, 0), (a, \mathit{tv}, 1), \\ &(a, \mathit{var}_1, x_1), (a, \mathit{var}_1, y_1), (a, \mathit{var}_2, x_1), (a, \mathit{var}_2, y_1), \\ &(a, x_1, x_1), (a, y_1, y_1)\}, \end{aligned}$$

where x_1, y_1 are URIs. We next define the expression $P_\psi := P_{C_1}$ AND P_{C_2} with

$$\begin{aligned} P_{C_1} := ((a, \mathit{var}_1, ?\mathit{var}_1) \ \textsc{Opt}\ ((a, x_1, ?\mathit{var}_1) \ \textsc{And}\ (a, \mathit{true}, ?X_1))) \\ \textsc{Opt}\ ((a, y_1, ?\mathit{var}_1) \ \textsc{And}\ (a, \mathit{false}, ?Y_1)), \\ P_{C_2} := ((a, \mathit{var}_2, ?\mathit{var}_2) \ \textsc{Opt}\ ((a, y_1, ?\mathit{var}_2) \ \textsc{And}\ (a, \mathit{true}, ?Y_1))) \\ \textsc{Opt}\ ((a, x_1, ?\mathit{var}_2) \ \textsc{And}\ (a, \mathit{false}, ?X_1)). \end{aligned}$$

When evaluating these expressions we obtain

$$\begin{aligned} [\![P_{C_1}]\!]_D &= (\{\{?\mathit{var}_1 \mapsto x_1\}, \{?\mathit{var}_1 \mapsto y_1\}\} \bowtie \{\{?\mathit{var}_1 \mapsto x_1, ?X_1 \mapsto 1\}\}) \\ &\qquad \bowtie \{\{?\mathit{var}_1 \mapsto y_1, ?Y_1 \mapsto 0\}\} \\ &= \{\{?\mathit{var}_1 \mapsto x_1, ?X_1 \mapsto 1\}, \{?\mathit{var}_1 \mapsto y_1, ?Y_1 \mapsto 0\}\}, \\ [\![P_{C_2}]\!]_D &= \{\{?\mathit{var}_2 \mapsto x_1\}, \{?\mathit{var}_2 \mapsto y_1\}\} \bowtie \{\{?\mathit{var}_2 \mapsto y_1, ?Y_1 \mapsto 1\}\} \\ &\qquad \bowtie \{\{?\mathit{var}_2 \mapsto x_1, ?X_1 \mapsto 0\}\} \\ &= \{\{?\mathit{var}_2 \mapsto x_1, ?X_1 \mapsto 0\}, \{?\mathit{var}_2 \mapsto y_1, ?Y_1 \mapsto 1\}\}, \\ [\![P_\psi]\!]_D &= [\![P_{C_1} \ \textsc{And}\ P_{C_2}]\!]_D \\ &= \{\{?\mathit{var}_1 \mapsto x_1, ?\mathit{var}_2 \mapsto y_1, ?X_1 \mapsto 1, ?Y_1 \mapsto 1\}, \\ &\quad \{?\mathit{var}_1 \mapsto y_1, ?\mathit{var}_2 \mapsto x_1, ?X_1 \mapsto 0, ?Y_1 \mapsto 0\}\}. \end{aligned}$$

We observe that the mappings in $[\![P_{C_1}]\!]_D$, $[\![P_{C_2}]\!]_D$, and $[\![P_\psi]\!]_D$ reflect the satisfying truth assignments for the respective boolean formulas C_1, C_2, and ψ. For instance, formula ψ is true iff either both x_1 and y_1 are *true* or both are *false*. This is reflected by the two result mappings of $[\![P_\psi]\!]_D$: the first mapping binds the corresponding SPARQL variables $?X_1$, $?Y_1$ to 1 (*true*) and the second one maps both to 0 (*false*).

The final step of the encoding is to set up the expressions P_1, Q_1, and P_φ:

$$P_1 := ((a, tv, ?X_1) \text{ AND } (a, false, ?A_0) \text{ AND } (a, true, ?A_1))$$
$$Q_1 := ((a, tv, ?X_1) \text{ AND } (a, tv, ?Y_1) \text{ AND } (a, false, ?B_0) \text{ AND } (a, true, ?B_1))$$
$$P_\varphi := (a, true, ?B_0) \text{ OPT } (P_1 \text{ OPT } (Q_1 \text{ AND } P_\psi))$$

It is easily shown that $\mu := \{?B_0 \mapsto 1\} \in [\![P_\varphi]\!]_D$, which confirms that ψ is valid.□

Note that, in contrast to the PSPACE-hardness proofs for $\mathcal{AOU}$, and $\mathcal{E}$ in [PAG06a] (cf. Theorem 3.1(3) above), the database used in the previous reduction from QBF to fragment $\mathcal{AO}$ is not fixed, but depends on the input formula. Therefore, it is an open question whether the PSPACE-hardness result for $\mathcal{AO}$ carries over to expression complexity (i.e., the evaluation complexity when fixing the database).

Having shown that the EVALUATION problem for class $\mathcal{AO}$ is PSPACE-complete, another interesting question that arises is whether we can find tight complexity bounds for the OPT-only expression fragment $\mathcal{O}$. The following theorem subsumes the previous one and constitutes one of the main results in this chapter.

Theorem 3.4 EVALUATION is PSPACE-complete for class $\mathcal{O}$. □

Given that the EVALUATION problem for fragments $\mathcal{A}$, $\mathcal{F}$, and $\mathcal{U}$ is in PTIME (cf. Theorems 3.1 and 3.2), this finding clarifies that OPT is by far the most complicated operator in the language. The intuition behind this result is that operator $\bowtie$, the algebraic counterpart of OPT, is defined using operators $\bowtie$, $\cup$, and $\setminus$; the mix of these operations (and in particular the operator $\setminus$, which allows to encode negation) compensates for missing AND and UNION operators at syntax level.

In the remainder of this subsection, we will sketch the technically involved proof for Theorem 3.4. Adapting the idea from the proof of Theorem 3.3, we present a reduction from QBF to the EVALUATION problem for SPARQL queries, where the main challenge is to encode the quantified boolean formula using **only** operator OPT. Rather than starting from scratch, our strategy is to take the proof of Theorem 3.3 as a starting point and to replace all AND expressions by OPT-only constructions. As we will see later, most of the AND operators in the encoding can simply be replaced by OPT without changing the semantics. However, for the innermost AND expressions in the encoding of P_φ it turns out that the situation is not that easy. We therefore start with a lemma that will later help us to solve this situation elegantly.

Lemma 3.2 Let

- $Q, Q_1, Q_2, \ldots, Q_n$ ($n \geq 2$) be SPARQL expressions,
- S denote the set of all variables appearing in Q, Q_1, Q_2, ..., Q_n,
- $D := \{(a, \mathit{false}, 0), (a, \mathit{true}, 1), (a, \mathit{tv}, 0), (a, \mathit{tv}, 1)\} \cup D'$ be an RDF database such that $\mathit{dom}(D') \cap \{\mathit{true}, \mathit{false}\} = \emptyset$,
- $?V_2, ?V_3, \ldots, ?V_n$ be a set of $n-1$ variables distinct from the variables in S.

Further, we define the expressions

$$
\begin{array}{l}
V_i := (a, \mathit{true}, ?V_i), \\
\overline{V}_i := (a, \mathit{false}, ?V_i), \\
Q' := ((\ldots((Q \textsc{ Opt } V_2) \textsc{ Opt } V_3)\ldots) \textsc{ Opt } V_n), \text{ and} \\
Q'' := ((\ldots((Q_1 \textsc{ Opt } (Q_2 \textsc{ Opt } V_2)) \\
\qquad\qquad \textsc{Opt } (Q_3 \textsc{ Opt } V_3)) \\
\qquad\qquad \ldots) \\
\qquad\qquad \textsc{Opt } (Q_n \textsc{ Opt } V_n)).
\end{array}
$$

The following claims hold.

$$
\begin{array}{l}
(1)\ [\![Q']\!]_D = \{\mu \cup \{?V_2 \mapsto 1, \ldots, ?V_n \mapsto 1\} \mid \mu \in [\![Q]\!]_D\} \\
(2)\ [\![Q' \textsc{ Opt } (Q_1 \textsc{ And } Q_2 \textsc{ And } \ldots \textsc{ And } Q_n)]\!]_D \\
\qquad = [\![Q' \textsc{ Opt } ((\ldots((Q'' \textsc{ Opt } \overline{V}_2) \textsc{ Opt } \overline{V}_3)\ldots) \textsc{ Opt } \overline{V}_n)]\!]_D
\end{array}
$$

□

Informally speaking, claim (2) of the lemma provides a mechanism to rewrite an And expression that is encapsulated in the right side of an Opt expression by means of an Opt expression. It is important to realize that there is a restriction imposed on the left side expression Q', i.e. Q' is obtained from Q by extending each result mapping in $[\![Q]\!]_D$ by $\{?V_2 \mapsto 1, \ldots, ?V_n \mapsto 1\}$, as stated in claim (1). Before proving the lemma, let us illustrate the construction by means of a small example:

Example 3.2 Consider database $D := \{(a, \mathit{false}, 0), (a, \mathit{true}, 1), (a, \mathit{tv}, 0), (a, \mathit{tv}, 1)\}$ and the three SPARQL expressions

$$
\begin{array}{lll}
Q := (a, \mathit{tv}, ?x) & \text{, thus } [\![Q]\!]_D = & \{\{?x \mapsto 0\}, \{?x \mapsto 1\}\}, \\
Q_1 := (a, \mathit{tv}, ?y) & \text{, thus } [\![Q_1]\!]_D = & \{\{?y \mapsto 0\}, \{?y \mapsto 1\}\}, \\
Q_2 := (a, \mathit{true}, ?y) & \text{, thus } [\![Q_2]\!]_D = & \{\{?y \mapsto 1\}\}.
\end{array}
$$

Concerning claim (1) of Lemma 3.2, we observe that

$$
\begin{array}{l}
[\![Q']\!]_D = [\![Q \textsc{ Opt } V_2]\!]_D \\
\qquad = [\![Q \textsc{ Opt } (a, \mathit{true}, ?V_2)]\!]_D \\
\qquad = \{\{?x \mapsto 0, ?V_2 \mapsto 1\}, \{?x \mapsto 1, ?V_2 \mapsto 1\}\},
\end{array}
$$

so $[\![Q']\!]_D$ differs from $[\![Q]\!]_D$ only in that each mapping contains an additional binding $?V_2 \mapsto 1$. As for claim (2) of the lemma, we observe that the left expression

$$\begin{aligned}
&[\![Q' \text{ Opt } (Q_1 \text{ And } Q_2)]\!]_D \\
&= [\![Q']\!]_D \bowtie \{\{?y \mapsto 1\}\} \\
&= \{\{?x \mapsto 0, ?y \mapsto 1, ?V_2 \mapsto 1\}, \{?x \mapsto 1, ?y \mapsto 1, ?V_2 \mapsto 1\}\}
\end{aligned}$$

yields the same result as the right side expression

$$\begin{aligned}
&[\![Q' \text{ Opt } ((Q_1 \text{ Opt } (Q_2 \text{ Opt } V_2)) \text{ Opt } \overline{V_2})]\!]_D \\
&= [\![Q']\!]_D \bowtie (([\![Q_1]\!]_D \bowtie ([\![Q_2]\!]_D \bowtie [\![V_2]\!]_D)) \bowtie [\![\overline{V_2}]\!]_D) \\
&\overset{(1)}{=} [\![Q']\!]_D \bowtie (([\![Q_1]\!]_D \bowtie \{\{?y \mapsto 1, ?V_2 \mapsto 1\}\}) \bowtie [\![\overline{V_2}]\!]_D) \\
&\overset{(2)}{=} [\![Q']\!]_D \bowtie (\{\{?y \mapsto 0\}, \{?y \mapsto 1, ?V_2 \mapsto 1\}\} \bowtie [\![\overline{V_2}]\!]_D) \\
&\overset{(3)}{=} [\![Q']\!]_D \bowtie \{\{?y \mapsto 0, ?V_2 \mapsto 0\}, \{?y \mapsto 1, ?V_2 \mapsto 1\}\} \\
&\overset{(4a)}{=} \{\{?x \mapsto 0, ?V_2 \mapsto 1\}, \{?x \mapsto 1, ?V_2 \mapsto 1\}\} \\
&\qquad \bowtie \{\{?y \mapsto 0, ?V_2 \mapsto 0\}, \{?y \mapsto 1, ?V_2 \mapsto 1\}\} \\
&\overset{(4b)}{=} \{\{?x \mapsto 0, ?y \mapsto 1, ?V_2 \mapsto 1\}, \{?x \mapsto 1, ?y \mapsto 1, ?V_2 \mapsto 1\}\}.
\end{aligned}$$

The right side expression simulates the inner And expression from the left side using a series of Opt expressions. The idea of the construction is as follows. In step (1) we extend each mapping in $[\![Q_2]\!]_D$ by an additional binding $?V_2 \mapsto 1$. Now recall that $\Omega_1 \bowtie \Omega_2 := (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$. When computing the left outer join between $[\![Q_1]\!]_D$ and the mapping set from step (1) in step (2), the binding $?V_2 \mapsto 1$ will be carried over to mappings that result from the $\bowtie$ part of the left outer join (cf. mapping $\{?y \mapsto 1, ?V_2 \mapsto 1\}$), but does not appear in mappings that are generated from the $\setminus$ part of the left outer join (cf. mapping $\{?y \mapsto 0\}$). Next, in step (3) we extend all mappings from the prior set for which $?V_2$ is not bound by a binding $?V_2 \mapsto 0$. This extension affects only the mapping obtained from the $\setminus$ part, while the mapping from the $\bowtie$ part is left unchanged. In the final steps (4a) and (4b), the bindings $?V_2 \mapsto 1$ in each $\mu \in [\![Q']\!]_D$ serve as filters, which reject all mappings that come from the $\setminus$ part. Thus, only those mappings that have been created by the $\bowtie$ part are retained. Hence, the construction simulates the behavior of the And expression (the syntactic counterparts of operator $\bowtie$) using Opt operators. □

Proof of Lemma 3.2

Lemma 3.2(1): First, we observe that all $?V_i$ are unbound in each $\mu \in [\![Q]\!]_D$, because by assumption the $?V_i$ are fresh variables that do not appear in Q. Next, given that $dom(D')$ does not contain the URI *true* it follows that no triple in D' matches the triple pattern $V_i := (a, true, ?V_i)$, so we have that $[\![V_i]\!]_D = \{\{?V_i \mapsto 1\}\}$. Hence, in $[\![Q']\!]_D$ each mapping $\mu \in [\![Q]\!]_D$ is successively extended by the (compatible) mappings $\{?V_2 \mapsto 1\}, \ldots, \{?V_n \mapsto 1\}$, which implies that the claim holds.

Lemma 3.2(2): We study the evaluation of the right side expression and argue that it yields exactly the same result as the left side expression. Rather than working out all technical details, we try to give the intuition behind the equivalence. We start the discussion with the right side subexpression Q''. First observe that the result of evaluating Q_i Opt V_i corresponds to the result of Q_i, except that each result mapping is extended by $?V_i \mapsto 1$. We use the abbreviation $Q_i^{V_i} := Q_i$ Opt V_i, which allows us to compactly denote Q'' by $((\ldots((Q_1 \text{ Opt } Q_2^{V_2}) \text{ Opt } Q_3^{V_3}) \text{ Opt } \ldots) \text{ Opt } Q_n^{V_n})$. By application of semantics and some simple algebraic laws, such as distributivity of $\bowtie$ over $\cup$ (cf. Chapter 4), we can bring $[\![Q'']\!]_D$ into the form

$$\begin{aligned}[\![Q'']\!]_D &= [\![((\ldots((Q_1 \text{ Opt } Q_2^{V_2}) \text{ Opt } Q_3^{V_3}) \text{ Opt } \ldots) \text{ Opt } Q_n^{V_n})]\!]_D \\ &= \ldots \\ &= [\![Q_1 \text{ And } Q_2^{V_2} \text{ And } Q_3^{V_3} \text{ And } \ldots \text{ And } Q_n^{V_n}]\!]_D \cup P_D,\end{aligned}$$

where we call the left subexpression of the union *join part* and P_D at the right side is an algebra expression (over database D) with the following property: for each mapping $\mu \in P_D$ there is at least one $?V_i$ $(2 \leq i \leq n)$ s.t. $?V_i \notin dom(\mu)$. We observe that, in contrast, for each mapping μ that is generated by the join part, we have that $dom(\mu) \supseteq \{?V_2, \ldots, ?V_n\}$ and, even more, $\mu(?V_i) = 1$, for $2 \leq i \leq n$. Hence, for all these mappings it holds that $\mu(?V_2) = \mu(?V_3) = \cdots = \mu(?V_n) = 1$.

To clarify the previous claims by example, let us consider the case $n = 3$ (the argumentation for this case naturally generalizes to larger n). We then have

$$\begin{aligned}&[\![(Q_1 \text{ Opt } Q_2^{V_2}) \text{ Opt } Q_3^{V_3}]\!]_D \\ &= ([\![Q_1]\!]_D \mathbin{⟕} [\![Q_2^{V_2}]\!]_D) \mathbin{⟕} [\![Q_3^{V_3}]\!]_D \\ &= (([\![Q_1]\!]_D \bowtie [\![Q_2^{V_2}]\!]_D) \cup ([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D)) \mathbin{⟕} [\![Q_3^{V_3}]\!]_D \\ &\overset{(*_1)}{=} (([\![Q_1]\!]_D \bowtie [\![Q_2^{V_2}]\!]_D) \mathbin{⟕} [\![Q_3^{V_3}]\!]_D) \cup (([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \mathbin{⟕} [\![Q_3^{V_3}]\!]_D) \\ &= ((([\![Q_1]\!]_D \bowtie [\![Q_2^{V_2}]\!]_D) \bowtie [\![Q_3^{V_3}]\!]_D) \cup (([\![Q_1]\!]_D \bowtie [\![Q_2^{V_2}]\!]_D) \setminus [\![Q_3^{V_3}]\!]_D)) \cup \\ &\quad ((([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \bowtie [\![Q_3^{V_3}]\!]_D) \cup (([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \setminus [\![Q_3^{V_3}]\!]_D)) \\ &\overset{(*_2)}{=} ([\![Q_1]\!]_D \bowtie [\![Q_2^{V_2}]\!]_D \bowtie [\![Q_3^{V_3}]\!]_D) \cup (([\![Q_1]\!]_D \bowtie [\![Q_2^{V_2}]\!]_D) \setminus [\![Q_3^{V_3}]\!]_D) \cup \\ &\quad (([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \bowtie [\![Q_3^{V_3}]\!]_D) \cup (([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \setminus [\![Q_3^{V_3}]\!]_D) \\ &= [\![Q_1 \text{ And } Q_2^{V_2} \text{ And } Q_3^{V_3}]\!]_D \cup ((([\![Q_1]\!]_D \bowtie [\![Q_2^{V_2}]\!]_D) \setminus [\![Q_3^{V_3}]\!]_D) \cup \\ &\quad (([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \bowtie [\![Q_3^{V_3}]\!]_D) \cup (([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \setminus [\![Q_3^{V_3}]\!]_D)) \\ &= [\![Q_1 \text{ And } Q_2^{V_2} \text{ And } Q_3^{V_3}]\!]_D \cup P_D, \text{ where}\end{aligned}$$

$$\begin{aligned}P_D := &(([\![Q_1]\!]_D \bowtie [\![Q_2^{V_2}]\!]_D) \setminus [\![Q_3^{V_3}]\!]_D) \cup (([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \bowtie [\![Q_3^{V_3}]\!]_D) \cup \\ &(([\![Q_1]\!]_D \setminus [\![Q_2^{V_2}]\!]_D) \setminus [\![Q_3^{V_3}]\!]_D).\end{aligned}$$

Step $(*_1)$ is justified by the right distributivity of the left outer join over union and step $(*_2)$ follows from the associativity of union and join (we refer the reader to the rules from Figure 4.2 in Section 4.2.3 for details). Now observe that each

mapping generated by P_D is generated by (exactly) one of the three subexpressions $P_{\bowtie\setminus} := ((\llbracket Q_1 \rrbracket_D \bowtie \llbracket Q_2^{V_2} \rrbracket_D) \setminus \llbracket Q_3^{V_3} \rrbracket_D)$, $P_{\setminus\bowtie} := ((\llbracket Q_1 \rrbracket_D \setminus \llbracket Q_2^{V_2} \rrbracket_D) \bowtie \llbracket Q_3^{V_3} \rrbracket_D)$, or by $P_{\setminus\setminus} := ((\llbracket Q_1 \rrbracket_D \setminus \llbracket Q_2^{V_2} \rrbracket_D) \setminus \llbracket Q_3^{V_3} \rrbracket_D)$. It is easy to see that in mappings generated by $P_{\bowtie\setminus}$ variable $?V_3$ is not bound, in mappings generated by $P_{\setminus\bowtie}$ variable $?V_2$ is not bound, and in mappings generated by $P_{\setminus\setminus}$ neither $?V_2$ nor $?V_3$ is bound. Hence, P_D cannot generate a mapping in which both $?V_2$ and $?V_3$ are bound. In contrast, both variable $?V_2$ and $?V_3$ are bound in the join part $\llbracket Q_1 \text{ And } Q_2^{V_2} \text{ And } Q_3^{V_3} \rrbracket_D$.

Let us now go one step further and consider the larger right side subexpression

$$P' := ((\ldots((Q'' \text{ Opt } \overline{V}_2) \text{ Opt } \overline{V}_3) \text{ Opt } \ldots) \text{ Opt } \overline{V}_n).$$

It is easily verified that, when evaluating expression P', we obtain exactly the mappings from $\llbracket Q'' \rrbracket_D$, but each mapping $\mu \in \llbracket Q'' \rrbracket_D$ is extended by $?V_i \mapsto 0$ for all variables $?V_i \notin dom(\mu)$ with $2 \leq i \leq n$. As argued before, all mappings in the join part of Q'' are complete in the sense that **all** $?V_i$ are bound to 1, so these mappings are not modified. The remaining mappings (i.e. those originating from P_D) will be extended by bindings $?V_i \mapsto 0$ for at least one $?V_i$. The resulting situation can be summarized as follows: first, for each $\mu \in \llbracket P' \rrbracket_D$ we have $dom(\mu) \supseteq \{?V_2, \ldots, ?V_n\}$; second, for those $\mu \in \llbracket P' \rrbracket_D$ that evolve from the join part of $\llbracket Q'' \rrbracket_D$ we have that $\mu(?V_2) = \cdots = \mu(?V_n) = 1$; third, for those $\mu \in \llbracket P' \rrbracket_D$ that evolve from the subexpression P_D (i.e., not from the join part) there is $i \in \{2, \ldots, n\}$ such that $\mu(?V_i) = 0$.

Going one step further, we finally consider the whole right side expression, namely $\llbracket Q' \text{ Opt } P' \rrbracket_D$. From claim (1) of the lemma we know that each mapping in $\llbracket Q' \rrbracket_D$ maps all $?V_i$ to 1. Hence, when computing $\llbracket Q' \text{ Opt } P' \rrbracket_D = \llbracket Q' \rrbracket_D \bowtie\!\!\!\!\!- \llbracket P' \rrbracket_D$, the bindings $?V_i \mapsto 1$ for all $i \in \{2, \ldots, n\}$ in every $\mu \in \llbracket Q' \rrbracket_D$ assert that the mappings in $\llbracket Q' \rrbracket_D$ are pairwise incompatible with those mapping from $\llbracket P' \rrbracket_D$ that bind one or more $?V_i$ to 0. As discussed before, the condition that at least one $?V_i$ maps to 0 holds for exactly those mappings that originate from P_D, so all mappings originating from P_D do not contribute to the result of $\llbracket Q' \text{ Opt } P' \rrbracket_D$. Hence, it holds that

$$\begin{aligned}
\llbracket Q' \text{ Opt } P' \rrbracket_D &= \llbracket Q' \rrbracket_D \bowtie\!\!\!\!\!- \llbracket P' \rrbracket_D \\
&= \llbracket Q' \rrbracket_D \bowtie\!\!\!\!\!- \llbracket Q_1 \text{ And } Q_2^{V_2} \text{ And } Q_3^{V_3} \text{ And } \ldots \text{ And } Q_n^{V_n} \rrbracket_D \\
&= \llbracket Q' \text{ Opt } (Q_1 \text{ And } Q_2^{V_2} \text{ And } Q_3^{V_3} \text{ And } \ldots \text{ And } Q_n^{V_n}) \rrbracket_D.
\end{aligned}$$

Even more, we know from claim (1) of the lemma that all $?V_i$ are bound to 1 for each $\mu \in \llbracket Q' \rrbracket_D$. It follows that we can replace $Q_i^{V_i} := Q_i \text{ Opt } V_i$ by Q_i in P', without changing the semantics of expression $\llbracket Q' \text{ Opt } P' \rrbracket_D$:

$$\begin{aligned}
\llbracket Q' \text{ Opt } P' \rrbracket_D &= \llbracket Q' \text{ Opt } (Q_1 \text{ And } Q_2^{V_2} \text{ And } Q_3^{V_3} \text{ And } \ldots \text{ And } Q_n^{V_n}) \rrbracket_D \\
&= \llbracket Q' \text{ Opt } (Q_1 \text{ And } Q_2 \text{ And } Q_3 \text{ And } \ldots \text{ And } Q_n) \rrbracket_D
\end{aligned}$$

The final step in our transformation corresponds exactly to the left side expression of the original claim (2), which completes the proof. □

Proof of Theorem 3.4

Having established Lemma 3.2 we are now in the position to prove PSPACE-completeness for fragment $\mathcal{O}$. As before in the proof of Theorem 3.3, it suffices to show hardness. Following the idea discussed before, we show that each AND expression in the proof of Theorem 3.3 can be replaced by a construction using only OPT expressions. Let us again start with a quantified boolean formula

$$\varphi := \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_m \exists y_m \psi,$$

where ψ is a quantifier-free formula in conjunctive normal form, i.e. ψ is a conjunction of clauses $\psi := C_1 \wedge \cdots \wedge C_n$ where the C_i ($i \in [n]$), are disjunctions of literals. As before, by V_ψ we denote the set of variables inside ψ, by V_{C_i} the variables in clause C_i (either in positive of negative form), and we define the database

$$D := \{(a, tv, 0), (a, tv, 1), (a, false, 0), (a, true, 1)\} \cup \{(a, var_i, v) \mid v \in V_{C_i}\} \cup \{(a, v, v) \mid v \in V_\psi\}.$$

The first modification of the proof for class $\mathcal{AO}$ concerns the encoding of clauses $C_i := v_1 \vee \cdots \vee v_j \vee \neg v_{j+1} \vee \cdots \vee \neg v_k$. In the prior encoding we used both AND and OPT operators to encode such clauses. It is easy to see that we can simply replace each AND operator there by OPT without changing semantics. The reason is that, for all subexpressions P_1 OPT P_2 in the encoding of P_{C_i}, we have $vars(P_1) \cap vars(P_2) = \emptyset$ and $[\![P_2]\!]_D \neq \emptyset$. More precisely, each AND expression in the encoding P_{C_i} is of the form $(a, v_j, ?var_i)$ AND $(a, false, ?V_j)$ (or $(a, v_j, ?var_i)$ AND $(a, true, ?V_j)$), so the right side pattern generates one result mapping $\{?V_j \mapsto 0\}$ (or $\{?V_j \mapsto 1\}$), which is compatible with the single mapping $\{?var_i \mapsto v_j\}$ obtained when evaluating the left pattern. Clearly, in this case the left join is identical to the join. When replacing all AND operators by OPT, we obtain the OPT-only encoding $P_{C_i}^{\text{OPT}}$ for clauses C_i:

$$\begin{aligned} P_{C_i}^{\text{OPT}} := (\ldots((\ldots((a, var_i, ?var_i) \\ &\text{OPT } ((a, v_1, ?var_i) \text{ OPT } (a, true, ?V_1))) \\ &\quad \ldots \\ &\quad \text{OPT } ((a, v_j, ?var_i) \text{ OPT } (a, true, ?V_j))) \\ &\quad\quad \text{OPT } ((a, v_{j+1}, ?var_i) \text{ OPT } (a, false, ?V_{j+1}))) \\ &\quad\quad\quad \ldots \\ &\quad\quad\quad \text{OPT } ((a, v_k, ?var_i) \text{ OPT } (a, false, ?V_k))). \end{aligned}$$

This encoding gives us a preliminary encoding P'_ψ for formula ψ (as a replacement for P_ψ from the proof for Theorem 3.3), defined as $P'_\psi := P_{C_1}^{\text{OPT}}$ AND ... AND $P_{C_n}^{\text{OPT}}$; we will tackle the replacement of the remaining AND expressions in P'_ψ later. Let us next consider the P_i and Q_i used for simulating the quantifier alternation. With a similar argumentation as before, we can replace each occurrence of operator AND by OPT without changing the semantics. This modification results in the equivalent OPT-only encodings P_i^{OPT} (for P_i) and Q_i^{OPT} (for Q_i), $i \in [m]$, defined as

$$
\begin{aligned}
P_i^{\text{OPT}} := {} & ((a, tv, ?X_1)\ \text{OPT}\ \ldots\ \text{OPT}\ (a, tv, ?X_i)\ \text{OPT} \\
& (a, tv, ?Y_1)\ \text{OPT}\ \ldots\ \text{OPT}\ (a, tv, ?Y_{i-1})\ \text{OPT} \\
& (a, \mathit{false}, ?A_{i-1})\ \text{OPT}\ (a, \mathit{true}, ?A_i)), \\
Q_i^{\text{OPT}} := {} & ((a, tv, ?X_1)\ \text{OPT}\ \ldots\ \text{OPT}\ (a, tv, ?X_i)\ \text{OPT} \\
& (a, tv, ?Y_1)\ \text{OPT}\ \ldots\ \text{OPT}\ (a, tv, ?Y_i)\ \text{OPT} \\
& (a, \mathit{false}, ?B_{i-1})\ \text{OPT}\ (a, \mathit{true}, ?B_i)).
\end{aligned}
$$

Let us shortly summarize what we have achieved so far. Given the modifications presented before, our preliminary encoding P'_φ for φ is

$$
\begin{aligned}
P'_\varphi := {} & (a, \mathit{true}, ?B_0)\ \text{OPT}\ (P_1^{\text{OPT}}\ \text{OPT}\ (Q_1^{\text{OPT}} \\
& \qquad \ldots \\
& \qquad \text{OPT}\ (P_{m-1}^{\text{OPT}}\ \text{OPT}\ (Q_{m-1}^{\text{OPT}} \\
& \qquad\quad \text{OPT}\ P_*))\ldots)), \text{ where}
\end{aligned}
$$

$$
\begin{aligned}
P_* &:= P_m^{\text{OPT}}\ \text{OPT}\ (Q_m^{\text{OPT}}\ \text{AND}\ P'_\psi) \\
&= P_m^{\text{OPT}}\ \text{OPT}\ (Q_m^{\text{OPT}}\ \text{AND}\ P_{C_1}^{\text{OPT}}\ \text{AND}\ \ldots\ \text{AND}\ P_{C_n}^{\text{OPT}}).
\end{aligned}
$$

Expression P_* is the only subexpression of P'_φ that still contains AND operators (where Q_m^{OPT}, $P_{C_1}^{\text{OPT}}$, …, $P_{C_n}^{\text{OPT}}$ are OPT-only expressions). We now exploit the rewriting from Lemma 3.2(2) and replace P_* by the $\mathcal{O}$ expression P_*^{OPT} defined as

$$
P_*^{\text{OPT}} := Q'\ \text{OPT}\ ((\ldots((Q''\ \text{OPT}\ \overline{V}_2)\ \text{OPT}\ \overline{V}_3)\ \text{OPT}\ \ldots)\ \text{OPT}\ \overline{V}_{n+1})), \text{ where}
$$

$$
\begin{aligned}
Q' &:= ((\ldots((P_m^{\text{OPT}}\ \text{OPT}\ V_2)\ \text{OPT}\ V_3)\ldots)\ \text{OPT}\ V_{n+1}), \\
Q'' &:= ((\ldots((Q_m^{\text{OPT}}\ \text{OPT}\ (P_{C_1}^{\text{OPT}}\ \text{OPT}\ V_2)) \\
&\qquad\qquad \text{OPT}\ (P_{C_2}^{\text{OPT}}\ \text{OPT}\ V_3)) \\
&\qquad\qquad \ldots \\
&\qquad\qquad \text{OPT}\ (P_{C_n}^{\text{OPT}}\ \text{OPT}\ V_{n+1}))), \\
V_i &:= (a, \mathit{true}, ?V_i), \overline{V}_i := (a, \mathit{false}, ?V_i), \\
&\text{and the } ?V_i\ (i \in \{2, \ldots, n+1\}) \text{ are fresh variables.}
\end{aligned}
$$

Let P_φ^{OPT} denote the expression obtained from P'_φ by replacing the subexpression P_* by P_*^{OPT}. First observe that P_φ^{OPT} is an $\mathcal{O}$ expression. From Lemma 3.2(2) it follows that $[\![P_*^{\text{OPT}}]\!]_D$ equals to $[\![Q'\ \text{OPT}\ (Q_m^{\text{OPT}}\ \text{AND}\ P_{C_i}^{\text{OPT}} \ldots\ \text{AND}\ P_{C_n}^{\text{OPT}})]\!]_D$, where the evaluation result $[\![Q']\!]_D$ is obtained from $[\![P_m^{\text{OPT}}]\!]_D$ by extending each $\mu \in [\![P_m^{\text{OPT}}]\!]_D$ with bindings $?V_2 \mapsto 1, \ldots, ?V_{n+1} \mapsto 1$, according to Lemma 3.2(1). Consequently, the result obtained when evaluating P_*^{OPT} is identical to $[\![P_*]\!]_D$ except for the additional bindings for (the fresh) variables $?V_2, \ldots, ?V_{n+1}$. It is straightforward to verify that these bindings do not harm the overall construction, i.e. it is straightforward to show that $\{?B_0 \mapsto 1\} \in [\![P_\varphi^{\text{OPT}}]\!]_D$ iff φ is valid. □

We conclude this subsection with a corollary that follows from Theorems 3.1 and 3.4 and makes the complexity study of the expression fragments complete:

Corollary 3.2 The EVALUATION problem for every expression fragment involving operator OPT is PSPACE-complete. □

3.2.4. The Source of Complexity

Given the high complexity of operator Opt, an interesting question is whether we can find natural syntactic conditions that lower the complexity of fragments that involve this operator. A closer investigation reveals that the proofs of Theorems 3.3 and 3.4 both rely on a nesting of Opt expression, which increases with the number of quantifier alternations in the quantified boolean formula that is used in the reduction of QBF to the SPARQL Evaluation problem. It turns out that, when restricting the nesting depth of Opt expressions, better complexity bounds in the polynomial hierarchy can be derived (assuming that the complexity classes in the polynomial hierarchy are strictly contained in PSpace, a widely accepted conjecture in complexity theory). We start with the notion of Opt-rank, which is a measure for the nesting depth of Opt expressions and will be used to formalize our restriction:

Definition 3.2 (Opt-rank) The nesting depth of Opt expressions in expression Q, called Opt-rank $rank(Q)$, is defined inductively on the structure of Q as

$$
\begin{array}{ll}
rank(\mathrm{t}) & := 0 \\
rank(Q_1 \text{ Filter } R) & := rank(Q_1) \\
rank(Q_1 \text{ And } Q_2) & := max(rank(Q_1), rank(Q_2)) \\
rank(Q_1 \text{ Union } Q_2) & := max(rank(Q_1), rank(Q_2)) \\
rank(Q_1 \text{ Opt } Q_2) & := max(rank(Q_1), rank(Q_2)) + 1,
\end{array}
$$

where function $max(n_1,n_2)$ returns the maximum of n_1 and n_2. □

Let F be an expression fragment. By $F_{\leq n}$ we denote the class of expressions $Q \in F$ with $rank(Q) \leq n$. When fixing the Opt-rank of $\mathcal{E}$ expressions, the SPARQL Evaluation problem falls into some fixed class of the polynomial hierarchy:

Theorem 3.5 For every $n \in \mathbb{N}_0$, the Evaluation problem is Σ^P_{n+1}-complete for the SPARQL fragment $\mathcal{E}_{\leq n}$. □

Observe that, according to the theorem, Evaluation for class $\mathcal{E}_{\leq 0}$ is complete for Σ^P_1=NP, which is identical to the result for Opt-free expressions (i.e., class $\mathcal{AFU}$) stated in Theorem 3.1. With increasing nesting-depth of Opt expressions we climb up the polynomial hierarchy. This is reminiscent of the QBF problem for quantified boolean formulas with restricted quantifier alternation, where the number of quantifier alternations fixes the complexity class in the polynomial hierarchy. In fact, the hardness part of the proof (see Appendix B.1) makes these similarities explicit.

3.2.5. From Expressions to Queries

We finally turn towards a discussion of SPARQL queries, i.e. fragments involving top-level projection in the form of a SELECT operator (see Definition 2.5). We extend the notation for classes as follows. Let F be an expression fragment. We denote by F^π the class of queries of the form $\text{SELECT}_S(Q)$, where $S \subset V$ is a finite set of variables and $Q \in F$ is an expression. The next lemma shows that we obtain (top-level) projection for free in all fragments that are at least NP-complete.

Lemma 3.3 Let C be a complexity class and F a class of expressions. If EVALUATION is C-complete for F and C $\supseteq$ NP then EVALUATION is C-complete for F^π. □

Proof of Lemma 3.3

Let F be a fragment for which the EVALUATION problem is C-complete, where C is a complexity class such that C $\supseteq$ NP. We argue that, for a query $Q \in F^\pi$, document D, and mapping μ, testing if $\mu \in [\![Q]\!]_D$ is contained in C (C-hardness follows trivially from C-completeness of fragment F). By definition, each query in F^π is of the form $Q := \text{SELECT}_S(Q')$, where $S \subset V$ is a finite set of variables and $Q' \in F$. According to the semantics of SELECT, we have that $\mu \in [\![Q]\!]_D$ iff there is a mapping $\mu' \supseteq \mu$ in $[\![Q']\!]_D$ such that $\pi_S(\{\mu'\}) = \{\mu\}$. We observe that the domain of candidate mappings μ' is bounded by the set of variables in Q' and $dom(D)$. Hence, we can first guess a mapping $\mu' \supseteq \mu$ (recall that we are at least in NP) and subsequently check if $\pi_S(\{\mu'\}) = \{\mu\}$ (in polynomial time) and $\mu' \in [\![Q']\!]_D$ (using a C-algorithm, by assumption). Clearly, this algorithm falls into class C. □

We naturally extend the OPT-rank from Definition 3.2 from SPARQL expressions to queries and define $rank(\text{SELECT}_S(Q)) := rank(Q)$. Combining the observation in Lemma 3.3 with previous findings we obtain several new complexity results:

Corollary 3.3 The SPARQL EVALUATION problem is

1. PSPACE-complete for all query fragments involving operator OPT,
2. Σ^P_{n+1}-complete for the SPARQL fragment $\mathcal{E}^\pi_{\leq n}$ (for $n \in \mathbb{N}_0$), and
3. NP-complete for fragment $\mathcal{AU}^\pi$. □

Proof of Corollary 3.3

Corollary 3.3(1): Follows immediately from Lemma 3.3 and Corollary 3.2.
Corollary 3.3(2): Follows immediately from Lemma 3.3 and Theorem 3.5.
Corollary 3.3(3): Follows immediately from Lemma 3.3 and Theorem 3.2(2). □

It is still an open question whether top-level projection increases the complexity of expression fragments that are in PTIME. The following theorem clarifies this issue.

Theorem 3.6 The EVALUATION problem is

1. in PTIME for classes $\mathcal{FU}^\pi$, $\mathcal{F}^\pi$, $\mathcal{U}^\pi$, and
2. NP-complete for classes $\mathcal{A}^\pi$ and $\mathcal{AF}^\pi$. □

Proof of Theorem 3.6

Theorem 3.6(1): We prove membership in PTIME for fragment $\mathcal{FU}^\pi$, which directly implies PTIME-membership for $\mathcal{F}^\pi$ and $\mathcal{U}^\pi$. Let D be an RDF database, μ be a mapping, and $Q := \text{SELECT}_S(Q')$ be an $\mathcal{FU}^\pi$ expression. We show that there is a PTIME-algorithm that checks if $\mu \in [\![Q]\!]_D$. Let $t_1, \ldots, t_n$ be all triple patterns occurring in Q. Our strategy is as follows: we process triple pattern by triple pattern and check for each $\mu' \in [\![t_i]\!]_D$ if the following two conditions hold: (1) all filter conditions that are defined on top of t_i in Q' satisfy μ' and (2) $\pi_S(\{\mu'\}) = \{\mu\}$. We return *true* if there is a mapping that satisfies both conditions, *false* otherwise.

The idea behind this algorithm is that condition (1) implies that $\mu' \in [\![Q']\!]_D$, while condition (2) asserts that the top-level projection generates mapping μ from μ'. It is straightforward to show that $\mu \in [\![Q]\!]_D$ if and only if there is some $i \in [n]$ such that $[\![t_i]\!]_D$ contains a mapping μ' that satisfies both conditions, and clearly our algorithm (which checks all candidates) would find such a mapping, if it exists. The number of triple patterns is linear to the size of the query and the number of mappings in each $[\![t_i]\!]_D$ is linear to the size of D (where each mapping is of bounded size); further, conditions (1) and (2) can be checked in PTIME, so the algorithm is in PTIME.

Theorem 3.6(2): First, we show that EVALUATION for $\mathcal{AF}^\pi$-queries is contained in NP (membership for $\mathcal{A}^\pi$ queries then follows). By definition, each query in $\mathcal{AF}^\pi$ is of the form $Q := \text{SELECT}_S(Q')$, where $S \subset V$ is a finite set of variables and Q' is an $\mathcal{AF}$ expression. We fix a document D and a mapping μ. To prove membership, we follow the approach taken in the proof of Lemma 3.3 and eliminate the SELECT-clause. More precisely, we guess a mapping $\mu' \supseteq \mu$ s.t. $\pi_S(\{\mu'\}) = \mu$ and check if $\mu' \in [\![Q']\!]_D$ (see the proof of Lemma 3.3 for more details). The size of the mapping to be guessed is bounded, and it is easy to see that the resulting algorithm is in NP.

To prove NP-hardness for $\mathcal{A}^\pi$ and $\mathcal{AF}^\pi$ we reduce 3SAT, a prototypical NP-complete problem, to the EVALUATION problem for class $\mathcal{A}^\pi$. The subsequent proof was inspired by the reduction of 3SAT to the evaluation problem for conjunctive queries in [BEE+07]. It nicely illustrates the relation between AND-only queries and conjunctive queries. We start with a formal definition of the 3SAT problem.

> 3SAT: given a boolean formula $\psi := C_1 \wedge \cdots \wedge C_n$ in conjunctive normal form as input, where each clause C_i is a disjunction of exactly three literals: is the formula ψ satisfiable?

Let $\psi := C_1 \wedge \cdots \wedge C_n$ be a boolean formula in CNF, where each C_i is of the form $C_i := l_{i1} \vee l_{i2} \vee l_{i3}$ and the l_{ij} are literals. For our encoding we use the fixed database

$$D := \{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),\\ (1,0,1),(1,1,0),(1,1,1),(0,c,1),(1,c,0)\},$$

where we assume that 0 and 1 are URIs. Further let $V_\psi = \{x_1, \ldots x_m\}$ denote the set of variables occurring in formula ψ. We define the AND-only expression

$$P' := (L^*_{11}, L^*_{12}, L^*_{13}) \text{ AND } \ldots \text{ AND } (L^*_{n1}, L^*_{n2}, L^*_{n3})\\ \text{AND } (?X_1, c, ?\overline{X}_1) \text{ AND } \ldots \text{ AND } (?X_m, c, ?\overline{X}_m)\\ \text{AND } (0, c, ?A),$$

where $L^*_{ij} := ?X_k$ if $l_{ij} = x_k$, and $L^*_{ij} := ?\overline{X}_k$ if $l_{ij} = \neg x_k$.

Finally, define $P := \text{SELECT}_{?A}(P')$. It is easily verified that $[\![P']\!]_D$ computes all satisfying truth assignments for ψ, extended by an additional binding $?A \mapsto 1$. Consequently, formula ψ is satisfiable if and only if $\mu := \{?A \mapsto 1\} \in [\![P]\!]_D$. □

3.2.6. Summary of Results

We summarize (and slightly extend) the results established throughout Sections 3.2.2 to 3.2.5 in Figure 3.1. We point out that all fragments that fall into the classes NP, Σ^P_i, and PSPACE also are complete for the respective complexity class.

First, according to Corollary 3.2, all expressions including operator OPT fall into PSPACE if the nesting depth is not explicitly fixed. In addition, as stated in Corollary 3.3(1), also the corresponding query classes are PSPACE-complete.

When fixing the nesting depth of classes containing operator OPT, we obtain fragments that fall into the polynomial hierarchy. As indicated in the survey, each class Σ^P_{n+1} contains the fragment $\mathcal{E}_{\leq n}$ (cf. Theorem 3.5) and the corresponding query fragment $\mathcal{E}^\pi_{\leq n}$ (cf. Corollary 3.3(2)). Note that class Σ^P_1 equals to NP and that fragment $\mathcal{AFOU}_{\leq 0}$ equals to $\mathcal{AFU}$, which is contained in $\Sigma^P_1 = \text{NP}$.

In addition to the fragments $\mathcal{E}_{\leq n}$ and $\mathcal{E}^\pi_{\leq n}$, each Σ^P_{n+1} also contains the fragment $\mathcal{AFO}_{\leq n}$ and the respective query fragment $\mathcal{AFO}^\pi_{\leq n}$. These results were not explicitly stated before, but follow from the proof of Theorem 3.5 in Appendix B.1, which does not use the UNION operator for the encoding of the given quantified boolean formula.

Going downwards in the complexity hierarchy, we next identify several fragments that are NP-complete. The survey indicates that there are exactly two constellations that are responsible for NP-hardness, namely either the combination of AND and UNION or the combination of AND with projection. The associated results were established in Theorems 3.1(2), 3.2(2), 3.6(2), and Corollary 3.3(3).

Finally, Figure 3.1 lists all fragments that have been shown to be in PTIME. The latter class contains the single-operator expression fragments $\mathcal{A}$, $\mathcal{F}$, $\mathcal{U}$ and the classes obtained when combining either operator AND or operator UNION with FILTER expressions (cf. Theorems 3.1(1) and 3.2(1)). In addition, the lightweight query classes $\mathcal{F}^\pi$, $\mathcal{U}^\pi$, and $\mathcal{FU}^\pi$ fall into PTIME, according to Theorem 3.6(1).

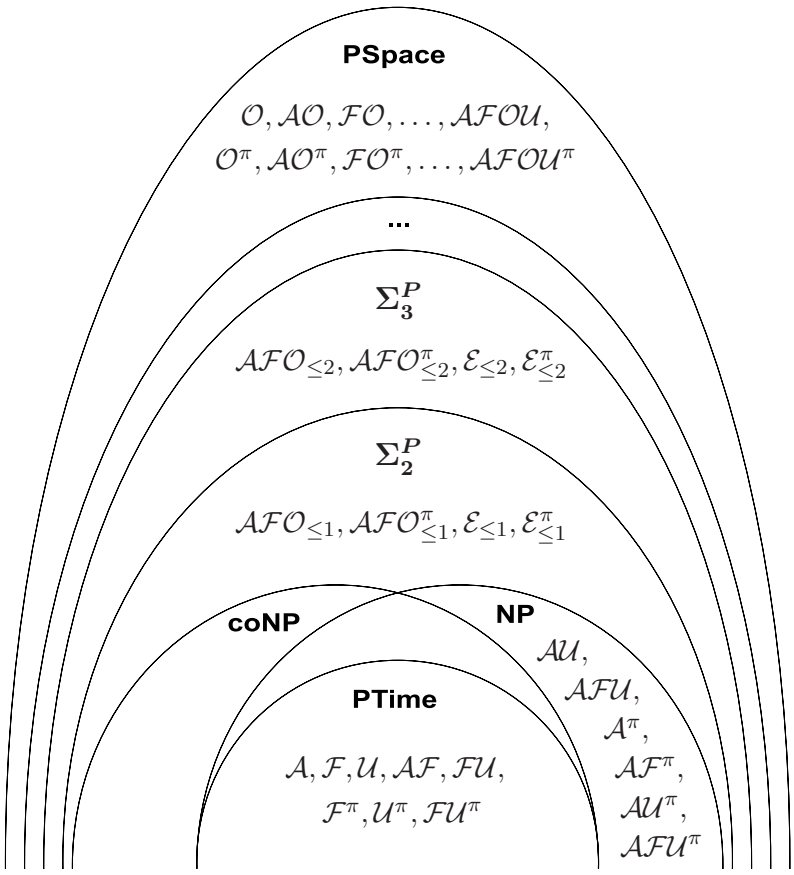

Figure 3.1.: Summary of complexity results.

3.3. Related Work

Complexity of Query Languages. The study of complexity, expressiveness, and relationships between database query languages has been an important topic ever since the invention of the relational model by Codd [Cod69; Cod70]. Few years after its proposal, Codd himself introduced a predicate calculus called relational calculus [Cod72], which can be understood as a dialect of first-order logic[5] over the relational data model. Following Codd's argumentation, the relational calculus should fix a lower bound for the expressive power of database query languages, thus serving as a yardstick when designing new query languages for the relational model. He coined the notion of *relational completeness* to identify query languages that are at least as expressive as the relational calculus and showed that relational algebra (RA), which was introduced before in [Cod70], is relationally complete, a

[5]We introduce first-order logic in Chapter 5 and will therefore not go into more detail here.

result that is widely known as *Codd's Theorem* in the literature. Due to the close connection between the relational calculus and first-order logic, it essentially shows that relational algebra has the same expressiveness as first-order logic.

In the years after these early results, lots of research effort was spent in investigating the complexity and expressiveness of relational algebra, its fragments, and possible extensions. A simple yet expressive fragment of relational algebra are so-called conjunctive queries, a subclass of RA comprising the (relational) selection, projection, and join operator. In [CM77] it was shown that query evaluation is NP-complete for conjunctive queries and a near-optimal algorithm for implementing such queries was developed. One property that makes conjunctive queries particularly attractive is that the containment problem, and hence also the equivalence problem, is decidable (in fact, it can easily be reduced to the problem of conjunctive query answering). Going one step further, [JK82] studied the problem of conjunctive query containment under data dependencies and showed that this problem is still decidable for large, practically relevant classes of dependencies. These results are of particular interest in this thesis: as a practical application, we will later show in Chapter 5 that the minimization and optimization problem for SPARQL And-only queries under data dependencies can be solved by falling back on the containment and optimization problem for conjunctive queries under data dependencies.

In 1979, Aho and Ullman revisited the notion of relational completeness and highlighted important classes of queries that cannot be expressed in relational algebra [AU79]. Among others, they formally proved that relational algebra is not expressive enough to compute the transitive closure of binary relations. For such queries, more expressive query languages are required, which go beyond first-order logic and have their logical foundations in higher-order logics. Investigating the relations between relational algebra, fragments, and extensions, a classification of query languages using the so-called *fixpoint query hierarchy* was worked out in [CH80].

Another paper that influenced subsequent studies of query language complexity was [Var82]. The work proposes two novel complexity measures for query languages: *data complexity* considers the complexity of a query language as a function of the size of the database, whereas *expression complexity* denotes the study of the evaluation complexity as a function of the size of the query. These measures complement the study of combined complexity used in previous investigations, in which neither database nor query are fixed. As an important result, a comparative study of existing query languages reveals that the expression complexity of query languages is typically one exponential higher than their associated data complexity.

In addition to relational algebra, other query languages and data models have been proposed and theoretically investigated. One important line of research that goes into this direction is the study of datalog [MW88], a rule-based query language for deductive databases, which syntactically and semantically resembles the Prolog programming language [SS86]. To date, a variety of results have been established, relating datalog and possible extensions (e.g., [EGM97]) or fragments of the language

to relational algebra, both in terms of expressiveness and complexity. A detailed discussion of all these results is beyond the scope of this summary. We refer the interested reader to [AHV] for a survey of important results and to [BEE+07] for a comprehensive introduction to rule-based query languages. Another notable line of related research is the theoretical investigation of query languages for the tree-structured XML data format, such as XQuery and XPath, e.g. in [GKP03; GKPS05].

Complexity of RDF Query Languages. Early complexity results on query answering on RDF(S) databases have been presented in [GHM04]. Amongst others, the authors investigate a datalog-style, rule-based query language for RDF(S) graphs. In contrast to our investigation of SPARQL complexity, the focus of the latter work is on query containment and answering on the implied database. While the query language is rather simple compared to SPARQL, additional challenges arise due to the fact that query answering must take RDF(S) inferencing into account.

The first complexity analysis for the SPARQL query language has been presented in [PAG06a]. Beyond the results on combined complexity that we summarized in Theorem 3.1, the authors show that the data complexity version of the Evaluation problem is LogSpace-complete. The refined journal version [PAG09] presents some more interesting results. One finding is that so-called well-designed graph patterns, a practically relevant class of SPARQL expressions defined by a syntactic restriction, are coNP-complete (although such patterns might contain nested Opt expressions). As a further result in [PAG09], the authors prove that the SPARQL Evaluation problem is PSpace-complete for class $\mathcal{AFO}$, which is subsumed by our result that already fragment $\mathcal{O}$ is PSpace-complete.[6] Related to [PAG06a; PAG09] is also the complexity study for a navigational, path-based extension of SPARQL, called nSPARQL, presented in [PAG08] (cf. the discussion in Section 2.4).

In addition to the investigation of SPARQL complexity, there has been work concerning the expressiveness of the language. Early mapping schemes for SPARQL into relational algebra and SQL [Cyg05; CLJF06] indicate a close connection between SPARQL and relational algebra in terms of expressiveness. In [Pol07] then, a translation of SPARQL queries into a datalog fragment that is known to be equally expressive as relational algebra was presented. This translation makes the close connection between SPARQL and rule-based languages explicit and shows that RA is at least as expressive as SPARQL. Tackling the opposite direction, it was recently shown in [AG08a] that SPARQL is relationally complete, by providing a translation of the above-mentioned datalog fragment into SPARQL. As argued in [AG08a], the results from [Pol07] and [AG08a] taken together imply that SPARQL has exactly the same expressive power as relational algebra.

[6] Note that we already published this result informally several months earlier in [SML08].

3.4. Conclusion

In this chapter we presented a comprehensive complexity analysis for SPARQL and fragments of the language. While all results have been proven in the context of the set semantics introduced in Definition 2.11, we showed that – from a complexity point of view – the bag semantics from Definition 2.15 does not differ, so all results immediately carry over to the official W3C SPARQL Recommendation.

As a key result, our investigation of SPARQL complexity reveals that operator Opt alone is responsible for the high complexity (i.e., PSpace-completeness) of the query language. The theorem upgrades the claim from [PAG06a] that "the main source of complexity in SPARQL comes from the combination of Union and Opt operators", by showing that Union (and And) are not necessary to obtain PSpace-hardness. The high complexity is caused by the possibility of an unlimited nesting of Opt-expressions and arises from the fact that the definition of the Opt operator implicitly involves negation: on the one hand, it joins mappings from the left side expression with compatible mappings in the right side, on the other hand, it also retains all mappings from the left side mapping set for which **no** compatible mapping in the right mapping set exists. When fixing the nesting depth of Opt-expressions, we implicitly limit the nesting depth of such negation encodings and in that case better complexity bounds in the polynomial hierarchy can be derived (the classes in this hierarchy are conjectured to be strictly contained in PSpace).

Beyond this central result, we presented a complete study for all SPARQL expression and query fragments, in particular showing that, in Opt-free fragments, it is the combination of operators And and Union or And and Select that makes the Evaluation problem for SPARQL hard (in this case, NP-complete). Such fragments are quite related to conjunctive queries, whose evaluation complexity falls into the same class. Later, in Chapter 5, we will use a translation of SPARQL And queries to conjunctive queries, which allows us to apply algorithms and results that have been developed for this important and well-studied query class.

Ultimately, the complexity results that have been developed in this chapter characterize the operators, their interrelation, and give hints on the expressive power of the query language. The high complexity of Opt, for instance, suggests that in query optimization special care should be taken in optimizing Opt expressions. We will address these needs in the subsequent study of SPARQL query optimization.

Chapter 4.

Algebraic SPARQL Query Optimization

Roy: "*What about optimization of SPARQL?*"
Jen: "*I would be interested in its relation to relational algebra, maybe we can transfer established optimization strategies.*"
Moss: "*Sounds good, but we should not forget to address the particularities of RDF and SPARQL.*"
Roy: "*Great, let's see what we can do there!*"

Query optimization has been a central topic in database research from the beginning and up to the present a variety of query optimization techniques for the relational context have been proposed, including (but not limited to) algebraic rewriting (e.g. [Hal75; SC75; Tod75]), statistics and cost-based query optimization (e.g. [ABC+76; MD88]), indices for fast data access (e.g. [BM72]), and semantic query optimization (e.g. [Kin81]). Standard optimization techniques like indexing and cost-based rewriting have found entrance in virtually every major relational database system and it is well-known that such techniques may reduce query evaluation time by orders of magnitude, ultimately making database systems utilizable in practical scenarios (see e.g. the experiments presented in [Hal75]).

One fundamental prerequisite to query optimization in the relational context is the study of equivalences over relational algebra (RA) expressions [Hal75; SC75; Tod75; GLR97]. When interpreted as rewriting rules and enhanced by a cost estimation function (or, alternatively, adequate heuristics), these equivalences allow to transform RA expressions into structurally different, but equivalent expressions that can be evaluated more efficiently than the original expressions. Established optimization techniques like filter pushing or join reordering, for instance, lastly rely on the knowledge of equivalences over RA expressions (cf. [Hal75; SC75; Tod75]).

Given the central role of query optimization in the relational context, it is natural to assume that the development of efficient evaluation approaches for SPARQL queries is an important step towards the realization of the Semantic Web, where engines must be able to deal efficiently with RDF repositories containing millions or even billions of RDF triples (see e.g. [BC07; Tau; lin] for existing large-scale

RDF databases). The complexity results for SPARQL evaluation established in the previous chapter indicate that the efficient processing of SPARQL queries is a non-trivial task and – given that years of research have been spent in the investigation of SQL and relational algebra optimization – seems to be an ambitious goal.

Addressing the issue of SPARQL optimization, over the last years various proposals for the efficient evaluation of SPARQL have been made. These approaches comprise a wide range of optimization techniques, including normal forms [PAG06a], graph pattern reordering based on selectivity estimations [NW08; SSB+08; NW09] (similar to relational join reordering), query graph models for SPARQL [HH07], RISC-style query processing [NW08], and semantic SPARQL optimization [SKCT05; LMS08]. In addition, there has been a corpus of research on specialized indices [HD05; GGL07; FB08] and storage schemes [ACKP01; BKvH02; HG03; TCK05; AMMH07; WKB08] for RDF, with the aim to provide efficient data access paths. Another notable line of research is the translation of SPARQL queries and RDF data into established data models like SQL [Cyg05; CDES05; CLJF06] or datalog [Pol07; AG08a], thus facilitating SPARQL evaluation with traditional engines, to exploit optimization techniques implemented in existing SQL (respectively, datalog) systems.

One interesting observation is that the majority of the "native" optimization proposals for SPARQL (i.e. those that do not rely on a mapping into the relational context or datalog) have a strong focus on optimizing SPARQL And-only queries and mostly disregard the optimization of queries involving operators like Union, Filter, or Opt (cf. [HD05; GGL07; FB08; NW08; SSB+08; WKB08; NW09]). The efficient evaluation of And-only queries (or And-connected blocks inside queries) is undoubtedly an important task in SPARQL evaluation, so the latter approaches form valuable groundwork for SPARQL optimizers. Still, a comprehensive optimization framework should also address the optimization of more involved SPARQL queries. To give evidence for this claim, the experimental studies in [SHK+08; SHLP09] reveal severe performance bottlenecks when evaluating more complex SPARQL queries, in particular queries involving the complex Opt operator, for both existing SPARQL engines and state-of-the-art mapping schemes from SPARQL into SQL.

We argue that – like in relational algebra, where the study of algebraic rewriting rules has facilitated the development of diverse optimization techniques – a study of SPARQL algebra (SA) rewritings will improve the understanding of the algebraic operators and alleviate the development of comprehensive optimization approaches for SPARQL. To date, however, surprisingly few fundamental work has been done in the context of SPARQL algebra (we will resume some initial results from [PAG06a; AG08a] in the course of this chapter). Based on all these considerations, we believe that a schematic investigation of SPARQL algebra is long overdue. Therefore, in this chapter we present an elaborate study of SA equivalences, covering all its operators and their interrelations. When interpreted as rewriting rules, these equivalences form the theoretical foundations for transferring established RA optimization techniques, such as projection and filter pushing, into the context of SPARQL optimization.

Going beyond the adaption of existing techniques, we also tackle SPARQL-specific issues, such as the simplification of expressions involving negation, which – when translating SPARQL expressions or queries into SA according to the semantics – manifests into a characteristic combination of operators $\rtimes$ and σ. Ultimately, our results improve the understanding of SPARQL algebra and lay the foundations for the development of advanced and comprehensive SPARQL optimization approaches.

Akin to the operators defined in RA, SPARQL algebra comprises operations such as join, union, left outer join, minus, projection, and selection, so at first glance there are many parallels between SA and RA. In fact, the study in [AG08a] reveals that SA and RA have exactly the same expressive power. In spite of all these similarities, existing mappings of SPARQL into relational algebra [Cyg05] or SQL [CLJF06] indicate that a semantics-preserving translation of SPARQL into the relational context is far from being trivial. This shows that, although both algebras provide similar operators, there are still fundamental differences between both. One of the most striking discrepancies, as also argued in [PAG06a], is that joins in RA are rejecting over null values, but in SA, where the schema is loose in the sense that mappings may bind an arbitrary set of variables, joins over unbound variables (essentially the counterpart of RA null values) are accepting. Let us exemplify the issue of unbound variables in SPARQL mappings by means of the candidate equivalence

$$\sigma_{?x=c}(A \bowtie B) \stackrel{?}{\equiv} \sigma_{?x=c}(A) \bowtie B, \tag{4.1}$$

where A and B are SPARQL algebra expressions. Whether or not this equivalence holds, depends on the variables that appear in expressions A and B. Let us discuss the most general case and assume that variable $?x$ appears in both A and B. If we could guarantee that variable $?x$ is bound in **every** result mapping obtained when evaluating expression A, then it would follow that the equivalence holds: informally speaking, in the latter case the join of A with B does not modify bindings for variable $?x$, so it does not matter whether the selection is applied early, i.e. on top of A, or late, on top of $A \bowtie B$. The problem, however, is that the appearance of $?x$ in A does not imply that $?x$ also is **bound** in each result mapping obtained when evaluating A. Consider for example the RDF database $D := \{(c, c, c)\}$ and the algebra expressions $A := [\![(c, c, ?y)]\!]_D$ Opt $[\![(d, d, ?x)]\!]_D$ and $B := [\![(c, c, ?x)]\!]_D$, where $?x$ appears in both the left and right side expression. It is easily verified that $[\![A]\!]_D = \{\{?y \mapsto c\}\}$ and $[\![B]\!]_D = \{\{?x \mapsto c\}\}$, so the left side of Equation (4.1) evaluates to $\{\{?x \mapsto c, ?y \mapsto c\}\}$, while the right side evaluates to $\emptyset$. Although $?x$ appears in A, it is unbound in $[\![A]\!]_D$, which causes the equivalence to fail.

Addressing the issue of unbound variables in result mappings and their importance for the study of SPARQL algebra equivalences, we propose the concepts of *certain* and *possible variables*: the certain variables of a SPARQL algebra expression are an underestimation for the set of variables that are bound in **every** result mapping (independently from the RDF input document), while the associated set of possible

variables constitutes an overestimation for the variables that **might** be bound in result mappings. Taken together, these two concepts capture the rationale behind unbound variables in SPARQL mappings and allow us to state equivalences in a compact and precise way. Coming back to the equivalence in Equation (4.1), for instance, we will show in Section 4.2.5 that the equivalence holds whenever $?x$ is a certain variables of expression A or $?x$ is **not** a possible variable of expression B.

We finally want to note that we investigate both the theoretically motivated SPARQL set algebra from Definition 2.10 and the SPARQL bag algebra from Definition 2.14, which reflects the approach that is proposed by the W3C [spac]. Our investigations reveal that there are indeed differences between set and bag semantics w.r.t. the rewriting rules that they exhibit and, in response, we both highlight equivalences for which the two semantics differ and identify a large fragment of SPARQL algebra for which both semantics coincide, allowing engines to fall back on the simpler set semantics for a broad class of SPARQL algebra expressions.

We summarize the major contributions of this chapter as follows.

- We develop the concepts of certain and possible variables, which account for the issue of unbound variables in SPARQL result mappings and allow us to state equivalences over SPARQL algebra in a clean and precise way.
- We both summarize existent and develop new equivalences over SPARQL algebra. In total, we discuss about forty rewriting rules for SA (of which almost three-fourths are new), covering all algebraic operators and their interaction.
- In our study of algebraic rewriting, we systematically address established rewriting strategies for the relational model, such as filter and projection pushing. This allows us to transfer established optimization strategies from relational algebra into the context of SPARQL query evaluation.
- Beyond the investigation of rewriting rules that have proven useful in the relational context, we develop equivalences that address SPARQL-specific optimization tasks. For instance, we present a rewriting scheme for queries involving negation, typically encountered in SPARQL algebra expressions as a characteristic combination of operators $\bowtie$, σ, and filter predicate *bnd*.
- We study interrelations between SPARQL set and bag algebra. Our analysis reveals that most (but not all) of the equivalences hold under both set and bag semantics. We also discuss the implications of our results for SPARQL engines that – following the official W3C proposal – implement the bag semantics.

Structure. We start with the concepts of certain and possible variables in Section 4.1. The subsequent study of algebraic rewriting divides into the study of equivalences for the SPARQL set algebra from Definition 2.10 (Section 4.2) and the investigation of rewriting rules for the SPARQL bag algebra from Definition 2.14 (Section 4.3). We summarize our results and discuss their implications for engines that implement the W3C SPARQL semantics in Section 4.4. The chapter ends with a discussion of related work in Section 4.5 and a short conclusion in Section 4.6.

4.1. Possible and Certain Variables

Before starting our investigation of SPARQL algebra, we introduce two functions to statically classify the variables that appear in some fixed SPARQL algebra expression A. The first one, function $pVars(A)$, gives an upper bound for the so-called *possible variables* of A, i.e. provides an overestimation for the set of variables that **might** be bound in result mappings. The second one, function $cVars(A)$, estimates the *certain variables* of A and fixes a lower bound for variables that are bound in **every** result mapping obtained when evaluating A. Note that both functions are independent from the input document. We first illustrate the concepts by example:

Example 4.1 Consider the SPARQL set algebra expression

$$A := [\![(a, a, ?u)]\!]_D \bowtie ([\![(a, ?v, ?w)]\!]_D \cup [\![(a, ?v, ?x)]\!]_D).$$

It is easy to see that, according to the semantics from Definition 2.10, when evaluating expression A on some document D, variable $?u$ will be bound in every result mapping; the same observation holds for $?v$, which is contained in both triple patterns of the union subexpression. Next, we observe that each result mapping binds either $?w$ or $?x$. Consequently, $?u$ and $?v$ are categorized as both possible and certain variables of A, while $?w$ and $?x$ are possible but not certain variables. □

The concepts of certain and possible variables account for the specifics of the SPARQL query language, in which – caused by the operators union $\cup$, left outer join $⟕$, and projection π – variables occurring in the expression may be unbound in (some) result mappings. As discussed in the introduction of this chapter, they will take a central role in our subsequent investigation of SPARQL algebra: the applicability of many rewritings ultimately depends on these two estimations, in particular the rules concerning projection pushing in Section 4.2.4 and filter manipulation in Section 4.2.5. We start with the estimation function for possible variables:

Definition 4.1 (Function *pVars*) Let A be a set algebra expression, $S \subset V$ a finite set of variables and R be a filter condition. We define function $pVars(A)$, which extracts so-called *possible variables*, inductively on the structure of A:

$$\begin{aligned}
pVars([\![t]\!]_D) &:= vars(t) \\
pVars(A_1 \bowtie A_2) &:= pVars(A_1) \cup pVars(A_2) \\
pVars(A_1 \cup A_2) &:= pVars(A_1) \cup pVars(A_2) \\
pVars(A_1 \setminus A_2) &:= pVars(A_1) \\
pVars(A_1 ⟕ A_2) &:= pVars(A_1) \cup pVars(A_2) \\
pVars(\pi_S(A_1)) &:= pVars(A_1) \cap S \\
pVars(\sigma_R(A_1)) &:= pVars(A_1)
\end{aligned}$$

□

The following proposition shows that possible variables always constitute a superset of the variables that appear in result mappings:

Proposition 4.1 Let A be a SPARQL set algebra expression and let Ω_A denote the mapping set obtained when evaluating expression A on any RDF document D. Then for all $\mu \in \Omega_A : ?x \in dom(\mu) \rightarrow ?x \in pVars(A)$. □

The proof of the proposition works by induction on the structure of SPARQL algebra expressions, the application of the definition of the algebraic operators, and the definition of function $pVars(A)$ above. We omit the technical details.

Complementarily to function $pVars(A)$, we propose an underestimation for the certain variables of an algebra expression, implemented through function $cVars(A)$:

Definition 4.2 (Function *cVars*) Let A be a SPARQL set algebra expression, $S \subset V$ a set of variables, and R a filter condition. We define function $cVars(A)$, which extracts so-called *certain variables*, inductively on the structure of A:

$$
\begin{array}{ll}
cVars([\![t]\!]_D) & := vars(t) \\
cVars(A_1 \bowtie A_2) & := cVars(A_1) \cup cVars(A_2) \\
cVars(A_1 \cup A_2) & := cVars(A_1) \cap cVars(A_2) \\
cVars(A_1 \setminus A_2) & := cVars(A_1) \\
cVars(A_1 \ltimes A_2) & := cVars(A_1) \\
cVars(\pi_S(A_1)) & := cVars(A_1) \cap S \\
cVars(\sigma_R(A_1)) & := cVars(A_1)
\end{array}
$$
□

The definition of $cVars(A)$ differs from $pVars(A)$ in two cases. First, for union expressions only those variables are certain that are certain for both subexpression. Second, for left outer join expressions, no guarantees for the right side variables can be made. The key property of certain variables can be formalized as follows.

Proposition 4.2 Let A be a SPARQL set algebra expression and let Ω_A denote the mapping set obtained when evaluating expression A on any RDF document D. It holds that $?x \in cVars(A) \rightarrow \forall \mu \in \Omega_A : ?x \in dom(\mu)$. □

Again, the proof of the proposition is straightforward and we omit the details. We next provide an example that illustrates the definitions of the two functions:

Example 4.2 Consider the SPARQL set algebra expression

$$
\begin{aligned}
A := \pi_{?u,?v,?x,?y}((([\![(?u, a, ?v)]\!]_D \bowtie [\![(?u, a, ?w)]\!]_D) \ltimes [\![(?u, a, ?x)]\!]_D) \cup \\
\sigma_{?x=1}([\![(?u, ?x, ?y)]\!]_D)).
\end{aligned}
$$

It is easily verified that $pVars(A) = \{?u, ?v, ?x, ?y\}$ and $cVars(A) = \{?u\}$. □

We conclude with the remark that possible (certain) variables are indeed only upper (lower) bounds for variables that might be (are always) bound in result mappings. For instance, we could further improve the definitions of $pVars(A)$ and $cVars(A)$ when catching some special cases of filter expressions. To be concrete, consider the two algebra expressions $B := \sigma_{bnd(?y)}([\![(a, b, ?x)]\!]_D \bowtie [\![(a, c, ?y)]\!]_D)$ and $B^{\neg} := \sigma_{\neg bnd(?y)}([\![(a, b, ?x)]\!]_D \bowtie [\![(a, c, ?y)]\!]_D)$. It is easy to see that $?y$ is a certain variable for B in the sense of Proposition 4.2 (i.e. it will be bound in every result mapping), but we observe that $?y \notin cVars(B)$. Analogously, $?y$ does never appear in result mappings when evaluating expression $B^{\neg}$, but $?y \in pVars(B^{\neg})$. It is straightforward to incorporate such conditions into Definitions 4.1 and 4.2. However, the analysis of such special cases does not bring further insights and we decided to ignore them in the interest of a compact definition for the two functions.

From Set to Bag Algebra. In Definitions 4.1 and 4.2 we introduced functions $pVars(A)$ and $cVars(A)$ over SPARQL set algebra expressions (cf. Definition 2.10). The definitions of the two functions can be easily adapted for bag algebra expressions (cf. Definition 2.14), by simply replacing the first rule in Definitions 4.1 and 4.2 by $pVars([\![t]\!]^+_D) := vars(t)$ and $cVars([\![t]\!]^+_D) := vars(t)$, respectively. Therefore, we shall overload the two functions and use them for both SPARQL set algebra and SPARQL bag algebra expressions in the remainder of this chapter. It is easy to see that, according to Lemma 3.1, Propositions 4.1 and 4.2 naturally carry over from set to bag algebra: for each SPARQL bag algebra expressions A^+ it holds that (i) the possible variables of A^+ are a superset of the variables that might appear in result mappings (more precisely, in the mapping set component of the result mapping multi-set) and (ii) certain variables of A^+ are bound in every result mapping.

Example 4.3 Consider the SPARQL bag algebra expression

$$A^+ := \pi_{?u,?v,?x,?y}((([\![(?u, a, ?v)]\!]^+_D \bowtie [\![(?u, a, ?w)]\!]^+_D) \bowtie [\![(?u, a, ?x)]\!]^+_D) \cup \sigma_{?x=1}([\![(?u, ?x, ?y)]\!]^+_D)).$$

We can observe that A^+ is structurally identical to the set algebra expression A from Example 4.2, so the possible and certain variables are exactly the same as for A, namely $pVars(A^+) = \{?u, ?v, ?x, ?y\}$ and $cVars(A^+) = \{?u\}$. □

4.2. Optimization Rules for Set Algebra

We start with the discussion of algebraic equivalences for SPARQL set algebra, covering all the algebraic operators introduced in Definition 2.10. In query optimization, such equivalences are typically interpreted as rewriting rules and therefore we shall use the terms *equivalence* and *(rewriting) rule* interchangeably in the following.

In the interest of a complete survey, we will include equivalences that have been stated before in [PAG06a].[1] To be concrete, most of the equivalences from Figures 4.1 and 4.2, equivalences *(FDecompI)*, *(FDecompII)*, and *(FUPush)* from Figure 4.4, as well as equivalence *(MJ)* stated in Proposition 4.4 are borrowed from [PAG06a] and listed for completeness only. Furthermore, $\widetilde{(JIdem)}$ in Figure 4.1, *(FJPush)* in Figure 4.4, and $\widetilde{(LJ)}$ in Lemma 4.4 are generalizations of Lemma (2), Lemma 1(2), and Lemma 3(3) in [PAG06a], respectively. These generalizations rely on the novel notion of incompatibility property and the fragment $\widetilde{\mathbb{A}}$ (which will be introduced in Section 4.2.1) and extend the applicability of the original rules. We conclude with the remark that in total almost three-fourths of the rules presented in this section are new. The subsequent discussion will focus on these newly-discovered rules.

In our investigation of rewriting rules for the set semantics we study two fragments of SPARQL set algebra. The first fragment of interest is the full class of SPARQL set algebra expressions. We call this fragment $\mathbb{A}$ and define it as follows.

Definition 4.3 (Fragment $\mathbb{A}$) The fragment $\mathbb{A}$ denotes the full class of SPARQL set algebra expressions, i.e. expression built using operators $\cup$, $\bowtie$, $\setminus$, $\leftouterjoin$, π, σ, and (bracket-enclosed) triple patterns of the form $[\![t]\!]_D$. □

We understand fragment $\mathbb{A}$ as a set of purely syntactic entities. Yet, according to the SPARQL set semantics in Definition 2.11, each expression $A \in \mathbb{A}$ implicitly defines a mapping set (given that document D is fixed). Therefore, when document D is known from the context, we will sometimes refer to the mapping set obtained by application of the semantics as the *result of evaluating A on D* and, abusing notation, shall write $\mu \in A$ for a mapping that is contained in this result.

4.2.1. The Incompatibility Property

In addition to the full fragment of SPARQL set algebra expressions $\mathbb{A}$ we introduce a subfragment of $\mathbb{A}$ that has a special property, called *incompatibility property*. As we shall see, expressions that satisfy the incompatibility property exhibit some rewriting rules that do not hold in the general case and therefore will be of particular interest.

Definition 4.4 (Incompatibility Property for Set Algebra) A SPARQL set algebra expression A has the *incompatibility property* if, for every document D and each two distinct mappings $\mu_1 \neq \mu_2$ contained in the result of evaluating A on D, it holds that $\mu_1 \not\sim \mu_2$. □

[1] Most equivalences in [PAG06a] were established at the syntactic level, while we study optimization at the algebraic level. Still, according to Definition 2.11 there is essentially a one-to-one correspondence between abstract syntax operators and algebraic operators.

We next define a large fragment $\widetilde{\mathbb{A}} \subset \mathbb{A}$ of SPARQL set algebra, which comprises only expressions that satisfy the incompatibility property. The fragment is defined inductively on the structure of expressions and can be checked efficiently:

Definition 4.5 (Fragment $\widetilde{\mathbb{A}}$) We define the fragment $\widetilde{\mathbb{A}} \subset \mathbb{A}$ inductively on the structure of $\mathbb{A}$ expressions. An expression $\widetilde{A} \in \mathbb{A}$ is an $\widetilde{\mathbb{A}}$ expression if

- $\widetilde{A} := [\![t]\!]_D$ is a triple pattern,
- $\widetilde{A} := \widetilde{A_1} \bowtie \widetilde{A_2}$, where $\widetilde{A_1}$ and $\widetilde{A_2}$ are $\widetilde{\mathbb{A}}$ expressions,
- $\widetilde{A} := \widetilde{A_1} \setminus A_2$, where $\widetilde{A_1}$ is an $\widetilde{\mathbb{A}}$ expression and $A_2 \in \mathbb{A}$,
- $\widetilde{A} := \widetilde{A_1} \mathbin{⟕} \widetilde{A_2}$, where $\widetilde{A_1}$ and $\widetilde{A_2}$ are $\widetilde{\mathbb{A}}$ expressions,
- $\widetilde{A} := \sigma_R(\widetilde{A_1})$, where R is a filter condition and $\widetilde{A_1} \in \widetilde{\mathbb{A}}$,
- $\widetilde{A} := \pi_S(\widetilde{A_1})$, where S is a set of variables, $\widetilde{A_1} \in \widetilde{\mathbb{A}}$, and $S \supseteq pVars(\widetilde{A_1})$ or $S \subseteq cVars(\widetilde{A_1})$, or
- $\widetilde{A} := \widetilde{A_1} \cup \widetilde{A_2}$, where $\widetilde{A_1}$ $\widetilde{A_2}$ are $\widetilde{\mathbb{A}}$ expressions and $pVars(\widetilde{A_1}){=}cVars(\widetilde{A_1}){=}pVars(\widetilde{A_2}){=}cVars(\widetilde{A_2})$. □

Lemma 4.1 Every expression $\widetilde{A} \in \widetilde{\mathbb{A}}$ has the incompatibility property. □

Proof of Lemma 4.1

We prove the lemma by induction on the structure of $\widetilde{\mathbb{A}}$ expressions, thereby exploiting the structural constraints imposed by Definition 4.5. The basic case is $\widetilde{A} := [\![t]\!]_D$. By semantics (see Definition 2.11), all mappings in the result then bind exactly the same set of variables, and consequently the values of each two distinct mappings must differ in at least one variable, which makes them incompatible. We assume that every $\widetilde{A} \in \widetilde{\mathbb{A}}$ has the incompatibility property and distinguish six cases.

(1) Consider an expression $\widetilde{A} := \widetilde{A_1} \bowtie \widetilde{A_2}$. By Definition 4.5, both $\widetilde{A_1}$, $\widetilde{A_2}$ are $\widetilde{\mathbb{A}}$ expressions and by induction hypothesis both have the incompatibility property. We observe that each mapping $\mu \in \widetilde{A}$ is of the form $\mu = \mu_1 \cup \mu_2$ with $\mu_1 \in \widetilde{A_1}$, $\mu_2 \in \widetilde{A_2}$, and $\mu_1 \sim \mu_2$ (by semantics of $\bowtie$). We fix μ and show that each mapping $\mu' \in \widetilde{A}$ that is distinct from μ is incompatible. Any distinct mapping $\mu' \in \widetilde{A}$ is of the form $\mu'_1 \cup \mu'_2$ with $\mu'_1 \in \widetilde{A_1}$, $\mu'_2 \in \widetilde{A_2}$, and it holds that μ'_1 is different from μ_1 or that μ'_2 is different from μ_2 (because μ is distinct from μ'). Let us w.l.o.g. assume that μ'_1 is different from μ_1. We know that $\widetilde{A_1} \in \widetilde{\mathbb{A}}$, so it holds that μ_1 is incompatible with μ'_1. It follows that $\mu = \mu_1 \cup \mu_2$ is incompatible with $\mu' = \mu'_1 \cup \mu'_2$, since μ_1 and μ'_1 disagree in the value of at least one variable. (2) Let $\widetilde{A} := \widetilde{A_1} \setminus \widetilde{A_2}$ where $\widetilde{A_1} \in \widetilde{\mathbb{A}}$, so each two distinct mappings in $\widetilde{A_1}$ are pairwise incompatible by induction hypothesis. By semantics of $\setminus$, $\widetilde{A}$ is a subset of $\widetilde{A_1}$, so the incompatibility property trivially holds for $\widetilde{A}$. (3) Let $\widetilde{A} := \widetilde{A_1} \mathbin{⟕} \widetilde{A_2}$, where both $\widetilde{A_1}$ and $\widetilde{A_2}$ are $\widetilde{\mathbb{A}}$ expressions. We rewrite the left outer join according to its semantics: $\widetilde{A} = \widetilde{A_1} \mathbin{⟕} \widetilde{A_2} = (\widetilde{A_1} \bowtie \widetilde{A_2}) \cup (\widetilde{A_1} \setminus \widetilde{A_2})$.

Following the argumentation in cases (1) and (2), the incompatibility property holds for both subexpressions $\widetilde{A}_{\bowtie} := \widetilde{A_1} \bowtie \widetilde{A_2}$ and $\widetilde{A}_{\setminus} := \widetilde{A_1} \setminus \widetilde{A_2}$, so it suffices to show that the mappings in $\widetilde{A}_{\bowtie}$ are pairwise incompatible to those in $\widetilde{A}_{\setminus}$. First note that $\widetilde{A}_{\setminus}$ is a subset of $\widetilde{A_1}$. Further, by semantics each mapping $\mu \in \widetilde{A}_{\bowtie}$ is of the form $\mu = \mu_1 \cup \mu_2$, where $\mu_1 \in \widetilde{A_1}$, $\mu_2 \in \widetilde{A_2}$, and $\mu_1 \sim \mu_2$. Applying the induction hypothesis, we conclude that each mapping in $\widetilde{A_1}$ and hence each mapping $\mu_1' \in \widetilde{A}_{\setminus}$ is (3a) either incompatible with μ_1 or (3b) identical to μ_1. (3a) If μ_1' is incompatible with μ_1, then it follows that μ_1' is incompatible with $\mu_1 \cup \mu_2 = \mu$ and we are done. (3b) Let $\mu_1 = \mu_1'$. By assumption, mapping μ_2 (which is generated by $\widetilde{A_2}$) is compatible with $\mu_1 = \mu_1'$. We conclude that $\widetilde{A_1} \setminus \widetilde{A_2}$ does not generate μ_1', which is a contradiction (i.e., assumption (3b) was invalid). (4) Let $\widetilde{A} := \sigma_R(\widetilde{A_1})$, where $\widetilde{A_1} \in \widetilde{\mathbb{A}}$. By semantics of σ, $\widetilde{A}$ is a subset of $\widetilde{A_1}$, so the property trivially follows by application of the induction hypothesis (5) Let $\widetilde{A} := \pi_S(\widetilde{A_1})$, where $\widetilde{A_1} \in \widetilde{\mathbb{A}}$ and by Definition 4.5 it holds that (5a) $S \supseteq pVars(\widetilde{A_1})$ or (5b) $S \subseteq cVars(\widetilde{A_1})$. (5a) If $S \supseteq pVars(\widetilde{A_1})$ then, according to Proposition 4.1, the projection maintains all variables that might occur in result mappings, so $\widetilde{A}$ is equivalent to $\widetilde{A_1}$. The claim then follows by induction hypothesis. Concerning case (5b) $S \subseteq cVars(\widetilde{A_1})$ it follows from Proposition 4.2 that each result mapping produced by expression $\widetilde{A}$ binds **all** variables in $S \subseteq cVars(\widetilde{A_1})$, and consequently all result mappings bind exactly the same set of variables. Recalling that we assume set semantics, we conclude that two distinct mappings must differ in the value of at least one variable, which makes them incompatible. (6) Let $\widetilde{A} := \widetilde{A_1} \cup \widetilde{A_2}$, where $\widetilde{A_1}$, $\widetilde{A_2}$ are $\widetilde{\mathbb{A}}$ expressions and it holds that $pVars(\widetilde{A_1}) = cVars(\widetilde{A_1}) = pVars(\widetilde{A_2}) = cVars(\widetilde{A_2})$. From Propositions 4.1 and 4.2 it follows that each two mappings that are generated by $\widetilde{A_1} \cup \widetilde{A_2}$ bind exactly the same set of variables. Analogical to the argumentation in case (5b), two distinct mappings then disagree in the value of at least one variable, which makes them incompatible. □

We want to note that the property of mappings in the result being pairwise incompatible has been used before in the proof of Lemma 2 in [PAG06a]. There, it was shown at syntax level that UNION-free SPARQL expressions always exhibit the incompatibility property.[2] We emphasize that Lemma 4.1 above strictly generalizes this case, i.e. it is easily shown that algebra expressions derived from UNION-free expressions are always $\widetilde{\mathbb{A}}$ expressions. More precisely, algebra expressions built using only operators $\bowtie$, $\setminus$, $\ltimes$, and σ (i.e., those that may appear when translating UNION-free SPARQL expressions into algebra) are a **proper** subclass of $\widetilde{\mathbb{A}}$, so the incompatibility property always holds for such expressions. The following example illustrates that expressions involving the remaining two operators, namely projection π and union $\cup$, do not exhibit the incompatibility property in the general case.

[2]The property was not called "incompatibility property" there and was only used for proving a single equivalence. We will apply it in different contexts, which justifies its prominent role.

I. Algebraic Equivalences: Idempotence and Inverse

$A \cup A \equiv A$	*(UIdem)*
$\widetilde{A} \bowtie \widetilde{A} \equiv \widetilde{A}$	$(\widetilde{JIdem})$
$\widetilde{A} \between \widetilde{A} \equiv \widetilde{A}$	$(\widetilde{LIdem})$
$A \setminus A \equiv \emptyset$	*(Inv)*

Figure 4.1.: Algebraic equivalences: idempotence and inverse. Expression A stands for an $\mathbb{A}$ expression and $\widetilde{A}$ stands for an $\widetilde{\mathbb{A}}$ expression.

Example 4.4 Let $D := \{(0, f, 0), (1, t, 1), (a, tv, 0), (a, tv, 1)\}$ be an RDF database. When evaluating the two algebra expressions

$A_1 := [\![(0, f, ?x)]\!]_D \cup [\![(1, t, ?y)]\!]_D$ and
$A_2 := \pi_{?x,?y}(([\![(a, tv, ?z)]\!]_D \between [\![(?z, f, ?x)]\!]_D) \between [\![(?z, t, ?y)]\!]_D)$

on D we obtain the mapping set $\Omega = \{\{?x \mapsto 0\}, \{?y \mapsto 1\}\}$ for both. The two mappings in Ω are compatible. Note that neither A_1 nor A_2 are $\widetilde{\mathbb{A}}$ expressions. □

4.2.2. Idempotence and Inverse

We start our investigation of SPARQL set algebra equivalences with some very basic rules that hold with respect to idempotence and inverse algebraic laws in Figure 4.1, where A stands for an $\mathbb{A}$ expression and $\widetilde{A}$ represents an $\widetilde{\mathbb{A}}$ expression. Following common notation, we write $A \equiv B$ if SPARQL algebra expression A is equivalent to B on every document D. As a notational convention, we distinguish equivalences that specifically hold for fragment $\widetilde{\mathbb{A}}$ by a tilde symbol, e.g. writing $(\widetilde{JIdem})$ for the idempotence of the join operator over expressions in class $\widetilde{\mathbb{A}}$.

The rules *(UIdem)* and *(Inv)* in Figure 4.1 hold for the whole fragment $\mathbb{A}$. They follow trivially from the definition of the algebraic operators $\cup$ and $\setminus$. More interesting are rules $(\widetilde{JIdem})$ and $(\widetilde{LIdem})$, established for fragment $\widetilde{\mathbb{A}}$. We next present the proofs for these rules; observe that both exploit the incompatibility property.

Proof of Equivalences $(\widetilde{JIdem})$ and $(\widetilde{LIdem})$ from Figure 4.1

$(\widetilde{JIdem})$. Let $\widetilde{A}$ be an $\widetilde{\mathbb{A}}$ expression. We show that both directions of the equivalence hold. $\Rightarrow$: Consider a mapping $\mu \in \widetilde{A} \bowtie \widetilde{A}$. Then $\mu = \mu_1 \cup \mu_2$ where $\mu_1, \mu_2 \in \widetilde{A}$ and $\mu_1 \sim \mu_2$. From Lemma 4.1 we know that each $\widetilde{\mathbb{A}}$ expression has the incompatibility property, so each pair of distinct mappings in $\widetilde{A}$ is incompatible. It follows that $\mu_1 = \mu_2$ and, consequently, $\mu_1 \cup \mu_2 = \mu_1$, which is generated by $\widetilde{A}$, and hence by the right side expression. $\Leftarrow$: Consider a mapping $\mu \in \widetilde{A}$. Choose μ for both the left and right expression in $\widetilde{A} \bowtie \widetilde{A}$. By assumption, $\mu \cup \mu = \mu$ is contained in the left side expression of the equation, which completes the proof.

(LIdem). Let $\widetilde{A} \in \widetilde{\mathbb{A}}$. The following rewriting proves the equivalence.

$$\begin{aligned}
\widetilde{A} \ltimes \widetilde{A} &= (\widetilde{A} \bowtie \widetilde{A}) \cup (\widetilde{A} \setminus \widetilde{A}) && \text{[semantics]} \\
&= (\widetilde{A} \bowtie \widetilde{A}) \cup \emptyset && \text{[(Inv)]} \\
&= \widetilde{A} \bowtie \widetilde{A} && \text{[semantics]} \\
&= \widetilde{A} && \text{[(JIdem)]} \square
\end{aligned}$$

The observation that both proofs exploit the incompatibility property suggests (yet does not logically imply) that equivalences *(JIdem)* and *(LIdem)* do not hold for fragments that do not satisfy the incompatibility property. The following proposition clarifies this issue, showing that the two equivalences do generally not hold for expressions involving operators $\cup$ and π (i.e., when relaxing the restrictions for operators π and $\cup$ imposed on $\widetilde{\mathbb{A}}$ expressions in the last two bullets of Definition 4.5).

Proposition 4.3 The two equivalences *(JIdem)* and *(LIdem)* do not hold for fragments involving operator $\cup$ and/or π in the general case. $\square$

Proof of Proposition 4.3

We argue that the RDF database D and the SPARQL algebra expressions A_1, A_2 from Example 4.4 constitute counterexamples for both operator $\cup$ and π. First note that A_1 contains operator $\cup$ and A_2 contains operator π (and recall that neither A_1 nor A_2 are $\widetilde{\mathbb{A}}$ expressions). We now show that, for A_1 and A_2, neither rule *(JIdem)* nor rule *(LIdem)* does hold. As discussed in Example 4.4 the result of evaluating A_1 and A_2 on D is $\Omega := \{\{?x \mapsto 0\}, \{?y \mapsto 1\}\}$. It is easy to verify that $A_1 \bowtie A_1 = A_1 \ltimes A_1 = A_2 \bowtie A_2 = A_2 \ltimes A_2 = \{\{?x \mapsto 0\}, \{?y \mapsto 1\}, \{?x \mapsto 0, ?y \mapsto 1\}\}$. Obviously, this result differs from Ω, which proves Proposition 4.3. $\square$

4.2.3. Associativity, Commutativity, and Distributivity

We survey associativity, commutativity, and distributivity in Figure 4.2. The rules speak for themselves and do not require further explanation. One outstanding question is whether the rules for the algebraic laws listed in Figure 4.2 are complete w.r.t. possible operator combinations. The following lemma answers this question:

Lemma 4.2 Let $O_1 := \{\bowtie, \setminus, \ltimes\}$ and $O_2 := O_1 \cup \{\cup\}$ be sets of operators.

1. Associativity and commutativity do not hold for operators $\setminus$ and $\ltimes$.
2. Neither $\setminus$ nor $\ltimes$ are left-distributive over $\cup$.
3. Let $o_1 \in O_1$, $o_2 \in O_2$, and $o_1 \neq o_2$. Then operator o_2 is neither left- nor right-distributive over operator o_1. $\square$

II. Algebraic Equivalences: Associativity		
$(A_1 \cup A_2) \cup A_3$	$\equiv A_1 \cup (A_2 \cup A_3)$	*(UAss)*
$(A_1 \bowtie A_2) \bowtie A_3$	$\equiv A_1 \bowtie (A_2 \bowtie A_3)$	*(JAss)*
III. Algebraic Equivalences: Commutativity		
$A_1 \cup A_2$	$\equiv A_2 \cup A_1$	*(UComm)*
$A_1 \bowtie A_2$	$\equiv A_2 \bowtie A_1$	*(JComm)*
IV. Algebraic Equivalences: Distributivity		
$(A_1 \cup A_2) \bowtie A_3$	$\equiv (A_1 \bowtie A_3) \cup (A_2 \bowtie A_3)$	*(JUDistR)*
$A_1 \bowtie (A_2 \cup A_3)$	$\equiv (A_1 \bowtie A_2) \cup (A_1 \bowtie A_3)$	*(JUDistL)*
$(A_1 \cup A_2) \setminus A_3$	$\equiv (A_1 \setminus A_3) \cup (A_2 \setminus A_3)$	*(MUDistR)*
$(A_1 \cup A_2) \overline{\bowtie} A_3$	$\equiv (A_1 \overline{\bowtie} A_3) \cup (A_2 \overline{\bowtie} A_3)$	*(LUDistR)*

Figure 4.2.: Algebraic equivalences: associativity, commutativity, and distributitvity. Expressions A_1, A_2, A_3 stand for $\mathbb{A}$ expressions.

The proof in Appendix C.1 is by an exhaustive listing of counterexamples. It is easily verified that the lemma rules out associativity, commutativity, and distributivity for **all** operator combinations different from those listed in Figure 4.2. Amongst others, the lemma shows that $\overline{\bowtie}$ is not left-distributive over $\cup$. This implies that Proposition 1(3) from [PAG06a], stating that operator Opt is left-distributive over Union, is actually wrong. We prove this claim in the following example.

Example 4.5 We show that the SPARQL expression equivalence

$$A_1 \text{ Opt } (A_2 \text{ Union } A_3) \equiv (A_1 \text{ Opt } A_2) \text{ Union } (A_1 \text{ Opt } A_3)$$

stated in Proposition 1(3) in [PAG06a] does not hold in the general case. As a counterexample, we choose the database $D := \{(0, c, 1)\}$ and set $A_1 := (0, c, ?x)$, $A_2 := (?x, c, 1)$, and $A_3 := (0, c, ?y)$. It is easily verified that

$[\![A_1 \text{ Opt } (A_2 \text{ Union } A_3)]\!]_D = \{\{?x \mapsto 1, ?y \mapsto 1\}\}$, but
$[\![(A_1 \text{ Opt } A_2) \text{ Union } (A_1 \text{ Opt } A_3)]\!]_D = \{\{?x \mapsto 1\}, \{?x \mapsto 1, ?y \mapsto 1\}\}$.

Obviously, the left and right side evaluation results differ. □

Remark 4.1 The union normal form proposed in Section 2.3 in [PAG06a] builds upon this invalid equivalence, so this finding calls the existence of such a normal form into question. In response to our observation, the authors published a corrected version of the proof in the journal version [PAG09], proposing a rewriting scheme to bring expressions into union normal form without using the invalid equivalence. □

4.2.4. Projection Pushing

We will next introduce a set of rules that allow us to manipulate expressions involving the algebraic projection operator π, with focus on pushing down projection operators in the operator tree. We start with the observation that, according to the definition of SPARQL queries (Definition 2.5), projection initially occurs only at the top-level, in form of a SELECT clause. Hence, when translating queries into algebra, there is exactly one projection operator π and this operator stands on top of the whole algebra expression (the same observation holds for the W3C SPARQL Recommendation [spac]). Arguably, a good optimization scheme should include the possibility to choose among evaluation plans where projection is applied at different positions in the operator tree. It is well-known from SQL optimizers that early projection may serve as a filter that can significantly decrease the size of intermediate results, thus speeding up subsequent computations. We expect similar benefits for SPARQL query processing and underline this claim by the following example.

Example 4.6 Consider the algebra expression B_1 below, which selects URIs and names of persons that know at least one other person:

$$B_1 := \pi_{?person,?name}([\![(?person, name, ?name)]\!]_D \bowtie [\![(?person, knows, ?person2)]\!]_D).$$

When evaluating this query strictly according to the semantics, we would first compute the join between the result of the two inner triple patterns and afterwards project for variables *?person* and *?name*. Let us assume that the number of *knows* relationships is very high (assume, for instance, a social networking site), so the second triple pattern would extract a large number of mappings, and many of them may share the value for variable *?person* (i.e., whenever a fixed person knows more than one person). In that case, it would be desirable to project for *?person* in the evaluation result of the right pattern **before** joining with the left pattern, to reduce the cost of the subsequent join. Such an early projection is of particular benefit when the triples containing predicate *knows* are sorted on disk by the subject: then the projection can be carried out in linear time (even on-the-fly while extracting the bindings). In fact, state-of-the-art RDF processing systems like the *Hexastore* [WKB08] system or the Vertical Partitioning scheme proposed in [AMMH07] support sorted extraction of data, so this situation is not unrealistic.

As we will see later, SPARQL algebra expression B_1 is equivalent to the expression

$$B_1^{opt} := [\![(?person, name, ?name)]\!]_D \bowtie \pi_{?person}([\![(?person, knows, ?person2)]\!]_D),$$

which – under the assumptions discussed before – constitutes a more efficient query evaluation plan. The algebraic equivalences that we will present in the following can be used to transform the two expressions B_1 and B_1^{opt} into each other. □

V. Algebraic Equivalences: Projection Pushing

$$\pi_{pVars(A)\cup S}(A) \equiv A \qquad \textit{(PBaseI)}$$
$$\pi_S(A) \equiv \pi_{S\cap pVars(A)}(A) \qquad \textit{(PBaseII)}$$

We define the shortcuts $S' := S \cup (pVars(A_1) \cap pVars(A_2))$ and $S'' := pVars(A_1) \cap pVars(A_2)$. The following equivalences hold.

$$\pi_S(A_1 \cup A_2) \equiv \pi_S(A_1) \cup \pi_S(A_2) \qquad \textit{(PUPush)}$$
$$\pi_S(A_1 \bowtie A_2) \equiv \pi_S(\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)) \qquad \textit{(PJPush)}$$
$$\pi_S(A_1 \setminus A_2) \equiv \pi_S(\pi_{S'}(A_1) \setminus \pi_{S''}(A_2)) \qquad \textit{(PMPush)}$$
$$\pi_S(A_1 \ltimes A_2) \equiv \pi_S(\pi_{S'}(A_1) \ltimes \pi_{S'}(A_2)) \qquad \textit{(PLPush)}$$
$$\pi_S(\sigma_R(A)) \equiv \pi_S(\sigma_R(\pi_{S\cup vars(R)}(A))) \qquad \textit{(PFPush)}$$
$$\pi_{S_1}(\pi_{S_2}(A)) \equiv \pi_{S_1\cap S_2}(A) \qquad \textit{(PMerge)}$$

Figure 4.3.: Algebraic equivalences: projection pushing. A, A_1, A_2 stand for $\mathbb{A}$ expressions and $S, S_1, S_2 \subset V$ denote sets of variables.

Figure 4.3 presents equivalences to simplify, eliminate, and push down (or, alternatively, pull out) projection operators. We will present the proof for rule *(PJPush)* later in this section and refer the reader to Appendix C.2 for the remaining proofs.

The first two equivalences in Figure 4.3, *(PBaseI)* and *(PBaseII)*, establish general-purpose rewriting rules for projection expressions. Firstly, *(PBaseI)* shows that, when projecting a variable set that contains all possible variables (and possibly some additional variables S), the projection can simply be dropped. The second equivalence, *(PBaseII)*, complements *(PBaseI)* by showing that all variables in S that do not belong to the possible variables of A can be dropped when projecting S. Intuitively, one might argue that these two equivalences are irrelevant for query rewriting in practice, given that users typically do not write queries that project variables that cannot be bound in the inner expression. We therefore want to emphasize that the benefit of these two equivalences stems from a combination with the remaining equivalences, which may introduce such redundant variables within the rewriting process. We will clarify this observation later in Example 4.7.

The remaining six rules in Figure 4.3 address the issue of pushing down projection expressions. The simplest rule among them is *(PUPush)*. It shows that, in case of the union operator $\cup$, projection can be independently applied to the two subexpressions instead of the whole expression. Equivalence *(PJPush)* for join expressions is more involved. It relies on the observation that, when pushing projections inside join subexpressions, we must keep variables that may occur in both subexpressions, because such variables may cause incompatibility between mappings and affect the result of the inner join, thus changing the semantics of the whole expression. To this end, we define $S' := S \cup (pVars(A_1) \cap pVars(A_2))$ as an extension of S and

project the variables in S' in the two subexpressions. Note that, due to the fact that $S' \supseteq S$, we cannot eliminate the topmost projection in the general case. Yet, as we will show in Example 4.7, a subsequent application of rules *(PBaseI)* and *(PBaseII)* may help to eliminate the topmost projection in cases where it became redundant. The subsequent proof of rule *(PJPush)* further clarifies the above considerations.

Proof of Equivalence *(PJPush)* from Figure 4.3

$\Rightarrow$: We show *(Claim1)* that, for each mapping μ that is generated by the left side subexpression $A_1 \bowtie A_2$, there is a mapping μ' generated by the right side subexpression $\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)$ s.t. for all $?x \in S$ either $?x \in dom(\mu) \cap dom(\mu')$ and $\mu(?x) = \mu'(?x)$ holds or $?x$ is unbound in both μ and μ'. It is easy to see that, if this claim holds, then the right side generates all mappings that are generated by the left side: the mapping that is generated by the left side expression of the equation is obtained from μ when projecting variables in S, and this mapping is also generated by the right side expression when projecting S in μ'. So let us consider a mapping $\mu \in A_1 \bowtie A_2$. By the semantics of $\bowtie$, mapping μ is of the form $\mu := \mu_1 \cup \mu_2$, where $\mu_1 \in A_1$, $\mu_2 \in A_2$, and $\mu_1 \sim \mu_2$ holds. We observe that, on the right side, $\pi_{S'}(A_1)$ then generates a mapping $\mu'_1 \subseteq \mu_1$, obtained from $\mu_1 \in A_1$ by projecting S'; similarly $\pi_{S'}(A_2)$ generates a mapping $\mu'_2 \subseteq \mu_2$, obtained from μ_2 by projecting S'. Then μ'_1 (μ'_2) agrees with μ_1 (μ_2) on variables in S (where "agrees" means that they either map the variable to the same value or the variable is unbound in both mappings), because $S' \supseteq S$ holds and therefore no variables in S are projected away when computing $\pi_{S'}(A_1)$ and $\pi_{S'}(A_2)$. It is easy to see that $\mu_1 \sim \mu_2$ implies $\mu'_1 \sim \mu'_2$, so the right side expression $\mu_{S'}(A_1) \bowtie \mu_{S'}(A_2)$ generates $\mu' := \mu'_1 \cup \mu'_2$. From the observation that μ_1 (μ_2) agrees with mapping μ'_1 (μ'_2) on all variables in S it follows that μ' agrees with μ on all variables in S and we conclude that *(Claim1)* holds.

$\Leftarrow$: We show *(Claim2)* that, for each mapping μ' that is generated by the right side subexpression $\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)$ there is a mapping $\mu \in A_1 \bowtie A_2$ such that for all $?x \in S$ either $?x \in dom(\mu) \cap dom(\mu')$ and $\mu(?x) = \mu'(?x)$ holds or $?x$ is unbound in both μ and μ'. Analogously to the other direction, it then follows immediately that all mappings generated by the right side also are generated by the left side of the equation. So let us consider a mapping $\mu' \in \pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)$. Then μ' is of the form $\mu' := \mu'_1 \cup \mu'_2$, where $\mu'_1 \in \pi_{S'}(A_1)$, $\mu'_2 \in \pi_{S'}(A_2)$, and $\mu'_1 \sim \mu'_2$ holds. Assume that μ'_1 is obtained from mapping $\mu_1 \in A_1$ by projecting S', and similarly assume that μ'_2 is obtained from $\mu_2 \in A_2$ by projecting S'. We distinguish two cases: (a) if μ_1 and μ_2 are compatible, then $\mu := \mu_1 \cup \mu_2$ is the desired mapping that agrees with $\mu' := \mu'_1 \cup \mu'_2$ on variables in S, because $\mu_1 \supseteq \mu'_1$ and $\mu_2 \supseteq \mu'_2$ holds. Otherwise, (b) if μ_1 and μ_2 are incompatible this means there is a variable $?x \in dom(\mu_1) \cap dom(\mu_2)$ such that $\mu_1(?x) \neq \mu_2(?x)$. From Proposition 4.1 we know that $?x \in pVars(A_1) \cap pVars(A_2)$, which implies that $?x \in S' \supseteq pVars(A_1) \cap pVars(A_2)$. Hence, $?x$ is bound in μ'_1 and in μ'_2 and it follows that $\mu'_1(?x) \neq \mu'_2(?x)$, which contradicts the assumption that $\mu'_1 \sim \mu'_2$ (i.e., assumption (b) was invalid). This completes the proof. $\square$

Having discussed rule *(PJPush)*, we resume the discussion of the equivalences from Figure 4.3. Equivalences *(PMPush)* and *(PLPush)* rely on similar ideas as *(PJPush)* and cover projection pushing for minus and left outer join expressions. Finally, *(PFPush)* covers projection pushing into filter expressions and *(PMerge)* shows that nested projection expressions can always be merged into a single projection. We conclude with a small example, which shall also serve as a proof of concept:

Example 4.7 We use the rules from Figure 4.3 to transform the algebra expression B_1 from Example 4.6 into its optimized version B_1^{opt}. We start with application of rule *(PJPush)*, to push projection down inside the join expression and obtain

$$B_1' := \pi_{?person,?name}(\, \pi_{?person,?name}([\![(?person, name, ?name)]\!]_D) \bowtie \pi_{?person,?name}([\![(?person, knows, ?person2)]\!]_D))$$

We observe that, according to equivalence *(PBaseI)*, the left inner projection expression can simply be dropped. Further, by application of rule *(PBaseII)* we can drop *?name* from the second inner projection. This gives us the expression

$$B_1'' := \pi_{?person,?name}(\, [\![(?person, name, ?name)]\!]_D \bowtie \pi_{?person}([\![(?person, knows, ?person2)]\!]_D))$$

Finally, we again apply *(PBaseI)* to eliminate the outer projection, which gives us the optimized version B_1^{opt} from Example 4.6. We point out that the application of rules *(PBaseI)* and *(PBaseII)* helped us to eliminate redundant filters introduced during the rewriting process and to refine the second filter expression. □

4.2.5. Filter Decomposition, Elimination, and Pushing

Figure 4.4 presents equivalences to decompose, eliminate, and rearrange (parts of) filter conditions. When interpreted as rewriting rules, they form the basis for transferring established relational algebra filter pushing techniques into the context of SPARQL. We emphasize that these rules are more than simple translations of existing relational algebra equivalences: firstly, they rely on the specifics of SPARQL and build upon the concepts of possible and certain variables from Section 4.1; secondly, they address specifics of SPARQL algebra, such as filter predicate *bnd* (cf. equivalences *(FBndI)-(FBndIV)*). In the following, we will discuss all rules and sketch their possible interplay. Missing proofs can be found in Appendix C.3.

The first three equivalences in group `VI` cover decomposition and reordering of filter conditions. More precisely, *(FDecompI)* and *(FDecompII)* allow to decompose filter conditions that are connected using the boolean connectives $\wedge$ and $\vee$, while *(FReord)* shows that the application order of nested filters does not matter (this follows from equivalence *(FDecompI)* and the commutativity of $\wedge$). When combined,

they allow to split complex filter conditions, which may be useful to push individual components down in the operator tree using the filter pushing rules in group `VII`.

The subsequent four equivalences *(FBndI)-(FBndIV)* are SPARQL-specific and address the issue of filter predicate *bnd*. They cover cases where predicate *bnd* is used in either a redundant or contradicting way. In some sense, these rules reflect the intuition behind the concepts of possible and certain variables. To give an example, precondition $?x \in cVars(A_1)$ in rule *(BndI)* implies that $?x$ is bound in each result mapping (by Proposition 4.2), so the filter is redundant and can be dropped.

Finally, the rules in group `VII` cover the issue of filter pushing. While the two equivalences *(FUPush)* and *(FMPush)* hold independently from the variables that are used in the filter conditions, the equivalences *(FJPush)* and *(FLPush)* heavily rely on the notions of safe and certain variables. According to these rules, the filter can be pushed inside the first component of a join $A_1 \bowtie A_2$ (respectively, left outer join $A_1 ⟕ A_2$) if each variable that is used inside the filter expression is a certain variable of A_1 (i.e., definitely bound in left side mappings) or is **not** a possible variable of A_2 (i.e., definitely **not** bound in right side mappings). Informally speaking, these conditions guarantee that the join (respectively, left outer join) does not affect the validity of the filter condition. In the generarl case, the equivalences do not hold if this precondition is violated, as illustrated by the next example:

Example 4.8 Consider the algebra expressions $A_1 := [\![(?x, c, c)]\!]_D ⟕ [\![(?x, d, ?y)]\!]_D$, $A_2 := [\![(?y, c, c)]\!]_D$, and the expressions

$$A_l^{\bowtie} := \sigma_{?y=c}(A_1 \bowtie A_2) \quad \text{vs.} \quad A_r^{\bowtie} := \sigma_{?y=c}(A_1) \bowtie A_2,$$
$$A_l^{⟕} := \sigma_{?y=c}(A_1 ⟕ A_2) \quad \text{vs.} \quad A_r^{⟕} := \sigma_{?y=c}(A_1) ⟕ A_2.$$

First observe that $?y \notin cVars(A_1)$ and $?y \in pVars(A_2)$, so neither *(FJPush)* nor *(FLPush)* are applicable. In particular, $A_l^{\bowtie} \equiv A_r^{\bowtie}$ and $A_l^{⟕} \equiv A_r^{⟕}$ do **not** follow. We fix the database $D := \{(c, c, c)\}$. It is easily verified that both $A_l^{\bowtie}$ and $A_l^{⟕}$ evaluate to $\{\{?x \mapsto c, ?y \mapsto c\}\}$ on D, whereas $A_r^{\bowtie}$ and $A_r^{⟕}$ both evaluate to $\emptyset$. This shows that the equivalences generally do not hold if the precondition is violated. □

We further clarify the idea of the precondition in the proof for rule *(FJPush)*:

Proof of Equivalence *(FJPush)* from Figure 4.4

$\Rightarrow$: Let $\mu \in \sigma_R(A_1 \bowtie A_2)$. By semantics, $\mu \models R$ and we know that μ is of the form $\mu = \mu_1 \cup \mu_2$ with $\mu_1 \in A_1$, $\mu_2 \in A_2$, and $\mu_1 \sim \mu_2$. Further, by assumption each variable $?x \in vars(R)$ is (i) contained in $cVars(A_1)$ or (ii) not contained in $pVars(A_2)$ (or both). It suffices to show that *(Claim1)* $\mu_1 \models R$ holds, because this implies that the right side generates μ. Let us, for the sake of contradiction, assume that $\mu_1 \not\models R$. Now consider the semantics of filter expressions in Definition 2.9. and recall that $\mu_1 \subseteq \mu$. Given that $\mu \models R$, it is clear that μ_1 does not satisfy R if and only if there is one or more $?x \in vars(R)$ such that $?x \in dom(\mu)$, $?x \notin dom(\mu_1)$ and $?x$ causes

VI. Algebraic Equivalences: Filter Decomposition and Elimination

$\sigma_{R_1 \wedge R_2}(A)$	$\equiv \sigma_{R_1}(\sigma_{R_2}(A))$	*(FDecompI)*
$\sigma_{R_1 \vee R_2}(A)$	$\equiv \sigma_{R_1}(A) \cup \sigma_{R_2}(A)$	*(FDecompII)*
$\sigma_{R_1}(\sigma_{R_2}(A))$	$\equiv \sigma_{R_2}(\sigma_{R_1}(A))$	*(FReord)*
$\sigma_{bnd(?x)}(A)$	$\equiv A$, if $?x \in cVars(A)$	*(FBndI)*
$\sigma_{bnd(?x)}(A)$	$\equiv \emptyset$, if $?x \notin pVars(A)$	*(FBndII)*
$\sigma_{\neg bnd(?x)}(A)$	$\equiv \emptyset$, if $?x \in cVars(A)$	*(FBndIII)*
$\sigma_{\neg bnd(?x)}(A)$	$\equiv A$, if $?x \notin pVars(A)$	*(FBndIV)*

VII. Algebraic Equivalences: Filter Pushing

$\sigma_R(A_1 \cup A_2)$	$\equiv \sigma_R(A_1) \cup \sigma_R(A_2)$	*(FUPush)*
$\sigma_R(A_1 \setminus A_2)$	$\equiv \sigma_R(A_1) \setminus A_2$	*(FMPush)*

If for all $?x \in vars(R) : ?x \in cVars(A_1) \vee ?x \notin pVars(A_2)$, then

$\sigma_R(A_1 \bowtie A_2)$	$\equiv \sigma_R(A_1) \bowtie A_2$	*(FJPush)*
$\sigma_R(A_1 \mathbin{⟕} A_2)$	$\equiv \sigma_R(A_1) \mathbin{⟕} A_2$	*(FLPush)*

Figure 4.4.: Algebraic equivalences: filter manipulation. A, A_1, A_2 stand for $\mathbb{A}$ expressions; $S \subset V$ is a set of variables; R, R_1, R_2 denote filter conditions.

the filter to evaluate to false. We now exploit the constraints (i) and (ii) that are imposed on the variables in $vars(R)$: if variable $?x$ satisfies constraint (i), then it follows from Proposition 4.2 that $?x \in dom(\mu_1)$, which is a contradiction; otherwise, if $?x$ satisfies constraint (ii) we know from Proposition 4.1 that $?x \notin dom(\mu_2)$. Given that $?x \in dom(\mu)$, this implies that $?x$ must be contained in $dom(\mu_1)$, which again gives us a contradiction. We conclude that $\mu_1 \models R$ and therefore *(Claim1)* holds. Direction "$\Leftarrow$" can be shown with an analogical argumentation. □

We conclude our discussion of equivalences for filter manipulation with two rules that can be used to make implicit joins and selections in filter expressions explicit:

Lemma 4.3 Let A be a SPARQL set algebra expression built using only operators $\bowtie$, $\cup$, and triple patterns of the form $[\![t]\!]_D$. Further let $?x, ?y \in cVars(A)$. By $A^{\frac{?y}{?x}}$ we denote the expression obtained from A by replacing all occurrences of $?x$ in A by $?y$; similarly, $A^{\frac{c}{?x}}$ is obtained from A by replacing $?x$ by a URI or a literal c. The following two equivalences hold.

(FElimI)	$\pi_{S \setminus \{?x\}}(\sigma_{?x=?y}(A))$	$\equiv \pi_{S \setminus \{?x\}}(A^{\frac{?y}{?x}})$
(FElimII)	$\pi_{S \setminus \{?x\}}(\sigma_{?x=c}(A))$	$\equiv \pi_{S \setminus \{?x\}}(A^{\frac{c}{?x}})$

□

The two equivalences *(FElimI)* and *(FElimII)* allow to eliminate atomic filter conditions of the form $?x = ?y$ and $?x = c$, by replacing all occurrences of $?x$ in the inner expression by $?y$ and c, respectively. The proof of the lemma is technically involved and can be found in Appendix C.4. Note that in both equivalences the filter expression must be embedded in a projection expressions that projects the variable set $S \setminus \{?x\}$, i.e. **not** including variable $?x$ that is to be replaced (otherwise, variable $?x$ may be bound in left side mappings but not in right side mappings). Given our complete rewriting framework, this is not a major restriction: using the projection pushing rules from Figure 4.4, we can always push projections down on top of filter expressions and afterwards check if rule *(FElimI)* or *(FElimII)* applies. As an important precondition for the equivalence, expression A must be contained in a restricted fragment of set algebra, composed of operators $\bowtie$, $\cup$, and triple patterns. A second precondition is that $?x \in cVars(A)$ and $?y \in cVars(A)$ hold. The following example illustrates the rationale behind all these preconditions.

Example 4.9 Consider the expression-document pairs

$$A_1 := [\![(a, b, ?x)]\!]_{D_1} \bowtie ([\![(?z, b, ?y)]\!]_{D_1} \setminus [\![(?z, d, ?x)]\!]_{D_1}),\ D_1 := \{(a, b, c), (a, d, e)\},$$
$$A_2 := [\![(a, a, ?x)]\!]_{D_2} \bowtie [\![(a, a, ?y)]\!]_{D_2} \bowtie \sigma_{bnd(?y)}([\![(a, a, ?x]\!]_{D_2}),\ D_2 := \{(a, a, a)\},$$
$$A_3 := [\![(a, a, ?x)]\!]_{D_3} \cup [\![(?x, b, ?y)]\!]_{D_3},\ D_3 := \{(a, a, a)\},$$
$$A_4 := [\![(a, a, ?y)]\!]_{D_4} \cup [\![(?x, b, ?y)]\!]_{D_4},\ D_4 := \{(a, a, a)\}.$$

Let $S := \{?y\}$. It is easily verified that, for each $i \in [4]$, the results of evaluating $\pi_{S\setminus\{?x\}}(\sigma_{?x=?y}(A_i))$ and $\pi_{S\setminus\{?x\}}(A_i{}^{?y}_{?x})$ on document D_i differ. Expression A_1 constitutes a counterexample for rule *(FElimI)* for fragments involving operator $\setminus$, while A_2 gives a counterexample for fragments involving operator σ. Concerning A_3 and A_4 we observe that $?y \notin cVars(A_3)$ and $?x \notin cVars(A_4)$, respectively. This shows that the preconditions $?y \in cVars(A)$ and $?x \in cVars(A)$ are necessary. □

We conclude our discussion of filter manipulation with an elaborate example that illustrates both the application of the filter manipulation rules from Figure 4.4 and their possible interplay with the previous rules from Figures 4.1, 4.2, and 4.3:

Example 4.10 Consider the SPARQL algebra expression

$$\pi_{?p,?e}(\sigma_{?sn \neq \text{“Smith”} \wedge ?gn = \text{“Pete”}}($$
$$([\![(?p, \mathit{givenname}, ?gn)]\!]_D \bowtie [\![(?p, \mathit{surname}, ?sn)]\!]_D \bowtie [\![(?p, \mathit{rdf{:}type}, \mathit{Person})]\!]_D)$$
$$\bowtie [\![(?p, \mathit{email}, ?e)]\!]_D))$$

which extracts all persons (*?p*) with givenname (*?gn*) *“Pete”*, surname (*?sn*) different from *“Smith”*, and optionally their email (*?e*). We transform this expression, pursuing the goal to push the filter condition down as far as possible. To keep the example short, we only sketch the strategy and skip several intermediate results.

First, we apply rule *(FLPush)* to push the filter condition inside the left subexpression of the left outer join and afterwards split the filter condition into its components using equivalence *(FDecompI)*. We then push the filter component *?gn*="*Pete*" down, to the top of triple pattern $[\![(?p, \mathit{givenname}, ?gn)]\!]_D$, using rule *(FJPush)*. This series of rewriting steps gives us the following algebra expression.

$$\pi_{?p,?e}\,(\sigma_{?sn\neq \text{“}Smith\text{”}}(\\ \quad \sigma_{?gn=\text{“}Pete\text{”}}([\![(?p, \mathit{givenname}, ?gn)]\!]_D)\\ \quad \bowtie [\![(?p, \mathit{surname}, ?sn)]\!]_D \bowtie [\![(?p, \mathit{rdf{:}type}, \mathit{Person})]\!]_D\\ )\ \text{⟕} [\![(?p, \mathit{email}, ?e)]\!]_D)$$

We next apply rules from Figure 4.3, to push down the top-level projection to the top of the filter $\sigma_{?gn=\text{“}Pete\text{”}}$, then eliminate the filter using *(FElimII)* (by replacing variable *?gn* with *“Pete”*), and reapply the rules from Figure 4.3 in inverse direction, to pull out projection subexpressions again. This gives us the expression below.

$$\pi_{?p,?e}\,(\sigma_{?sn\neq \text{“}Smith\text{”}}(\\ \quad [\![(?p, \mathit{givenname}, \text{“}Pete\text{”})]\!]_D\\ \quad \bowtie [\![(?p, \mathit{surname}, ?sn)]\!]_D \bowtie [\![(?p, \mathit{rdf{:}type}, \mathit{Person})]\!]_D\\ )\ \text{⟕} [\![(?p, \mathit{email}, ?e)]\!]_D).$$

In the final step, we reorder the triple patterns in the join sequence using equivalences *(JAss)* and *(JComm)* and push the filter expression $\sigma_{?sn\neq \text{“}Smith\text{”}}$ down using rule *(FJPush)*. We obtain the following SPARQL set algebra expression.

$$\pi_{?p,?e}(\\ \quad (\ \sigma_{?sn\neq \text{“}Smith\text{”}}([\![(?p, \mathit{surname}, ?sn)]\!]_D)\\ \quad\ \bowtie [\![(?p, \mathit{givenname}, \text{“}Pete\text{”})]\!]_D \bowtie [\![(?p, \mathit{rdf{:}type}, \mathit{Person})]\!]_D\\ \quad)\ \text{⟕} [\![(?p, \mathit{email}, ?e)]\!]_D)$$

We may assume that this expression can be evaluated more efficiently than the original expression, because both filters are applied early. The filter $\sigma_{?gn=\text{“}Pete\text{”}}$ is even implicitly evaluated when extracting data for the respective triple pattern. □

The example illustrates that the rewriting rules provided so far establish a powerful framework for finding alternative evaluation plans. It should be clear that further techniques like heuristics, statistics about the data, knowledge about data access paths, and cost estimation functions are necessary to implement an efficient and comprehensive optimizer on top of these rules, just like it is the case in the context of relational algebra (cf. [Hal75; SC75; Tod75; GLR97]). With our focus on algebraic rewritings and the goal to establish the foundations for advanced optimization approaches, the study of such techniques is beyond the scope of this work.

4.2.6. Rewriting Closed World Negation

We conclude the discussion of SPARQL set algebra optimization with a thorough investigation of operator $\setminus$. First recall that an expression $A_1 \setminus A_2$ retains exactly those mappings from A_1 for which **no** compatible mapping in A_2 exists (cf. Definition 2.10), so the minus operator essentially implements closed-world negation. It is crucial for our discussion here that, in contrast to the other algebraic operations, operator $\setminus$ has no direct counterpart at the syntactic level, but – in SPARQL syntax – is only implicit by the semantics of operator Opt (i.e., Opt is mapped into $\bowtie$ and the definition of $\bowtie$ relies on operator $\setminus$). As argued before in [AG08a], the lack of a syntactic counterpart complicates the encoding of queries involving negation and, as we will show in the following, poses specific challenges to the query optimizer.

We also stress that, as discussed in Section 3.2.3, it is mainly the operator $\setminus$ that is responsible for the high complexity of the (syntactic) operator Opt. Therefore, at the algebraic level special care should be taken in optimizing expressions involving $\setminus$. We start our discussion with the observation from [AG08a] that operator $\setminus$ can always be encoded at the syntactic level using a combination of operators Opt, Filter, and (the negated) filter predicate *bnd*.[3] We illustrate the idea by example:

Example 4.11 The SPARQL expression Q_1 and the corresponding algebra expression C_1 below (obtained by application of the set semantics from Definition 2.11 to Q_1) select all persons for which no name is specified in the data set.

$$Q_1 := ((?p, \textit{rdf:type}, \textit{Person}) \text{ Opt } ((?p, \textit{rdf:type}, \textit{Person}) \text{ And } (?p, \textit{name}, ?n))) \text{ Filter } (\neg \textit{bnd}(?n))$$

$$C_1 := \sigma_{\neg \textit{bnd}(?n)}([\![(?p, \textit{rdf:type}, \textit{Person})]\!]_D \bowtie ([\![(?p, \textit{rdf:type}, \textit{Person})]\!]_D \bowtie [\![(?p, \textit{name}, ?n)]\!]_D))$$

□

From an optimization point of view it would be desirable to have a clean translation of the operator constellation in query Q_1 using **only** operator $\setminus$, but the semantics maps Q_1 into C_1, which involves a comparably complex construction using operators σ, $\bowtie$, $\bowtie$, and predicate *bnd* (thus using operator $\setminus$ implicitly, according to the semantics of $\bowtie$). This translation seems overly complicated and in fact better translations exist for a large class of practical queries, using only $\setminus$, without operators σ, $\bowtie$, $\bowtie$, and predicate *bnd*. The expression C_1 from Example 4.11, for instance, is equivalent to the simpler expression $C_1^{opt} := [\![(?p, \textit{rdf:type}, \textit{Person})]\!]_D \setminus [\![(?p, \textit{name}, ?n)]\!]_D$. In the following we present equivalences that allow us to simplify expressions like C_1, making such "simulated" negation – an artifact of the missing negation operator at syntax level – explicit. We start with a proposition that states some equivalences over expressions involving the minus and the left outer join operator.

[3] The SPARQL fragment used in the proof in [AG08a] slightly differs from our fragment and it is an open problem if negation can **always** be encoded using our SPARQL fragment (cf. Definitions 2.5 and 2.4). Nevertheless, the encoding proposed in [AG08a] works in most practical cases. We will come back to this discussion in Section 5.3.

Proposition 4.4 Let A_1, A_2 be $\mathbb{A}$ expressions and $\widetilde{A_1}$, $\widetilde{A_2}$ be $\widetilde{\mathbb{A}}$ expressions. The following equivalences hold.

$$
\begin{array}{lll}
\textit{(MReord)} & (A_1 \setminus A_2) \setminus A_3 & \equiv (A_1 \setminus A_3) \setminus A_2 \\
\textit{(MMUCorr)} & (A_1 \setminus A_2) \setminus A_3 & \equiv A_1 \setminus (A_2 \cup A_3) \\
\textit{(MJ)} & A_1 \setminus A_2 & \equiv A_1 \setminus (A_1 \bowtie A_2) \\
\textit{(}\widetilde{LJ}\textit{)} & \widetilde{A_1} \between \widetilde{A_2} & \equiv \widetilde{A_1} \between (\widetilde{A_1} \bowtie \widetilde{A_2})
\end{array}
$$

□

The proof of the proposition can be found in Appendix C.5. Rules *(MReord)* and *(MMUCorr)* are general-purpose rewriting rules, listed for completeness. Most important in our context is *(*$\widetilde{LJ}$*)*. It allows to eliminate redundant subexpressions in the right side of $\between$ expressions (over fragment $\widetilde{\mathbb{A}}$). We provide an example:

Example 4.12 The application of rule *(*$\widetilde{LJ}$*)* simplifies C_1 from Example 4.11 to $C_1' := \sigma_{\neg bnd(?n)}(\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \between \llbracket(?p, \textit{name}, ?n)\rrbracket_D)$. □

It is worth mentioning that equivalence *(*$\widetilde{LJ}$*)* taken alone only allows to eliminate right side subexpressions that exactly correspond to the left side $\between$ subexpression. When combining *(*$\widetilde{LJ}$*)* with previous rules, though, we can even eliminate AND-connected subexpressions of the left side expression that occur in the right side:

Example 4.13 Consider the expression C_2 below, which selects all persons that know at least one other person and, optionally, their name.

$$
\begin{aligned}
C_2 := (&\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{knows}, ?p2)\rrbracket_D) \between \\
&(\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{name}, ?n)\rrbracket_D)
\end{aligned}
$$

Rule $(\widetilde{LJ})$ cannot be applied directly to remove the pattern $\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D$ from the right side of the left join expression, but we can apply the rewriting

$$
\begin{aligned}
C_2 \overset{(\widetilde{LJ})}{=}\ & (\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{knows}, ?p2)\rrbracket_D) \between \\
& ((\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{knows}, ?p2)\rrbracket_D) \bowtie \\
& (\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{name}, ?n)\rrbracket_D))) \\
\overset{(*)}{=}\ & (\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{knows}, ?p2)\rrbracket_D) \between \\
& ((\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D) \bowtie \\
& (\llbracket(?p, \textit{knows}, ?p2)\rrbracket_D \bowtie \llbracket(?p, \textit{name}, ?n)\rrbracket_D)) \\
\overset{(\widetilde{JIdem})}{=}\ & (\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{knows}, ?p2)\rrbracket_D) \between \\
& (\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{knows}, ?p2)\rrbracket_D \bowtie \llbracket(?p, \textit{name}, ?n)\rrbracket_D) \\
\overset{(\widetilde{LJ})}{=}\ & (\llbracket(?p, \textit{rdf:type}, \textit{Person})\rrbracket_D \bowtie \llbracket(?p, \textit{knows}, ?p2)\rrbracket_D) \between \llbracket(?p, \textit{name}, ?n)\rrbracket_D,
\end{aligned}
$$

where the correctness of step $(*)$ follows from *(JAss)* and *(JComm)*. □

The two previous examples show that equivalence *(LJ̃)* is a useful tool to simplify subexpressions appearing in the right side of left outer join expressions.

The following lemma completes our rewriting scheme for simulated negation.

Lemma 4.4 Let $A_1, A_2 \in \mathbb{A}$ and $?x \in cVars(A_2) \setminus pVars(A_1)$ be a variable. The following two equivalences hold:

$$\sigma_{\neg bnd(?x)}(A_1 \not\bowtie A_2) \equiv A_1 \setminus A_2 \qquad \textit{(FLBndI)}$$
$$\sigma_{bnd(?x)}(A_1 \not\bowtie A_2) \equiv A_1 \bowtie A_2 \qquad \textit{(FLBndII)}$$ □

Proof of Lemma 4.4

(FLBndI). Let A_1, A_2 be $\mathbb{A}$ expressions and $?x \in cVars(A_2) \setminus pVars(A_1)$ be a variable, which implies that $?x \in cVars(A_1 \bowtie A_2)$ and $?x \notin pVars(A_1 \setminus A_2)$. We transform the left side expression into the right side expression:

$$\begin{aligned}
&\sigma_{\neg bnd(?x)}(A_1 \not\bowtie A_2) \\
&= \sigma_{\neg bnd(?x)}((A_1 \bowtie A_2) \cup (A_1 \setminus A_2)) && \text{[semantics]} \\
&= \sigma_{\neg bnd(?x)}(A_1 \bowtie A_2) \cup \sigma_{\neg bnd(?x)}(A_1 \setminus A_2) && \text{[}\textit{(FUPush)}\text{]} \\
&= \emptyset \cup \sigma_{\neg bnd(?x)}(A_1 \setminus A_2) && \text{[}\textit{(FBndIII)}\text{]} \\
&= \sigma_{\neg bnd(?x)}(A_1 \setminus A_2) && \text{[semantics]} \\
&= A_1 \setminus A_2 && \text{[}\textit{(FBndIV)}\text{]}
\end{aligned}$$

(FLBndII). Similar to *(FLBndI)*, see Appendix C.6 for details. □

The relevant rule in our context is *(FLBndI)*. We apply it to the running example:

Example 4.14 The application of the rewriting rule *(FLBndI)* from Lemma 4.4 to expression C_1' from Example 4.12 yields the expression

$$C_1^{opt} := [\![(?p, \textit{rdf:type}, \textit{Person})]\!]_D \setminus [\![(?p, \textit{name}, ?n)]\!]_D.$$

The construction in expression C_1 from Example 4.11, involving $\not\bowtie$, $\bowtie$, σ, and filter predicate *bnd*, has been replaced by a simple minus expression. □

4.3. From Set to Bag Algebra

We now take the leap from SPARQL set algebra to the bag algebra introduced in Definition 2.14. The results presented in this section are more interesting from a practical point of view, because they immediately carry over to the official W3C semantics for SPARQL [spac], which follows the bag algebra approach.

Analogously to our discussion of SPARQL set algebra, we start with the definition of the fragment comprising all SPARQL bag algebra expressions, called $\mathbb{A}^+$:

Definition 4.6 (Fragment $\mathbb{A}^+$) Fragment $\mathbb{A}^+$ denotes the full class of SPARQL bag algebra expressions, i.e. expression built using operators $\cup$, $\bowtie$, $\setminus$, $\overline{\bowtie}$, π, σ, and (bracket-enclosed) triple patterns of the form $[\![t]\!]_D^+$. □

Like Definition 4.3, the above definition introduces a class of purely syntactic entities. It should be clear that, when evaluating an expression $A^+ \in \mathbb{A}^+$, this implicitly involves the SPARQL bag algebra operators introduced in Definition 2.14, because all triple patterns are of the form $[\![t]\!]_D^+$ and therefore extract multi-sets (as opposed to Definition 4.3, where the triple patterns of the form $[\![t]\!]_D$ extract simple sets of mappings). Following the conventions from before, if D is known from the context we shall refer to the mapping multi-set obtained from A^+ by application of the semantics from Definition 2.14 as the *result obtained when evaluating A^+ on D*.

The ultimate goal of our analysis of bag semantics is to identify those equivalences from Section 4.2 that carry over from set to bag semantics. To discuss the relations between the two semantics, we define a bijective mapping between SPARQL set algebra and SPARQL bag algebra, which allows us to map expressions from one algebra into same-structured expressions from the other algebra. This is simply done by replacing each triple pattern $[\![t]\!]_D$ with $[\![t]\!]_D^+$ when converting from set to bag algebra and, vice versa, by replacing each pattern $[\![t]\!]_D^+$ with $[\![t]\!]_D$ when converting from bag to set algebra. Formally, we define the function $s2b$ ("*set-to-bag*") as follows.

Definition 4.7 (Function *s2b*) Let $A_1, A_2 \in \mathbb{A}$ be SPARQL set algebra expressions, $S \subset V$ a set of variables, and R a filter condition. We define the bijective function $s2b : \mathbb{A} \mapsto \mathbb{A}^+$ inductively on the structure of $\mathbb{A}$-expression:

$$
\begin{array}{ll}
s2b([\![t]\!]_D) & := [\![t]\!]_D^+ \\
s2b(A_1 \bowtie A_2) & := s2b(A_1) \bowtie s2b(A_2) \\
s2b(A_1 \cup A_2) & := s2b(A_1) \cup s2b(A_2) \\
s2b(A_1 \setminus A_2) & := s2b(A_1) \setminus s2b(A_2) \\
s2b(A_1 \overline{\bowtie} A_2) & := s2b(A_1) \overline{\bowtie} s2b(A_2) \\
s2b(\pi_S(A_1)) & := \pi_S(s2b(A_1)) \\
s2b(\sigma_R(A_1)) & := \sigma_R(s2b(A_1))
\end{array}
$$
□

We shall use the inverse of the function, denoted as $s2b^{-1}(A^+)$, to transform a bag algebra expression $A^+ \in \mathbb{A}^+$ into its set algebra counterpart. Intuitively, the function reflects the close connection between the set and bag semantics from Definitions 2.11 and 2.15, which differ only in the translation for triple patterns. In particular, it is easily verified that, for each SPARQL expression or query Q, it holds that $[\![Q]\!]_D^+ = s2b([\![Q]\!]_D)$ and $[\![Q]\!]_D = s2b^{-1}([\![Q]\!]_D^+)$ (when interpreting the results of function $[\![.]\!]_D$ and $[\![.]\!]_D^+$ as SPARQL algebra expressions rather than sets or multi-sets of mappings). Given this connection, we next transfer Lemma 3.1, which relates the two semantics to each other, into the context of SPARQL set and bag algebra.

Lemma 4.5 The following claims hold.

1. Let $A \in \mathbb{A}$ and D be an RDF document. Let Ω denote the result of evaluating A on D and let (Ω^+, m^+) denote the mapping multi-set obtained when evaluating $s2b(A)$ on D. Then for all $\mu \in \mathcal{M} : \mu \in \Omega \Leftrightarrow \mu \in \Omega^+$.
2. Let $A^+ \in \mathbb{A}^+$ and D be an RDF document. Let (Ω^+, m^+) denote the mapping multi-set obtained when evaluating A^+ on D and let Ω denote the result of evaluating $s2b^{-1}(A^+)$ on D. Then for all $\mu \in \mathcal{M} : \mu \in \Omega^+ \Leftrightarrow \mu \in \Omega$. □

Essentially, the lemma shows that corresponding bag and set algebra expressions coincide w.r.t. the mappings that they compute and only may differ in that the bag semantics assigns a multiplicity greater than 1 to some result mappings. The proof of the lemma is similar in idea to the proof of Lemma 3.1 and therefore omitted.

Finally recall that the concepts of possible and certain variables directly carry over to bag algebra expressions (cf. Section 4.1), so we shall use functions $pVars(A^+)$ and $cVars(A^+)$ to extract possible and certain variables from some SPARQL bag algebra expression $A^+ \in \mathbb{A}^+$. The adaption of the incompatibility property for bag algebra expressions is not that straightforward and requires some advanced considerations, to properly cope with multi-sets. We will tackle this task in the following subsection.

4.3.1. The Incompatibility Property for SPARQL Bag Algebra

As before in the case of set algebra, we are interested in bag algebra expressions that exhibit the incompatibility property. Given that a mapping might appear multiple times in the result when evaluating a bag algebra expression, we need a refined definition of the incompatibility property from Definition 4.4:

Definition 4.8 (Incompatibility Property for Bag Algebra) Let A^+ be a bag algebra expression and let (Ω_D^+, m_D^+) denote the multi-set obtained when evaluating expression A^+ on some document D. Then A^+ has the *incompatibility property* if and only if for every document D it holds that (i) each two distinct mappings $\mu_1 \neq \mu_2$ in Ω_D^+ are incompatible and (ii) $m_D^+(\mu) = 1$ for all $\mu \in \Omega_D^+$. □

Informally speaking, as an additional constraint compared to the incompatibility property for SPARQL set algebra we enforce that (ii) no mapping appears multiple times in the result multi-set. Intuitively, this additional constraint arises from the observation that duplicates are always compatible to each other (i.e., $\mu \sim \mu$ holds trivially for each mapping μ) and may harm equivalences that – in the context of set algebra – hold for expressions that exhibit the incompatibility property.

An appealing question is whether the definition of fragment $\widetilde{\mathbb{A}}$, our syntactic restriction of $\mathbb{A}$ expressions for which the incompatibility property holds (cf. Definition 4.5 and Lemma 4.1) naturally carries over to bag algebra. Formally, we phrase

this question as follows: does $\widetilde{A} \in \widetilde{\mathbb{A}}$ imply that $s2b(\widetilde{A})$ has the incompatibility property (for bag algebra)? The following example shows that this is not the case.

Example 4.15 Consider the two SPARQL set algebra expressions

$$\widetilde{A_1} := [\![(?x, c, 1)]\!]_D \cup [\![(?x, c, 1)]\!]_D,$$
$$\widetilde{A_2} := \pi_{?x}([\![(?x, c, ?y)]\!]_D),$$

and the database $D := \{(c, c, 1), (c, c, 2)\}$. It is easily verified that $\widetilde{A_1}, \widetilde{A_2} \in \widetilde{\mathbb{A}}$. The corresponding SPARQL bag algebra expressions are

$$\widetilde{A_1^+} := s2b(\widetilde{A_1}) = [\![(?x, c, 1)]\!]_D^+ \cup [\![(?x, c, 1)]\!]_D^+,$$
$$\widetilde{A_2^+} := s2b(\widetilde{A_2}) = \pi_{?x}([\![(?x, c, ?y)]\!]_D^+).$$

Both $\widetilde{A_1^+}$ and $\widetilde{A_2^+}$ evaluate on D to $(\{\{?x \mapsto c\}\}, m)$ with $m(\{?x \mapsto c\}) := 2$, so neither $\widetilde{A_1^+}$ nor $\widetilde{A_2^+}$ has the incompatibility property, as they violate condition (ii) in Definition 4.8. Hence, both are counterexamples for the claim that the bag algebra expression $s2b(\widetilde{A})$ has the incompatibility property if $\widetilde{A} \in \widetilde{\mathbb{A}}$ holds. □

The example clarifies that we need to adjust the definition of fragment $\widetilde{\mathbb{A}}$ for SPARQL bag algebra. In particular, expressions $\widetilde{A_1^+}$ and $\widetilde{A_2^+}$ from the above example indicate that the restrictions for expressions involving operator projection π and union $\cup$ (the last two bullets in Definition 4.5) are not strong enough for SPARQL bag algebra. A closer investigation reveals that we can merely impose syntactic conditions such that fragments involving these operators still exploit the incompatibility property: intuitively, the syntactic restrictions for $\cup$- and π-expressions imposed in Definition 4.5 exploit the observation that in some cases compatible mappings collapse to a single mapping under set semantics, which is never the case under bag semantics. Therefore, we exclude these operators in our efforts to identify a SPARQL bag algebra fragment that exhibits the incompatibility property. This restriction leads to a compact fragment of SPARQL bag algebra, called $\widetilde{\mathbb{A}^+}$:

Definition 4.9 (Fragment $\widetilde{\mathbb{A}^+}$) The fragment $\widetilde{\mathbb{A}^+}$ comprises all SPARQL bag algebra expressions built using only operators $\bowtie$, $\setminus$, $\mathbin{\overline{\bowtie}}$, σ, and (bracket-enclosed) triple patterns of the form $[\![t]\!]_D^+$. □

First observe that $\widetilde{\mathbb{A}^+}$ is a subset of fragment $\widetilde{\mathbb{A}}$ w.r.t. to our bijective mapping:

Proposition 4.5 It holds that $\widetilde{A^+} \in \widetilde{\mathbb{A}^+} \rightarrow s2b^{-1}(\widetilde{A^+}) \in \widetilde{\mathbb{A}}$. □

The proof follows directly from the definitions of the two fragments. Also note that the opposite direction does not hold, as witnessed by the $\widetilde{\mathbb{A}}$ expressions $\widetilde{A_1}$ and $\widetilde{A_2}$ from Example 4.15 (i.e., we observe that $s2b(\widetilde{A_1}) \notin \widetilde{\mathbb{A}^+}$ and $s2b(\widetilde{A_2}) \notin \widetilde{\mathbb{A}^+}$). When combining the above proposition with Lemma 4.1 and Lemma 4.5 we obtain:

Corollary 4.1 Let $\widetilde{A^+} \in \widetilde{\mathbb{A}^+}$, D be an RDF document, and let (Ω^+, m^+) denote the result obtained when evaluating $\widetilde{A^+}$ on D. Then each pair of distinct mappings in Ω_+ is incompatible. □

Hence, condition (i) of Definition 4.8 always holds for $\widetilde{\mathbb{A}^+}$ expressions. Thus, to show that these expressions satisfy the incompatibility property, it remains to show that (ii) no mapping occurs multiple times. We prove the following lemma.

Lemma 4.6 Let $\widetilde{A^+} \in \widetilde{\mathbb{A}^+}$ and D be an RDF document. We denote by (Ω^+, m^+) the mapping multi-set obtained when evaluating $\widetilde{A^+}$ on D and by Ω the mapping set obtained when evaluating $s2b^{-1}(\widetilde{A^+})$ on D. It holds that $\Omega \cong (\Omega^+, m^+)$ (where symbol "$\cong$" denotes equality between sets and multi-sets, cf. Definition 2.13). □

Proof of Lemma 4.6

Given Lemma 4.5, which shows that the two semantics differ at most w.r.t. the associated multiplicity, it suffices to show that bag algebra expressions contained in $\widetilde{\mathbb{A}^+}$ have multiplicity 1 associated with every mapping. We prove this claim by induction on the structure of $\widetilde{\mathbb{A}^+}$ expressions. Consider expression $\widetilde{A^+} \in \widetilde{\mathbb{A}^+}$. We fix document D and denote by (Ω, m) the mapping set obtained when evaluating $\widetilde{A^+}$ on D. Similarly, if A_1^+ and A_2^+ are subexpressions of A^+, we denote their evaluation results on D by (Ω_1, m_1) and (Ω_2, m_2), respectively. The basic case $\widetilde{A^+} := [\![t]\!]_D^+$ follows trivially from Definition 2.14, which fixes $m(\mu) := 1$ for all $\mu \in \Omega$. Following the structure of fragment $\widetilde{\mathbb{A}^+}$ (cf. Definition 4.9), we distinguish four cases.

(1) Let $\widetilde{A^+} := \widetilde{A_1^+} \bowtie \widetilde{A_2^+}$. Then each $\mu \in \Omega$ is of the form $\mu = \mu_1 \cup \mu_2$ where $\mu_1 \in \Omega_1$ and $\mu_2 \in \Omega_2$ are compatible mappings. By induction hypothesis we know that $m_1(\mu_1) = 1$ and $m_2(\mu_2) = 1$, which implies (by the definition of m for the case of join expressions) that $m(\mu) = 1 + x$, where 1 is obtained from $m_1(\mu_1) * m_2(\mu_2)$ and x is some additional multiplicity obtained from mapping pairs (μ_1', μ_2') distinct from (μ_1, μ_2). Hence, we have to show that there are no mappings distinct from μ_1 and μ_2 that generate μ. Let us for the sake of contradiction assume there exist two mappings $\mu_1' \in \Omega_1$, $\mu_2' \in \Omega_2$ such that $\mu_1' \sim \mu_2'$, $\mu_1' \cup \mu_2' = \mu$ and (w.l.o.g) $\mu_1' \neq \mu_1$. Then μ_1' must be compatible with μ_1, since otherwise it would follow that $\mu_1 \cup \mu_2 \not\sim \mu_1' \cup \mu_2'$. This observation contradicts Corollary 4.1, which states that each two distinct mappings obtained when evaluating an $\widetilde{\mathbb{A}^+}$ expressions are incompatible (and clearly $A_1^+ \in \widetilde{\mathbb{A}^+}$). (2) Let $\widetilde{A^+} := \widetilde{A_1^+} \setminus \widetilde{A_2^+}$. The claim follows easily by application of the induction hypothesis and the observation that the multiplicity of each mapping in $\Omega \subseteq \Omega_1$ is either identical to its multiplicity in m_1 or equals to 0. (3) Let $\widetilde{A^+} := \widetilde{A_1^+} \rtimes \widetilde{A_2^+} = (\widetilde{A_1^+} \bowtie \widetilde{A_2^+}) \cup (\widetilde{A_1^+} \setminus \widetilde{A_2^+})$. We fix some $\mu \in \Omega$ and distinguish three case. (3a) If mapping μ is generated by the left side subexpression $A_\bowtie^+ := \widetilde{A_1^+} \bowtie \widetilde{A_2^+}$ but not by the right side-expression $A_\setminus^+ := \widetilde{A_1^+} \setminus \widetilde{A_2^+}$, then the

claim follows analogously to case (1) of the proof. (3b) If μ is generated by $A^+_{\setminus}$ but not by $A^+_{\bowtie}$, then the argumentation is similar to case (2) of the proof. Finally, (3c) assume that μ is generated by both (i) $A^+_{\bowtie}$ and (ii) $A^+_{\setminus}$. Then (i) implies that there are compatible mappings $\mu_1 \in \Omega_1$, $\mu_2 \in \Omega_2$ such that $\mu = \mu_1 \cup \mu_2$, while (ii) implies that $\mu \in \Omega_1$ and there is no compatible mapping to μ in Ω_2. Similar to the argumentation in case (1) we have that each pair of distinct mappings in Ω_1 is incompatible, which – given that both $\mu_1 \subseteq \mu \in \Omega_1$ and $\mu \in \Omega_1$ – implies that $\mu_1 = \mu$. This leads to a contradiction: according to (i), $\mu_2 \in \Omega_2$ is compatible with $\mu_1 = \mu$, but (ii) enforces that there is no mapping in Ω_2 compatible to μ. Finally, case (4) $\widetilde{A^+} := \sigma_R(\widetilde{A^+_1})$ follows by induction hypothesis, analogical to case (2). □

Note that a similar (weaker) result has been stated in Proposition 1 in [AG08b], namely that every result mapping obtained when evaluating an expression built using the (abstract syntax operators) And, Filter, and Opt under bag semantics has cardinality 1. When combining Corollary 4.1 and Lemma 4.6 above, it follows that the SPARQL bag algebra fragment $\widetilde{A^+}$ has the incompatibility property:

Lemma 4.7 Every expression $\widetilde{A^+}$ in $\widetilde{\mathbb{A}^+}$ has the incompatibility property. □

4.3.2. Optimization Rules for Bag Algebra

We now systematically revisit the rewriting rules from Section 4.2 and investigate which of these equivalences are valid for SPARQL bag algebra expressions in place of set algebra expressions. We introduce the following concept.

Definition 4.10 (Equivalence Carry-over from Set to Bag Algebra) Given an equivalence *(SomeEq)* for the SPARQL set algebra fragment $\mathbb{A}$ we say that *the equivalence carries over to SPARQL bag algebra* if it holds for all expressions in $\mathbb{A}^+$ as well. Similarly, an equivalence *(SomeEq)* for fragment $\widetilde{\mathbb{A}}$ *carries over to SPARQL bag algebra* if it holds for all expressions in $\widetilde{\mathbb{A}^+}$. We denote the resulting SPARQL bag algebra equivalences by *(SomeEq$^+$)* and *(SomeEq$^+$)*, respectively. □

It should be clear from Lemma 4.5 that the only reason why equivalences may not carry over to bag algebra is that some result mappings differ in their multiplicity; the set of mappings in the result is always guaranteed to be identical. Hence, to show that an equivalence does not carry over from SPARQL set to bag algebra we must provide an instantiation of the equivalence and a fixed document D such that, when evaluating the left and right side of the equivalence on D, we obtain a mapping that has different multiplicities associated in the left and right side result mapping multi-set. To clarify this issue, we start with an example that provides an equivalence that does not carry over to SPARQL bag algebra.

Example 4.16 Consider equivalence *(UIdem)* for SPARQL set algebra from Figure 4.1. The rule was originally established for fragment $\mathbb{A}$. To show that it does **not** carry over to SPARQL bag algebra, we provide an expression $A \in \mathbb{A}^+$ and a document D such that the left side of the equation (i.e., A) evaluated on D and the right side (i.e., $A \cup A$) evaluated on D generate a mapping that appears with different multiplicity. Consider expression $A := [\![(c, c, ?x)]\!]^+_D$, document $D := \{(c, c, c)\}$, and observe that $A \in \mathbb{A}^+$ The result of evaluating A on D is $(\{\{?x \mapsto c\}\}, m)$ with $m(\{?x \mapsto c\}) := 1$, whereas $A \cup A$ evaluates to $(\{\{?x \mapsto c\}\}, m')$ with $m'(\{?x \mapsto c\}) := 2$. Although the mapping sets coincide, the results differ w.r.t. the multiplicity that is assigned to the result mapping $\{?x \mapsto c\}$. □

Idempotence and Inverse. We start with the equivalences from Figure 4.1:

Lemma 4.8 Consider the equivalences for SPARQL set algebra from Figure 4.1.

1. *(UIdem)* does **not** carry over to SPARQL bag algebra.
2. $(\widetilde{JIdem})$, $(\widetilde{LIdem})$, and *(Inv)* carry over to SPARQL bag algebra. □

Proof of Lemma 4.8

Lemma 4.8(1): See Example 4.16 for a counterexample.

Lemma 4.8(2): To show that $(\widetilde{JIdem})$ carries over to SPARQL bag algebra we have to show that $\widetilde{A^+} \bowtie \widetilde{A^+} \equiv \widetilde{A^+}$ for every expression $\widetilde{A^+} \in \widetilde{\mathbb{A}^+}$. From Lemma 4.6 we know that the set and bag semantics coincide for $\widetilde{A^+}$ expressions and we know that the equivalence holds under set semantics. From Definition 4.8 it follows that $\widetilde{A^+} \bowtie \widetilde{A^+} \in \widetilde{\mathbb{A}^+}$, so the bag algebra equivalence $(\widetilde{JIdem^+})$ holds. The argumentation for $(\widetilde{LIdem^+})$ is the same. Finally, equivalence *(Inv$^+$)* follows easily from Lemma 4.5 and the observation that the equivalence holds under set semantics (the extracted mapping set is empty, so there cannot be any differences in the multiplicity). □

Associativity, Commutativity, and Distributivity. Concerning the eight equivalences from Figure 4.2 we obtain only positive results:

Lemma 4.9 The equivalences *(UAss)*, *(JAss)*, *(UComm)*, *(JComm)*, *(JUDistR)*, *(JUDistL)*, *(MUDistR)*, *(LUDistR)* introduced in Figure 4.2 for SPARQL set algebra carry over to SPARQL bag algebra. □

As a representative, we provide the proof for rewriting rule *(JUDistR)* below. The proofs for the remaining equivalences are provided in Appendix C.7.

Proof of Equivalence *(JUDistR$^+$)* from Lemma 4.9

We exploit Lemma 4.5, which states that the results of evaluating set and bag algebra expressions differ at most in the associated multiplicity. Given that *(JUDistR)* holds for SPARQL set algebra, it thus suffices to show that, for each mapping

μ that is contained in the left and right side result of the equivalence, the associated left and right side cardinalities of μ coincide. Let $A_1^+, A_2^+, A_3^+ \in \mathbb{A}^+$. We fix document D and define $A_l^+ := (A_1^+ \cup A_2^+) \bowtie A_3^+$, $A_r^+ := (A_1^+ \bowtie A_3^+) \cup (A_2^+ \bowtie A_3^+)$, $A_{1\cup 2}^+ := A_1^+ \cup A_2^+$, $A_{1\bowtie 3}^+ := A_1^+ \bowtie A_3^+$, and $A_{2\bowtie 3}^+ := A_2^+ \bowtie A_3^+$. Given a SPARQL bag algebra expression A_i^+ with some index i, we denote by (Ω_i, m_i) the result of evaluating A_i^+ on D. Now let us consider a mapping μ that is contained in both Ω_l and Ω_r. Using the semantics from Definition 2.14 we rewrite $m_l(\mu)$ step-by-step:

$$
\begin{aligned}
m_l(\mu) &= \textstyle\sum_{(\mu_{1\cup 2},\mu_3)\in\{(\mu^*_{1\cup 2},\mu^*_3)\in\Omega_{1\cup 2}\times\Omega_3 \mid \mu^*_{1\cup 2}\cup\mu^*_3=\mu\}} (m_{1\cup 2}(\mu_{1\cup 2}) * m_3(\mu_3)) \\
&= \textstyle\sum_{(\mu_{1\cup 2},\mu_3)\in\{(\mu^*_{1\cup 2},\mu^*_3)\in\Omega_{1\cup 2}\times\Omega_3 \mid \mu^*_{1\cup 2}\cup\mu^*_3=\mu\}} ((m_1(\mu_{1\cup 2}) + m_2(\mu_{1\cup 2})) * m_3(\mu_3)) \\
&= \textstyle\sum_{(\mu_{1\cup 2},\mu_3)\in\{(\mu^*_{1\cup 2},\mu^*_3)\in\Omega_{1\cup 2}\times\Omega_3 \mid \mu^*_{1\cup 2}\cup\mu^*_3=\mu\}} (\\
&\qquad (m_1(\mu_{1\cup 2}) * m_3(\mu_3)) + (m_2(\mu_{1\cup 2}) * m_3(\mu_3))) \\
&\overset{(S3)}{=} \textstyle\sum_{(\mu_{1\cup 2},\mu_3)\in\{(\mu^*_{1\cup 2},\mu^*_3)\in\Omega_{1\cup 2}\times\Omega_3 \mid \mu^*_{1\cup 2}\cup\mu^*_3=\mu\}} (m_1(\mu_{1\cup 2}) * m_3(\mu_3)) + \\
&\quad \textstyle\sum_{(\mu_{1\cup 2},\mu_3)\in\{(\mu^*_{1\cup 2},\mu^*_3)\in\Omega_{1\cup 2}\times\Omega_3 \mid \mu^*_{1\cup 2}\cup\mu^*_3=\mu\}} (m_2(\mu_{1\cup 2}) * m_3(\mu_3)) \\
&\overset{(*)}{=} \textstyle\sum_{(\mu_1,\mu_3)\in\{(\mu^*_1,\mu^*_3)\in\Omega_1\times\Omega_3 \mid \mu^*_1\cup\mu^*_3=\mu\}} (m_1(\mu_1) * m_3(\mu_3)) + \\
&\quad \textstyle\sum_{(\mu_2,\mu_3)\in\{(\mu^*_2,\mu^*_3)\in\Omega_2\times\Omega_3 \mid \mu^*_2\cup\mu^*_3=\mu\}} (m_2(\mu_2) * m_3(\mu_3)) \\
&= m_{1\bowtie 3}(\mu) + m_{2\bowtie 3}(\mu) \\
&= m_r(\mu),
\end{aligned}
$$

where *(S3)* stands for the application of rule *(S3)* for sum expression stated in Proposition C.1 in Appendix C.7 and step $(*)$ follows by semantics of operator $\cup$. □

Projection Pushing. All projection pushing rules carry over to bag semantics:

Lemma 4.10 The equivalences *(PBaseI)*, *(PBaseII)*, *(PUPush)*, *(PJPush)*, *(PMPush)*, *(PLPush)*, *(PFPush)*, and *(PMerge)* introduced in Figure 4.3 for SPARQL set algebra carry over to SPARQL bag algebra. □

The proofs of the rules are systematic rewritings of sum expressions, similar in style to the proof of equivalence *(JUDistR)* discussed previously. The interested reader will find the technical details in Appendix C.8.

Filter Decomposition, Elimination, and Pushing. Concerning the filter manipulation rewriting rules from Figure 4.4 we observe the following:

Lemma 4.11 Consider the equivalences for SPARQL set algebra from Figure 4.4.

1. Equivalence *(FDecompII)* does **not** carry over to SPARQL bag algebra.
2. *(FDecompI)*, *(FReord)*, *(FBndI)*, *(FBndII)*, *(FBndIII)*, *(FBndIV)*, *(FUPush)*, *(FMPush)*, *(FJPush)*, and *(FLPush)* carry over to SPARQL bag algebra. □

Proof of Lemma 4.11

Lemma 4.11(1): As a counterexample for the equivalence consider the document $D := \{(c, c, c)\}$ and the bag algebra expressions $A_l^+ := \sigma_{(\neg?x=a)\vee(\neg?x=b)}([\![(c, c, ?x)]\!]_D^+)$ and $A_r^+ := \sigma_{\neg?x=a}([\![(c, c, ?x)]\!]_D^+) \cup \sigma_{\neg?x=b}([\![(c, c, ?x)]\!]_D^+)$. The result obtained when evaluating expression A_l on D is $(\{\{?x \mapsto c\}\}, m_l)$ with $m_l(\{?x \mapsto c\}) := 1$, but expression A_r evaluates on D to $(\{\{?x \mapsto c\}\}, m_r)$ with $m_r(\{?x \mapsto c\}) := 2$.

Lemma 4.11(2): We only prove equivalence *(FJPush$^+$)* here; proof sketches for the remaining rules can be found in Appendix C.9. Again we exploit that, by Lemma 4.5, the results of evaluating set and bag SPARQL algebra expressions differ at most w.r.t. the associated multiplicity, i.e. we show that, for a fixed mapping μ that is contained in the left and right side of the equivalence, the associated left and right side cardinalities for the mapping coincide. We fix document D. As in previous proofs, given a SPARQL bag algebra expression A_i^+ with some index i we denote by (Ω_i, m_i) the mapping multi-set obtained when evaluating A_i^+ on D.

Let $A_1^+, A_2^+ \in \mathbb{A}^+$ and R be a filter condition such that for all $?x \in vars(R)$: $?x \in cVars(A_1) \vee ?x \notin pVars(A_2)$. Put $A_l^+ := \sigma_R(A_1^+ \bowtie A_2^+)$, $A_r^+ := \sigma_R(A_1^+) \bowtie A_2^+$, $A_{1\bowtie 2}^+ := A_1^+ \bowtie A_2^+$, and $A_{\sigma 1}^+ := \sigma_R(A_1^+)$. Consider a mapping μ that is contained in the result of evaluating A_l^+ and A_r^+. Clearly it holds that $\mu \models R$ and $\mu \in \Omega_{1\bowtie 2}$. Combining these observations with the semantics from Definition 2.10 we obtain

$$\begin{aligned} m_l(\mu) &= m_{1\bowtie 2}(\mu) \\ &= \textstyle\sum_{(\mu_1,\mu_2)\in\{(\mu_1^*,\mu_2^*)\in\Omega_1\times\Omega_2 \mid \mu_1^*\cup\mu_2^*=\mu\}} (m_1(\mu_1) * m_2(\mu_2)) \\ &\overset{(*)}{=} \textstyle\sum_{(\mu_1,\mu_2)\in\{(\mu_1^*,\mu_2^*)\in\Omega_{\sigma 1}\times\Omega_2 \mid \mu_1^*\cup\mu_2^*=\mu\}} (m_{\sigma 1}(\mu_1) * m_2(\mu_2)) \\ &= m_r(\mu), \end{aligned}$$

where rewriting step $(*)$ follows from the observation that the precondition for all $?x \in vars(R)$: $?x \in cVars(A_1) \vee ?x \notin pVars(A_2)$ implies that for all $\mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ s.t. $\mu_1 \cup \mu_2 = \mu$ the mappings μ_1 and μ agree on variables in $vars(R)$, i.e. each $?x \in vars(R)$ is either bound to the same value in μ_1 and μ or unbound in both. □

Finally, we address the rules *(FElimI)* and *(FElimII)* from Lemma 4.3. The following lemma shows that these rules are also applicable under bag semantics:

Lemma 4.12 Let A be a SPARQL bag algebra expression composed of operators $\bowtie$, $\cup$, and triple patterns of the form $[\![t]\!]_D^+$. Further let $?x, ?y \in cVars(A)$. By $A\frac{?y}{?x}$ we denote the expression obtained from A by replacing all occurrences of $?x$ in A by $?y$; similarly, expression $A\frac{c}{?x}$ is obtained from A by replacing $?x$ by URI or literal c. The following two equivalences hold.

$$\begin{aligned} \pi_{S\setminus\{?x\}}(\sigma_{?x=?y}(A)) &\equiv \pi_{S\setminus\{?x\}}(A\tfrac{?y}{?x}) && \textit{(FElimI}^+\textit{)} \\ \pi_{S\setminus\{?x\}}(\sigma_{?x=c}(A)) &\equiv \pi_{S\setminus\{?x\}}(A\tfrac{c}{?x}) && \textit{(FElimII}^+\textit{)} \end{aligned}$$

□

The proof is an adapted version of the proof of Lemma 4.3, which additionally takes the multiplicities of the mappings into account. We omit the details.

Rewriting Closed World Negation. Finally, we investigate the equivalences established in the context of our rewriting scheme for (simulated) negation. Like in most cases before, our finding is that all these rules carry over to bag algebra, so our rewriting scheme remains applicable when switching to SPARQL bag algebra:

Lemma 4.13 Rules *(MReord)*, *(MMUCorr)*, *(MJ)*, $(\widetilde{LJ})$ from Proposition 4.4 and *(FLBndI)*, *(FLBndII)* from Lemma 4.4 carry over to SPARQL bag algebra. □

The proofs are straightforward and can be found in Appendix C.10.

4.4. Summary of Results and Practical Implications

We summarize all results established throughout this chapter in Table 4.1 at the end of the chapter (due to space limitations, we do not repeat equivalences *FElimI* and *FElimII* from Lemma 4.3 and 4.12). The last four columns of the table indicate for which fragments the respective equivalence is valid. To give an example, the check mark "✓" in column $\widetilde{\mathbb{A}^+}$ for equivalence *LIdem* indicates that the equivalence holds for every expression $A \in \widetilde{\mathbb{A}^+}$; the minus symbol "−" in column $\mathbb{A}^+$ for the same equivalence shows that the equivalence does not hold in the general case when choosing some $A \in \mathbb{A}^+$. Clearly, all equivalences that are valid for fragment $\mathbb{A}$ (respectively $\mathbb{A}^+$) are also valid for the subfragment $\widetilde{\mathbb{A}}$ (respectively $\widetilde{\mathbb{A}^+}$). As discussed previously in Section 4.3.2, most of the equivalences carry over from SPARQL set to bag algebra, which is implicit in the figure whenever a check mark in column $\mathbb{A}$ (respectively $\widetilde{\mathbb{A}}$) implies a check mark in the corresponding $\mathbb{A}^+$ (respectively $\widetilde{\mathbb{A}^+}$) column. The only two equivalences that do not carry over to SPARQL bag algebra are *UIdem* and *FDecompII*; interestingly, in both cases it is the union operator that poses problems. The observation that almost all equivalences are valid under SPARQL bag algebra is good news from a practical point of view, since all equivalences except for the two outliers can be safely used for query optimization in the context of the official W3C SPARQL Recommendation [spac].

We finally discuss implications and extensions of previous results for SPARQL engines that build upon the official W3C semantics. Our focus in this study is on the interplay between the semantics and different SPARQL query forms and solution modifiers (cf. the discussion in Section 2.3.4). In particular, we will show that in many cases SPARQL engines can use the simpler set semantics for query evaluation, thus avoiding the overhead imposed by the additional multiplicity computation. We start with a discussion of SPARQL Ask queries in the following subsection.

4.4.1. SPARQL ASK Queries

First recall that SPARQL Ask queries (cf. Definition 2.6) return *true* if the extracted mapping set is not empty, and *false* otherwise. When combining and extending previous results on algebraic SPARQL query optimization, we obtain the following.

Lemma 4.14 Let Q, Q_1, Q_2 be SPARQL expressions. The following claims hold.

1. $[\![\text{Ask}(Q)]\!]_D \Leftrightarrow [\![\text{Ask}(Q)]\!]_D^+$
2. $[\![\text{Ask}(Q_1 \text{ Union } Q_2)]\!]_D \Leftrightarrow [\![\text{Ask}(Q_1)]\!]_D \vee [\![\text{Ask}(Q_2)]\!]_D$
3. $[\![\text{Ask}(Q_1 \text{ Opt } Q_2)]\!]_D \Leftrightarrow [\![\text{Ask}(Q_1)]\!]_D$
4. If $pVars([\![Q_1]\!]_D) \cap pVars([\![Q_2]\!]_D) = \emptyset$ then $[\![\text{Ask}(Q_1 \text{ And } Q_2)]\!]_D \Leftrightarrow [\![\text{Ask}(Q_1)]\!]_D \wedge [\![\text{Ask}(Q_2)]\!]_D$. □

Proof of Lemma 4.14

Lemma 4.14(1): Follows from the semantics of Ask queries and Lemma 3.1

Lemma 4.14(2): Follows from the semantics of Ask queries and the semantics of the Union operator (cf. Definitions 2.10 and 2.11).

Lemma 4.14(3): Follows from the semantics of Ask queries and the semantics of the Opt operator (cf. Definitions 2.10 and 2.11). In particular, observe that for $\bowtie$, the algebraic counterpart of Opt, we have (i) $[\![Q_1]\!]_D = \emptyset \rightarrow [\![Q_1]\!]_D \bowtie [\![Q_2]\!]_D = \emptyset$ and (ii) if there is some $\mu \in [\![Q_1]\!]_D$, then there also is some $\mu' \in [\![Q_1]\!]_D \bowtie [\![Q_2]\!]_D$.

Lemma 4.14(4): Follows from the semantics of Ask queries, the semantics of operator And, and Proposition 4.1. The crucial observation is that precondition $pVars([\![Q_1]\!]_D) \cap pVars([\![Q_2]\!]_D) = \emptyset$ together with Proposition 4.1 implies that for each pair of mappings $(\mu_1, \mu_2) \in [\![Q_1]\!]_D \times [\![Q_2]\!]_D$ it holds that $dom(\mu_1) \subseteq pVars([\![Q_1]\!]_D)$, $dom(\mu_2) \subseteq pVars([\![Q_2]\!]_D)$, and therefore $dom(\mu_1) \cap dom(\mu_2) = \emptyset$. □

The first claim of the lemma shows that, for Ask queries, it does not matter whether bag or set semantics is used for evaluating the inner expression. The remaining claims indicate cases where we can individually evaluate subqueries, which may help to reduce evaluation costs. For instance, to compute $[\![\text{Ask}(Q_1 \text{ Union } Q_2)]\!]_D$ we could choose the simpler expression among Q_1 and Q_2 (e.g., based on a cost function), evaluate this expression, and check if it evaluates to *true*. In that case, we could skip the computation of the second expressions and directly return *true*. This rule also could be combined with the union normal form proposed in [PAG06a] (cf. the discussion in Section 4.2.3): first, we transform the expression into a disjunction of union-free subexpressions and then decompose this disjunction by repeated application of Lemma 4.14(2) into a sequence of $\vee$-connected Ask queries.

Interesting is also the rule in Lemma 4.14(3) for operator Opt, which shows that top-level Opt-expressions can simply be replaced by the left side expression, thus saving the cost for computing the right side expression. Note, however, that it is generally not valid to drop Opt-expressions that are encapsulated in the query (i.e. do not stand at the top-level), as witnessed by the following example.

Example 4.17 Consider document $D := \{(c, c, c)\}$ and the SPARQL expressions

$Q_1 := ((c, c, ?x)$ OPT $(c, c, ?y))$ FILTER $(?y = c)$,
$Q_2 := (c, c, ?x)$ FILTER $(?y = c)$.

We can observe that $[\![\text{ASK}(Q_1)]\!]_D = \mathit{true}$, but $[\![\text{ASK}(Q_2)]\!]_D = \mathit{false}$. □

Finally, Lemma 4.14(4) shows that we can individually evaluate AND-connected subqueries that do not contain shared possible variables. The benefit of this rule is twofold: first, it allows to abort the computation if one of these subqueries, say Q_i, contains no result (i.e., if $[\![\text{ASK}(Q_i)]\!]_D = \mathit{false}$); second, we avoid the computation of the join between such subqueries, which can be seen as a cartesian product (because left and right side mappings never contain shared variables). This may sustainably reduce the size of intermediate results and speed up query evaluation. The rule may be particularly useful for ASK queries that check for a conjunction of independent conditions, expressed in SPARQL as a set of SPARQL expressions $Q_1, \ldots, Q_n$ with pairwise distinct variables that are interconnected through operator AND.

4.4.2. SPARQL DISTINCT and REDUCED Queries

Having discussed ASK queries, we now turn towards an investigation of the solution modifiers DISTINCT and REDUCED, which were informally discussed earlier in Section 2.3.4. Before presenting our results, we formally define their semantics:

Definition 4.11 (SELECT DISTINCT Query) Let Q be a SPARQL expression and $S \subset V$. A *SPARQL* SELECT DISTINCT *query* is an expression of the form SELECT DISTINCT$_S(Q)$. We extend the bag semantics from Definition 2.15 to SELECT DISTINCT queries as follows. Let $(\Omega^+, m^+) := [\![\text{SELECT}_S(Q)]\!]^+_D$. We define $[\![\text{SELECT DISTINCT}_S(Q)]\!]^+_D := (\Omega^+, m)$, where function m is defined as $m(\mu) := 1$ if $m^+(\mu) \geq 1$ and $m(\mu) := 0$ otherwise. □

Note that duplicates do not occur under set semantics, so we defined only the bag semantics (and we will use the SELECT DISTINCT query form only in this context). While SELECT DISTINCT queries eliminate duplicates from the evaluation result, the REDUCED query form gives a greater amount of freedom to the query optimizer in that it allows to eliminate (an arbitrary number of) duplicates. From a theoretical point of view, the result of a REDUCED query can be understood as a set of mapping sets, where each mapping set that is contained in the result describes a possible solution. In practice, though, the query engines does not need to compute all possible solutions, but can compute and return the solution that best fits its internal processing strategy. We next formalize SELECT REDUCED queries.

Definition 4.12 (SELECT REDUCED Query) Let Q be a SPARQL expression and $S \subset V$. A *SPARQL* SELECT REDUCED *query* is an expression of the form SELECT REDUCED$_S(Q)$. We extend the bag semantics from Definition 2.15 to SELECT REDUCED queries as follows. Let $(\Omega^+, m^+) := [\![\text{SELECT}_S(Q)]\!]^+_D$. The solution to query SELECT REDUCED$_S(Q)$ is defined as the set of mapping sets of the form (Ω^+, m) such that for all $\mu \in \mathcal{M}$ it holds that (i) $m^+(\mu) = 0 \rightarrow m(\mu) = 0$ and (ii) $m^+(\mu) > 0 \rightarrow m(\mu) \geq 1 \wedge m(\mu) \leq m^+(\mu)$. □

Note that SELECT REDUCED queries, like SELECT DISTINCT queries, are defined only for SPARQL bag algebra. We illustrate the previous definitions by example:

Example 4.18 Consider the SPARQL expression $Q := (c, c, ?x)$ UNION $(c, c, ?x)$, query $Q_1 :=$ SELECT$_{?x}(Q)$, DISTINCT query $Q_2 :=$ SELECT DISTINCT$_{?x}(Q)$, REDUCED query $Q_3 :=$ SELECT REDUCED$_{?x}(Q)$, and document $D := \{(c, c, c)\}$. Then

$[\![Q_1]\!]^+_D = (\{\{?x \mapsto c\}\}, m_1)$ where for $\mu \in \mathcal{M}$ function m_1 is defined as $m_1(\mu) := 2$ if $\mu = \{?x \mapsto c\}$ and $m_1(\mu) := 0$ otherwise,
$[\![Q_2]\!]^+_D = (\{\{?x \mapsto c\}\}, m_2)$ where for $\mu \in \mathcal{M}$ function m_2 is defined as $m_2(\mu) := 1$ if $\mu = \{?x \mapsto c\}$ and $m_2(\mu) := 0$ otherwise,
$[\![Q_3]\!]^+_D = \{(\{\{?x \mapsto c\}\}, m_1), (\{\{?x \mapsto c\}\}, m_2)\}$. □

The observant reader may have noticed that there is a close connection between SELECT DISTINCT queries for SPARQL bag semantics and simple SELECT queries for set semantics. The following lemma formalizes this and other connections:

Lemma 4.15 Let Q be a SPARQL expression and $S \subset V$. Then

1. $[\![\text{SELECT}_S(Q)]\!]_D \cong [\![\text{SELECT DISTINCT}_S(Q)]\!]^+_D$,
2. $[\![\text{SELECT DISTINCT}_S(Q)]\!]^+_D \in [\![\text{SELECT REDUCED}_S(Q)]\!]^+_D$, and
3. There is $(\Omega, m) \in [\![\text{SELECT REDUCED}_S(Q)]\!]^+_D$ s.t. $[\![\text{SELECT}_S(Q)]\!]_D \cong (\Omega, m)$. □

Proof of Lemma 4.15.
Lemma 4.15(1): Follows from the definition of SELECT DISTINCT queries and Lemma 3.1, which shows that bag and set semantics coincide w.r.t. mapping sets.
Lemma 4.15(2): Follows from the definition of the SELECT DISTINCT and SELECT REDUCED query forms, i.e. it is easily shown that the definition of function m in the SELECT DISTINCT query form satisfies the two conditions (i) and (ii) that are enforced for function m in the definition of SELECT REDUCED queries.
Lemma 4.15(3): Follows when combining claims (1) and (2) of the lemma. □

The lemma has some direct practical implications: it shows that, for both SELECT DISTINCT and SELECT REDUCED queries (posed in the context of bag semantics), engines can fall back on the simpler set semantics for query evaluation. Hence, for

such queries no cardinality computation is required at all and duplicate solutions can be discarded in each intermediate computation step. We conclude our discussion of algebraic SPARQL rewriting with a lemma that identifies another large class of queries that can be safely evaluated using set semantics rather than bag semantics (we reuse the notation for SPARQL fragments introduced in Section 3.2):

Lemma 4.16 Let $Q \in \mathcal{AFO}$ and $S \supseteq pVars([\![Q]\!]_D)$. Then

1. $[\![Q]\!]_D \cong [\![Q]\!]_D^+$ and
2. $[\![\textsc{Select}_S(Q)]\!]_D \cong [\![\textsc{Select}_S(Q)]\!]_D^+$. □

Proof of Lemma 4.16

Lemma 4.16(1): Follows from the observation that $Q \in \mathcal{AFO}^+ \rightarrow [\![Q]\!]_D^+ \in \widetilde{\mathbb{A}^+}$ (when interpreting $[\![Q]\!]_D^+$ as an expression), combined with Lemma 4.6.

Lemma 4.16(2): Follows from Lemma 4.16(1) and the observation that the projection for $S \supseteq pVars([\![Q]\!]_D)$ does not modify the evaluation result. □

Combined with the observation that we can safely use set semantics for Ask, Select Distinct, and Select Reduced queries, we conclude that set semantics is applicable in the context of a large and practical class of queries. Ultimately, engines that rely on SPARQL bag algebra for query evaluation may implement a separate module for set semantics – with simpler data structures and possibly more efficient operations – and switch between these modules based on the above results.

4.5. Related Work

Query Optimization in the Relational Context. Query optimization for the relational context has been a central topic of database research since its beginning. We refer the interested reader to [JK84; Cha98; AHV] for top-level surveys of optimization approaches that have been proposed for relational systems; an in-depth discussion of all these approaches is beyond the scope of this section. Most related to our work here are probably [Hal75; SC75; Tod75], which were (to the best of our knowledge) the first works to study and exploit equivalences over relational algebra. Rewriting approaches for expressions involving the relational left outer join operator, which can be seen as the relational counterpart of SPARQL operator Opt, were investigated in [GLR97]. We emphasize that, although all these studies form valuable groundwork for the investigations presented in this chapter, their results do not trivially carry over to SPARQL algebra. To give an example, the filter pushing rules from Sections 4.2.5 and 4.3.2 were inspired by the well-known filter pushing rules for relational algebra proposed in [Hal75; Tod75]; however, the specification of rules over SPARQL algebra is considerably complicated by the issue of bound and

unbound variables (whereas in relational algebra the underlying schema is fixed) and we have seen in Section 4.2.5 that the resulting filter pushing rules for SPARQL algebra heavily rely on our novel concepts of certain and possible variables.

SPARQL Query Optimization. Also in the context of the SPARQL query language a variety of optimization approaches have been proposed. Closely related to the study in this chapter is the work in [PAG06a; PAG09], which (amongst others) presents a set of rewriting rules over SPARQL abstract syntax expressions, used to transform SPARQL expressions into the so-called *union normal form*, an equivalent representation of the expression in form of a disjunction of UNION-free subqueries. Although stated at syntax level in [PAG06a], these rules carry immediately over to SPARQL set algebra. Apart from this goal-driven rewriting scheme (which comprises only few algebraic equivalences) a closer investigation of [PAG06a] reveals some more algebraic equivalences that are implicitly used in proofs there. We refer the reader back to the beginning of Section 4.2, where we listed all equivalences established in [PAG06a]. To conclude the discussion of the latter work, we want to remark that the focus of [PAG06a] is not on algebraic SPARQL rewriting, but on a study of two different semantics and normal forms for SPARQL expressions. The initial results on SPARQL rewriting can be seen as a by-product of the proper contributions. With the focus on equivalences over SPARQL algebra expressions, the investigation in this chapter was much more detailed and therefore considerably extends previous investigations. Going beyond the rewriting of set algebra expressions, we also have explored the rewriting of bag algebra expressions and syntax expressions evaluated under bag semantics, which gives our results immediate practical relevance.

Another notable optimization approach for SPARQL is the reordering of graph patterns based on selectivity estimations proposed in [BKS07; SSB+08]. The rewriting of SPARQL queries presented there has a strong focus on AND-connected blocks inside the query and relies on the commutativity and associativity of the join operator (cf. rules *(JComm)* and *(JAss)* from Figure 4.4 in our framework). In addition, in [BKS07] the decomposition and elimination of filter expressions, similar in style to rules *(FDecompI)* from Figure 4.4 and *(FElimII)* from Lemma 4.3 has been proposed. However, there the idea is only described by example and conditions when such rewritings are possible (which we provided in this thesis) are missing. In summary, [BKS07; SSB+08] rely on a small subset of the rules proposed in this chapter and can be seen as a first step towards efficient SPARQL query optimization. With our results, we lay foundations for advanced approaches beyond AND-only queries.

The work in [HH07] proposes a query graph model for SPARQL. In a first step, the query is translated into a graphical model that reflects the operators and the data flow between those operators. The authors propose to manipulate the query graph model to obtain more efficient query plans using so-called transformation rules. These rules are exemplified by means of a rule that allows to merge triple patterns that share a variable into a single operator, which can then be evaluated more efficiently. Unfortunately, the discussion of further transformation rules is not

subject of the paper. One may expect that our algebraic rewritings can be transferred into the context of the query graph model (i.e., can be expressed as transformation rules), so our algebraic results would be beneficial for suchlike approaches as well.

Next, in [GGK09] rewriting rules for manipulating filter conditions at abstract syntax level were proposed, some of which are overlapping with the filter rewriting rules presented in this chapter. We therefore emphasize that the latter work has been published after we made our results available (as technical report) in [SML08].

Another major line of research in the context of SPARQL query optimization is the investigation of specialized indices for RDF data [HD05; GGL07; WKB08]. In [GGL07], an index of precomputed joins for RDF is proposed; the experimental results show that the approach does not scale to large RDF data sets. Similar by idea, [HD05] proposes a combination of indices for the efficient processing of RDF data, such as keyword indices and quad indices (which, in addition to the three components of RDF triples, take the RDF document id into account). Further, [WKB08] proposes different combinations of linked lists to provide efficient access to RDF triples and (as a consequent enhancement of ideas found in [AMMH07]) supports fast merge joins when computing any kind of join between two triple patterns. One might expect that, like in the context of other data formats like the relational model or XML, indices for RDF are an important step towards efficient RDF processing, as they provide fast data access paths. A comprehensive optimization scheme may use them complementarily to the algebraic rules presented in this chapter and optimally would build upon the (estimated) efficiency of access paths to the RDF data.

Going into a similar direction as indices, in [CPST03] the labeling of large RDF Schema descriptions has been proposed, akin to XML labeling schemes for the efficient XPath [xpa] or XQuery [xqu] processing using native relational database systems (such as [LM01; Gru02]). The latter work studies the efficiency of different labeling schemes in order to accelerate path queries along the RDFS subsumption hierarchy; with this goal, the approach is not SPARQL-specific, yet constitutes an interesting idea that has not been explored before in the context of RDF data.

Much research effort has been spent in processing RDF data with traditional systems, such as relational DBMSs or datalog engines [TCK05; CDES05; AMMH07; FB08; NW08; SGK+08; SHK+08; WKB08; Pol07], thus falling back on established optimization strategies. The first line of research into this direction are proposals for mappings from SPARQL or SPARQL algebra to the relational data model [Cyg05; CLJF06]. From a more practical point of view, in [CDES05; TCK05] different storage schemes for RDF data were discussed, starting from simple triple tables (which contain all the triples of the RDF graph), to more advanced schemes like partitioning along the predicate values (thus essentially following the binary relation view on RDF) and clustering of data. The second, predicate partitioning scheme was reinvestigated in [AMMH07], where it is shown that in such a setting (called *Vertical Partitioning* there) efficient merge joins often can be exploited for the efficient RDF data processing using relational DBMS. Subsequent work identifies dif-

ferent deficiencies of the Vertical Partitioning scheme [SGK+08; SHK+08; WKB08] and proposes more advanced storage schemes for RDF. In fact, one can observe that the current trend of RDF data management goes away from using (and customizing) existing relational data management systems, to more specialized, native solutions [NW08; WKB08; NW09]. One reason for this might be that, as shown in [SHK+08], existing RDF data processing schemes that build upon relational systems experience severe performance bottlenecks, in particular for complex queries (such as queries involving negation). This indicates that traditional approaches are not laid out for the specific challenges that come along with SPARQL processing and served as an additional motivation for the investigation in this chapter. For instance, with our rewriting scheme for simulated negation presented in Section 4.2.6, we tackle the problem of the efficient evaluation of SPARQL negation queries.

Concerning the latter issue, it has been shown in [AG08a] that the algebraic minus operation can be simulated at the syntax level, essentially using operators Opt, Filter, and predicate *bnd*, as discussed in Section 4.2.6. The encoding scheme presented there was theoretically motivated; our approach to rewriting closed world negation presented in Section 4.2.6 is practically motivated and goes in the other direction: we presented a rewriting scheme to transform algebra expressions involving simulated negation into simple negation expressions using only the minus operator.

4.6. Conclusion

We have presented a large set of rewriting rules for SPARQL algebra, covering algebraic laws like idempotence, inverse, commutativity, associativity and distributivity, advanced optimization techniques like filter and projection pushing, as well as the rewriting of queries involving negation. Taken together, our rules establish a powerful framework for manipulating algebra expressions. We emphasize that, although the rules have been established in the context of SPARQL algebra, most of them can easily be transferred to SPARQL syntax. In particular, each rule that involves only operators $\bowtie$, $\cup$, σ, and $\ltimes$ directly carries over to the syntax level and – due to the close connection between SPARQL syntax and SPARQL algebra stated in Definition 2.11 (for set semantics) and Definition 2.15 (for bag semantics) – can be expressed at syntax level using the corresponding operators And, Union, Filter, and Opt, respectively.[4] To give an example, rule *(FJPush)* from Figure 4.4 can be expressed at syntax level for expressions Q_1, Q_2 and filter condition R as

$$(Q_1 \text{ And } Q_2) \text{ Filter } R \equiv (Q_1 \text{ Filter } R) \text{ And } Q_2.$$

[4]Care must be taken for equivalences involving the negation operator $\setminus$, which has no syntactic counterpart, and projection π, because its syntactic counterpart, operator Select, is always placed at the top-level of SPARQL queries (cf. Definition 2.5).

It holds whenever for all $?x \in vars(R)$:$?x \in cVars([\![Q_1]\!]_D^+) \vee ?x \notin pVars([\![Q_2]\!]_D^+)$. Assuming bag semantics, the correctness follows directly from the semantics in Definition 2.15, which translates the left side of the equation into $\sigma_R([\![Q_1]\!]_D^+ \bowtie [\![Q_2]\!]_D^+)$, the right side into $\sigma_R([\![Q_1]\!]_D^+) \bowtie [\![Q_2]\!]_D^+$, and both bag algebra expressions are known to be equivalent according to rule *(FJPush$^+$)* if the above precondition holds.

With this observation in mind, most of our algebraic rewriting rules also are useful for engines that rely on evaluation mechanisms that differ from the algebraic evaluation approach (whenever they are equivalent, of course). To give a concrete example, mapping schemes that build upon a translation of SPARQL queries into the relational context or datalog, such as [AMMH07; Pol07; FB08; NW08; SGK$^+$08; SHK$^+$08; WKB08]), may first transform the SPARQL syntax expression according to syntactic rules derived from our algebraic rules, then translate and evaluate it.

We again want to emphasize that, although we picked up several ideas from established relational algebra optimization approaches, such as filter and projection pushing, most equivalences fundamentally rely on characteristics of SPARQL algebra. With the novel concepts of possible and certain variables we introduced a useful tool to deal with these characteristics. The integration of our rules into an automated optimization scheme based on heuristics and/or cost estimation functions is left as future work. In our opinion, nothing speaks against the assumption that SPARQL optimizers will benefit from our rewriting techniques in the same way SQL optimizers benefit from relational algebra equivalences.

Group	Rule	Name	$\mathbb{A}$	$\widetilde{\mathbb{A}}$	$\mathbb{A}^+$	$\widetilde{\mathbb{A}^+}$
I	$A \cup A \equiv A$	*UIdem*	✓	✓	–	–
	$A \bowtie A \equiv A$	*JIdem*	–	✓	–	✓
	$A \bowtie\!\!\!\!- A \equiv A$	*LIdem*	–	✓	–	✓
	$A \setminus A \equiv \emptyset$	*Inv*	✓	✓	✓	✓
II	$(A_1 \cup A_2) \cup A_3 \equiv A_1 \cup (A_2 \cup A_3)$	*UAss*	✓	✓	✓	✓
	$(A_1 \bowtie A_2) \bowtie A_3 \equiv A_1 \bowtie (A_2 \bowtie A_3)$	*JAss*	✓	✓	✓	✓
III	$A_1 \cup A_2 \equiv A_2 \cup A_1$	*UComm*	✓	✓	✓	✓
	$A_1 \bowtie A_2 \equiv A_2 \bowtie A_1$	*JComm*	✓	✓	✓	✓
IV	$(A_1 \cup A_2) \bowtie A_3 \equiv (A_1 \bowtie A_3) \cup (A_2 \bowtie A_3)$	*JUDistR*	✓	✓	✓	✓
	$A_1 \bowtie (A_2 \cup A_3) \equiv (A_1 \bowtie A_2) \cup (A_1 \bowtie A_3)$	*JUDistL*	✓	✓	✓	✓
	$(A_1 \cup A_2) \setminus A_3 \equiv (A_1 \setminus A_3) \cup (A_2 \setminus A_3)$	*MUDistR*	✓	✓	✓	✓
	$(A_1 \cup A_2) \bowtie\!\!\!\!- A_3 \equiv (A_1 \bowtie\!\!\!\!- A_3) \cup (A_2 \bowtie\!\!\!\!- A_3)$	*LUDistR*	✓	✓	✓	✓
V	$\pi_{pVars(A) \cup S}(A) \equiv A$	*PBaseI*	✓	✓	✓	✓
	$\pi_S(A) \equiv \pi_{S \cap pVars(A)}(A)$	*PBaseII*	✓	✓	✓	✓
	Let $S' := S \cup (pVars(A_1) \cap pVars(A_2))$, $S'' := pVars(A_1) \cap pVars(A_2)$. Then					
	$\pi_S(A_1 \cup A_2) \equiv \pi_S(A_1) \cup \pi_S(A_2)$	*PUPush*	✓	✓	✓	✓
	$\pi_S(A_1 \bowtie A_2) \equiv \pi_S(\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2))$	*PJPush*	✓	✓	✓	✓
	$\pi_S(A_1 \setminus A_2) \equiv \pi_S(\pi_{S'}(A_1) \setminus \pi_{S''}(A_2))$	*PMPush*	✓	✓	✓	✓
	$\pi_S(A_1 \bowtie\!\!\!\!- A_2) \equiv \pi_S(\pi_{S'}(A_1) \bowtie\!\!\!\!- \pi_{S'}(A_2))$	*PLPush*	✓	✓	✓	✓
	$\pi_S(\sigma_R(A)) \equiv \pi_S(\sigma_R(\pi_{S \cup vars(R)}(A)))$	*PFPush*	✓	✓	✓	✓
	$\pi_{S_1}(\pi_{S_2}(A)) \equiv \pi_{S_1 \cap S_2}(A)$	*PMerge*	✓	✓	✓	✓
VI	$\sigma_{R_1 \wedge R_2}(A) \equiv \sigma_{R_1}(\sigma_{R_2}(A))$	*FDecompI*	✓	✓	✓	✓
	$\sigma_{R_1 \vee R_2}(A) \equiv \sigma_{R_1}(A) \cup \sigma_{R_2}(A)$	*FDecompII*	✓	✓	–	–
	$\sigma_{R_1}(\sigma_{R_2}(A)) \equiv \sigma_{R_2}(\sigma_{R_1}(A))$	*FReord*	✓	✓	✓	✓
	$\sigma_{bnd(?x)}(A) \equiv A$, if $?x \in cVars(A)$	*FBndI*	✓	✓	✓	✓
	$\sigma_{bnd(?x)}(A) \equiv \emptyset$, if $?x \notin pVars(A)$	*FBndII*	✓	✓	✓	✓
	$\sigma_{\neg bnd(?x)}(A) \equiv \emptyset$, if $?x \in cVars(A)$	*FBndIII*	✓	✓	✓	✓
	$\sigma_{\neg bnd(?x)}(A) \equiv A$, if $?x \notin pVars(A)$	*FBndIV*	✓	✓	✓	✓
VII	$\sigma_R(A_1 \cup A_2) \equiv \sigma_R(A_1) \cup \sigma_R(A_2)$	*FUPush*	✓	✓	✓	✓
	$\sigma_R(A_1 \setminus A_2) \equiv \sigma_R(A_1) \setminus A_2$	*FMPush*	✓	✓	✓	✓
	If for all $?x \in vars(R)$ it holds that $?x \in cVars(A_1) \vee ?x \notin pVars(A_2)$, then					
	$\sigma_R(A_1 \bowtie A_2) \equiv \sigma_R(A_1) \bowtie A_2$	*FJPush*	✓	✓	✓	✓
	$\sigma_R(A_1 \bowtie\!\!\!\!- A_2) \equiv \sigma_R(A_1) \bowtie\!\!\!\!- A_2$	*FLPush*	✓	✓	✓	✓
Prop. 4.4	$(A_1 \setminus A_2) \setminus A_3 \equiv (A_1 \setminus A_3) \setminus A_2$	*MReord*	✓	✓	✓	✓
	$(A_1 \setminus A_2) \setminus A_3 \equiv A_1 \setminus (A_2 \cup A_3)$	*MMUCorr*	✓	✓	✓	✓
	$A_1 \setminus A_2 \equiv A_1 \setminus (A_1 \bowtie A_2)$	*MJ*	✓	✓	✓	✓
	$A_1 \bowtie\!\!\!\!- A_2 \equiv A_1 \bowtie\!\!\!\!- (A_1 \bowtie A_2)$	*LJ*	–	✓	–	✓
Lemma 4.4	Let $?x \in cVars(A_2) \setminus pVars(A_1)$ be a variable. Then					
	$\sigma_{\neg bnd(?x)}(A_1 \bowtie\!\!\!\!- A_2) \equiv A_1 \setminus A_2$	*FLBndI*	✓	✓	✓	✓
	$\sigma_{bnd(?x)}(A_1 \bowtie\!\!\!\!- A_2) \equiv A_1 \bowtie A_2$	*FLBndII*	✓	✓	✓	✓

Table 4.1.: Survey of algebraic equivalences: R, R_1, R_2 are conditions, S, S_1, S_2 sets of variables. The four algebra fragment columns $\mathbb{A}$, $\widetilde{\mathbb{A}}$, $\mathbb{A}^+$, $\widetilde{\mathbb{A}^+}$ indicate fragments for which the rules are valid (✓) or invalid (–).

Chapter 5.

Constraints for RDF and Semantic SPARQL Query Optimization

Roy: "*So we're done with SPARQL query optimization. What's next?*"
Moss: "*Hold on a second, we only discussed algebraic optimization. I would expect that semantic query optimization is particularly promising in the context of SPARQL and RDFS data.*"
Jen: "*Why do you think so?*"
Moss: "*Well... First, constraints may compensate for the homogeneous structure of RDF databases. Second, there are already some implicit constraints by the semantics of RDFS, right?*"
Roy: "*Oh, I see... Sounds interesting! Let's have a look at this!*"

The study of integrity constraints such as keys, foreign keys, and functional dependencies (also known as data dependencies in the literature) has received considerable attention in the relational context; to give only few examples, early work in this area includes [Cod70; Cod71; Cod74; HM75; Che76; Cod79]. Integrity constraints fix vital relationships between data entities that must hold on every database instance and that way restrict the state space of the database. The importance of such constraints is twofold. First of all, they play a major role in schema design [Cod71; Cod74; Che76], where different types of normals forms and normalization techniques based on data dependencies have been developed, with the goal to avoid insertion, update, and deletion anomalies (cf. [Cod71; Cod74; AHV]). Second, beyond their importance for schema design and normalization it is well-known that integrity constraints are valuable input to query optimizers: the knowledge about relationships between data entities often allows to transform the original query into queries that are equivalent on every database that satisfies the set of integrity constraints, but can be evaluated more efficiently (see e.g. [Kin81; BV84; CGM90]).

Coming along with their central role, data dependencies have become an inherent part of relational database systems, where they are implicit in primary and foreign key specifications. In addition, the SQL92 standard provides the CREATE ASSERTION statement, which allows to add user-defined constraints on demand. Due to their importance in the relational model, dependencies also have been studied in the

context of other data formats like XML (see e.g. [FL01; FS03]), OWL [MHS07], and for deductive databases (see e.g. [CGM90; CGL09]). In this chapter, we propose integrity constraints for RDF, highlight their relations to the RDFS inferencing rules, and investigate the role of the SPARQL query language in such a setting. We discuss three major aspects of RDF constraints, which will be sketched in the following.

Firstly, motivated by dependencies that are commonly encountered in the relational context and XML, we propose and formalize several types of data dependencies for RDF, such as keys, foreign keys, and cardinality constraints. Complementarily, we also address the specific needs of the RDFS data format and propose constraints like subclass, subproperty, property domain, and property range specifications. We formalize all these dependencies in first-order logic and classify them as tuple-generating [BV84], equality-generating [BV84], and disjunctive embedded dependencies [DT05], showing that many practical RDF(S) constraints fall into constraints classes that have been studied before in the relational context and XML.

Secondly, we investigate the role of SPARQL as a constraint language. We argue that a natural way to check if a constraint holds on some RDF document D is to write a SPARQL Ask query that checks for constraint violations in the input document. We then investigate whether, given a constraint in form of a first-order logic sentence as input, it is always possible to write a SPARQL Ask that implements this task. Our investigation reveals that there are some minor issues that make it impossible to check first-order logic sentences with SPARQL in the general case. In response, we extend SPARQL by so-called constant mappings (which can be understood as the natural counterparts of constants in relational algebra) to a language which we call SPARQL^C and prove that every first-order sentence can be checked when using this extended version of the query language. The proof of this result makes the close connection between SPARQL^C and first-order logic sentences (over a ternary schema, which stores all the triples of the RDF database) explicit.

Finally, in the third part of the chapter we complement the study of algebraic SPARQL query optimization from the previous chapter with a comprehensive approach to constraint-based query optimization, also known as semantic query optimization (SQO), for the SPARQL query language. As sketched previously, the central idea behind SQO is to exploit integrity constraints in order to find more efficient query evaluation plans that yield the same result on every database instance that satisfies the constraints. The data dependencies that are used for the optimization process could be provided by the user, be automatically extracted from the underlying database, or – in case SPARQL is evaluated on top of an RDFS inference system – may be implicitly given by the semantics of the RDFS vocabulary, which implies a variety of correlations among entities in the RDF database, such as subclass or subproperty relationships (cf. the discussion in Section 2.2.2).

The first part of our semantic optimization approach for SPARQL is to translate SPARQL And-blocks inside queries into conjunctive queries over a ternary relation that is assumed to store all RDF triples. This allows us to optimize the individ-

ual (conjunctive) subqueries using the well-known chase algorithm [MMS79; JK82; BV84], which has been proposed 25+ years ago to check equivalence between conjunctive queries under a set of integrity constraints. In particular, we fall back on the so-called Chase & Backchase (C&B) algorithm [DPT06] for query optimization, which builds upon the standard chase and enumerates all minimal equivalent (under the given set of constraints) queries. The minimized conjunctive queries can then easily be translated back into SPARQL, which gives us minimized SPARQL queries.

The second part of our SQO scheme is highly SPARQL-specific and goes beyond the optimization of SPARQL And-only queries and And-connected subqueries. We provide a collection of optimization rules for different SPARQL operators that allow us to rewrite and optimize queries with more complex operator constellations. Motivated by the high complexity of operator Opt established in Section 3.2.3, we put a special focus on optimizing queries that involve Opt expressions. To give an example, we propose a rewriting rule that allows to replace operator Opt by operator And in cases where the expression inside the Opt clause is implied by the constraint set. In summary, the main contributions of this chapter are the following.

- We formalize a set of practically relevant constraints for RDF in first-order logic, showing that these constraints fall into existing constraint classes (such as tuple-generating, equality-generating, or embedded dependencies) that have been studied extensively in the context of relational databases and XML.
- We investigate the role of SPARQL as a constraint language. Our results show that, with only minor extensions, the SPARQL query language can be used to specify and check all RDF constraints that can be expressed by first-order sentences. This result shows that SPARQL is an excellent candidate for dealing with constraints. From a practical point of view, it motivates the extension of SPARQL by a query form that allows to enforce user-defined constraints over RDF databases, akin to SQL Create Assertion statements for RDBMSs.
- We present an SQO scheme for SPARQL that can be used to optimize queries under a set of constraints over the RDF database. Falling back on the well-known Chase & Backchase algorithm from [DPT06], we derive guarantees for cases in which we can find minimal equivalent subqueries. Going beyond And-only queries, we also tackle SPARQL-specific desiderata, such as the optimization of queries that involve operators Filter and Opt.

Structure. We revisit the necessary background from first-order logic, relational databases, constraints, and the chase algorithm in Section 5.1. Next, in Section 5.2 we introduce and classify different types of constraints for RDF data. Section 5.3 investigates the role of SPARQL as a constraint language. Our semantic query optimization scheme for SPARQL is presented in Section 5.4. We wrap up with a discussion of related work and a short conclusion in Sections 5.5 and 5.6.

5.1. Preliminaries: First-order Logic, Relational Databases, Constraints, and Chase

In the following we introduce the necessary background for the study in the remainder of this chapter. We start with preliminaries from first-order logic in Section 5.1.1, then turn towards relational databases and conjunctive queries in Section 5.1.2, provide background on integrity constraints and their formalization in first-order logic in Section 5.1.3, and introduce the chase algorithm in Section 5.1.4.

5.1.1. First-order Logic

Our introduction to first-order logic follows the formalization in [EFT94]. We define three pairwise disjoint, infinite sets: the set of *constants* Δ, the set of labeled nulls Δ_{null}, and the set of variables V_R.[1] Usually, we use identifiers x, y, z to denote variables in V_R, letters c, d for constants from Δ, and n_1, n_2, ... to denote elements from the set Δ_{null} of labeled nulls. We start with the definition of signatures:

Definition 5.1 (Relational Signature) A *relational signature* $\mathcal{R}$ is a finite set of relational symbols $\{R_1, ..., R_n\}$, where we assign to each relational symbol $R_i \in \mathcal{R}$ a number $ar(R_i) \in \mathbb{N}$, called *arity* of R_i. When used in the context of relational databases, we often call a relational signature *database schema*.

We use the term *position* to refer to the positions of a relational symbol $R_i \in \mathcal{R}$. If $ar(R_i) = n$, then R_i has n positions, which we denote by $R_i^1, \ldots, R_i^n$. □

Building upon relational signatures, we next introduce the concept of $\mathcal{R}$-formulas:

Definition 5.2 ($\mathcal{R}$-formula) Let $\mathcal{R}$ be a relational signature. An *$\mathcal{R}$-formula* is an expression that is obtained by a finite number of applications of the following rules.

- If $t_1, t_2 \in \Delta \cup V_R$, then $t_1 = t_2$ is an $\mathcal{R}$-formula.
- If $R \in \mathcal{R}$ and $t_1, \ldots, t_{ar(R)} \in \Delta \cup V_R$, then $R(t_1, \ldots, t_{ar(R)})$ is an $\mathcal{R}$-formula.
- If φ is an $\mathcal{R}$-formula, then $\neg\varphi$ is an $\mathcal{R}$-formula.
- If φ, ψ are $\mathcal{R}$-formulas, then $(\varphi \vee \psi)$ and $(\varphi \wedge \psi)$ are $\mathcal{R}$-formulas.
- If φ is an $\mathcal{R}$-formula and $x \in V_R$, then $\exists x\varphi$ and $\forall x\varphi$ are $\mathcal{R}$-formulas.

We call $\mathcal{R}$-formulas of the form $R_i(t_1, \ldots, t_{ar(R_i)})$ (i.e., those obtained by application of the second bullet) *$\mathcal{R}$-atoms*, formulas of the form $t_1 = t_2$ *equality $\mathcal{R}$-formulas*, and formulas of the form $\neg t_1 = t_2$ *non-equality $\mathcal{R}$-formulas*. □

We next define free variables in $\mathcal{R}$-formulas and the related notion of sentences:

[1] We use V_R to distinguish this set from the set V of SPARQL variables.

Definition 5.3 (Free Variables of $\mathcal{R}$-formulas) For any $\mathcal{R}$-formula φ, the set of free variables, $\mathit{free}(\varphi)$, is inductively defined as follows.

$$
\begin{array}{ll}
\mathit{free}(t_1 = t_2) & := \{t_1, t_2\} \cap V_R \\
\mathit{free}(R(t_1, ..., t_{ar(R)})) & := \{t_1, ..., t_{ar(R)}\} \cap V_R \\
\mathit{free}(\neg\psi) & := \mathit{free}(\psi) \\
\mathit{free}((\psi_1 \vee \psi_2)) & := \mathit{free}(\psi_1) \cup \mathit{free}(\psi_2) \\
\mathit{free}((\psi_1 \wedge \psi_2)) & := \mathit{free}(\psi_1) \cup \mathit{free}(\psi_2) \\
\mathit{free}(\forall x\psi) & := \mathit{free}(\psi)\backslash\{x\} \\
\mathit{free}(\exists x\psi) & := \mathit{free}(\psi)\backslash\{x\}
\end{array}
$$

□

Definition 5.4 ($\mathcal{R}$-sentence) Let φ be an $\mathcal{R}$-formula. Then φ is called $\mathcal{R}$-*sentence*, or simply *sentence* if $\mathcal{R}$ is known from the context, iff $\mathit{free}(\varphi) = \emptyset$. □

Structures, Interpretations, and Satisfaction. Having introduced the syntax of first-order formulas, we now come to a formal definition of their semantics.

Definition 5.5 ($\mathcal{R}$-structure) An $\mathcal{R}$-structure is a pair $I := (\Delta \cup \Delta_{null}, a)$, where symbol a denotes a function that is defined on $\mathcal{R}$ and assigns to every relational symbol $R \in \mathcal{R}$ a subset of $(\Delta \cup \Delta_{null})^{ar(R)}$, i.e. $a(R) \subseteq (\Delta \cup \Delta_{null})^{ar(R)}$. □

We introduce some additional notation. Given a relational symbol $R \in \mathcal{R}$, we often write R^I instead of $a(R)$. Further, by $R(t_1, \ldots, t_{ar(R)})$ we denote the *fact* that $(t_1, \ldots, t_{ar(R)}) \in R^I$. Given a signature $\mathcal{R} := \{R_1, \ldots, R_n\}$, we shall use the previous notation and denote a structure $I := (\Delta \cup \Delta_{null}, a)$ as $(\Delta \cup \Delta_{null}, R_1^I \cup \cdots \cup R_n^I)$, where we represent each R_i^I (for $1 \leq i \leq n$) as the set of its facts.

Example 5.1 Let $\mathcal{R} := \{S, T\}$ with $ar(S) := 1$ and $ar(T) := 2$. Consider the $\mathcal{R}$-structure $I := (\Delta \cup \Delta_{null}, a)$ where $a(S) := \{(a), (b)\}$, $a(T) := \{(a, b)\}$. The structure contains three facts, namely $S(a)$, $S(b)$, and $T(a, b)$. Using the notation introduced above, we denote I as $(\Delta \cup \Delta_{null}, \{S(a), S(b), T(a, b)\})$. □

Definition 5.6 (Variable Assignment, Interpretation) An *assignment of variables* in an $\mathcal{R}$-structure I is a mapping $\gamma : V_R \to \Delta \cup \Delta_{null}$. An $\mathcal{R}$-*interpretation* $\mathcal{I}$ is a pair (I, γ), where I is an $\mathcal{R}$-structure and γ an assignment of variables in I.

If γ is an assignment of variables in I, then we denote by $\gamma\frac{a}{x}$ the assignment that maps x to a and coincides with γ for all variables different from x. Similarly, given an interpretation $\mathcal{I} := (I, \gamma)$, we write $\mathcal{I}\frac{a}{x}$ for the interpretation $(I, \gamma\frac{a}{x})$. □

We are now in the position to define the semantics of $\mathcal{R}$-formulas. The definition is folklore and works by induction on the structure of such formulas:

Definition 5.7 (Satisfaction of $\mathcal{R}$-formulas by $\mathcal{R}$-interpretations) Consider an $\mathcal{R}$-interpretation $\mathcal{I} := (I, \gamma)$. For every $x \in V_R$ we define $\mathcal{I}(x) := \gamma(x)$ and for every constant $c \in \Delta$ we put $\mathcal{I}(c) := c$. We define the notion of *satisfaction* of $\mathcal{R}$-formulas by $\mathcal{R}$-interpretations by induction on the structure of $\mathcal{R}$-formulas:

- $\mathcal{I} \models t_1 = t_2 \Leftrightarrow \mathcal{I}(t_1) = \mathcal{I}(t_2)$,
- $\mathcal{I} \models R(t_1, ..., t_{ar(R)}) \Leftrightarrow (\mathcal{I}(t_1), ..., \mathcal{I}(t_{ar(R)})) \in R^I$,
- $\mathcal{I} \models \neg\varphi \Leftrightarrow$ not $\mathcal{I} \models \varphi$,
- $\mathcal{I} \models (\varphi \vee \psi) \Leftrightarrow \mathcal{I} \models \varphi$ or $\mathcal{I} \models \psi$,
- $\mathcal{I} \models (\varphi \wedge \psi) \Leftrightarrow \mathcal{I} \models \varphi$ and $\mathcal{I} \models \psi$,
- $\mathcal{I} \models \exists x\varphi \Leftrightarrow$ there is some $a \in \Delta \cup \Delta_{null}$ such that $\mathcal{I}\frac{a}{x} \models \varphi$, and
- $\mathcal{I} \models \forall x\varphi \Leftrightarrow$ for all $a \in \Delta \cup \Delta_{null}$ it holds that $\mathcal{I}\frac{a}{x} \models \varphi$.

For a set Σ of $\mathcal{R}$-formulas we write $\mathcal{I} \models \Sigma$ iff for all $\varphi \in \Sigma : \mathcal{I} \models \varphi$. □

From the coincidence lemma in classical logic [EFT94] we know that if an $\mathcal{R}$-formula φ is an $\mathcal{R}$-sentence, then for every $\mathcal{R}$-structure I and each two assignments of variables γ_1, γ_2 in I it holds that

$$(I, \gamma_1) \models \varphi \Leftrightarrow (I, \gamma_2) \models \varphi. \tag{5.1}$$

This equation implies that for every sentence φ and $\mathcal{R}$-structure I there is either no variable assignment γ such that $(I, \gamma) \models \varphi$ or it holds that $(I, \gamma) \models \varphi$ for every variable assignment γ. If the second case applies, we shall also write $I \models \varphi$, thus implicitly stating that $(I, \gamma) \models \varphi$ for every variable assignment γ.

Notational Conventions and Shortcuts. We define the common shortcut $(\varphi \rightarrow \psi) := (\neg\varphi \vee \psi)$ and write $t_1 \neq t_2$ for $\neg t_1 = t_2$. Usually, we abbreviate a sequence of quantifiers $\exists x_1 \ldots \exists x_n\varphi$ (or $\forall x_1 \ldots \forall x_n\varphi$) as $\exists x_1, \ldots, x_n\varphi$ (respectively, $\forall x_1, \ldots, x_n\varphi$). Similarly, if $\overline{x} := (x_1, \ldots, x_n)$ is a tuple of variables, we often write $\exists\overline{x}\varphi$ and $\forall\overline{x}\varphi$ in place of $\exists x_1 \ldots \exists x_n\varphi$ and $\forall x_1 \ldots \forall x_n\varphi$, respectively.

To improve the readability of $\mathcal{R}$-formulas, we will use brackets liberally. We assume that $\wedge$ and $\vee$ are left-associative and define the following precedence order: (i) $\neg$ binds stronger than $\wedge$, $\vee$; (ii) $\wedge$, $\vee$ bind stronger than $\forall$, $\exists$; (iii) $\forall$, $\exists$ bind stronger than $\rightarrow$. Occasionally, we use brackets redundantly, to facilitate reading.

If a formula φ contains at most variables from $\overline{x}$, we denote this by writing $\varphi(\overline{x})$; note that this does not mean that all variables occurring in $\overline{x}$ must appear in φ.

5.1.2. Relational Databases and Conjunctive Queries

We introduced the concept of database schema (also called relational signature) in Definition 5.1. Complementarily, we now define the notion of database instances:

Definition 5.8 (Database Instance) A *database instance* I over some database schema $\mathcal{R} := \{R_1, \ldots, R_n\}$ is a set of relational atoms with predicate symbols from $\mathcal{R}$ that contain only elements from $\Delta \cup \Delta_{null}$ in its positions, i.e. atoms of the form $R(a_1, ..., a_{ar(R)})$ where $R \in \mathcal{R}$ and $a_1, ..., a_{ar(R)} \in \Delta \cup \Delta_{null}$.

The *domain* of an instance I, $dom(I)$, is the set of elements from $\Delta \cup \Delta_{null}$ that appear in I, i.e. $dom(I) = \{a_1, ..., a_{ar(R)} \mid R(a_1, ..., a_{ar(R)}) \in I\}$. □

Each database instance I can be understood as an $\mathcal{R}$-structure $(\Delta \cup \Delta_{null}, I)$, where I contains the set of all facts in the structure. Given a sentence φ and an instance I, we therefore shall write $I \models \varphi$ as a shortcut for $(\Delta \cup \Delta_{null}, I) \models \varphi$.

Conjunctive Queries. The definition of conjunctive queries is folklore:

Definition 5.9 (Conjunctive Query) A *conjunctive query* (CQ) is an expression of the form $q : ans(\overline{x}) \leftarrow \varphi(\overline{x}, \overline{y})$, where (a) $\varphi(\overline{x}, \overline{y})$ is a conjunction of $\mathcal{R}$-atoms, (b) $\overline{x}, \overline{y}$ are tuples of variables and constants, and (c) every variable that occurs in $\overline{x}$ also occurs in $\varphi(\overline{x}, \overline{y})$. We denote by $body(q)$ the set of $\mathcal{R}$-atoms in $\varphi(\overline{x}, \overline{y})$. The evaluation result of q on instance I is defined as $q(I) := \{\ \overline{a} \mid I \models \exists \overline{y} \varphi(\overline{a}, \overline{y})\ \}$. □

5.1.3. Relational Constraints

We use the terms *(integrity) constraints* and *(data) dependencies* interchangeably. Following [Nic78; Deu08], we encode constraints as first-order sentences. We next introduce different classes of constraints proposed in previous work [Fag77; Fag82; BV84; DT01; DT05], highlight their expressiveness, relevance in practice, and their interrelationships. We start with equality- and tuple-generating dependencies:

Definition 5.10 (Equality-generating Dependency) Let $\overline{x}, \overline{y}$ be tuples of variables. An *equality-generating dependency* (EGD) is a first-order sentence

$$\varphi := \forall \overline{x} (\phi(\overline{x}) \rightarrow x_i = x_j)$$

such that (a) $\phi(\overline{x})$ is a non-empty conjunction of $\mathcal{R}$-atoms, (b) x_i, x_j are variables from $\overline{x}$, and (c) x_i, x_j occur in $\phi(\overline{x})$. We denote by $body(\varphi)$ the set of $\mathcal{R}$-atoms in $\phi(\overline{x})$ and by $head(\varphi)$ the set $\{x_i = x_j\}$. □

Definition 5.11 (Tuple-generating Dependency) Let $\overline{x}, \overline{y}$ be tuples of variables. A *tuple-generating dependency* (TGD) is a first-order sentence

$$\varphi := \forall \overline{x} (\phi(\overline{x}) \rightarrow \exists \overline{y} \psi(\overline{x}, \overline{y}))$$

such that (a) $\phi(\overline{x})$ and $\psi(\overline{x}, \overline{y})$ are conjunctions of $\mathcal{R}$-atoms, (b) $\psi(\overline{x}, \overline{y})$ is not empty, (c) $\phi(\overline{x})$ is possibly empty, and (d) all variables from $\overline{x}$ that occur in $\psi(\overline{x}, \overline{y})$ also occur in $\phi(\overline{x})$. We denote by $body(\varphi)$ the set of $\mathcal{R}$-atoms in $\phi(\overline{x})$ and by $head(\varphi)$ the set of $\mathcal{R}$-atoms in $\psi(\overline{x}, \overline{y})$. □

TGDs can be seen as a generalization of join and inclusion dependencies, while EGDs generalize functional dependencies (the interested reader will find more background on join, inclusion, and functional dependencies in [AHV]). We illustrate equality-generating and tuple-generating dependencies by a small example:

Example 5.2 Let $\mathcal{R} := \{R, S\}$ with $ar(R) := 2$ and $ar(S) := 2$ be a relational signature. Further consider the EGD φ_1 and the TGD φ_2 defined as

$$\varphi_1 := \forall x, y_1, y_2(R(x, y_1) \wedge R(x, y_2) \rightarrow y_1 = y_2),$$
$$\varphi_2 := \forall x, y(R(x, y) \rightarrow \exists z S(y, z)).$$

The dependency φ_1 can be understood as a primary key constraint. Informally speaking, it states that for each database instance $I \models \varphi_1$ the value in the first position R^1 of predicate R uniquely identifies tuples in R^I. It is well-known that (in the relational context) primary keys are special case of functional dependencies.

The TGD φ_2 represents a foreign key constraint (which is a special case of inclusion dependencies). It enforces that each value that appears in position R^2 of some tuple $\overline{t_R} \in R^I$ is also encountered in the first position of some tuple $\overline{t_S} \in S^I$. □

Having introduced EGDs and TGDs, we go one step further and define the class of so-called *embedded dependencies*, originally proposed in [Fag77; Fag82]:

Definition 5.12 (Embedded Dependency) Let $\overline{x}$, $\overline{y}$ be tuples of variables. An *embedded dependency* (ED) is a first-order sentence

$$\varphi := \forall \overline{x}(\phi(\overline{x}) \rightarrow \exists \overline{y} \psi(\overline{x}, \overline{y}))$$

such that (a) $\phi(\overline{x})$ is a possibly empty conjunction of $\mathcal{R}$-atoms, (b) $\psi(\overline{x}, \overline{y})$ is a non-empty conjunction of $\mathcal{R}$-atoms and equality $\mathcal{R}$-formulas, and (c) each variable from $\overline{x}$ that occurs in $\psi(\overline{x}, \overline{y})$ also occurs in $\phi(\overline{x})$. □

It is easy to see that both EGDs and TGDs are special cases of EDs: EGDs are obtained from EDs when restricting to a single equality $\mathcal{R}$-formula in the head, while TGDs are obtained when restricting to a conjunction of $\mathcal{R}$-atoms in the head of the embedded dependency. Conversely, it is well-known that every set of embedded dependencies can be expressed as a set of TGDs and EGDs [Fag82].

As argued in previous work such as [Deu08; DPT06], embedded dependencies comprise most of the natural constraints used in practical database scenarios, such as keys, join dependencies, inclusion dependencies, and functional dependencies. As a contribution, we will later discuss natural counterparts of relational constraints for RDF in Section 5.2, showing that embedded dependencies also capture many natural constraints that appear in the context of RDF data.

As an extension of EDs, we finally define *disjunctive embedded dependencies* [DT01] and *disjunctive embedded dependencies with non-equality* [DT05]:

Definition 5.13 (Disjunctive Embedded Dependency) Let $\overline{x}$, $\overline{y}$ be tuples of variables. A *disjunctive embedded dependency* (DED) is a first-order sentence

$$\varphi := \forall \overline{x}(\phi(\overline{x}) \rightarrow \bigvee_{i=1}^{l}(\exists \overline{y_i}\psi_i(\overline{x}, \overline{y_i})))$$

such that (a) $\phi(\overline{x})$ is a possibly empty conjunction of $\mathcal{R}$-atoms, (b) $\psi_i(\overline{x}, \overline{y_i})$ is a non-empty conjunction of $\mathcal{R}$-atoms and equality $\mathcal{R}$-formulas, and (c) each variable from $\overline{x}$ that occurs in $\psi_i(\overline{x}, \overline{y_i})$ (for some $1 \leq i \leq l$) also occurs in $\phi(\overline{x})$.

The class of *disjunctive embedded dependencies with non-equality* (DED$^{\neq}$) is obtained when we replace condition (b) above by the condition (b$^{\neq}$) $\psi_i(\overline{x}, \overline{y_i})$ is a non-empty conjunction of $\mathcal{R}$-atoms, equality $\mathcal{R}$-formulas, and non-equality $\mathcal{R}$-formulas.□

5.1.4. The Chase Algorithm

The chase procedure [MMS79; BV84] is a fundamental algorithm that has been successfully applied in different areas of database research (we will list possible application scenarios when discussing related work in Section 5.5). Our focus here is the use of the algorithm for SPARQL query optimization under data dependencies. To achieve this goal, we exploit known results and properties of the chase algorithm that have been established in the context of conjunctive query optimization under data dependencies [ASU79; JK82; DPT06]. We will restrict ourselves to *chasing* sets of TGDs and EGDs here (which are known to be as expressive as sets of EDs, as discussed previously in Section 5.1.3). Note, however, that the chase also has been successfully applied in the context of DEDs (see e.g. [DT05]), so the techniques presented in this chapter can easily be extended to this larger class of constraints.

The core idea of the chase is simple: given a set of dependencies and a database instance as input, the algorithm successively fixes constraint violations in the instance. When using the chase for query optimization under a set of dependencies, this means that the query – interpreted as database instance – is provided as input, together with the constraints that are known to hold on every database instance.

One major problem with the chase is that – given an arbitrary set of constraints – it does not terminate in the general case; even worse, the termination problem is undecidable in general, even for a fixed instance [DNR08]. We start with a formal definition of the chase algorithm and come back to the issue of chase termination later. As a prerequisite, we give a formal definition of homomorphisms:

Definition 5.14 (Homomorphism) A homomorphism from database instance I_1 to database instance I_2 is a mapping $\nu : \Delta \cup \Delta_{null} \rightarrow \Delta \cup \Delta_{null}$ such that the following three conditions hold: (1) if $c \in \Delta$, then $\nu(c) = c$; (2) if $n \in \Delta_{null}$, then $\nu(n) \in \Delta \cup \Delta_{null}$; (3) if $R(t_1, ..., t_{ar(R)}) \in I_1$, then $R(\nu(t_1), ..., \nu(t_{ar(R)})) \in I_2$. □

Using the notion of homomorphism, we are now ready to define the chase algorithm. We fix a set of TGDs and EGDs Σ and a database instance I.

A *tuple-generating dependency* $\forall\overline{x}\varphi \in \Sigma$ *is applicable to* I if there is a homomorphism ν from $body(\forall\overline{x}\varphi)$ to I and ν cannot be extended to a homomorphism $\nu' \supseteq \nu$ from $head(\forall\overline{x}\varphi)$ to I. In such a case, the *chase step* $I \overset{\forall\overline{x}\varphi,\nu(\overline{x})}{\longrightarrow} J$ is defined as follows. We define a homomorphism ν_+ with the following two properties: (a) ν_+ agrees with ν on all universally quantified variables in φ and (b) for every existentially quantified variable y in $\forall\overline{x}\varphi$ we choose a "fresh" labeled null $n_y \in \Delta_{null}$ and define $\nu_+(y) := n_y$. Finally, we set J to $I \cup \nu_+(head(\forall\overline{x}\varphi))$.

We say that an *equality-generating dependency* $\forall\overline{x}\varphi \in \Sigma$ *is applicable to* I if there is a homomorphism ν from $body(\forall\overline{x}\varphi)$ to I and $\nu(x_i) \neq \nu(x_j)$. In such a case the *chase step* $I \overset{\forall\overline{x}\varphi,\nu(\overline{x})}{\longrightarrow} J$ is defined as follows. We set J to be

- I except that all occurrences of $\nu(x_j)$ are substituted by $\nu(x_i)$, if $\nu(x_j) \in \Delta_{null}$,
- I except that all occurrences of $\nu(x_i)$ are substituted by $\nu(x_j)$, if $\nu(x_i) \in \Delta_{null}$,
- undefined, if both $\nu(x_j)$ and $\nu(x_i)$ are constants. In this case we say that the chase fails.

A *chase sequence* is an exhaustive application of applicable constraints

$$I_0 \overset{\varphi_0,\overline{a}_0}{\longrightarrow} I_1 \overset{\varphi_1,\overline{a}_1}{\longrightarrow} I_2 \overset{\varphi_2,\overline{a}_2}{\longrightarrow} \ldots,$$

where we impose no strict order on what constraint must be applied in case several constraints are applicable. If the chase sequence is finite, say I_r being its final element, the chase terminates and its result I_0^Σ is defined as I_r. Although different orders of application of applicable constraints may lead to different chase results, it is common knowledge that two different chase orders always lead to homomorphically equivalent results, if these exist (see e.g. [FKMP05]). Therefore, we write I^Σ for the result of the chase on an instance I under constraint set Σ. It has been shown in [MMS79; JK82; BV84] that $I^\Sigma \models \Sigma$. If a chase step cannot be performed (e.g., because a homomorphism would have to equate two constants, see bullet three above) or in case of an infinite chase sequence, the chase result is undefined.

Example 5.3 Consider the database schema $\mathcal{R} := \{E, S\}$ with $ar(E) := 2$ and $ar(S) := 1$, where E stores graph edges and S stores a subset of graph nodes. Put

$$\alpha_1 := \forall x, y(E(x, y) \rightarrow E(y, x)),$$
$$\alpha_2 := \forall x(S(x) \rightarrow \exists y E(x, y)),$$
$$\alpha_3 := \forall x, y(S(x) \wedge E(x, y) \wedge E(y, x) \rightarrow x = y),$$

and $\Sigma := \{\alpha_1, \alpha_2, \alpha_3\}$. One chase sequence for $I := \{S(a), E(c, d)\}$ with Σ is

$$
\begin{aligned}
&I := \{S(a), E(c,d)\} \\
&\xrightarrow{\alpha_1,(c,d)} \{S(a), E(c,d), E(d,c)\} \\
&\xrightarrow{\alpha_2,(a)} \{S(a), E(c,d), E(d,c), E(a,n_1)\} \\
&\xrightarrow{\alpha_1,(a,n_1)} \{S(a), E(c,d), E(d,c), E(a,n_1), E(n_1,a)\} \\
&\xrightarrow{\alpha_3,(a,n_1)} \{S(a), E(c,d), E(d,c), E(a,a)\} =: I^{\Sigma},
\end{aligned}
$$

where $n_1 \in \Delta_{null}$. Observe that I^{Σ} satisfies all constraints from Σ. □

Chase Termination. Non-termination of the chase can be caused by fresh labeled null values that are repeatedly created when fixing constraint violations:

Example 5.4 Consider the database schema from Example 5.3, the tuple-generating dependency $\alpha_4 := \forall x, y(E(x,y) \rightarrow \exists z E(y,z))$, and the instance $I := \{E(a,b)\}$. We chase instance I with the constraint set $\Sigma' := \{\alpha_4\}$:

$$
\begin{aligned}
&I := \{E(a,b)\} \\
&\xrightarrow{\alpha_4,(a,b)} \{E(a,b), E(b,n_1)\} \\
&\xrightarrow{\alpha_4,(b,n_1)} \{E(a,b), E(b,n_1), E(n_1,n_2)\} \\
&\xrightarrow{\alpha_4,(n_1,n_2)} \{E(a,b), E(b,n_1), E(n_1,n_2), E(n_2,n_3)\} \\
&\xrightarrow{\alpha_4,(n_2,n_3)} \dots,
\end{aligned}
$$

where $n_1, n_2, \dots$ are fresh labeled nulls. The chase does not terminate. □

Addressing the issue of non-terminating chase sequences, sufficient conditions for the constraint set have been proposed that guarantee termination on every database instance [FKMP05; DNR08; SML08; MSL09d]. Whenever such conditions apply, one can safely chase without risking non-termination. One such condition is *weak acyclicity* [FKMP05], which asserts that there are no cyclically connected positions in the constraint set that may introduce fresh labeled null values, by a global study of relations between the constraints. Although more general termination conditions exist [DNR08; MSL09d; MSL09a] (we will survey them in Section 5.5), we will restrict our discussion to weak acyclicity, which is sufficient to illustrate the idea behind such conditions and is strong enough to guarantee termination for all examples that we will discuss later. In practical scenarios, however, it makes sense to fall back on more general termination conditions, to extend the applicability of the chase algorithm.

Weak acyclicity builds on the so-called *dependency graph* introduced in [FKMP05]:

Definition 5.15 (Dependency Graph [FKMP05]) Let Σ be a set of TGDs and EGDs. The *dependency graph* $\mathrm{dep}(\Sigma) := (V, E)$ of Σ is the directed graph defined as follows. There are two kinds of edges in E. Add them as follows: for every TGD $\forall \overline{x}(\phi(\overline{x}) \rightarrow \exists \overline{y}\psi(\overline{x}, \overline{y})) \in \Sigma$ and for every x in $\overline{x}$ that occurs in $\psi(\overline{x}, \overline{y})$ and every occurrence of x in $\phi(\overline{x})$ in position π_1

- for every occurrence of x in $\psi(\overline{x}, \overline{y})$ in position π_2, add an edge $\pi_1 \rightarrow \pi_2$ (if it does not already exist), and
- for every existentially quantified variable y and for every occurrence of y in a position π_2, add a so-called *special edge* $\pi_1 \stackrel{*}{\rightarrow} \pi_2$ (if it does not already exist).□

On top of the dependency graph we can easily define the notion of weak acyclicity:

Definition 5.16 (Weak Acyclicity [FKMP05]) Let Σ be a set of TGDs and EGDs. Σ is *weakly acyclic* iff dep(Σ) has no cycles going through a special edge.□

It is known that, for weakly acyclic constraint sets, the chase terminates on every database instance in polynomial-time data complexity [FKMP05].

Example 5.5 The dependency graphs for the constraint set Σ from Example 5.3 and for the constraint set Σ' from Example 5.4 are

$$dep(\Sigma) := (\{E^1, E^2, S^1\}, \{E^1 \rightarrow E^2, E^2 \rightarrow E^1, S^1 \rightarrow E^1, S^1 \stackrel{*}{\rightarrow} E^2\}),$$
$$dep(\Sigma') := (\{E^1, E^2, S^1\}, \{E^2 \rightarrow E^1, E^2 \stackrel{*}{\rightarrow} E^2\}).$$

We observe that $dep(\Sigma)$ has no cycle going through a special edge, so Σ is weakly acyclic and the chase with Σ always terminates. In contrast, $dep(\Sigma')$ contains the cycle $E^2 \stackrel{*}{\rightarrow} E^2$, so no chase termination guarantees for Σ' can be made (in fact, Example 5.4 shows a non-terminating chase sequence for Σ'). □

The Chase for Semantic Query Optimization. In the context of SQO, the chase takes a CQ q and a set of TGDs and EGDs Σ as input. It interprets the body of the query, $body(q)$, as database instance and successively fixes constraint violations in $body(q)$. We denote the output obtained when chasing q with Σ as q^Σ. It is known that $body(q^\Sigma) \models \Sigma$ and that q^Σ is equivalent to q on every instance $D \models \Sigma$. Note that q^Σ is undefined whenever the chase fails or does not terminate.

The semantic optimization approach for SPARQL that we will present in Section 5.4 relies on the Chase & Backchase (C&B) algorithm for semantic optimization of CQs proposed in [DPT06], which builds upon the standard chase procedure. To give a short description, the C&B algorithm does the following: given a CQ q and a set of constraints Σ as input, it lists all Σ-equivalent minimal (with respect to the number of atoms in the body) rewritings of q, up to isomorphism. We do not describe the C&B algorithm in detail here, but use it as a black-box with the above-mentioned properties. It is important to note that also the C&B algorithm does not necessarily terminate, because it uses the standard chase as a subprocedure. However, all sufficient termination conditions for the chase also apply to the C&B algorithm. Given a set Σ of EGDs and TGDs and a conjunctive query q, we denote the output of the C&B algorithm, namely the set of all minimal conjunctive queries that are Σ-equivalent to q, by $cb_\Sigma(q)$, if it is defined (the output is defined whenever all chase sequences that are triggered by C&B are defined).

5.2. Constraints for RDF

In the previous section, we formalized dependencies for relational databases as first-order sentences. RDF constraints can be seen as a special case of such sentences:

Definition 5.17 (RDF Constraint) An *RDF constraint* is a first-order sentence over signature $\mathcal{R} := \{T\}$ with $ar(T) := 3$. We define the satisfaction of an RDF constraint φ as follows. Let D be an RDF document. We define the $\mathcal{R}$-structure $I_D := (dom(D), \{T(s,p,o) \mid (s,p,o) \in D\})$. We say that D *satisfies* φ, written $D \models \varphi$, iff $I_D \models \varphi$. Similarly, for a set of RDF constraints Σ, $D \models \Sigma$ iff $I_D \models \Sigma$. □

Example 5.6 Consider the RDF constraints

$$\begin{aligned}\beta_1 &:= \forall x_1, x_2, m(\, T(x_1, \textit{rdf:type}, \textit{Student}) \wedge T(x_2, \textit{rdf:type}, \textit{Student}) \wedge \\ &\qquad T(x_1, \textit{matric}, m) \wedge T(x_2, \textit{matric}, m) \rightarrow x_1 = x_2),\\ \beta_2 &:= \forall x (T(x, \textit{rdf:type}, \textit{Student}) \rightarrow \exists m\; T(x, \textit{matric}, m)).\end{aligned}$$

The EGD β_1 can be understood as a key stating that the matriculation number *matric* identifies objects of type student; TGD β_2 states that each student has at least one matriculation number associated. Now consider the RDF document

$$\begin{aligned}D \;:= \{&(P_1, \textit{rdf:type}, \textit{Student}), (P_1, \textit{matric}, \textit{“111111”}), (P_2, \textit{rdf:type}, \textit{Student}),\\ &(P_3, \textit{rdf:type}, \textit{Student}), (P_3, \textit{matric}, \textit{“222222”})\}.\end{aligned}$$

It is easily verified that $D \models \varphi_1$, $D \not\models \varphi_2$, and therefore $D \not\models \{\varphi_1, \varphi_2\}$. □

In the remainder of Section 5.2, we propose a collection of constraint templates for RDF and classify them as TGDs, EGDs, and DED$^{\neq}$s. The subsequent listing is not complete in any sense, but pursues the goals (a) to show that constraints like keys and foreign keys for structured and semi-structured data models or cardinality constraints for XML (as found e.g. in XML Schema [xml]), naturally carry over to RDF and fall into established constraint classes and (b) to present a proper formalization of different constraint types, which can be used in subsequent discussions.

5.2.1. Equality-generating Dependencies

In Figure 5.1 we summarize EGD templates for RDF, where C is to be instantiated by some RDF class and $p, p_1, \ldots, p_n, q$ are to be replaced by RDF properties. When instantiated, these templates express different types of functional dependencies over RDF data. First, constraint $Key(C, [p_1, \ldots, p_n])$ states that properties $p_1, \ldots, p_n$ form an n-ary key for objects of type C in the sense that they identify objects of this type. Note that the constraint “applies” only in cases where $p_1, \ldots, p_n$ are

Name	Formalization in First-order Logic (EGD)
$Key(C, [p_1, \ldots, p_n])$	$\forall x_1, x_2, o_1, \ldots, o_n($ $T(x_1, \textit{rdf:type}, C) \wedge T(x_2, \textit{rdf:type}, C) \wedge$ $T(x_1, p_1, o_1) \wedge \cdots \wedge T(x_1, p_n, o_n) \wedge$ $T(x_2, p_1, o_1) \wedge \cdots \wedge T(x_2, p_n, o_n)$ $\rightarrow x_1 = x_2)$
$FD([p_1, \ldots, p_n], q)$	$\forall x_1, x_2, o_1, \ldots, o_n, o_1', o_2'($ $T(x_1, p_1, o_1) \wedge \cdots \wedge T(x_1, p_n, o_n) \wedge T(x_1, q, o_1') \wedge$ $T(x_2, p_1, o_1) \wedge \cdots \wedge T(x_2, p_n, o_n) \wedge T(x_2, q, o_2')$ $\rightarrow o_1' = o_2')$
$FD^\bullet(C, [p_1, \ldots, p_n], q)$	$\forall x_1, x_2, o_1, \ldots, o_n, o_1', o_2'($ $T(x_1, \textit{rdf:type}, C) \wedge T(x_2, \textit{rdf:type}, C) \wedge$ $T(x_1, p_1, o_1) \wedge \cdots \wedge T(x_1, p_n, o_n) \wedge T(x_1, q, o_1') \wedge$ $T(x_2, p_1, o_1) \wedge \cdots \wedge T(x_2, p_n, o_n) \wedge T(x_2, q, o_2')$ $\rightarrow o_1' = o_2')$
$Func(p)$	$\forall x, o_1, o_2 (T(x, p, o_1) \wedge T(x, p, o_2) \rightarrow o_1 = o_2)$
$Func^\bullet(C, p)$	$\forall x, o_1, o_2 (T(x, \textit{rdf:type}, C) \wedge T(x, p, o_1) \wedge T(x, p, o_2) \rightarrow o_1 = o_2)$

Figure 5.1.: Templates of equality-generating dependencies for RDF, where C stands for an RDF class and $p, p_1, \ldots, p_n, q$ denote RDF properties.

all bound; we will present a variant of the constraint in Section 5.2.3, called Key_*, which additionally enforces that $p_1, \ldots, p_n$ are bound (but is not an EGD anymore).

The templates $FD([p_1, \ldots, p_n], q)$ and $FD^\bullet(C, [p_1, \ldots, p_n], q)$ express general functional dependencies, stating that a set of properties $p_1, \ldots, p_n$ with their corresponding values identify the value of property q whenever all properties $p_1, \ldots, p_n, q$ are present. While FD can be understood as a universal functional dependency that holds for objects of any type (and even untyped objects), template $FD^\bullet$ allows to encode functional dependencies for objects of some fixed type C only.

Finally, the two constraint templates $Func(p)$ and $Func^\bullet(C, p)$ express that RDF property p is functional in the sense that it occurs either zero or one times for each object (cf. $Func(p)$) or objects of some fixed type C (cf. $Func^\bullet(C, p)$).

Example 5.7 β_1 from Example 5.6 can be encoded as $Key(\textit{Student}, [\textit{matric}])$. □

5.2.2. Tuple-generating Dependencies

Figure 5.2 presents TGD templates for RDF. $FKey(C, [p_1, \ldots, p_n], D, [q_1, \ldots, q_n])$ can be understood as an n-ary foreign key constraint for RDF. It states that, for each object of type C with properties $p_1, \ldots, p_n$, there is an object of type D with properties $q_1, \ldots, q_n$ such that each q_i $(1 \leq i \leq n)$ maps to the same value as p_i.

Next, $SubC(C, D)$ and $SubP(p, q)$ express the subclass and subproperty relationships between classes and properties, respectively. These constraints are motivated

Name	Formalization in First-order Logic (TGD)
$FKey(C, [p_1, \ldots, p_n], D, [q_1, \ldots, q_n])$	$\forall x, o_1, \ldots, o_n($ $T(x, \textit{rdf:type}, C) \wedge T(x, p_1, o_1) \wedge \cdots \wedge T(x, p_n, o_n)$ $\rightarrow \exists y\ T(y, \textit{rdf:type}, D) \wedge T(y, q_1, o_1) \wedge \cdots \wedge T(y, q_n, o_n))$
$SubC(C, D)$	$\forall x(T(x, \textit{rdf:type}, C) \rightarrow T(x, \textit{rdf:type}, D))$
$SubP(p, q)$	$\forall x, o(T(x, p, o) \rightarrow T(x, q, o))$
$PDom(p, C)$	$\forall x, y(T(x, p, y) \rightarrow T(x, \textit{rdf:type}, C))$
$PRan(p, C)$	$\forall x, y(T(x, p, y) \rightarrow T(y, \textit{rdf:type}, C))$
$PChainP([p_1, \ldots, p_n], q)$	$\forall x, o_1, \ldots, o_n($ $T(x, p_1, o_1) \wedge T(o_1, p_2, o_2) \wedge \cdots \wedge T(o_{n-1}, p_n, o_n)$ $\rightarrow T(x, q, o_n))$
$PChainP^{\bullet}(C, [p_1, \ldots, p_n], q)$	$\forall x, o_1, \ldots, o_n(T(x, \textit{rdf:type}, C) \wedge$ $T(x, p_1, o_1) \wedge T(o_1, p_2, o_2) \wedge \cdots \wedge T(o_{n-1}, p_n, o_n)$ $\rightarrow T(x, q, o_n))$
$PChainC([p_1, \ldots, p_n], D)$	$\forall x, o_1, \ldots, o_n($ $T(x, p_1, o_1) \wedge T(o_1, p_2, o_2) \wedge \cdots \wedge T(o_{n-1}, p_n, o_n)$ $\rightarrow T(o_n, \textit{rdf:type}, D))$
$PChainC^{\bullet}(C, [p_1, \ldots, p_n], D)$	$\forall x, o_1, \ldots, o_n(T(x, \textit{rdf:type}, C) \wedge$ $T(x, p_1, o_1) \wedge T(o_1, p_2, o_2) \wedge \cdots \wedge T(o_{n-1}, p_n, o_n)$ $\rightarrow T(o_n, \textit{rdf:type}, D))$

Figure 5.2.: Templates of tuple-generating dependencies for RDF, where C, D stand for RDF classes and $p, p_1, \ldots, p_n, q, q_1, \ldots, q_m$ denote RDF properties.

by the predefined RDFS properties `rdfs:subClassOf` and `rdfs:subPropertyOf` (cf. the discussion in Section 2.2.2). Note, however, that there is a very basic conceptual difference: the RDFS vocabulary can be seen as a set of axioms that imply subclass and subproperty relationships that are not explicit in the RDF database according to the semantics of RDFS; in contrast, we understand $SubC(C, D)$, $SubP(p, q)$ as hard database constraints, which assert that the specified subclass and subproperty relationships are **explicitly** contained in the database. Also closely related to the RDFS vocabulary are the constraints $PDom(p, C)$ and $PRan(p, C)$, which fix the domain and range of property p to C and can be understood as the constraint versions of the RDFS properties `rdfs:domain` and `rdfs:range`, respectively. We will resume the discussion about connections between RDFS and constraints later in Section 5.4.1 when motivating constraint-based query optimization for SPARQL.

Next, Figure 5.2 contains the template constraints $PChainP([p_1, \ldots, p_n], q)$ and $PChainP^{\bullet}(C, [p_1, \ldots, p_n], q)$. These two constraints can be understood as referential integrity constraints along a chain of properties $p_1, \ldots, p_n$, stating that each value reachable when walking along $p_1, \ldots, p_n$ is linked directly through property q (again, there is a universal version and a version for objects of some class C). Complementarily, the path constraints $PChainC([p_1, \ldots, p_n], D)$ and $PChainC^{\bullet}(C, [p_1, \ldots, p_n], D)$ assert that objects reached when walking along properties $p_1, \ldots, p_n$ are of type D.

Name	Formalization (DEDs/DED$^{\neq}$s or sets thereof)
We define $\mathit{someEq}([o_1,\ldots,o_m]) := \bigvee_{1\le i\le m}\bigvee_{i<j\le m} o_i = o_j$, $\mathit{allDist}([o_1,\ldots,o_m]) := \bigwedge_{1\le i\le m}\bigwedge_{i<j\le m} o_i \neq o_j$.	
$\mathit{Min}(n,p)$	$\forall x,o(T(x,\mathit{rdf{:}type},o) \rightarrow$ $\exists o_1,\ldots,o_n T(x,p,o_1)\wedge\cdots\wedge T(x,p,o_n)\wedge \mathit{allDist}([o_1,\ldots,o_n]))$
$\mathit{Min}^{\bullet}(C,n,p)$	$\forall x(T(x,\mathit{rdf{:}type},C) \rightarrow$ $\exists o_1,\ldots,o_n T(x,p,o_1)\wedge\cdots\wedge T(x,p,o_n)\wedge \mathit{allDist}([o_1,\ldots,o_n]))$
$\mathit{Max}(n,p)$	$\forall x,o,o_1,\ldots,o_{n+1}(T(x,\mathit{rdf{:}type},o)\wedge$ $T(x,p,o_1)\wedge\cdots\wedge T(x,p,o_{n+1}) \rightarrow \mathit{someEq}([o_1,\ldots,o_{n+1}]))$
$\mathit{Max}^{\bullet}(C,n,p)$	$\forall x,o_1,\ldots,o_{n+1}(T(x,\mathit{rdf{:}type},C)\wedge$ $T(x,p,o_1)\wedge\cdots\wedge T(x,p,o_{n+1}) \rightarrow \mathit{someEq}([o_1,\ldots,o_{n+1}]))$
$\mathit{Exact}(n,p)$	$\{\mathit{Min}(n,p),\mathit{Max}(n,p)\}$
$\mathit{Exact}^{\bullet}(C,n,p)$	$\{\mathit{Min}^{\bullet}(C,n,p),\mathit{Max}^{\bullet}(C,n,p)\}$
$\mathit{Total}(p)$	$\mathit{Exact}(1,p)$
$\mathit{Total}^{\bullet}(C,p)$	$\mathit{Exact}^{\bullet}(C,1,p)$
$\mathit{PDom}^{\vee}(p,[C_1,\ldots,C_n])$	$\forall x,y(T(x,p,y)\rightarrow T(x,\mathit{rdf{:}type},C_1)\vee\cdots\vee T(x,\mathit{rdf{:}type},C_n))$
$\mathit{PRan}^{\vee}(p,[C_1,\ldots,C_n])$	$\forall x,y(T(x,p,y)\rightarrow T(y,\mathit{rdf{:}type},C_1)\vee\cdots\vee T(y,\mathit{rdf{:}type},C_n))$
$\mathit{CProp}^{\vee}(C,[p_1,\ldots,p_n])$	$\forall x,p,o(T(x,\mathit{rdf{:}type},C)\wedge T(x,p,o)\rightarrow p=p_1\vee\cdots\vee p=p_n)$
$\mathit{Key}_*(C,[p_1,\ldots,p_n])$	$\{\mathit{Key}(C,[p_1,\ldots,p_n])\}\cup \mathit{Total}^{\bullet}(C,p_1)\cup\cdots\cup \mathit{Total}^{\bullet}(C,p_n)$

Figure 5.3.: Templates of disjunctive embedded dependencies with non-equality, where $C, C_1, \ldots, C_n$ stand for RDF classes and $p, p_1, \ldots, p_n, q, q_1, \ldots, q_n$ denote RDF properties.

5.2.3. Disjunctive Embedded Dependencies

We conclude our discussion of constraints for RDF with a collection of constraint templates that can be expressed as (sets of) disjunctive embedded dependencies or disjunctive embedded dependencies with non-equality. The first constraint types that are listed in Figure 5.3 are the cardinality constraints $\mathit{Min}(n,p)$ and $\mathit{Max}(n,p)$, stating that for objects of any o type the RDF property p is present minimally or maximally n times, respectively. As usual, the corresponding constraints $\mathit{Min}^{\bullet}(C,n,p)$ and $\mathit{Max}^{\bullet}(C,n,p)$ impose the respective restrictions to objects of type C only. Note that we use the shortcuts $\mathit{someEq}([o_1,\ldots,o_m])$ (encoding that there must be some $i \neq j$ such that $o_i = o_j$) and $\mathit{allDist}([o_1,\ldots,o_m]$ (stating that for all $i \neq j$ it holds that $o_i \neq o_j$); the formal definition of these functions is given in the figure. As can be seen, the templates $\mathit{Total}(p)$, $\mathit{Total}^{\bullet}(C,p)$, $\mathit{Exact}(n,p)$, and $\mathit{Exact}^{\bullet}(C,n,p)$ are easily obtained when combining the different versions of min- and max-cardinality constraints; note that the latter templates are expressed as sets of constraints.

Example 5.8 β_2 from Example 5.6 can be encoded as $\mathit{Min}^{\bullet}(\mathit{Student},1,\mathit{matric})$. □

Next, as generalizations of the tuple-generating constraint templates $PDom(p, C)$ and $PRan(p, C)$, we introduce $PDom^{\vee}(p, [C_1, \ldots, C_n])$ and $PRan^{\vee}(p, [C_1, \ldots, C_n])$. They enforce that URIs or literals appearing in the subject and object position of property p are typed with at least one of the classes $C_1, \ldots, C_n$; observe that, in line with the philosophy of RDF, we allow for multiple domain assignments. Complementarily to the extended versions of the property domain and range restriction constraints, the template $CProp^{\vee}(C, [p_1, \ldots, p_n])$ allows to restrict the set of properties that are used in combination with objects of type C to $p_1, \ldots, p_n$.

Finally, the constraint $Key_*(C, [p_1, \ldots, p_n])$ is a variant of the key constraint $Key(C, [p_1, \ldots, p_n])$ presented in Figure 5.1. Like the original key constraint, it enforces that the values of properties $p_1, \ldots, p_n$ identify objects of type C, and additionally asserts that all these properties are present exactly once.

5.3. SPARQL as a Constraint Language

One interesting question that arises in the context of constraints for RDF data is to which degree SPARQL can be used to check if constraints hold over the input RDF database. Before starting, we give a formal notion of constraint checking with SPARQL. We argue that the natural way to verify if a constraint holds on some RDF document is using the SPARQL Ask query form introduced in Definition 2.6:

Definition 5.18 (Constraint Checking with SPARQL) Let Q be a SPARQL Ask query and φ be an RDF constraint. We say that Q *checks* φ iff for every RDF document D it holds that $[\![Q]\!]_D \Leftrightarrow D \not\models \varphi$. □

Informally speaking, to check a constraint we have to write a SPARQL Ask query that detects exactly the constraint violations in the database. If the constraint holds on some database, then there is no constraint violation in the database and consequently the Ask query evaluates to *false*; otherwise, it evaluates to *true*.

Example 5.9 Let us consider the equality-generating dependency β_1 from Example 5.6 and the tuple-generating dependency $\beta_3 := \exists x T(x, \mathit{rdf{:}type}, \mathit{Student})$. It is easy to see that the SPARQL query

$$Q_{\beta_1} := \textsc{Ask}(((?x_1, \mathit{rdf{:}type}, \mathit{Student}) \textsc{ And } (?x_2, \mathit{rdf{:}type}, \mathit{Student}) \textsc{ And } (?x_1, \mathit{matric}, ?m) \textsc{ And } (?x_2, \mathit{matric}, ?m)) \textsc{ Filter } (\neg(?x_1 = ?x_2))),$$

checks constraint β_1: obviously, the query returns *true* if and only if there are two distinct students in the database that have the same matriculation number.

We further argue that there is no SPARQL query that checks constraint β_3. To see why this is the case, consider the empty RDF document $D := \emptyset$. We observe that $D \not\models \beta_3$, so one restriction for the SPARQL check query is that it must return *true* on the empty document. It can easily be shown (by induction on the structure of SPARQL expressions) that such a SPARQL query does not exist. □

The example shows that we cannot check TGDs in the general case. It follows immediately that we generally cannot check the constraint classes ED, DED, and $\text{DED}^{\neq}$, which generalize the class of TGDs. We next present a simple extension of SPARQL by *constant mappings*, called SPARQL^C, which can be understood as the counterpart of constants in relational algebra; we shall see later that this extension gives SPARQL the expressive power to check arbitrary first-order logic sentences.

Given the result from [AG08a] that SPARQL has the same expressiveness as relational algebra and the close connection between relational algebra and first-order logic, one might wonder why such an extension is necessary. The crucial observation is that, in the proof of the result that relational algebra has the same expressiveness as SPARQL in [AG08a], the authors use a version of SPARQL that extends our fragment from Definition 2.4 by (i) so-called empty graph patterns $\{\}$ (with semantics $[\![\{\}]\!]_D := \{\emptyset\}$) and by (ii) a MINUS operator at syntax level (with semantics $[\![Q_1 \text{ MINUS } Q_2]\!]_D := [\![Q_1]\!]_D \setminus [\![Q_2]\!]_D$). We emphasize that the encoding of RDF constraints (cf. Theorem 5.1 below) is also possible when using these two extensions. With SPARQL^C we propose another extension of SPARQL that compensates for both missing empty graph patterns and a missing syntactic MINUS operator:[2]

Definition 5.19 (SPARQL^C) Let $?x \in V$ and $c \in LU$ be a literal or a URI. A *constant triple pattern*, or *c-pattern* for short, is an expression of the form $t(?x \mapsto c)$. We extend the set and bag semantics from Definitions 2.11 and 2.15 as follows:

$$[\![t(?x \mapsto c)]\!]_D := \{\{?x \mapsto c\}\}$$
$$[\![t(?x \mapsto c)]\!]_D^+ := (\{\{?x \mapsto c\}\}, m), \text{ where for } \mu \in \mathcal{M} \text{ we define}$$
$$m(\mu) := 1 \text{ if } \mu = \{?x \mapsto c\}, \text{ and } m(\mu) := 0 \text{ otherwise.}$$

A SPARQL^C expression or query is defined as a SPARQL expression or query in which *c*-patterns can be used in place of simple triple patterns. □

Example 5.10 Consider the SPARQL^C query

$$Q_C := \text{SELECT}_{?p,?status}(t(?status \mapsto \textit{“Student”}) \text{ AND } (?p, \textit{rdf:type}, \textit{Person}) \text{ AND } (?p, \textit{matric}, ?m))$$

which extracts all persons with matriculation number (property *matric*) from the database and assigns status *“Student”* to these persons. Let us assume that the database $D := \{(\textit{Jil}, \textit{rdf:type}, \textit{Person}), (\textit{Jil}, \textit{matric}, \textit{“1234”}), (\textit{John}, \textit{rdf:type}, \textit{Person})\}$ is given. It is easily verified that $[\![Q_C]\!]_D = \{\{?p \mapsto \textit{Jil}, ?status \mapsto \textit{“Student”}\}\}$. □

Resuming our discussion of constraint checking with SPARQL, first observe that we can check the constraint β_3 from Example 5.9 in SPARQL^C:

[2] In fact, similar extensions have already been used in practice. For instance, the ARQ SPARQL processor [arq] supports such constant patterns by means of a LET clause, which allows to bind variables to some fixed value, see `http://jena.sourceforge.net/ARQ/assignment.html`.

Example 5.11 The SPARQL^C query

$$Q_{\beta_3} := \textsc{Ask}((t(?x \mapsto c)\ \textsc{Opt}\ (?y, \mathit{rdf{:}type}, \mathit{Student}))\ \textsc{Filter}\ (\neg \mathit{bnd}(?y)))$$

checks constraint β_3 from Example 5.9: if database D contains no object of type *Student* then $D \not\models \beta_3$ and $[\![Q_{\beta_3}]\!]_D = \mathit{true}$, otherwise $D \models \beta_3$ and $[\![Q_{\beta_3}]\!]_D = \mathit{false}$. □

As illustrated in the example, the SPARQL^C language allows to encode (top-level) negation in SPARQL queries. We generalize this construction as follows (we use the concept of *possible variables*, i.e. function *pVars* introduced in Section 4.1).

Proposition 5.1 Let Q be a SPARQL^C expression and $?x, ?y \in V \setminus \mathit{pVars}([\![Q]\!]_D)$. Define $Q^\neg := (t(?x \mapsto c)\ \textsc{Opt}\ (t(?y \mapsto c)\ \textsc{And}\ Q))\ \textsc{Filter}\ (\neg \mathit{bnd}(?y))$. Then for every RDF document D it holds that $[\![\textsc{Ask}(Q)]\!]_D \Leftrightarrow \neg[\![\textsc{Ask}(Q^\neg)]\!]_D$. □

Proof of Proposition 5.1

⇒: (a) Assume that $[\![\textsc{Ask}(Q)]\!]_D = \mathit{true}$. Then $[\![Q]\!]_D$ is not empty. Now consider what happens when evaluating $[\![Q^\neg]\!]_D$. First recall that $?x, ?y \notin \mathit{pVars}([\![Q]\!]_D)$ by assumption, so it is easy to see that the inner And expression extends each mapping in $[\![Q]\!]_D$ by $?y \mapsto c$, and the Opt expression further extends each mapping by $?x \mapsto c$; consequently, the surrounding filter discards all mappings, so $[\![Q^\neg]\!]_D = \emptyset$ and $[\![\textsc{Ask}(Q^\neg)]\!]_D = \mathit{false}$. If (b) $[\![\textsc{Ask}(Q)]\!]_D = \mathit{false}$, then $[\![Q]\!]_D = \emptyset$. It is easily verified that, in this case, $[\![Q^\neg]\!]_D = \{\{?x \mapsto c\}\}$ and therefore $[\![\textsc{Ask}(Q^\neg)]\!]_D = \mathit{true}$.

⇐: (a) Assume that $[\![\textsc{Ask}(Q^\neg)]\!]_D = \mathit{true}$. Then $[\![Q^\neg]\!]_D$ is not empty. This implies that there is a mapping in $[\![t(?x \mapsto c)\ \textsc{Opt}\ (t(?y \mapsto c)\ \textsc{And}\ Q)]\!]_D$ such that $?y$ is not bound. It is easily verified that $?y$ is bound in every result mapping whenever $[\![Q]\!]_D$ is not empty, so we conclude that $[\![Q]\!]_D = \emptyset$ and therefore $[\![\textsc{Ask}(Q)]\!]_D = \mathit{false}$. (b) If $[\![\textsc{Ask}(Q^\neg)]\!]_D = \mathit{false}$, then $[\![Q^\neg]\!]_D = \emptyset$. Hence, it must hold that $?y$ is bound in every mapping obtained when evaluating $[\![t(?x \mapsto c)\ \textsc{Opt}\ (t(?y \mapsto c)\ \textsc{And}\ Q)]\!]_D$. Assume for the sake of contradiction that $[\![Q]\!]_D = \emptyset$. It is easily verified that, in this case, $[\![Q^\neg]\!]_D = \{\{?x \mapsto c\}\}$, which is a contradiction. We conclude that $[\![Q]\!]_D \neq \emptyset$ and therefore it holds that $[\![\textsc{Ask}(Q)]\!]_D = \mathit{true}$. □

We emphasize that top-level negation cannot be encoded in standard SPARQL, because we cannot write a SPARQL query that returns *true* on the empty document and therefore we cannot negate the result of a query in the general case. The main result in this subsection is that the extension from SPARQL to SPARQL^C is sufficient to check every first-order sentence (over a single, ternary signature). Of course, this comprises all dependency classes (i.e., EGDs, EDs, DEDs, and DED$^\neq$s) and consequently all the constraint templates proposed in Section 5.2.

Theorem 5.1 Let φ be an RDF constraint. Then there is a SPARQL^C query that checks φ. □

As a practical implication, the theorem shows that SPARQL (when marginally extended) is a good candidate for a constraint language. In particular, SPARQL^C can be used for checking if a constraint holds on some database instance and – phrased more generally – for expressing arbitrary RDF constraints. In practice, it would make sense to define an ASSERT query form for SPARQL^C, which – akin to CREATE ASSERTION statements in SQL – allows to specify user-defined constraints on top of RDF database management systems. Such fixed constraints could then be used internally to check data integrity and also for SQO. We finally point out that, although we defined constraint checking over set semantics in Definition 5.18, our results immediately carry over to bag semantics, because the evaluation of ASK queries is invariant w.r.t. the underlying semantics (see Lemma 4.14(1)). In the remainder of this subsection we present the proof for Theorem 5.1. It is constructive and makes the connection between first-order logic and SPARQL explicit.

Proof of Theorem 5.1

We show that for each RDF constraint, i.e. each first-order sentence φ over signature $\mathcal{R} := \{T\}$ with $ar(T) := 3$, there is a SPARQL^C query Q_φ such that $[\![\text{ASK}(Q_\varphi)]\!]_D = \mathit{true} \Leftrightarrow D \models \varphi$. This is sufficient, because the construction from Proposition 5.1 then gives us a query $Q_\varphi^\neg$ such that $[\![\text{ASK}(Q_\varphi^\neg)]\!]_D = \mathit{false} \Leftrightarrow D \models \varphi$.[3]

More precisely, we present a SPARQL^C encoding of a first-order sentence φ that is built using (1) equality $\mathcal{R}$-formulas $t_1 = t_2$, (2) $\mathcal{R}$-atoms of the form $T(t_1, t_2, t_3)$, (3) the negation operator $\neg$, (4) the conjunction operator $\wedge$, and (5) $\mathcal{R}$-formulas of the form $\neg\exists x\psi$. It is folklore that $\psi_1 \vee \psi_2$ can be written as $\neg(\neg\psi_1 \wedge \neg\psi_2)$ and each quantifier $\mathcal{R}$-formula can easily be brought into the form (5), i.e. $\forall x\psi$ is equivalent to $\neg\exists x\neg\psi$ and $\exists x\psi$ can be written as $\neg(\neg\exists x\psi)$, so cases (1)-(5) are sufficient. Note that we chose the variant $\neg\exists x\psi$ because its encoding is simpler than e.g. $\forall x\psi$ or $\exists x\psi$.

Having fixed the structural constraints for the input formula φ, we now show how to construct a SPARQL^C query Q_φ such that $[\![Q_\varphi]\!]_D = \emptyset$ if and only if $I_D \not\models \varphi$, where $I_D := (\mathit{dom}(D), \{T(s,p,o) \mid (s,p,o) \in D\})$. Before presenting this encoding, we introduce some additional notation and shortcut definitions.

Let $\mathit{var}(\varphi) := \{x_1, \ldots, x_n\}$ denote all variables appearing in formula φ. We define the set $S := \{?x_1, \ldots, ?x_n\}$ of corresponding SPARQL variables. Further, we introduce the function $v : \mathit{var}(\varphi) \mapsto S$ that translates each variable occurring in φ into its corresponding SPARQL variable, i.e. we define $v(x_i) :=?x_i$ for $1 \leq i \leq n$. Further assume that $S^\neg$ is an infinite set of variables disjoint from S, i.e. $S \cap S^\neg = \emptyset$. For each subexpression ψ of the input formula φ we define an infinite partition $S_\psi^\neg \subset S^\neg$ such that, for each pair of distinct subexpressions $\psi_1 \neq \psi_2$ of φ it holds that $S_{\psi_1}^\neg \cap S_{\psi_2}^\neg = \emptyset$. Based on these variable set partitions, we define for each subexpression ψ of φ its so-called *active domain expression* Q_ψ as follows. Let $\mathit{free}(\psi) := \{v_1, \ldots, v_k\} \subseteq \mathit{var}(\varphi)$ be the set of free variables in subexpression ψ. Then we define Q_ψ as

[3] This shows that we also could negate the definition of SPARQL check queries (Definition 5.18), i.e. define a check query as an ASK query that returns *true* iff the constraint is satisfied.

$$
\begin{aligned}
Q_\psi := {} & t(?a \mapsto c)\ \textsc{And} \\
& ((v(v_1), ?a_{11}, ?a_{12})\ \textsc{Union}\ (?a_{11}, v(v_1), ?a_{12})\ \textsc{Union}\ (?a_{11}, ?a_{12}, v(v_1)))\ \textsc{And} \\
& ((v(v_2), ?a_{21}, ?a_{22})\ \textsc{Union}\ (?a_{21}, v(v_2), ?a_{22})\ \textsc{Union}\ (?a_{21}, ?a_{22}, v(v_2)))\ \textsc{And} \\
& \ldots\ \textsc{And} \\
& ((v(v_k), ?a_{k1}, ?a_{k2})\ \textsc{Union}\ (?a_{k1}, v(v_k), ?a_{k2})\ \textsc{Union}\ (?a_{k1}, ?a_{k2}, v(v_k)))
\end{aligned}
$$

where $?a, ?a_{11}, ?a_{12}, \ldots, ?a_{k1}, ?a_{k2}$ are pairwise distinct variables from $S^\neg_\psi$. It is crucial to note that the active domain expressions for two distinct subexpressions share at most variables from S, because $?a$ and all $?a_{ij}$ are chosen from partitions that belong to the respective subexpressions. Furthermore, observe that $Q_\varphi := t(?a \mapsto c)$, because φ is a sentence and therefore $\mathit{free}(\varphi) = \emptyset$. To give the intuition, each Q_ψ represents all combinations of binding the free variables $v_1, \ldots, v_k$ in ψ (more precisely, the corresponding SPARQL variables $v(v_1), \ldots, v(v_k)$) to elements of the input document, where $?a$ and the $?a_{ij}$ are globally unique dummy variables that are not of further importance (but were required for the construction).

The remainder of the proof follows a naive evaluation of first-order formulas on finite structures. With the help of the active domain subexpressions Q_ψ we generate all possible bindings for the free variables in a subformula. Note that there is no need to project away the dummy variables $?a_{ij}$: we use fresh, distinct variables for every subformula ψ, so they never affect compatibility between two mappings (and hence do not influence the evaluation process); in the end, we are only interested in the boolean value, so these bindings do not harm the construction.

Coming to the encoding, we define $\mathit{enc}(t) := v(t)$ if t is a variable and $\mathit{enc}(t) := t$ if t is a constant (i.e. we simply interpret t as a URI). The idea of the encoding $\mathit{enc}(\varphi)$ for φ is to inductively simulate the formula's semantics by generating all possible bindings for the free variables of some (sub)formula using SPARQL. Thereby, we follow the structural constraints for φ fixed in cases (1)-(5) before:

(1) For $\psi := t_1 = t_2$ we define $\mathit{enc}(\psi) := Q_\psi\ \textsc{Filter}\ (\mathit{enc}(t_1) = \mathit{enc}(t_2))$.
(2) For $\psi := T(t_1, t_2, t_3)$ we define $\mathit{enc}(\psi) := Q_\psi\ \textsc{And}\ (\mathit{enc}(t_1), \mathit{enc}(t_2), \mathit{enc}(t_3))$.
(3) For $\psi := \neg\psi_1$ we define
$\mathit{enc}(\psi) := (Q_\psi\ \textsc{Opt}\ (t(?b \mapsto c)\ \textsc{And}\ \mathit{enc}(\psi_1)))\ \textsc{Filter}\ (\neg \mathit{bnd}(?b))$,
where $?b \in S^\neg$ neither appears in the encoding $\mathit{enc}(\psi_1)$ nor in Q_ψ.
(4) For $\psi := (\psi_1 \wedge \psi_2)$ we define $\mathit{enc}(\psi) := \mathit{enc}(\psi_1)\ \textsc{And}\ \mathit{enc}(\psi_2)$.
(5) For $\psi := \neg\exists x\psi_1$ we define
$\mathit{enc}(\psi) := (Q_\psi\ \textsc{Opt}\ (t(?b \mapsto c)\ \textsc{And}\ \mathit{enc}(\psi_1)))\ \textsc{Filter}\ (\neg \mathit{bnd}(?b))$,
where $?b \in S^\neg$ neither appears in the encoding $\mathit{enc}(\psi_1)$ nor in Q_ψ.

Rather than going into detail, we sketch the idea behind the encoding. It satisfies the following property: for each formula ψ with $\mathit{free}(\psi) := \{v_1, \ldots, v_k\}$ it holds that ($\Rightarrow$) foreach interpretation $\mathcal{I} := (I_D, \gamma)$ such that $\mathcal{I} \models \psi$ there exists a mapping $\mu \in [\![\mathit{enc}(\varphi)]\!]_D$ such that $\mu \supseteq \{?v_1 \mapsto \gamma(v_1), \ldots, ?v_k \mapsto \gamma(v_k)\}$ and ($\Leftarrow$) foreach mapping $\mu \in [\![\mathit{enc}(\varphi)]\!]_D$ it holds that every interpretation (I_D,γ) with $\gamma(v_i) := \mu(?v_i)$

for $1 \leq i \leq k$ satisfies φ. It is straightforward to verify that these two directions imply the original claim, i.e. it follows from "$\Leftarrow$" and "$\Rightarrow$" that $I_D \not\models \varphi \Leftrightarrow [\![enc(\varphi)]\!]_D = \emptyset$.

The two directions can be proven by induction on the structure of formulas. Concerning the two basic cases (1) and (2) it is easy to see that in their encoding the active domain expressions generate the universe of all bindings for the free variables, which is then restricted by application of the filter (for case (1) $\psi := t_1 = t_2$) or by joining the active domain expression with the respective triple pattern (for case (2) $\psi := T(t_1, t_2, t_3)$). We omit the detailed proof of these cases. In the induction step there are three cases that remain to be shown. The idea of the encoding for (3) $\psi := \neg\psi_1$ is to subtract from the universe of solutions exactly the solutions of ψ_1, encoded by $enc(\psi_1)$. To see how the encoding works, observe that

$$\begin{aligned}
[\![enc(\psi)]\!]_D &= [\![(Q_\psi \text{ Opt } (t(?b \mapsto c) \text{ And } enc(\psi_1)) \text{ Filter } (\neg bnd(?b))]\!]_D \\
&= \sigma_{\neg bnd(?n)}([\![Q_\psi]\!]_D \bowtie ([\![t(?b \mapsto c)]\!]_D \bowtie [\![enc(\psi_1)]\!]_D)) \\
&\overset{(*_1)}{=} [\![Q_\psi]\!]_D \setminus ([\![t(?b \mapsto c)]\!]_D \bowtie [\![enc(\psi_1)]\!]_D) \\
&\overset{(*_2)}{=} [\![Q_\psi]\!]_D \setminus [\![enc(\psi_1)]\!]_D
\end{aligned}$$

where step $(*_1)$ follows by application of equivalence *(FLBndI)* from Lemma 4.4 presented in Section 4.2.6 and step $(*_2)$ follows from the observation that the single mapping in $[\![t(?b \mapsto c)]\!]_D = \{\{?b \mapsto c\}\}$ is compatible with every mapping in $[\![\psi_1]\!]_D$ ($?b$ does not appear in $enc(\psi_1)$) and the extension of mappings from $[\![\psi_1]\!]_D$ by $?b \mapsto c$ does not modify the result. Hence, the encoding implements negation.

To discuss the remaining cases, observe that (4) a conjunction $\psi := \psi_1 \wedge \psi_2$ is straightforwardly mapped to a join operation between $enc(\psi_1)$ and $enc(\psi_2)$. Finally, the encoding for (5) $\psi := \neg\exists x\varphi$ is similar to the encoding for the negation; however, in this case it holds that $?x \notin free(\psi)$, so the active domain expression does not contain variable $?x$ anymore, which gives us an implicit projection. □

We conclude with a small example that illustrates the construction.

Example 5.12 Consider the RDF constraint $\exists x T(x, a, a)$, which can be rewritten as $\varphi := \neg(\neg\exists x T(x, a, a))$. First, we set up the active domain expressions for the subexpressions of φ:

$$\begin{aligned}
&Q_{T(x,a,a)} := t(?a \mapsto c) \text{ And } ((?x, ?a_1, ?a_2) \text{ Union } (?a_1, ?x, ?a_2) \text{ Union } (?a_1, ?a_2, ?x)), \\
&Q_{\neg\exists x T(x,a,a)} := t(?a' \mapsto c), \\
&Q_\varphi := t(?a'' \mapsto c).
\end{aligned}$$

Next, we encode the formula using these active domain expressions:

$$\begin{aligned}
&enc(T(x, a, a)) := Q_{T(x,a,a)} \text{ And } (?x, a, a), \\
&enc(\neg\exists x T(x, a, a)) := (Q_{\neg\exists x T(x,a,a)} \text{ Opt } (t(?b \mapsto c) \text{ And } enc(T(x, a, a))) \\
&\qquad\qquad \text{Filter } (\neg bnd(?b)), \\
&enc(\varphi) := (Q_\varphi \text{ Opt } (t(?b' \mapsto c) \text{ And } enc(\neg\exists x T(x, a, a)))) \text{ Filter } (\neg bnd(?b')).
\end{aligned}$$

Now consider the RDF database $D := \{(a, a, a), (b, b, b)\} \models \varphi$. We observe that

$$\begin{aligned}
[\![Q_{T(x,a,a)}]\!]_D &= \{\{?x \mapsto a, ?a \mapsto c, ?a_1 \mapsto a, ?a_2 \mapsto a\}, \\
&\quad\ \{?x \mapsto b, ?a \mapsto c, ?a_1 \mapsto b, ?a_2 \mapsto b\}\}, \\
[\![Q_{\neg\exists x T(x,a,a)}]\!]_D &= \{\{?a' \mapsto c\}\}, \\
[\![Q_{\varphi}]\!]_D &= \{\{?a'' \mapsto c\}\},
\end{aligned}$$

$$\begin{aligned}
[\![enc(T(x, a, a))]\!]_D &= \{\{?x \mapsto a, ?a \mapsto c, ?a_1 \mapsto a, ?a_2 \mapsto a\}\}, \\
[\![enc(\neg\exists T(x, a, a))]\!]_D &= \emptyset, \text{ and finally} \\
[\![enc(\varphi)]\!]_D &= \{\{?a'' \mapsto c\}\}.
\end{aligned}$$

This coincides with the observation that $D \models \varphi$. It is easily verified that, when evaluating $enc(\varphi)$ on e.g. $D' := \{(b, b, b)\} \not\models \varphi$ we obtain the empty result. □

5.4. Semantic Query Optimization for SPARQL

We now present a scheme for the semantic optimization (SQO) of SPARQL queries. Generally speaking, the key idea of SQO is as follows. Given a query and a set of integrity constraints over the database, the goal is to find more efficient queries that are semantically equivalent to the original query for each database instance that satisfies the constraints. The constraints that are given as input may have been specified by the user, automatically extracted from the underlying database, or – in our scenario – may be implicitly given by the semantics of RDFS when SPARQL is coupled with an RDFS inference system (we will describe the latter setting in more detail later in Section 5.4.1). Formally, we define the SQO problem for SPARQL as follows: given a SPARQL expression or query Q and a set of RDF constraints Σ, we want to enumerate expressions or queries Q' (typically with some desired property, such as minimality or optimality w.r.t. some cost measure) that are equivalent to Q on every database D such that $D \models \Sigma$. If Q is equivalent to Q' on every D such that $D \models \Sigma$, we say that Q and Q' are Σ-equivalent and denote this by $Q \equiv_\Sigma Q'$.

We note that constraint-based query optimization in the context of RDFS inference has been discussed before in [SKCT05]. Our approach is much more general and supports constraints beyond those implied by the semantics of RDFS, i.e. it also works on top of user-defined or automatically extracted constraints. In [LMS08], for instance, we propose to carry over constraints from relational databases, such as primary and foreign keys, when translating relational data into RDF. Also the latter may serve as input to our semantic optimization scheme. As another difference to [SKCT05], our approach addresses the specifics of SPARQL, e.g. we also provide rules for the semantic optimization of queries that involve operator Opt.

5.4.1. A Motivating Scenario

Before presenting our constraint-based SPARQL optimization scheme, we will sketch a motivating scenario that illustrates the benefits of semantic query optimization in the context of SPARQL and RDF. We start with the observation that, as discussed earlier in Section 2.2.2, the SPARQL standard disregards RDFS reasoning, but evaluates queries on the database as is. Consider for instance the RDF database

$$D := \{(\mathit{knows}, \mathit{rdfs{:}domain}, \mathit{Person}), (\mathit{knows}, \mathit{rdfs{:}range}, \mathit{Person}), \\ (\mathit{P1}, \mathit{knows}, \mathit{P2}), (\mathit{P1}, \mathit{knows}, \mathit{P3})\}$$

The first and the second triple in the database state that the domain and range of property *knows* is *Person*. According to the RDFS semantics definition (cf. Section 2.2.2, Figure 2.5), these two triples imply that, for each triple (s, knows, o) in the database the subject s and the object o are of type *Person*. Formally speaking, the triples $(\mathit{knows}, \mathit{rdfs{:}domain}, \mathit{Person})$ and $(\mathit{knows}, \mathit{rdfs{:}range}, \mathit{Person})$ thus can be understood as tuple-generating dependencies

$$\varphi_d := \mathit{PDom}(\mathit{knows}, \mathit{Person}) = \forall s, o(T(s, \mathit{knows}, o) \rightarrow T(s, \mathit{rdf{:}type}, \mathit{Person})), \\ \varphi_r := \mathit{PRan}(\mathit{knows}, \mathit{Person}) = \forall s, o(T(s, \mathit{knows}, o) \rightarrow T(o, \mathit{rdf{:}type}, \mathit{Person})),$$

which assert that subjects and objects occurring in triples with predicate *knows* are typed accordingly with class *Person*. In this line, the RDFS reasoning process can be seen as a chase sequence that fixes constraint violations in the database, i.e. for database D this process would derive the fresh triples $t_1 := (\mathit{P1}, \mathit{rdf{:}type}, \mathit{Person})$, $t_2 := (\mathit{P2}, \mathit{rdf{:}type}, \mathit{Person})$, $t_3 := (\mathit{P3}, \mathit{rdf{:}type}, \mathit{Person})$. Hence, according to the semantics of RDFS, database D is equivalent to $D' := D \cup \{t_1, t_2, t_3\}$.

Now let us see what happens when evaluating SPARQL queries on top of the two RDF databases defined above. Let us exemplarily consider the SPARQL query

$$Q := \textsc{Select}_{?p1,?p2}((?p1, \mathit{rdf{:}type}, \mathit{Person}) \textsc{ And } (?p2, \mathit{rdf{:}type}, \mathit{Person}) \\ \textsc{And } (?p1, \mathit{knows}, ?p2)),$$

which is supposed to extract all pairs of persons (represented through variables *?p1* and *?p2*) that know each other. When evaluated on the original database D, we observe that $[\![Q]\!]_D = \emptyset$, because D lacks triples stating that *P1*, *P2*, and *P3* are *Person*-typed objects. In contrast, when evaluating Q on the implied database D' we get the desired result $[\![Q]\!]_{D'} = \{\{?p1 \mapsto \mathit{P1}, ?p2 \mapsto \mathit{P2}\}, \{?p1 \mapsto \mathit{P1}, ?p2 \mapsto \mathit{P3}\}\}$.

At first glance, this seems paradoxical and one might argue that the decision to exclude RDFS inferencing from the SPARQL semantics specification was an unfortunate design decision. However, the idea is that SPARQL can easily be coupled with an RDFS inference engine whenever RDFS reasoning is desired. In such a scenario, the SPARQL engine would request the implied database from the underlying RDFS reasoning engine and therefore always operates on top of the implied database.

The interesting observation here – which also is one of the major motivations for our study of constraint-based SPARQL optimization – is that, whenever SPARQL engines are coupled with RDFS inference engines, then φ_d and φ_r can be understood as hard database constraints: the RDFS inference mechanism guarantees that these constraints are always satisfied on the implied database. Consequently, the SPARQL engine can take these constraints as granted and may exploit them for query optimization. In fact, it is easy to see that, for each database instance that satisfies constraints φ_d and φ_r, query Q is equivalent to the (arguably simpler) query

$$Q^{opt} := \textsc{Select}_{?p1,?p2}((?p1, \mathit{knows}, ?p2)),$$

because φ_r and φ_d imply the existence of the missing tuples $(?p1, \mathit{rdf{:}type}, \mathit{Person})$ and $(?p2, \mathit{rdf{:}type}, \mathit{Person})$ in the database. Thus, instead of evaluating the original query Q, a SPARQL engine that operates on top of an RDFS inference layer could evaluate the simpler query Q^{opt}. The SQO scheme that we present in the following would propose Q^{opt} as an alternative when given Q and constraints φ_r, φ_d as input.

In the example above, we interpreted RDF triples containing RDFS vocabulary as data dependencies. We can even go one step further and interpret the RDFS reasoning rules (cf. Figure 2.5) as integrity constraints. Consider for instance the two rules in group (III) of Figure 2.5. These rules can be expressed by the TGDs

$$\varphi_{\mathit{typing1}} := \forall a, c, x, y(T(a, \mathit{rdfs{:}domain}, c) \wedge T(x, a, y) \rightarrow T(x, \mathit{rdf{:}type}, c)),$$
$$\varphi_{\mathit{typing2}} := \forall a, c, x, y(T(a, \mathit{rdfs{:}range}, c) \wedge T(x, a, y) \rightarrow T(y, \mathit{rdf{:}type}, c)),$$

which – like φ_d and φ_r – imply database D' when D is given as input. Analogously to $\varphi_{\mathit{typing1}}$ and $\varphi_{\mathit{typing2}}$, we can easily express the remaining rules from Figure 2.5 (and other RDFS rules [rdfd; GHM04]) as first-order sentences. All these RDFS constraints then can be used as a basis for semantic query optimization by SPARQL engines that are built on top of an RDFS reasoning layer. This fixed RDFS constraint base can be extended by user-defined constraints and automatically extracted constraints that are known to hold on the (implied) input database, whenever available.

5.4.2. Chase-based Optimization of SPARQL

Having motivated semantic query optimization for SPARQL, we now come to a detailed discussion of our SQO scheme. The basic idea of our approach is as follows. Given a SPARQL query and a set of constraints, we first translate And-only blocks (or full And-only queries), into conjunctive queries. In a second step, we then use the Chase & Backchase (C&B) algorithm [DPT06] described in Section 5.1.4 to minimize these conjunctive queries and translate the minimized CQs (i.e., the output of the C&B algorithm) back into SPARQL, which usually gives us more efficient SPARQL queries. It should be mentioned that the C&B algorithm by default returns the set of all Σ-equivalent queries that are minimal w.r.t. the number of atoms in the body

of the query. There are, of course, no guarantees that these queries can be evaluated more efficiently than the original one. We therefore emphasize that, as described in [DPT06], the C&B algorithm also can be coupled with a cost estimation function and in that case would return the set of queries that are minimal w.r.t. the estimated evaluation cost. In the absence of a cost function, though, we restrict ourselves to the minimality property in the following discussion, but point out that our approach also works in combination with advanced, implementation-specific cost measures.

The optimization scheme described above, which is restricted to AND-only queries or AND-only subqueries, will be described in more detail in Section 5.4.3. Complementarily, in Section 5.4.4 we discuss SPARQL-specific rules that allow for the semantic optimization of complex queries involving operators FILTER and OPT.

5.4.3. Optimizing AND-only Blocks

We start with a translation scheme for AND-only queries into conjunctive queries. We reuse the notation introduced in Section 3.2, e.g. writing $\mathcal{A}^\pi$ for SPARQL queries of the form $\text{SELECT}_S(Q)$, where $S \subset V$ and Q is an AND-only expression.

Definition 5.20 Let $S \subset V$ and $Q \in \mathcal{A}^\pi$ be a SPARQL query defined as

$$Q := \text{SELECT}_S((s_1, p_1, o_1) \text{ AND } \ldots \text{ AND } (s_n, p_n, o_n)),$$

We define the translation $cq(Q) := q$ of SPARQL query Q into CQ q, where

$$q := ans(\overline{s}) \leftarrow T(s_1, p_1, o_1) \wedge \cdots \wedge T(s_n, p_n, o_n),$$

the tuple $\overline{s}$ contains exactly the variables from S, and in q we interpret variables from S as elements from V_R and elements from LU as constants from Δ.

Further, we define $cq^{-1}(q)$ as follows. It takes a CQ in the form of q as input and returns Q if it is a valid SPARQL query, i.e. if $(s_i, p_i, o_i) \in UV \times UV \times LUV$ for all $i \in [n]$; in case Q is not a valid SPARQL query, $cq^{-1}(q)$ is undefined. □

Functions cq and cq^{-1} are straightforward translations from AND-only SPARQL queries to CQs and back. We illustrate their definition by example:

Example 5.13 Consider query Q from the running example in Section 5.4.1. Then

$$\begin{aligned} cq(Q) = ans(\textit{?p1}, \textit{?p2}) \leftarrow\ & T(\textit{?p1}, \textit{knows}, \textit{?p2}) \wedge T(\textit{?p1}, \textit{rdf:type}, \textit{Person}) \\ & \wedge T(\textit{?p2}, \textit{rdf:type}, \textit{Person}) \end{aligned}$$

and $cq^{-1}(cq(Q)) = Q$. As another example, $cq^{-1}(ans(?x) \leftarrow T(\textit{“a”}, \textit{rdf:type}, ?x))$ is undefined: $\text{SELECT}_{?x}((\textit{“a”}, \textit{rdf:type}, ?x))$ is not valid SPARQL, because literal *“a”* appears in subject position. □

The key property of our translation functions is that they are semantics-preserving:

Lemma 5.1 Let $q : ans(?x_1, \ldots, ?x_n) \leftarrow \varphi(?x_1, \ldots, ?x_n, \overline{y})$ be a conjunctive query and $Q^\pi \in \mathcal{A}$ be a SPARQL query s.t. $cq(Q) = q$ and $cq^{-1}(q) = Q$. For every RDF document D and associated relational instance $I := \{T(s,p,o) \mid (s,p,o) \in D\}$ it holds that $\{?x_1 \mapsto c_1, \ldots, ?x_n \mapsto c_n\} \in [\![Q]\!]_D \Leftrightarrow (c_1, \ldots, c_n) \in q(I)$. □

The lemma shows that we can safely transform SPARQL AND-only queries into conjunctive queries, apply equivalence transformations, and translate the resulting CQ back into SPARQL without changing the semantics. The proof of the lemma is straightforward and relies on the observation that operator AND implements a join in the style of conjunctive queries, where the atoms in the body are joined together (observe that all variables in AND-only expressions are bound in SPARQL result mappings, so unbound variables are not an issue here). We omit the details.

Our first result is that, when coupled with the C&B algorithm, the forth-and-back translations cq and cq^{-1} provide a sound approach to semantic query optimization for AND-only queries whenever the underlying chase algorithm terminates regularly:

Lemma 5.2 Let Q be an $\mathcal{A}^\pi$ expression, D be an RDF database, and let Σ be a set of EGDs and TGDs. If $cb_\Sigma(cq(Q))$ is defined, $q \in cb_\Sigma(cq(Q))$, and $cq^{-1}(q)$ is defined, then $cq^{-1}(q) \equiv_\Sigma Q$. □

The lemma follows immediately from the correctness of the C&B algorithm and the correctness of the functions cq and cq^{-1} stated in Lemma 5.1:

Proof of Lemma 5.2
Let $Q' \in cq^{-1}(cb_\Sigma(cq(Q))) \cap \mathcal{A}^\pi$. Then $cq(Q') \in cb_\Sigma(cq(Q))$. This directly implies that $cq(Q') \equiv_\Sigma cq(Q)$ and it follows (by Lemma 5.1) that $Q' \equiv_\Sigma Q$. □

Lemma 5.2 formalizes the key idea of our SQO scheme: given that the chase result for the translation $cq(Q)$ of an AND-only query Q is defined (which implies that the chase terminates), we can apply the C&B algorithm to $cq(Q)$ and translate the resulting minimal queries back into SPARQL, to obtain minimal SPARQL AND-only queries that are Σ-equivalent to Q. The following example clarifies the idea.

Example 5.14 Consider queries Q, Q^{opt}, and constraints $\varphi_{typing1}$, $\varphi_{typing2}$ from Section 5.4.1. Put $\Sigma := \{\varphi_{typing1}, \varphi_{typing2}\}$. First note that Σ is weakly acyclic, so the chase with Σ always terminates. Moreover, we have that $cq(Q^{opt}) \in cb_\Sigma(cq(Q))$. It follows from Lemma 5.2 that $cq^{-1}(cq(Q^{opt})) = Q^{opt}$ is Σ-equivalent to Q. □

We emphasize that our approach is not restricted to AND-only queries, but also works for AND-connected blocks inside more complex queries, i.e. we can apply our scheme to AND-connected blocks and replace the original blocks by the minimized blocks delivered by the C&B algorithm. Given that our translation function cq is

defined for SPARQL queries (rather than expressions), there are two possibilities to implement this idea. First, observe that every expression $Q \in \mathcal{A}$ is equivalent to the $\mathcal{A}^\pi$ query $\textsc{Select}_{pVars([\![Q]\!]_D)}(Q)$, so we could simply use the latter query for our optimization scheme. A second approach is to map the whole SPARQL expression to SPARQL algebra, apply the projection pushing rules from Figure 4.3 to push top-level projection down on top of $\bowtie$-connected blocks, and interpret the inner projection expressions as AND-only queries. Interestingly, this may gives us better optimization results as the first approach, as illustrated in the following example.

Example 5.15 Consider the SPARQL query

$$Q' := \textsc{Select}_{?p}(((?p, \mathit{rdf{:}type}, \mathit{Person}) \textsc{ And } (?p, \mathit{email}, ?e)) \textsc{ Filter } (\neg ?p = \mathit{P1})),$$

which selects all persons different from *P1* that have an email address. Assume that the constraint set $\Sigma' := \{Min^\bullet(\mathit{Person}, \mathit{email}, 1)\}$ is given, stating that each object of type *Person* has an email address. Now assume we want to optimize the inner AND block $(?p, \mathit{rdf{:}type}, \mathit{Person})$ AND $(?p, \mathit{email}, ?e)$. One possibility is to interpret this block as query $Q_i := \textsc{Select}_{?p,?e}((?p, \mathit{rdf{:}type}, \mathit{Person}) \textsc{ And } (?p, \mathit{email}, ?e))$. This query, however, is already minimal and the C&B algorithm (when given Σ' as input) is not able to minimize $cq(Q_i)$, because variable $?e$ appears in the result.

An alternative way is to translate Q' into SPARQL algebra, which gives us

$$[\![Q']\!]_D := \pi_{?p}(\sigma_{\neg ?p=P1}([\![(?p, \mathit{rdf{:}type}, \mathit{Person})]\!]_D \bowtie [\![(?p, \mathit{email}, ?e)]\!]_D))$$

Applying rule (*PFPush*) from Figure 4.3, we transform this expression into

$$A' := \pi_{?p}(\sigma_{\neg ?p=P1}(\pi_{?p}([\![(?p, \mathit{rdf{:}type}, \mathit{Person})]\!]_D \bowtie [\![(?p, \mathit{email}, ?e)]\!]_D))).$$

In the next step we translate the inner projection expression back into the SPARQL query $Q'_i := \textsc{Select}_{?p}((?p, \mathit{rdf{:}type}, \mathit{Person}) \textsc{ And } (?p, \mathit{email}, ?e))$ and pass the conjunctive query $cq(Q'_i)$ and Σ' to the C&B algorithm. The output of C&B (interpreted as SPARQL query) is $\textsc{Select}_{?p}((?p, \mathit{rdf{:}type}, \mathit{Person}))$, which is equivalent to $Q_{min} := (?p, \mathit{rdf{:}type}, \mathit{Person})$. Substituting Q_{min} into the original query, we get

$$Q'_{opt} := \textsc{Select}_{?p}((?p, \mathit{rdf{:}type}, \mathit{Person}) \textsc{ Filter } (\neg ?p = \mathit{P1}))$$

as an optimized version of Q' (which is equivalent to Q' on every $D \models \Sigma$).

The crucial point here is that Q'_i has an optimized SELECT clause, which projects only $?p$ instead of $?p$ and $?e$. The example shows that our semantic optimization approach for SPARQL can benefit from the algebraic rewriting rules in Chapter 4.□

Coming back to the discussion of Lemma 5.2, the observant reader may have noticed that it states only **soundness** of the SQO scheme for AND-only queries. In fact, one can observe that under certain circumstance the scheme proposed in Lemma 5.2 is not **complete**. We demonstrate the problem in the following example.

Example 5.16 Consider the SPARQL queries

$Q_1 := \textsc{Select}_{?x}((?x, a, \text{“}l\text{”}))$,
$Q_2 := \textsc{Select}_{?x}((?x, a, \text{“}l\text{”}) \textsc{ And } (?x, b, c))$,

and $\Sigma := \{\forall x, y, z(T(x, y, z) \rightarrow T(z, y, x))\}$. It holds that $Q_1 \equiv_\Sigma Q_2$ because the answer to both Q_1 and Q_2 is always the empty set on documents that satisfy Σ: the single constraint in Σ enforces that all RDF documents satisfying Σ have no literal in object position, because otherwise this literal would also appear in subject position, which is invalid RDF. On the other hand, $cq(Q_1) \equiv_\Sigma cq(Q_2)$ does not hold. To see why, consider the relational instance $I := \{T(a, a, \text{“}l\text{”}), T(\text{“}l\text{”}, a, a)\}$, where $a, \text{“}l\text{”} \in \Delta$. We observe that $I \models \Sigma$, $(cq(Q_1))(I) = \{(a)\}$, and $(cq(Q_2))(I) = \emptyset$. Therefore, our scheme would not detect Σ-equivalence between Q_1 and Q_2. □

Arguably, Example 5.16 presents a constructed scenario and it seems reasonable to assume that such constraints (which in some sense contradict the type restrictions of RDF) do not occur in practice. We next provide a precondition that guarantees completeness for virtually all practical scenarios. It relies on the observation that, in the example above, $(cq(Q_1))^\Sigma$ and $(cq(Q_2))^\Sigma$ (i.e., the queries obtained when chasing $cq(Q_1)$ and $cq(Q_2)$ with Σ, respectively) do not reflect valid SPARQL queries. We can guarantee completeness if we explicitly exclude such cases:

Lemma 5.3 Let D be an RDF database and let Q be an $\mathcal{A}^\pi$ expression such that $cq^{-1}((cq(Q))^\Sigma) \in \mathcal{A}^\pi$. If $cb_\Sigma(cq(Q))$ terminates then for all $Q' \in \mathcal{A}^\pi$ such that $cq^{-1}((cq(Q'))^\Sigma) \in \mathcal{A}^\pi$ we have $Q' \in cq^{-1}(cb_\Sigma(cq(Q))) \Leftrightarrow Q' \equiv_\Sigma Q$ and Q' minimal.□

Conditions $cq^{-1}((cq(Q))^\Sigma) \in \mathcal{A}^\pi$ and $cq^{-1}((cq(Q'))^\Sigma) \in \mathcal{A}^\pi$ in the lemma encode exactly the before-mentioned restrictions that the back-translation of the chase result for $cq(Q)$ and $cq(Q')$ with Σ are valid SPARQL (And-only) queries.

Proof of Lemma 5.3
Direction “⇒” follows from Lemma 5.2, so it suffices to prove direction “⇐”. So let us assume that $Q' \equiv_\Sigma Q$ and Q' is minimal. First observe that $cq^{-1}((cq(Q'))^\Sigma)$ and $cq^{-1}((cq(Q))^\Sigma)$ are $\mathcal{A}^\pi$ expressions. From Lemma 5.1 it follows that $cq(Q') \equiv_\Sigma cq(Q)$. From this observation, the minimality of Q', and the correctness of the translation in Lemma 5.1 it follows that $cq(Q') \in cb_\Sigma(cq(Q))$ and $Q' \in cq^{-1}(cb_\Sigma(cq(Q)))$.□

5.4.4. SPARQL-specific Optimization

By now we have established a mechanism that allows us to enumerate equivalent minimal queries of SPARQL And-only queries or subqueries. Next, we present extensions that go beyond And-only queries. We open the discussion with rewritings that can be used to simplify queries involving operator Filter:

Lemma 5.4 Let $Q_1, Q_2 \in \mathcal{A}$, $S \subset V \setminus \{?y\}$ be a set of variables, Σ be a set of TGDs and EGDs, $?x, ?y \in pVars([\![Q_2]\!]_D)$, and D be a document such that $D \models \Sigma$. We write $Q_2\frac{?x}{?y}$ to denote the query obtained from Q_2 by replacing each occurrence of $?y$ through $?x$. The following rules are valid.

(FSI) If $Q_2 \equiv_\Sigma Q_2$ FILTER $(?x = ?y)$, then
$\text{SELECT}_S(Q_2) \equiv \text{SELECT}_S(Q_2\frac{?x}{?y})$.

(FSII) If $Q_2 \equiv_\Sigma Q_2$ FILTER $(?x = ?y)$, then
$[\![Q_2 \text{ FILTER } (\neg(?x = ?y))]\!]_D = \emptyset$.

(FSIII) If $Q_1 \equiv_\Sigma \text{SELECT}_{pVars([\![Q_1]\!]_D)}(Q_1 \text{ AND } Q_2)$, then
$[\![(Q_1 \text{ OPT } Q_2) \text{ FILTER}(\neg bnd(?x))]\!]_D = \emptyset$. □

The intended use of the rewriting rules in Lemma 5.4 is as follows. We utilize the chase algorithm to check if the precondition holds; whenever this is the case, then we may exploit the equivalences in the conclusion. More precisely, the precondition $Q_1 \equiv_\Sigma \text{SELECT}_{pVars([\![Q_1]\!]_D)}(Q_1 \text{ AND } Q_2)$ in rule *(FSIII)* can be checked by testing if $cq(\text{SELECT}_{pVars([\![Q_1]\!]_D)}(Q_1))$ and $cq(\text{SELECT}_{pVars([\![Q_1]\!]_D)}(Q_1 \text{ AND } Q_2))$ are Σ-equivalent (it is folklore that Σ-equivalence between CQs can be tested with the chase whenever the chase result for both queries is defined). To test the preconditions for *(FSI)* and *(FSII)*, we can check if $\Sigma \models body(cq(Q_2)) \rightarrow ?x = ?y$ holds, which can easily be proven to be equivalent to the precondition $Q_2 \equiv_\Sigma Q_2$ FILTER $(?x = ?y)$; thus, we reduce the problem of checking the precondition to the implication problem for constraints, which has been studied e.g. in [BB79; MMS79; BV84; CV85].

The first rule *(FSI)* in Lemma 5.4 states that, if the constraint set implies equivalence between $?x$ and $?y$ (in some AND-only query), we can replace each occurrence of $?y$ by $?x$ if $?y$ if projected away (observe that $S \subset V \setminus \{?y\}$ by assumption). Rule *(FSII)* states that, under the same precondition, a filter for condition $\neg(?x = ?y)$ is never satisfied. Finally *(FSIII)* tackles negation queries discussed before in Section 4.2.6. We illustrate the idea behind rule *(FSIII)* in the following example.

Example 5.17 Consider the SPARQL query

$$Q_f := \text{SELECT}_{?s,?m}(((?s, \textit{rdf:type}, \textit{Student}) \text{ OPT } (?s, \textit{matric}, ?m)) \text{ FILTER } (\neg bnd(?m))),$$

which selects all students without matriculation number. Further assume that the constraint set $\Sigma := \{Min^\bullet(\textit{Student}, 1, \textit{matric})\}$ is given, which asserts that every student has a matriculation number. First note that Σ is weakly acyclic, so the chase with Σ terminates for every instance. Next, we observe that the precondition of rule *(FSIII)* from Lemma 5.4 holds, i.e. it is easy to see that $(?s, \textit{rdf:type}, \textit{Student})$ is Σ-equivalent to $\text{SELECT}_{?s}((?s, \textit{rdf:type}, \textit{Student}) \text{ AND } (?s, \textit{matric}, ?m))$. We therefore conclude that $[\![Q_f]\!]_D = \emptyset$ holds on every document $D \models \Sigma$. □

Having discussed the idea behind the rules, we next prove them formally.

Proof of Lemma 5.4

Rule (FSI): $\Rightarrow$: Assume that $Q_2 \equiv_\Sigma Q_2$ FILTER $(?x = ?y)$ and consider a mapping $\mu \in [\![\text{SELECT}_S(Q_2)]\!]_D$. Then μ is obtained from some $\mu_l \in [\![Q_2]\!]_D$ by projecting S. By precondition, μ_l is also contained in $[\![Q_2 \text{ FILTER } (?x = ?y)]\!]_D$, so we know that $?x, ?y \in dom(\mu_l)$ and $\mu_l(?x) = \mu_l(?y)$. It is easily verified that in this case there is a mapping $\mu_r \in Q_2\frac{?x}{?y}$ that agrees with μ_l on all variables $dom(\mu_l) \backslash ?y$ and is unbound for $?y$ (cf. the proof of rule *(FElimI)* from Lemma 4.3). Given that $?y \notin S$ and the observation that μ is obtained from μ_l by projecting S, we conclude that μ is also obtained from μ_r when projecting S. Consequently, μ is generated by the right side expression $\text{SELECT}_S(Q_2\frac{?x}{?y})$. $\Leftarrow$: Assume that $Q_2 \equiv_\Sigma Q_2$ FILTER $(?x = ?y)$ and consider a mapping $\mu \in [\![\text{SELECT}_S(Q_2\frac{?x}{?y})]\!]_D$. Then μ is obtained from some $\mu_r \in [\![Q_2\frac{?x}{?y}]\!]_D$ by projecting the variables S. It can be shown that then the mapping $\mu_l := \mu_r \cup \{?y \mapsto \mu_r(?x)\}$ is contained in $[\![Q_2]\!]_D$ (cf. the proof of rule *(FElimI)* from Lemma 4.3). Given that $?y \notin S$ and the observation that μ is obtained from μ_r by projecting S, we conclude that μ is also obtained from μ_l when projecting S. Consequently, the left side expression $[\![\text{SELECT}_S(Q_2)]\!]_D$ generates mapping μ.

Rule (FSII): Follows directly by assumption, i.e. each $\mu \in [\![Q_2]\!]_D$ is also generated by $[\![Q_2 \text{ FILTER } (?x = ?y)]\!]_D$ and therefore cannot satisfy the filter condition $\neg ?x = ?y$.

Rule (FSIII): First observe that precondition $?x \in pVars([\![Q_2]\!]_D)$ and $Q_2 \in \mathcal{A}$ implies that $?x \in cVars([\![Q_2]\!]_D)$. From Proposition 4.2 we obtain that $?x \in dom(\mu_2)$ foreach $\mu_2 \in [\![Q_2]\!]_D$ and it easily follows from Definition 4.2 that $?x \in dom(\mu)$ foreach mapping $\mu \in [\![Q_1 \text{ AND } Q_2]\!]_D$. Now consider the expression

$$[\![Q_1 \text{ OPT } Q_2]\!]_D = ([\![Q_1]\!]_D \bowtie [\![Q_2]\!]_D) \cup ([\![Q_1]\!]_D \setminus [\![Q_2]\!]_D)$$

and put $\Omega_\bowtie := [\![Q_1]\!]_D \bowtie [\![Q_2]\!]_D = [\![Q_1 \text{ AND } Q_2]\!]_D$, $\Omega_\setminus := [\![Q_1]\!]_D \setminus [\![Q_2]\!]_D$. From the above considerations we know that $?x \in dom(\mu_\bowtie)$ foreach $\mu_\bowtie \in \Omega_\bowtie$. We now argue that $\Omega_\setminus = \emptyset$, which implies that the equivalence holds, because then $?x \in dom(\mu)$ for every $\mu \in [\![Q_1 \text{ OPT } Q_2]\!]_D$ and therefore no mapping satisfies the filter condition $\neg bnd(?x)$. To show that $\Omega_\setminus = \emptyset$ let us for the sake of contradiction assume there is $\mu_\setminus \in \Omega_\setminus$. This implies that $\mu_\setminus \in [\![Q_1]\!]_D$ and there is no compatible mapping $\mu_2 \sim \mu_\setminus$ in $[\![Q_2]\!]_D$. Now observe that by assumption $\mu_\setminus \in [\![\text{SELECT}_{pVars([\![Q_1]\!]_D)}(Q_1 \text{ AND } Q_2)]\!]_D$. Hence, there must be $\mu_1 \in [\![Q_1]\!]_D$, $\mu_2 \in [\![Q_2]\!]_D$ such that $\mu_1 \sim \mu_2$ and $\mu_1 \cup \mu_2 \supseteq \mu_\setminus$. Consequently, it trivially holds that $\mu_2 \sim \mu_\setminus$, which contradicts the initial assumption that there is no compatible mapping $\mu_2 \sim \mu_\setminus$ in $[\![Q_2]\!]_D$. $\square$

Having presented semantic rewriting rules for the FILTER operator, we finally turn towards operator OPT, the most complex construct in the SPARQL query language (as discussed in Chapter 3). The following lemma lists two useful semantic rewriting rules for the OPT operator, similar in style to the previous filter rewriting rules.

Lemma 5.5 Let $Q_1, Q_2, Q_3 \in \mathcal{A}$ and $S \subset V$.

(OSI) If $Q_1 \equiv_\Sigma \textsc{Select}_{pVars([\![Q_1]\!]_D)}(Q_1 \textsc{ And } Q_2)$, then $Q_1 \textsc{ Opt } Q_2 \equiv_\Sigma Q_1 \textsc{ And } Q_2$.

(OSII) If $Q_1 \equiv_\Sigma Q_1 \textsc{ And } Q_2$, then $Q_1 \textsc{ Opt } (Q_2 \textsc{ And } Q_3) \equiv_\Sigma Q_1 \textsc{ Opt } Q_3$. □

Analogical to Lemma 5.4, we can check the preconditions using the chase algorithm. Rule *(OSI)* shows that in some cases Opt can be replaced by And; informally speaking, this rule applies when the expression in the Opt clause is implied by the constraints set. We argue that it is particularly useful when the author of the query is not fully aware of all integrity constraints that hold on the input database: in such cases, she may specify parts of the query that are implicit by the constraints as optional, just to be sure not to miss relevant answers. Rule *(OSII)* is useful to eliminate redundant And-only subexpressions in Opt clauses. Before proving the rules, we demonstrate *(OSI)* by means of a small, easy-to-verify example.

Example 5.18 We sketch a possible optimization for query

$Q_o := \textsc{Select}_{?p,?gn,?sn}(t_1 \textsc{ Opt } (t_2 \textsc{ And } t_3))$, with

$t_1 := (?p, \mathit{rdf{:}type}, \mathit{Person})$, $t_2 := (?p, \mathit{givenname}, ?gn)$, $t_3 := (?p, \mathit{surname}, ?sn)$ under constraint set $\Sigma := \mathit{Total}^\bullet(\mathit{Person}, \mathit{givenname}) \cup \mathit{Total}^\bullet(\mathit{Person}, \mathit{surname})$. Note that, when instantiating the constraint templates according to their definition in Figure 5.3, we obtain a set of TGDs and EGDs and it is straightforward to show that this constraint set is weakly acyclic, so the chase will always terminate.

Next, it is easily verified that t_1 is Σ-equivalent to $\textsc{Select}_{?p}(t_1 \textsc{ And } (t_2 \textsc{ And } t_3))$, so optimization rule *(OSI)* from Lemma 5.5 is applicable. We therefore conclude that

$Q'_o := \textsc{Select}_{?p,?gn,?sn}(t_1 \textsc{ And } (t_2 \textsc{ And } t_3))$

is Σ-equivalent to Q_o. This transformation also clears the way for further optimizations. For instance, using the algebraic equivalences we now could reorder the triples t_1, t_2, t_3 in Q'_o according to rules *(JIdem)* and *(JAss)* from Figure 4.1. □

We conclude our discussion of SQO for SPARQL with the proof of Lemma 5.5:

Proof of Lemma 5.5

Rule (OSI): We transform $Q := [\![Q_1 \textsc{ Opt } Q_2]\!]_D$ systematically. Let D be an RDF database such that D satisfies all constraints in Σ. Then

$$\begin{aligned}[\![Q]\!]_D &= [\![Q_1 \textsc{ Opt } Q_2]\!]_D \\ &= ([\![Q_1]\!]_D \bowtie [\![Q_2]\!]_D) \cup ([\![Q_1]\!]_D \setminus [\![Q_2]\!]_D) \\ &= [\![Q_1 \textsc{ And } Q_2]\!]_D \cup (\pi_{pVars([\![Q_1]\!]_D)}([\![Q_1 \textsc{ And } Q_2]\!]_D) \setminus [\![Q_2]\!]_D)\end{aligned}$$

It is easy to verify that each mapping in $[\![Q_1 \text{ AND } Q_2]\!]_D$ is compatible with at least one mapping in Q_2, and the same holds for $\pi_{pVars([\![Q_1]\!]_D)}([\![Q_1 \text{ AND } Q_2]\!]_D)$. Hence, the right side union subexpression can be dropped and we obtain $Q \equiv_\Sigma Q_1 \text{ AND } Q_2$.

Rule (OSII): Let D be an RDF database such that $D \models \Sigma$. We transform the expression $Q := [\![Q_1 \text{ OPT } (Q_2 \text{ AND } Q_3)]\!]_D$ schematically:

$$\begin{aligned}
[\![Q]\!]_D &= [\![(Q_1 \text{ OPT } (Q_2 \text{ AND } Q_3))]\!]_D \\
&= [\![Q_1 \text{ AND } Q_2 \text{ AND } Q_3]\!]_D \cup ([\![Q_1]\!]_D \setminus [\![Q_2 \text{ AND } Q_3]\!]_D) \\
&= [\![Q_1 \text{ AND } Q_3]\!]_D \cup ([\![Q_1 \text{ AND } Q_2]\!]_D \setminus [\![Q_2 \text{ AND } Q_3]\!]_D) \\
&\overset{(*)}{=} [\![Q_1 \text{ AND } Q_3]\!]_D \cup ([\![Q_1 \text{ AND } Q_2]\!]_D \setminus [\![Q_3]\!]_D) \\
&= ([\![Q_1]\!]_D \bowtie [\![Q_3]\!]_D) \cup ([\![Q_1]\!]_D \setminus [\![Q_3]\!]_D) \\
&= [\![Q_1 \text{ OPT } Q_3]\!]_D,
\end{aligned}$$

where step (*) follows from the observation that the equation

$$[\![Q_1 \text{ AND } Q_2]\!]_D \setminus [\![Q_2 \text{ AND } Q_3]\!]_D \equiv [\![Q_1 \text{ AND } Q_2]\!]_D \setminus [\![Q_3]\!]_D$$

holds; the formal proof of this equation is similar in style to the rewriting presented in Example 4.13 (at algebraic level). We omit the details. □

5.5. Related Work

Data Dependencies. Short time after the invention of the relational model [Cod69; Cod70], Codd proposed functional dependencies for relational databases and used them to develop the notions of 2NF and 3NF [Cod71; Cod72]. Since then, considerable effort has been spent in the investigation of integrity constraints for the relational model. New constraint classes beyond functional dependencies have been proposed and systematically explored, such as multi-valued dependencies [Fag77], join and inclusion dependencies [Dat81], or embedded multivalued dependencies [Fag77] (see [AHV] for an overview of these classes and a summary of important results).

Closely related to the study of constraints is the theory of data integrity and normal forms for the relational model, which are important to avoid update, insert, and deletion anomalies in database instances. Beyond the above-mentioned 2NF and 3NF, more advanced normal forms have been proposed over the years, such as the Boyce-Codd normal form [Cod74] or the fourth normal form [Fag77].

Another central problem in the context of data dependencies is the constraint implication problem for different classes of constraints [BB79; MMS79; BV84; CV85], i.e. the question whether, given a constraint set Σ and a single constraint φ, Σ implies φ. Whether this problem is decidable or not depends on the classes of the input constraints. To give an example, the implication problem for the class of functional dependencies is decidable, but it becomes undecidable when adding inclusion dependencies [CV85]. The interested reader will find initial results on the decidability

of the implication problem for RDF constraints in [LMS08], where we identify subclasses of RDF constraints for which the implication is decidable and undecidable.

Related to the formalization of constraints in first-order logic, in [Nic78] it was shown that constraint classes like functional, join, and inclusion dependencies (originally defined using ad-hoc syntax) have natural representations in first-order logic. A recent summary of first-order logic representations of integrity constraints was given in [Deu08]. The formalization in this chapter followed these two approaches.

Semantic Query Optimization and Chase. Beyond their investigation in the context of data integrity, data dependencies have been extensively studied in the area of constraint-based query optimization (see e.g. [ASU79; Kin81; JK82; SO87; CGM90; BFW99; DT01; DT05; CGK08]). Most of these semantic optimization approaches build upon the chase algorithm [ASU79; MMS79; JK82; BV84], which has originally been proposed to tackle the implication problem for data dependencies [MMS79; BV84] and to optimize conjunctive queries under data dependencies [ASU79; JK82]. Since then, the chase has been successfully applied in many other application areas, such as data exchange [FKMP05], data integration [Len02], query answering using views [Hal01], and probabilistic databases [OHK09].

Given the central role of the chase and the observation that the algorithm does not necessarily terminate (cf. the discussion in Section 5.1.4), much research effort has been spent in finding sufficient conditions that guarantee its termination [FKMP05; DNR08; Mar09; SML08; MSL09d; MSL09a]. The common idea of all these termination conditions is to statically assert that there are no positions in the database schema where fresh labeled null values might be cyclically created in. We exemplarily presented weak acyclicity as a sufficient termination conditions in Section 5.1.4. In [DNR08], weak acyclicity was generalized to a condition called stratification. In [MSL09d; MSL09a] we introduced termination conditions that further generalize stratification, namely safe restriction, inductive restriction, and the so-called T-hierarchy of constraint classes.[4] Like weak acyclicity and stratification, these sufficient conditions guarantee chase termination on every database instance. Further, in these publications we studied the problem of data-dependent chase termination and, as a key result, developed specific termination conditions for the chase algorithm w.r.t. fixed database instances. We will not go into more detail here and refer the interested reader to the original publications for details.

SQO for SPARQL. By the best of our knowledge, constraint-based optimization in the context of RDF has first been proposed in [SKCT05]; we refer the reader back to Section 5.4 for a short discussion of that work. In addition, we discussed constraints for RDF and SQO for SPARQL in [LMS08; SML08]. The results presented in this chapter extend the ideas of the two previous publications.

[4] There is a minor bug in the stratification condition from [DNR08], which also carries over to some of the conditions presented in [MSL09d; MSL09a]. We propose a fix for this bug in [MSL09b].

5.6. Conclusion

In this chapter we discussed different aspects of constraints in the context of RDF and RDFS data. In particular, we showed that such constraints typically fall into well-known constraints classes that have been extensively studied in the context of other data formats and that SPARQL can easily be extended to check and specify constraints, similar in style to the SQL query language, which offers mechanisms to test and define constraints for the relational model. Furthermore, we presented a semantic query optimization approach for SPARQL. Falling back on established algorithms like the chase and the C&B algorithm for conjunctive query optimization, our scheme gives us guarantees to find minimal conjunctive queries if the chase terminates. Tackling the issue of more complex SPARQL queries, we presented semantic optimization rules for queries involving operators Filter and Opt.

We conclude with some final remarks on our SQO scheme for SPARQL. First, we want to note that semantic optimization strategies are basically orthogonal to algebraic optimization schemes. Hence, the semantic optimization approach presented in this chapter can be coupled with the algebraic rewriting rules from Chapter 4. To give an example, we may get better optimization results when combining the rules for filter decomposition, elimination, and pushing from Figure 4.4 with the semantic rewriting rules for filter expressions stated in Lemma 5.4. As another example, in Example 5.15 we demonstrated that the projection pushing rules from Figure 4.3 can be used to increase the benefit of our semantic optimization scheme.

A second important property of our SQO approach is that, as proposed in [DPT06], the C&B algorithm can be enhanced by a cost function, which makes it easy to factor in cost-based query optimization approaches for SPARQL such as [SSB+08]. This flexibility strengthens the prospectives and practicability of our semantic optimization scheme. The study of rewriting heuristics and the integration of a cost function, though, is beyond the scope of this thesis and is left as future work.

Chapter 6.

SP^{2}Bench: A SPARQL Performance Benchmark

Roy: "*We're finally done with optimization. Now others can use our algebraic and semantic optimization schemes to develop new SPARQL optimization approaches and build efficient engines!*"
Jen: "*But how do they prove they're better than others?*"
Moss: "*And how do they show they're efficient at all?*"
Roy: "*Right, we need a comprehensive benchmark for SPARQL!*"

As discussed in Section 4.5, over the last years a variety of proposals for the efficient SPARQL evaluation have been made [ACKP01; BKvH02; HG03; Cyg05; HD05; SKCT05; TCK05; CLJF06; PAG06a; AMMH07; GGL07; HH07; Pol07; AG08a; FB08; LMS08; NW08; SSB+08; WKB08; NW09]. As a proof of concept, most of these approaches have been evaluated experimentally either in user-defined scenarios, on top of the LUBM benchmark [GPH05], or on top of the Barton Library benchmark [AMMH]. We argue that none of these scenarios is adequate for testing SPARQL implementations in a general and comprehensive way.

1. User-defined scenarios are typically designed to demonstrate very specific properties of engines and, for this reason, often lack generality. To assess the performance of engines in an independent way, though, it is desirable to have a comprehensive benchmark platform that covers a broad range of challenges that may arise in the context of SPARQL query evaluation.
2. The LUBM benchmark was primarily designed to test the reasoning and inference mechanisms of Knowledge Base Systems. In this line, the main challenge of the LUBM queries lies in the efficiency of the inferencing process, which is not even part of the SPARQL evaluation process, where it is assumed that inferencing is realized by a separate layer (see the discussion in Section 5.4.1). Coming along with this design decision, in the LUBM benchmark queries central SPARQL operators like Union and Opt are missing. We argue that, to test the performance of SPARQL evaluation approaches, one should rather consider SPARQL evaluation on top of a fixed data set, i.e. without inference or where the inferred triples have already been materialized.

3. The Barton Library benchmark is strongly application-oriented. Its queries, which have been derived from a typical browsing session through the Barton Library online catalog, are specified in SQL and it is assumed that RDF is stored in a relational database. One problem is that not all of these queries can be expressed in SPARQL, due to missing language support for aggregation. Beyond that, in the corresponding translations of those queries that can be expressed, central SPARQL operators like OPT are missing. Further, a closer investigation of the Barton queries reveals that the join patterns in these queries are rather uniform (cf. the discussion in [SHK+08]). A comprehensive benchmark should cover all SPARQL operators, important operator constellations, as well as a variety of data access and join patterns.

Based on the observations above, there is an urgent need for a benchmark framework that allows to assess the performance of SPARQL engines and, more generally, RDF storage schemes in an extensive and uniform way. In response, we have developed a benchmark suite for the SPARQL query language that fulfills these requirements, called **SP**ARQL **P**erformance **Bench**mark (SP²Bench).[1]

The focus and design goals of SP²Bench vary from the benchmark projects discussed before. In particular, SP²Bench differs in that it is neither application-oriented nor use-case driven, but falls into the class of *language-specific* benchmarks. This means that, compared to the other benchmarks, the document and query design in SP²Bench is not driven by a specific use-case, but instead is specifically laid out to test common SPARQL constructs, operator constellations, and a variety of RDF data access patterns. In this line, SP²Bench covers a broad range of challenges that SPARQL engines may face in different contexts and constitutes a framework that allows for comprehensive performance evaluation, rather than performance assessment in a specific, application-driven scenario. The SP²Bench queries are not intended to be evaluated in a work load setting, but rather on a one by one basis, where each query poses different challenges to the tested SPARQL engine and may help to identify deficiencies in the underlying evaluation strategy. With these design decisions, SP²Bench allows to assess the generality of optimization approaches and to compare evaluation strategies in a universal, application-independent setting. We want to emphasize that such language-specific benchmarks (such as, for instance, the XQuery benchmark XMark [SWK+02]) have found broad acceptance in the past.

The first component of SP²Bench is a data generator, which supports the creation of DBLP-style models in RDF format. DBLP [Ley] is a well-known bibliographic library that contains publications made in the area of databases and, more generally, computer science. The generated DBLP-like documents mirror vital key characteristics and distributions found in the original DBLP database, making it possible to create arbitrarily large documents with realistic data that exhibits many

[1] The SP²Bench data generator and benchmark queries can be downloaded in a ready-to-use format at `http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B`.

real-world characteristics. The data mimics natural correlations between entities, such as power law distributions and limited growth curves. Complementarily to the generator, SP^{2}Bench comprises 17 meaningful queries specified over the generated documents. These queries cover important SPARQL constructs, operator constellations, and vary in their characteristics, such as complexity and result size. The detailed knowledge of data characteristics, which is obtained by an elaborate analysis of the original DBLP database, makes it possible to predict the challenges that the queries impose on SPARQL engines. This contributes to the understanding of the SP^{2}Bench queries and facilitates the interpretation of benchmark results.

The key contributions of this chapter are the following.

- We present SP^{2}Bench, a comprehensive benchmark for SPARQL. Our framework comprises a data generator and a collection of 17 benchmark queries.
- The benchmark data generator supports the creation of arbitrarily large documents in RDF format, reflecting key characteristics and social-world distributions found in the DBLP database, a well-known bibliographic library. In addition, the generated documents cover a variety of RDF constructs, such as blank nodes and RDF containers. While, in the context of SP^{2}Bench, this data forms the basis for the design of challenging and predictable benchmark queries, the data generator may also be useful in other Semantic Web projects, i.e. whenever large amounts of test data with natural distributions are needed.
- The benchmark queries have been carefully designed to test common operator constellations, data access patterns, and SPARQL optimization strategies. They comprise both SPARQL Select and Ask queries. In the exhaustive discussion of these queries we also highlight the specific challenges that these queries impose on SPARQL engines and discuss possible evaluation strategies.
- We finally propose a set of performance metrics that capture different aspects of the evaluation process, such as runtime and memory consumption. These metrics allow for a schematic analysis and interpretation of SP^{2}Bench benchmark results, as well as a comparison of results from different engines.

Structure. We start with a discussion of general and SPARQL-specific desiderata for benchmarks in Section 6.1, including design decisions made in the SP^{2}Bench framework. Subsequently, in Sections 6.2 and 6.3 we turn towards a study of the SP^{2}Bench data generator. This discussion includes an analysis of key characteristics of the DBLP data set, which forms the basis for the implementation of the data generator. The profound knowledge of DBLP characteristics helps to understand the key challenges of the SP^{2}Bench queries, which are presented in Section 6.4. The chapter ends with a discussion of performance metrics for the SP^{2}Bench suite (Section 6.5), a summary of related work (Section 6.6) and a short conclusion (Section 6.7).

6.1. Benchmark Design Decisions

A central aspect in the design of a benchmark is the choice of an appropriate domain. Clearly, the domain of a language-specific benchmark should not only constitute a representative scenario that captures the philosophy behind the data format, but also leave room for challenging benchmark queries. With the choice of the DBLP library [Ley], a bibliographic database that contains a large collection of publications made in the area of computer science, SP²Bench satisfies both desiderata. First, the RDF data format has been particularly designed to encode meta data (cf. [rdfb]), which makes DBLP an excellent candidate for an RDF scenario. Further, as shown in [EL05], DBLP reflects interesting social-world distributions. One may expect that such distributions are frequently found in the Semantic Web, which integrates a great many of individual databases into one global database and therefore can be seen as a large social network. As an example, it has been shown in [TTKC08] that power-law distributions are naturally contained in large RDF Schema specifications. These observations justify the choice of DBLP as the underlying scenario and facilitate the design of interesting queries on top of these natural social-world distributions.

In the context of semi-structured data, one often distinguishes between data- and document-centric scenarios. Document-centric design typically involves large amounts of free-form text, while data-centric documents are more structured and usually processed by machines rather than humans. As discussed in Section 2.2, one major design goal of RDF was to provide a mechanism to encode information in a machine-readable way, so the RDF data format basically follows the data-centric approach. We point out that also in this respect, the DBLP domain – which provides structured data and only little free text – constitutes an adequate RDF scenario.

Rather than using an RDF version of the existing DBLP data set (such as [BC07]), SP²Bench comes with a generator that supports the creation of arbitrarily large DBLP-like documents in RDF format, hence overcoming an upper limit on the size of benchmark documents.[2] The generator itself relies on an in-depth study of characteristics and relationships between entities found in the original DBLP database, comprising the analysis of data entities (such as articles and authors), their properties, frequency, and also their interaction. Consequently, the generated documents mimic a broad range of natural, social-world distributions such as power laws (found in the citation system or the distribution of publications among authors) and limited growth curves (e.g., the increasing number of publications over time).

Complementarily to the data generator, we have designed 17 meaningful benchmark queries that operate on top of the generated documents. As we will show later, these queries cover not only the most important SPARQL constructs and operator constellations, but also vary in other characteristics, such as complexity, result size,

[2] In September 2009, the XML-to-RDF translation of the original DBLP database in [BC07] comprised only about 15 million RDF triples, which is far away from a large-scale RDF scenario.

and join patterns. We want to stress that the detailed knowledge of data characteristics is crucial to design meaningful, challenging queries and makes it possible to predict the challenges that the benchmark queries impose on SPARQL engines. This, in turn, facilitates the interpretation of the benchmark results.

Benchmark Design Principles. The Benchmark Handbook [Gra93] postulates four key requirements for domain specific benchmarks. First, such a benchmark should be (1) *relevant*, thus testing representative and typical operations within the specific domain. Second, benchmarks should be (2) *portable*, i.e. should be executable on different platforms. Third, each benchmark should be (3) *scalable*, which in particular means that it should be possible to run the benchmark on both small and very large data sets. Last but not least, every benchmark must be (4) *understandable*, since otherwise it will with high probability not be adopted in practice.

For a language-specific benchmark, the relevance requirement (1) suggests that queries implement realistic requests on top of the data. Moreover, the benchmark queries should not focus on verifying the correctness of the tested engine, but rather on common operator constellations that impose representative, particular challenges. To give an example for the manifestation of these ideas in SP^{2}Bench, two of our queries test (closed-world) negation, which – as discussed previously in Section 4.2.6 – can be expressed in SPARQL through a characteristic combination of operators Opt, Filter, and filter predicate *bnd* (see queries $Q6$ and $Q7$ in Section 6.4).

Requirements (2) portability and (3) scalability bring along technical challenges concerning the implementation of the data generator. Addressing those, our data generator is deterministic, platform independent, and accurate w.r.t. the desired size of generated documents. It is efficient and gets by with a constant amount of main memory, and hence supports the generation of arbitrarily large RDF documents.

From the viewpoint of an engine developer, a benchmark should give hints on deficiencies in the design and implementation of the respective engine. This is where requirement (4) understandability comes into play, i.e. it is important to keep queries simple and understandable. At the same time, the queries should leave room for diverse optimizations. In this regard, we took special care to design them in such a way that they are amenable to a wide range of optimization strategies.

6.2. The DBLP Data Set

We start our presentation of SP^{2}Bench with a study of the DBLP data set, which lays the foundations for the design and implementation of the generator. To this end, we analyze properties, relations, and distributions of bibliographic entities and persons found in the original DBLP data set. Such analysis of frequency distributions in scientific production is not new, but has first been discussed in [Lot26]. Similar in spirit, a study of characteristics of the DBLP database has been presented in [EL05]. The latter work considers a subset of DBLP, called DBLP-DB, restricting the dis-

cussion to publications in database venues. As a key contribution, it is shown that DBLP-DB, reflects vital relations that are typically encountered in social networks, thus forming a "small world" on its own. We relax the restriction from [EL05] and consider the whole DBLP database (instead of DBLP-DB). Our analysis shows that similar social-world relations can also be found in the complete database. We note that, although the analysis in [EL05] forms valuable groundwork for the study presented in this section, our approach differs in that it is of more pragmatic nature: the ultimate goal of our investigation is to approximate real-world distributions by concrete functions that can be used for the implementation of a data generator.

In order to approximate distributions, we use function families that naturally reflect the scenarios, such as logistics curves for modeling limited growth or power equations for power law distributions. All approximations presented in this section have been computed with the *ZunZun* [zun] data modeling tool and the *gnuplot* [gnu] curve fitting module. Our starting point for the study of DBLP was the February 25, 2008 XML version of the DBLP data. Data extraction from the DBLP XML database (such as, for instance, the extraction of authors or different types of publications) was realized using the MonetDB/XQuery [CWI] processor.

Apart from the pure extraction and approximation of distributions found in the DBLP data set, an important objective of this section is to provide insights into key characteristics of DBLP data. Although it is impossible to mirror all relations found in the original data, we work out a variety of interesting relationships, considering different types of publications, the role of persons as authors and editors, as well as the citation system. In this line, we mainly study the development of such entities over time, on a year-by-year basis. The insights that we gain establish a deep understanding of the benchmark queries and their specific challenges. As an example, the benchmark queries $Q3a$, $Q3b$, and $Q3c$ (discussed in Section 6.4) look quite similar, but pose different challenges based on the probability distribution of article properties (i.e. `swrc:pages`, `swrc:month`, `swrc:isbn`), which will be discussed within this section. As another example, $Q7$ (also given in Section 6.4) heavily depends on the DBLP citation system, which will be another central topic in this section.

In the end, our analysis allows us to implement a data generator that creates data that is very similar to the original DBLP data, at least for years up to present. Of course we cannot guarantee that the generated data goes hand in hand with the original DBLP data when generating data for future years. However, and this is even more important, by implementing reasonable social-world distributions, also data for the future reflects real-world relationships between data entities, which is clearly preferable to purely synthetic data. We finally stress that the benchmark queries are designed to operate on top of exactly those relations and distributions that are mirrored by our data generator, which makes them realistic, predictable and understandable. For instance, some queries cover the citation system, which is mirrored by our generator. As a counterexample, the generator abstracts from a realistic distribution of article release months, so no query relies on this property.

```
<!ELEMENT dblp
  (article|inproceedings|proceedings|book|
   incollection|phdthesis|mastersthesis|www)*>
<!ENTITY % field
  "author|editor|title|booktitle|pages|year|address|
   journal|volume|number|month|url|ee|cdrom|cite|
   publisher|note|crossref|isbn|series|school|chapter">
<!ELEMENT article (%field;)*>
<!ELEMENT inproceedings (%field;)*>
...
<!ELEMENT www (%field;)*>
```

Figure 6.1.: Extract of the DBLP DTD.

	Article	Inproc.	Proc.	Book	WWW	PhDTh.	MastTh.	Incoll
author	0.9895	0.9970	0.0001	0.8937	0.9973	1.0000	1.0000	0.8459
cite	0.0048	0.0104	0.0001	0.0079	0.0000	0.0000	0.0000	0.0047
editor	0.0000	0.0000	0.7992	0.1040	0.0004	0.0000	0.0000	0.0000
isbn	0.0000	0.0000	0.8592	0.9294	0.0000	0.0222	0.0000	0.0073
journal	0.9994	0.0000	0.0004	0.0000	0.0000	0.0000	0.0000	0.0000
month	0.0065	0.0000	0.0001	0.0008	0.0000	0.0333	0.0000	0.0000
pages	0.9261	0.9489	0.0000	0.0000	0.0000	0.0000	0.0000	0.6849
title	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Table 6.1.: Probability distribution for selected attributes.

6.2.1. Structure of Document Classes

Our starting point for the discussion of the DBLP data set is the DBLP DTD, which comes with the public release of the DBLP XML data set.[3] An extract of the DTD is provided in Figure 6.1. The DTD entry point `dblp` defines eight child entities, namely Article, Inproceedings, Proceedings, ..., and WWW resources. In the following, we will call these entities *document classes*, and instances thereof *documents*. Furthermore, we distinguish between Proceedings documents, called *conferences*, and instances of the remaining classes, called *publications*.

The DTD defines 22 possible child tags for each document class, such as `author`, `editor`, and `title`. These tags *describe* documents and we call them *attributes* in the following. As can be seen, according to the DTD each document might be described by an arbitrary combination of attributes and even repeated occurrences of the same attribute are allowed. For instance, a document may have one `title` attribute specifying its title and, in addition, several `author` attributes. As one may expect, in practice only a subset of all document class/attribute combinations occur.

[3] See `http://www.informatik.uni-trier.de/~ley/db/`.

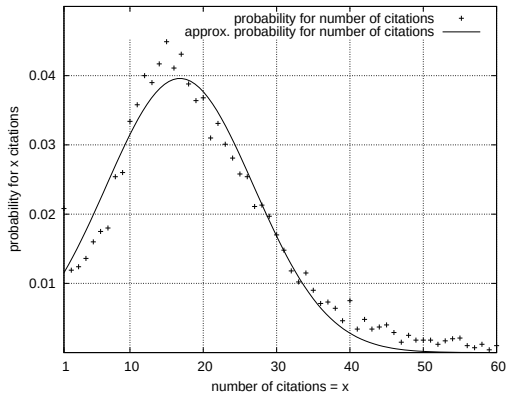

Figure 6.2.: Distribution of citations for documents having at least one citation.

For instance, it turns out that attribute `pages` is never associated with WWW documents, but instead is typically used in combination with Article documents.

We summarize these connections between document classes and attributes in Table 6.1. It shows, for selected document class/attribute pairs, the probability that the attribute describes a document of this class.[4] To give an example, about 92.61% of Article documents are described by the attribute `pages`, but none of them has an `editor` associated. In contrast, about 79.92% of all Proceedings documents are described by one or more `editor` attributes. It is important to note that this analysis does not account for multiple occurrences of an attribute in a single document. We will discuss such repeated occurrences of attributes separately in Section 6.2.2.

The probability distribution in Table 6.1 forms the basis for generating document class instances of any type. Note that we simplify and assume that the presence of an attribute does not depend on the presence of other attributes, i.e. we ignore conditional probabilities. We will further elaborate on this decision in Section 6.7.

6.2.2. Repeated Attributes

We now turn towards an investigation of attributes that occur repeatedly within a single document, called *repeated attributes*. A closer investigation of DBLP reveals that in practice only few attributes occur repeatedly and, for the majority of them, the number of repeated occurrences is very small. We decided to restrict ourselves to an analysis of the most frequent repeated attributes, i.e. `cite`, `editor`, and `author`.

Figure 6.2 exemplifies our analysis for attribute `cite`. It shows, for each document with at least one `cite` occurrence, the probability (y-axis) that the document has exactly n `cite` attributes (x-axis). Recall that, according to Table 6.1, only a small fraction of documents are described by `cite` (e.g., only 0.48% of all Article documents). Arguably, this value should be close to 100% in a complete scenario,

[4]The full correlation matrix can be found in the technical report [SHLP08].

which shows that DBLP contains only a fraction of all existing citations. This is also why, in our analysis (and hence in Figure 6.2), we consider only those documents with at least one outgoing citation, since otherwise over 99% would get assigned zero citations. Still, when assigning the number of citations to documents of a specific class later on, we first use the probability distribution of attributes in Table 6.1 to estimate the number of documents with at least one outgoing citation and afterwards apply the citation distribution in Figure 6.2 to only those documents that – according to the previous step – get assigned citations. We emphasize that this strategy exactly mirrors the distribution found in the original DBLP data.

Based on experiments with different function families, we decided to use bell-shaped Gaussian curves for approximating the distribution of citations from Figure 6.2.[5] It is well-known that Gaussian curves are typically used to model normal distributions. Although, strictly speaking, our data is not normally distributed (due to the left limit at position $x = 1$), these curves nicely fit the data for $x \geq 1$ (see Figure 6.2). Formally, Gaussian curves are described by functions

$$p_{gauss}^{(\mu,\sigma)}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-0.5(\frac{x-\mu}{\sigma})^2}, \tag{6.1}$$

where the *expected value* $\mu \in \mathbb{R}$ fixes the x-position of the peak and $\sigma \in \mathbb{R}_{>0}$ specifies the statistical spread. For instance, the approximation function for the `cite` distribution in Figure 6.2 is defined by instantiating Equation (6.1):

$$d_{cite}(x) := p_{gauss}^{(16.82,10.07)}(x). \tag{6.2}$$

The analysis and the resulting distribution of repeated `editor` attributes is structurally similar to that of the `citation` attribute, so we omit the details; the concrete approximation function for `editor` attributes is the Gaussian curve

$$d_{editor}(x) := p_{gauss}^{(2.15,1.18)}(x). \tag{6.3}$$

The approximation function for repeated `author` attributes is based on a Gaussian curve, too. However, we observed that the average number of authors per publication has increased over the years. The same observation was made in [EL05] and explained by an increasing pressure to publish and the proliferation of new communication platforms, like the Internet. Due to the prominent role of authors, we decided to mimic this property. As a consequence, parameters μ and σ are not fixed for the `author` attribute (as it was the case for the distributions $d_{cite}(x)$ and $d_{editor}(x)$), but modeled as functions over time. We make the assumption that the average number of authors per publication will eventually stabilize and model both μ and σ as limited growth functions (also called logistic curves). We consider logistic curves of the form

[5] We experimented with other distributions, such as Gamma distributions, log normal distributions, and Chi distributions (cf. [McL99]), but obtained the best fitting for Gaussian curves.

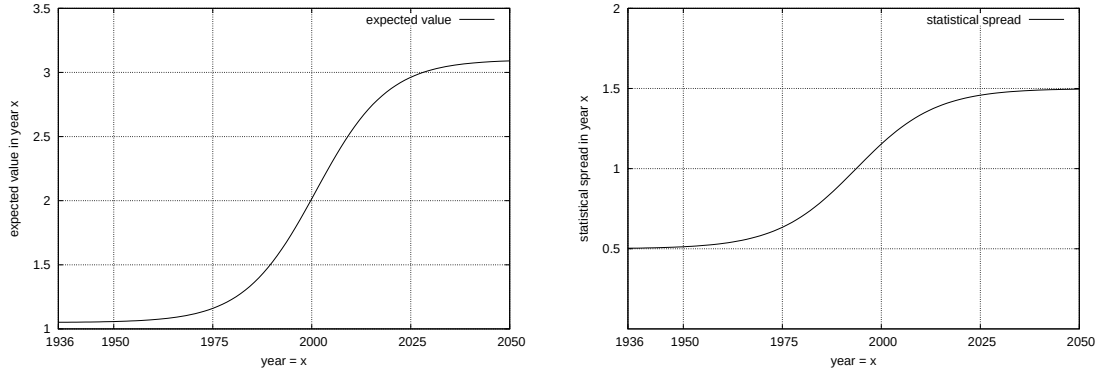

Figure 6.3.: Development of author (a) expected value ($\mu_{auth}(yr)$) and (b) statistical spread ($\sigma_{auth}(yr)$) for publications having at least one author.

$$f_{logistic}(x) = \frac{a}{1 + be^{-cx}}, \tag{6.4}$$

where $a, b, c \in \mathbb{R}_{>0}$. For this parameter setting, a constitutes the upper asymptote and the x-axis forms the lower asymptote. The curve is "caught" in-between its asymptotes and increases continuously, i.e. it is S-shaped (two examples for logistic curves are plotted in Figure 6.3; we will discuss them in the following).

The resulting distribution $d_{auth}(x, yr)$ for the `author` attribute, which gives the probability for a publication in year yr having exactly x authors given that the publication has at least one author, is then described by the Gaussian function

$$\begin{aligned} d_{auth}(x, yr) &:= p_{gauss}^{(\mu_{auth}(yr), \sigma_{auth}(yr))}(x), \text{where} \\ \mu_{auth}(yr) &:= \frac{2.05}{1 + 17.59e^{-0.11(yr-1975)}} + 1.05, \text{and} \\ \sigma_{auth}(yr) &:= \frac{1.00}{1 + 6.46e^{-0.10(yr-1975)}} + 0.50. \end{aligned} \tag{6.5}$$

Figures 6.3 (a) and (b) show the logistic curves for functions $\mu_{auth}(yr)$ and $\sigma_{auth}(yr)$. We can observe that both $\mu_{auth}(yr)$ and $\sigma_{auth}(yr)$ increase steadily over time, but stabilize in future years. The concrete limited growth approximation was obtained by fitting original DBLP data up to year 2005. The distribution of the `author` attribute (i.e., function $d_{auth}(x, yr)$) builds on $\mu_{auth}(yr)$ and $\sigma_{auth}(yr)$. We plot this distribution for selected years (1980, 2000, and the expected distribution for years 2020, 2040) in Figure 6.4(a). According to the growth of the expected value and statistical spread, we can observe that the average number of authors increases over time. Due to the fact that the growth is limited, this increase is significant for early years (cf. year 1980 vs. 2000), but diminishes in future years (cf. year 2020 vs. 2040).

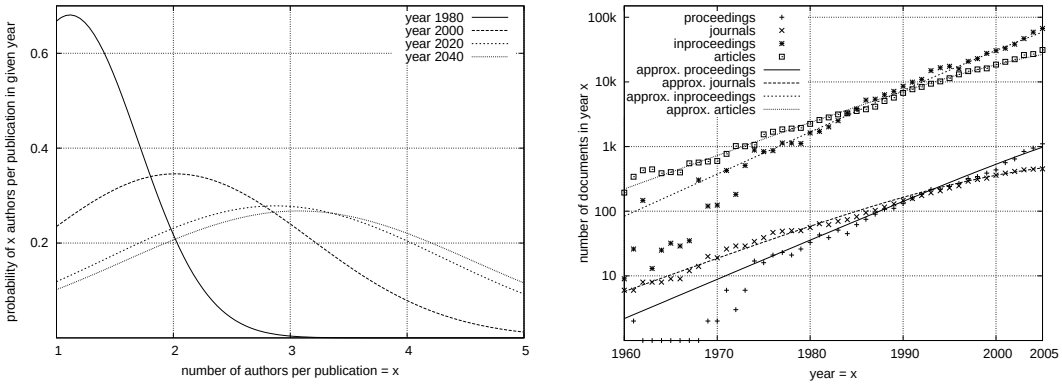

Figure 6.4.: (a) Expected number of authors for publications having at least one author ($d_{auth}(x, yr)$); (b) Development of documents over time.

6.2.3. Development of Document Classes over Time

We next investigate the quantity of document class instances over time. We noticed that DBLP contains only little and incomplete information in its early years, and also found anomalies in the final years, mostly in form of lowered growth rates. We suspect that, in the coming years, some more conferences for the last years may be added belatedly (i.e. data may not yet be complete) and therefore we restrict ourselves to the years in-between 1960 and 2005 in the subsequent discussion

Figure 6.4(b) plots the number of PROCEEDINGS, JOURNAL, INPROCEEDINGS, and ARTICLE documents as a function of time. The y-axis is in log scale. Note that JOURNAL is not an explicit document class according to the DBLP DTD in Figure 6.1, but we assume that this class is implicitly defined by the `journal` attribute of ARTICLE documents, which specifies the journal an article appeared in. As can be seen in Figure 6.4(b), the inproceedings and articles are closely coupled to the proceedings and journals, respectively. For instance, there are always about 50-60 times more inproceedings than proceedings. This indicates that the average number of inproceedings per proceeding remains stable over time. Similar observations hold with respect to the article documents and the journals they appeared in.

As a further observation, Figure 6.4(b) shows exponential growth for all document classes under consideration, where the growth rate of JOURNAL and ARTICLE documents decreases in the final years. This observation strongly suggests a limited growth scenario, to be modeled by the logistic curve formula introduced in Equation (6.4). For instance, we approximate the number of JOURNAL documents over time, which is also visualized in Figure 6.4(b), by the logistic curve

$$f_{journal}(yr) := \frac{740.43}{1 + 426.28e^{-0.12(yr-1950)}}. \tag{6.6}$$

Approximation functions for ARTICLE, PROCEEDINGS, INPROCEEDINGS, BOOK,

$$
\begin{array}{lll}
f_{article}(yr) & := & \frac{58519.12}{1+876.80e^{-0.12(yr-1950)}} \\
f_{proc}(yr) & := & \frac{5502.31}{1+1250.26e^{-0.14(yr-1965)}} \\
f_{inproc}(yr) & := & \frac{337132.34}{1+1901.05e^{-0.15(yr-1965)}} \\
f_{incoll}(yr) & := & \frac{3577.31}{196.49e^{-0.09(yr-1980)}}
\end{array}
\qquad
\begin{array}{lll}
f_{book}(yr) & := & \frac{52.97}{40739.38e^{-0.32(yr-1950)}} \\
f_{phd}(yr) & := & random[0..20] \\
f_{masters}(yr) & := & random[0..10] \\
f_{www}(yr) & := & random[0..10]
\end{array}
$$

Figure 6.5.: Approximation functions for document class counts.

and Incollection documents are also modeled as logistic curves and differ only in the parameters. Concerning the remaining document classes PhDThesis, MastersThesis, and WWW we found that documents thereof were distributed unsteadily, so we modeled them by random functions *random*[x..y] (which generate random numbers in the interval [x..y]) that reflect their distribution in the original DBLP data set. The concrete approximation functions for all these document classes are summarized in Figure 6.5, where we name each formula according to the document class, e.g. $f_{article}(yr)$ denotes the function that belongs to document class Article. It is worth mentioning that the number of articles and inproceedings per year clearly dominates the number of instances of the remaining classes.

Based on the previous analysis, we can easily estimate the total number of documents $f_{docs}(yr)$ in year yr by summing up the individual counts:

$$
\begin{aligned}
f_{docs}(yr) := f_{journal}(yr) + f_{article}(yr) + f_{proc}(yr) + f_{inproc}(yr) + \\
f_{incoll} + f_{book}(yr) + f_{phd}(yr) + f_{masters}(yr) + f_{www}(yr).
\end{aligned}
\tag{6.7}
$$

6.2.4. Authors and Editors

Authors. An estimation for the *total number of authors* in a given year, which we define as the number of `author` attributes associated to the year, is obtained as follows. First, we estimate the number of documents described by attribute `author` for each document class individually (using the distribution in Table 6.1). All these counts are summed up, which gives us an estimation for the total number of documents with one or more `author` attributes. Finally, this value is multiplied with the expected average number of authors per paper in the respective year (which is implicitly given by the distribution $d_{auth}(x, yr)$ discussed Section 6.2.2).

To be close to reality, we also consider the number of distinct persons that appear as authors (relative to the year), called *distinct authors*, as well as the number of *new authors* in a given year, i.e. those persons that publish for the first time. We found that the number of distinct authors $f_{dauth}(yr)$ per year can be approximated in dependence of $f_{auth}(yr)$ according to the formula

$$
f_{dauth}(yr) := (\frac{-0.67}{1 + 169.41e^{-0.07(yr-1936)}} + 0.84) * f_{auth}(yr). \tag{6.8}
$$

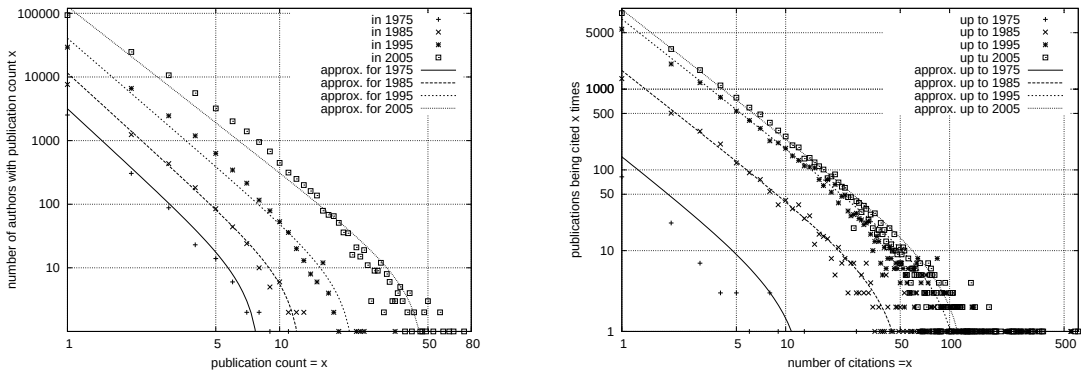

Figure 6.6.: Powerlaw distributions in DBLP: (a) Number of publications per author over time; (b) Distribution of citations among publications.

The distribution indicates that the number of distinct authors relative to the total authors decreases steadily, from 0.84% to 0.17% (= 0.84% − 0.67%). This reflects the increasing productivity of authors over time discussed in Section 6.2.2.

The approximation formula for the number $f_{new}(yr)$ of new authors builds on the previous one and is also based on the logistic curve function in Equation (6.4):

$$f_{new}(yr) := \left(\frac{-0.29}{1749.00e^{-0.14(yr-1937)}} + 0.628\right) * f_{dauth}(yr). \tag{6.9}$$

Publications. In Figure 6.6(a) we plot, for selected year and publication count x, the number of authors with exactly x publications in this year. The graph is in log-log scale. Here, we observe a typical power law distribution, which is characterized by the fact that there are only a couple of authors having a large number of publications, while a majority of the authors has only few publications.

Power law distributions are modeled by functions of the form

$$f_{powerlaw}(x) = ax^k + b, \tag{6.10}$$

with constants $a \in \mathbb{R}_{>0}$, $b \in \mathbb{R}$, and exponent $k \in \mathbb{R}_{<0}$. In such functions, parameter a affects the x-axis intercept, exponent k defines the gradient, and parameter b constitutes a shift in y-direction. For the given parameter restriction, the powerlaw functions decrease steadily for increasing $x \geq 0$. Examples for powerlaw functions are the approximation functions shown in Figure 6.6(a) and 6.6(b).

Another interesting trend that can be observed in Figure 6.6(a) is that the curves move upwards throughout the years. Informally speaking, this means that the publication count of the leading author(s) has steadily increased over the last 30 years and it also reflects an increasing number of authors over time. We estimate the number of authors with x publications in year yr as

$$f_{awp}(x, yr) := 1.50 f_{publ}(yr) x^{-f'_{awp}(yr)} - 5, \text{where}$$
$$f'_{awp}(yr) := \frac{-0.60}{1 + 216223 e^{-0.20(yr-1936)}} + 3.08. \tag{6.11}$$

In the formula above, $f_{publ}(yr)$ returns the total number of publications in yr.[6] Note that the logistic curve $f_{awp'}(yr)$ reflects the increasing number of publications of the leading author and is used in $f_{awp}(yr)$. Parameters $k := f'_{awp}(yr)$ and b in function $f_{awp}(x, yr)$ were fitted experimentally, using the original DBLP data.

Coauthors. With respect to coauthor characteristics, we also investigated relations between the publication count of authors and its numbers of total and distinct coauthors in DBLP. In particular, given an author with a total of x publications, we approximate the average number of total coauthors by

$$\mu_{coauth}(x) := 2.12 * x \tag{6.12}$$

and the number of its distinct coauthors by

$$\mu_{dcoauth}(x) := x^{0.81}. \tag{6.13}$$

These functions are used later when distributing publications among authors.

Editors. The analysis of authors is complemented by a study of their relations to editors. As one may expect, editors tend to be persons that have published before, i.e. persons that are well-known in the community. Following the distributions in the original DBLP database, we thus assign editor activity primarily to persons that have published in (earlier) venues. We omit the concrete analysis and formula, which is rather technical and does not bring further insights into this relationship.

6.2.5. Citations

In Section 6.2.1 we studied repeated occurrences of attribute `cite`, namely outgoing citations. This analysis is complemented by a study of incoming references, i.e. the distribution of incoming citations among publications. Figure 6.6(b) plots the citation count distribution in DBLP, up to selected years. For instance, up to 1995 there were about 500 papers that have been cited 5 times. Similar to the distribution of publications, we observe a typical powerlaw distribution: most of the papers have only a small fraction of citations, while only few papers have lots of citations. As for the distribution of publications among authors, we use powerlaw functions (cf. Equation (6.10)) to model this distribution. We omit the concrete formula.

As another observation, we found that in DBLP the number of incoming citations is smaller than the number of outgoing citations. This may be surprising at first

[6] It is obtained by subtracting the venue-type documents from $f_{docs}(yr)$ in Equation (6.7), i.e. it is defined as $f_{publ}(yr) := f_{docs}(yr) - (f_{journal}(yr) + f_{proc}(yr))$.

glance, but is simply explained by the fact that DBLP contains many untargeted citations (in form of empty `cite` tags). Recalling that only a fraction of all papers have outgoing citations (cf. Section 6.2.1), we conclude that the DBLP citation system is very incomplete, although in some sense natural in that it follows natural distributions such as power law distributions (w.r.t. incoming citations, see Figure 6.6(b)) and the Gaussian distribution (w.r.t. outgoing references, see Figure 6.2).

6.3. The SP²Bench Data Generator

Having studied the key characteristics of the DBLP database, we now give some background on the data generator implementation, including a discussion of the RDF scheme for DBLP, design decisions, and implementation details.

6.3.1. An RDF Scheme for DBLP

The first task in data generator implementation is to find an appropriate RDF scheme for the DBLP domain. We decided to follow the approach taken in [BC07], an existing XML-to-RDF mapping of the original DBLP database. However, with the goal to generate arbitrarily-sized documents, the SP²Bench data generator uses lists of first and last names, publishers, and random words, rather than real author, publication, and conference names. The generated SP²Bench conference and journal names are always strings of the form *"Conference $i ($year)"* and *"Journal $i ($year)"*, where i is a unique conference (respectively journal) number in year *$year*. Further, lists of random words are used to generate string content, e.g. for titles. Analogously, fresh person names are built using the list of first and last names mentioned above.[7] Finally, domain-specific string-content, such as the pages specification or the ISBN number of documents, are filled with strings of a reasonable domain (for instance, we generate ISBN-like random strings for ISBN numbers).

Similar to [BC07], we use existing RDF vocabularies to describe resources in a uniform way. We borrow vocabulary from FOAF [foa] for describing persons, and from SWRC [swr] (namespace `swrc`) and Dublin Core [dub] (namespace `dc`) for describing scientific resources. Additionally, we introduce a fresh namespace `bench`, which defines DBLP-specific document classes, such as `bench:Book` and `bench:Article`.

In the sense of RDF, each XML attribute is mapped to an RDF property. For instance, the attribute `author` is mapped to the RDF property `dc:creator`, a predefined URI from the `dc` namespace for specifying the creator of bibliographic entities. Consequently, an `author` attribute in the XML database will be translated into a single triple with predicate `dc:creator`, which provides the author for the respective

[7] The name lists that we provide to the data generator are large enough to generate distinct authors for DBLP data containing billions of RDF triples. However, in case these lists are not large enough, the data generator will notify the user and request larger name lists as input.

Attribute	Mapped to property	Type
address	swrc:address	xsd:string
author	dc:creator	foaf:Person
booktitle	bench:booktitle	xsd:string
cdrom	bench:cdrom	xsd:string
chapter	swrc:chapter	xsd:integer
cite	dcterms:references	foaf:Document
crossref	dcterms:partOf	foaf:Document
editor	swrc:editor	foaf:Person
ee	rdfs:seeAlso	xsd:string
isbn	swrc:isbn	xsd:string
journal	swrc:journal	bench:Journal
month	swrc:month	xsd:integer
note	bench:note	xsd:string
number	swrc:number	xsd:integer
page	swrc:pages	xsd:string
publisher	dc:publisher	xsd:string
school	dc:publisher	xsd:string
series	swrc:series	xsd:integer
title	dc:title	xsd:string
url	foaf:homepage	xsd:string
volume	swrc:volume	xsd:integer
year	dcterms:issued	xsd:integer

Figure 6.7.: Translation of DBLP attributes into RDF properties.

document. Figure 6.7 surveys the translation of XML attributes to RDF properties. For each attribute, we also list its range restriction, i.e. the type of elements it refers to. For instance, property `dc:creator` always refers to a URI of type `foaf:Person`.

Unfortunately, the XML-to-RDF translation of DBLP from [BC07] neither contains blank nodes nor RDF containers, such as lists or bags. With the goal to build a comprehensive benchmark, though, we want to design queries on top of such RDF-specific constructs. For this reason, we decided to model persons in the data set as (unique) blank nodes *_:firstname_lastname*, instead of URIs (using the lists of first- and lastnames mentioned above). To have RDF containers present in the generated data, we model outgoing citations of documents using standard `rdf:Bag` containers. In addition to this modification, we enrich a small fraction of Article and Inproceedings documents with the new property `bench:abstract` (only about 1%, to keep the modification low), which constitutes comparably large strings.[8]

[8]We use a Gaussian distribution with $\mu = 150$ expected words and a statistical spread of $\sigma = 30$.

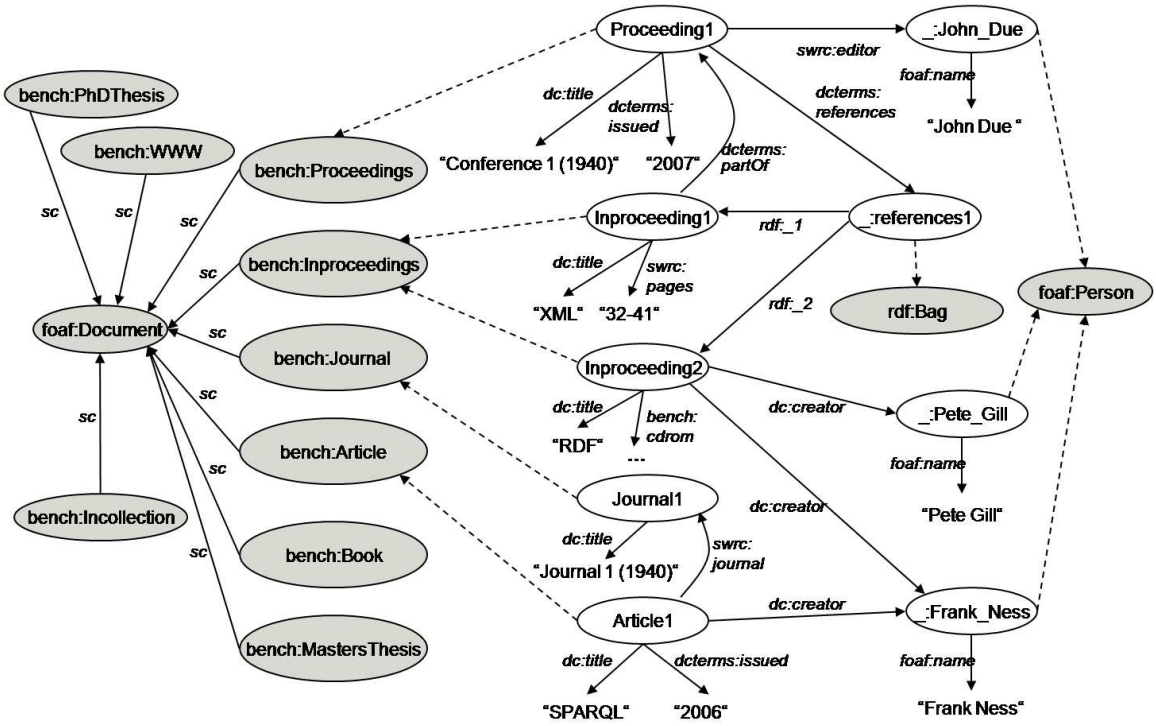

Figure 6.8.: Sample RDF database.

Figure 6.8 shows a sample DBLP instance in RDF format, as it may be generated by the SP²Bench data generator. Dashed edges are typing edges (i.e., `rdf:type`) and `sc` is an abbreviation for `rdfs:subClassOf`. On the logical level we distinguish between the *schema* layer (gray) and the *instance* layer (white). As discussed before, reference lists are modeled as blank nodes of type `rdf:Bag` (see e.g. *_:references1*), while both authors and editors are modeled as blank nodes of type `foaf:Person`. On the schema level, class `foaf:Document` splits up into the individual DBLP document classes `bench:Journal`, `bench:Article`, ... In summary, the sample graph defines three persons, one proceeding, two inproceedings, one journal, and one article. For readability reasons, we plot only selected properties of these entities. As also illustrated, property `dcterms:partOf` links inproceedings and proceedings together, while `swrc:journal` connects articles to the journals they appeared in.

In order to provide an entry point for queries that access authors and to provide a person with fixed characteristics, we created a special author, named after the famous mathematician Paul Erdös (this person is not shown in Figure 6.8). Per year, we assign 10 publications and 2 editor activities to this prominent person, starting from year 1940 and up to year 1996. For the ease of access, Paul Erdös is modeled as a fixed URI.[9] As an example query consider $Q8$ in Section 6.4, which extracts

[9]Recall that blank nodes cannot be accessed directly through SPARQL queries.

```
foreach year:
  calculate counts for documents and generate document class instances;
  calculate number of total, new, distinct, and retiring authors;

  choose publishing authors;
  assign number of new publications, number of coauthors, and
         number of distinct coauthors to publishing authors;
  // such that constraints for number of publications per author hold

  assign from publishing authors to papers;
  // satisfying authors per paper and coauthor constraints

  choose editors and assign editors to papers;
  // such that constraints for number of publications and editors hold

  generate outgoing citations;
  assign expected incoming and outgoing citations to papers;

  write output until done or until output limit reached;
  // permanently keeping output consistent
```

Figure 6.9.: Data generation algorithm.

all persons with *Erdös number* one or two (the Erdös number of a scientist is its distance to Paul Erdös in the coauthor graph: persons that have published directly with Paul Erdös have Erdös number one, those that have published with a coauthor of Paul Erdös have Erdös number two etc., see `http://www.oakland.edu/enp/`).

6.3.2. Data Generator Implementation

We implemented the scheme described in the previous section in a data generator, written in C++. It takes into account all relationships and characteristics that have been studied in Section 6.2. Figure 6.9 shows the key steps in data generation. The data generator simulates data year by year and takes into account all the structural constraints implied by the distributions studied within Section 6.2, in a carefully selected order. The generation process is simulation-based, which, amongst others, means that we assign life times to authors and individually estimate their future behavior, thereby taking into account global publications and coauthor constraints, as well as the fraction of distinct and new authors (cf. Section 6.2.4).

The generator writes RDF output in N-Triples format [ntr] (cf. Section 2.2.3). It offers two command line parameters, to fix either a triple count limit or the year up to which data will be generated. When the triple count limit is set, the implementation asserts that the generated data is in a "consistent" state, e.g. whenever proceedings

#Triples	10^3	10^4	10^5	10^6	10^7	10^8	10^9
Elapsed time [s]	0.08	0.13	0.60	5.76	70	1011	13306

Table 6.2.: Performance of document generation.

have been written to the output, the corresponding conference will also included.

All random functions (which, for example, are used to assign the attributes according to Table 6.1, or to sample data according to the approximation functions in Figure 6.2) are based on a fixed seed. This makes data generation deterministic, i.e. the parameter setting uniquely identifies the outcome. In addition, the data generator is implemented in plain ANSI C++, which asserts that it also is platform-independent. The fixed random seed and the platform-independent implementation ensure that documents generated on different platforms are identical, so experimental results from different platforms remain comparable (modulo hardware and system settings). Further, this asserts that data generation is incremental, which means that small documents are always contained in larger documents.

6.3.3. Data Generator Evaluation

In order to prove the practicability of our data generator implementation, we performed experiments and measured data generation times for documents of different size. The experiments were conducted under Linux ubuntu v7.10 gutsy, on top of an Intel Core2 Duo E6400 2.13GHz CPU and 3GB DDR2 667 MHz nonECC physical memory, using a 250GB Hitachi P7K500 SATA-II hard drive with $8MB$ Cache.

Table 6.2 summarizes the performance results for documents containing up to one billion RDF triples. We observe that the generator scales almost linearly with document size and creates even large documents very fast. For instance, the 10^9 triples document, which has a physical size of about $103GB$, was generated in less than four hours. An important prerequisite for this efficiency and scaling is that the generator runs with constant main memory consumption (it gets by with at most $1.2GB$ RAM, independent from the size of the generated documents).

In addition to the performance assessment, we verified the implementation of all characteristics from Section 6.2. Table 6.3 shows selected data generator and output document characteristics for documents up to $25M$ RDF triples. We list the size of the output file, the year up to which data was simulated, the number of total authors and distinct authors contained in the data set (cf. Section 6.2.4), and the counts of the document class instances (cf. Section 6.2.3). The summary shows superlinear growth for the number of authors relative to the number of triples in the data set, which is primarily caused by the increasing average number of authors per publication, as discussed in Section 6.2.1. The growth of proceedings and inproceedings is also superlinear, while the number of journals and articles increases

#Triples	10k	50k	250k	1M	5M	25M
File size [MB]	1.0	5.1	26	106	533	2694
Data up to year	1955	1967	1979	1989	2001	2015
#Total authors	1.5k	6.8k	34.5k	151.0k	898.0k	5.4M
#Distinct authors	0.9k	4.1k	20.0k	82.1k	429.6k	2.1M
#Journals	25	104	439	1.4k	4.6k	11.7k
#Articles	916	4.0k	17.1k	56.9k	207.8k	642.8k
#Proceedings	6	37	213	903	4.7k	24.4k
#Inproceedings	169	1.4k	9.2k	43.5k	255.2k	1.5M
#Incollections	18	56	173	442	1.4k	4.5k
#Books	0	0	39	356	973	1.7k
#PhD theses	0	0	0	101	237	365
#Masters theses	0	0	0	50	95	169
#WWWs	0	0	0	35	92	168

Table 6.3.: Characteristics of generated documents.

sublinear. These observations reflect the development of proceedings, inproceedings, journals, and articles sketched in Figure 6.4(b). According to the approximation functions in Figure 6.5, we also can observe that the number of inproceedings and articles clearly dominates the remaining document classes. Finally, we remark that – like in the original DBLP database – in the early years instances of selected document classes are missing, e.g. there are no Book documents for these years.

6.4. The SP²Bench Queries

In addition to the data generator, the SP²Bench framework provides a set of benchmark queries. These queries have been designed to test common SPARQL operator constellations, RDF data access patterns, and also particular SPARQL optimization strategies. In the following, we discuss RDF and SPARQL characteristics that were considered in query design (in Section 6.4.1 and Section 6.4.2, respectively), before presenting and discussing the benchmark queries one by one in Section 6.4.3.

6.4.1. RDF Characteristics of Interest

Like in the context of relational data, decisions on how RDF data is stored and accessed by engines may heavily influence system performance (see for instance the discussions in [AMMH07; WKB08; SHK+08]). Consequently, a SPARQL benchmark should consider the specifics of the underlying RDF data representation language.

The first aspect that is interesting with respect to the RDF data format is that RDF constitutes elements from three different sets, namely URIs, blank nodes, and

	Query	1	2	3abc	4	5ab	6
1	Data access: **B**LANK NODES, **L**ITERALS, **U**RIs, **La**RGE LITERALS, **C**ONTAINERS	L,U	L,U,La	L,U	B,L,U	B,L,U	B,L,U
2	Access pattern: **S**UBJ., **P**RED., **O**BJ., **N**ONE.	P,PO	P,PO	N,PO	P,PO	P,PO	P,PO
3	Operators: **A**ND, **F**ILTER, **U**NION, **O**PT	A	A,O	A,F	A,F	A,F	A,F,O
4	Modifiers: **D**ISTINCT, **L**IMIT, **Of**FSET, **O**RDER **B**Y	-	Ob	-	D	D	
5	Filter pushing possible?	-	-	✓	-	✓/-	✓
6	Reusing of graph patterns possible?	-	-	-	✓	-	✓

	Query	7	8	9	10	11	12c
1	Data access: **B**LANK NODES, **L**ITERALS, **U**RIs, **La**RGE LITERALS, **C**ONTAINERS	L,U,C	B,L,U	B,L,U	U	L,U	U
2	Access pattern: **S**UBJ., **P**RED., **O**BJ., **N**ONE.	P,PO	P,PO	N,PO	O	P	SPO
3	Operators: **A**ND, **F**ILTER, **U**NION, **O**PT	A,F,O	A,F,U	A,U	-	-	-
4	Modifiers: **D**ISTINCT, **L**IMIT, **Of**FSET, **O**RDER **B**Y	D	D	D	-	L,Ob,Of	-
5	Filter pushing possible?	✓	✓	-	-	-	-
6	Reusing of graph patterns possible?	✓	✓	✓	-	-	-

Table 6.4.: Selected properties of the SP²Bench benchmark queries.

literals. SPARQL engines might represent elements from these domains differently, e.g. it could make sense to have a special index for literals to accelerate text search. The SP²Bench queries therefore access all three entities and different combinations thereof; line 1 in Table 6.4 surveys this characteristic for the SP²Bench queries, where abbreviations are indicated by bold font (e.g. we use **B** as a shortcut for **B**LANK NODES).[10] Line 1 also indicates queries that access comparably large literals (namely the abstracts of documents) and RDF containers (i.e. outgoing references, which are of type `rdf:Bag`). Containers are of particular interest due to their special semantics and the fact that they induce a (possibly large) set of membership properties `rdf:_1`, `rdf:_2`, `rdf:_3`, ... As argued in [SHK+08], these membership properties may induce problems for RDF storage schemes like Vertical Partitioning [AMMH07].

A second important requirement imposed by the RDF data format is to test different data access patterns at triple pattern level. Triple patterns in SPARQL queries may contain variables in any position and the efficient evaluation of single patterns forms the basis for the fast evaluation of more complex queries. On the one hand, it seems reasonable to assume that in most triple patterns the predicate is fixed, such as is the case in patterns like `(?book,dc:creator,?name)` or `(Book1,dc:creator,?name)`, asking for authors of all or a fixed book, respectively. Such patterns can be seen as the natural counterpart of SQL queries, which select a fixed set of properties from some entities, by accessing exactly those table attributes that contain the desired properties. On the other hand, as also argued in [WKB08],

[10] As we shall see in Section 6.4.3, *Q*12*a* and *Q*12*b* are ASK-counterparts of the SELECT queries *Q*5*a* and *Q*8, respectively. The ASK versions are not explicitly listed in the table.

one strength of RDF querying is the flexibility of having variables in predicate position. This makes it possible to write patterns like `(Book1,?prop,?val)`, to obtain a compact description of a certain resource, or `(?subj,?prop,Person1)`, to extract all entities that stand in some relation to `Person1`. Suchlike data access patterns may be of particular importance if the underlying domain contains rather unstructured data and the properties of entities are not or only partially known.

We survey the data access patterns that are used in the SP²Bench queries in Table 6.4, Line 2. To give an example, the shortcut `PO` denotes that the respective query contains at least one triple pattern with fixed predicate and object position and a variable in subject position. Although in most triple patterns the predicate position is fixed, we took special care to design queries with other access patterns, such as triple patterns containing only variables (cf. $Q3a$, $Q3b$, $Q3c$, and $Q9$), patterns where only the object is bound (cf. $Q10$), or variable-free triple patterns (cf. $Q12c$). We will resume this discussion when presenting the queries individually in Section 6.4.3.

As a final RDF-related aspect, it is important that RDF provides graph-structured data. Hence, SPARQL engines should perform well on different kinds of graph patterns. Unfortunately, up to the present there exist only few real-world SPARQL scenarios. It would be necessary to analyze a large set of such scenarios, to extract graph patterns that frequently occur in practice. In the absence of this possibility, we roughly distinguish between long path chains, i.e. nodes linked to each other via a long path, and bushy patterns, i.e. single nodes that are linked to a multitude of other nodes, in the style of star-joins. It is impossible to give a precise characterization of "long" and "bushy", so we designed meaningful queries containing *comparably* long chains (i.e. $Q4$, $Q6$) and bushy patterns (i.e. $Q2$) w.r.t. our scenario.

6.4.2. SPARQL Characteristics of Interest

Next, we turn towards a discussion of SPARQL characteristics that were of particular interest in query design. Rows 3 and 4 in Table 6.4 survey the operators and solution modifiers used in the Select benchmark queries. It can be observed that the queries cover various operator constellations, combined with selected solution modifier combinations. We want to stress that we took special care at operator Opt, which has been shown to be the most complex operator in the SPARQL query language in Section 3.2.3 and can be used to encode closed-world negation (cf. the discussion in Section 4.2.6). Many interesting queries involve negation and therefore we explicitly test it in two benchmark queries, namely $Q6$ and $Q7$ (see Section 6.4.3).

Query Optimization. Another important objective of SP²Bench was to design queries that are amenable to a wide range of SPARQL optimization approaches. One promising approach to SPARQL optimization is the reordering of triple patterns based on selectivity estimations [NW08; SSB+08], similar in idea to join reordering in relational algebra optimization. A beneficial ordering of triple patterns depends on both the selectivity of triple patterns and data access paths provided by the engine.

Query	$Q1$	$Q2$	$Q3a$	$Q3b$	$Q3c$	$Q4$	$Q5a$	$Q5b$
$10k$	1	147	846	9	0	23226	155	155
$50k$	1	965	3647	25	0	104746	1085	1085
$250k$	1	6197	15853	127	0	542801	6904	6904
$1M$	1	32770	52676	379	0	2586733	35241	35241
$5M$	1	248738	192373	1317	0	18362955	210662	210662
$25M$	1	1876999	594890	4075	0	n/a	696681	696681

Query	$Q6$	$Q7$	$Q8$	$Q9$	$Q10$	$Q11$	$Q12a$	$Q12b$	$Q12c$
$10k$	229	0	184	4	166	10	*yes*	*yes*	*no*
$50k$	1769	2	264	4	307	10	*yes*	*yes*	*no*
$250k$	12093	62	332	4	452	10	*yes*	*yes*	*no*
$1M$	62795	292	400	4	572	10	*yes*	*yes*	*no*
$5M$	417625	1200	493	4	656	10	*yes*	*yes*	*no*
$25M$	1945167	5099	493	4	656	10	*yes*	*yes*	*no*

Table 6.5.: Number of query results ($Q1$–$Q11$) and results of ASK queries ($Q12abc$) on documents containing up to 25 million RDF triples.

Actually, most SP²Bench queries may benefit from such an optimization, because most of them contain large AND-connected blocks and require series of joins.

Closely related to triple reordering is filter pushing, which aims at an early evaluation of filter conditions (cf. Section 4.2.5). Like triple pattern reordering, filter pushing may speed up evaluation by decreasing the intermediate result size. Row 5 in Table 6.4 identifies SP²Bench queries that are amenable to such techniques.

Another reasonable idea is to reuse evaluation results of graph pattern evaluation (or even of whole subqueries). This strategy is applicable whenever the same pattern or subquery is used multiple times in the same query. As a simple example consider $Q4$ in Section 6.4.3. In that query, `?article1` and `?article2` in the first and second triple pattern will be bound to exactly the same nodes of the input RDF graph, so it suffices to evaluate this pattern only once and use this result for both subqueries. We investigate the applicability of this technique in Table 6.4, row 6. We will sketch more optimization approaches, such as semantic query optimization (cf. Chapter 5), when discussing the individual queries in the following subsection.

6.4.3. Discussion of Benchmark Queries

Before starting our discussion, we survey the result sizes of the individual queries on RDF documents of different size in Table 6.5. The overview shows that the queries vary in their result size (e.g., SP²Bench contains queries with increasing, constant, and empty result size). This survey forms the basis for the subsequent discussion.

In the following we discuss the SPARQL versions of the SP²Bench queries, which

can be processed directly by SPARQL engines (complementarily, there exist SQL-translations of these queries for relational storage schemes like a triple table approach or Vertical Partitioning [AMMH07] at the SP²Bench project page, see [SHK+08] for a discussion of these translations). In particular, we study the challenges the queries impose to SPARQL engines. Thereby, we distinguish between *in-memory* engines, which load the document from file and process queries in main memory (e.g., the Jena ARQ engine [arq]), and *native* engines, which rely on a physical database system (e.g., the Virtuoso engine [Bla07]). When discussing the challenges for native engines, we always assume that the document has been loaded into the database prior to query processing. We refer the reader to the survey of namespaces in Appendix A for a complete listing of all namespaces that are used in the benchmark queries.

Benchmark Query Q1: *Return the year of publication of "Journal 1 (1940)".*

```
SELECT ?yr
WHERE {
  ?journal rdf:type bench:Journal.
  ?journal dc:title "Journal 1 (1940)"^^xsd:string.
  ?journal dcterms:issued ?yr }
```

Benchmark query $Q1$ returns exactly one result (for sufficiently large documents). Native engines may use index lookups to answer this query in (almost) constant time, i.e. execution time should be independent from document size. In-memory engines must scan the whole document and should scale linearly with document size.

Benchmark Query Q2: *Extract all inproceedings with properties* dc:creator, bench:booktitle, dcterms:issued, dcterms:partOf, rdfs:seeAlso, dc:title, swrc:pages, foaf:homepage, *and optionally* bench:abstract, *including the respective values.*

```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings.
  ?inproc dc:creator ?author.
  ?inproc bench:booktitle ?booktitle.
  ?inproc dc:title ?title.
  ?inproc dcterms:partOf ?proc.
  ?inproc rdfs:seeAlso ?ee.
  ?inproc swrc:pages ?page.
  ?inproc foaf:homepage ?url.
  ?inproc dcterms:issued ?yr
  OPTIONAL { ?inproc bench:abstract ?abstract }
} ORDER BY ?yr
```

This query implements a large star-join pattern, where different properties of inproceedings (variable `?inproc`) are requested. It contains a simple OPT clause,

and accesses large strings (i.e. the abstracts). Result size grows with database size and a final result ordering is necessary due to operator ORDER BY. Both native and in-memory engines may reach evaluation times almost linear to document size.

Benchmark Queries Q3abc: *Select all articles with property (a)* swrc:pages*, (b)* swrc:month*, or (c)* swrc:isbn.

```
(a) SELECT ?article
    WHERE { ?article rdf:type bench:Article.
            ?article ?property ?value
    FILTER (?property=swrc:pages) }

(b) Q3a, but "swrc:month" instead of "swrc:pages"

(c) Q3a, but "swrc:isbn" instead of "swrc:pages"
```

These three queries test FILTER expressions with varying selectivity. According to Table 6.1, the FILTER expression in $Q3a$ is not very selective (i.e. retains about 92.61% of all articles). Data access through a secondary index for $Q3a$ is probably not efficient, but may work well for $Q3b$, which selects only 0.65% of all articles. The filter in $Q3c$ is never satisfied, because articles never have predicate swrc:isbn. Native engines may use statistics to answer $Q3c$ in constant time. As an alternative strategy, filter elimination techniques in the style of Lemma 4.3 may be applied.

Benchmark Query Q4: *Select all distinct pairs of article author names for authors that have published in the same journal.*

```
SELECT DISTINCT ?name1 ?name2
WHERE { ?article1 rdf:type bench:Article.
        ?article2 rdf:type bench:Article.
        ?article1 dc:creator ?author1.
        ?author1 foaf:name ?name1.
        ?article2 dc:creator ?author2.
        ?author2 foaf:name ?name2.
        ?article1 swrc:journal ?journal.
        ?article2 swrc:journal ?journal
        FILTER (?name1<?name2) }
```

$Q4$ contains a rather long graph chain, i.e. variables `?name1`, and `?name2` are linked through the articles that different authors have published in the same journal. As one may expect, the result is very large (cf. Table 6.5). Instead of evaluating the outer pattern block and applying the FILTER afterwards, engines may embed the FILTER expression in the computation of the inner block, e.g. by exploiting indices on author names. The DISTINCT modifier further complicates the query. We shall expect superlinear behavior for this query, even for native engines.

Benchmark Queries Q5ab: *Return the names of all persons that occur as author of at least one inproceeding and at least one article.*

```
(a) SELECT DISTINCT ?person ?name
    WHERE { ?article rdf:type bench:Article.
            ?article dc:creator ?person.
            ?inproc rdf:type bench:Inproceedings.
            ?inproc dc:creator ?person2.
            ?person foaf:name ?name.
            ?person2 foaf:name ?name2
            FILTER(?name=?name2) }

(b) SELECT DISTINCT ?person ?name
    WHERE { ?article rdf:type bench:Article.
            ?article dc:creator ?person.
            ?inproc rdf:type bench:Inproceedings.
            ?inproc dc:creator ?person.
            ?person foaf:name ?name }
```

$Q5a$ and $Q5b$ test different join variants: $Q5a$ implements an implicit join on author names (encoded in the FILTER condition), while $Q5b$ explicitly joins the authors on variable `?name`. Although the queries are not equivalent in the general case, the one-to-one mapping between authors and their names in the SP²Bench scenario implies equivalence (i.e., the attribute `foaf:name` is a primary key for objects of type `foaf:Person`). In Section 5.4, semantic query optimization using such keys for RDF has been proposed. When coupled with filter manipulation rules (e.g. in the style of Lemma 4.3), such approaches may detect the equivalence of $Q5a$ and $Q5b$ in this scenario and choose the more efficient version among $Q5a$ and $Q5b$.

Benchmark Query Q6: *Return, for each year, the set of all publications authored by persons that have not published in years before.*

```
SELECT ?yr ?name ?doc
WHERE {
?class rdfs:subClassOf foaf:Document.
?doc rdf:type ?class.
?doc dcterms:issued ?yr.
?doc dc:creator ?author.
?author foaf:name ?name
OPTIONAL {
  ?class2 rdfs:subClassOf foaf:Document.
  ?doc2 rdf:type ?class2.
  ?doc2 dcterms:issued ?yr2.
  ?doc2 dc:creator ?author2
  FILTER (?author=?author2 && ?yr2<?yr) }
FILTER (!bound(?author2)) }
```

This query implements closed-world negation, expressed through the combination of the operators OPT, FILTER, and filter predicate *bnd* (denoted as BOUND in the official syntax) sketched in Section 4.2.6. To shortly discuss the query, the idea of the construction is that the block outside the OPT expression computes all publications, while the inner one constitutes earlier publications from authors that appear outside. The outer FILTER expression then retains publications for which variable `?author2` is unbound, i.e. exactly those publications from authors that have not published in earlier years. For this query, SPARQL-specific optimization like the rewriting of closed-world negation discussed in Section 4.2.6 may be beneficial.

Benchmark Query Q7: *Return the titles of all publications that have been cited at least once, but not by any paper that has not been cited itself.*

```
SELECT DISTINCT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document.
  ?doc rdf:type ?class.
  ?doc dc:title ?title.
  ?bag2 ?member2 ?doc.
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document.
    ?doc3 rdf:type ?class3.
    ?doc3 dcterms:references ?bag3.
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document.
      ?doc4 rdf:type ?class4.
      ?doc4 dcterms:references ?bag4.
      ?bag4 ?member4 ?doc3
    } FILTER (!bound(?doc4))
  } FILTER (!bound(?doc3))
}
```

Benchmark query $Q7$ tests double negation, which requires the encoding of nested closed-world negation through a nesting of the OPT + FILTER + *bnd* construction used in the previous query. Given that the citation system of DBLP contains only few incoming and outgoing citations (cf. Section 6.2.5), the query returns only few results, even on very large documents (see Table 6.5). However, we expect the query to be challenging, due to the double negation. As for $Q7$, engines may use the rewriting scheme for closed-world negation presented in Section 4.2.6. Another possible optimization strategy is to reuse graph pattern evaluation results. For instance, the AND-connected block `$1 rdfs:subClassOf foaf:Document. $2 rdf:type $1` occurs three times in the query, for (i) `$1:=?class`, `$2:=?doc`, for (ii) `$1:=?class3`, `$2:=?doc3`, and also for (iii) `$1:=?class4`, `$2:=?doc4`.

Benchmark Query Q8: *Compute authors with Erdös number one or two.*

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes rdf:type foaf:Person.
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string.
  { ?doc dc:creator ?erdoes.
    ?doc dc:creator ?author.
    ?doc2 dc:creator ?author.
    ?doc2 dc:creator ?author2.
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?doc2!=?doc &&
            ?author2!=?erdoes &&
            ?author2!=?author)
  } UNION {
    ?doc dc:creator ?erdoes.
    ?doc dc:creator ?author.
    ?author foaf:name ?name
    FILTER (?author!=?erdoes) }
}
```

Here, the evaluation of the second UNION part is basically "contained" in the evaluation of the first part. Hence, techniques like graph pattern (or subexpression) reusing are applicable. Another possible strategy is to decompose the first filter expression and push its components down in the operator tree (cf. Section 4.2.5), to apply atomic filter conditions early and decrease the size of intermediate results.

Benchmark Query Q9: *Return incoming and outgoing properties of persons.*

```
SELECT DISTINCT ?predicate
WHERE {
  { ?person rdf:type foaf:Person.
    ?subject ?predicate ?person } UNION
  { ?person rdf:type foaf:Person.
    ?person ?predicate ?object } }
```

*Q*9 has been primarily designed to test non-standard data access patterns. Naive implementations would compute the triple patterns of the UNION subexpressions separately, thus evaluating patterns where no component is bound. Then, pattern `?subject ?predicate ?person` selects all graph triples, which is rather inefficient. Another approach is to evaluate the first triple in each UNION subexpression, afterwards using the bindings for variable `?person` to evaluate the second pattern efficiently. Note that the result size is exactly 4 for sufficiently large documents (see Table 6.5). RDF-specific statistics about incoming and outgoing properties of `foaf:Person`-typed objects (in native engines) may help to answer this query in

constant time, even without data access. In-memory engines, however, must always load the whole document and therefore should scale linearly with document size.

Benchmark Query Q10: *Return all subjects that stand in some relation to person "Paul Erdös", including the type of their relation.*

```
SELECT ?subj ?pred
WHERE { ?subj ?pred person:Paul_Erdoes }
```

In the SP²Bench scenario, this query can be reformulated as: *Return publications and venues in which "Paul Erdös" is involved as author or as editor.* It implements an object bound-only access pattern. In contrast to $Q9$, statistics are not immediately useful, because the result includes the subjects (i.e., the subjects must be extracted from the data set). Recalling that Paul Erdös is active only between 1940 and 1996 (cf. Section 6.3.1), the result size stabilizes for large documents. Native engines that exploit indices may reach (almost) constant execution time.

Benchmark Query Q11: *Return (up to) 10 electronic edition URLs starting from the 51^{th} publication, in lexicographical order.*

```
SELECT ?ee
WHERE { ?publication rdfs:seeAlso ?ee }
ORDER BY ?ee LIMIT 10 OFFSET 50
```

The focus of this query lies on the combination of solution modifiers Order By, Limit, and Offset. In-memory engines have to read, process, and sort electronic editions prior to application of the Limit and Offset modifiers. In contrast, native engines may exploit indices to access only a fraction of all electronic editions and, as the result size is limited to 10 due to the Limit modifier, would optimally reach constant runtime, independently from the size of the input document.

Benchmark Query Q12: *(a) Return yes if a person occurs as author of at least one inproceeding and at least one article, no otherwise; (b) Return yes if there is an author with Erdös number one or two in the database, and no otherwise; (c) Return yes if the person "John Q Public" is contained in the database.*

```
(Q12a) Q5a as ASK query

(Q12b) Q8 as ASK query

(Q12c) ASK {person:John_Q_Public rdf:type foaf:Person}
```

All three queries are boolean queries, designed to test the efficient implementation of the SPARQL Ask query form. $Q12a$ and $Q12b$ share the properties of their Select counterparts $Q5a$ and $Q8$, respectively. They always return *yes* for sufficiently

large documents. When evaluating Ask queries, engines should stop the evaluation process as soon as a solution has been found. A reasonable optimization approach would be to adapt the query execution plan, trying to efficiently locate a witness (e.g. using the rewriting rules for Ask queries in Lemma 4.14, Section 4.4.1). For instance, based on execution time estimations it may be favorable to evaluate the (simpler) second part of the Union in $Q12b$ first. $Q12c$ asks for a single triple that is not present in the database. With indices, native engines may answer $Q12c$ in constant time. Again, in-memory engines must read and scan the whole document.

6.5. Benchmark Metrics

In this section we propose several benchmark metrics that cover different aspects of the evaluation process. These metrics reflect the scope and design decisions of SP²Bench and can be used to systematically evaluate benchmark results.

We propose to perform three runs over documents comprising $10k$, $50k$, $250k$, $1M$, $5M$, and $25M$ RDF triples, using a fixed timeout of 30min per query and document. The reported time should include the average value over all three runs and, if significant, the errors within these runs. This setting was tested in [SHK+08; SHLP09] and can be evaluated in reasonable time for state-of-the-art engines (typically within few days). If the tested engine is fast enough, nothing prevents the user from adding larger documents. We recommend the following five benchmark metrics.

1. Success Rate: As a first indicator we propose to survey the success rates for the engine on top of all document sizes, distinguishing between Success, Timeout (e.g. an execution time $> 30min$), Memory Exhaustion (if an additional memory limit was set), and general Errors. This metric gives a survey over scaling properties and first insights into the engine's overall behavior.
2. Loading Time: The loading time for documents of different sizes is interesting to get insights into the efficiency and, particularly, to see how document loading scales with document size. This metric primarily applies to engines with a physical database backend and may be irrelevant for in-memory engines, where document loading is usually part of the query evaluation process.
3. Memory Consumption: In particular for engines with a physical backend, the main memory consumption for the individual queries and also the average memory consumption over all queries may be of interest. Optimally, physical backend database engines should get by with a constant main memory consumption, independently from the size of the input document.
4. Per-Query Performance: Individual performance results for all queries over all document sizes give more detailed insights into the behavior than the Success Rate metric discussed before. The Per-Query Performance metric forms the basis for a deep study of the results and allows to investigate

strengths, weaknesses, and scaling of the tested implementation based on a one by one discussion of the engine results for the individual benchmark queries.

5. Global Performance: This metric integrates the individual per-query results into a global performance measure. It contains, for each tested document size, the arithmetic and the geometric mean[11] of the engine's average execution time over all queries. We propose to penalize timeouts and other errors with $3600s$ (i.e., twice the evaluation time limit). Global Performance is well-suited to investigate the scaling properties of engines and to compare the performance of different evaluation approaches. Note that the arithmetic mean imposes a stronger penalty on outlier queries than the geometric mean. Generally speaking, the geometric mean better reflects the average behavior of engines, while the arithmetic mean captures the worst-case behavior.

6.6. Related Work

The Benchmark Handbook [Gra93] provides a summary of important database benchmarks. Probably the most "complete" benchmark suite for relational systems is TPC [tpc], which defines performance and correctness benchmarks for a large variety of scenarios. Going beyond relational data, a variety of benchmarks have been developed for other data models, such as the OO7 benchmark [CDN93] for object-oriented databases and the XMark benchmark [SWK+02] for XML data processing.

Coming along with the proliferation of the Semantic Web, benchmarking has become an increasingly important topic in the context of data formats like RDF(S) and OWL. In response, also in this context several benchmark platforms have been developed. These platforms address both structural aspects of the data (e.g., [MACP02]) as well as efficient data processing (e.g., [GPH05; AMMH; BS09]).

The LUBM and the Barton Library benchmark have been discussed in the beginning of this chapter and we will not further elaborate on them here. Another notable benchmark project in the context of SPARQL is the application-oriented Berlin SPARQL Benchmark (BSBM) [BS; BS08; BS09]. BSBM has been developed independently, in parallel to the SP^{2}Bench framework. It tests the performance of SPARQL engines in a prototypical e-commerce scenario, so it is use-case driven. Compared to the SP^{2}Bench queries, the BSBM queries are typically simpler and laid out to be evaluated in a work load setting. This makes the BSBM benchmark particularly interesting to compare the performance of engines that expose SPARQL endpoints via the SPARQL protocol and gives the benchmark a more industrial flavor. Hence, with its focus BSBM is supplementary to the SP^{2}Bench suite.

The RDF(S) data model benchmark in [MACP02] focuses on structural properties of RDF Schemas. In [TTKC08], graph features of RDF Schemas are studied. The results constitute a valuable basis for synthetic schema generation. With their focus

[11] The geometric mean is defined as the nth root of the product over n numbers.

on schemas, however, both [MACP02] and [TTKC08] are complementary to our work. As an interesting observation, the investigations in [TTKC08] reveal that power law distributions are naturally encountered in large RDF Schema descriptions, which justifies the use of such distributions in the SP²Bench data.

The topic of data generation is not new either. In the context of the Semantic Web, a synthetic data generation approach for OWL based on test data is described in [WGQH05]. There, the focus is on rapidly generating large data sets from representative sample data of a fixed domain. Our data generation approach is more fine-grained, as we analyze the development of entities (e.g. articles) over time and the generated documents reflect many characteristics found in social communities.

6.7. Conclusion

The SP²Bench performance benchmark for SPARQL constitutes the first methodical approach for testing the performance of engines w.r.t. different operator constellations, RDF access paths, typical RDF constructs, and a variety of possible optimization approaches. In this line, our framework allows to assess the performance of SPARQL in a general, comprehensive way and is a useful tool for quality assessment for both research prototypes and industrial-strength SPARQL implementations.

The SP²Bench data generator relies on an in-depth study of DBLP. Although it is impossible to mirror all correlations found in the original DBLP data (e.g., we simplified when assuming independence between attributes in Section 6.2.1), many aspects are modeled in faithful detail. Hence, the data generator forms a contribution on its own and may be useful in other Semantic Web projects where large amounts of test data with natural distributions are needed. In SP²Bench, this data forms the foundation for the design of challenging benchmark queries, which – due to the real-world distributions – are realistic, understandable, and predictable.

We conclude with the remark that, with the choice of the well-known DBLP scenario, we also clear the way for coming extensions of the SPARQL query language. For instance, SPARQL update and aggregation support are currently planned as extensions [spaa]. Updates, for example, require only minor modifications to the SP²Bench data generator. Concerning aggregations, we argue that the detailed knowledge of the document class counts and distributions (cf. Section 6.2) facilitates the design of challenging aggregate queries with predictable characteristics.

Chapter 7.

Conclusion and Outlook

Although over the last years a variety of research has been done in the context of RDF data and SPARQL query processing, both technologies are still in their infancy. As witnessed by our experimental results presented in [SHK+08; SHLP09], current SPARQL implementations like ARQ [arq], Sesame [ses], or Virtuoso [Bla07] still suffer from severe performance bottlenecks when dealing with medium- and large-scale RDF databases, even for presumably simple queries that can be processed efficiently in a comparable relational setting. Tackling the deficiencies of these implementations, a variety of promising optimization schemes and research prototypes have been developed (e.g. [AMMH07; NW08; WKB08; NW09]). However, as discussed earlier in the introduction of Chapter 4, these approaches typically focus on the optimization of And-only queries. Experiments that confirm their efficiency for larger SPARQL fragments (and hence, more complex queries) are still outstanding.

In addition, other important issues that may arise in the context of real-world data management systems for RDF, such as concurrency control, transaction management, or recovery, have not or only marginally been addressed to date. On the one hand, it seems reasonable to assume that – like for query optimization – the Semantic Web community can benefit from established approaches that have been proposed for relational database systems. On the other hand, one may expect that also in these areas new challenges arise, due to the specifics of the RDF data format and the SPARQL query language. To give an example, the discussion in [Muy08] indicates deficiencies of traditional serialization strategies for concurrent transactions in the context of RDF data (such as e.g. two-phase locking [EGLT76]) and proposes a concurrency control protocol specifically designed for RDF databases.

Beyond those mentioned above, other new challenges in SPARQL query processing and RDF data management will arise in response to future extensions of the SPARQL query language. In its current Working Draft [spaa], the W3C SPARQL Working Group tackles a variety of new features for SPARQL, such as aggregate functions, nested subqueries, negation, property paths, and updates, to increase the practicability of the SPARQL query language. While some of them are only syntactic sugar (e.g., as shown in [AG08a], negation can be simulated using operators Opt, Filter, and filter predicate *bnd*), others like aggregate functions increase the

expressive power of SPARQL and therefore complicate query processing. Again, the community may fall back on experience gathered in the relational context, as a basis for the development of efficient solutions for the SPARQL query language.

Based on all these observations, we expect that there is still a long way to go to industrial-strength SPARQL implementations that cope efficiently and reliably with challenging SPARQL queries in the context of large-scale RDF repositories. We hope that our investigations of SPARQL complexity, optimization, and benchmarking presented in this thesis are one more step towards the understanding of the theoretical foundations of SPARQL, towards the understanding of its relationship to established data models, towards the design, implementation, and testing of SPARQL engines, and lastly towards the realization of the Semantic Web vision.

Bibliography

[AB07] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2007.

[ABC+76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, William F. King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[ABE09] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenate. Extending SPARQL with regular expression patterns (for querying RDF). *Web Semantics*, 7(2):57–73, 2009.

[ACKP01] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, and Dimitris Plexousakis. On Storing Voluminous RDF Descriptions: The case of Web Portal Catalogs. In *WebDB*, pages 43–48, 2001.

[AG08a] Renzo Angles and Claudio Gutierrez. The Expressive Power of SPARQL. In *ISWC*, pages 114–129, 2008.

[AG08b] Renzo Angles and Claudio Gutierrez. The Expressive Power of SPARQL. Technical report, Universidad de Chile, TR/DCC-2008-5, 2008.

[AHV] Serge Abiteboul, Richard Hull, and Victor Vianu. Foundations of Databases. Addison-Wesley 1995.

[AMMH] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Using the Barton libraries dataset as an RDF benchmark. TR MIT-CSAIL-TR-2007-036, MIT.

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422, 2007.

[arq] ARQ SPARQL Processor for Jena. `http://jena.sourceforge.net/ARQ/`.

[ASU79] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient Optimization of a Class of Relational Expressions. *ACM Trans. Database Syst.*, 4(4):435–454, 1979.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. The Universality of Data Retrieval Languages. In *POPL*, pages 110–120, 1979.

[BB79] Catriel Beeri and Philip A. Bernstein. Computational Problems Related to the Design of Normal Form Relational Schemas. *ACM Trans. Database Syst.*, 4(1):30–59, 1979.

[BC07] Christian Bizer and Richard Cyganiak. D2R Server publishing the DBLP Bibliography Database, 2007. `http://www4.wiwiss.fu-berlin.de/dblp/`.

[BCM+03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[BEE+07] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of Rule-based Query Answering. In *Reasoning Web*, pages 1–153, 2007.

[BFL+08] François Bry, Tim Furche, Clemens Ley, Benedikt Linse, and Bruno Marnette. RDFLog: It's Like Datalog for RDF. In *WLP*, 2008.

[BFW99] Peter Buneman, Wenfei Fan, and Scott Weinstein. Query Optimization for Semistructured Data Using Path Constraints in a Deterministic Data Model. In *DBLP*, pages 208–223, 1999.

[BKS07] Abraham Bernstein, Christoph Kiefer, and Markus Stocker. OptARQ: A SPARQL Optimization Approach based on Triple Pattern Selectivity Estimation. Technical report, University of Zurich, 2007.

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC*, pages 54–68, 2002.

[Bla07] Carl Blakeley. Mapping Relational Data to RDF with Virtuoso's RDF Views, 2007. OpenLink Software.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 2001.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173–189, 1972.

[BS] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/`.

[BS08] Christian Bizer and Andreas Schultz. Benchmarking the Performance of Storage Systems that expose SPARQL Endpoints. In *SSWS*, 2008.

[BS09] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. In *International Journal On Semantic Web and Information Systems - Special Issue on Scalability and Performance of Semantic Web Systems*, 2009.

[BV84] Catriel Beeri and Moshe Y. Vardi. A Proof Procedure for Data Dependencies. *J. ACM*, 31(4):718–741, 1984.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In *SIGMOD Workshop, Vol. 1*, pages 249–264, 1974.

[CDES05] Eugene I. Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *VLDB*, pages 1216–1227, 2005.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *SIGMOD*, pages 12–21, 1993.

[CGK08] Andrea Cali, Georg Gottlob, and Michael Kifer. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In *Descr. Logics*, 2008.

[CGL09] Andrea Cali, Georg Gottlob, and Thomas Lukasiewicz. Datalog±: a Unified Approach to Ontologies and Integrity Constraints. In *ICDE*, pages 14–30, 2009.

[CGM90] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based Approach to Semantic Query Optimization. *TODS*, 15(2):162–207, 1990.

[CH80] Ashok K. Chandra and David Harel. Structure and Complexity of Relational Queries. In *FOCS*, pages 333–347, 1980.

[Cha98] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, pages 34–43, 1998.

[Che76] Peter P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[CLJF06] Artem Chebotko, Shiyong Lu, Hasan M. Jamil, and Farshad Fotouhi. Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical report, Wayne State University, TR-DB-052006-CLJF, 2006.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *STOC*, pages 77–90, 1977.

[Cod69] Edgar F. Codd. Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks. *IBM Research Report, San Jose, California*, RJ599, 1969.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.

[Cod71] Edgar F. Codd. Further Normalization of the Data Base Model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[Cod72] Edgar F. Codd. Relational Completeness of Data Base Sublanguages. *Database Systems*, pages 65–98, 1972.

[Cod74] Edgar F. Codd. Recent Investigations into Relational Data Base Systems. *IBM Research Report, San Jose, California*, RJ1385, 1974.

[Cod79] Edgar F. Codd. Extending the Data Base Model to Capture More Meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.

[CPST03] Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Sotirios Tourtounis. On Labeling Schemes for the Semantic Web. In *WWW*, pages 544–555, 2003.

[CV85] Ashok K. Chandra and Moshe Y. Vardi. The Implication Problem for Functional and Inclusion Dependencies is Undecidable. *SIAM J. Comput.*, 14(3):671–677, 1985.

[CWI] CWI Amsterdam. MonetDB. `http://monetdb.cwi.nl/`.

[Cyg05] Richard Cyganiac. A relational algebra for SPARQL. Technical report, HP Laboratories Bristol, 2005.

[Dat81] Christopher J. Date. Referential Integrity. In *VLDB*, pages 2–12, 1981.

[Deu08] Alin Deutsch. FOL Modeling of Integrity Constraints (Dependencies). *DB Encyclopedia*, 2008.

[DNR08] Alin Deutsch, Alan Nash, and Jeff Remmel. The Chase Revisited. In *PODS*, pages 149–158, 2008.

[DPT06] Alin Deutsch, Lucian Popa, and Val Tannen. Query Reformulation with Constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[DT01] Alin Deutsch and Val Tannen. Containment for Classes of XPath Expressions Under Integrity Constraints. In *Knowledge Representation Meets Databases*, 2001.

[DT05] Alin Deutsch and Val Tannen. XML Queries and Constraints, Containment and Reformulation. *Theor. Comput. Sci.*, 336(1):57–87, 2005.

[dub] Dublin Core Metadata Initiative. `http://dbulincore.org`.

[EFT94] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic*. Springer, 1994.

[EGLT76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997.

[EL05] Ergin Elmacioglu and Dongwon Lee. On Six Degrees of Separation in DBLP-DB and More. *SIGMOD*, 34(2), 2005.

[Fag77] Ronald Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Trans. Database Syst.*, 2(3):262–278, 1977.

[Fag82] Ronald Fagin. Horn Clauses and Database Dependencies. *J. ACM*, 29(4):952–985, 1982.

[FB08] George H. L. Fletcher and Peter W. Beck. A Role-free Approach to Indexing Large RDF Data Sets in Secondary Memory for Efficient SPARQL Evaluation. Technical report, arXiv:0811.1083 cs.DB, 2008. `http://arxiv.org/abs/0811.1083`.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data Exchange: Semantics and Query Answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[FL01] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. In *J. ACM*, pages 114–125, 2001.

[foa] The Friend of a Friend (FOAF) Project. `http://www.foaf-project.org/`.

[FS03] Wenfei Fan and Jérôme Siméon. Integrity Constraints for XML. *J. Comput. Syst. Sci.*, 66(1):254–291, 2003.

[FT05] Enrico Franconi and Sergio Tessaris. The Semantics of SPARQL, 2005. `http://www.inf.unibz.it/krdb/w3c/sparql/`. Editors working draft.

[GdMB08] Aurona Gerber, Alta Van der Merwe, and Andries Barnard. A Functional Semantic Web Architecture. In *ESWC*, pages 273–287, 2008.

[GGK09] Jinghua Groppe, Sven Groppe, and Jan Kolbaum. Optimization of SPARQL by using coreSPARQL. In *ICEIS*, pages 107–112, 2009.

[GGL07] Sven Groppe, Jinghua Groppe, and Volker Linnemann. Using an Index of Precomputed Joins in order to speed up SPARQL Processing. In *ICEIS*, pages 13–20, 2007.

[GHM04] Claudio Gutierrez, Carlos A. Hurtado, and Alberto O. Mendelzon. Foundations of Semantic Web Databases. In *PODS*, pages 95–106, 2004.

[GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The Complexity of XPath Query Evaluation. In *PODS*, pages 179–190, 2003.

[GKPS05] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The Complexity of XPath Query Evaluation and XML Typing. *J. ACM*, 52(2), 2005.

[GLR97] César A. Galindo-Legaria and Arnon Rosenthal. Outerjoin Simplification and Reordering for Query Optimization. *ACM Trans. Database Syst.*, 22(1):43–73, 1997.

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation.* Prentice Hall, 2000.

[gnu] Gnuplot. `http://www.gnuplot.info/`.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the WWW*, 3(2-3):158–182, 2005.

[Gra93] Jim Gray. *The Benchmark Handbook for Database and Transaction Systems.* Morgan Kaufmann, 1993.

[Gru02] Thorsten Grust. Accelerating XPath Location Steps. In *SIGMOD*, pages 109–120, 2002.

[Hal75] Patrick A. V. Hall. Optimization of a Single Expression in a Relational Data Base System. Technical report, IBM UKSC Report 76, 1975.

[Hal01] Alon Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, pages 270–294, 2001.

[Har09] Olaf Hartig. Trustworthiness of Data on the Web. In *STI Berlin & CSW PhD Workshop*, 2008/09.

[HBEV04] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF Query Languages. In *ISWC*, pages 502–517, 2004.

[HD05] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *LA-WEB*, pages 71–80, 2005.

[HG03] Stephen Harris and Nicholas Gibbins. 3store: Efficient Bulk RDF Storage. In *PSSS*, 2003.

[HH07] Olaf Hartig and Ralf Heese. The SPARQL Query Graph Model for Query Optimization. In *ESWC*, pages 564–578, 2007.

[HM75] Michael M. Hammer and Dennis J. McLeod. Semantic Integrity in a Relational Data Base System. In *VLDB*, pages 25–47, 1975.

[HS08] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *SIGMOD*, pages 405–418, 2008.

[JK82] David S. Johnson and Anthony C. Klug. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. In *PODS*, pages 164–169, 1982.

[JK84] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[KACP02] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, and Dimitris Plexousakis. RQL: a Declarative Query Language for RDF. In *WWW*, pages 592–603, 2002.

[Kin81] Jonathan J. King. QUIST: a system for semantic query optimization in relational databases. In *VLDB*, pages 510–517, 1981.

[KJ07] Krys Kochut and Maciej Janik. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In *ESWC*, pages 145–159, 2007.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *J. ACM*, 42(4):741–843, 1995.

[KM01] Marja R. Koivunen and Eric Miller. W3C Semantic Web Activity. In *Semantic Web Kick-Off in Finland – Vision, Technologies, Research, and Applications*, pages 27–43, 2001.

[Len02] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[Ley] Michael Ley. DBLP Database. `http://www.informatik.uni-trier.de/~ley/db/`.

[lin] Linked Data. `http://linkeddata.org/`.

[LM01] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB*, pages 361–370, 2001.

[LMS08] Georg Lausen, Michael Meier, and Michael Schmidt. SPARQLing Constraints for RDF. In *EDBT*, pages 499–509, 2008.

[Lot26] Alfred J. Lotka. The Frequency Distribution of Scientific Production. *Acad. Sci.*, 16:317–323, 1926.

[MACP02] Aimilia Magkanaraki, Sofia Alexaki, Vassilis Christophides, and Dimitris Plexousakis. Benchmarking RDF Schemas for the Semantic Web. In *ISWC*, pages 132–146, 2002.

[Mar04] Draltan Marin. RDF Formalization. Technical report, Universidad de Chile, TR/DCC-2006-8, 2004. `http://www.dcc.uchile.cl/~cgutierr/ftp/draltan.pdf`.

[Mar09] Bruno Marnette. Generalized Schema-Mappings: From Termination To Tractability. In *PODS*, pages 13–22, 2009.

[McL99] Michael P. McLaughlin. A Compendium of Common Probability Distributions, 1999.

[MD88] M. Muralikrishna and David J. DeWitt. Equi-Depth Histograms for Estimating Selectivity Factors For Multi-Dimensional Queries. In *SIGMOD*, pages 28–36, 1988.

[MHS07] Boris Motive, Ian Horrocks, and Ulrike Sattler. Adding Integrity Constraints to OWL. In *OWLED-07*, 2007.

[MMS79] David Maier, Alberto Mendelzon, and Yehoshua Sagiv. Testing Implications of Data Dependencies. *TODS*, pages 455–469, 1979.

[MPG07] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Minimal Deductive Systems for RDF. In *ESWC*, pages 53–67, 2007.

[MSL09a] Michael Meier, Michael Schmidt, and Georg Lausen. On Chase Termination Beyond Stratification. In *VLDB*, 2009.

[MSL09b] Michael Meier, Michael Schmidt, and Georg Lausen. On Chase Termination Beyond Stratification. Technical report, arXiv:0906.4228v2 cs.DB, 2009. `http://arxiv.org/abs/0906.4228`.

[MSL09c] Michael Meier, Michael Schmidt, and Georg Lausen. Stop the Chase. Technical report, arXiv:0901.3984 cs.DB, 2009. `http://arxiv.org/abs/0901.3984`.

[MSL09d] Michael Meier, Michael Schmidt, and Georg Lausen. Stop the Chase: Short Contribution. In *Alberto Mendelzon Workshop on Foundations of Data Management*, 2009.

[Muy08] Andrae Muys. A Concurrency Protocol for Multiversion RDF Datastores (Discussion Paper). In *Mulgara Workshop, San Francisco*, 2008. `http://www.netymon.com/papers/XA2-Discussion-Papers/XA2-CC-Design.pdf`.

[MW88] David Maier and David S. Warren. *Computing with logic: logic programming with Prolog*. Benjamin-Cummings Publishing Co., 1988.

[Nic78] Jean-Marie Nicloas. First Order Logic Formalization for Functional, Multivalued, and Mutual Dependencies. In *SIGMOD*, pages 40–46, 1978.

[not] Notation 3 (N3). `http://www.w3.org/DesignIssues/Notation3`.

[ntr] N-Triples. `http://www.w3.org/TR/rdf-testcases/#ntriples`.

[NW08] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[NW09] Thomas Neumann and Gerhard Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD*, pages 627–640, 2009.

[OH08] Kieron O'Hara and Wendy Hall. Trust on the Web: Some Web Science Research Challenges. *e-Journal on the Knowledge Society*, 2008.

[OHK09] Dan Olteanu, Jiewen Huang, and Christoph Koch. SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases. In *ICDE*, 2009.

[ope] openRDF.org. The SeRQL query language (revision 1.2). `http://www.openrdf.org/doc/sesame/users/ch06.html`.

[owl] Web Ontology Language OWL. `http://www.w3.org/2004/OWL/`.

[PAG06a] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. Technical report, arXiv:0605124 cs.DB, 2006. `http://arxiv.org/abs/0605124`.

[PAG06b] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics of SPARQL. Technical report, Universidad de Chile, TR/DCC-2006-16, 2006.

[PAG08] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A Navigational Language for RDF. In *ISWC*, pages 66–81, 2008.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 2009.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pol07] Axel Polleres. From SPARQL to Rules (and back). In *WWW*, pages 787–796, 2007.

[PSS07] Axel Polleres, Francois Scharffe, and Roman Schindlauer. SPARQL++ for Mapping Between RDF Vocabularies. *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, pages 878–896, 2007.

[rdfa] Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/rdf-concepts/`.

[rdfb] RDF Primer. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/REC-rdf-syntax/`.

[rdfc] RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/rdf-schema/`.

[rdfd] RDF Semantics. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/rdf-mt/`.

[rdfe] RDF/XML Syntax Specification (Revised). W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[rifa] RIF Working Group. W3C Initiative. `http://www.w3.org/2005/rules/wiki/RIF_Working_Group`.

[rifb] Rule Interchange Format Working Group Charter. `http://www.w3.org/2005/rules/wg/charter.html`.

[SC75] John M. Smith and Philip Y. Chang. Optimizing the Performance of a Relational Algebra Database Interface. *Commun. ACM*, 18(10):568–579, 1975.

[SD01] Michael Sintek and Stefan Decker. TRIPLE – An RDF Query, Inference, and Transformation Language. In *INAP*, pages 47–56, 2001.

[ses] openRDF.org – home of Sesame. `http://www.openrdf.org/documentation.jsp`.

[SGK+08] Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store Support for RDF Data Management: not all swans are white. In *VLDB*, pages 1553–1563, 2008.

[SHK+08] Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, and Christoph Pinkel. An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In *ISWC*, pages 82–97, 2008.

[SHLP08] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. Technical report, arXiv:0806.4627 cs.DB, 2008. `http://arxiv.org/abs/0806.4627`.

[SHLP09] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *ISWC*, 2009.

[SHM+09] Michael Schmidt, Thomas Hornung, Michael Meier, Christoph Pinkel, and Georg Lausen. SP^{2}Bench: A SPARQL Performance Benchmark. In *Semantic Web Information Management: a model based perspective*. Springer, 2009.

[Siz07] Sergej Sizov. What Makes You Think That? The Semantic Web's Proof Layer. *IEEE Intelligent Systems*, 22(6):94–99, 2007.

[SKCT05] Giorgos Serfiotis, Ioanna Koffina, Vassilis Christophides, and Val Tannen. Containment and Minimization of RDF/S Query Patterns. In *ISWC*, pages 607–623, 2005.

[SML08] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. Technical report, arXiv:0812.3788 cs.DB, 2008. `http://arxiv.org/abs/0812.3788`.

[SO87] Sreekumar T. Shenoy and Zehra M. Ozsoyoglu. A System for Semantic Query Optimization. In *SIGMOD*, pages 181–195, 1987.

[SOO99] Lei Sheng, Zehra M. Ozsoyoglu, and Gultekin Ozsoyoglu. A Graph Query Language and Its Query Processing. In *ICDE*, pages 572–581, 1999.

[spaa] SPARQL New Features and Rationale. W3C Working Draft, 2 July 2009. `http://www.w3.org/TR/2009/WD-sparql-features-20090702/`.

[spab] SPARQL Protocol for RDF. W3C Recommendation, 15 January 2008. `http://www.w3.org/TR/rdf-sparql-protocol/`.

[spac] SPARQL Query Language for RDF. W3C Recommendation, 15 January 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[SS86] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques*. MIT Press, 1986.

[SSB+08] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *WWW*, pages 595–604, 2008.

[Sto76] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3:1–22, 1976.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, 2002.

[swr] SWRC – Semantic Web Research Community Ontology. `http://ontoware.org/projects/swrc/`.

[sww] W3C Semantic Web Activity. `http://www.w3org/2001/sw/`.

[Tau] Joshua Tauberer. U.S. Census RDF Data. `http://www.rdfabout.com/demo/census/`.

[TCK05] Yannis Theoharis, Vassilis Christophides, and Grigoris Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *ISWC*, pages 685–701, 2005.

[Tod75] Stephen Todd. PRTV: An Efficient Implementation for Large Relational Data Bases. In *VLDB*, pages 554–556, 1975.

[tpc] TPC. `http://www.tpc.org`.

[TTKC08] Yannis Theoharis, Yannis Tzitzikas, Dimitris Kotzinos, and Vassilis Christophides. On Graph Features of Semantic Web Schemas. *IEEE Trans. Knowl. Data Eng.*, 20(5):692–702, 2008.

[tur] Turtle – Terse RDF Triple Language. W3C Team Submission, 14 January 2008. `http://www.w3.org/TeamSubmission/turtle/`.

[Var82] Moshe Y. Vardi. The Complexity of Relational Query Languages (Extended Abstract). In *STOC*, pages 137–146, 1982.

[W3Ca] W3C. The Semantic Web layer cake. `http://www.w3.org/2007/03/layerCake.png`.

[w3cb] Internationalized Resource Identifiers (IRIs). RFC 3987, January 2005. `http://tools.ietf.org/html/rfc3987`.

[w3cc] Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, January 2005. `http://tools.ietf.org/html/rfc3986`.

[w3cd] Universal Resource Identifiers in WWW. RFC 1630, June 1994. `http://tools.ietf.org/html/rfc1630`.

[w3ce] Extensible Markup Language (XML). `http://www.w3.org/XML/`.

[WGQH05] Sui-Yu Wang, Yuanbo Guo, Abir Qasem, and Jeff Heflin. Rapid Benchmarking for Semantic Web Knowledge Base Systems. In *ISWC*, pages 758–772, 2005.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *VLDB*, pages 1008–1019, 2008.

[xml] XML Schema Part 0: Primer Second Edition. W3C Recommendation, 28 October 2004. `http://www.w3.org/TR/xmlschema-0`.

[xpa] XML Path Language (XPath) 2.0. W3C Recommendation, 23 January 2007. `http://www.w3.org/TR/xpath20/`.

[xqu] XQuery 1.0: An XML Query Language. W3C Recommendation, 23 January 2007. `http://www.w3.org/TR/xquery/`.

[zun] ZunZun Curve Fitting and Surface Fitting. `http://zunzun.com/`.

Appendix A.

Survey of Namespaces

Prefix	Description
`bench`	*Shortcut for* `http://localhost/vocabulary/bench/` User-defined namespace used in SP^{2}Bench (cf. Chapter 6).
`dc`	*Shortcut for* `http://purl.org/dc/elements/1.1/` Namespace of the Dublin Core Metadata project [dub], used for describing bibliographic entities.
`dcterms`	*Shortcut for* `http://purl.org/dc/terms/` Namespace of the Dublin Core Metadata project [dub], used for describing bibliographic entities.
`foaf`	*Shortcut for* `http://xmlns.com/foaf/0.1/` Namespace of the Friend-of-a-Friend project [foa], which provides vocabulary for describing persons.
`myns`	*Shortcut for* `http://my.namespace.com#` User-defined namespace used in various examples throughout the thesis.
`person`	*Shortcut for* `http://localhost/persons/` User-defined namespace used in SP^{2}Bench (cf. Chapter 6).
`rdf`	*Shortcut for* `http://www.w3.org/1999/02/22-rdf-syntax-ns#` Standard RDF [rdfa] namespace.
`rdfs`	*Shortcut for* `http://www.w3.org/2000/01/rdf-schema#` Standard RDFS [rdfc] namespace.
`swrc`	*Shortcut for* `http://swrc.ontoware.org/ontology#` Namespace of the SWRC [swr] (Semantic Web for Research Communities) project for modeling entities of research communities.
`xsd`	*Shortcut for* `http://www.w3.org/2001/XMLSchema#` Contains common RDF datatype definitions, such as `xsd:integer`.

Figure A.1.: Survey of namespaces used in the thesis.

Appendix B.

Proofs of Complexity Results

B.1. Proof of Theorem 3.5

We start with a more general form of the QBF problem, which will be required later in the main proof of Theorem 3.5. The new version of QBF differs from the QBF versions used in the proofs of Theorems 3.3 and 3.4 (cf. Section 3.2.3) in that we relax the condition that the inner, quantifier-free part of the formula is in CNF. We call this generalized version QBF* and define it as follows.

QBF*: given a quantified boolean formula $\varphi := \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_m \exists y_m \psi$, as input, where ψ is a quantifier-free formula: is φ valid?

Lemma B.1 There is a polynomial-time reduction from QBF* to the SPARQL Evaluation problem for class $\mathcal{AFO}$.[1] □

Proof of Lemma B.1

The correctness of this lemma follows from the observations that (i) QBF* is known to be PSpace-complete (like QBF), (ii) the subfragment $\mathcal{AO} \subset AFO$ is PSpace-hard and (iii) the superfragment $\mathcal{E} \supset AFO$ is contained in PSpace. Thus, fragment $\mathcal{AFO}$ also is PSpace-complete, which implies the existence of a reduction.

We are, however, interested in some specific properties of the reduction, so we will shortly sketch the construction. We restrict ourselves on showing how to encode the inner, quantifier-free boolean formula φ (which is not necessarily in CNF) using operators And and Filter. The second part of the reduction, namely the encoding of the quantifier sequence, is the same as in the proof of Theorem 3.3.

Let us start with a quantified boolean formula of the form

$$\varphi := \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_m \exists y_m \psi,$$

[1] The same result was proven in [PAG09]. This lemma, however, was developed independently from [PAG09]. We informally published it already several months earlier in [SML08].

where ψ is a quantifier-free boolean formula. We assume w.l.o.g. that ψ is constructed using the boolean connectives $\wedge$, $\vee$ and $\neg$. By $V_\psi := \{v_1, \ldots, v_n\}$ we denote the set of boolean variables in formula ψ. We fix the database

$$D := \{(a, \mathit{false}, 0), (a, \mathit{true}, 1), (a, \mathit{tv}, 0), (a, \mathit{tv}, 1)\}$$

and encode the formula ψ as

$$P_\psi := ((a, \mathit{tv}, ?V_1) \text{ AND } \ldots \text{ AND } (a, \mathit{tv}, ?V_n)) \text{ FILTER } f(\psi),$$

where $?V_1, \ldots, ?V_n$ represent the boolean variables $v_1, \ldots, v_n$ and function $f(\psi)$ generates a SPARQL filter condition that precisely mirrors the boolean formula ψ. Formally, function $f(\psi)$ is defined inductively on the structure of ψ as follows.

$$\begin{array}{ll} f(v_i) & := ?V_i = 1 \\ f(\psi_1 \wedge \psi_2) & := f(\psi_1) \wedge f(\psi_2) \\ f(\psi_1 \vee \psi_2) & := f(\psi_1) \vee f(\psi_2) \\ f(\neg\psi_1) & := \neg\, f(\psi_1) \end{array}$$

In the expression P_ψ, the AND-block generates all possible valuations for the variables in ψ, while the FILTER-expression retains exactly those valuations that satisfy formula ψ. It is straightforward to verify that ψ is satisfiable iff there is a mapping $\mu \in [\![P_\psi]\!]_D$. Even more, for each $\mu \in [\![P_\psi]\!]_D$ the truth assignment ρ_μ defined as $\rho_\mu(v) := \mu(?V)$ for all variables $v \in V_\psi$ satisfies the formula ψ and, vice versa, for each truth assignment ρ that satisfies ψ there is a mapping $\mu \in [\![P_\psi]\!]_D$ that defines ρ. The rest of the proof (i.e., the encoding of the surrounding quantifier sequence) is the same as in the proof of Theorem 3.3. Ultimately, this gives us a SPARQL expression P_φ (which contains P_ψ above as a subexpression) such that the formula φ is valid if and only the mapping $\mu := \{B_0 \mapsto 1\}$ is contained in $[\![P_\varphi]\!]_D$. □

The next lemma follows from the construction in the previous lemma:

Lemma B.2 Let

$$D := \{(a, \mathit{false}, 0), (a, \mathit{true}, 1), (a, \mathit{tv}, 0), (a, \mathit{tv}, 1)\}$$

be an RDF database and $\varphi := \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m \psi$ ($m \geq 1$) be a quantified boolean formula, where ψ is quantifier-free. There is an encoding $\mathit{enc}(\varphi)$ such that

1. $\mathit{enc}(\varphi) \in \mathcal{E}_{\leq 2m}$,
2. φ is valid iff $\{?B_0 \mapsto 1\} \in [\![\mathit{enc}(\varphi)]\!]_D$, and
3. φ is invalid iff for each $\mu \in [\![\mathit{enc}(\varphi)]\!]_D$ it holds that $\mu \supseteq \{?B_0 \mapsto 1, ?A_1 \mapsto 1\}$. □

Proof of Lemma B.2

We argue that expression P_φ from the proof of Lemma B.1 is an encoding that satisfies all three conditions (observe that the database D defined in Lemma B.2 corresponds to the database D used in the proof of Lemma B.1). To prove Lemma B.2, we thus set $enc(\varphi) := P_\varphi$ and verify the claims one by one.

Lemma B.2(1): It is easy to see that the OPT-rank of P_φ is exactly $2m$: the subexpression P_ψ presented in the proof of Lemma B.1 is OPT-free and the surrounding construction that simulates the quantifier alternation requires a nesting of $2m$ OPT expressions (see the proof of Theorem 3.3). Hence, $P_\varphi =: enc(\varphi) \in \mathcal{E}_{\leq 2m}$.

Lemma B.2(2): Follows directly from the proof of Lemma B.1.

Lemma B.2(3): We consider the expression P_φ defined in Theorem 3.3:

$$\begin{aligned} P_\varphi := (a, \mathit{true}, ?B_0) \ \textsc{Opt}\ (P_1 \ \textsc{Opt}\ (Q_1 \\ \textsc{Opt}\ (P_2 \ \textsc{Opt}\ (Q_2 \\ \dots \\ \textsc{Opt}\ (P_m \ \textsc{Opt}\ (Q_m \ \textsc{And}\ P_\psi)) \dots)))), \end{aligned}$$

where the subexpression P_ψ denotes the $\mathcal{AF}$ encoding for ψ from the proof of Lemma B.1. First consider subexpression P_1, which is defined as

$$\begin{aligned} P_1 &:= ((a, tv, ?X_1) \ \textsc{And}\ (a, \mathit{false}, ?A_0) \ \textsc{And}\ (a, \mathit{true}, ?A_1)), \text{ so we have} \\ [\![P_1]\!]_D &= \{\{?X_1 \mapsto 0, ?A_0 \mapsto 0, ?A_1 \mapsto 1\}, \{?X_1 \mapsto 1, ?A_0 \mapsto 0, ?A_1 \mapsto 1\}\}. \end{aligned}$$

Further, $[\![(a, \mathit{true}, ?B_0)]\!]_D = \{\{?B_0 \mapsto 1\}\}$. We now study the subexpression

$$\begin{aligned} P' := P_1 \ \textsc{Opt}\ (Q_1 \\ \textsc{Opt}\ (P_2 \ \textsc{Opt}\ (Q_2 \\ \dots \\ \textsc{Opt}\ (P_m \ \textsc{Opt}\ (Q_m \ \textsc{And}\ P_\psi)) \dots))), \end{aligned}$$

of P_φ. Recall that $m \geq 1$ by assumption. It is easy to see that, by semantics of OPT and the previous evaluation result for $[\![P_1]\!]_D$, in each mapping $\mu' \in [\![P']\!]_D$, variable $?A_1$ is bound to 1. Furthermore, $[\![P']\!]_D$ contains at least two mappings, because the OPT operator either joins the two mappings from $[\![P_1]\!]_D$ with others or retains them without joining. Finally, consider the evaluation of expression P_φ. We can assume that there is at least one mapping $\mu' \in [\![P']\!]_D$ s.t. $\mu' \sim \{?B_0 \mapsto 1\}$ (otherwise $\{?B_0 \mapsto 1\} \in [\![P_\varphi]\!]_D$, which implies that φ is valid and the third case of Lemma B.1 would not apply). Consequently, each mapping $\mu \in [\![P_\varphi]\!]_D$ is of the form $\mu := \{?B_0 \mapsto 1\} \cup \mu'$, where $\mu' \in [\![P']\!]_D$. From the previous argumentation it follows that μ maps both variable $?B_0$ and $?A_1$ to 1. This completes the proof. □

Lemma B.3 Let P_1 and P_2 be SPARQL expressions for which the evaluation problem is in Σ_i^P, $i \geq 1$, and let R be a filter condition. The following claims hold.

1. The EVALUATION problem for the expression P_1 UNION P_2 is in Σ_i^P.
2. The EVALUATION problem for the expression P_1 AND P_2 is in Σ_i^P.
3. The EVALUATION problem for the expression P_1 FILTER R is in Σ_i^P. □

Proof of Lemma B.3

Lemma B.3(1): According to the semantics we have that $\mu \in [\![P_1 \text{ UNION } P_2]\!]_D$ if and only if $\mu \in [\![P_1]\!]_D$ or $\mu \in [\![P_2]\!]_D$. By assumption, both conditions can be checked individually by a Σ_i^P-algorithm, and so both can be checked in sequence in Σ_i^P.

Lemma B.3(2): It is easy to see that $\mu \in [\![P_1 \text{ AND } P_2]\!]_D$ iff μ can be decomposed into two mappings $\mu_1 \sim \mu_2$ such that $\mu = \mu_1 \cup \mu_2$ and $\mu_1 \in [\![P_1]\!]_D$ and $\mu_2 \in [\![P_2]\!]_D$. By assumption, both testing $\mu_1 \in [\![P_1]\!]_D$ and $\mu_2 \in [\![P_2]\!]_D$ is in Σ_i^P. Since $i \geq 1$, we have that $\Sigma_i^P \supseteq \Sigma_1^P = \text{NP}$. Hence, we can guess a decomposition $\mu = \mu_1 \cup \mu_2$ and check the two conditions one after the other. The whole procedure is in Σ_i^P.

Lemma B.3(3): The condition $\mu \in [\![P_1 \text{ FILTER } R]\!]_D$ holds iff $\mu \in [\![P_1]\!]_D$ (which can be tested in Σ_i^P by assumption) and R satisfies μ (which can be tested in polynomial time). We have that $\Sigma_i^P \supseteq \text{NP} \supseteq \text{PTIME}$ for $i \geq 1$, so the algorithm is in Σ_i^P. □

Proof of Theorem 3.5

We are now ready to tackle Theorem 3.5. The completeness proof divides into two parts, namely hardness and membership. We start with the hardness part, which is a reduction from QBF_n, a variant of the QBF problem used in previous proofs where the number n of quantifier alternations is fixed. We formally define QBF_n:

QBF_n: given a quantified boolean formula $\varphi := \exists x_1 \forall x_2 \exists x_3 \ldots Q x_n \psi$ as input, where ψ is a quantifier-free formula, $Q := \exists$ if n is odd, and $Q := \forall$ if n is even: is the formula φ valid?

It is known that QBF_n is Σ_n^P-complete for $n \geq 1$ (see e.g. [Pap94]).

(Hardness) Recall that our goal is to show that fragment $\mathcal{E}_{\leq n}$ is $\Sigma_{\leq n+1}^P$-hard. To prove this claim, we present a reduction from QBF_{n+1} to the EVALUATION problem for class $\mathcal{E}_{\leq n}$, i.e. we encode a quantified boolean formula with $n+1$ quantifier alternations by an $\mathcal{E}$ expression with OPT-rank $\leq n$. We distinguish two cases.

(1) Let $Q := \exists$, so the quantified boolean formula is of the form

$$\varphi := \exists y_0 \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m \psi.$$

Formula φ has $2m+1$ quantifier alternations, so we need to find an $\mathcal{E}_{\leq 2m}$ encoding for this expressions. We rewrite φ into an equivalent formula $\varphi := \varphi_1 \vee \varphi_2$, where

$$\varphi_1 := \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m (\psi \wedge y_0),$$
$$\varphi_2 := \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m (\psi \wedge \neg y_0).$$

According to Lemma B.2 there is a fixed document D and $\mathcal{E}_{\leq 2m}$ encodings $enc(\varphi_1)$ and $enc(\varphi_2)$ (for φ_1 and φ_2, respectively) s.t. $[\![enc(\varphi_1)]\!]_D$ (resp. $[\![enc(\varphi_2)]\!]_D$) contains the mapping $\mu := \{?B_0 \mapsto 1\}$ iff φ_1 (resp. φ_2) is valid. It is easy to see that the expression $enc(\varphi) := enc(\varphi_1)$ UNION $enc(\varphi_2)$ contains μ iff φ_1 or φ_2 is valid, i.e. iff $\varphi := \varphi_1 \vee \varphi_2$ is valid. Given that $enc(\varphi_1)$ and $enc(\varphi_2)$ are $\mathcal{E}_{\leq 2m}$ expressions, it follows that $enc(\varphi) := enc(\varphi_1)$ UNION $enc(\varphi_2)$ is in $\mathcal{E}_{\leq 2m}$, which completes part (1).

(2) Let $Q := \forall$, so the quantified boolean formula is of the form

$$\varphi := \exists x_0 \forall y_0 \exists x_1 \forall x_1 \ldots \exists x_m \forall y_m \psi.$$

φ has $2m+2$ quantifier alternations, so we need to find a reduction to the $\mathcal{E}_{\leq 2m+1}$ fragment. We eliminate the outer $\exists$-quantifier by rewriting φ as $\varphi := \varphi_1 \vee \varphi_2$, where

$$\varphi_1 := \forall y_0 \exists x_1 \forall y_1 \ldots \exists x_m \forall y_m (\psi \wedge y_0),$$
$$\varphi_2 := \forall y_0 \exists x_1 \forall y_1 \ldots \exists x_m \forall y_m (\psi \wedge \neg y_0).$$

Abstracting from the details of the inner formula, both φ_1 and φ_2 are of the form

$$\varphi' := \forall y_0 \exists x_1 \forall y_1 \ldots \exists x_m \forall y_m \psi',$$

where ψ' is a quantifier-free boolean formula. We now proceed as follows: we show (*) how to encode φ' by an $\mathcal{E}_{\leq 2m+1}$ expression $enc(\varphi')$ that, when evaluated on a fixed document D, yields a fixed mapping μ exactly if φ' is valid. This is sufficient, because then expression $enc(\varphi_1)$ UNION $enc(\varphi_2)$ is an $\mathcal{E}_{\leq 2m+1}$ encoding that contains μ exactly if the original formula $\varphi := \varphi_1 \vee \varphi_2$ is valid (analogously to the argumentation in case (1) of the proof). We first rewrite φ':

$$\begin{aligned}
\varphi' &:= \forall y_0 \exists x_1 \forall y_1 \ldots \exists x_m \forall y_m \psi' \\
&= \neg \exists y_0 \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m \neg \psi' \\
&= \neg(\varphi'_1 \vee \varphi'_2), \text{ where} \\
\varphi'_1 &:= \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m (\neg \psi' \wedge y_0), \\
\varphi'_2 &:= \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m (\neg \psi' \wedge \neg y_0).
\end{aligned}$$

According to Lemma B.2, each φ'_i can be encoded by an $\mathcal{E}_{\leq 2m}$ expressions $enc(\varphi'_i)$ such that, on the fixed database D given there, (1) $\mu := \{?B_0 \mapsto 1\} \in [\![\varphi'_i]\!]_D$ iff φ'_i is valid and (2) if φ'_i is not valid, then all mappings $[\![enc(\varphi'_i)]\!]_D$ bind both variable $?A_1$ and $?B_0$ to 1. It follows that (1') $\mu \in enc(\varphi'_1)$ UNION $enc(\varphi'_2)$ iff $\varphi'_1 \vee \varphi'_2$ and (2') all mappings $\mu \in enc(\varphi'_1)$ UNION $enc(\varphi'_2)$ bind both $?A_1$ and $?B_0$ to 1 iff $\neg(\varphi'_1 \vee \varphi'_2)$. Now consider the expression $Q := (a, \mathit{false}, ?A_1)$ OPT $(enc(\varphi'_1)$ UNION $enc(\varphi'_2))$. From claims (1') and (2') it follows that $\mu' := \{?A_1 \mapsto 0\} \in [\![Q]\!]_D$ iff $\neg(\varphi'_1 \vee \varphi'_2)$. Now recall that $\varphi' = \neg(\varphi'_1 \vee \varphi'_2)$, hence $\mu' \in [\![Q]\!]_D$ iff φ' is valid. We know that both $enc(\varphi'_1)$ and $enc(\varphi'_2)$ are $\mathcal{E}_{\leq 2m}$ expressions, so $Q \in \mathcal{E}_{\leq 2m+1}$. This implies that claim (*) holds and completes the hardness part of the proof.

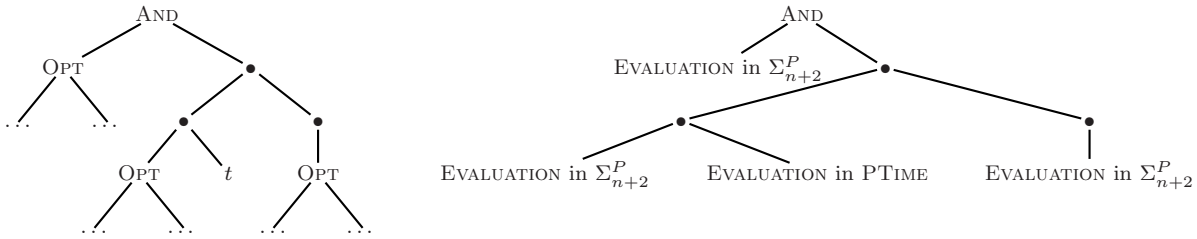

Figure B.1.: (a) AND-expression with increased OPT-rank; (b) Associated complexity classes for Opt-subexpressions and triple patterns.

(Membership) We next prove membership of $\mathcal{E}_{\leq n}$ expressions in Σ^P_{n+1} by induction on the Opt-rank. Let us assume that for each $\mathcal{E}_{\leq n}$ expression ($n \in \mathbb{N}_0$) Evaluation is in Σ^P_{n+1}. As stated in Theorem 3.1(2), Evaluation is Σ^P_1 = NP-complete for Opt-free expressions (i.e., $\mathcal{E}_{\leq 0}$), so the hypothesis holds for the basic case. In the induction step we increase the Opt-rank from n to $n+1$ and show that, for the resulting $\mathcal{E}_{\leq n+1}$ expression, the Evaluation problem can be solved in Σ^P_{n+2}. We consider an expression Q with $rank(Q) := n+1$ and distinguish four cases.

(1) Assume that $Q := P_1$ Opt P_2. By assumption, $Q \in \mathcal{E}_{\leq n+1}$ and from the definition of the Opt-rank (cf. Definition 3.2) it follows that both P_1 and P_2 are in $\mathcal{E}_{\leq n}$. Hence, by induction hypothesis, both P_1 and P_2 can be evaluated in Σ^P_{n+1}. By semantics, we have that $[\![P_1 \text{ Opt } P_2]\!]_D = [\![P_1 \text{ And } P_2]\!]_D \cup ([\![P_1]\!]_D \setminus [\![P_2]\!]_D)$, so it holds that $\mu \in [\![P_1 \text{ Opt } P_2]\!]_D$ iff it is generated by (i) $[\![P_1 \text{ And } P_2]\!]_D$ or generated by (ii) $[\![P_1]\!]_D \setminus [\![P_2]\!]_D$. According to Lemma B.3(2), condition (i) can be checked in Σ^P_{n+1}. The more interesting part is to check if (ii) holds. Applying the semantics of operator $\setminus$, this check can be formulated as $C := C_1 \wedge C_2$, where $C_1 := \mu \in [\![P_1]\!]_D$ and $C_2 := \neg\exists \mu' \in [\![P_2]\!]_D : \mu \sim \mu'$. By induction hypothesis, C_1 can be checked in Σ^P_{n+1}. We now argue that $\neg C_2 = \exists \mu' \in [\![P_2]\!]_D : \mu \sim \mu'$ can be checked in Σ^P_{n+1}: we can guess a mapping μ' (because $\Sigma^P_{n+1} \supseteq$ NP) and then check if $\mu \in [\![P_2]\!]_D$ (which, by application of the induction hypothesis, can be done by a Σ^P_{n+1}-algorithm), and test if μ and μ' are compatible (in polynomial time). Checking the inverse problem, i.e. if C_2 holds, is then possible in $co\Sigma^P_{n+1} = \Pi^P_{n+1}$. Summarizing cases (i) and (ii) we observe that (i) Σ^P_{n+1} and (ii) Π^P_{n+1} are both contained in Σ^P_{n+2}, so the two checks in sequence can be performed in Σ^P_{n+2}. This completes case (1).

(2) Assume that $Q := P_1$ And P_2. Figure B.1(a) shows the structure of a sample And expression, where the $\bullet$ symbols represent non-Opt operators (i.e. And, Union, or Filter), and t stands for triple patterns. Expression Q has an arbitrary number of Opt subexpressions (which might, of course, contain Opt subexpressions themselves). Each of these subexpressions has Opt-rank $\leq n+1$. Using the same argumentation as in case (1), the evaluation problem for all of them is in Σ^P_{n+2}. Further, each leaf node of the tree carries a triple pattern, which can be evaluated in

PTime $\subseteq \Sigma^P_{n+2}$. Figure B.1(b) illustrates the tree that is obtained when replacing all Opt-expressions and triple patterns by the complexity of their Evaluation problem. This simplified tree is now Opt-free, i.e. carries only operators And, Union, and Filter. We then proceed as follows. We apply Lemma B.3(1)-(3) repeatedly, folding the remaining And, Union, and Filter subexpressions bottom up. The lemma guarantees that these folding operations do not increase the complexity class, so it follows that the Evaluation problem falls in Σ^P_{n+2} for the whole expression.

Cases (3) $Q := P_1$ Union P_2 and (4) $Q := P_1$ Filter R are analogical to (2). □

Appendix C.

Proofs of Algebraic Optimization Results

C.1. Proof of Lemma 4.2

Proof of Lemma 4.2(1): Trivial (by counterexample).

Proof of Lemma 4.2(2): We provide counterexamples that rule out distributivity of operators $\bowtie$ and $\setminus$ over $\cup$, designed for the fixed database $D := \{(0, c, 1)\}$:

- The equivalence $A_1 \setminus (A_2 \cup A_3) \equiv (A_1 \setminus A_2) \cup (A_1 \setminus A_3)$ does **not** hold, as witnessed by $A_1 := [\![(0, c, ?x)]\!]_D$, $A_2 := [\![(?x, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?y)]\!]_D$.
- The equivalence $A_1 \bowtie (A_2 \cup A_3) \equiv (A_1 \bowtie A_2) \cup (A_1 \bowtie A_3)$ does **not** hold, as witnessed by $A_1 := [\![(0, c, ?x)]\!]_D$, $A_2 := [\![(?x, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?y)]\!]_D$.

Proof of Lemma 4.2(3): We provide counterexamples for all operator constellations that are listed in the lemma. Again, the counterexamples are designed for the database $D := \{(0, c, 1)\}$. We start with invalid distributivity rules over operator $\bowtie$:

- The equivalence $A_1 \cup (A_2 \bowtie A_3) \equiv (A_1 \cup A_2) \bowtie (A_1 \cup A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(?y, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?y)]\!]_D$.
- The equivalence $(A_1 \bowtie A_2) \cup A_3 \equiv (A_1 \cup A_3) \bowtie (A_2 \cup A_3)$ does **not** hold (the counterexample is symmetrical to the previous one).
- The equivalence $A_1 \setminus (A_2 \bowtie A_3) \equiv (A_1 \setminus A_2) \bowtie (A_1 \setminus A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(?y, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?y)]\!]_D$.
- The equivalence $(A_1 \bowtie A_2) \setminus A_3 \equiv (A_1 \setminus A_3) \bowtie (A_2 \setminus A_3)$ does **not** hold, as witnessed by $A_1 := [\![(0, c, ?x)]\!]_D$, $A_2 := [\![(0, c, ?y)]\!]_D$, and $A_3 := [\![(?x, c, 1)]\!]_D$.
- The equivalence $A_1 \bowtie (A_2 \bowtie A_3) \equiv (A_1 \bowtie A_2) \bowtie (A_1 \bowtie A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(?y, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?x)]\!]_D$.
- The equivalence $(A_1 \bowtie A_2) \bowtie A_3 \equiv (A_1 \bowtie A_3) \bowtie (A_2 \bowtie A_3)$ does **not** hold, as witnessed by $A_1 := [\![(0, c, ?x)]\!]_D$, $A_2 := [\![(0, c, ?y)]\!]_D$, and $A_3 := [\![(?x, c, 1)]\!]_D$.

Next, we provide counterexamples for distributivity rules over $\setminus$:

- The equivalence $A_1 \cup (A_2 \setminus A_3) \equiv (A_1 \cup A_2) \setminus (A_1 \cup A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(0, c, ?x)]\!]_D$, and $A_3 := [\![(?x, c, 1)]\!]_D$.
- The equivalence $(A_1 \setminus A_2) \cup A_3 \equiv (A_1 \cup A_3) \setminus (A_2 \cup A_3)$ does **not** hold (the counterexample is symmetrical to the previous one).
- The equivalence $A_1 \bowtie (A_2 \setminus A_3) \equiv (A_1 \bowtie A_2) \setminus (A_1 \bowtie A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(?y, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?x)]\!]_D$.
- The equivalence $(A_1 \setminus A_2) \bowtie A_3 \equiv (A_1 \bowtie A_3) \setminus (A_2 \bowtie A_3)$ does **not** hold (the counterexample is symmetrical to the previous one).
- The equivalence $A_1 \overline{\bowtie} (A_2 \setminus A_3) \equiv (A_1 \overline{\bowtie} A_2) \setminus (A_1 \overline{\bowtie} A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(?y, c, 1)]\!]_D$, and $A_3 := [\![(?y, c, 1)]\!]_D$.
- The equivalence $(A_1 \setminus A_2) \overline{\bowtie} A_3 \equiv (A_1 \overline{\bowtie} A_3) \setminus (A_2 \overline{\bowtie} A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(?y, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?y)]\!]_D$.

Finally, we provide counterexamples for invalid distributivity rules over $\overline{\bowtie}$:

- The equivalence $A_1 \cup (A_2 \overline{\bowtie} A_3) \equiv (A_1 \cup A_2) \overline{\bowtie} (A_1 \cup A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(c, c, c)]\!]_D$, and $A_3 := [\![(?y, c, 1)]\!]_D$.
- The equivalence $(A_1 \overline{\bowtie} A_2) \cup A_3 \equiv (A_1 \cup A_3) \overline{\bowtie} (A_2 \cup A_3)$ does **not** hold (the counterexample is symmetrical to the previous one).
- The equivalence $A_1 \bowtie (A_2 \overline{\bowtie} A_3) \equiv (A_1 \bowtie A_2) \overline{\bowtie} (A_1 \bowtie A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(?y, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?x)]\!]_D$.
- The equivalence $(A_1 \overline{\bowtie} A_2) \bowtie A_3 \equiv (A_1 \bowtie A_3) \overline{\bowtie} (A_2 \bowtie A_3)$ does **not** hold (the counterexample is symmetrical to the previous one).
- The equivalence $A_1 \setminus (A_2 \overline{\bowtie} A_3) \equiv (A_1 \setminus A_2) \overline{\bowtie} (A_1 \setminus A_3)$ does **not** hold, as witnessed by $A_1 = [\![(?x, c, 1)]\!]_D$, $A_2 = [\![(?y, c, 1)]\!]_D$, and $A_3[\![(0, c, ?x)]\!]_D$.
- The equivalence $(A_1 \overline{\bowtie} A_2) \setminus A_3 \equiv (A_1 \setminus A_3) \overline{\bowtie} (A_2 \setminus A_3)$ does **not** hold, as witnessed by $A_1 := [\![(?x, c, 1)]\!]_D$, $A_2 := [\![(?y, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?y)]\!]_D$.

The list of counterexamples is exhaustive. □

C.2. Proof of the Equivalences in Figure 4.3

We introduce some notation. Given a mapping μ and variable set $S \subseteq V$, we define the mapping $\mu_{|S}$ as the mapping obtained when projecting S in μ. To give an example, $\{?x \mapsto 1, ?y \mapsto 2\}_{|\{?x\}} = \{?x \mapsto 1\}$. Further, given two mappings μ_1, μ_2 and a variable $?x$ we say that μ_1 *and* μ_2 *agree on* $?x$ iff either it holds that $?x \in dom(\mu_1) \cap dom(\mu_2) \wedge \mu_1(?x) = \mu_2(?x)$ or $?x \notin dom(\mu_1) \cup dom(\mu_2)$.

(PBaseI). Follows from the definition of the projection operator (see Definition 2.10) and the observation that $pVars(A)$ extracts all variables that are potentially bound in any result mapping, as stated in Proposition 4.1.

(PBaseII). For each set of variables S^* it holds that $S = (S \cap S^*) \cup (S \setminus S^*)$, so we can rewrite the left side of the equation as $\pi_{(S \cap pVars(A)) \cup (S \setminus pVars(A))}(A)$. This

shows that, compared to the right side expression of the equation, the left side projection differs in that it additionally considers variables in $S \setminus pVars(A)$. However, as stated in Proposition 4.1, for each mapping μ that is generated by A we have that $dom(\mu) \subseteq pVars(A)$, so $S \setminus pVars(A)$ contains only variables that are unbound in each result mapping and thus can be dropped without changing the semantics.

(PUPush). The equivalence follows easily from the definition of the projection and the union operator (cf. Definition 2.10). We omit the details.

(PJPush). See Section 4.2.4.

(PMPush). $\Rightarrow$: Let $\mu \in \pi_S(A_1 \setminus A_2)$. By semantics, μ is obtained from some mapping $\mu_1 \in A_1$ that is incompatible with each mapping in A_2, by projecting S, i.e. $\mu = \mu_{1|S}$. We show that μ is also generated by the right side expression $\pi_S(\pi_{S'}(A_1) \setminus \pi_{S''}(A_2))$. First observe that $\pi_{S'}(A_1)$ generates a mapping $\mu_1' \subseteq \mu_1$ that agrees with μ_1 on all variables in S and also on all variables in $pVars(A_1) \cap pVars(A_2)$, because A_1 generates μ_1 and $S' := S \cup (pVars(A_1) \cap pVars(A_2))$. We distinguish two cases. (a) Assume that μ_1' is incompatible with each mapping generated by $\pi_{S''}(A_2)$. Then $\pi_{S'}(A_1) \setminus \pi_{S''}(A_2)$ generates μ_1' and, going one step further, the whole expression at the right side (i.e., including the outermost projection for S) generates $\mu_{1|S}'$. We know that μ_1' agrees with μ_1 on all variables in S, so $\mu_{1|S}' = \mu_{1|S} = \mu$ and the right side generates μ. (b) Assume there is a mapping $\mu_2' \in \pi_{S''}(A_2)$ that is compatible with μ_1', i.e. for all $?x \in dom(\mu_1') \cap dom(\mu_2')$: $\mu_1'(?x) = \mu_2'(?x)$. From before we know that $\mu_1 \supseteq \mu_1'$ and that μ_1 agrees with μ_1' on all variables in $pVars(A_1) \cap pVars(A_2)$. From $\mu_2' \in \pi_{S''}(A_2)$ it follows that there is a mapping $\mu_2 \in A_2$ such that $\mu_2 \supseteq \mu_2'$ and μ_2 agrees with μ_2' on all variables in $S'' := pVars(A_1) \cap pVars(A_2)$. Taking both observations together, we conclude that $\mu_1 \sim \mu_2$, because all shared variables in-between μ_1 and μ_2 are contained in $pVars(A_1) \cap pVars(A_2)$ and each of these variables either maps to the same value in μ_1 (μ_2) and μ_1' (μ_2') or, alternatively, is unbound in both. This is a contradiction to the initial claim that μ_1 is incompatible with each mapping in A_2 and we conclude that assumption (b) was invalid.

$\Leftarrow$: Assume that $\mu' \in \pi_S(\pi_{S'}(A_1) \setminus \pi_{S''}(A_2))$. We show that μ' is also generated by the left side of the equivalence. By semantics, μ' is obtained from a mapping $\mu_1' \in \pi_{S'}(A_1)$ by projecting S, i.e. $\mu' = \mu_{1|S}'$. where μ_1' is incompatible with each mapping in $\pi_{S''}(A_2)$. First observe that the left side subexpression A_1 generates a mapping $\mu_1 \supseteq \mu_1'$ that agrees with μ_1' on all variables in S'. From the observation that μ_1' is incompatible with each mapping in $\pi_{S''}(A_2)$ we conclude that also $\mu_1 \supseteq \mu_1'$ is incompatible with each mapping in A_2 (which contains only mappings of the form $\mu_2 \supseteq \mu_2'$ for some $\mu_2' \in \pi_{S''}(A_2)$). Hence, also the left side expression $A_1 \setminus A_2$ generates μ_1. From $\mu_1 \supseteq \mu_1'$ and the observation that μ_1 and μ_1' agree on all variables in S' we conclude that μ_1 and μ_1' also agree on the variables in $S \subseteq S'$. Consequently, $\mu_{1|S} = \mu_{1|S}' = \mu'$ and it follows that the left side expression generates mapping μ'.

(PLPush). The following rewriting proves the claim, where we use the shortcuts

$S' := S \cup (pVars(A_1) \cap pVars(A_2))$ and $S'' := pVars(A_1) \cap pVars(A_2)$.

$$
\begin{array}{ll}
\pi_S(A_1 \overline{\bowtie} A_2) & \\
= \pi_S((A_1 \bowtie A_2) \cup (A_1 \setminus A_2)) & \text{[semantics]} \\
= \pi_S(A_1 \bowtie A_2) \cup \pi_S(A_1 \setminus A_2) & \text{[\textit{(PUPush)}]} \\
= \pi_S(\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)) \cup \pi_S(\pi_{S'}(A_1) \setminus \pi_{S''}(A_2)) & \text{[\textit{(PJPush)},\textit{(PMPush)}]} \\
= \pi_S(\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)) \cup \pi_S(\pi_{S'}(A_1) \setminus \pi_{S'}(A_2)) & \text{[$(*)$]} \\
= \pi_S((\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)) \cup (\pi_{S'}(A_1) \setminus \pi_{S'}(A_2))) & \text{[\textit{(PUPush)}]} \\
= \pi_S(\pi_{S'}(A_1) \overline{\bowtie} \pi_{S'}(A_2)) & \text{[semantics]}
\end{array}
$$

Most interesting is step $(*)$, where we replace S'' by S' (all other steps are straightforward). This rewriting step is justified by the equivalence

$$\pi_S(\pi_{S'}(A_1) \setminus \pi_{S''}(A_2)) \equiv \pi_S(\pi_{S'}(A_1) \setminus \pi_{S'}(A_2)).$$

The idea behind this equivalence is the following. First note that S' can be written as $S' = S'' \cup (S \setminus (pVars(A_1) \cup pVars(A_2))$, which shows that S' and S'' differ only by variables that are contained in S but not in $pVars(A_1) \cap pVars(A_2)$. Intuitively, these variables are harmless because they cannot induce incompatibility between the A_1 and the A_2 part on either side of the equivalence, since they occur at most in one of both mapping sets.

(PFPush). Follows from the semantics of operator π and operator σ in Definition 2.11. The crucial observation is that filtering leaves mappings unchanged, and if we do not project away variables that are required to evaluate the filter (which is implicit by the equation), then the preprojection does not change the semantics.

(PMerge). The rule follows trivially from the definition of operator π. □

C.3. Proof of the Equivalences in Figure 4.4

(FDecompI). Follows from Lemma 1(1) in [PAG06a].

(FDecompII). Follows from Lemma 1(2) in [PAG06a].

(FReord). Follows from *(FDecompI)* and the commutativity of operator $\wedge$.

(FBndI). Follows from Proposition 4.2.

(FBndII). Follows from Proposition 4.1.

(FBndIII). Follows from Proposition 4.2.

(FBndIV). Follows from Proposition 4.1.

(FUPush). Follows from Proposition 1(5) in [PAG06a].

(FMPush). $\Rightarrow$: Let $\mu \in \sigma_R(A_1 \setminus A_2)$. By semantics, $\mu \in A_1$, there is no $\mu_2 \in A_2$ compatible with μ_1, and $\mu \models R$. From these preconditions it follows immediately that $\mu \in \sigma_R(A_1) \setminus A_2$. $\Leftarrow$: Let $\mu \in \sigma_R(A_1) \setminus A_2$. Then $\mu \in A_1$, $\mu \models R$, and there is no compatible mapping in A_2. Clearly, then also $\mu \in A_1 \setminus A_2$ and $\mu \in \sigma_R(A_1 \setminus A_2)$.

(FJPush). See Section 4.2.5.

(FLPush). We rewrite the expression schematically:

$$
\begin{array}{ll}
\sigma_R(A_1 \bowtie\!\!\!\!- A_2) & \\
= \sigma_R((A_1 \bowtie A_2) \cup (A_1 \setminus A_2)) & \text{[semantics]} \\
= \sigma_R(A_1 \bowtie A_2) \cup \sigma_R(A_1 \setminus A_2) & \text{[(FUPush)]} \\
= (\sigma_R(A_1) \bowtie A_2) \cup (\sigma_R(A_1) \setminus A_2) & \text{[(FJPush),(FMPush)]} \\
= \sigma_R(A_1) \bowtie\!\!\!\!- A_2 & \text{[semantics]}\ \square
\end{array}
$$

C.4. Proof of Lemma 4.3

(FElimI). We first introduce three functions $rem_{?x} : \mathcal{M} \mapsto \mathcal{M}$, $add_{?x \mapsto c} : \mathcal{M} \mapsto \mathcal{M}$, and $subst^{?y}_{?x} : \mathcal{M} \mapsto \mathcal{M}$, which manipulate mappings as follows.

- $rem_{?x}(\mu)$ removes $?x$ from μ (if it is bound), i.e. outputs mapping μ' such that $dom(\mu') := dom(\mu) \setminus \{?x\}$ and $\mu'(?y) := \mu(?y)$ for all $?y \in dom(\mu')$.
- $add_{?x \mapsto c}(\mu)$ binds variable $?x$ to c in μ, i.e. outputs mapping $\mu' := \mu \cup \{?x \mapsto c\}$ (we will apply this function only if $?x \notin dom(\mu)$, so μ' is defined).
- $subst^{?y}_{?x}(\mu) := rem_{?x}(add_{?y \mapsto \mu(?x)}(\mu))$ replaces variable $?x$ by $?y$ in μ (we will apply this function only if $?x \in dom(\mu)$ and $?y \notin dom(\mu)$).

We fix document D. To prove that *(FElimI)* holds, we show that, for every expression A built using operators $\bowtie$, $\cup$, and triple patterns $[\![t]\!]_D$ (i.e., expressions as defined in rule *(FElimI)*) the following five claims hold (abusing notation, we write $\mu \in A$ if μ is contained in the result of evaluating expression A on document D).

(C1) If $\mu \in A$, $dom(\mu) \supseteq \{?x, ?y\}$, and $\mu(?x) = \mu(?y)$ then $rem_{?x}(\mu) \in A^{?y}_{?x}$.
(C2) If $\mu \in A$ and $?x \notin dom(\mu)$ then $\mu \in A^{?y}_{?x}$.
(C3) If $\mu \in A$ and $?x \in dom(\mu)$, and $?y \notin dom(\mu)$ then $subst^{?y}_{?x}(\mu) \in A^{?y}_{?x}$.
(C4) If $\mu \in A^{?y}_{?x}$ and $?y \notin dom(\mu)$ then $\mu \in A$.
(C5) If $\mu \in A^{?y}_{?x}$ and $?y \in dom(\mu)$ then $\mu \in A$ or $add_{?x \mapsto \mu(?y)}(\mu) \in A$ or $subst^{?x}_{?y}(\mu) \in A$.

Before proving that these conditions hold for every expression A build using only operators $\bowtie$, $\cup$, and triple patterns $[\![t]\!]_D$, we argue that the above five claims in combination with the precondition $?x, ?y \in cVars(A)$ stated in Lemma 4.3 imply equivalence *(FElimI)*. $\Rightarrow$: Let $\mu \in \pi_{S \setminus \{?x\}}(\sigma_{?x=?y}(A))$. From the semantics of operators π

and σ it follows that μ is obtained from some $\mu' \supseteq \mu$ s.t. $\mu' \in A$, $?x, ?y \in dom(\mu')$, $\mu'(?x) = \mu'(?y)$, and $\pi_{S\setminus\{?x\}}(\{\mu'\}) = \{\mu\}$. Given all these prerequisites, condition (C1) implies that $\mu'' := rem_{?x}(\mu')$ is generated by $A\frac{?y}{?x}$. Observe that mapping μ'' agrees with μ' on all variables but $?x$. Hence, $\pi_{S\setminus\{?x\}}(\{\mu''\}) = \pi_{S\setminus\{?x\}}(\{\mu'\}) = \{\mu\}$, which shows that μ is generated by the right side expression $\pi_{S\setminus\{?x\}}(A\frac{?y}{?x})$. $\Leftarrow$: Consider a mapping $\mu \in \pi_{S\setminus\{?x\}}(A\frac{?y}{?x})$. Then there is some mapping $\mu' \in A\frac{?y}{?x}$ such that $\mu' \supseteq \mu$ and $\pi_{S\setminus\{?x\}}(\{\mu'\}) = \{\mu\}$. By assumption we have that $?x \in cVars(A)$ and it is easily verified that this implies $?y \in cVars(A\frac{?y}{?x})$. Hence, variable $?y$ is bound in μ' (according to Proposition 4.2). Condition (C5) now implies that (i) $\mu' \in A$, or (ii) $add_{?x\mapsto\mu'(?y)}(\mu') \in A$, or (iii) $subst_{\frac{?x}{?y}}(\mu') \in A$ holds. Concerning case (i) first observe that $?x \notin dom(\mu')$, since all occurrences of $?x$ have been replaced by $?y$ in $A\frac{?y}{?x}$. On the other hand, we know that $?x \in cVars(A) \rightarrow ?x \in dom(\mu')$, so we have a contradiction (i.e., assumption (i) was invalid). With similar argumentation, we obtain a contradiction for case (iii), because $?y \in cVars(A) \rightarrow ?y \in dom(\mu'')$ for all $\mu'' \in A$, but obviously $?y \notin dom(subst_{\frac{?x}{?y}}(\mu'))$. Therefore, given that condition (C5) is valid by assumption, we conclude that case (ii) $\mu'' := add_{?x\mapsto\mu'(?y)}(\mu') \in A$ must hold. Observe that $\mu''(?x) = \mu''(?y)$ by construction and that μ'' differs from μ' only by an additional binding for variable $?x$. Hence, μ'' passes the filter $\sigma_{?x=?y}$ in the left side expression and from $\pi_{S\setminus\{?x\}}(\{\mu''\}) = \pi_{S\setminus\{?x\}}(\{\mu'\}) = \{\mu\}$ we deduce that the left side expression $\pi_{S\setminus\{?x\}}(\sigma_{?x=?y}(A))$ generates μ.

Having shown that the five claims imply the equivalence, we now prove the claims by structural induction (over expressions built using operators $\bowtie$, $\cup$ and triple patterns of the form $[\![t]\!]_D$). We leave the basic case $A := [\![t]\!]_D$ as an exercise to the reader and assume that the induction hypothesis holds. In the induction step, we distinguish two cases. (1) Let $A := A_1 \bowtie A_2$. Consider a mapping $\mu \in A$. Then μ is of the form $\mu = \mu_1 \cup \mu_2$ where $\mu_1 \in A_1$ and $\mu_2 \in A_2$ are compatible mappings. Observe that $A\frac{?y}{?x} = A_1\frac{?y}{?x} \bowtie A_2\frac{?y}{?x}$. (1.1) To see why condition (C1) holds first note that by induction hypothesis conditions (C1)-(C3) hold for A_1, A_2. Further assume that $dom(\mu) \supseteq \{?x, ?y\}$, and $\mu(?x) = \mu(?y)$ (otherwise we are done). It is straightforward to verify that conditions (C1), (C2), and (C3) imply that $A_1\frac{?y}{?x} \bowtie A_2\frac{?x}{?y}$ generates $rem_{?x}(\mu)$: the claim follows when distinguishing several cases, covering the possible domains of μ_1 and μ_2, and applying the induction hypothesis; we omit the details. (1.2) To prove condition (C2) let us assume that $?x \notin dom(\mu)$. This implies that $?x \notin dom(\mu_1)$ and $?x \notin dom(\mu_2)$, so μ_1 and μ_2 are also generated by $A_1\frac{?y}{?x}$ and $A_2\frac{?y}{?x}$ (by induction hypothesis and claim (C2)). Hence, μ is generated by $A\frac{?y}{?x} = A_1\frac{?y}{?x} \bowtie A_2\frac{?y}{?x}$. (1.3) The proof that condition (C3) holds follows by application of the induction hypothesis and conditions (C2), (C3). (1.4) Claim (C4) can be shown by application of the induction hypothesis in combination with condition (C4). (1.5) Claim (C5) can be shown by application of the induction hypothesis and conditions (C4), (C5). (2) Let $A := A_1 \cup A_2$ and consequently $A\frac{?y}{?x} = A_1\frac{?y}{?x} \cup A_2\frac{?y}{?x}$. (2.1) Assume that $\mu \in A$, $dom(\mu) \supseteq \{?x, ?y\}$, and $\mu(?x) = \mu(?y)$. Then μ is gener-

ated by A_1 or by A_2. Let us w.l.o.g. assume that μ is generated by A_1. By induction hypothesis, $rem_{?x}(\mu)$ is generated by $A_1\frac{?y}{?x}$, and consequently also by $A\frac{?y}{?x}$. The proofs for the remaining conditions (C2)-(C5) proceed analogously.

(FElimII). Similar in idea to *(FElimI)*. □

C.5. Proof of Proposition 4.4

(MReord). We fix a mapping μ and show that it is contained in the left side expression if and only if it is contained in the right side expression. First observe that if μ is not contained in A_1, then it is neither contained in the right side nor in the left side of the expressions (both are subsets of A_1). So let us assume that $\mu \in A_1$. We distinguish three cases. Case (1): consider a mapping $\mu \in A_1$ and assume there is a compatible mapping in A_2. Then μ is not contained in $A_1 \setminus A_2$, and also not in $(A_1 \setminus A_2) \setminus A_3$, which by definition is a subset of the former. Now consider the right-hand side of the equation and let us assume that $\mu \in A_1 \setminus A_3$ (otherwise we are done). Then, given that there is a mapping in A_2 that is compatible to μ, the expression $(A_1 \setminus A_3) \setminus A_2$ will not contain μ. Case (2): The case of $\mu \in A_1$ being compatible with any mapping from A_3 is symmetrical to (2). Case (3): Let $\mu \in A_1$ be a mapping that is not compatible with any mapping in A_2 and A_3. Then both $(A_1 \setminus A_2) \setminus A_3$ on the left side and $(A_1 \setminus A_3) \setminus A_2$ on the right side contain μ. In all cases, μ is contained in the right side iff it is contained in the left side.

(MMUCorr). We show both directions of the equivalence. $\Rightarrow$: Let $\mu \in (A_1 \setminus A_2) \setminus A_3$. Then $\mu \in A_1$ and there is neither a compatible mapping $\mu_2 \in A_2$ nor a compatible mapping $\mu_3 \in A_3$. Then both A_2 and A_3 contain only incompatible mappings, and clearly $A_2 \cup A_3$ contains only incompatible mappings. Hence, the right side $A_1 \setminus (A_2 \cup A_3)$ produces μ. $\Leftarrow$: Let $\mu \in A_1 \setminus (A_2 \cup A_3)$. Then $\mu \in A_1$ and there is no compatible mapping in $A_2 \cup A_2$, which means that there is neither a compatible mapping in A_2 nor in A_3. It follows that $A_1 \setminus A_2$ generates μ (as there is no compatible mapping in A_2 and $\mu \in A_1$) and, going one step further, from the fact that there is no compatible mapping in A_3 we deduce that $\mu \in (A_1 \setminus A_2) \setminus A_3$.

(MJ). See Lemma 3(2) in [PAG06a].

$(\widetilde{LJ})$. Let $\widetilde{A_1}$, $\widetilde{A_2}$ be $\widetilde{\mathbb{A}}$-expressions. The following sequence of rewriting steps proves the equivalence.

$$
\begin{array}{ll}
\widetilde{A_1} \,\widetilde{\rtimes}\, \widetilde{A_2} & \\
= (\widetilde{A_1} \bowtie \widetilde{A_2}) \cup (\widetilde{A_1} \setminus \widetilde{A_2}) & \text{[by semantics]} \\
= (\widetilde{A_1} \bowtie (\widetilde{A_1} \bowtie \widetilde{A_2})) \cup (\widetilde{A_1} \setminus (\widetilde{A_1} \bowtie \widetilde{A_2})) & [(\widetilde{JIdem}),(JAss),(MJ)] \\
= (\widetilde{A_1} \,\widetilde{\rtimes}\, (\widetilde{A_1} \bowtie \widetilde{A_2})) & \text{[by semantics]} \ \square
\end{array}
$$

C.6. Proof of Lemma 4.4

(FLBndI). See Section 4.2.6.

(FLBndII). First recall that by assumption $?x \in cVars(A_2) \setminus pVars(A_1)$, which implies that $?x \notin pVars(A_1 \setminus A_2)$ and $?x \in cVars(A_1 \bowtie A_2)$. The following step-by-step rewriting proves the equivalence.

$$
\begin{aligned}
&\sigma_{bnd(?x)}(A_1 \mathbin{\overline{\bowtie}} A_2) \\
&= \sigma_{bnd(?x)}((A_1 \bowtie A_2) \cup (A_1 \setminus A_2)) && \text{[semantics]} \\
&= \sigma_{bnd(?x)}(A_1 \bowtie A_2) \cup \sigma_{bnd(?x)}(A_1 \setminus A_2) && \text{[\textit{(FUPush)}]} \\
&= \sigma_{bnd(?x)}(A_1 \bowtie A_2) \cup \emptyset && \text{[\textit{(FBndII)}]} \\
&= \sigma_{bnd(?x)}(A_1 \bowtie A_2) && \text{[semantics]} \\
&= A_1 \bowtie A_2 && \text{[\textit{(FBndI)}]}\ \square
\end{aligned}
$$

C.7. Proof of Lemma 4.9

Recall that by Lemma 4.5 the result of evaluating set and bag algebra expressions differs at most in the associated cardinality, so (given that the rules hold for SPARQL set algebra) in all cases it suffices to show that, for a fixed mapping μ that is contained in (by assumption both) the left and right side of the equivalence, the associated left and right side cardinalities for the mapping coincide. We fix document D. Further, given a SPARQL bag algebra expression A_i^+ with some index i, we denote by (Ω_i, m_i) the mapping multi-set obtained when evaluating A_i^+ on D.

In subsequent proofs we exploit the following well-known rewriting rules for sums.

Proposition C.1 (Sum Rewriting Rules, Folklore) Let a_x, b_x, denote expressions that depend on some x, λ be an expression that does not depend on x, and let C_x be a condition that depends on x. The following rewritings are valid.

(S1) $\sum_{x \in X} \lambda * a_x = \lambda * \sum_{x \in X} a_x$,

(S2) $\sum_{x \in \{x^* \in X | C_{x^*}\}} \sum_{y \in \{y^* \in Y | C_{y^*}\}} a_x * b_y = \sum_{(x,y) \in \{(x^*,y^*) \in (X,Y) | C_{x^*} \wedge C_{y^*}\}} a_x * b_y$,

(S3) $\sum_{x \in X} a_x + b_x = \sum_{x \in X} a_x + \sum_{x \in X} b_x$. $\square$

In the following, we shall refer to these equivalences as *(S1)*, *(S2)*, and *(S3)*.

(UAss$^+$). Let $A_1^+, A_2^+, A_3^+ \in \mathbb{A}^+$. We define expressions $A_l^+ := (A_1^+ \cup A_2^+) \cup A_3^+$ and $A_r^+ := A_1^+ \cup (A_2^+ \cup A_3^+)$. Consider a mapping μ that is contained both in the result of evaluating A_l^+ and A_r^+ on D. We apply the semantics of operator $\cup$ for multi-set expressions and rewrite the multiplicity that is associated with μ for A_l^+ step-by-step: $m_l(\mu) = (m_1(\mu)+m_2(\mu))+m_3(\mu) = m_1(\mu)+(m_2(\mu)+m_3(\mu)) = m_r(\mu)$.

($JAss^+$). Let $A_1^+, A_2^+, A_3^+ \in \mathbb{A}^+$. We define the shortcuts $A_l^+ := (A_1^+ \bowtie A_2^+) \bowtie A_3^+$, $A_r^+ := A_1^+ \bowtie (A_2^+ \bowtie A_3^+)$, $A_{1\bowtie 2}^+ := A_1^+ \bowtie A_2^+$, and $A_{2\bowtie 3}^+ := A_2^+ \bowtie A_3^+$. Consider a mapping μ that is contained in both the result of evaluating A_l^+ and A_r^+. We apply the semantics from Definition 2.14 and rewrite the left side multiplicity $m_l(\mu)$:

$$
\begin{aligned}
m_l(\mu) &= \textstyle\sum_{(\mu_{1\bowtie 2},\mu_3)\in\{(\mu^*_{1\bowtie 2},\mu^*_3)\in\Omega_{1\bowtie 2}\times\Omega_3 \mid \mu^*_{1\bowtie 2}\cup\mu^*_3=\mu\}} (m_{1\bowtie 2}(\mu_{1\bowtie 2}) * m_3(\mu_3)) \\
&= \textstyle\sum_{(\mu_{1\bowtie 2},\mu_3)\in\{(\mu^*_{1\bowtie 2},\mu^*_3)\in\Omega_{1\bowtie 2}\times\Omega_3 \mid \mu^*_{1\bowtie 2}\cup\mu^*_3=\mu\}} (\\
&\quad (\textstyle\sum_{(\mu_1,\mu_2)\in\{(\mu^*_1,\mu^*_2)\in\Omega_1\times\Omega_2 \mid \mu^*_1\cup\mu^*_2=\mu_{1\bowtie 2}\}} (m_1(\mu_1) * m_2(\mu_2))) * m_3(\mu_3)) \\
&\overset{(S1)}{=} \textstyle\sum_{(\mu_{1\bowtie 2},\mu_3)\in\{(\mu^*_{1\bowtie 2},\mu^*_3)\in\Omega_{1\bowtie 2}\times\Omega_3 \mid \mu^*_{1\bowtie 2}\cup\mu^*_3=\mu\}} (\\
&\quad \textstyle\sum_{(\mu_1,\mu_2)\in\{(\mu^*_1,\mu^*_2)\in\Omega_1\times\Omega_2 \mid \mu^*_1\cup\mu^*_2=\mu_{1\bowtie 2}\}} (m_1(\mu_1) * m_2(\mu_2) * m_3(\mu_3))) \\
&\overset{(S2)}{=} \textstyle\sum_{((\mu_{1\bowtie 2},\mu_3),(\mu_1,\mu_2))\in\{((\mu^*_{1\bowtie 2},\mu^*_3),(\mu^*_1,\mu^*_2))\in(\Omega_{1\bowtie 2}\times\Omega_3)\times(\Omega_1\times\Omega_2) \mid \mu^*_{1\bowtie 2}\cup\mu^*_3=\mu\wedge\mu^*_1\cup\mu^*_2=\mu^*_{1\bowtie 2}\}} \\
&\quad (m_1(\mu_1) * m_2(\mu_2) * m_3(\mu_3)) \\
&= \textstyle\sum_{(\mu_1,\mu_2,\mu_3)\in\{(\mu^*_1,\mu^*_2,\mu^*_3)\in\Omega_1\times\Omega_2\times\Omega_3 \mid \mu^*_1\cup\mu^*_2\cup\mu^*_3=\mu\}} (m_1(\mu_1) * m_2(\mu_2) * m_3(\mu_3)) \\
&= \textstyle\sum_{((\mu_1,\mu_{2\bowtie 3}),(\mu_2,\mu_3))\in\{((\mu^*_1,\mu^*_{2\bowtie 3}),(\mu^*_2,\mu^*_3))\in(\Omega_1\times\Omega_{2\bowtie 3})\times(\Omega_2\times\Omega_3) \mid \mu^*_1\cup\mu^*_{2\bowtie 3}=\mu\wedge\mu^*_2\cup\mu^*_3=\mu^*_{2\bowtie 3}\}} \\
&\quad (m_1(\mu_1) * m_2(\mu_2) * m_3(\mu_3)) \\
&\overset{(S2)}{=} \textstyle\sum_{(\mu_1,\mu_{2\bowtie 3})\in\{(\mu^*_1,\mu^*_{2\bowtie 3})\in\Omega_1\times\Omega_{2\bowtie 3} \mid \mu^*_1\cup\mu^*_{2\bowtie 3}=\mu\}} (\\
&\quad \textstyle\sum_{(\mu_2,\mu_3)\in\{(\mu^*_2,\mu^*_3)\in\Omega_2\times\Omega_3 \mid \mu^*_2\cup\mu^*_3=\mu_{2\bowtie 3}\}} (m_1(\mu_1) * m_2(\mu_2) * m_3(\mu_3))) \\
&\overset{(S1)}{=} \textstyle\sum_{(\mu_1,\mu_{2\bowtie 3})\in\{(\mu^*_1,\mu^*_{2\bowtie 3})\in\Omega_1\times\Omega_{2\bowtie 3} \mid \mu^*_1\cup\mu^*_{2\bowtie 3}=\mu\}} (\\
&\quad m_1(\mu_1) * \textstyle\sum_{(\mu_2,\mu_3)\in\{(\mu^*_2,\mu^*_3)\in\Omega_2\times\Omega_3 \mid \mu^*_2\cup\mu^*_3=\mu_{2\bowtie 3}\}} (m_2(\mu_2) * m_3(\mu_3))) \\
&= \textstyle\sum_{(\mu_1,\mu_{2\bowtie 3})\in\{(\mu^*_1,\mu^*_{2\bowtie 3})\in\Omega_1\times\Omega_{2\bowtie 3} \mid \mu^*_1\cup\mu^*_{2\bowtie 3}=\mu\}} (m_1(\mu_1) * m_{2\bowtie 3}(\mu_{2\bowtie 3})) \\
&= \mu_r(\mu)
\end{aligned}
$$

($UComm^+$). Let $A_1^+, A_2^+ \in \mathbb{A}^+$. Put $A_l^+ := A_1^+ \cup A_2^+$ and $A_r^+ := A_2^+ \cup A_1^+$. Consider a mapping μ that is contained in both the result of evaluating A_l^+ and A_r^+. We apply the semantics of operator $\cup$ (cf. Definition 2.14) and rewrite the left side multiplicity step-by-step: $m_l(\mu) = m_1(\mu) + m_2(\mu) = m_2(\mu) + m_1(\mu) = m_r(\mu)$.

($JComm^+$). Let $A_1^+, A_2^+ \in \mathbb{A}^+$. Put $A_l^+ := A_1^+ \bowtie A_2^+$, $A_r^+ := A_2^+ \bowtie A_1^+$. Consider a mapping μ that is contained in both the result of evaluating A_l^+ and A_r^+. Applying the semantics of operator $\bowtie$ (cf. Definition 2.14) we rewrite the left side multiplicity:

$$
\begin{aligned}
m_l(\mu) &= \textstyle\sum_{(\mu_1,\mu_2)\in\{(\mu^*_1,\mu^*_2)\in\Omega_1\times\Omega_2 \mid \mu^*_1\cup\mu^*_2=\mu\}} (m_1(\mu_1) * m_2(\mu_2)) \\
&= \textstyle\sum_{(\mu_2,\mu_1)\in\{(\mu^*_2,\mu^*_1)\in\Omega_2\times\Omega_1 \mid \mu^*_2\cup\mu^*_1=\mu\}} (m_2(\mu_2) * m_1(\mu_1)) \\
&= m_r(\mu)
\end{aligned}
$$

($JUDistR^+$). See Section 4.3.2.

($JUDistL^+$). Symmetrical to the proof of ($JUDistR^+$).

($MUDistR^+$). Let $A_1^+, A_2^+, A_3^+ \in \mathbb{A}^+$. We define expressions $A_l^+ := (A_1^+ \cup A_2^+) \setminus A_3^+$, $A_r^+ := (A_1^+ \setminus A_3^+) \cup (A_2^+ \setminus A_3^+)$, $A_{1\cup 2}^+ := A_1^+ \cup A_2^+$, $A_{1\setminus 3}^+ := A_1^+ \setminus A_3^+$, and $A_{2\setminus 3}^+ := A_2^+ \setminus A_3^+$.

Consider a mapping μ that is contained in the result of evaluating A_l^+ and A_r^+. It is easily verified that $m_l(\mu) = m_{1\cup 2}(\mu) = m_1(\mu)+m_2(\mu) = m_{1\setminus 3}(\mu)+m_{2\setminus 3}(\mu) = m_r(\mu)$.

($LUDistR^+$). Let $A_1^+, A_2^+, A_3^+ \in \mathbb{A}^+$. The following rewriting proves the claim.

$$
\begin{array}{ll}
(A_1^+ \cup A_2^+) \ltimes A_3^+ & \\
= ((A_1^+ \cup A_2^+) \bowtie A_3^+) \cup ((A_1^+ \cup A_2^+) \setminus A_3^+) & [\text{semantics}] \\
= ((A_1^+ \bowtie A_3^+) \cup (A_2^+ \bowtie A_3^+)) \cup ((A_1^+ \cup A_2^+) \setminus A_3^+) & [(JUDistR^+)] \\
= ((A_1^+ \bowtie A_3^+) \cup (A_2^+ \bowtie A_3^+)) \cup ((A_1^+ \setminus A_3^+) \cup (A_2^+ \setminus A_3^+)) & [(MUDistR^+)] \\
= ((A_1^+ \bowtie A_3^+) \cup (A_1^+ \setminus A_3^+)) \cup ((A_2^+ \bowtie A_3^+) \cup (A_2^+ \setminus A_3^+)) & [(UAss^+), (UComm^+)] \\
= (A_1^+ \ltimes A_3^+) \cup (A_2^+ \ltimes A_3^+) & [\text{semantics}] \square
\end{array}
$$

C.8. Proof of Lemma 4.10

As before in the proof of Lemma C.7 we exploit the assumption that by Lemma 4.5 the results of evaluating set and bag algebra expressions differs at most in the associated cardinality, so (given that the rules hold for SPARQL set algebra) in all cases it suffices to show that, for a fixed mapping μ that is contained in the left and right side of the equivalence, the associated left and right side cardinalities for the mapping coincide. We fix a document D. Further, given a SPARQL bag algebra expression A^+, we denote by (Ω, m) the mapping multi-set obtained when evaluating A^+. Similarly, for a bag algebra expression A_i^+ with some index i, we denote by (Ω_i, m_i) the mapping multi-set obtained when evaluating A_i^+. We shall again use the rewritings for sum expressions from Proposition C.1 (see Appendix C.7), which we denote by *(S1)*, *(S2)*, and *(S3)*, respectively.

($PBaseI^+$). Let $A^+ \in \mathbb{A}^+$ and $S \subset V$. Consider a mapping μ that is contained in the result of evaluating $A_l^+ := \pi_{pVars(A^+)\cup S}(A^+)$ and $A_r^+ := A^+$. We apply the semantics from Definition 2.14 and rewrite the multiplicity $m_l(\mu)$ step-by-step:

$$
\begin{array}{rl}
m_l(\mu) = & \sum_{\mu_+ \in \{\mu_+^* \in \Omega \mid \pi_{pVars(A^+)\cup S}(\{\mu_+^*\}) = \{\mu\}\}} m(\mu_+) \\
\overset{(*)}{=} & \sum_{\mu_+ \in \{\mu\}} m(\mu_+) \\
= & m(\mu) \\
= & m_r(\mu),
\end{array}
$$

where step $(*)$ follows from the observation that $\pi_{pVars(A^+)\cup S}(\{\mu_+^*\}) = \{\mu\}$ holds if and only if $\mu_+^* = \mu$ (this claim follows easily from Proposition 4.1, the definition of operator π, and the fact that $\mu \in \Omega_r = \Omega$ by assumption).

($PBaseII^+$). Let $A^+ \in \mathbb{A}^+$ and $S \subset V$. Consider a mapping μ that is contained in the result of evaluating $A_l^+ := \pi_S(A^+)$ and $A_r^+ := \pi_{S\cap pVars(A^+)}(A^+)$. We apply the semantics from Definition 2.14 and rewrite the (right side) multiplicity $m_r(\mu)$:

$$\begin{aligned} m_r(\mu) &= \sum_{\mu_+ \in \{\mu_+^* \in \Omega | \pi_{S \cap pVars(A^+)}(\{\mu_+^*\}) = \{\mu\}\}} m(\mu_+) \\ &\overset{(*)}{=} \sum_{\mu_+ \in \{\mu_+^* \in \Omega | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m(\mu_+) \\ &= m_l(\mu), \end{aligned}$$

where step $(*)$ follows by semantics and Proposition 4.1.

$(PUPush^+)$. Let $A_1^+, A_2^+ \in \mathbb{A}^+$ and $S \subset V$. Consider a mapping μ that is contained in the result of evaluating $A_l^+ := \pi_S(A_1^+ \cup A_2^+)$ and $A_r^+ := \pi_S(A_1^+) \cup \pi_S(A_2^+)$. Put $A_{1\cup 2}^+ := A_1^+ \cup A_2^+$, $A_{\pi 1}^+ := \pi_S(A_1^+)$, and $A_{\pi 2}^+ := \pi_S(A_2^+)$. We apply the semantics from Definition 2.14 and rewrite the multiplicity $m_l(\mu)$ step-by-step:

$$\begin{aligned} m_l(\mu) &= \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{1\cup 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_{1\cup 2}(\mu_+) \\ &= \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{1\cup 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} (m_1(\mu_+) + m_2(\mu_+)) \\ &\overset{(S3)}{=} (\sum_{\mu_+ \in \{\mu_+^* \in \Omega_{1\cup 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_1(\mu_+)) + \\ &\quad (\sum_{\mu_+ \in \{\mu_+^* \in \Omega_{1\cup 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_2(\mu_+)) \\ &\overset{(*)}{=} (\sum_{\mu_+ \in \{\mu_+^* \in \Omega_1 | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_1(\mu_+)) + (\sum_{\mu_+ \in \{\mu_+^* \in \Omega_2 | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_2(\mu_+)) \\ &= m_{\pi 1}(\mu) + m_{\pi 2}(\mu) \\ &= m_r(\mu), \end{aligned}$$

where step $(*)$ follows by semantics of operator $\cup$.

$(PJPush^+)$. To facilitate the proof, we establish the following proposition:

Proposition C.2 Let $A_1^+, A_2^+ \in \mathbb{A}^+$ and $S' \subset V$ with $S' \supseteq pVars(A_1^+) \cap pVars(A_2^+)$. Then the following equivalence holds.

$$\pi_{S'}(A_1^+ \bowtie A_2^+) \equiv \pi_{S'}(A_1^+) \bowtie \pi_{S'}(A_2^+) \qquad (FJPush2^+) \qquad \square$$

To see why Proposition C.2 implies *(FJPush)*, consider the latter equivalence, recall that $S' := S \cup (pVars(A_1^+) \cap pVars(A_2^+))$, and observe that by construction we have $S' \supseteq pVars(A_1^+) \cap pVars(A_2^+)$. We rewrite the left side of *(FJPush)*:

$$\begin{aligned} \pi_S(A_1^+ \bowtie A_2^+) &= \pi_S(\pi_{S'}(A_1^+ \bowtie A_2^+)) && [(PMerge^+)] \\ &= \pi_S(\pi_{S'}(A_1^+) \bowtie \pi_{S'}(A_2^+)) && [(FJPush2^+)] \end{aligned}$$

Given this rewriting, it remains to show that $(FJPush2^+)$ is valid. We split this proof into two parts. First, we show that the mapping sets coincide. To this end, we show that $(FJPush2^+)$ holds for SPARQL set algebra (the result carries over to bag algebra by Lemma 4.5). Let $A_1, A_2 \in \mathbb{A}$ and $S' \supseteq pVars(A_1) \cap pVars(A_2)$.

$\Rightarrow$: Consider a mapping μ generated by the left side expression $\pi_{S'}(A_1 \bowtie A_2)$. Then μ is obtained from some mapping $\mu' \supseteq \mu$ s.t. $\pi_{S'}(\{\mu'\}) = \{\mu\}$. Further, μ' is of the form $\mu_1' \cup \mu_2'$ where μ_1' and μ_2' are compatible mappings that are generated by

A_1 and A_2, respectively. We observe that the right side subexpressions $\pi_{S'}(A_1)$ and $\pi_{S'}(A_2)$ then generate mappings $\mu_1'' \subseteq \mu_1'$ and $\mu_2'' \subseteq \mu_2'$ that agree with μ_1' and μ_2' on all variables in S', respectively (where "agree" means that each such variable is either bound to the same value in the two mappings or unbound in both mappings). Clearly, $\mu_1'' \subseteq \mu_1' \wedge \mu_2'' \subseteq \mu_2' \wedge \mu_1' \sim \mu_2' \rightarrow \mu_1'' \sim \mu_2''$, so the right side expression generates the mapping $\mu'' := \mu_1'' \cup \mu_2''$. It is easily verified that (i) $dom(\mu'') \subseteq S'$ and that (ii) μ'' agrees with μ' on all variables in S'. This implies that $\mu'' = \mu$ and we conclude that μ is generated by the right side expression. $\Leftarrow$: Consider a mapping μ' that is generated by the right side expression $\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)$. Then μ' is of the form $\mu' = \mu_1' \cup \mu_2'$, where $\mu_1' \sim \mu_2'$ are generated by the subexpressions $\pi_{S'}(A_1)$ and $\pi_{S'}(A_2)$, respectively. Consequently, A_1 and A_2 generate mappings $\mu_1 \supseteq \mu_1'$ and $\mu_2 \supseteq \mu_2'$ such that μ_1 and μ_2 agree with μ_1' and μ_2' on all variables in S', respectively. We distinguish two cases. First, (i) if μ_1 and μ_2 are compatible then $\mu := \mu_1 \cup \mu_2$ agrees with μ' on all variables in S', and therefore $\pi_{S'}(\{\mu\}) = \mu'$, so the left side expression generates μ'. Second, (ii) if μ_1 and μ_2 are not compatible then there is $?x \in dom(\mu_1) \cap dom(\mu_2)$ such that $\mu_1(?x) \neq \mu_2(?x)$. From precondition $S' \supseteq pVars(A_1) \cap pVars(A_2)$ and Proposition 4.1 it follows that $?x \in S'$. We know that μ_1' and μ_2' agree with μ_1 and μ_2 on all variables in S'. It follows that $\mu_1'(?x) \neq \mu_2'(?x)$, which contradicts the assumption $\mu_1' \sim \mu_2'$.

Having shown that the mapping sets coincide, it remains to show that the left and right side multiplicities under bag semantics agree for each result mapping. We therefore switch to SPARQL bag algebra again. Let $A_1^+, A_2^+ \in \mathbb{A}^+$ and $S' \subset V$ such that $S' \supseteq pVars(A_1) \cap pVars(A_2)$ holds. We define $A_l^+ := \pi_{S'}(A_1^+ \bowtie A_2^+)$, $A_r^+ := \pi_{S'}(A_1^+) \bowtie \pi_{S'}(A_2^+)$, $A_{1\bowtie 2}^+ := A_1^+ \bowtie A_2^+$, $A_{\pi 1}^+ := \pi_{S'}(A_1^+)$, and $A_{\pi 2}^+ := \pi_{S'}(A_2^+)$. Using the notation introduced in the beginning of Appendix C.2, we write $\mu_{|S}$ for the mapping obtained when projecting S in μ (e.g. $\{?x \mapsto 1, ?y \mapsto 2\}_{|\{?x\}} = \{?x \mapsto 1\}$). Applying the semantics from Definition 2.14 we rewrite $m_l(\mu)$ schematically:

$$
\begin{aligned}
m_l(\mu) &= \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{1\bowtie 2} | \pi_{S'}(\{\mu_+^*\}) = \{\mu\}\}} m_{1\bowtie 2}(\mu_+) \\
&= \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{1\bowtie 2} | \pi_{S'}(\{\mu_+^*\}) = \{\mu\}\}} \\
&\quad \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in \Omega_1 \times \Omega_2 | \mu_1^* \cup \mu_2^* = \mu_+\}} (m_1(\mu_1) * m_2(\mu_2)) \\
&\overset{(S2)}{=} \sum_{(\mu_+,(\mu_1,\mu_2)) \in \{(\mu_+^*,(\mu_1^*,\mu_2^*)) \in \Omega_{1\bowtie 2} \times (\Omega_1 \times \Omega_2) | \pi_{S'}(\{\mu_+^*\}) = \{\mu\} \wedge \mu_1^* \cup \mu_2^* = \mu_+^*\}} \\
&\quad (m_1(\mu_1) * m_2(\mu_2)) \\
&= \sum_{(\mu_+,(\mu_1,\mu_2)) \in \{(\mu_+^*,(\mu_1^*,\mu_2^*)) \in \Omega_{1\bowtie 2} \times (\Omega_1 \times \Omega_2) | \pi_{S'}(\{\mu_1^* \cup \mu_2^*\}) = \{\mu\} \wedge \mu_1^* \cup \mu_2^* = \mu_+^*\}} \\
&\quad (m_1(\mu_1) * m_2(\mu_2)) \\
&= \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in \Omega_1 \times \Omega_2 | \pi_{S'}(\{\mu_1^* \cup \mu_2^*\}) = \{\mu\}\}} (m_1(\mu_1) * m_2(\mu_2)) \\
&= \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in \Omega_1 \times \Omega_2 | (\mu_1^* \cup \mu_2^*)_{|S'} = \mu\}} (m_1(\mu_1) * m_2(\mu_2)) \\
&\overset{(*)}{=} \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in \Omega_1 \times \Omega_2 | \mu_{1|S'}^* \cup \mu_{2|S'}^* = \mu\}} (m_1(\mu_1) * m_2(\mu_2)) \\
&= \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in \Omega_{\pi 1} \times \Omega_{\pi 2} | \mu_1^* \cup \mu_2^* = \mu\}} (m_{\pi 1}(\mu_1) * m_{\pi 2}(\mu_2)) \\
&= m_r(\mu),
\end{aligned}
$$

where step $(*)$ follows from the precondition $S' \supseteq pVars(A_1) \cap pVars(A_2)$.

(*PMPush*$^+$). Let $A_1^+, A_2^+ \in \mathbb{A}^+$ and $S \subset V$ be a set of variables. Further recall that by definition $S' := S \cup (pVars(A_1) \cap pVars(A_2))$ and $S'' := pVars(A_1) \cap pVars(A_2)$. We define $A_l^+ := \pi_S(A_1^+ \setminus A_2^+)$, $A_r^+ := \pi_S(\pi_{S'}(A_1^+) \setminus \pi_{S''}(A_2^+))$, $A_{1\setminus 2}^+ := A_1^+ \setminus A_2^+$, $A_{\pi 1}^+ := \pi_{S'}(A_1^+)$, $A_{\pi 2}^+ := \pi_{S''}(A_2^+)$, $A_{\pi 1\setminus\pi 2}^+ := A_{\pi 1}^+ \setminus A_{\pi 2}^+$, and fix document D and a mapping μ that is contained both in Ω_l and Ω_r. Applying the semantics from Definition 2.14, we rewrite the (right side) multiplicity $m_r(\mu)$ schematically:

$$\begin{aligned}
m_r(\mu) &= \textstyle\sum_{\mu_+ \in \{\mu_+^* \in \Omega_{\pi 1\setminus\pi 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_{\pi 1\setminus\pi 2}(\mu_+) \\
&\overset{(*_1)}{=} \textstyle\sum_{\mu_+ \in \{\mu_+^* \in \Omega_{\pi 1\setminus\pi 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_{\pi 1}(\mu_+) \\
&= \textstyle\sum_{\mu_+ \in \{\mu_+^* \in \Omega_{\pi 1\setminus\pi 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} \sum_{\mu'_+ \in \{\mu_+^\bullet \in \Omega_1 | \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+\}\}} m_1(\mu'_+) \\
&\overset{(S2)}{=} \textstyle\sum_{(\mu_+, \mu'_+) \in \{(\mu_+^*, \mu_+^\bullet) \in \Omega_{\pi 1\setminus\pi 2} \times \Omega_1 | \pi_S(\{\mu_+^*\}) = \{\mu\} \wedge \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m_1(\mu'_+) \\
&= \textstyle\sum_{(\mu_+, \mu'_+) \in \{(\mu_+^*, \mu_+^\bullet) \in \Omega_{\pi 1\setminus\pi 2} \times \Omega_1 | \pi_S(\pi_{S'}(\{\mu_+^\bullet\})) = \{\mu\} \wedge \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m_1(\mu'_+) \\
&\overset{(*_2)}{=} \textstyle\sum_{(\mu_+, \mu'_+) \in \{(\mu_+^*, \mu_+^\bullet) \in \Omega_{\pi 1\setminus\pi 2} \times \Omega_1 | \pi_S(\{\mu_+^\bullet\}) = \{\mu\} \wedge \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m_1(\mu'_+) \\
&\overset{(*_3)}{=} \textstyle\sum_{(\mu_+, \mu'_+) \in \{(\mu_+^*, \mu_+^\bullet) \in \Omega_{\pi 1\setminus\pi 2} \times \Omega_{1\setminus 2} | \pi_S(\{\mu_+^\bullet\}) = \{\mu\} \wedge \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m_1(\mu'_+) \\
&\overset{(*_4)}{=} \textstyle\sum_{\mu'_+ \in \{\mu_+^\bullet \in \Omega_{1\setminus 2} | \pi_S(\{\mu_+^\bullet\}) = \{\mu\}\}} m_1(\mu'_+) \\
&\overset{(*_5)}{=} \textstyle\sum_{\mu'_+ \in \{\mu_+^\bullet \in \Omega_{1\setminus 2} | \pi_S(\{\mu_+^\bullet\}) = \{\mu\}\}} m_{1\setminus 2}(\mu'_+) \\
&= m_l(\mu),
\end{aligned}$$

where step $(*_1)$ follows from the observation that $m_{\pi 1\setminus\pi 2}(\mu_+^*) = m_{\pi 1}(\mu_+^*)$ for all $\mu_+^* \in \Omega_{\pi 1\setminus\pi 2}$, rewriting step $(*_2)$ holds because $S \subseteq S'$, step $(*_3)$ follows from the observation that only those mappings from Ω_1 contribute to the result that are also contained in $\Omega_{1\setminus 2}$, step $(*_4)$ holds because every mapping $\mu_+^\bullet \in \Omega_{1\setminus 2}$ uniquely determines a mapping $\mu_+^* \in \Omega_{\pi 1\setminus\pi 2}$ through condition $\pi_{S'}(\{\mu_+^\bullet\}) = \mu_+^*$, and step $(*_5)$ follows from the observation that $m_1(\mu_+^\bullet) = m_{1\setminus 2}(\mu_+^\bullet)$ for all $\mu_+^\bullet \in \Omega_{1\setminus 2}$.

(*PLPush*$^+$). Analogical to the proof of *(PLPush)* for SPARQL set algebra from Appendix C.2 (observe that all rules that are used in the latter proof are also valid in the context of SPARQL bag algebra).

(*PFPush*$^+$). Similar in idea to (*PMPush*$^+$).

(*PMerge*$^+$). Let $A^+ \in \mathbb{A}^+$ and $S_1, S_2 \subset V$. Define expressions $A_l^+ := \pi_{S_1}(\pi_{S_2}(A^+))$, $A_r^+ := \pi_{S_1 \cap S_2}(A^+)$, and $A_{\pi 2}^+ := \pi_{S_2}(A^+)$. According to Lemma 4.5, it suffices to show that for each mapping μ that is contained in Ω_l and Ω_r it holds that $m_l(\mu) = m_r(\mu)$. We rewrite the left side multiplicity $m_l(\mu)$ schematically:

$$\begin{aligned}
m_l(\mu) &= \textstyle\sum_{\mu_+ \in \{\mu_+^* \in \Omega_{\pi 2} | \pi_{S_1}(\{\mu_+^*\}) = \{\mu\}\}} m_{\pi 2}(\mu_+) \\
&= \textstyle\sum_{\mu_+ \in \{\mu_+^* \in \Omega_{\pi 2} | \pi_{S_1}(\{\mu_+^*\}) = \{\mu\}\}} \sum_{\mu'_+ \in \{\mu_+^\bullet \in \Omega | \pi_{S_2}(\{\mu_+^\bullet\}) = \{\mu_+\}\}} m(\mu'_+) \\
&\overset{(S2)}{=} \textstyle\sum_{(\mu_+, \mu'_+) \in \{(\mu_+^*, \mu_+^\bullet) \in \Omega_{\pi 2} \times \Omega | \pi_{S_1}(\{\mu_+^*\}) = \{\mu\} \wedge \pi_{S_2}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m(\mu'_+) \\
&= \textstyle\sum_{(\mu_+, \mu'_+) \in \{(\mu_+^*, \mu_+^\bullet) \in \Omega_{\pi 2} \times \Omega | \pi_{S_1}(\pi_{S_2}(\{\mu_+^\bullet\})) = \{\mu\} \wedge \pi_{S_2}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m(\mu'_+)
\end{aligned}$$

$$\begin{aligned}
&\overset{(*_1)}{=} \sum_{\mu'_+ \in \{\mu^\bullet_+ \in \Omega \mid \pi_{S_1}(\pi_{S_2}(\{\mu^\bullet_+\})) = \{\mu\}\}} m(\mu'_+) \\
&\overset{(*_2)}{=} \sum_{\mu'_+ \in \{\mu^\bullet_+ \in \Omega \mid \pi_{S_1 \cap S_2}(\{\mu^\bullet_+\}) = \{\mu\}\}} m(\mu'_+) \\
&= m_r(\mu),
\end{aligned}$$

where step $(*_1)$ follows from the observation that mapping μ^*_+ is uniquely determined by $\mu^\bullet_+$ and $(*_2)$ follows directly from the semantics of operator π. □

C.9. Proof of Lemma 4.11

As before in the proof of Lemma 4.10 in Appendix C.8 we exploit the assumption that by Lemma 4.5 the results of evaluating set and bag algebra expressions differ at most in the associated cardinality, so (given that the rules hold for SPARQL set algebra) in all cases it suffices to show that, for a fixed mapping μ that is contained in the left and right side of the equivalence, the associated left and right side cardinalities for the mapping coincide. We fix a document D. Further, given a SPARQL bag algebra expression A^+, we denote by (Ω, m) the mapping multi-set obtained when evaluating A^+. Similarly, for a bag algebra expression A^+_i with some index i, we denote by (Ω_i, m_i) the mapping multi-set obtained when evaluating A^+_i.

(*FDecompI*$^+$). Let $A^+ \in \mathbb{A}^+$ and R be a filter condition. Put $A^+_l := \sigma_{R_1 \wedge R_2}(A^+)$, $A^+_r := \sigma_{R_1}(\sigma_{R_2}(A^+))$, and $A^+_{\sigma 2} := \sigma_{R_2}(A^+)$. Consider a mapping μ that is contained in the result of evaluating A^+_l and A^+_r, which implies that $\mu \in \Omega$ and $\mu \models R_1$, $\mu \models R_2$, $\mu \models R_1 \wedge R_2$. Applying the semantics from Definition 2.14 we can easily derive that $m_l(\mu) = m(\mu) = m_{\sigma 2}(\mu) = m_r(\mu)$.

(*FReord*$^+$). Follows from equivalence (*FDecompI*$^+$) for SPARQL bag algebra and the commutativity of the boolean operator $\wedge$.

(*FBndI*$^+$) - (*FBndIV*$^+$). Follow from the semantics of σ in Definition 2.10.

(*FUPush*$^+$). Follows from the semantics of σ and $\cup$ in Definition 2.10.

(*FMPush*$^+$). Let $A^+_1, A^+_2 \in \mathbb{A}^+$ and R be a filter condition. Put $A^+_l := \sigma_R(A^+_1 \backslash A^+_2)$, $A^+_r := \sigma_R(A^+_1) \setminus A^+_2$, $A^+_{1\backslash 2} := A^+_1 \setminus A^+_2$, and $A^+_{\sigma 1} := \sigma_R(A^+_1)$. Consider a mapping μ that is contained in the result of evaluating A^+_l and A^+_r. This implies that $\mu \models R$, $\mu \in \Omega_{1\backslash 2}$, $\mu \in \Omega_1$, and $\mu \in \Omega_{\sigma 1}$. Combining the semantics from Definition 2.14 with the above observations we obtain $m_l(\mu) = m_{1\backslash 2}(\mu) = m_1(\mu) = m_{\sigma 1}(\mu) = m_r(\mu)$.

(*FJPush*$^+$). See Section 4.3.2.

(*FLPush*$^+$). Analogical to the proof of *(FLPush)* for SPARQL set algebra from Appendix C.3 (observe that all rules that are used in the latter proof are also valid in the context of SPARQL bag algebra). □

C.10. Proof of Lemma 4.13

($MReord^+$), ($MMUCorr^+$), (MJ^+). The three equivalences follow easily from the semantics of operator $\setminus$ from Definition 2.14.

($\widetilde{LJ}^+$). Analogical to the proof of ($\widetilde{LJ}$) for SPARQL set algebra from Appendix C.5 (observe that all rules that are used in the latter proof are also valid in the context of SPARQL bag algebra).

($FLBndI^+$), ($FLBndII^+$). Analogical to the proof of *(FLBndI)* and *(FLBndII)* for SPARQL set algebra from Lemma 4.4 (observe that all rules that are used in the proof of the latter lemma are also valid in the context of SPARQL bag algebra). □