Foundations of SPARQL Query Optimization

Michael Schmidt* Michael Meier* Georg Lausen
University of Freiburg
Institute for Computer Science
Georges-Köhler-Allee, Building 051
79110 Freiburg i. Br., Germany
{mschmidt, meierm, lausen}@informatik.uni-freiburg.de

ABSTRACT

We study fundamental aspects related to the efficient processing of the SPARQL query language for RDF, proposed by the W3C to encode machine-readable information in the Semantic Web. Our key contributions are (i) a complete complexity analysis for all operator fragments of the SPARQL query language, which – as a central result – shows that the SPARQL operator OPTIONAL alone is responsible for the PSPACE-completeness of the evaluation problem, (ii) a study of equivalences over SPARQL algebra, including both rewriting rules like filter and projection pushing that are well-known from relational algebra optimization as well as SPARQL-specific rewriting schemes, and (iii) an approach to the semantic optimization of SPARQL queries, built on top of the classical chase algorithm. While studied in the context of a theoretically motivated set semantics, almost all results carry over to the official, bag-based semantics and therefore are of immediate practical relevance.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*

General Terms

SPARQL, RDF, Complexity, Query Optimization, SPARQL Algebra, Semantic Query Optimization

1. INTRODUCTION

The Resource Description Framework (RDF) [29] is a data format proposed by the W3C to encode information in a machine-readable way. From a technical point of view, RDF databases are collections of (*subject*,*predicate*,*object*) triples, where each triple encodes the binary relation *predicate* between *subject* and *object* and represents a single knowledge fact. Due to their homogeneous structure, RDF databases can be understood as labeled directed graphs, where each triple defines an edge from the *subject* to the *object* node under label *predicate* [12]. While originally designed to encode knowledge in the Semantic Web, RDF has found its way out of the Semantic Web community and entered the wider discourse of Computer Science. Coming along with its application in other areas, such as bio informatics or data integration, large RDF repositories have been created (see e.g. [18]) and it has repeatedly been observed that the database community is facing new challenges to cope with the specifics of RDF [6, 16, 19, 3].

With SPARQL [32], the W3C has recommended a declarative query language to extract data from RDF graphs. SPARQL comes with a powerful graph matching facility, whose basic construct are so-called triple patterns. During query evaluation, variables inside these patterns are matched against the RDF input graph. The solution of the evaluation process is then described by a set of mappings, where each mapping associates a set of variables with graph components. Beyond triple patterns, SPARQL provides advanced operators (namely SELECT, AND, FILTER, OPTIONAL, and UNION) which can be used to compose more expressive queries.

In this work we investigate fundamental aspects that are directly related to the evaluation of SPARQL queries. In particular, we revisit the complexity of the SPARQL query language (considerably extending and refining previous investigations from [26]) and study both algebraic and semantic optimization of the query language from a theoretical perspective. In this line, we present a collection of results that we have gathered in previous projects on SPARQL query processing, all of which are important background for understanding the basics of the SPARQL query language and for building efficient SPARQL optimizers. In our study, we abstract from implementation-specific issues like cost estimation functions, but rather provide fundamental results, techniques, and optimization schemes that may be fruitfully applied in virtually every SPARQL implementation. Accounting for this objective, we partially include important results from previous investigations (e.g. on the complexity of SPARQL or on algebraic optimization from [26, 1]), to make this paper an extensive reference for people who are planning to work in the context of SPARQL or to implement SPARQL engines.

Our first major contribution is a complete complexity analysis, comprising all possible operator fragments of the SPARQL query language. Our investigation separates subsets of the language that can be evaluated efficiently from more complex (and hence, more expressive) fragments and relates fragments of SPARQL to established query models, like e.g. conjunctive queries. Ultimately, our results deepen the understanding of the individual operators and their interrelations, and allow to transfer established results from other data models into the context of SPARQL query evaluation.

In our analysis of SPARQL complexity, we take the combined complexity of the SPARQL EVALUATION problem as a yardstick: given query Q, data set D, and candidate solution S as input, is S contained in the result of evaluating Q on D? Previous in-

*The work of this author was funded by Deutsche Forschungsgemeinschaft, grant GRK 806/03 and grant LA 598/7-1.

vestigations of SPARQL complexity in [26] have shown that full SPARQL is PSPACE-complete. Refining this important result, we show that already operator OPTIONAL alone makes query evaluation PSPACE-hard. From a practical perspective, we observe that the high complexity is caused by the unlimited nesting of OPTIONAL expressions and derive complexity bounds in the polynomial hierarchy for fragments with fixed OPTIONAL nesting depth. In summary, our results show that operator OPTIONAL is by far the most involved construct in SPARQL, which suggests that special care in query optimization should be taken in this specific operator.

Having established these theoretical results we then turn towards SPARQL optimization. To give some background, the semantics of SPARQL is formally defined on top of a compact algebra over mapping sets. In the evaluation process, the SPARQL operators are first translated into algebraic operations, which are then evaluated on the data set. More precisely, AND is mapped to a join operation, UNION to an algebraic union, OPTIONAL to a left outer join (which allows for the optional padding of information), FILTER to a selection, and SELECT to a projection. On the one hand, there are many parallels between these SPARQL algebra (SA) operators and the operators defined in relational algebra (RA), e.g. the study in [1] reveals that SA and RA have the same expressive power. On the other hand, the technical proof in [1] indicates that a semantics-preserving SA-to-RA translation is far from being trivial and shows that there are still fundamental differences between both.

Tackling the specific challenges of the SPARQL query language, over the last years various proposals for the efficient evaluation of SPARQL have been made, comprising a wide range of optimization techniques such as normal forms [26], triple pattern reordering based on selectivity estimations [20, 23], or RISC-style query processing [23]. In addition, indices [13] and storage schemes [30, 34, 6, 3] for RDF have been explored, to provide efficient data access paths. Another line of research is the translation of SPARQL queries into established data models like SQL [5, 9] or datalog [27], to evaluate them with traditional engines that exploit established optimization techniques implemented in traditional systems.

One interesting observation is that the "native" optimization proposals for SPARQL (i.e. those that do not rely on a mapping into the relational context or datalog) typically have a strong focus on SPARQL AND-only queries, i.e. mostly disregard the optimization of queries involving operators like FILTER or OPTIONAL (cf. [13, 20, 23, 3]). The efficient evaluation of AND-only queries (or AND-connected blocks inside queries) is undoubtedly an important task in SPARQL evaluation, so the above-mentioned approaches are valuable groundwork for SPARQL optimizers. Still, a comprehensive optimization scheme should also address the optimization of more involved queries. To give evidence for this claim, the experimental study in [19] reveals severe performance bottlenecks when evaluating complex SPARQL queries (in particular queries involving operator OPTIONAL) for both existing SPARQL implementations and state-of-the-art mapping schemes from SPARQL to SQL.

One reason for these deficiencies may be that in the past only few fundamental work has been done in the context of SPARQL query optimization (we resume central results from [26, 27] later in this paper) and that the basics of SA and its relation towards RA are still insufficiently understood. We argue that – like in relational algebra, where the study of algebraic rewriting rules has triggered the development of manifold optimization techniques – a study of SPARQL algebra would alleviate the development of comprehensive optimization approaches and therefore believe that a schematic investigation of SPARQL algebra is long overdue. Addressing this task, we present an elaborate study of SA equivalences, covering all its operators and their interrelations. When interpreted as rewriting rules, these equivalences allow to transfer 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 queries into SA according to the SPARQL semantics – manifests into a characteristic combination of the selection and left outer join operator. Ultimately, our results improve the understanding of SPARQL algebra and lay the foundations for the design of comprehensive optimization schemes.

Complementary to algebraic optimization, we study constraint-based optimization, also known as semantic query optimization (SQO), for SPARQL. The idea of SQO, which is well-known from the context of conjunctive query optimization (e.g., [2]), deductive database (e.g., [4]), and relational databases (e.g., [15]), is to exploit integrity constraints over the input database. Such constraints are valuable input to query optimizers, because they restrict the state space of the database and often can be used to rewrite queries into equivalent, but more efficient, ones. Constraints could be user-specified, automatically extracted from the underlying database, or – if SPARQL is evaluated on top of an RDFS inference system – may be implicitly given by the semantics of the RDFS vocabulary.

Our SQO approach splits into two parts. First, we translate AND-connected blocks inside queries into conjunctive queries, optimize them using the well-known chase algorithm [21, 14, 2, 8], and translate the optimized conjunctive queries back into SPARQL. In a second step, we apply SPARQL-specific rules that allow us to optimize more complex subqueries, such as queries involving operator OPTIONAL. To give an example, we propose a rule that allows us to replace operator OPTIONAL by AND in cases where the pattern inside the OPTIONAL clause is implied by the given constraint set.

We summarize the central contributions of this work as follows.

- We present novel complexity results for SPARQL fragments, showing as a central result that already the fragment containing operator OPTIONAL alone is PSPACE-complete (in combined complexity). Further, we derive tight complexity bounds in the polynomial hierarchy for expressions with fixed nesting depth of OPTIONAL subexpressions. Finally, we show that all OPTIONAL-free fragments are either NP-complete (whenever operator AND cooccurs with UNION or SELECT) or in PTIME.
- We identify a large set of equivalences over SPARQL algebra. As a central tool, we develop the concepts of *possible* and *certain variables*, which constitute upper and lower bounds for the variables that may be bound in result mappings, account for the characteristics of SPARQL, and allow us to state equivalences over SPARQL algebra in a clean and compact way. Our investigation comprises both the study of optimization schemes known from the relational context (such as filter and projection pushing) and SPARQL-specific rewriting techniques.
- We present an SQO scheme to optimize SPARQL queries under a set of integrity constraints over the RDF database. Our optimization approach adheres (yet is not limited) to constraints obtained from the RDFS inference mechanism [29]. It builds upon the classical chase algorithm to optimize AND-only queries, but also supports rule-based optimization of more complex queries.
- While established for a theoretically motivated set semantics, we show that almost all results carry over to the official, bag-based semantics proposed by the W3C, so both our complexity and optimization results are of immediate practical interest.

We start with the preliminaries in Section 2, discuss the complexity of SPARQL evaluation in Section 3, study SPARQL algebra in Section 4, and present our SQO scheme for SPARQL in Section 5.

2. PRELIMINARIES

We assume that the set of natural numbers $\mathbb{N}$ does not include the element 0 and define $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. Furthermore, we introduce the notation $i \in [n]$ as a shortcut for $i \in \{1, \ldots, n\}$.

2.1 The RDF Data Format

We follow the notation from [26] and consider three disjoint sets B (blank nodes), L (literals), and U (URIs) and use the shortcut BLU to denote the union of B, L, and U. As a convention, we use quoted strings to denote literals (e.g. "Joe", "30") and prefix blank nodes with "_:". An *RDF triple* $(v_1,v_2,v_3) \in BU \times U \times BLU$ connects subject v_1 through predicate v_2 to object v_3. An *RDF database*, also called *RDF document*, is a finite set of triples.

2.2 The SPARQL Query Language

We now introduce two alternative semantics for SPARQL evaluation, namely a *set* and a *bag semantics*. The set-based semantics is inspired by previous theoretical investigations in [24, 1]; the bag semantics closely follows the W3C Recommendation [32] and [25].

Syntax. Let V be a set of variables disjoint from BLU. We distinguish variables by a leading question mark symbol, e.g. writing $?x$ or $?name$. We start with an abstract syntax for filter conditions. For $?x, ?y \in V$ and $c, d \in LU$ we define *filter conditions* recursively as follows. The expressions $?x = c$, $?x = ?y$, $c = d$, and *bnd*$(?x)$ are atomic filter conditions. Second, if R_1, 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. Next, we introduce an abstract syntax for expressions (where we use OPT as a shortcut for operator OPTIONAL):

DEFINITION 1 (SPARQL EXPRESSION). A *SPARQL expression* is an expression that is built recursively as follows. (1) A *triple pattern* $t \in UV \times UV \times LUV$ is an expression. (2) If Q_1, Q_2 are expressions and R is a filter condition, then Q_1 FILTER R, Q_1 UNION Q_2, Q_1 OPT Q_2, and Q_1 AND Q_2 are expressions. □

The official W3C Recommendation [32] defines four different types of queries on top of expressions, namely SELECT, ASK, CONSTRUCT, and DESCRIBE queries. We will restrict our discussion to SPARQL SELECT and ASK queries.[1] SELECT queries extract the set of all result mappings, while ASK queries are boolean queries that return *true* iff there is one or more result, *false* otherwise.

DEFINITION 2 (SELECT QUERY, ASK 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 SELECT$_S(Q)$. A SPARQL ASK query is an expression of the form ASK(Q). □

In the remainder of the paper 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 SELECT$_{?x,?y}(Q)$ for SELECT$_{\{?x,?y\}}(Q)$.

A Set-based Semantics for SPARQL. Central to the evaluation process in SPARQL is the notion of so-called mappings, which express variable-to-document bindings during evaluation. Formally, a *mapping* is a partial function $\mu : V \to BLU$ from a subset of variables V to RDF terms BLU. By $\mathcal{M}$ we denote the universe of all mappings. The domain of a mapping μ, *dom*(μ), is the subset of V for which μ is defined. We say that two mappings μ_1, μ_2 are compatible, written $\mu_1 \sim \mu_2$, if they agree on all shared variables, i.e. if $\mu_1(?x) = \mu_2(?x)$ for all $?x \in$ *dom*$(\mu_1) \cap$ *dom*(μ_2).

[1]The main challenge of query evaluation (and our focus here) lies in the core evaluation phase, which is the same for all query forms.

We overload function $vars$ (defined previously for filter conditions) and denote by $vars(t)$ all variables in triple pattern t. Further, by $\mu(t)$ we denote the triple pattern obtained when replacing all variables $?x \in dom(\mu) \cap vars(t)$ in t by $\mu(?x)$.

EXAMPLE 1. Consider the three mappings $\mu_1 := \{?x \mapsto a1\}$, $\mu_2 := \{?x \mapsto a2, ?y \mapsto b2\}$, and $\mu_3 := \{?x \mapsto a1, ?z \mapsto c1\}$. It is easy to see that $dom(\mu_1) = \{?x\}$, $dom(\mu_2) = \{?x, ?y\}$, and $dom(\mu_3) = \{?x, ?z\}$. Further, we can observe that $\mu_1 \sim \mu_3$, but $\mu_1 \not\sim \mu_2$ and $\mu_2 \not\sim \mu_3$. Given triple pattern $t_1 := (c, ?x, ?y)$ we have $vars(t_1) = \{?x, ?y\}$ and e.g. $\mu_2(t_1) = (c, a2, b2)$. □

We next define the semantics of filter conditions w.r.t mappings. A mapping μ *satisfies* the filter condition *bnd*$(?x)$ if variable $?x$ is contained in the $dom(\mu)$; the filter conditions $?x = c$, $?x = ?y$, and $c = d$ are equality checks that compare the value of $\mu(?x)$ with c, $\mu(?x)$ with $\mu(?y)$, and c with d, respectively; these checks fail whenever one of the variables is not bound in μ. The boolean connectives $\neg$, $\vee$, and $\wedge$ are defined in the usual way. We write $\mu \models R$ iff μ satisfies filter condition R (cf. Appendix A.1 for details).

The solution of a SPARQL expression or query over document D is described by a set of mappings, where each mapping represents a possible answer. The semantics of SPARQL query evaluation is then defined by help of a compact algebra over such mapping sets:

DEFINITION 3 (SPARQL SET ALGEBRA). Let $\Omega, \Omega_l, \Omega_r$ 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 ($\leftouterjoin$), projection (π), and selection (σ):

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

We refer to these algebraic operations as *SPARQL set algebra*.□

To define the evaluation result of expressions, SELECT queries, and ASK queries we follow the compositional semantics from [26] and define a function $[\![.]\!]_D$ that translates them into SA:

DEFINITION 4 (SPARQL SET SEMANTICS). Let D be an RDF document, t a triple pattern, Q, Q_1, Q_2 SPARQL expressions, R a filter condition, and $S \subset V$ a set of variables. We define

$$
\begin{aligned}
[\![t]\!]_D &:= \{\mu \mid \mathit{dom}(\mu) = \mathit{vars}(t) \text{ and } \mu(t) \in D\} \\
[\![Q_1 \text{ AND } Q_2]\!]_D &:= [\![Q_1]\!]_D \bowtie [\![Q_2]\!]_D \\
[\![Q_1 \text{ OPT } Q_2]\!]_D &:= [\![Q_1]\!]_D \leftouterjoin [\![Q_2]\!]_D \\
[\![Q_1 \text{ UNION } Q_2]\!]_D &:= [\![Q_1]\!]_D \cup [\![Q_2]\!]_D \\
[\![Q \text{ FILTER } R]\!]_D &:= \sigma_R([\![Q]\!]_D) \\
[\![\text{SELECT}_S(Q)]\!]_D &:= \pi_S([\![Q]\!]_D) \\
[\![\text{ASK}(Q)]\!]_D &:= \neg(\emptyset = [\![Q]\!]_D)
\end{aligned}
$$
□

EXAMPLE 2. Consider the SPARQL SELECT query

$$
\begin{aligned}
Q_1 := \text{SELECT}_{?p,?e}(((?p, age, ?a) \text{ OPT } (?p, email, ?e)) \\
\text{FILTER } (?a = \text{"30"}))
\end{aligned}
$$

which retrieves all 30-year-old persons (*?p*) and, optionally (i.e., if available), their email address (*?e*). Further assume that the database $D := \{(P1, age, \text{"30"}), (P2, age, \text{"29"}), (P3, age, \text{"30"}), (P3, email, \text{"joe@tld.com"})\}$ is given. It is easily verified that $[\![Q_1]\!]_D = \{\{?p \mapsto P1\}, \{?p \mapsto P3, ?e \mapsto \text{"joe@tld.com"}\}\}$. □

From Set to Bag Semantics. We next consider the corresponding bag semantics, obtained from the set semantics when interpreting mappings sets as bags of mappings. The bag semantics thus differs in that mappings can appear multiple times in the evaluation result. Formally, we define the bag semantics using mapping multi-sets, which associate a multiplicity with each mapping:

DEFINITION 5 (MAPPING MULTI-SET). A mapping multi-set is a tuple (Ω,m), where Ω is a mapping set and $m : \mathcal{M} \mapsto \mathbb{N}_0$ is a total function s.t. $m(\mu^+) \geq 1$ 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 and say that μ^+ occurs $m(\mu^+)$ times in Ω. □

We can easily formalize the bag semantics using an adapted versions of the algebraic operations from Definition 3 that operate on top of multi-sets and take the multiplicity of the set elements into account. To give an example, the union operation over multi-sets, $(\Omega_l, m_l) \cup (\Omega_r, m_r)$, yields the multi-set $(\Omega_l \cup \Omega_r, m)$, where $m(\mu) := m_l(\mu) + m_r(\mu)$ for all $\mu \in \mathcal{M}$. We call this algebra over multi-sets *SPARQL bag algebra*. Given the SPARQL bag algebra, we immediately obtain the bag semantics for SPARQL when modifying the first rule in Definition 4 (the triple pattern case) such that it returns a multi-set instead of a set. We use function $[\![.]\!]^+_D$ to denote the mapping multi-set obtained when evaluating a SPARQL expression or query according to the bag semantics. The interested will find a proper formalization in Appendix A.2.

EXAMPLE 3. Let $Q := (?x, c, c)$ UNION $(c, c, ?x)$, document $D := \{(c, c, c)\}$, and $\mu := \{?x \mapsto c\}$. Then $[\![Q]\!]^+_D = (\{\mu\}, m)$ where $m(\mu) := 2$ and $m(\mu') := 0$ for all $\mu' \in \mathcal{M} \setminus \{\mu\}$. □

If Ω is a mapping set and (Ω', m') a mapping multi-set such that $\Omega = \Omega'$ and $m(\mu') = 1$ for all $\mu' \in \Omega'$, we say that Ω equals to (Ω', m') and denote this by $\Omega \cong (\Omega', m')$. Going one step further, given a SPARQL query or expression Q we say that the *bag and set semantics coincide for* Q iff it holds that $[\![Q]\!]_D \cong [\![Q]\!]^+_D$ for every RDF document D. In general, the two semantics do not coincide, as witnessed by the previous example (observe that $m(\mu) > 1$).

3. COMPLEXITY OF SPARQL

We introduce the SPARQL operator shortcuts $\mathcal{A} :=$ AND, $\mathcal{F} :=$ FILTER, $\mathcal{O} :=$ OPT, and $\mathcal{U} :=$ UNION and denote the class of SPARQL expressions that can be constructed using a set of operators (plus triple patterns) by concatenating the respective shortcuts. For instance, class $\mathcal{AU}$ comprises all SPARQL expressions built using only AND, UNION, and triple patterns. By $\mathcal{E} := \mathcal{AFOU}$ we denote the full class of SPARQL expressions (cf. Definition 1). We will use the terms *class* and *fragment* interchangeably.

We follow the approach from [26] and take the complexity of the EVALUATION problem as a reference: given a mapping μ, a document D, and a SPARQL expression or query Q as input: is $\mu \in [\![Q]\!]_D$? The next theorem summarizes results on the combined complexity of SPARQL from [26], rephrased in our notation.[2]

THEOREM 1. (see [26]) The EVALUATION problem is (1) in PTIME for class $\mathcal{AF}$ (membership in PTIME for classes $\mathcal{A}$ and $\mathcal{F}$ follows immediately), (2) NP-complete for fragment $\mathcal{AFU}$, and (3) PSPACE-complete for classes $\mathcal{AOU}$, $\mathcal{AFO}$, and $\mathcal{E}$. □

The theorem (and hence, the study in [26]) leaves several questions unanswered. In particular, it is not clear whether there are smaller fragments causing NP-hardness (resp. PSPACE-hardness) than those listed in Theorem 1(2) (resp. Theorem 1(3)). Further, projection (in form of SELECT clauses) was not investigated in [26].

Set vs. Bag Semantics. The previous definition of the EVALUATION problem 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. We consider the associated evaluation problem for bag semantics, denoted by EVALUATION$^+$: given a mapping μ, document D, and SPARQL expression or query Q as input: let $[\![Q]\!]^+_D := (\Omega, m)$, is $\mu \in \Omega$?[3] The following Lemma shows that the bag semantics differs from the set semantics at most in the multiplicity associated to each mapping:

LEMMA 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 $\mu \in \Omega \Leftrightarrow \mu \in \Omega^+$. □

It follows easily as a corollary that the set and bag semantics do not differ w.r.t. to the complexity of the evaluation problem:

COROLLARY 1. Let μ be a mapping, D an RDF document, and Q be an expression or query. Then EVALUATION$(\mu, D, Q) \Leftrightarrow$ EVALUATION$^+(\mu, D, Q)$. □

This result allows us to use the simpler set semantics for our study of SPARQL complexity, while all results carry over to bag semantics (and therefore apply to the SPARQL W3C standard).

3.1 OPT-free Expressions

Our first goal is to establish a more precise characterization of the UNION operator, to improve the understanding of the operator and its relation to others beyond the known NP-completeness result for class $\mathcal{AFU}$. The following theorem gives the results for all OPT-free fragments that are not covered by Theorem 1.

THEOREM 2. The EVALUATION problem is (1) in PTIME for classes $\mathcal{U}$ and $\mathcal{FU}$, and (2) NP-complete for class $\mathcal{AU}$. □

Proof Sketch. We sketch the NP-hardness part of claim (2), the remaining parts can be found in Appendix B.2. To prove hardness, we reduce the SETCOVER problem to the EVALUATION problem for class $\mathcal{AU}$. SETCOVER is known to be NP-complete, so the reduction gives us the desired hardness result. The SETCOVER problem is defined as follows. 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 $|I| \leq l$ 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 $P_{S_i} := (c, c, ?X_1)$ AND ... AND $(c, c, ?X_m)$. Next, to encode the set $S := \{S_1, \ldots, S_n\}$ of all S_i we define the expression $P_S := P_{S_1}$ UNION ... UNION P_{S_n}. Finally we define expression $P := P_S$ AND ... AND P_S, where P_S 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 idea is that the encoding P allows us to "merge" (at most) l arbitrary sets S_i. It is straightforward to show that the SETCOVER problem is true if and only if $\mu := \{?U_1 \mapsto c, \ldots, ?U_k \mapsto c\} \in [\![P]\!]_D$, i.e. if the complete universe U can be obtained by merging these sets. □

[2] [26] contains some more complexity results for the class of so-called well designed graph patterns, obtained from a syntactic restriction for SPARQL expressions, which we do not repeat here.

[3] 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.

3.2 Complexity of Expressions Including OPT

We next investigate the complexity of operator OPT and its interaction with other operators beyond the PSPACE-completeness results for $\mathcal{AOU}$, $\mathcal{AFO}$, and $\mathcal{E}$ stated in Theorem 1(3). The following theorem refines the three previously mentioned results.

THEOREM 3. EVALUATION is PSPACE-complete for $\mathcal{AO}$. □

Proof Sketch. We reduce QBF, the validity problem for 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 CNF, to the EVALUATION problem for fragment $\mathcal{AO}$. The reduction divides into (i) the encoding of the inner formula ψ and (ii) the encoding of the surrounding quantifier-sequence. Part (ii) has been presented in [24], so we discuss only part (i) here. We illustrate the idea of the encoding by example, showing how to encode the quantifier-free boolean CNF formula $\psi := C_1 \wedge C_2$ with $C_1 := (x_1 \vee \neg y_1)$ and $C_2 := (\neg x_1 \vee y_1)$ using only operators OPT and AND (the technical proof can be found in Appendix B.3). For this formula, we set up the database

$$\begin{aligned} D := \{&(a, \mathit{tv}, 0), (a, \mathit{tv}, 1), (a, \mathit{false}, 0), (a, \mathit{true}, 1), \\ &(a, var_1, x_1), (a, var_1, y_1), (a, var_2, x_1), (a, var_2, y_1), \\ &(a, x_1, x_1), (a, y_1, y_1)\}, \end{aligned}$$

where the first four tuples are fixed, the next four tuples encode the variables that appear in the clauses of ψ (e.g., (a, var_1, x_1) means that variable x_1 appears in clause C_1), and the final two tuples stand for the two variables that appear in ψ. We then define $P_\psi := P_{C_1}$ AND P_{C_2}, where

$$\begin{aligned} P_{C_1} := &((a, var_1, ?var_1) \\ &\text{OPT } ((a, x_1, ?var_1) \text{ AND } (a, true, ?X_1))) \\ &\quad \text{OPT } ((a, y_1, ?var_1) \text{ AND } (a, false, ?Y_1)), \text{ and} \\ P_{C_2} := &((a, var_2, ?var_2) \\ &\text{OPT } ((a, y_1, ?var_2) \text{ AND } (a, true, ?Y_1))) \\ &\quad \text{OPT } ((a, x_1, ?var_2) \text{ AND } (a, false, ?X_1)). \end{aligned}$$

In these expression, variables $?X_1$, $?Y_1$ stand for the respective variables x_1, y_1 in ϕ, and a binding $?X_1 \mapsto 1$ encodes $x_1 = true$. The intuition behind P_{C_1} and P_{C_2} is best seen when evaluating them on the input database D. For instance, for P_{C_1} we have

$$\begin{aligned} [\![P_{C_1}]\!]_D &= (\{\{?var_1 \mapsto x_1\}, \{?var_1 \mapsto y_1\}\} \\ &\qquad \bowtie \{\{?var_1 \mapsto x_1, ?X_1 \mapsto 1\}\}) \\ &\qquad \bowtie \{\{?var_1 \mapsto y_1, ?Y_1 \mapsto 0\}\} \\ &= \{\{?var_1 \mapsto x_1, ?X_1 \mapsto 1\}, \{var_1 \mapsto y_1\}\} \\ &\qquad \bowtie \{\{?var_1 \mapsto y_1, ?Y_1 \mapsto 0\}\} \\ &= \{\{?var_1 \mapsto x_1, ?X_1 \mapsto 1\}, \{?var_1 \mapsto y_1, ?Y_1 \mapsto 0\}\}. \end{aligned}$$

We observe that the subexpression $Q := (a, var_1, ?var_1)$ with $[\![Q]\!]_D = \{\{?var_1 \mapsto x_1\}, \{?var_1 \mapsto y_1\}\}$ sets up one mapping for each variable in C_1. When computing the left outer join of $[\![Q]\!]_D$ with $\{\{?var_1 \mapsto x_1, ?X_1 \mapsto 1\}\}$, the first mapping in $[\![Q]\!]_D$ is extended by binding $?X_1 \mapsto 1$ and the second one is kept unmodified; in the next step, the second mapping from $[\![Q]\!]_D$ is extended instead. The final result contains two mappings, which reflect exactly the satisfying truth assignments for C_1: it evaluates to true if x_1 is true (binding $?X_1 \mapsto 1$ in the first mapping) or if y_1 is false ($?Y_1 \mapsto 0$ in the second mapping). It is easily verified that

$$\begin{aligned} [\![P_\psi]\!]_D = \{&\{?var_1 \mapsto x_1, ?var_2 \mapsto y_1, ?X_1 \mapsto 1, ?Y_1 \mapsto 1\}, \\ &\{?var_1 \mapsto y_1, ?var_2 \mapsto x_1, ?X_1 \mapsto 0, ?Y_1 \mapsto 0\}\}, \end{aligned}$$

which represents exactly the two satisfying truth assignments for formula ψ, i.e. $?X_1 \mapsto 1$, $?Y_1 \mapsto 1$ and $?X_1 \mapsto 0$, $?Y_1 \mapsto 0$. □

Note that, in contrast to the PSPACE-hardness proofs for $\mathcal{AOU}$, $\mathcal{AFO}$, and $\mathcal{E}$ in [26] (cf. Theorem 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. 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).

So far, tight bounds for fragment $\mathcal{O}$ are still missing. The next theorem gives the central result of our complexity study:

THEOREM 4. EVALUATION is PSPACE-complete for $\mathcal{O}$. □

Analogously to Theorem 3, the result follows from an encoding of quantified boolean formulas, now using only operator OPT. The intuition behind this high complexity is that $\bowtie$, the algebraic counterpart of OPT, is defined using $\bowtie$, $\cup$, $\setminus$; the mix of these operations (in particular the negation operator $\setminus$) makes evaluation hard. We conclude this subsection with a corollary of Theorems 1(3) and 4:

COROLLARY 2. The EVALUATION problem for every expression fragment involving operator OPT is PSPACE-complete. □

3.3 The Source of Complexity

The proofs of Theorems 3 and 4 both rely on a nesting of OPT expression that increases with the number of quantifier alternations encountered in the encoded quantified formula. When fixing the nesting depth of OPT expressions, lower complexity bounds in the polynomial hierarchy [33] can be derived. We denote by $rank(Q)$ the maximal nesting depth of OPT expressions in Q, where OPT-free expressions have rank zero (see Appendix A.4 for a formal definition). Given a fragment F, we denote by $F_{\leq n}$ the class of expressions $Q \in F$ with $rank(Q) \leq n$. Then:

THEOREM 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 the EVALUATION problem for class $\mathcal{E}_{\leq 0}$ is complete for Σ^P_1=NP, which coincides with the result for OPT-free expressions (i.e., class $\mathcal{AFU}$) stated in Theorem 1. With increasing nesting-depth of OPT expressions we climb up the polynomial hierarchy. The proof in Appendix B.5 relies on a version of the QBF problem with fixed quantifier alternations, in which the number of alternations fixes the complexity class in the polynomial hierarchy.

3.4 From Expressions to Queries

We conclude the complexity study with a discussion of SPARQL queries, i.e. fragments involving projection in the form of a SELECT operator (see Definition 2). We extend the notation for classes. For some expression class F, we denote by F^π the class of queries $\text{SELECT}_S(Q)$, where $S \subset V$ is a finite set of variables and $Q \in F$.

It is easily shown that projection comes for free in fragments that are at least NP-hard. Based on this observation and an additional study of the remaining query fragments (i.e., those with PTIME complexity), we obtain the following complete classification:

THEOREM 6. EVALUATION is (1) PSPACE-complete for all query fragments involving operator OPT, (2) Σ^P_{n+1}-complete for fragment $\mathcal{E}^\pi_{\leq n}$ (for $n \in \mathbb{N}_0$), (3) NP-complete for $\mathcal{A}^\pi$, $\mathcal{AF}^\pi$, $\mathcal{AU}^\pi$, and $\mathcal{AFU}^\pi$, and (4) in PTIME for classes $\mathcal{F}^\pi$, $\mathcal{U}^\pi$, and $\mathcal{FU}^\pi$. □

3.5 Summary of Results

We summarize the complexity results in Figure 1. All fragments that fall into NP, Σ^P_i, and PSPACE also are complete for the respective complexity class. As an extension of previous results, the figure shows that each Σ^P_{n+1} also contains the fragment $\mathcal{AFO}_{\leq n}$ and the corresponding query fragment $\mathcal{AFO}^\pi_{\leq n}$. These results were

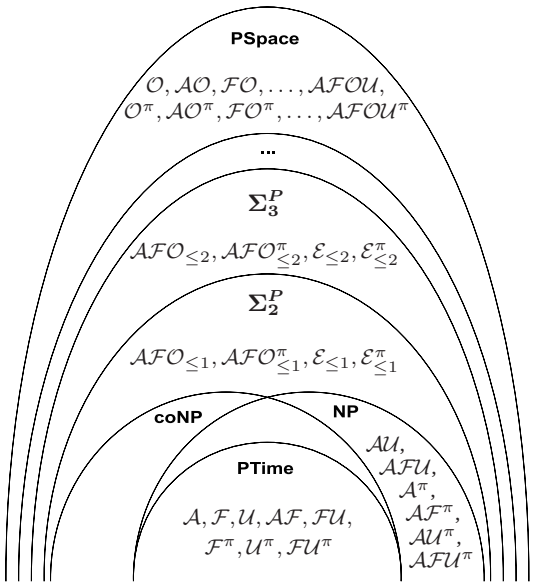

Figure 1: Summary of Complexity Results

not explicitly stated before, but follow directly from the proof of Theorem 5 for class $\mathcal{E}^\pi_{\leq n}$ (cf. Appendix B.5), which does not use the UNION operator in its encoding. Observe that the categorization is complete w.r.t. all possible expression and query fragments.

4. ALGEBRAIC SPARQL OPTIMIZATION

Having established the theoretical background, we now turn towards algebraic optimization. We start with two functions that statically classify the variables that appear in some SPARQL algebra expression A. The first one, $cVars(A)$, estimates the *certain variables* of A and fixes a lower bound for variables that are bound in every result mapping obtained when evaluating A. The second one, $pVars(A)$, gives an upper bound for the so-called *possible variables* of A, an overestimation for the set of variables that might be bound in result mapping. Both functions are independent from the input document and can be computed efficiently. They account for the specifics of SPARQL, where variables occurring in the expression may be unbound in result mappings, and take a central role in subsequent investigations. We start with the certain variables:

DEFINITION 6 (FUNCTION *cVars*). Let A be a SPARQL set algebra expression. Function $cVars(A)$ extracts the set of so-called *certain variables* and is recursively defined as

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

□

Observe that the case $A_1 \rtimes A_2$ is not explicitly listed, but follows by the semantics of operator $\rtimes$, i.e. we can rewrite $A_1 \rtimes A_2$ into $(A_1 \bowtie A_2) \cup (A_1 \setminus A_2)$ and apply the rules for $\bowtie$, $\cup$, and $\setminus$ to the rewritten expression. Also note that the function is defined for set algebra, as witnessed by rule $cVars([\![t]\!]_D) := vars(t)$. We can easily transfer the function to bag algebra by replacing this rule through $cVars([\![t]\!]^+_D) := vars(t)$ and therefore shall also use it for bag algebra expressions. The key property of certain variables is:

PROPOSITION 1. Let A be a set algebra expression and let Ω_A denote the mapping set obtained when evaluating A on any document D. Then $?x \in cVars(A) \rightarrow \forall \mu \in \Omega_A : ?x \in dom(\mu)$. □

The definition of function $pVars(A)$ is similar (see Definition 17 in Appendix A.5). It is simply obtained from the definition of $cVars(A)$ by replacing $\cap$ in the right side of the rules for $A_1 \cup A_2$ by $\cup$, because both the variables from A_1 and A_2 may appear in result mappings. Possible variables exhibit the following property.

PROPOSITION 2. Let A be a set algebra expression and let Ω_A denote the mapping set obtained when evaluating A on any document D. Then for all $\mu \in \Omega_A : ?x \in \mathit{dom}(\mu) \rightarrow ?x \in \mathit{pVars}(A)$.□

We note that both Proposition 1 and 2 naturally carry over to bag semantics and complement the previous discussion with an example that illustrates the definition of $cVars(A)$ and $pVars(A)$:

EXAMPLE 4. Consider the SPARQL set algebra expression $A := \pi_{?x,?y}((\![(a, q, ?x)]\!]_D \bowtie [\![(a, ?y, ?z)]\!]_D) \cup [\![(a, p, ?x)]\!]_D)$. We have that $pVars(A) = \{?x, ?y\}$ and $cVars(A) = \{?x\}$. □

Outline and Related Work. In the remainder of this section we present a set of algebraic equivalences for SPARQL algebra, covering all the algebraic operators introduced in Definition 3. 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. We will first study rewriting rules for SPARQL set algebra in Section 4.1, see what changes when switching to bag algebra in Section 4.2, and discuss practical implications and extensions in Section 4.3.

In the interest of a complete survey, we include equivalences that have been stated before in [24]. Among the equivalences in Figure 2, a majority of the rules from groups `I` and `II`, as well as *(FDecompI+II)*, *(MJ)*, and *(FUPush)* are borrowed from [24]. Further, rules $\widetilde{(JIdem)}$, *(FJPush)*, and $\widetilde{(LJ)}$ generalize Lemma (2), Lemma 1(2), and Lemma 3(3) from [24], respectively. These generalizations rely on the novel notion of incompatibility property (which will be introduced in Section 4.1) and extend the applicability of the original rules. We emphasize that almost three-fourths of the rules presented in this section are new. In the subsequent discussion we put a strong focus on these newly-discovered rules.

4.1 Rewriting under Set Semantics

We investigate two fragments of SPARQL set algebra. The first one, called *fragment* $\mathbb{A}$, comprises the full class of SPARQL set algebra expressions, i.e. expressions built using $\cup$, $\bowtie$, $\setminus$, $\rtimes$, π, σ, and triple patterns of the form $[\![t]\!]_D$. We understand a set algebra expression $A \in \mathbb{A}$ as a purely syntactic entity. Yet, according to the SPARQL set semantics (cf. Definition 4) each set algebra expression A implicitly defines a mapping set if document D is fixed. Therefore, we refer to the mapping set obtained by application of the semantics as the *the result of evaluating A on document D*.

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

DEFINITION 7 (INCOMPATIBILITY PROPERTY). 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 $\mu_1 \not\sim \mu_2$. □

DEFINITION 8 (FRAGMENT $\widetilde{\mathbb{A}}$). We define class $\widetilde{\mathbb{A}} \subset \mathbb{A}$ recursively as follows. An expression $A \in \mathbb{A}$ is contained in $\widetilde{\mathbb{A}}$ iff

- $\widetilde{A} := [\![t]\!]_D$ 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,
- $\widetilde{A} := \widetilde{A_1} \setminus \widetilde{A_2}$, where $\widetilde{A_1}$ and $\widetilde{A_2}$ are $\widetilde{\mathbb{A}}$ expressions,
- $\widetilde{A} := \widetilde{A_1} \between \widetilde{A_2}$, 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 2. Every $\widetilde{A} \in \widetilde{\mathbb{A}}$ has the incompatibility property. □

The following example illustrates that expressions outside fragment $\widetilde{\mathbb{A}}$ generally do not exhibit the incompatibility property:

EXAMPLE 5. Let $D := \{(0, f, 0), (1, t, 1), (a, v, 0), (a, v, 1)\}$ be an RDF database, and $A_1 := [\![(0, f, ?x)]\!]_D \cup [\![(1, t, ?y)]\!]_D$, $A_2 := \pi_{?x,?y}(([\![(a, v, ?z)]\!]_D \between [\![(?z, f, ?x)]\!]_D) \between [\![(?z, t, ?y)]\!]_D)$ be set algebra expressions. When evaluating A_1 and A_2 on D we obtain the mapping set $\Omega = \{\{?x \mapsto 0\}, \{?y \mapsto 1\}\}$ for both expressions. Obviously, the two mappings in Ω are compatible. Note that neither A_1 nor A_2 are $\widetilde{\mathbb{A}}$ expressions. As a positive example, observe that $A_3 := \pi_{?x}([\![(a, ?x, ?y)]\!]_D \cup [\![(b, ?x, ?y)]\!]_D) \in \widetilde{\mathbb{A}}$. □

Algebraic Laws. We start our investigation of SPARQL set algebra equivalences with some basic rules that hold with respect to common algebraic laws in groups `I` and `II` in Figure 2, 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 *(JIdem)* for the idempotence of the join operator over expressions in class $\widetilde{\mathbb{A}}$.

Most interesting in group `I` are rules *(JIdem)* and *(LIdem)*, established for fragment $\widetilde{\mathbb{A}}$. In fact, these rules do not generally hold for expressions that violate the incompatibility property. To give a concrete counterexample, substitute for instance expression A_1 (or A_2) from Example 5 for $\widetilde{A}$ in either *(JIdem)* or *(LIdem)*.

The rules for associativity, commutativity, and distributivity in group `II` speak for themselves. An outstanding question is whether they are complete w.r.t. all possible operator combinations. The lemma below rules out all combinations that are not explicitly stated:

LEMMA 3. Let $O_1 := \{\bowtie, \setminus, \between\}$ and $O_2 := O_1 \cup \{\cup\}$ be sets of operators. Then (1) operators $\setminus$ and $\between$ are neither associative nor commutative; (2) neither $\setminus$ nor $\between$ are left-distributive over $\cup$; (3) if $o_1 \in O_1$, $o_2 \in O_2$, and $o_1 \neq o_2$, then operator o_2 is neither left- nor right-distributive over operator o_1. □

Projection Pushing. Next, we shortly discuss the rules for projection pushing in group `III` of Figure 2. These rules are motivated by the desideratum that a good optimization scheme should include the possibility to choose among evaluation plans where projection is applied at different positions in the operator tree.

The first two rules in group `III`, *(PBaseI)* and *(PBaseII)*, are general-purpose rewritings for projection expressions. *(PBaseI)* shows that, when projecting a variable set that contains all possible variables (and possibly some additional variables S), the projection can be dropped. *(PBaseII)* complements *(PBaseI)* by showing that all variables in S that do not belong to $pVars(A)$ can be dropped when projecting S. The main benefit of these two rules stems from a combination with the other equivalences from group `III`, which may introduce such redundant variables within the rewriting process (cf. Example 6 below). The remaining six rules address the issue of pushing down projection expressions. Equivalence *(PFPush)* covers projection pushing into filter expressions, while *(PMerge)* shows that nested projection expression can be merged into a single projection. The four rules for the binary operations build upon the notion of possible variables. To give an example, rule *(PJPush)* 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 affect the result of the join (as they might cause incompatibility). Therefore, we define $S' := S \cup S'' = S \cup (pVars(A_1) \cap pVars(A_2))$ as an extension of S and project the variables in S' in the two subexpressions. *(PMPush)* and *(PLPush)* exhibit similar ideas. Note that we generally cannot eliminate the topmost projection, because $S' \supseteq S$.

EXAMPLE 6. Using rules *(PBaseI)*, *(PBaseII)*, and *(PJPush)*, we can easily prove that expressions B_1 and B_1^{opt} below, which select all persons that know at least one other person, are equivalent:

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

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

One may expect that B_1^{opt} is more efficient than B_1 on databases containing many *knows* relationships, where the early projection removes duplicates and accelerates the join operation. □

Filter Manipulation. Groups `IV` and `V` in Figure 2 contain rules to decompose, eliminate, and rearrange filter conditions. They form the basis for transferring relational algebra filter pushing techniques into the context of SPARQL. We emphasize, though, that these rules are more than simple translations of existing relational algebra equivalences: firstly, they rely on the SPARQL-specific concepts of possible and certain variables and, secondly, address specifics of SPARQL algebra, such as predicate *bnd* (cf. *(FBndI)-(FBndIV)*).

The first three equivalences in group `IV` cover decomposition and reordering of filter conditions, exploiting connections between SPARQL operators and the boolean connectives $\wedge$ and $\vee$. The subsequent four rules *(FBndI)-(FBndIV)* are SPARQL-specific and address the predicate *bnd*. They 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 1), so the filter can be dropped.

Finally, the rules in group `V` cover the issue of filter pushing. Particularly interesting are *(FJPush)* and *(FLPush)*, which crucially rely on the notions of possible and certain variables: the filter can be pushed inside the first component of a join $A_1 \bowtie A_2$ (or left outer join $A_1 \between A_2$) if each variable used inside the filter is a certain variable of A_1 (i.e., bound in every left side mapping) or is not a possible variable of A_2 (i.e., not bound in any right side mapping). This ultimately guarantees that the join (respectively left outer join) does not affect the validity of the filter condition. In general, the equivalences do not hold if this precondition is violated:

EXAMPLE 7. Consider the SPARQL algebra expressions $A_1 := [\![(?x, c, c)]\!]_D \between [\![(?x, d, ?y)]\!]_D$, $A_2 := [\![(?y, c, c)]\!]_D$ and the document $D := \{(c, c, c)\}$. We observe that $?y \notin cVars(A_1)$ and $?y \in pVars(A_2)$, so neither *(FJPush)* nor *(FLPush)* are applicable. Indeed, we have that $\sigma_{?y=c}(A_1 \bowtie A_2)$ and $\sigma_{?y=c}(A_1 \between A_2)$ evaluate to $\{\{?x \mapsto c, ?y \mapsto c\}\}$ on D, whereas $\sigma_{?y=c}(A_1) \bowtie A_2$ and $\sigma_{?y=c}(A_1) \between A_2$ both evaluate to $\emptyset$. □

I. Idempotence and Inverse

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

II. Associativity, Commutativity, Distributivity

$(A_1 \cup A_2) \cup A_3 \equiv A_1 \cup (A_2 \cup A_3)$ *(UAss)*
$(A_1 \bowtie A_2) \bowtie A_3 \equiv A_1 \bowtie (A_2 \bowtie A_3)$ *(JAss)*

$A_1 \cup A_2 \equiv A_2 \cup A_1$ *(UComm)*
$A_1 \bowtie A_2 \equiv A_2 \bowtie A_1$ *(JComm)*

$(A_1 \cup A_2) \bowtie A_3 \equiv (A_1 \bowtie A_3) \cup (A_2 \bowtie A_3)$ *(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)*
$(A_1 \cup A_2) \between A_3 \equiv (A_1 \between A_3) \cup (A_2 \between A_3)$ *(LUDistR)*

III. Projection Pushing

$\pi_{pVars(A) \cup S}(A) \equiv A$ *(PBaseI)*
$\pi_S(A) \equiv \pi_{S \cap pVars(A)}(A)$ *(PBaseII)*
$\pi_S(\sigma_R(A)) \equiv \pi_S(\sigma_R(\pi_{S \cup vars(R)}(A)))$ *(PFPush)*
$\pi_{S_1}(\pi_{S_2}(A)) \equiv \pi_{S_1 \cap S_2}(A)$ *(PMerge)*

Let $S'' := pVars(A_1) \cap pVars(A_2)$ and $S' := S \cup S''$. 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)*
$\pi_S(A_1 \between A_2) \equiv \pi_S(\pi_{S'}(A_1) \between \pi_{S'}(A_2))$ *(PLPush)*

IV. 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)*

V. 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)*
$\sigma_R(A_1 \between A_2) \equiv \sigma_R(A_1) \between A_2$ *(FLPush)*

VI. Minus and Left Outer Join Rewriting

$(A_1 \setminus A_2) \setminus A_3 \equiv (A_1 \setminus A_3) \setminus A_2$ *(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)*
$\widetilde{A_1} \between \widetilde{A_2} \equiv \widetilde{A_1} \between (\widetilde{A_1} \bowtie \widetilde{A_2})$ *(LJ)*

Let $?x \in V$ such that $?x \in cVars(A_2) \setminus pVars)(A_1)$. Then

$\sigma_{\neg bnd(?x)}(A_1 \between A_2) \equiv A_1 \setminus A_2$ *(FLBndI)*
$\sigma_{bnd(?x)}(A_1 \between A_2) \equiv A_1 \bowtie A_2$ *(FLBndII)*

Figure 2: Algebraic Equivalences, where $A, A_1, A_2, A_3 \in \mathbb{A}$; $\widetilde{A} \in \widetilde{\mathbb{A}}$; $S, S_1, S_2 \subset V$; R, R_1, R_2 Denote Filter Conditions

We conclude our discussion of filter manipulation with two additional rules to make atomic equalities in filter conditions explicit:

LEMMA 4. 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 URI or literal c. Then the following two equivalences hold.

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

(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. Observe that in both equivalences the filter expression must be embedded in a projection expressions that projects variable set $S \setminus \{?x\}$, i.e. not including variable $?x$ that is to be replaced (otherwise, $?x$ might appear in left side result mappings but not in right side mappings). Given our complete rewriting framework, this is not a major restriction: using the projection pushing rules from group `III`, we can push projections down on top of filter expressions and subsequently check if rule *(FElimI)* or *(FElimII)* applies.

We conclude our discussion of filter manipulation with an example that illustrates the filter manipulation rules and their possible interplay with the previous rules from groups `I-III` in Figure 2:

EXAMPLE 8. Consider the SPARQL algebra expression

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

which extracts persons (*?p*) with givenname (*?gn*) *“Sue”*, surname (*?sn*) different from *“Smith”*, and optionally their email (*?e*). It is left as an exercise to the reader to verify that, using rules from groups `I-V` in Figure 2, the expression can be transformed into

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

We may assume that the latter expression can be evaluated more efficiently than the original expression, because both filters are applied early; the atomic filter condition $?gn = \text{“Sue”}$ has been embedded into the triple pattern $[\![(?p, \textit{givenname}, ?gn)]\!]_D$. □

The example illustrates that the rewriting rules provided so far establish a powerful framework for finding alternate query 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 (see e.g. [31]). The study of such techniques is beyond the scope of this work.

Rewriting Closed World Negation. We conclude the discussion of SPARQL set algebra optimization with an 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 3), so the minus operator essentially implements closed world negation. 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 $\between$ and the definition of $\between$ relies on operator $\setminus$). As argued in [1], the lack of a syntactic counterpart complicates the encoding of queries involving negation and, as we shall see soon, poses specific challenges to query optimizers.

We also stress that, as discussed in Section 3.2, 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 [1] that operator $\setminus$ can be encoded at the syntactic level using operators OPT, FILTER, and (the negated) filter predicate *bnd*. The following example illustrates the idea behind the encoding of negation in SPARQL.

EXAMPLE 9. The SPARQL expression Q_1 and the corresponding algebra expression $C_1 := [\![Q_1]\!]_D$ below select all persons for which no name is specified in the data set.

$$Q_1 := ((?p, type, Person) \text{ OPT } ((?p, type, Person) \text{ AND } (?p, name, ?n))) \text{ FILTER } (\neg bnd(?n))$$

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

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, ⟕, $\bowtie$, and predicate *bnd* (thus using operator $\setminus$ implicitly, according to the semantics of ⟕). This translation seems overly complicated and we will now show that better translations exist for a large class of practical queries, using only $\setminus$, without ⟕, σ and predicate *bnd*.

We argue that the rewriting rules in Figure 2, group VI can accomplish such a rewriting in many practical cases. Most important in our context are rule *($\widetilde{LJ}$)*, which allows to eliminate redundant subexpressions in the right side of ⟕ expressions (over fragment $\widetilde{\mathbb{A}}$), and rule *(FLBndI)*. The idea is as follows. In a first step, we apply rule *($\widetilde{LJ}$)* to C_1 from Example 9, which gives us expression $C_1' := \sigma_{\neg bnd(?n)}([\![(?p, type, Person)]\!]_D ⟕ [\![(?p, name, ?n)]\!]_D)$. In a second step, we apply rule *(FLBndI)* to expression C_1' and obtain $C_1^{opt} := [\![(?p, type, Person)]\!]_D \setminus [\![(?p, name, ?n)]\!]_D$.

The construction in C_1, involving operators ⟕, $\bowtie$, σ, and predicate *bnd*, has been replaced by a simple minus expression in C_1^{opt}.

4.2 From Set to Bag Semantics

We now switch from set to bag algebra. Analogously to our discussion of SPARQL set algebra, we define a fragment called $\mathbb{A}^+$ that contains all bag algebra expressions. It differs from set algebra fragment $\mathbb{A}$ in that triple patterns are of the form $[\![t]\!]_D^+$ and therefore all operations are interpreted as operations over multi-sets. The ultimate goal in our analysis is to identify those equivalences from Section 4.1 that hold for bag algebra expressions.

We modify the definition of the incompatibility property (cf. Definition 7) as follows for bag semantics. A bag algebra expression A has the *incompatibility property* if, for every document D and result multi-set (Ω_D,m_D) obtained when evaluating A on D it holds that (i) each two distinct mappings in Ω_D are incompatible and (ii) $m_D(\mu) = 1$ for all $\mu \in \Omega_D$. The constraint (ii) arises from the fact that duplicate mappings are always compatible to each other (i.e., $\mu \sim \mu$) and may harm equivalences that – under set algebra – hold for expressions that exhibit the incompatibility property.

It turns out that we also need to adjust the definition of the fragment that satisfies the incompatibility property. We define the bag algebra class $\widetilde{\mathbb{A}^+}$ (the natural counterpart of set algebra class $\widetilde{\mathbb{A}}$) as the set of expressions built using operators $\bowtie$, $\setminus$, ⟕, σ, and (bracket-enclosed) triple patterns of the form $[\![t]\!]_D^+$. Then, in analogy to Lemma 2 for set semantics, we can show the following.

LEMMA 5. Every $\widetilde{A^+} \in \widetilde{\mathbb{A}^+}$ has the incompatibility property. $\square$

Given a set algebra equivalence (E), we say that (E) *carries over from set to bag algebra* if either (E) was specified for expressions $A, A_1, A_2, A_3 \in \mathbb{A}$ and it also holds for all bag algebra expressions $A, A_1, A_2, A_3 \in \mathbb{A}^+$ or (E) is specified for expressions from fragment $\widetilde{\mathbb{A}}$ and also holds for expressions from $\widetilde{\mathbb{A}^+}$.

EXAMPLE 10. Equivalence *(UIdem)* from Figure 2 does not carry over to bag algebra. To see why, consider $D := \{(c, c, c)\}$ and $A := [\![(c, c, ?x)]\!]_D \in \mathbb{A}^+$. The result of evaluating A on D is $(\{\{?x \mapsto c\}\}, m)$ with $m(\{?x \mapsto c\}) := 1$, but $A \cup A$ evaluates to $(\{\{?x \mapsto c\}\}, m')$ with $m'(\{?x \mapsto c\}) := 2$. $\square$

To keep the discussion short, we will not go to deep into detail, but only present the final outcome of our investigation.

THEOREM 7. All equivalences from Figure 2 except *(UIdem)* and *(FDecompII)* carry over to bag algebra. Further, rules *(FElimI)* and *(FElimII)* from Lemma 4 carry over to bag algebra. $\square$

4.3 Extensions and Practical Implications

We conclude with a discussion of implications for SPARQL engines that build upon the official W3C (bag) semantics. To this end, we switch from the algebraic level back to the syntax level and discuss conclusion we can draw for engines that follow the bag semantics approach. We start with a result on ASK queries:

LEMMA 6. Let Q, Q_1, Q_2 be SPARQL expressions. Then

- $[\![\text{ASK}(Q)]\!]_D \Leftrightarrow [\![\text{ASK}(Q)]\!]_D^+$
- $[\![\text{ASK}(Q_1 \text{ UNION } Q_2)]\!]_D \Leftrightarrow [\![\text{ASK}(Q_1)]\!]_D \vee [\![\text{ASK}(Q_2)]\!]_D$
- $[\![\text{ASK}(Q_1 \text{ OPT } Q_2)]\!]_D \Leftrightarrow [\![\text{ASK}(Q_1)]\!]_D$
- If $pVars([\![Q_1]\!]_D) \cap pVars([\![Q_2]\!]_D) = \emptyset$ then $[\![\text{ASK}(Q_1 \text{ AND } Q_2)]\!]_D \Leftrightarrow [\![\text{ASK}(Q_1)]\!]_D \wedge [\![\text{ASK}(Q_2)]\!]_D$. $\square$

The first bullet states that for ASK queries the set and bag semantics coincide. The remaining three rewritings are optimization rules, designed to reduce evaluation costs for ASK queries. For instance, the rule for operator OPT 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.

Like SQL, the SPARQL standard [32] proposes a set of solution modifiers. Our focus here is on the solution modifiers DISTINCT and REDUCED. The DISTINCT modifier removes duplicates from the result set, i.e. the result of evaluating SELECT DISTINCT$_S(Q)$ under bag semantics is obtained from $(\Omega, m) := [\![\text{SELECT}_S(Q)]\!]_D^+$ by replacing m by m' defined as $m'(\mu) := 1$ for all $\mu \in \Omega$ and $m'(\mu) := 0$ otherwise (DISTINCT and REDUCED queries make only sense under bag semantics, where duplicate answers may occur). While SELECT DISTINCT queries ensure that duplicates are eliminated, SELECT REDUCED queries *permit* to eliminate them; the idea is that optimizers can freely choose whether to eliminate duplicates or not, based on their internal processing strategy. We describe the result of SELECT REDUCED queries as the set of all valid answers, i.e. a set of mapping sets. We clarify the idea by example here; the interested reader will find a formal definition of both DISTINCT and REDUCED queries in Appendix A.6.

EXAMPLE 11. Let $Q := (?x, c, c) \text{ UNION } (c, c, ?x)$ and consider $Q_1 := \text{SELECT}_{?x}(Q)$, $Q_2 := \text{SELECT DISTINCT}_{?x}(Q)$, $Q_3 := \text{SELECT REDUCED}_{?x}(Q)$, and $D := \{(c, c, c)\}$. Then

$[\![Q_1]\!]_D^+ = (\{\{?x \mapsto c\}\}, m_1)$ where $m_1(\{?x \mapsto c\}) := 2$ and $m_1(\mu) := 0$ otherwise,
$[\![Q_2]\!]_D^+ = (\{\{?x \mapsto c\}\}, m_2)$ where $m_2(\{?x \mapsto c\}) := 1$ and $m_2(\mu) := 0$ otherwise,
$[\![Q_3]\!]_D^+ = \{(\{\{?x \mapsto c\}\}, m_1), (\{\{?x \mapsto c\}\}, m_2)\}$. $\square$

We next summarize relations between modifiers and semantics:

LEMMA 7. Let Q be a SPARQL expression and $S \subset V$. Then

- $[\![\text{SELECT}_S(Q)]\!]_D \cong [\![\text{SELECT DISTINCT}_S(Q)]\!]_D^+$
- $[\![\text{SELECT DISTINCT}_S(Q)]\!]_D^+ \in [\![\text{SELECT REDUCED}_S(Q)]\!]_D^+$
- There is some $(\Omega, m) \in [\![\text{SELECT REDUCED}_S(Q)]\!]_D^+$ such that $[\![\text{SELECT}_S(Q)]\!]_D \cong (\Omega, m)$. □

The first bullet shows that the (bag) semantics of SELECT DISTINCT queries coincides with the set semantics for the corresponding SELECT query. Bullets two and three imply that set semantics can also be used to evaluate SELECT REDUCED queries.

Summarizing all previous results, we observe that the (simpler) set semantics is applicable in the context of a large class of queries (i.e. all ASK, SELECT DISTINCT, and SELECT REDUCED queries). Engines that rely on SPARQL bag algebra for query evaluation may opt to implement a separate module for set semantics and switch between these modules based on the results above. We conclude with a lemma that identifies another large class of queries that can be evaluated using set semantics in place of bag semantics:

LEMMA 8. Let $Q \in \mathcal{AFO}$ and let $S \supseteq pVars([\![Q]\!]_D)$. Then $[\![\text{SELECT}_S(Q)]\!]_D \cong [\![\text{SELECT}_S(Q)]\!]_D^+$. □

5. SEMANTIC SPARQL OPTIMIZATION

We assume that the reader is familiar with the concepts of first-order logic, relational databases, and conjunctive queries. To be self-contained, we summarize the most important concepts below.

Conjunctive Queries. A conjunctive query (CQ) is an expression of the form $q : ans(\overline{x}) \leftarrow \varphi(\overline{x}, \overline{y})$, where φ is a conjunction of relational atoms, $\overline{x}$ and $\overline{y}$ are tuples of variables and constants, and every variable in $\overline{x}$ also occurs in φ. The semantics of q on database instance I is defined as $q(I) := \{\ \overline{a} \mid I \models \exists \overline{y} \varphi(\overline{a}, \overline{y})\ \}$.

Constraints. We specify constraints in form of first-order sentences over relational predicates. As special cases, we consider the well-known classes of tuple-generating dependencies (TGDs) and equality-generating dependencies (EGDs) [2], which cover most practical relations between data entities, such as functional and inclusion dependencies. Abstracting from details, TGDs and EGDs have the form $\forall \overline{x}(\varphi(\overline{x}) \rightarrow \exists \overline{y} \psi(\overline{x}, \overline{y}))$ and $\forall \overline{x}(\varphi(\overline{x}) \rightarrow x_i = x_j)$, respectively. Technical background can be found in Appendix A.3.

Chase. We assume familiarity with the basics of the chase algorithm (cf. [21, 2, 14]), a useful tool in semantic query optimization [14, 8]. 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 may be undefined, since the chase may fail or not terminate (see Appendix A.3). Still, there has been work on chase termination conditions that guarantee its termination in many practical cases (e.g. [28, 7, 22]).

Our SQO scheme builds on the Chase & Backchase (C&B) algorithm [8], which uses the chase as a subprocedure. Given a CQ q and a set of TGDs and EGDs Σ as input, the C&B algorithm returns the set of minimal rewritings (w.r.t. the number of atoms in the body) of q that are equivalent to q on every instance $D \models \Sigma$. We denote its output as $cb_\Sigma(q)$, if it is defined (the result is undefined if and only if the underlying chase result is undefined).

Constraints for RDF. We interpret an RDF database D as a ternary relation T that stores all the RDF triples and express constraints for RDF as first-order sentences over predicate T. For instance, the TGD $\forall x(T(x, \textit{rdf:type}, C) \rightarrow \exists y T(x, p, y))$ asserts that each resource that is typed with C also has the property p. When talking about constraints in the following, we always mean RDF constraints that are expressed as first-order sentences.

Before presenting our SQO scheme for SPARQL, we shortly investigate the general capabilities of SPARQL in the context of RDF constraints. More precisely, we are interested in the question whether the SPARQL query language can be used to express (i.e., check and encode) RDF constraints. An intuitive way to check if a constraint φ holds on some RDF document is by writing an ASK query that returns *true* on document D if and only if $D \models \varphi$. To be in line with previous investigations on the expressiveness of SPARQL, we extend our fragment by so-called empty graph patterns of the form $\{\}$ (which may be used in place of triple patterns), and a syntactic MINUS operator; we define their semantics as $[\![\{\}]\!]_D := \{\emptyset\}$ and $[\![Q_1 \text{ MINUS } Q_2]\!]_D := [\![Q_1]\!]_D \setminus [\![Q_2]\!]_D$.[4] Empty graph patterns are supported by the current W3C standard and operator MINUS is planned as a future extension.[5] Both constructs were also used in [1], where it is shown that SPARQL has the same expressiveness as Relational Algebra. Given the latter result and the close connection between RA and first-order logic, one may expect that (first-order logic) constraints can be expressed in SPARQL. The next theorem confirms this expectation:

THEOREM 8. Let φ be an RDF constraint. There is an ASK query Q s.t. for every document D it holds that $[\![Q]\!]_D \Leftrightarrow D \models \varphi$.□

The constructive proof in Appendix D.1 shows how to encode RDF constraints in SPARQL and makes the connection between SPARQL and first-order logic explicit. From a practical perspective, the result shows that SPARQL is expressive enough to deal with first-order constraints and qualifies SPARQL for extensions to encode user-defined constraints, e.g. in the style of SQL CREATE ASSERTION statements. In the remainder of the paper, we switch back to the original SPARQL fragment from Definitions 1 and 2.

5.1 SQO for SPARQL

The key idea of semantic query optimization is, given a query and a set of integrity constraints, to find minimal (or more efficient) queries that are equivalent to the original query on each database instance that satisfies the constraints. We define the problem for SPARQL as follows: given a SPARQL expression or query Q and a set of TGDs and EGDs Σ over the RDF database, we want to enumerate (minimal) expressions or queries Q' that are equivalent to Q on every database D such that $D \models \Sigma$. In that case, we say that Q and Q' are Σ-equivalent and denote this by $Q \equiv_\Sigma Q'$.

The constraints that are given as input might have been specified by the user, automatically extracted from the underlying database, or – in our setting – may be implicitly given by the semantics of RDFS when SPARQL is coupled with an RDFS inference system. In fact, one aspect that served as a central motivation for the investigation of SQO for SPARQL is the close connection between constraints and the semantics of RDF and RDFS [29]. To be concrete, RDF(S) comes with a set of reserved URIs with predefined semantics, such as *rdf:type* for typing entities, or *rdfs:domain* and *rdfs:range* for fixing the domain and range of properties (cf. [12]). As an example, let us consider the fixed RDF database

$$D := \{(\textit{knows}, \textit{rdfs:domain}, \textit{Person}), (\textit{knows}, \textit{rdfs:range}, \textit{Person}), (\textit{P1}, \textit{knows}, \textit{P2})\}.$$

According to the semantics of *rdfs:domain* (*rdfs:range*), each URI or blank node that is used in the subject (object) position of

[4]The extension is necessary to obtain the result in Theorem 8, see Remark 1 in Appendix D.1 for more background information.

[5]`http://www.w3.org/TR/2009/WD-sparql-features-20090702/`

triples with predicate *knows* is implicitly of type *Person*, i.e. for D the semantics implies two fresh triples $t_1 := (P1, \textit{rdf:type}, \textit{Person})$ and $t_2 := (P2, \textit{rdf:type}, \textit{Person})$. The SPARQL query language ignores the semantics of this vocabulary and operates on the RDF database "as is", thus disregarding triples that are not explicitly contained in the database but only implicit by the RDF(S) semantics. Still, when SPARQL is coupled with an RDF(S) inferencing system, it implicitly operates on top of the implied database, which satisfies all the constraints imposed by RDF(S) vocabulary. For instance, implied databases always satisfy the two constraints

$$\varphi_d := \forall p, c, x, y(T(p, \textit{rdfs:domain}, c), T(x, p, y) \rightarrow T(x, \textit{rdf:type}, c)),$$
$$\varphi_r := \forall p, c, x, y(T(p, \textit{rdfs:range}, c), T(x, p, y) \rightarrow T(y, \textit{rdf:type}, c)),$$

which capture the semantics of *rdfs:domain* (φ_d) and *rdfs:range* (φ_r). Thus, whenever SPARQL is evaluated on top of an RDFS inferencing engine, we can use these constraints (and others that are implicit by the semantics of RDF(S) [12, 11]) for SQO.

We note that constraint-based query optimization in the context of RDFS inference has been discussed before in [11]. 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 [17], for instance, we proposed 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 [11], our approach addresses the specifics of SPARQL, e.g. we also provide rules for the semantic optimization of queries that involve operator OPT.

Outline. We now come to the 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 subqueries of the input query into conjunctive queries. In a second step, we use the C&B algorithm to minimize these CQs, translate the minimized CQs (i.e., the output of C&B) back into SPARQL, and substitute them for the initial subqueries. By default, the C&B algorithm returns Σ-equivalent queries that are minimal w.r.t. the number of atoms in the body of the query. Yet, as described in [8], the C&B algorithm also can be coupled with a cost estimation function and in that case would return queries that are minimal w.r.t. the cost function. In the absence of a cost measure, we focus on the minimality property in the following, but point out that the approach per se also supports more sophisticated 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 5.1.1. Complementarily, in Section 5.1.2 we discuss SPARQL-specific rules that allow for the semantic optimization of complex queries involving operators FILTER and OPT.

5.1.1 Optimizing AND-only Blocks

We start with translation functions that map SPARQL AND-only queries to conjunctive queries and vice versa:

DEFINITION 9. Let $S \subset V$ and let $Q \in \mathcal{A}^\pi$ be 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 $cq(Q) := q$, where q is defined as

$$q := ans(\overline{s}) \leftarrow T(s_1, p_1, o_1), \ldots, T(s_n, p_n, o_n)$$

and tuple $\overline{s}$ contains exactly the variables from S.

Further, we define the back-translation $cq^{-1}(q)$ as follows. It takes a CQ in the form of q 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. □

EXAMPLE 12. Consider the SPARQL AND-only query
$Q := \text{SELECT}_{?p1,?p2}((?p1, \textit{knows}, ?p2) \text{ AND } (?p1, \textit{rdf:type}, \textit{Person}) \text{ AND } (?p2, \textit{rdf:type}, \textit{Person}))$. Then

$$cq(Q) = ans(?p1, ?p2) \leftarrow T(?p1, \textit{knows}, ?p2), T(?p1, \textit{rdf:type}, \textit{Person}), T(?p2, \textit{rdf:type}, \textit{Person})$$

and $cq^{-1}(cq(Q)) = Q$. As another example, we can observe that $cq^{-1}(ans(?x) \leftarrow T(\text{"}a\text{"}, p, ?x))$ is undefined, because expression $\text{SELECT}_{?x}((\text{"}a\text{"}, p, ?x))$ has literal *"a"* in subject position. □

Although defined for $\mathcal{A}^\pi$ queries, the translation scheme can easily be applied to $\mathcal{A}$ expressions (i.e., AND-blocks in queries): every $Q \in \mathcal{A}$ is equivalent to the $\mathcal{A}^\pi$ query $\text{SELECT}_{pVars([\![Q]\!]_D)}(Q)$.

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 9. Let Q be an $\mathcal{A}^\pi$ query, let 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 $cq^{-1}(q) \equiv_\Sigma Q$. □

Lemma 9 formalizes the key idea of our SQO scheme: given that the chase result for $cq(Q)$ with Σ is defined for some AND-only query Q, we can apply the C&B algorithm to $cq(Q)$ and translate the resulting minimal queries back into SPARQL, to obtain SPARQL AND-only queries that are Σ-equivalent to Q.

EXAMPLE 13. Consider query Q from Example 12 and query $Q^{opt} := \text{SELECT}_{?p1,?p2}((?p1, \textit{knows}, ?p2))$. Further consider the constraints φ_d, φ_r from Section 5.1 and define $\Sigma := \{\varphi_d, \varphi_r\}$. We have $cq(Q^{opt}) \in cb_\Sigma(cq(Q))$ and it follows from Lemma 9 that $cq^{-1}(cq(Q^{opt})) = Q^{opt} \equiv_\Sigma Q$. An engine that builds upon an RDFS inference engine thus may evaluate Q^{opt} in place of Q. □

Lemma 9 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 9 is not complete:

EXAMPLE 14. Consider the SPARQL queries

$$Q_1 := \text{SELECT}_{?x}((?x, a, \text{"}l\text{"})),$$
$$Q_2 := \text{SELECT}_{?x}((?x, a, \text{"}l\text{"}) \text{ AND } (?x, b, c)),$$

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 appear in subject position, which is invalid RDF. Contrarily, observe that $cq(Q_1) \not\equiv_\Sigma cq(Q_2)$. To see why, consider for example the relational instance $I := \{T(a, a, \text{"}l\text{"}), T(\text{"}l\text{"}, a, a)\}$, where $I \models \Sigma$, $(cq(Q_1))(I) = \{(a)\}$, but $(cq(Q_2))(I) = \emptyset$. Therefore, our scheme would not detect Σ-equivalence between Q_1 and Q_2. □

Arguably, Example 14 presents a constructed scenario and it seems reasonable to assume that such situations (which in some sense contradict to the type restrictions of RDF) barely 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 10. Let D be an RDF database and let Q be an $\mathcal{A}^\pi$ query 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 $cq^{-1}((cq(Q'))^\Sigma) \in \mathcal{A}^\pi$ then $Q' \in cq^{-1}(cb_\Sigma(cq(Q))) \Leftrightarrow Q' \equiv_\Sigma Q$ and Q' minimal. □

5.1.2 SPARQL-specific Optimization

By now we have a mechanism to enumerate equivalent minimal queries of AND-only (sub)queries. Next, we present extensions beyond AND-only queries. We start with the FILTER operator:

LEMMA 11. Let $Q_1, Q_2 \in \mathcal{A}$, $S \subset V \setminus \{?y\}$ a set of variables, Σ be a set of TGDs and EGDs, D be a document s.t. $D \models \Sigma$, and $?x, ?y \in pVars([\![Q_2]\!]_D)$. By $Q_2 \frac{?x}{?y}$ we denote the query obtained from Q_2 by replacing each occurrence of $?y$ through $?x$. Then:

(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 Lemma 11 is as follows. Using standard chase techniques we can check if the preconditions hold; if this is the case, we may exploit the equivalences in the conclusion. Informally, *(FSI)* states that, if the constraints imply equivalence between $?x$ and $?y$, we can replace each occurrence of $?y$ by $?x$ if $?y$ is projected away (observe that $S \subset V \setminus \{?y\}$). Under the same precondition, *(FSII)* shows that a filter $\neg(?x = ?y)$ is never satisfied. Finally, *(FSIII)* detects contradicting filters of the form $\neg bnd(?x)$. Our next results are semantic rewriting rules for operator OPT:

LEMMA 12. Let $Q_1, Q_2, Q_3 \in \mathcal{A}$ and $S \subset V$. Then

(OSI) If $Q_1 \equiv_\Sigma \text{SELECT}_{pVars([\![Q_1]\!]_D)}(Q_1 \text{ AND } Q_2)$ then $Q_1 \text{ OPT } Q_2 \equiv_\Sigma Q_1 \text{ AND } Q_2$.

(OSII) If $Q_1 \equiv_\Sigma Q_1 \text{ AND } Q_2$ then $Q_1 \text{ OPT } (Q_2 \text{ AND } Q_3) \equiv_\Sigma Q_1 \text{ OPT } Q_3$. □

Rule *(OSI)* shows that OPT can be replaced by AND if the OPT subexpression is implied by the constraint set; *(OSII)* eliminates redundant AND expressions in OPT clauses. We illustrate *(OSI)*:

EXAMPLE 15. Consider patterns $t_1 := (?p, \textit{rdf:type}, \textit{Person})$, $t_2 := (?p, \textit{name}, ?n)$, and $t_3 := (?p, \textit{age}, ?a)$ and define $Q_o := \text{SELECT}_{?p,?n,?a}(t_1 \text{ OPT } (t_2 \text{ AND } t_3))$. Let $\Sigma := \{\alpha_1, \alpha_2\}$ with

$\alpha_1 := \forall x(T(x, \textit{rdf:type}, \textit{Person}) \rightarrow \exists y T(x, \textit{name}, y))$,
$\alpha_2 := \forall x(T(x, \textit{rdf:type}, \textit{Person}) \rightarrow \exists y T(x, \textit{age}, y))$.

It is easily verified that $t_1 \equiv_\Sigma \text{SELECT}_{?p}(t_1 \text{ AND } (t_2 \text{ AND } t_3))$, so rewriting rule *(OSI)* is applicable. Therefore, we shall conclude that $\text{SELECT}_{?p,?n,?a}(t_1 \text{ AND } (t_2 \text{ AND } t_3)) \equiv_\Sigma Q_o$. □

The rules in Lemma 11 and 12 exemplify an approach to rule-based semantic SPARQL optimization and can be extended on demand by user-defined optimization rules. Particularly rule *(OSI)* seems useful in practice: in the Semantic Web, queries are often submitted to databases hidden behind SPARQL endpoints, so users may not be aware of integrity constraints that hold in the database. In such cases, they may specify parts of the query as optional, not to miss relevant answers. If there are constraints that guarantee the presence of such data, the engine can replace the OPT operator by AND, which may accelerate query processing.

We conclude with the remark that optimization schemes may couple our algebraic and semantic techniques, e.g. by decomposing and moving filter conditions using the rules for filter manipulation in Figure 2 to create situations in which the above lemmas apply.

Acknowledgments

The authors wish to thank the anonymous reviewers for their comments and suggestions and Claudio Gutierrez for fruitful discussions on the expressive power of SPARQL.

6. REFERENCES

[1] R. Angles and C. Gutierrez. The Expressive Power of SPARQL. In *ISWC*, pages 114–129, 2008.
[2] C. Beeri and M. Y. Vardi. A Proof Procedure for Data Dependencies. *J. ACM*, 31(4):718–741, 1984.
[3] C. Weiss et al. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *VLDB*, pages 1008–1019, 2008.
[4] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based Approach to Semantic Query Optimization. *TODS*, 15(2):162–207, 1990.
[5] R. Cyganiac. A relational algebra for SPARQL. Technical report, HP Laboratories Bristol, 2005.
[6] D. J. Abadi et al. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422, 2007.
[7] A. Deutsch, A. Nash, and J. Remmel. The Chase Revisited. In *PODS*, pages 149–158, 2008.
[8] A. Deutsch, L. Popa, and V. Tannen. Query Reformulation with Constraints. *SIGMOD Record*, 35(1):65–73, 2006.
[9] E. I. Chong et al. An Efficient SQL-based RDF Querying Scheme. In *VLDB*, pages 1216–1227, 2005.
[10] François Bry et al. Foundations of Rule-based Query Answering. In *Reasoning Web*, pages 1–153, 2007.
[11] G. Serfiotis et al. Containment and Minimization of RDF/S Query Patterns. In *ISWC*, pages 607–623, 2005.
[12] C. Gutierrez, C. A. Hurtado, and A. O. Mendelzon. Foundations of Semantic Web Databases. In *PODS*, pages 95–106, 2004.
[13] A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *LA-WEB*, pages 71–80, 2005.
[14] D. S. Johnson and A. Klug. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. In *PODS*, pages 164–169, 1982.
[15] J. J. King. QUIST: a system for semantic query optimization in relational databases. In *VLDB*, pages 510–517, 1981.
[16] L. Sidirourgos et al. Column-store Support for RDF Data Management: not all swans are white. In *VLDB*, page 1553, 2008.
[17] G. Lausen, M. Meier, and M. Schmidt. SPARQLing Constraints for RDF. In *EDBT*, pages 499–509, 2008.
[18] Linked Data. `http://linkeddata.org/`.
[19] M. Schmidt et al. An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In *ISWC*, pages 82–97, 2008.
[20] M. Stocker et al. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *WWW*, pages 595–604, 2008.
[21] D. Maier, A. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. In *SIGMOD*, pages 152–152, 1979.
[22] M. Meier, M. Schmidt, and G. Lausen. On Chase Termination Beyond Stratification. In *VLDB*, 2009.
[23] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
[24] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. Technical report, arXiv:0605124 cs.DB, 2006.
[25] J. Pérez, M. Arenas, and C. Gutierrez. Semantics of SPARQL, 2006. TR/DCC-2006-16, Universidad de Chile.
[26] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
[27] A. Polleres. From SPARQL to Rules (and back). In *WWW*, pages 787–796, 2007.
[28] R. Fagin et al. Data Exchange: Semantics and Query Answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.
[29] Resource Description Framework (RDF). `http://www.w3.org/RDF/`.
[30] S. Alexaki et al. On Storing Voluminous RDF descriptions: The case of Web Portal Catalogs. In *WebDB*, pages 43–48, 2001.
[31] J. Smith and P. Chang. Optimizing the Performance of a Relational Algebra Database Interface. *Commun. ACM*, 18(10):568–579, 1975.
[32] SPARQL Query Language for RDF. W3C Recommendation, 15 Januray 2008.
[33] L. J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3:1–22, 1976.
[34] Y. Theoharis et al. Benchmarking Database Representations of RDF/S Stores. In *ISWC*, pages 685–701, 2005.

APPENDIX

A. ADDITIONAL DEFINITIONS

We give some background definitions that are not relevant for the understanding of the paper itself, but are important for the understanding of the proofs in the remainder of the appendix and clarify technical issues that were left out in the main section of the paper.

A.1 Semantics of Filter Conditions

DEFINITION 10 (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 c equals to d.
- 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 it holds that $\mu(?x) = \mu(?y)$.
- 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 $\mu \models R_2$. □

A.2 SPARQL Bag Semantics

Implementing the ideas sketched in Section 2, we formally define the bag semantics for SPARQL as follows. First, we overload the algebraic operations from Definition 3:

DEFINITION 11 (SPARQL BAG ALGEBRA). Let R be a filter condition, $S \subset V$ a finite set of variables, and $M := (\Omega, m)$, $M_l := (\Omega_l, m_l)$, $M_r := (\Omega_r, m_r)$ be mapping multi-sets We define the operations join ($\bowtie$), union ($\cup$), set minus ($\setminus$), left outer join ($\bowtie$), 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
$m'(\mu) := \sum_{(\mu_l,\mu_r)\in\{(\mu_l^*,\mu_r^*)\in\Omega_l\times\Omega_r \mid \mu_l^*\cup\mu_r^*=\mu\}} (m_l(\mu_l) * m_r(\mu_r))$
for all $\mu \in \mathcal{M}$.

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

$M_l \setminus M_r := (\Omega', m')$, where
$\Omega' := \{\mu_l \in \Omega_l \mid$ for all $\mu_r \in \Omega_r : \mu_l \not\sim \mu_r\}$ and
$m'(\mu) := m_l(\mu)$ if $\mu \in \Omega'$, and $m'(\mu) := 0$ otherwise.

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

$\pi_S(M) := (\Omega', m')$, 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
$m'(\mu) := \sum_{\mu_+ \in \{\mu_+^* \in \Omega \mid \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m(\mu_+)$ for all $\mu \in \mathcal{M}$.

$\sigma_R(M) := (\Omega', m')$, where
$\Omega' := \{\mu \in \Omega \mid \mu \models R\}$ and
$m'(\mu) := m(\mu)$ if $\mu \in \Omega'$, and $m'(\mu) := 0$, otherwise.

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

The above definition exactly corresponds to Definition 3 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), but additionally fixes the multiplicities for generated set members (cf. function m'). It is easily verified that the above definition always returns multi-sets valid according to Definition 5. Now we are in the position to the define the bag semantics for SPARQL:

DEFINITION 12 (SPARQL BAG SEMANTICS). Let D be an RDF database, t be a triple pattern, 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 recursively as follows.

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

Note that this definition is identical to Definition 4, except for the case of triple pattern evaluation, where we represent the result of evaluating a triple pattern as a multi-set. Hence, when evaluating a SPARQL expression bottom-up using bag semantics, algebraic operations will always be interpreted as multi-set operations.

A.3 TGDs, EGDs, and Chase

We fix three pairwise disjoint infinite sets: the set of *constants* Δ, the set of *labeled nulls* Δ_{null}, and the set of *variables* V. Often we will denote a sequence of variables, constants or labeled nulls by $\overline{a}$. A *database schema* $\mathcal{R}$ is a finite set of relational symbols $\{R_1, ..., R_n\}$. To every $R \in \mathcal{R}$ we assign a natural number $ar(R)$ called its *arity*. A *database instance* I is a finite set of $\mathcal{R}$-atoms that contains only elements from $\Delta \cup \Delta_{null}$ in their positions. The domain of I, $dom(I)$, is the set of elements appearing in I.

DEFINITION 13 (TUPLE-GENERATING DEPENDENCY). A *tuple-generating dependency* (TGD) is a first-order sentence

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

such that (a) both ϕ and ψ are conjunctions of atomic formulas (possibly with parameters from Δ), (b) ψ is not empty, (c) ϕ is possibly empty, (d) both ϕ and ψ do not contain equality atoms and (e) all variables from $\overline{x}$ that occur in ψ must also occur in ϕ. We denote by $pos(\alpha)$ the set of positions in ϕ. □

DEFINITION 14 (EQUALITY-GENERATING DEPENDENCY). An equality-generating dependency (EGD) is a first-order sentence

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

where x_i, x_j occur in ϕ and ϕ is a non-empty conjunction of equality-free $\mathcal{R}$-atoms (possibly with parameters from Δ). We denote the set of positions in ϕ by $pos(\alpha)$. □

DEFINITION 15 (HOMOMORPHISM). A *homomorphism* from a set of atoms A_1 to a set of atoms A_2 is a mapping $\mu : \Delta \cup V \rightarrow \Delta \cup \Delta_{null}$ such that the following conditions hold: (i) if $c \in \Delta$, then $\mu(c) = c$ and (ii) if $R(c_1, ..., c_n) \in A_1$, then $R(\mu(c_1), ..., \mu(c_n)) \in A_2$. □

We are now in the position to introduce the chase algorithm. Let Σ be a set of TGDs and EGDs and I an instance, represented as a set of atoms. We say that a TGD $\forall \overline{x}\varphi \in \Sigma$ is applicable to I if there

is a homomorphism μ from $body(\forall\overline{x}\varphi)$ to I and μ cannot be extended to a homomorphism $\mu' \supseteq \mu$ from $head(\forall\overline{x}\varphi)$ to I. In such a case the chase step $I \overset{\forall\overline{x}\varphi,\mu(\overline{x})}{\longrightarrow} J$ is defined as follows. We define a homomorphism ν as follows: (a) ν agrees with μ on all universally quantified variables in φ, (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 $\nu(y) := n_y$. We set J to $I \cup \nu(head(\forall\overline{x}\varphi))$. We say that an EGD $\forall\overline{x}\varphi \in \Sigma$ is applicable to I if there is a homomorphism μ from $body(\forall\overline{x}\varphi)$ to I and it holds that $\mu(x_i) \neq \mu(x_j)$. In this case the chase step $I \overset{\forall\overline{x}\varphi,\mu(\overline{x})}{\longrightarrow} J$ is defined as follows. We set J to

- I except that all occurrences of $\mu(x_j)$ are substituted by $\mu(x_i) =: a$, if $\mu(x_j)$ is a labeled null,
- I except that all occurrences of $\mu(x_i)$ are substituted by $\mu(x_j) =: a$, if $\mu(x_i)$ is a labeled null,
- undefined, if both $\mu(x_j)$ and $\mu(x_i)$ 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 folklore that two different chase orders always lead to homomorphically equivalent results, if these exist. Therefore, we write I^Σ for the result of the chase on an instance I under Σ. It has been shown in [21] that $I^\Sigma \models \Sigma$. If a chase step cannot be performed (e.g., because a homomorphism would have to equate two constants) or in case of an infinite chase sequence, the chase result is undefined.

A.4 OPT-rank

Formally, the *rank* of an expression is defined as follows.

DEFINITION 16 (OPT-RANK). The nesting depth of OPT expressions in SPARQL expression Q, called OPT-rank $rank(Q)$, is defined recursively on the structure of Q as

$$\begin{aligned}
&rank(\text{t}) &&:= 0,\\
&rank(Q_1 \text{ FILTER } R) &&:= rank(Q_1),\\
&rank(Q_1 \text{ AND } Q_2) &&:= max(rank(Q_1), rank(Q_2)),\\
&rank(Q_1 \text{ UNION } Q_2) &&:= max(rank(Q_1), rank(Q_2)),\\
&rank(Q_1 \text{ OPT } Q_2) &&:= max(rank(Q_1), rank(Q_2)) + 1,
\end{aligned}$$

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

A.5 Function pVars(A)

We provide a formal definition of function $pVars(A)$:

DEFINITION 17 (FUNCTION *pVars*). Let A be a SPARQL set algebra expression. Function $pVars(A)$ extracts the set of so-called *possible variables* from a SPARQL algebra expression:

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

A.6 DISTINCT and REDUCED Queries

We provide a definition of DISTINCT and REDUCED queries.

DEFINITION 18 (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 12 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 m is defined as $m(\mu) := 1$ if $m^+(\mu) \geq 1$ and $m(\mu) := 0$ otherwise. □

DEFINITION 19 (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 12 to SELECT REDUCED queries. Let $(\Omega^+, m^+) := [\![\text{SELECT}_S(Q)]\!]_D^+$. The solution to query SELECT REDUCED$_S(Q)$ is the set of mapping sets of the form (Ω^+, m) s.t. for all $\mu \in \mathcal{M}$ it holds that (i) $m^+(\mu) = 0 \rightarrow m(\mu) = 0$ and (ii) $m^+(\mu) > 0 \rightarrow m(\mu) \geq 1 \wedge m(\mu) \leq m^+(\mu)$.□

B. PROOFS OF COMPLEXITY RESULTS

B.1 Proof of Lemma 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 $[\![Q]\!]_D^+ := (\Omega^+, m^+)$. This notation is justified by the property that $m^+(\mu) \geq 1$ for each $\mu \in \Omega^+$. Note that the SPARQL bag algebra operators introduced in Definition 3 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 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 3 and 11 it follows immediately that $\Omega = \Omega^+$ and thus $\mu \in [\![Q]\!]_D \Leftrightarrow \mu \in [\![Q]\!]_D^+$, which completes the basic case. We therefore may assume that the hypothesis holds for each 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 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 $\mu \in [\![P_1]\!]_D \setminus [\![P_2]\!]_D \rightarrow \mu \in [\![P_1]\!]_D^+ \setminus [\![P_2]\!]_D^+$. 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) $Q :=$ SELECT$_S(P)$ and (5) $Q := P$ FILTER R are easily obtained by application of the induction hypothesis. □

B.2 Proof of Theorem 2

Theorem 2(1): We provide a PTIME-algorithm that solves the EVALUATION problem for fragment $\mathcal{FU}$. It is defined recursively 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 2(2): To prove that the EVALUATION problem for fragment $\mathcal{AU}$ is NP-complete we need to show membership and hardness. The hardness part was sketched in Section 3.1, so we restrict on proving membership here. 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 $\cup$. Clearly, this algorithm can be implemented by a non-deterministic Turing Machine that runs in polynomial time. □

B.3 Proof of Theorem 3

We reduce QBF, a prototypical PSPACE-complete problem, to EVALUATION for class $\mathcal{AO}$. The hardness part of the proof is in parts inspired by the proof of Theorem 1(3) stated before, which has been formally proven in [24]: 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. Membership in PSPACE, and hence completeness, then follows from the PSPACE membership of class $\mathcal{AOU} \supset \mathcal{AO}$ (cf. Theorem 1(3)). Formally, QBF is defined as follows.[6]

> 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[7]. 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

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

where the second and the third part of the union should be understood as a syntactic replacement of v by variable names in V_{C_i} and V_ψ, respectively (and the variable names are understood as URIs).

[6]Note that, like the proof in [24], 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.

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

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 a fresh URI for 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

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

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 $P_\psi := P_{C_1}$ AND ... 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 $?V_i \in dom(\mu)$, or define either $\rho_\mu(v_i) := 0$ or $\rho_\mu(v_i) := 1$ if $?V_i \notin dom(\mu)$; 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; then, the value of the variable is not relevant to obtain a satisfying truth assignment and we can randomly choose a value for v_i in truth assignment ρ_μ.

Given P_ψ, we can encode the quantifier-sequence using a series of nested OPT statements as shown in [24]. 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

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

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

$$\begin{aligned} P_\varphi := (a, true, ?B_0) \text{ OPT } (P_1 \text{ OPT } (Q_1 \\ \text{OPT } (P_2 \text{ OPT } (Q_2 \\ \ldots \\ \text{OPT } (P_m \text{ OPT } (Q_m \text{ AND } P_\psi)) \ldots))))). \end{aligned}$$

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 [24] for details. □

B.4 Proof of Theorem 4

Adapting the idea from the proof of Theorem 3, we present a reduction from QBF to the EVALUATION problem for SPARQL queries, where the 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 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 13. Let

- $Q, Q_1, Q_2, \ldots, Q_n$ ($n \geq 2$) be SPARQL expressions,
- S denote the set of all variables 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

$$Q' := ((\ldots((Q \text{ OPT } V_2) \text{ OPT } V_3)\ldots) \text{ OPT } V_n),$$
$$Q'' := ((\ldots((Q_1 \text{ OPT } (Q_2 \text{ OPT } V_2)) \text{ OPT } (Q_3 \text{ OPT } V_3)) \ldots) \text{ OPT } (Q_n \text{ OPT } V_n)), \text{ where}$$
$V_i := (a, true, ?V_i)$ and $\overline{V}_i := (a, false, ?V_i)$.

The following claims hold.

(1) $[\![Q']\!]_D = \{\mu \cup \{?V_2 \mapsto 1, \ldots, ?V_n \mapsto 1\} \mid \mu \in [\![Q]\!]_D\}$
(2) $[\![Q' \text{ OPT } (Q_1 \text{ AND } Q_2 \text{ AND } \ldots \text{ AND } Q_n)]\!]_D = [\![Q' \text{ OPT } ((\ldots((Q'' \text{ OPT } \overline{V}_2) \text{ OPT } \overline{V}_3)\ldots) \text{ OPT } \overline{V}_n)]\!]_D$ □

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 16. Consider the database

$$D := \{(a, \mathit{false}, 0), (a, \mathit{true}, 1), (a, \mathit{tv}, 0), (a, \mathit{tv}, 1)\}$$

and the expressions

$$Q := (a, tv, ?a) \quad \text{, so } [\![Q]\!]_D = \{\{?a \mapsto 0\}, \{?a \mapsto 1\}\},$$
$$Q_1 := (a, tv, ?b) \quad \text{, so } [\![Q_1]\!]_D = \{\{?b \mapsto 0\}, \{?b \mapsto 1\}\},$$
$$Q_2 := (a, true, ?b), \text{ so } [\![Q_2]\!]_D = \{\{?b \mapsto 1\}\}.$$

Concerning claim (1) of Lemma 13, we observe that

$$[\![Q']\!]_D = [\![Q \text{ OPT } V_2]\!]_D = [\![Q \text{ OPT } (a, true, ?V_2)]\!]_D = \{\{?a \mapsto 0, ?V_2 \mapsto 1\}, \{?a \mapsto 1, ?V_2 \mapsto 1\}\},$$

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

$$[\![Q' \text{ OPT } (Q_1 \text{ AND } Q_2)]\!]_D = [\![Q']\!]_D \bowtie \{\{?b \mapsto 1\}\} = \{\{?a \mapsto 0, ?b \mapsto 1, ?V_2 \mapsto 1\}, \{?a \mapsto 1, ?b \mapsto 1, ?V_2 \mapsto 1\}\}$$

is equal to the right side expression

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

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 $\{?b \mapsto 1, ?V_2 \mapsto 1\}$), but does not appear in mappings that are generated from the $\setminus$ part of the left outer join (cf. mapping $\{?b \mapsto 0\}$). 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, we simulated the behavior of the AND expression (the syntactic counterparts of operator $\bowtie$) using OPT operators. □

Proof of Lemma 13

Lemma 13(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 $\mathit{dom}(D')$ does not contain the URI *true* it follows that no triple in D' matches the triple pattern $V_i := (a, \mathit{true}, ?V_i)$, so we have that $[\![V_i]\!]_D = \{\{?V_i \mapsto 1\}\}$. Hence, what expression Q' does is to successively extend each mapping $\mu \in [\![Q]\!]_D$ by the (compatible) mappings $\{?V_2 \mapsto 1\}$, …, $\{?V_n \mapsto 1\}$, so the claim holds.

Lemma 13(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 details, we give the intuition of 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 binding $?V_i \mapsto 1$. 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 algebraic laws, such as distributivity of $\bowtie$ over $\cup$ (cf. Figure 2) we bring $[\![Q'']\!]_D$ into the form

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

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. $?V_i \notin \mathit{dom}(\mu)$. We observe that, in contrast, for each mapping μ that is generated by the join part, we have that $\mathit{dom}(\mu) \supseteq \{?V_2, \ldots, ?V_n\}$ and, even more, $\mu(?V_i) =$

1, for $2 \leq i \leq n$. Hence, these mappings are identified by the property $\mu(?V_2) = \mu(?V_3) = \cdots = \mu(?V_n) = 1$. Going one step further, we next 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 $[\![Q'']\!]_D$, but each mapping $\mu \in [\![Q'']\!]_D$ is extended by $?V_i \mapsto 0$ for all variables $?V_i \notin dom(\mu)$. 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 $\mu \in [\![P']\!]_D$ it holds that $dom(\mu) \supseteq \{?V_2, \ldots, ?V_n\}$; second, for those $\mu \in [\![P']\!]_D$ that evolve from the join part of $[\![Q'']\!]_D$ we have that $\mu(?V_2) = \cdots = \mu(?V_n) = 1$; third, for those $\mu \in [\![P']\!]_D$ 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 $[\![Q' \text{ OPT } P']\!]_D$. From claim (1) of the lemma we know that each mapping in $[\![Q']\!]_D$ maps all $?V_i$ to 1. Hence, when computing $[\![Q' \text{ OPT } P']\!]_D = [\![Q']\!]_D \bowtie [\![P']\!]_D$, the bindings $?V_i \mapsto 1$ for all $i \in \{2, \ldots, n\}$ in every $\mu \in [\![Q']\!]_D$ assert that the mappings in $[\![Q']\!]_D$ are pairwise incompatible with those mapping from $[\![P']\!]_D$ 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 $[\![Q' \text{ OPT } P']\!]_D$. Hence,

$$\begin{aligned}
&[\![Q' \text{ OPT } P']\!]_D = [\![Q']\!]_D \bowtie [\![P']\!]_D \\
&= [\![Q']\!]_D \bowtie [\![Q_1 \text{ AND } Q_2^{V_2} \text{ AND } Q_3^{V_3} \text{ AND } \ldots \text{ AND } Q_n^{V_n}]\!]_D \\
&= [\![Q' \text{ OPT } (Q_1 \text{ AND } Q_2^{V_2} \text{ AND } Q_3^{V_3} \text{ AND } \ldots \text{ AND } Q_n^{V_n})]\!]_D.
\end{aligned}$$

Even more, we know from claim (1) of the lemma that all $?V_i$ are bound to 1 for each $\mu \in [\![Q']\!]_D$. 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 $[\![Q' \text{ OPT } P']\!]_D$:

$$\begin{aligned}
&[\![Q' \text{ OPT } P']\!]_D \\
&= [\![Q' \text{ OPT } (Q_1 \text{ AND } Q_2^{V_2} \text{ AND } Q_3^{V_3} \text{ AND } \ldots \text{ AND } Q_n^{V_n})]\!]_D \\
&= [\![Q' \text{ OPT } (Q_1 \text{ AND } Q_2 \text{ AND } Q_3 \text{ AND } \ldots \text{ AND } Q_n)]\!]_D
\end{aligned}$$

The final step in our transformation corresponds exactly to the left side expression of the original claim (2). Thus, we have shown that the equivalence holds. □

Proof of Theorem 4

Having established Lemma 13 we are ready to prove PSPACE-completeness for fragment $\mathcal{O}$. As before in the proof of Theorem 3, it suffices to show hardness. Following the idea discussed before, we show that each AND expression in the proof of Theorem 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

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

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 simply that, for all subexpressions P_1 OPT P_2 in the prior encoding of P_{C_i}, it holds that $vars(P_1) \cap vars(P_2) = \emptyset$ and $[\![P_2]\!]_D \neq \emptyset$. To be more precise, each AND expression in the encoding P_{C_i} of a clause C_i is of the form $(a, v_j, ?var_i)$ AND $(a, \mathit{false}/\mathit{true}, ?V_j)$, so the right side triple pattern generates one result mapping $\{?V_j \mapsto 0/1\}$, 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 an $\mathcal{O}$-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), defined as $P'_\psi := P_{C_1}^{\text{OPT}} \text{ AND } \ldots \text{ AND } P_{C_n}^{\text{OPT}}$; 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

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

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

$$\begin{aligned}
P'_\varphi :=& (a, true, ?B_0) \text{ OPT } (P_1^{\text{OPT}} \text{ OPT } (Q_1^{\text{OPT}} \\
& \quad \ldots \\
& \quad \text{OPT } (P_{m-1}^{\text{OPT}} \text{ OPT } (Q_{m-1}^{\text{OPT}} \\
& \quad \text{OPT } P_*)) \ldots)), \text{ where} \\
P_* :=& P_m^{\text{OPT}} \text{ OPT } (Q_m^{\text{OPT}} \text{ AND } P'_\psi) \\
=& P_m^{\text{OPT}} \text{ OPT } (Q_m^{\text{OPT}} \text{ AND } P_{C_1}^{\text{OPT}} \text{ AND } \ldots \text{ AND } P_{C_n}^{\text{OPT}}).
\end{aligned}$$

Expression P_* is the only subexpression of P_φ that still contains AND operators (where $Q_m^{\text{OPT}}, P_{C_1}^{\text{OPT}}, \ldots, P_{C_n}^{\text{OPT}}$ are OPT-only expressions). We now exploit the rewriting from Lemma 13 and replace P_* by the $\mathcal{O}$ expression P_*^{OPT} defined as

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

where

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

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 13(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 13(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. we can show that $\{?B_0 \mapsto 1\} \in [\![P_\varphi^{\text{OPT}}]\!]_D$ iff φ is valid. □

B.5 Proof of Theorem 5

We start with a more general form of the QBF problem, which will be required later in the main proof of Theorem 5. The new version of QBF differs from the QBF versions used in the proofs of Theorems 3 and 4 (cf. Section 3.2) 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 14. There is a polynomial-time reduction from QBF* to the SPARQL EVALUATION problem for class $\mathcal{AFO}$.[8] □

Proof of Lemma 14

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, $\mathcal{AFO}$ 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.

Let us start with a quantified boolean formula of the form

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

where ψ is a quantifier-free boolean formula. We assume w.l.o.g. assume that formula ψ is constructed using 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

$$P_\psi := ((a, tv, ?V_1) \text{ AND } \ldots \text{ AND } (a, tv, ?V_n)) \text{ FILTER } f(\psi),$$

where $?V_1, \ldots, ?V_n$ represent the boolean variables $v_1, \ldots, v_n$ and function $f(\psi)$ generates a SPARQL condition that precisely mirrors the boolean formula ψ. Formally, function $f(\psi)$ is defined recursively on the structure of ψ as follows.

$$\begin{aligned} f(v_i) &:= ?V_i = 1 \\ f(\psi_1 \wedge \psi_2) &:= f(\psi_1) \wedge f(\psi_2) \\ f(\psi_1 \vee \psi_2) &:= f(\psi_1) \vee f(\psi_2) \\ f(\neg\psi_1) &:= \neg f(\psi_1) \end{aligned}$$

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. 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 essentially from the previous one:

LEMMA 15. 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 $enc(\varphi)$ such that

1. $enc(\varphi) \in \mathcal{E}_{\leq 2m}$,
2. φ is valid iff $\{?B_0 \mapsto 1\} \in [\![enc(\varphi)]\!]_D$, and
3. φ is invalid iff for each $\mu \in [\![enc(\varphi)]\!]_D$ it holds that $\mu \supseteq \{?B_0 \mapsto 1, ?A_1 \mapsto 1\}$. □

We omit the technical proof and state a last result before turning towards the proof of Lemma 5:

LEMMA 16. 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 P_1 UNION P_2 is in Σ_i^P.
2. The EVALUATION problem for P_1 AND P_2 is in Σ_i^P.
3. The EVALUATION problem for P_1 FILTER R is in Σ_i^P. □

Proof of Lemma 16

Lemma 16(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 can both checks in sequence.

Lemma 16(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. It is easy to see that the whole procedure is in Σ_i^P.

Lemma 16(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 procedure is in Σ_i^P. □

Proof of Theorem 5

We are now ready to tackle Theorem 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:

[8] The same result was proven in [26]. This lemma, however, was developed independently from [26]. We informally published it already one year earlier, in a same-named technical report.

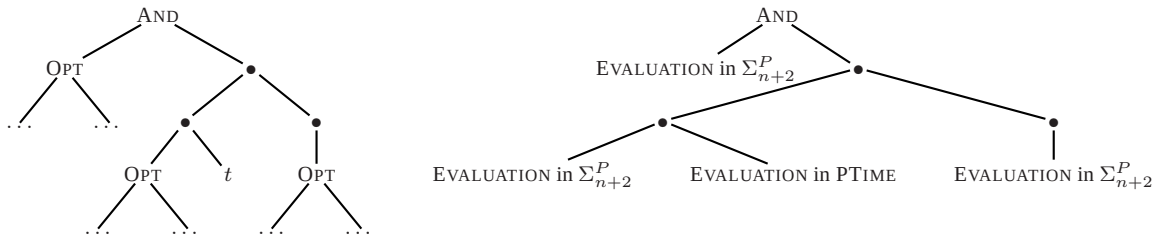

Figure 3: (a) AND-expression with Increased OPT-rank; (b) Associated Complexity Classes for OPT-expressions and Triple Patterns

> 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^P_n-complete for $n \geq 1$.

(Hardness) Recall that our goal is to show that fragment $\mathcal{E}_{\leq n}$ is $\Sigma^P_{\leq n+1}$-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

$$\begin{aligned}\varphi_1 :=& \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m (\psi \wedge y_0), \text{ and}\\ \varphi_2 :=& \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m (\psi \wedge \neg y_0).\end{aligned}$$

According to Lemma 15 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. $[\![enc(\varphi_1)]\!]_D$ ($[\![enc(\varphi_2)]\!]_D$) contains the mapping $\mu := \{?B_0 \mapsto 1\}$ iff φ_1 (φ_2) is valid. It is easy to see that the expression $enc(\varphi) := enc(\varphi_1)$ UNION $enc(\varphi_2)$ then contains μ if and only if φ_1 or φ_2 is valid, i.e. iff $\varphi := \varphi_1 \vee \varphi_2$ is valid. Given that both $enc(\varphi_1)$ 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

$$\begin{aligned}\varphi_1 :=& \forall y_0 \exists x_1 \forall y_1 \ldots \exists x_m \forall y_m (\psi \wedge y_0), \text{ and}\\ \varphi_2 :=& \forall y_0 \exists x_1 \forall y_1 \ldots \exists x_m \forall y_m (\psi \wedge \neg y_0).\end{aligned}$$

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 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 iff 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 iff the original formula $\varphi := \varphi_1 \vee \varphi_2$ is valid. We first rewrite φ':

$$\begin{aligned}\varphi' :=& \forall y_0 \exists x_1 \forall y_1 \ldots \exists x_m \forall y_m \psi'\\ =& \neg\exists y_0 \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m \neg\psi'\\ =& \neg(\varphi'_1 \vee \varphi'_2), \text{ where}\\ \varphi'_1 :=& \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m (\neg\psi' \wedge y_0), \text{ and}\\ \varphi'_2 :=& \forall x_1 \exists y_1 \ldots \forall x_m \exists y_m (\neg\psi' \wedge \neg y_0).\end{aligned}$$

According to Lemma 15, 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 $Q := ((a, false, ?A_1)$ 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 $\varphi' = \neg(\varphi'_1 \vee \varphi'_2)$ holds, 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.

(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 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 16) 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 $[\![P_1$ OPT $P_2]\!]_D = [\![P_1$ AND $P_2]\!]_D \cup ([\![P_1]\!]_D \setminus [\![P_2]\!]_D)$, so $\mu \in [\![P_1$ OPT $P_2]\!]_D$ iff it is generated by (i) $[\![P_1$ AND $P_2]\!]_D$ or (i) $[\![P_1]\!]_D \setminus [\![P_2]\!]_D$. According to Lemma 16(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 $C_2 := \neg\exists \mu' \in [\![P_2]\!]_D : \mu$ *and* μ' *are compatible*. By induction hypothesis, C_1 can be checked in Σ^P_{n+1}. We now argue that $\neg C_2 = \exists \mu' \in [\![P_2]\!]_D : \mu$ *and* μ' *are compatible* can be evaluated 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}. Therefore, the two checks in sequence can be executed in Σ^P_{n+2}, i.e. the whole algorithm is in Σ^P_{n+2}. This completes case (1) of the induction.

(2) Assume that $Q := P_1$ AND P_2. Figure 3(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 3(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 16(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.

Finally, it is easily verified that the two remaining cases, namely (3) $Q := P_1$ UNION P_2 and (4) $Q := P_1$ FILTER R follow by analogical arguments as used in case (2). □

B.6 Proof of Theorem 6

Before proving the theorem, we show the following:

LEMMA 17. 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 17

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 show 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). This completes the proof. □

Proof of Theorem 6

Theorem 6(1): Follows from Lemma 17 and Corollary 2.

Theorem 6(2): Follows from Lemma 17 and Theorem 5.

Theorem 6(3): NP-completeness for fragment $\mathcal{AU}^\pi$ follows directly from Lemma 17 and Theorem 2 and NP-completeness for $\mathcal{AFU}^\pi$ follows from Lemma 17 and Theorem 1. Therefore, it remains to show that fragments $\mathcal{A}^\pi$ and $\mathcal{AF}^\pi$ are NP-complete.

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 proof of Lemma 17 and eliminate the SELECT-clause. More precisely, we guess a mapping $\mu' \supseteq \mu$ s.t. $\pi_S(\{\mu'\}) = \mu$ and check if $\mu' \in [\![Q']\!]_D$ (cf. the proof of Lemma 17). 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 hardness for both classes 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 [10]. 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 $0, 1 \in U$ 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 $\mu := \{?A \mapsto 1\} \in [\![P]\!]_D$ if and only if formula ψ is satisfiable.

Theorem 6(4): We prove membership in PTIME for fragment $\mathcal{FU}^\pi$, which 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 (2) asserts that the top-level projection generates mapping μ from μ'. It is easy 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, 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. □

C. PROOFS OF ALGEBRAIC RESULTS

C.1 Proof of Lemma 2

We prove the lemma by induction on the structure of $\widetilde{\mathbb{A}}$ expressions, thereby exploiting the structural constraints imposed by Definition 8. The basic case is $\widetilde{A} := [\![t]\!]_D$. By semantics (see Definition 4), 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 8, 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 assume w.l.o.g. 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 $\widetilde{A} := \widetilde{A_1} \rtimes \widetilde{A_2}$, where both $\widetilde{A_1}$ and $\widetilde{A_2}$ are $\widetilde{\mathbb{A}}$ expressions. We rewrite the left outer join according to its semantics: $\widetilde{A} = \widetilde{A_1} \rtimes \widetilde{A_2} = (\widetilde{A_1} \bowtie \widetilde{A_2}) \cup (\widetilde{A_1} \setminus \widetilde{A_2})$. 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 8 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 2, 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 1 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 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 1 and 2 it follows that each two mappings generated by $\widetilde{A_1} \cup \widetilde{A_2}$ bind exactly the same set of variables. Following the argumentation in case (5b), two distinct mapping then disagree in at least one variable and thus are incompatible. □

C.2 Proofs Equivalences Figure 2

Rules in Group I

(UIdem). Follows trivially from the Definition of operator $\cup$.

($\widetilde{JIdem}$). 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 2 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.

($\widetilde{LIdem}$). Let $\widetilde{A} \in \widetilde{\mathbb{A}}$. We rewrite the left side expression schematically:

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

(UIdem). Follows trivially from the Definition of operator $\setminus$. □

Rules in Group II

Equivalences *(UAss)*, *(JAss)*, *(UComm)*, *(JComm)*, *(JUDistL)*, and *(LUDistR)* have been shown in [24] in Proposition 1 (the results there are stated at syntactic level and easily carry over to SPARQL algebra). *(JUDistR)* follows from *(JComm)* and *(JUDistL)*.

(MUDistR). We show that both directions of the equation hold. $\Rightarrow$: Consider a mapping $\mu \in (A_1 \cup A_2) \setminus A_3$. Hence, μ is contained in A_1 or in A_2 and there is no compatible mapping in A_3. If $\mu \in A_1$ then the right side subexpression $A_1 \setminus A_3$ generates μ, in the other case $A_2 \setminus A_3$ generates does. $\Leftarrow$: Consider a mapping μ in $(A_1 \setminus A_3) \cup (A_2 \setminus A_3)$. Then $\mu \in (A_1 \setminus A_3)$ or $\mu \in (A_2 \cup A_3)$. In the first case, μ is contained in A_1 and there is no compatible mapping in A_3. Clearly, μ is then also contained in $A_1 \cup A_2$ and $(A_1 \cup A_2) \setminus A_3$. The second case is symmetrical. □

Rules in Group III

We introduce some notation. Given a mapping μ and variable set $S \subseteq V$, we define $\mu_{|S}$ as the mapping obtained by projecting for the variables S in μ, e.g. $\{?x \mapsto 1, ?y \mapsto 2\}_{|\{?x\}} = \{?x \mapsto 1\}$. 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 and the observation that $pVars(A)$ extracts all variables that are potentially bound in any result mapping, as stated in Proposition 2.

(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 2, 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.

(PFPush). Follows from the semantics of operator π and operator σ in Definition 4. 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 preprojection does not change the semantics.

(PMerge). Follows trivially from the definition of operator π.

(PUPush). Follows easily from the definition of the projection and the union operator. We omit the details.

(PJPush). $\Rightarrow$: 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 $\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)$ such that for all $?x \in S$ either $\mu(?x) = \mu'(?x)$ 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 for variables in S, and this mapping is also generated by the right side expression when projecting for S in μ'. So let us consider a mapping $\mu \in A_1 \bowtie A_2$. By semantics of $\bowtie$, 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 generate a mapping $\mu'_1 \subseteq \mu_1$, obtained from $\mu_1 \in A_1$ by projecting on the variables S'; similarly, $\pi_{S'}(A_2)$ generates a mapping $\mu'_2 \subseteq \mu_2$, obtained from μ_2 by projecting on the variables 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 $\mu_1 \sim \mu_2 \rightarrow \mu'_1 \sim \mu'_2$, 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 $\mu(?x) = \mu'(?x)$ 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 on S', and similarly assume that μ'_2 is obtained from $\mu_2 \in A_2$ by projecting on 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 2 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.

(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 on the variables 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 also $\pi_{S'}(A_1) \setminus \pi_{S''}(A_2)$ generates μ'_1. Going one step further, we observe that the whole expression at the right side (i.e., including the outermost projection for S) generates the mapping $\mu'_{1|S}$. We know that μ'_1 agrees with μ_1 on all variables in S, so $\mu'_{1|S} = \mu_{1|S} = \mu$. Hence, 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 is unbound in both. This is a contradiction to the initial claim that μ_1 is incompatible with each mapping in A_2, so 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 $\mu'_1 \in \pi_{S'}(A_1)$ that is incompatible with each mapping in $\pi_{S''}(A_2)$ by projecting on the variables S, i.e. $\mu' = \mu'_{1|S}$. 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 we conclude that the left side expression generates mapping μ'. This completes the proof.

(PLPush). The following rewriting proves the claim, where we use the shortcuts $S' := S \cup (pVars(A_1) \cap pVars(A_2))$ and $S'' := pVars(A_1) \cap pVars(A_2)$.

$$
\begin{array}{ll}
\pi_S(A_1 \rtimes A_2) & \\
= \pi_S((A_1 \bowtie A_2) \cup (A_1 \setminus A_2)) & \text{[semantics]} \\
= \pi_S(A_1 \bowtie A_2) \cup \pi_S(A_1 \setminus A_2) & \text{[\textit{(PUPush)}]} \\
= \pi_S(\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)) \cup & \\
\quad \pi_S(\pi_{S'}(A_1) \setminus \pi_{S''}(A_2)) & \text{[\textit{(PJPush)},\textit{(PMPush)}]} \\
= \pi_S(\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)) \cup & \\
\quad \pi_S(\pi_{S'}(A_1) \setminus \pi_{S'}(A_2)) & [*] \\
= \pi_S((\pi_{S'}(A_1) \bowtie \pi_{S'}(A_2)) \cup & \\
\quad (\pi_{S'}(A_1) \setminus \pi_{S'}(A_2))) & \text{[\textit{(PUPush)}]} \\
= \pi_S(\pi_{S'}(A_1) \rtimes \pi_{S'}(A_2)) & \text{[semantics]}
\end{array}
$$

Most interesting is step $*$, where we replace S' by S''. 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 the latter rule 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 contained in S but not in $pVars(A_1) \cap pVars(A_2)$. These variables are harmless because they cannot induce incompatibility between the A_1 and the A_2-part on either side of the equivalence, as they occur at most in one of both mapping sets. We omit the technical details. □

Group IV

(FDecompI). Follows from Lemma 1(1) in [24].

(FDecompII). Follows from Lemma 1(2) in [24].

(FReord). Follows from *(FDecompI)* and the commutativity of $\wedge$.

(FBndI). Follows from Proposition 1.

(FBndII). Follows from Proposition 2.

(FBndIII). Follows from Proposition 1.

(FBndIV). Follows from Proposition 2. □

Group V

(FUPush). Follows from Proposition 1(5) in [24].

(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). $\Rightarrow$: Let $\mu \in \sigma_R(A_1 \bowtie A_2)$. By semantics, $\mu \models R$ and we know that μ is of the form $\mu = \mu_1 \cup \mu_2$, where $\mu_1 \in A_1$, $\mu_2 \in A_2$, 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 10. 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 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 1 that $?x \in dom(\mu_1)$, which is a contradiction; otherwise, if $?x$ satisfies constraint (ii) we know from Proposition 2 that $?x \notin dom(\mu_2)$. Given that $?x \in dom(\mu)$, this implies that $?x$ must be contained in $dom(\mu_1)$, which is again a contradiction. We conclude that $\mu_1 \models R$, hence *(Claim1)* holds. $\Leftarrow$: Let $\mu \in \sigma_R(A_1) \bowtie A_2$, so μ is of the form $\mu = \mu_1 \cup \mu_2$, where $\mu_1 \in A_1$, $\mu_2 \in A_2$, and $\mu_1 \models R$. 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 *(Claim2)* $\mu \models R$ holds, because this implies that the left side generates μ. Let us, for the sake of contradiction, assume that $\mu \not\models R$. Consider the semantics of filter expressions in Definition 10 and recall that $\mu \supseteq \mu_1$. Given that $\mu_1 \models R$, we can easily derive that μ 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 the filter to evaluate to false. The rest is analogous to the proof of direction $\Rightarrow$.

(FLPush). We rewrite the expression:

$$
\begin{array}{ll}
\sigma_R(A_1 \rtimes A_2) & \\
= \sigma_R((A_1 \bowtie A_2) \cup (A_1 \setminus A_2)) & \text{[semantics]} \\
= \sigma_R(A_1 \bowtie A_2) \cup \sigma_R(A_1 \setminus A_2) & \text{[(FUPush)]} \\
= (\sigma_R(A_1) \bowtie A_2) \cup (\sigma_R(A_1) \setminus A_2) & \text{[(FJPush),(FMPush)]} \\
= \sigma_R(A_1) \rtimes A_2 & \text{[semantics]}
\end{array}
$$

The rewriting proves the equivalence. □

Group VI

(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, as there is a compatible mapping to μ in A_2, the expression $\mu \in (A_1 \setminus A_3) \setminus A_2$ 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 exactly if 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 $A_1 \setminus A_2$ contains μ (as there is no compatible mapping in A_2 and $\mu \in A_1$). From the fact that there is no compatible mapping in A_3, we deduce $\mu \in (A_1 \setminus A_2) \setminus A_3$.

(MJ). See Lemma 3(2) in [24].

($\widetilde{LJ}$). Let $\widetilde{A_1}$, $\widetilde{A_2}$ be $\widetilde{\mathbb{A}}$-expressions. The following sequence of rewriting steps proves the equivalence.

$$
\begin{array}{ll}
\widetilde{A_1} \rtimes \widetilde{A_2} & \\
= (\widetilde{A_1} \bowtie \widetilde{A_2}) \cup (\widetilde{A_1} \setminus \widetilde{A_2}) & \text{[by semantics]} \\
= (\widetilde{A_1} \bowtie (\widetilde{A_1} \bowtie \widetilde{A_2})) \cup (\widetilde{A_1} \setminus (\widetilde{A_1} \bowtie \widetilde{A_2})) & \text{[}\widetilde{(JIdem)}\text{,(JAss),(MJ)]} \\
= (\widetilde{A_1} \rtimes (\widetilde{A_1} \bowtie \widetilde{A_2})) & \text{[by semantics]}
\end{array}
$$

(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:

$$
\begin{array}{ll}
\sigma_{\neg bnd(?x)}(A_1 \rtimes A_2) & \\
= \sigma_{\neg bnd(?x)}((A_1 \bowtie A_2) \cup (A_1 \setminus A_2)) & \text{[semantics]} \\
= \sigma_{\neg bnd(?x)}(A_1 \bowtie A_2) \cup \sigma_{\neg bnd(?x)}(A_1 \setminus A_2) & \text{[(FUPush)]} \\
= \emptyset \cup \sigma_{\neg bnd(?x)}(A_1 \setminus A_2) & \text{[(FBndIII)]} \\
= \sigma_{\neg bnd(?x)}(A_1 \setminus A_2) & \text{[semantics]} \\
= A_1 \setminus A_2 & \text{[(FBndIV)]}
\end{array}
$$

(FLBndII). 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.

$$
\begin{array}{ll}
\sigma_{bnd(?x)}(A_1 \rtimes A_2) & \\
= \sigma_{bnd(?x)}((A_1 \bowtie A_2) \cup (A_1 \setminus A_2)) & \text{[semantics]} \\
= \sigma_{bnd(?x)}(A_1 \bowtie A_2) \cup \sigma_{bnd(?x)}(A_1 \setminus A_2) & \text{[(FUPush)]} \\
= \sigma_{bnd(?x)}(A_1 \bowtie A_2) \cup \emptyset & \text{[(FBndII)]} \\
= \sigma_{bnd(?x)}(A_1 \bowtie A_2) & \text{[semantics]} \\
= A_1 \bowtie A_2 & \text{[(FBndI)]}\ \square
\end{array}
$$

C.3 Proof of Lemma 3

Proof of Lemma 3(1): Trivial (by counterexample).

Proof of Lemma 3(2): We provide counterexamples that rule out distributivity of operators $\rtimes$ and $\setminus$ over $\cup$, all of which are designed for the fixed database $D := \{(0, c, 1)\}$:

- 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 expressions $A_1 := [\![(0, c, ?a)]\!]_D$, $A_2 := [\![(?a, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?b)]\!]_D$.
- Equivalence $A_1 \rtimes (A_2 \cup A_3) \equiv (A_1 \rtimes A_2) \cup (A_1 \rtimes A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(0, c, ?a)]\!]_D$, $A_2 := [\![(?a, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?b)]\!]_D$.

Proof of Lemma 3(3): We provide counterexamples for all operator constellations that are listed in the lemma. As before, the counterexamples are designed for the database $D := \{(0, c, 1)\}$. We start with invalid distributivity rules over operator $\bowtie$:

- 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 expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(?b, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?b)]\!]_D$.
- Equivalence $(A_1 \bowtie A_2) \cup A_3 \equiv (A_1 \cup A_3) \bowtie (A_2 \cup A_3)$ does not hold (symmetrical to the previous one).
- 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 expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(?b, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?b)]\!]_D$.
- 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 expressions $A_1 := [\![(0, c, ?a)]\!]_D$, $A_2 := [\![(0, c, ?b)]\!]_D$, and $A_3 := [\![(?a, c, 1)]\!]_D$.
- Equivalence $A_1 \bowtie\!\!\!\!\!\!\bowtie (A_2 \bowtie A_3) \equiv (A_1 \bowtie\!\!\!\!\!\!\bowtie A_2) \bowtie (A_1 \bowtie\!\!\!\!\!\!\bowtie A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(?b, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?a)]\!]_D$.
- Equivalence $(A_1 \bowtie A_2) \bowtie\!\!\!\!\!\!\bowtie A_3 \equiv (A_1 \bowtie\!\!\!\!\!\!\bowtie A_3) \bowtie (A_2 \bowtie\!\!\!\!\!\!\bowtie A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(0, c, ?a)]\!]_D$, $A_2 := [\![(0, c, ?b)]\!]_D$, and $A_3 := [\![(?a, c, 1)]\!]_D$.

Next, we provide counterexamples for distributivity rules over $\setminus$:

- 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 expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(0, c, ?a)]\!]_D$, and $A_3 := [\![(?a, c, 1)]\!]_D$.
- Equivalence $(A_1 \setminus A_2) \cup A_3 \equiv (A_1 \cup A_3) \setminus (A_2 \cup A_3)$ does not hold (symmetrical to the previous one).
- 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 expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(?b, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?a)]\!]_D$.
- Equivalence $(A_1 \setminus A_2) \bowtie A_3 \equiv (A_1 \bowtie A_3) \setminus (A_2 \bowtie A_3)$ does not hold (symmetrical to the previous one).
- Equivalence $A_1 \bowtie\!\!\!\!\!\!\bowtie (A_2 \setminus A_3) \equiv (A_1 \bowtie\!\!\!\!\!\!\bowtie A_2) \setminus (A_1 \bowtie\!\!\!\!\!\!\bowtie A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(?b, c, 1)]\!]_D$, and $A_3 := [\![(?b, c, 1)]\!]_D$.
- Equivalence $(A_1 \setminus A_2) \bowtie\!\!\!\!\!\!\bowtie A_3 \equiv (A_1 \bowtie\!\!\!\!\!\!\bowtie A_3) \setminus (A_2 \bowtie\!\!\!\!\!\!\bowtie A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(?b, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?b)]\!]_D$.

Finally, we provide counterexamples for invalid rules over $\bowtie\!\!\!\!\!\!\bowtie$:

- Equivalence $A_1 \cup (A_2 \bowtie\!\!\!\!\!\!\bowtie A_3) \equiv (A_1 \cup A_2) \bowtie\!\!\!\!\!\!\bowtie (A_1 \cup A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(c, c, c)]\!]_D$, and $A_3 := [\![(?b, c, 1)]\!]_D$.
- Equivalence $(A_1 \bowtie\!\!\!\!\!\!\bowtie A_2) \cup A_3 \equiv (A_1 \cup A_3) \bowtie\!\!\!\!\!\!\bowtie (A_2 \cup A_3)$ does not hold (symmetrical to the previous one).
- Equivalence $A_1 \bowtie (A_2 \bowtie\!\!\!\!\!\!\bowtie A_3) \equiv (A_1 \bowtie A_2) \bowtie\!\!\!\!\!\!\bowtie (A_1 \bowtie A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(?b, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?a)]\!]_D$.
- Equivalence $(A_1 \bowtie\!\!\!\!\!\!\bowtie A_2) \bowtie A_3 \equiv (A_1 \bowtie A_3) \bowtie\!\!\!\!\!\!\bowtie (A_2 \bowtie A_3)$ does not hold (symmetrical to the previous one).
- Equivalence $A_1 \setminus (A_2 \bowtie\!\!\!\!\!\!\bowtie A_3) \equiv (A_1 \setminus A_2) \bowtie\!\!\!\!\!\!\bowtie (A_1 \setminus A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 = [\![(?b, c, 1)]\!]_D$, and $A_3 [\![(0, c, ?a)]\!]_D$.
- Equivalence $(A_1 \bowtie\!\!\!\!\!\!\bowtie A_2) \setminus A_3 \equiv (A_1 \setminus A_3) \bowtie\!\!\!\!\!\!\bowtie (A_2 \setminus A_3)$ does not hold, as witnessed by expressions $A_1 := [\![(?a, c, 1)]\!]_D$, $A_2 := [\![(?b, c, 1)]\!]_D$, and $A_3 := [\![(0, c, ?b)]\!]_D$.

The list of counterexamples is exhaustive. □

C.4 Proof of Lemma 4

(FElimI). We first introduce three functions $\mathit{rem}_{?x} : \mathcal{M} \mapsto \mathcal{M}$, $\mathit{add}_{?x \mapsto c} : \mathcal{M} \mapsto \mathcal{M}$, and $\mathit{subst}_{\frac{?y}{?x}} : \mathcal{M} \mapsto \mathcal{M}$, which manipulate mappings as described in the following listing:

- $\mathit{rem}_{?x}(\mu)$ removes $?x$ from μ (if it is bound), i.e. outputs mapping μ' such that $dom(\mu') := dom(\mu) \setminus \{?x\}$ and $\mu'(?a) := \mu(?a)$ for all $?a \in dom(\mu')$.
- $\mathit{add}_{?x \mapsto c}(\mu)$ 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).
- $\mathit{subst}_{\frac{?y}{?x}}(\mu) := \mathit{rem}_{?x}(\mathit{add}_{?y \mapsto \mu(?x)}(\mu))$ 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 $rem_{?x}(\mu) \in A^{\frac{?y}{?x}}$.

(C2) If $\mu \in A$ and $?x \notin dom(\mu)$ then $\mu \in A^{\frac{?y}{?x}}$.

(C3) If $\mu \in A$ and $?x \in dom(\mu)$, and $?y \notin dom(\mu)$ then $\mathit{subst}_{\frac{?y}{?x}}(\mu) \in A^{\frac{?y}{?x}}$.

(C4) If $\mu \in A^{\frac{?y}{?x}}$ and $?y \notin dom(\mu)$ then $\mu \in A$.

(C5) If $\mu \in A^{\frac{?y}{?x}}$ and $?y \in dom(\mu)$ then $\mu \in A$ or $\mathit{add}_{?x \mapsto \mu(?y)}(\mu) \in A$ or $\mathit{subst}_{\frac{?x}{?y}}(\mu) \in A$.

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 imply *(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 1). Condition (C5) now implies that (i) $\mu' \in A$, or (ii) $\mathit{add}_{?x \mapsto \mu'(?y)}(\mu') \in A$, or (iii) $\mathit{subst}_{\frac{?x}{?y}}(\mu') \in A$ 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 observe 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 $?y \notin dom(\mathit{subst}_{\frac{?x}{?y}}(\mu'))$. Therefore, given that condition (C5) is valid by assumption, we conclude that case (ii) $\mu'' := \mathit{add}_{?x \mapsto \mu'(?y)}(\mu') \in A$ 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 filter $\sigma_{?x=?y}$ in the left expression and from $\pi_{S \setminus \{?x\}}(\{\mu''\}) = \pi_{S \setminus \{?x\}}(\{\mu'\}) = \{\mu\}$ we deduce that the expression $\pi_{S \setminus \{?x\}}(\sigma_{?x=?y}(A))$ generates μ.

Having shown that the five claims imply the equivalence, we now prove them by structural inductions (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)-(C5) 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 generated 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_1. The proofs for the remaining conditions (C2)-(C5) proceed analogously.

(FElimII). Similar in idea to *(FElimI)*. □

C.5 Proof of Theorem 7

We denote the corresponding equivalences for bag algebra with superscript $+$, e.g. write *(Inv$^+$)* for rule *(Inv)* under bag semantics. Before presenting the proofs, we introduce some additional preliminaries. First we define a function that allows us to map expressions from one algebra into same-structured expressions of the other algebra.

DEFINITION 20 (FUNCTION *s2b*). Let $A_1, A_2 \in \mathbb{A}$ be set algebra expressions, $S \subset V$ a set of variables, and R a filter condition. We define the bijective function $s2b : \mathbb{A} \mapsto \mathbb{A}^+$ recursively on the structure of $\mathbb{A}$-expression:

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

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 4 and 12, 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 $[\![Q]\!]_D = s2b^{-1}([\![Q]\!]_D^+)$ holds (when interpreting the results of function $[\![.]\!]_D$ and $[\![.]\!]_D^+$ as SPARQL algebra expressions rather than sets of mappings). Given this connection, we can easily transfer Lemma 1, which relates the two semantics, into the context of SPARQL set and bag algebra:

LEMMA 18. The following claims hold.

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

Further, we will use some standard rewriting rules for sums.

PROPOSITION 3 (SUM REWRITING RULES). Let a_x, b_x, denote expressions that depend on some x, λ be an expression that does not depend on x, and 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) $\sum_{x \in X} a_x + b_x = \sum_{x \in X} a_x + \sum_{x \in X} b_x$. □

We refer to these equivalences as *(S1)*, *(S2)*, and *(S3)*.

Lemma 18 shows that the result of evaluating set and bag algebra expressions differs at most in the associated cardinality, so (given that the rules we are going to prove hold for SPARQL set algebra) it always 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.

Group I

It has been shown in Example 10 that *(UIdem)* does not carry over from set to bag semantics. 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 $\widetilde{A^+} \in \widetilde{\mathbb{A}^+}$. It is easily verified that the set and bag semantics always coincide for $\widetilde{A^+}$ expressions and that the equivalence holds under set semantics. Clearly, it holds that $\widetilde{A^+} \bowtie \widetilde{A^+} \in \widetilde{\mathbb{A}^+}$, so the SPARQL bag algebra equivalence $(\widetilde{JIdem^+})$ holds. The argumentation for $(\widetilde{LIdem^+})$ is the same. Finally, equivalence *(Inv$^+$)* follows easily from Lemma 18 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). □

Group II

(UAss$^+$). Let $A_1^+, A_2^+, A_3^+ \in \mathbb{A}^+$. Put $A_l^+ := (A_1^+ \cup A_2^+) \cup A_3^+$, $A_r^+ := A_1^+ \cup (A_2^+ \cup A_3^+)$. 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 (cf. Definition 11) 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 rewrite the left side multiplicity $m_l(\mu)$:

$$
\begin{aligned}
m_l(\mu) = & \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 (m_{1\bowtie 2}(\mu_{1\bowtie 2}) * m_3(\mu_3)) \\
= & \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 \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}\}} \\
& \quad (m_1(\mu_1) * m_2(\mu_2)) * m_3(\mu_3)) \\
\overset{(S1)}{=} & \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 \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}\}} \\
& \quad (m_1(\mu_1) * m_2(\mu_2) * m_3(\mu_3))) \\
\overset{(S2)}{=} & \sum_{((\mu_{1\bowtie 2},\mu_3),(\mu_1,\mu_2))\in\{((\mu^*_{1\bowtie 2},\mu^*_3),(\mu^*_1,\mu^*_2))\in} \\
& \quad ((\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)) \\
= & \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\}} \\
& \quad (m_1(\mu_1) * m_2(\mu_2) * m_3(\mu_3)) \\
= & \sum_{((\mu_1,\mu_{2\bowtie 3}),(\mu_2,\mu_3))\in\{((\mu^*_1,\mu^*_{2\bowtie 3}),(\mu^*_2,\mu^*_3))\in} \\
& \quad ((\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)}{=} & \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 \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}\}} \\
& \quad (m_1(\mu_1) * m_2(\mu_2) * m_3(\mu_3))) \\
\overset{(S1)}{=} & \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) * \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}\}} \\
& \quad (m_2(\mu_2) * m_3(\mu_3))) \\
= & \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) * 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 rewrite $m_l(\mu)$ stepwise: $m_l(\mu) = m_1(\mu) + m_2(\mu) = m_2(\mu) + m_1(\mu) = m_r(\mu)$.

(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 $\bowtie$ we rewrite the left side multiplicity:

$$
\begin{aligned}
m_l(\mu) = & \sum_{(\mu_1,\mu_2)\in\{(\mu^*_1,\mu^*_2)\in(\Omega_1\times\Omega_2)\mid \mu^*_1\cup\mu^*_2=\mu\}} \\
& \quad (m_1(\mu_1) * m_2(\mu_2)) \\
= & \sum_{(\mu_2,\mu_1)\in\{(\mu^*_2,\mu^*_1)\in(\Omega_2\times\Omega_1)\mid \mu^*_2\cup\mu^*_1=\mu\}} \\
& \quad (m_2(\mu_2) * m_1(\mu_1)) \\
= & m_r(\mu)
\end{aligned}
$$

(JUDistR$^+$) and *(JUDistL$^+$)* follow by rewritings that are similar in style to those presented in previous proofs.

(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 $A^+_1, A^+_2, A^+_3 \in \mathbb{A}^+$. Then

$$
\begin{aligned}
& (A^+_1 \cup A^+_2) \rtimes A^+_3 \\
& = ((A^+_1 \cup A^+_2) \bowtie A^+_3) \cup ((A^+_1 \cup A^+_2) \setminus A^+_3) \\
& = ((A^+_1 \bowtie A^+_3) \cup (A^+_2 \bowtie A^+_3)) \cup \\
& \quad ((A^+_1 \setminus A^+_3) \cup (A^+_2 \setminus A^+_3)) && \text{[\textit{(JUDistR$^+$)}, \textit{(MUDistR$^+$)}]} \\
& = ((A^+_1 \bowtie A^+_3) \cup (A^+_1 \setminus A^+_3)) \cup \\
& \quad ((A^+_2 \bowtie A^+_3) \cup (A^+_2 \setminus A^+_3)) && \text{[\textit{(UAss$^+$)}, \textit{(UComm$^+$)}]} \\
& = (A^+_1 \rtimes A^+_3) \cup (A^+_2 \rtimes A^+_3) && \text{[semantics] } \square
\end{aligned}
$$

Group III

(PBaseI$^+$). Let $A^+ \in \mathbb{A}^+$ and $S \subset V$. Consider a mapping μ contained in the result of evaluating $A^+_l := \pi_{pVars(A^+)\cup S}(A^+)$ and $A^+_r := A^+$. We rewrite $m_l(\mu)$ stepwise:

$$
\begin{aligned}
m_l(\mu) = & \sum_{\mu_+\in\{\mu^*_+\in\Omega\mid \pi_{pVars(A^*)\cup S}(\{\mu^*_+\})=\{\mu\}\}} m(\mu_+) \\
\overset{(*)}{=} & \sum_{\mu_+\in\{\mu\}} m(\mu_+) = m(\mu) = m_r(\mu),
\end{aligned}
$$

where step $(*)$ follows from the observation that equivalence $\pi_{pVars(A^*)\cup S}(\{\mu^*_+\}) = \{\mu\}$ holds if and only if $\mu^*_+ = \mu$ holds (this claim follows easily from Proposition 2 and 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 11 and rewrite the (right side) multiplicity $m_r(\mu)$:

$$
\begin{aligned}
m_r(\mu) = & \sum_{\mu_+\in\{\mu^*_+\in\Omega\mid \pi_{S\cap pVars(A^+)}(\{\mu^*_+\})=\{\mu\}\}} m(\mu_+) \\
\overset{(*)}{=} & \sum_{\mu_+\in\{\mu^*_+\in\Omega\mid \pi_S(\{\mu^*_+\})=\{\mu\}\}} m(\mu_+) \\
= & \; m_l(\mu),
\end{aligned}
$$

where step $(*)$ follows from the semantics and Proposition 2.

(PFPush$^+$). Similar in idea to *(PMPush$^+$)*.

(PMerge$^+$). Let $A^+ \in \mathbb{A}^+$ and $S_1, S_2 \subset V$. We define $A^+_l := \pi_{S_1}(\pi_{S_2}(A^+))$, $A^+_r := \pi_{S_1\cap S_2}(A^+)$, and $A^+_{\pi 2} := \pi_{S_2}(A^+)$. According to Lemma 18, 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 multiplicity $m_l(\mu)$ schematically:

$$
\begin{aligned}
m_l(\mu) = & \sum_{\mu_+\in\{\mu^*_+\in\Omega_{\pi 2}\mid \pi_{S_1}(\{\mu^*_+\})=\{\mu\}\}} m_{\pi 2}(\mu_+) \\
= & \sum_{\mu_+\in\{\mu^*_+\in\Omega_{\pi 2}\mid \pi_{S_1}(\{\mu^*_+\})=\{\mu\}\}} \\
& \quad \sum_{\mu'_+\in\{\mu^\bullet_+\in\Omega\mid \pi_{S_2}(\{\mu^\bullet_+\})=\{\mu_+\}\}} m(\mu'_+) \\
\overset{(S2)}{=} & \sum_{(\mu_+,\mu'_+)\in\{(\mu^*_+,\mu^\bullet_+)\in(\Omega_{\pi 2},\Omega)\mid} \\
& \quad \pi_{S_1}(\{\mu^*_+\})=\{\mu\}\wedge\pi_{S_2}(\{\mu^\bullet_+\})=\{\mu^*_+\}\} m(\mu'_+) \\
= & \sum_{(\mu_+,\mu'_+)\in\{(\mu^*_+,\mu^\bullet_+)\in(\Omega_{\pi 2},\Omega)\mid} \\
& \quad \pi_{S_1}(\pi_{S_2}(\{\mu^\bullet_+\}))=\{\mu\}\wedge\pi_{S_2}(\{\mu^\bullet_+\})=\{\mu^*_+\}\} m(\mu'_+) \\
\overset{(*_1)}{=} & \sum_{\mu'_+\in\{\mu^\bullet_+\in\Omega\mid} \\
& \quad \pi_{S_1}(\pi_{S_2}(\{\mu^\bullet_+\}))=\{\mu\}\} m(\mu'_+) \\
\overset{(*_2)}{=} & \sum_{\mu'_+\in\{\mu^\bullet_+\in\Omega\mid} \\
& \quad \pi_{S_1\cap S_2}(\{\mu^\bullet_+\}))=\{\mu\}\} m(\mu'_+) \\
= & \; m_r(\mu),
\end{aligned}
$$

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.

(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 11 and rewrite the multiplicity $m_l(\mu)$ step-by-step:

$$
\begin{aligned}
m_l(\mu) = & \sum_{\mu_+\in\{\mu^*_+\in\Omega_{1\cup 2}\mid \pi_S(\{\mu^*_+\})=\{\mu\}\}} m_{1\cup 2}(\mu_+) \\
= & \sum_{\mu_+\in\{\mu^*_+\in\Omega_{1\cup 2}\mid \pi_S(\{\mu^*_+\})=\{\mu\}\}} (m_1(\mu_+) + m_2(\mu_+)) \\
\overset{(S3)}{=} & (\sum_{\mu_+\in\{\mu^*_+\in\Omega_{1\cup 2}\mid \pi_S(\{\mu^*_+\})=\{\mu\}\}} m_1(\mu_+)) + \\
& \quad (\sum_{\mu_+\in\{\mu^*_+\in\Omega_{1\cup 2}\mid \pi_S(\{\mu^*_+\})=\{\mu\}\}} m_2(\mu_+)) \\
\overset{(*)}{=} & \; (\sum_{\mu_+\in\{\mu^*_+\in\Omega_1\mid \pi_S(\{\mu^*_+\})=\{\mu\}\}} m_1(\mu_+)) + \\
& \quad (\sum_{\mu_+\in\{\mu^*_+\in\Omega_2\mid \pi_S(\{\mu^*_+\})=\{\mu\}\}} m_2(\mu_+)) \\
= & \; m_{\pi 1}(\mu) + m_{\pi 2}(\mu) \\
= & \; m_r(\mu),
\end{aligned}
$$

where step $(*)$ follows by semantics of operator $\cup$.

(PJPush$^+$). The rule follows from the following proposition.

PROPOSITION 4. 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.

$$\pi_{S'}(A_1^+ \bowtie A_2^+) \equiv \pi_{S'}(A_1^+) \bowtie \pi_{S'}(A_2^+) \qquad \textit{(FJPush2}^+\textit{)} \qquad \square$$

To see why Proposition 4 implies *(FJPush)*, consider the original equivalence, where $S' := S \cup (pVars(A_1^+) \cap pVars(A_2^+))$. Observe that, by construction, $S' \supseteq pVars(A_1^+) \cap pVars(A_2^+)$. We rewrite the left side of *(FJPush)* into the right side:

$$\begin{array}{ll} & \pi_S(A_1^+ \bowtie A_2^+) \\ = & \pi_S(\pi_{S'}(A_1^+ \bowtie A_2^+)) \qquad [\textit{(PMerge}^+\textit{)}] \\ = & \pi_S(\pi_{S'}(A_1^+) \bowtie \pi_{S'}(A_2^+)) \qquad [\textit{(FJPush2}^+\textit{)}] \end{array}$$

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 18). 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 on 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 2 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 that $\mu'_1 \sim \mu'_2$. This completes the second direction.

Having shown that the mapping sets coincide under bag semantics, it remains to show that the left- and right side multiplicities 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 $A_l^+ := \pi_{S'}(A_1^+ \bowtie A_2^+)$, $A_r^+ := \pi_{S'}(A_1^+) \bowtie \pi_{S'}(A_2^+)$, $A_{1\bowtie 2}^+ := A_1^+ \bowtie A_2^+$, $A_{\pi 1}^+ := \pi_{S'}(A_1^+)$, $A_{\pi 2}^+ := \pi_{S'}(A_2^+)$, and $A_{\pi 1 \bowtie \pi 2}^+ := A_{\pi 1}^+ \bowtie A_{\pi 2}^+$. Consider a mapping μ contained in the result of evaluating A_l^+ and A_r^+. Applying the semantics from Definition 11 we rewrite $m_l(\mu)$:

$$\begin{array}{ll} m_l(\mu) = & \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{1\bowtie 2} | \pi_{S'}(\{\mu_+^*\}) = \{\mu\}\}} m_{1\bowtie 2}(\mu_+) \\ = & \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{1\bowtie 2} | \pi_{S'}(\{\mu_+^*\}) = \{\mu\}\}} \\ & \quad \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in \Omega_1 \times \Omega_2 | \mu_1^* \cup \mu_2^* = \mu_+\}} (m_1(\mu_1) * m_2(\mu_2)) \\ \overset{(S2)}{=} & \sum_{(\mu_+,(\mu_1,\mu_2)) \in \{(\mu_+^*,(\mu_1^*,\mu_2^*)) \in \Omega_{1\bowtie 2} \times (\Omega_1 \times \Omega_2) | \pi_{S'}(\{\mu_+^*\}) = \{\mu\} \wedge \mu_1^* \cup \mu_2^* = \mu_+^*\}} (m_1(\mu_1) * m_2(\mu_2)) \\ = & \sum_{(\mu_+,(\mu_1,\mu_2)) \in \{(\mu_+^*,(\mu_1^*,\mu_2^*)) \in \Omega_{1\bowtie 2} \times (\Omega_1 \times \Omega_2) | \pi_{S'}(\{\mu_1^* \cup \mu_2^*\}) = \{\mu\} \wedge \mu_1^* \cup \mu_2^* = \mu_+^*\}} (m_1(\mu_1) * m_2(\mu_2)) \\ = & \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in (\Omega_1 \times \Omega_2) | \pi_{S'}(\{\mu_1^* \cup \mu_2^*\}) = \{\mu\}\}} (m_1(\mu_1) * m_2(\mu_2)) \\ \overset{(*)}{=} & \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in (\Omega_1 \times \Omega_2) | \pi_{S'}(\{\mu_1^*\}) \cup \pi_{S'}(\{\mu_2^*\}) = \{\mu\}\}} (m_1(\mu_1) * m_2(\mu_2)) \\ = & \sum_{(\mu_1,\mu_2) \in \{(\mu_1^*,\mu_2^*) \in (\Omega_{\pi 1} \times \Omega_{\pi 2}) | \{\mu_1^*\} \cup \{\mu_2^*\} = \{\mu\}\}} (m_1(\mu_1) * m_2(\mu_2)) \\ = & m_r(\mu), \end{array}$$

where $(*)$ follows from $S' \supseteq pVars(A_1) \cap pVars(A_2)$.

(*PMPush$^+$*). Let $A_1^+, A_2^+ \in \mathbb{A}^+$ and $S \subset V$ be a set of variables. Recall that by definition $S' := S \cup (pVars(A_1) \cap pVars(A_2))$ and $S'' := pVars(A_1) \cap pVars(A_2)$. Put $A_l^+ := \pi_S(A_1^+ \setminus A_2^+)$, $A_r^+ := \pi_S(\pi_{S'}(A_1^+) \setminus \pi_{S''}(A_2^+))$, $A_{1\setminus 2}^+ := A_1^+ \setminus A_2^+$, $A_{\pi 1}^+ := \pi_{S'}(A_1^+)$, $A_{\pi 2}^+ := \pi_{S''}(A_2^+)$, $A_{\pi 1 \setminus \pi 2}^+ := A_{\pi 1}^+ \setminus A_{\pi 2}^+$, and fix document D and a mapping μ that is contained both in Ω_l and Ω_r. Applying the semantics from Definition 11, we rewrite the (right side) multiplicity $m_r(\mu)$ schematically:

$$\begin{array}{ll} m_r(\mu) = & \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{\pi 1 \setminus \pi 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_{\pi 1 \setminus \pi 2}(\mu_+) \\ \overset{(*_1)}{=} & \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{\pi 1 \setminus \pi 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} m_{\pi 1}(\mu_+) \\ = & \sum_{\mu_+ \in \{\mu_+^* \in \Omega_{\pi 1 \setminus \pi 2} | \pi_S(\{\mu_+^*\}) = \{\mu\}\}} \\ & \quad \sum_{\mu'_+ \in \{\mu_+^\bullet \in \Omega_1 | \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+\}\}} m_1(\mu'_+) \\ \overset{(S2)}{=} & \sum_{(\mu_+,\mu'_+) \in \{(\mu_+^*,\mu_+^\bullet) \in \Omega_{\pi 1 \setminus \pi 2} \times \Omega_1 | \pi_S(\{\mu_+^*\}) = \{\mu\} \wedge \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m_1(\mu'_+) \\ = & \sum_{(\mu_+,\mu'_+) \in \{(\mu_+^*,\mu_+^\bullet) \in \Omega_{\pi 1 \setminus \pi 2} \times \Omega_1 | \pi_S(\pi_{S'}(\{\mu_+^\bullet\})) = \{\mu\} \wedge \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m_1(\mu'_+) \\ \overset{(*_2)}{=} & \sum_{(\mu_+,\mu'_+) \in \{(\mu_+^*,\mu_+^\bullet) \in \Omega_{\pi 1 \setminus \pi 2} \times \Omega_1 | \pi_S(\{\mu_+^\bullet\}) = \{\mu\} \wedge \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m_1(\mu'_+) \\ \overset{(*_3)}{=} & \sum_{(\mu_+,\mu'_+) \in \{(\mu_+^*,\mu_+^\bullet) \in \Omega_{\pi 1 \setminus \pi 2} \times \Omega_{1\setminus 2} | \pi_S(\{\mu_+^\bullet\}) = \{\mu\} \wedge \pi_{S'}(\{\mu_+^\bullet\}) = \{\mu_+^*\}\}} m_1(\mu'_+) \\ \overset{(*_4)}{=} & \sum_{\mu'_+ \in \{\mu_+^\bullet \in \Omega_{1\setminus 2} | \pi_S(\{\mu_+^\bullet\}) = \{\mu\}\}} m_1(\mu'_+) \\ \overset{(*_5)}{=} & \sum_{\mu'_+ \in \{\mu_+^\bullet \in \Omega_{1\setminus 2} | \pi_S(\{\mu_+^\bullet\}) = \{\mu\}\}} m_{1\setminus 2}(\mu'_+) \\ = & m_l(\mu), \end{array}$$

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$^+$). Similar to the proof of *(PLPush)* for SPARQL set algebra (observe that all rules that are used in the latter proof are also valid in the context of SPARQL bag algebra). $\square$

Group IV

As an example that shows that *(FDecompII)* does not carry over to bag algebra, consider the expression $A := [\![(c, c, ?x)]\!]_D$, filter $R := (\neg ?x = a) \vee (\neg ?x = b)$ and document $D := \{(c, c, c)\}$.

(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 11 we can easily derive that $m_l(\mu) = m(\mu) = m_{\sigma 2}(\mu) = m_r(\mu)$.

(FReord$^+$). Follows from equivalence *(FDecompI$^+$)* and the commutativity of the boolean operator $\wedge$.

(FBndI$^+$) - (FBndIV$^+$). Follow from the semantics of σ in. □

Group V

(FUPush$^+$). Follows from the semantics of σ and $\cup$.

(FMPush$^+$). Let $A_1^+, A_2^+ \in \mathbb{A}^+$ and R be a filter condition. Put $A_l^+ := \sigma_R(A_1^+ \setminus A_2^+)$, $A_r^+ := \sigma_R(A_1^+) \setminus A_2^+$, $A_{1\setminus 2}^+ := A_1^+ \setminus A_2^+$, 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 $\mu \models R$, $\mu \in \Omega_{1\setminus 2}$, $\mu \in \Omega_1$, and $\mu \in \Omega_{\sigma 1}$. Combining the semantics from Definition 3 with the above observations we obtain $m_l(\mu) = m_{1\setminus 2}(\mu) = m_1(\mu) = m_{\sigma 1}(\mu) = m_r(\mu)$.

(FJPush$^+$). 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 3 we obtain

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

where (*) 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. Hence, for every μ_1 it holds that $\mu_1 \models R$, which justifies the rewriting.

(FLPush$^+$)). Similar to the proof of *(FLPush)* for SPARQL set algebra (observe that all rules that are used in the latter proof are also valid in the context of SPARQL bag algebra). □

Group VI

(MReord$^+$), (MMUCorr$^+$), (MJ$^+$). The three equivalences follow easily from the semantics of operator $\setminus$ from Definition 11.

($\widetilde{LJ^+}$). Similar to the proof of *($\widetilde{LJ}$)* for SPARQL set algebra (observe that all rules that are used in the latter proof are also valid in the context of SPARQL bag algebra).

(FLBndI$^+$), (FLBndII$^+$). Similar to the proof of *(FLBndI)* and *(FLBndII)* for SPARQL set algebra (all rules that are used in the latter proof are also valid in the context of SPARQL bag algebra). □

C.6 Proof of Lemma 6

Lemma 6(1): Follows from the semantics of ASK and Lemma 1

Lemma 6(2): Follows from the semantics of ASK queries and the semantics of the UNION operator (cf. Definition 3 and 4).

Lemma 6(3): Follows from the semantics of ASK queries and the semantics of the OPT operator (cf. Definition 4 and 3). In particular, the correctness follows from the semantics of operator $\bowtie$, the algebraic counterpart of operator OPT: it is straightforward to show that (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 6(4): Follows from the semantics of ASK, the semantics of operator AND, and Proposition 2. Observe that $pVars([\![Q_1]\!]_D) \cap pVars([\![Q_2]\!]_D) = \emptyset$ together with Proposition 2 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 $dom(\mu_1) \cap dom(\mu_2) = \emptyset$. □

C.7 Proof of Lemma 7

Lemma 7(1): Follows from the definition of SELECT DISTINCT queries (cf. Appendix A.6) and Lemma 1, which shows that bag and set semantics coincide w.r.t. mapping sets.

Lemma 7(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 7(3): Follows from claims (1) and (2) of the lemma. □

C.8 Proof of Lemma 8

Follows from the observation that for $Q \in \mathcal{AFO}^\pi$ we always have that each mapping $\mu \in [\![Q]\!]_D^+$ has multiplicity one associated (i.e. the semantics coincide) and the fact that the projection on variables $S \supseteq pVars([\![Q]\!]_D)$ does not modify the evaluation result. Please note that the first observation has already been made in the technical report of [1], claiming that for $\mathcal{AFO}$ expressions the multiplicity associated with each result mapping equals to one. □

D. PROOFS OF SEMANTIC RESULTS

D.1 Proof of Theorem 8

REMARK 1. Recall from Section 5 that in the following proof we consider a fragment of SPARQL extended by empty graph patterns $\{\}$ (with semantics $[\![\{\}]\!]_D := \{\emptyset\}$) and by an algebraic MINUS operator (with semantics $[\![Q_1 \text{ MINUS } Q_2]\!]_D := [\![Q_1]\!]_D \setminus [\![Q_2]\!]_D$). To see why empty graph patterns are necessary to obtain the power to encode first-order sentences observe that – in SPARQL without empty patterns – it is impossible to write an ASK query that returns *true* on the empty document. To give a concrete example, in the latter fragment (i.e., the one comprising expression according to Definition 1) the first-order constraint $\varphi := \neg\exists T(c,c,c)$ cannot be encoded as ASK query that returns *true* on every document $D \models \varphi$, because in particular $D := \emptyset \models \varphi$. Contrarily, observe that in SPARQL extended by empty graph patterns (and operator MINUS) we can easily encode φ as ASK($\{\}$ MINUS (c,c,c)).[9]

Concerning the extension by a syntactic MINUS operator it was argued in [1] that this operator can always be simulated using OPTIONAL and FILTER, by help of so-called copy patterns. Unfortunately, the encoding presented there relies on the presence of variables in the right side expression of the MINUS, i.e. fails for expression like Q_1 MINUS (c,c,c). One workaround to fix the construction seems to be the encoding of Q_1 MINUS Q_2 as

$$(Q_1 \text{ OPT } (Q_2 \text{ AND } (?x, ?y, ?z))) \text{ FILTER } (\neg bnd(?x)),$$

[9] We wish to thank Claudio Gutierrez for helpful discussions on the expressiveness of SPARQL and for pointing us to this encoding.

where $?x, ?y, ?z \notin pVars([\![Q_2]\!]_D)$. In fact, this works whenever we forbid empty graph patterns in Q_2. Yet in the general case (i.e. if empty graph patterns occur in Q_2), the encoding fails; unfortunately, such situation may occur in the encoding in the subsequent proof of Theorem 8. To see why the above encoding generally fails in the presence of empty patterns, choose $Q_1 := \{\}$, $Q_2 := \{\}$, and $D := \emptyset$. Then $[\![Q_1 \text{ MINUS } Q_2]\!]_D = \emptyset$, but $[\![(Q_1 \text{ OPT } (Q_2 \text{ AND } (?x, ?y, ?z))) \text{ FILTER } (\neg bnd(?x))]\!]_D = \{\emptyset\}$.

To conclude the discussion, it is an open question if operator MINUS can be encoded by the remaining operators in the presence of empty graph patterns and in response we decided to add the MINUS operator to our fragment. We emphasize, though, that this gives us exactly the same fragment that was used in [1] to prove that SPARQL has the same expressiveness as relational algebra. □

In the subsequent proof of Theorem 8 we assume that the reader is familiar with first-order logic. We show that for each RDF constraint, i.e. each first-order sentence φ over the ternary predicate T, there is a SPARQL query Q_φ such that $[\![\text{ASK}(Q_\varphi)]\!]_D \Leftrightarrow D \models \varphi$. More precisely, we encode a first-order sentence φ that is built using (1) equality formulas of the form $t_1 = t_2$, (2) relational atoms of the form $T(t_1, t_2, t_3)$, (3) the negation operator $\neg$, (4) the conjunction operator $\wedge$, and (5) formulas of the form $\neg\exists x\psi$. We encode φ as a SPARQL query Q_φ s.t. $[\![Q_\varphi]\!]_D = \emptyset$ exactly if $I_D \not\models \varphi$, where $I_D := (\{s, p, o \,|(s, p, o) \in D\}, \{T(s, p, o)|(s, p, o) \in D\})$, i.e. I_D is a structure that has as its domain the values from D and contains a relational fact $T(s, p, o)$ for each triple (s, p, o) in D. It should be mentioned that $\psi_1 \vee \psi_2$ can be written as $\neg(\neg\psi_1 \wedge \neg\psi_2)$ and each quantifier formula can 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 (we chose the variant $\neg\exists x\psi$ because its encoding is simpler than e.g. $\forall x\psi$).

Before presenting the encoding, we introduce some notation and definitions. Let $var(\varphi) := \{x_1, \ldots, x_n\}$ denote all variables appearing in formula φ and define a set $S := \{?x_1, \ldots, ?x_n\}$ of corresponding SPARQL variables. We introduce a total function $v : var(\varphi) \mapsto S$ that translates each variable occurring in φ into its corresponding SPARQL variable, i.e. $v(x_i) := ?x_i$ for $1 \leq i \leq n$. Further assume that $S^\neg \subset V$ is an infinite set of variables disjoint from S. For each subexpression ψ of φ we define an infinite partition $S^\neg_\psi \subset S^\neg$ such that, for each pair of distinct subexpressions $\psi_1 \neq \psi_2$ of φ it holds that $S^\neg_{\psi_1} \cap S^\neg_{\psi_2} = \emptyset$. Based on these partitions, we define for each subexpression ψ of φ its *active domain expression* Q_ψ as follows. Let $free(\psi) := \{v_1, \ldots, v_k\} \subseteq S$ be the free variables in subexpression ψ. Then we define Q_ψ as

$$\begin{array}{c}((v(v_1), ?a_{11}, ?a_{12}) \text{ UNION } (?a_{13}, v(v_1), ?a_{14}) \text{ UNION } (?a_{15}, ?a_{16}, v(v_1)))\\ \text{AND}\\ ((v(v_2), ?a_{21}, ?a_{22}) \text{ UNION } (?a_{23}, v(v_2), ?a_{24}) \text{ UNION } (?a_{25}, ?a_{26}, v(v_2)))\\ \text{AND}\\ \ldots\\ \text{AND}\\ ((v(v_k), ?a_{k1}, ?a_{k2}) \text{ UNION } (?a_{k3}, v(v_k), ?a_{k4}) \text{ UNION } (?a_{k5}, ?a_{k6}, v(v_k)))\end{array}$$

where $?a, ?a_{11}, \ldots, ?a_{16}, \ldots, ?a_{k1}, \ldots, ?a_{k6}$ are pairwise distinct variables taken from $S^\neg_\psi$. 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 the partition that belongs to the respective subexpressions. Further, note that $Q_\varphi := \{\}$ because φ is a sentence and therefore $free(\varphi) = \emptyset$. To give an intuition, each Q_ψ represents all combinations of binding the free variables $v_1, \ldots, v_k$ in ψ (more precisely, the corresponding 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. The subexpressions Q_ψ are only the first step. We set $enc(t) := t$ if t is a constant and $enc(t) := v(t)$ if t is a variable and follow the definition of a formula's semantics by generating all possible bindings for the free variables by induction on the formula's structure. The encoding thus follows the possible structure of φ given in (1)-(5) before:

(1) For $\psi := t_1 = t_2$ we define
$enc(\psi) := Q_\psi \text{ FILTER } (enc(t_1) = enc(t_2))$.
(2) For $\psi := T(t_1, t_2, t_3)$ we define
$enc(\psi) := Q_\psi \text{ AND } (enc(t_1), enc(t_2), enc(t_3))$.
(3) For $\psi := \neg\psi_1$ we define $enc(\psi) := Q_\psi \text{ MINUS } enc(\psi_1)$
(4) For $\psi := (\psi_1 \wedge \psi_2)$ we define
$enc(\psi) := enc(\psi_1) \text{ AND } enc(\psi_2)$.
(5) For $\psi := \neg\exists x\psi_1$ we define $enc(\psi) := Q_\psi \text{ MINUS } enc(\psi_1)$.

We now sketch the idea behind the encoding. It satisfies the following two properties: ($\Rightarrow$) foreach interpretation[10] $\mathcal{I} := (I_D, \gamma)$ such that $\mathcal{I} \models \varphi$ there exists a mapping $\mu \in [\![enc(\varphi)]\!]_D$ such that $\mu \supseteq \{?x_1 \mapsto \gamma(x_1), \ldots, ?x_n \mapsto \gamma(x_n)\}$ and ($\Leftarrow$) foreach mapping $\mu \in [\![enc(\varphi)]\!]_D$ it holds that every interpretation (I_D,γ) with $\gamma(x_i) := \mu(?x_i)$ for $1 \leq i \leq n$ satisfies φ. Both directions together imply the initial claim, since $I_D \not\models \varphi \Leftrightarrow [\![enc(\varphi)]\!]_D = \emptyset$.

The two directions can be proven by induction on the structure of formulas. Concerning the two basic cases (1) and (2) observe that, in their encoding, the active domain expressions generate the universe of all solutions, which is then restricted either 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) $\psi := T(t_1, t_2, t_3)$). In the induction step there are three cases that remain to be shown. First, the idea of the encoding for $\psi := \neg\psi_1$ is that we subtract from the universe of all solutions exactly the solutions of ψ_1, encoded by $enc(\psi_1)$. Second, a conjunction $\psi := \psi_1 \wedge \psi_2$ is straightforwardly mapped to a join operation between the encodings of ψ_1 and ψ_2. Third, the encoding for $\psi := \neg\exists x\varphi$ is similar to the encoding for the negation; observe, however, that in this case $?x \notin free(\psi)$, so the active domain expression does not contain variable $?x$ anymore, which can be understood as an implicit projection. □

D.2 Proof of Lemma 9

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 (from the correctness of the translation) that $Q' \equiv_\Sigma Q$. □

D.3 Proof of Lemma 10

Direction $\Rightarrow$ follows from Lemma 9, so it suffices to prove direction $\Leftarrow$. So let us assume that $Q' \equiv_\Sigma Q$ and Q' is minimal. First observe that both $cq^{-1}((cq(Q'))^\Sigma)$ and $cq^{-1}((cq(Q))^\Sigma)$ are $\mathcal{A}^\pi$-expressions. It follows that $cq(Q') \equiv_\Sigma cq(Q)$. From this observation, the minimality of Q', and the correctness of the translation it follows that $cq(Q') \in cb_\Sigma(cq(Q))$ and $Q' \in cq^{-1}(cb_\Sigma(cq(Q)))$. □

[10] An interpretation is a pair of a structure and function γ that maps variables to elements of the structure's domain.

D.4 Proof of Lemma 11

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 on the variables 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). Given that $?y \notin S$ and the observation that μ is obtained from μ_l by projecting on S, we conclude that μ is also obtained from μ_r when projecting on 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 on 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 *(FElim)* from Lemma 4). Given that $?y \notin S$ and the observation that μ is obtained from μ_r by projecting on S, we conclude that μ is also obtained from μ_l when projecting for S. Consequently, the left side expression $[\![\text{SELECT}_S(Q_2)]\!]_D$ generates μ.

Rule (FSII): Follows trivially by the observation that, by assumption, each $\mu \in [\![Q_2]\!]_D$ satisfies the filter condition $?x = ?y$.

Rule (FSIII): First note that preconditions $?x \in pVars([\![Q_2]\!]_D)$ and $Q_2 \in \mathcal{A}$ imply that $?x \in cVars([\![Q_2]\!]_D)$. From Proposition 1 we obtain that $?x \in dom(\mu_2)$ foreach $\mu_2 \in [\![Q_2]\!]_D$ and it easily follows from Definition 6 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 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 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 to the initial assumption that there is no compatible mapping $\mu_2 \sim \mu_\setminus$ in $[\![Q_2]\!]_D$. □

D.5 Proof of Lemma 12

Rule (OSI): We transform $Q := [\![Q_1 \text{ OPT } Q_2]\!]_D$ systematically. Let D be an RDF database s.t. D satisfies all constraints in Σ. Then

$$\begin{aligned}
&[\![Q]\!]_D = [\![Q_1 \text{ OPT } Q_2]\!]_D \\
&= ([\![Q_1]\!]_D \bowtie [\![Q_2]\!]_D) \cup ([\![Q_1]\!]_D \setminus [\![Q_2]\!]_D) \\
&= [\![Q_1 \text{ AND } Q_2]\!]_D \cup (\pi_{pVars([\![Q_1]\!]_D)}([\![Q_1 \text{ AND } Q_2]\!]_D) \setminus [\![Q_2]\!]_D)
\end{aligned}$$

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 s.t. $D \models \Sigma$. We transform 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 straightforward. □