SPARQLing Datalog for Rule-Based Reasoning over Large Knowledge Graphs

Alex Ivliev, Markus Krötzsch, and Maximilian Marx

Knowledge-Based Systems Group, TU Dresden
{alex.ivliev, markus.kroetzsch, maximilian.marx}@tu-dresden.de

Abstract We propose a new integration of rule reasoners with one or more RDF stores, using selective SPARQL queries to fetch relevant data. In contrast to previous implementations that merely import results of fixed SPARQL queries, our approach relies on pure logic programs over RDF triple data. Transparent to the user, optimised SPARQL queries then are constructed and evaluated during reasoning. To ensure good performance, we develop optimisation methods that adopt ideas from logic program optimisation, including semi-naive evaluation, magic sets, and static filtering. Based on the integration of our methods into the open source rule engine Nemo, we empirically evaluate our approach with complex rule sets over large knowledge graphs.

1 Introduction

Rule languages have a long history in relation to the Semantic Web, knowledge graphs, and ontologies [12,13,15,11,24,5,2,6,30]. Indeed, rules embody an attractive combination of declarative knowledge modelling and efficient practical evaluation that scales to large data sets. Example uses include ontological reasoning [13,19], query answering [29,5], graph transformation [27,8], and general data analysis [25]. Datalog has emerged as a robust and extensible rule language that can support many of these applications [21], and several modern Datalog systems natively support RDF [24,28,14].

Nevertheless, the use of rules over modern knowledge graphs remains challenging due to their increasing scale and dynamicity. The RDF export of Wikidata [9], e.g., currently consists of over 17 billion triples[1] with update rates reaching over 100 triples per second.[2] Rule reasoners with their highly recursive computations and analytic query patterns are often designed as in-memory systems,[3] which would require main memory in the range of terabytes to load such data.

As a possible way forward, some rule systems support the use of RDF database backends for managing large knowledge graphs, and access data by means of SPARQL queries [14,6].[4] This is a meaningful setting, since Datalog has many

[1] Source: grafana.wikimedia.org, triples of *main* + *scholarly* graph, 04 Dec 2025
[2] Source: grafana.wikimedia.org, WDQS streaming updater, 04 Dec 2025
[3] DBMS-based rule engines exist, but tend to be significantly slower [3].
[4] RDFox also added this feature recently; see docs.oxfordsemantic.tech

benefits that even an efficient SPARQL implementation cannot offer. For example, reasoning for lightweight ontology languages like OWL EL and OWL RL is possible in Datalog [19,20], but not in SPARQL. This is typical for PTime-hard problems, since SPARQL can only solve problems in NL (data complexity), widely conjectured to be strictly smaller than PTime.[5] Even tasks in NL often require Datalog. For example, SPARQL can check for recursive reachability (via path patterns), but not if additional conditions are required of each element along the path. In Wikidata, e.g., statements are represented by own graph nodes that can be marked as *deprecated*: checking reachability via non-deprecated statements is easy in Datalog but impossible in SPARQL. Finally, even in cases where SPARQL could suffice in principle, more advanced analyses quickly lead to excessively large and complex queries that are hard to maintain and slow to execute. For example, SPARQL has been shown to support OWL QL reasoning, but the resulting queries are too large to be feasible in practice [4].

It is therefore well-motivated to use Datalog on top of SPARQL databases to combine expressive power with scalable data management. However, in current systems, this requires users to be proficient in both SPARQL and Datalog, and to decide how to split the work between both technologies. We therefore propose a new approach where RDF data is still fetched from a (possibly remote) RDF database, but the required SPARQL queries are automatically computed and optimised by the system. From a user perspective, rules refer to RDF data as if it were locally loaded, and no knowledge of SPARQL is required.

To make this work, we develop static (compile-time) and dynamic (inference-time) optimisations to reduce the amount of data that needs to be transferred. We thereby take inspiration from several optimisation methods from logic programming, notably *semi-naive evaluation* [1], *magic sets* [1], and *static filtering* [10]. Since we support the use of data from several RDF databases, our work also can be seen as a framework for federated RDF query answering with a rule-based query language. Indeed, our basic SPARQL compilation resembles an optimised version of *exclusive groups* [26]. One way in which our optimisations improve upon these known methods is by avoiding Cartesian products.

We have integrated our approach into *Nemo* [14], which forms the basis for an empirical evaluation. Comparing our implementation to other Datalog engines that support SPARQL data imports, we observe advantages in terms of runtime and transferred data. A subsequent ablation study indicates that each of our optimisation steps individually contributes to these overall performance gains.

Regarding RDF and SPARQL, some general familiarity is sufficient to follow our paper; regarding Datalog, we provide a self-contained introduction. Evaluation resources are available online; additional proof details for our theoretical results are in the appendix.

[5] For RL and EL, Krötzsch's analyses of minimal Datalog predicate arity also exclude SPARQL without relying on complexity-theoretical conjectures [19,20].

2 Datalog over RDF

We consider a minimal rule language that includes only the features relevant to our approach: plain Datalog with input negation, extended with special RDF input predicates to access triples from one or more RDF datasets. Our approach is compatible with many further Datalog features, such as built-in filter expressions, stratified negation, or aggregation operators [21], but they are not needed to present our contributions.

Terms We consider a set $\mathbf{C}$ of *constants* that includes, in particular, all IRIs and RDF literals [7], and a set $\mathbf{V}$ of *variables* disjoint from $\mathbf{C}$. The set of (*Datalog*) *terms* is $\mathbf{T} = \mathbf{C} \cup \mathbf{V}$. We generally disallow RDF blank nodes (bnodes) in data, rules, and queries. On the one hand, SPARQL cannot be used to retrieve information on specific bnodes in data: query results may use arbitrary (local) bnode ids, and bnodes in queries are semantically equivalent to unselected variables, which may represent any RDF term. On the other hand, bnodes in rules and queries can always be replaced by suitably quantified variables.[6]

Rules and Facts An *atom* has the form $p(t_1, \ldots, t_k)$ where p is a *predicate symbol* of *arity* $\mathsf{ar}(p) = k$, and each t_i for $1 \leq i \leq k$ is a term. We write $\mathbf{P}$ for the set of all predicate symbols, and denote lists of terms $\langle t_1, \ldots, t_k \rangle$ as $\boldsymbol{t}$, where $|\boldsymbol{t}| = k$ (and similarly for lists of constants $\boldsymbol{c}$ and lists of variables $\boldsymbol{x}$). A *literal* is an expression that is an atom A or a negated atom $\neg A$.[7] A *rule* ρ has the form

$$H \leftarrow B_1 \wedge \cdots \wedge B_n \wedge \neg B_{n+1} \wedge \cdots \wedge \neg B_\ell \tag{1}$$

where H and B_i $(1 \leq i \leq \ell)$ are atoms, called *head* and *body*, respectively. We use notation $\mathsf{head}(\rho) = H$, $\mathsf{body}^+(\rho) = \{B_1, \ldots, B_n\}$, and $\mathsf{body}^-(\rho) = \{B_{n+1}, \ldots, B_\ell\}$. We require that each variable in ρ also occurs in at least one atom in $\mathsf{body}^+(\rho)$ (*safety*). We allow $\ell = 0$ and omit $\leftarrow$ in this case: then H contains no variables (by safety) and is called a *fact*.

Programs and Datasets A (*Datalog*) *program* is a triple $\langle \Pi, \mathbf{P}_{\mathsf{in}}, \mathbf{P}_{\mathsf{out}} \rangle$ where Π is a set of rules and $\mathbf{P}_{\mathsf{in}}$ and $\mathbf{P}_{\mathsf{out}}$ are finite sets of *input* and *output predicates*, respectively, such that (a) input predicates only occur in rule bodies, and (b) negated atoms only use input predicates. A *dataset* for $\langle \Pi, \mathbf{P}_{\mathsf{in}}, \mathbf{P}_{\mathsf{out}} \rangle$ is a finite set $\mathcal{D}$ of facts that use only predicates from $\mathbf{P}_{\mathsf{in}}$. We often omit $\mathbf{P}_{\mathsf{in}}$ and $\mathbf{P}_{\mathsf{out}}$ when clear from the context.

RDF Graphs as Inputs An RDF graph G is a finite set of triples $\langle s, p, o \rangle$ with s and p an IRI, and o an IRI or RDF literal.[8] Given a dataset $\mathcal{D}$, the input predicate $p \in \mathbf{P}_{\mathsf{in}}$ is an *RDF import for* G if $\mathsf{ar}(p) = 3$ and $p(s, p, o) \in \mathcal{D}$ if and only if $\langle s, p, o \rangle \in G$.

[6] Datalog has been generalised to *existential rules* to add the capacity for value invention [23,3], which corresponds to the use of bnodes in inferred facts.

[7] We always use *literal* in this sense and write *RDF literal* to refer to the RDF term.

[8] As before, we exclude bnodes. The other conditions reflect syntactic restrictions of the RDF standard; in Datalog, we generally allow arbitrary constants in all positions.

Example 1. The following Datalog program uses RDF import predicate triple and output predicate label to find all labels given for any subproperty of rdfs:label (we use common namespace abbreviations for readability):

$$\mathsf{labelProp}(\mathsf{rdfs{:}label}) \tag{2}$$

$$\mathsf{labelProp}(x) \leftarrow \mathsf{triple}(x, \mathsf{rdfs{:}subPropertyOf}, y) \wedge \mathsf{labelProp}(y) \tag{3}$$

$$\mathsf{label}(x, y) \leftarrow \mathsf{labelProp}(z) \wedge \mathsf{triple}(x, z, y) \tag{4}$$

Semantics A *substitution* σ is a mapping from finitely many variables to constants. For a term, rule, atom, or set of such expressions φ, we write $\varphi\sigma$ for the result of replacing each variable x in φ for which σ is defined by $\sigma(x)$. For a program Π and a set of facts $\mathcal{I}$, we define

$$\Pi(\mathcal{I}) \coloneqq \{\mathsf{head}(\rho)\sigma \mid \text{there is } \rho \in \Pi \text{ and a substitution } \sigma \text{ such that } \mathsf{body}^+(\rho)\sigma \subseteq \mathcal{I} \text{ and } \mathsf{body}^-(\rho)\sigma \cap \mathcal{I} = \emptyset\} \tag{5}$$

Given a dataset $\mathcal{D}$ for Π, let $\mathcal{D}^0 \coloneqq \mathcal{D}$, let $\mathcal{D}^{i+1} \coloneqq \mathcal{D}^i \cup \Pi(\mathcal{D}^i)$ for all $i \geq 0$, and let $\mathcal{D}^\infty \coloneqq \bigcup_{i \geq 0} \mathcal{D}^i$. The *output* $\mathsf{out}(\Pi, \mathcal{D})$ of Π over $\mathcal{D}$ is the set of all facts $p(\boldsymbol{t}) \in \mathcal{D}^\infty$ with $p \in \mathbf{P}_{\mathsf{out}}$.

Since there are only finitely many possible facts over the constants and predicates in $\mathcal{D}$ and Π, the monotonically increasing sequence $\mathcal{D}^0 \subseteq \mathcal{D}^1 \subseteq \ldots$ reaches a fixed point $\mathcal{D}^j = \mathcal{D}^{j+1}$ after finitely many steps j, and $\mathcal{D}^\infty = \mathcal{D}^j$.

Example 2. For the program Π of Example 1 and imported RDF graph

$$\{\langle \mathsf{schema{:}name}, \mathsf{rdfs{:}subPropertyOf}, \mathsf{rdfs{:}label}\rangle, \langle a, \mathsf{rdfs{:}label}, \text{"Alice"}\rangle, \langle b, \mathsf{schema{:}name}, \text{"Bob"}\rangle, \},$$

the output is $\{\mathsf{label}(a, \text{"Alice"}), \mathsf{label}(b, \text{"Bob"})\}$. With $\mathcal{D}$ the imported dataset, we get $\mathcal{D}^1 = \mathcal{D} \cup \{\mathsf{labelProp}(\mathsf{rdfs{:}label})\}$, $\mathcal{D}^2 = \mathcal{D}^1 \cup \{\mathsf{labelProp}(\mathsf{schema{:}name}), \mathsf{label}(a, \text{"Alice"})\}$, and $\mathcal{D}^3 = \mathcal{D}^2 \cup \{\mathsf{label}(b, \text{"Bob"})\} = \mathcal{D}^4 = \mathcal{D}^\infty$.

3 Incremental Data Imports via SPARQL

RDF imports through triple predicates are the simplest, most direct interface of rules and RDF data. Datalog systems that use similar triple predicates for presenting RDF data include Nemo [14], RDFox [24], and VLog [6]. When data is stored in a suitable RDF database management system, triple data can also be obtained by the SPARQL query `SELECT ?s ?p ?o WHERE { ?s ?p ?o }`.

However, running this query on a database with billions of triples is not practical, and more selective queries that only retrieve relevant triples would be desirable. For example, the triple atom in rule (3) only requires us to retrieve subjects of triples with predicate rdfs:subPropertyOf and some object o for which $\mathsf{labelProp}(o)$ holds true. We therefore introduce SPARQL-based import rules that incorporate data bindings from the Datalog program evaluation.

The next definition (like our earlier definitions) specifies an abstract syntactic structure, without considering concrete aspects of practical encoding. We assume that the variables used in SPARQL queries are in our variable set $\mathbf{V}$.

Definition 1. *A* SPARQL import rule *for a program* $\langle \Pi, \mathbf{P}_{\mathsf{in}}, \mathbf{P}_{\mathsf{out}} \rangle$ *has the form*

$$p(\boldsymbol{x}) \leftarrow Q@ep \wedge B_1 \wedge \ldots \wedge B_n \tag{6}$$

where $p \in \mathbf{P}_{\mathsf{in}}$ *is an input predicate,* Q *is a SPARQL query with result variables* $\boldsymbol{x}$*,* ep *is the URL of a SPARQL service endpoint, and* $B_1, \ldots, B_n$ *are atoms, called* binding atoms*, that only contain variables from* $\boldsymbol{x}$ *or constants.*

Example 3. Let $Q[s, p, o]$ be the above SPARQL query for all triples, with variables s, p, and o. To replace the triple predicate in rule (3) of Example 1, we can define a SPARQL import rule

$$\mathsf{spoTriple}(s, p, o) \leftarrow Q@ep \wedge p{=}\mathsf{rdfs{:}subPropertyOf} \wedge \mathsf{labelProp}(o) \tag{7}$$

for a new input predicate spoTriple and a suitable endpoint ep. We use a predicate =rdfs:subPropertyOf of arity 1, which could be defined by a single program fact. In Section 4, we just incorporate constants into queries. Note that bindings may use recursively defined predicates, such as labelProp.

Definition 2. *Consider a SPARQL import rule* ρ *of the form* (6)*, and let* Q^{ep} *be the set of all tuples that only contain IRIs or RDF literals.*[9] *For a set of facts* $\mathcal{I}$ *we define*

$$\rho(\mathcal{I}) \coloneqq \{p(\boldsymbol{x})\sigma \mid \boldsymbol{x}\sigma \in Q^{ep} \text{ and } B_1\sigma, \ldots, B_n\sigma \in \mathcal{I}\} \tag{8}$$

For a Datalog program Π*, a set* Σ *of SPARQL import rules, and a dataset* $\mathcal{D}$*, we define* $\mathcal{D}^0 \coloneqq \mathcal{D}$*,* $\mathcal{D}^{i+1} \coloneqq \mathcal{D}^i \cup \Pi(\mathcal{D}^i) \cup \bigcup_{\rho \in \Sigma} \rho(\mathcal{D}^i)$ *for all* $i \geq 0$*, and* $\mathcal{D}^\infty \coloneqq \bigcup_{i \geq 0} \mathcal{D}^i$*. We write* $\mathsf{out}(\Pi, \Sigma, \mathcal{D})$ *for the set of all output facts in* $\mathcal{D}^\infty$*.*

As before, the sequence $\mathcal{D}^0 \subseteq \mathcal{D}^1 \subseteq \ldots$ reaches the fixed point $\mathcal{D}^\infty$ in finitely many steps. SPARQL import rules ρ can be evaluated incrementally during this computation by syntactically restricting the SPARQL query to ensure that all results agree with a combination of facts for the binding atoms. The `VALUES` operator of SPARQL 1.1 can be used for that purpose:

Definition 3. *Consider a SPARQL import rule* ρ *of the form* (6)*. For a binding atom* $B_i = q(y_1, \ldots, y_m)$ *of* ρ *and a set of facts* $\mathcal{I}_q = \{q(c_1^1, \ldots, c_m^1), \ldots, q(c_1^\ell, \ldots, c_m^\ell)\}$*, let* $\mathsf{values}(B_i, \mathcal{I}_q)$ *be the following SPARQL expression:*

$$\texttt{VALUES}(y_1 \ \cdots \ y_m)\{(c_1^1 \ \cdots \ c_m^1) \ldots (c_1^\ell \ \cdots \ c_m^\ell)\} \tag{9}$$

Given fact sets $\mathcal{I}_1, \ldots, \mathcal{I}_n$ *for each binding atom in* ρ*,* $Q[\mathcal{I}_1, \ldots, \mathcal{I}_n]$ *is the SPARQL query obtained from* Q *by adding the expressions* $\mathsf{values}(B_i, \mathcal{I}_i)$ *for all* $1 \leq i \leq n$ *to the outermost group graph pattern in* Q*.*

[9] As mentioned before, we assume that no bnodes occur anywhere. We also filter unbound values here: they will never occur in the queries we use later on.

In practice, encoding possible values like this can lead to large queries, but we observed that current public SPARQL services can successfully handle in the order of 10^4–10^5 tuples in a single query. For even larger numbers of bindings, our evaluation prototype used in Section 6 partitions $\mathcal{I}_1 \times \cdots \times \mathcal{I}_n$ over several requests and combines the results. Our use of several binding atoms B_i and corresponding `VALUES` clauses already contributes greatly to the reduction in query size in cases where binding atoms do not share variables, and would therefore lead to a Cartesian product when incorporated into a single binding relation. We take this into account when extracting SPARQL import rules in Section 4.

A naive reasoning approach that uses all current binding facts in each iteration would still be impractical, since it would include all bindings from previous steps, leading to redundant results and increasingly large SPARQL queries. We prevent this by adopting the so-called *semi-naive* Datalog evaluation strategy [1] to SPARQL import rules, which allows iterative queries to focus on the subset of newly derived facts $\Delta \subseteq \mathcal{I}$:

Definition 4. *For a set of facts $\mathcal{I}$ and predicate q, let $\mathcal{I}|_q$ be the set of all q-facts in $\mathcal{I}$. Consider a SPARQL import rule ρ as in* (6)*, where q_i is the predicate of binding atom B_i. For a set of facts $\mathcal{I}$, a subset $\Delta \subseteq \mathcal{I}$, and a number $1 \leq j \leq n$, the* semi-naive rewriting *$Q[\mathcal{I}, \Delta, j]$ of Q is the SPARQL query $Q[\mathcal{J}_1^-, \ldots, \mathcal{J}_{j-1}^-, \mathcal{J}_j^{\Delta}, \mathcal{J}_{j+1}^+, \ldots, \mathcal{J}_n^+]$ as given in Definition 3 for the sets of facts $\mathcal{J}_i^- := (\mathcal{I} \setminus \Delta)|_{q_i}$, $\mathcal{J}_i^{\Delta} := \Delta|_{q_i}$, and $\mathcal{J}_i^+ := \mathcal{I}|_{q_i}$, for each $1 \leq i \leq n$. Evaluating all semi-naive rewritings for $\mathcal{I}$ and Δ over the given endpoint ep, we obtain:*

$$\rho(\mathcal{I}, \Delta) := \bigcup_{j=1}^{n} \{p(\boldsymbol{c}) \mid \boldsymbol{c} \in Q[\mathcal{I}, \Delta, j]^{ep}\} \tag{10}$$

For a Datalog program Π, a set Σ of SPARQL import rules, and a dataset $\mathcal{D}$, we define $\mathcal{D}^0 := \mathcal{D}$ and $\Delta^0 := \mathcal{D}$, $\mathcal{D}^{i+1} := \mathcal{D}^i \cup \Pi(\mathcal{D}^i) \cup \bigcup_{\rho \in \Sigma} \rho(\mathcal{D}^i, \Delta^i)$ and $\Delta^{i+1} = \mathcal{D}^{i+1} \setminus \mathcal{D}^i$ for all $i \geq 0$, and $\mathcal{D}^{\infty} := \bigcup_{i \geq 0} \mathcal{D}^i$.

The evaluation result defined in (10) can equivalently be defined by adding to (6) the requirement that there is some j such that $B_j\sigma \in \Delta$. Indeed, a tuple $\boldsymbol{c} = \langle c_1, \ldots, c_k \rangle$ is in $Q[\mathcal{I}, \Delta, j]^{ep}$ if and only if $\boldsymbol{c} \in \rho(\mathcal{I})$ and j is the smallest number in $\{1, \ldots, n\}$ for which $q_j(\boldsymbol{c}) \in \Delta$. In particular, $\rho(\mathcal{D}^{i+1}, \Delta^{i+1})$ therefore includes all tuples of $\rho(\mathcal{D}^{i+1})$ that were not already in $\rho(\mathcal{D}^i)$. We therefore obtain the following correctness result:

Theorem 1. *For every Datalog program Π with input rules Σ and input dataset $\mathcal{D}$, the semi-naive incremental SPARQL import of Definition 4 yields the same result $\mathcal{D}^{\infty}$ as the import of all query results as per Definition 2.*

Standard semi-naive evaluation can of course also be used for optimising the evaluation of Datalog rules, but this is not our concern here. Our new implementation ensures that suitable bindings are passed on to SPARQL imports, while relying on an existing (semi-naive) rule engine for standard Datalog rules.

4 From Triple Predicates to SPARQL Imports

To apply the efficient SPARQL import strategy of Section 3 to Datalog over RDF as introduced in Section 2, we rewrite triple atoms into more specialised SPARQL queries, and define suitable binding atoms. Therefore, we assume that some RDF import predicates $t \in \mathbf{P}_{\mathsf{in}}$ are associated with a SPARQL service URL ("endpoint") $\mathsf{orig}(t)$ that is the origin of the imported triples; for all other "local" predicates p, we set $\mathsf{orig}(p) = \mathsf{loc}$. As before, we require that the imported triple facts $p(s, p, o)$ in input datasets $\mathcal{D}$ agree with the RDF triples $\langle s, p, o\rangle$ accessible at the endpoint $\mathsf{orig}(p)$. This is a purely conceptual view to ease our presentation; we will not materialise $\mathcal{D}$ in practice.

We extend orig to negated or non-negated atoms (literals) L by setting $\mathsf{orig}(L) = \mathsf{orig}(p)$ for the predicate p of L. Moreover, if a set $\mathbf{L}$ contains only literals with the same origin, we may denote the latter by $\mathsf{orig}(\mathbf{L})$. For E a logical expression or set of such expressions, we write $\mathsf{var}(E)$ for the set of all variables in E.

Conjunctions of same-origin RDF import literals can be rewritten to SPARQL:

Definition 5. *For a set of literals* $\mathbf{L}$ *with uniform* $\mathsf{orig}(\mathbf{L}) = u \neq \mathsf{loc}$*, and a set of variables* $V \subseteq \mathsf{var}(\mathbf{L})$*, we write* $\mathsf{sparql}(\mathbf{L}, V)$ *for the SPARQL query* `SELECT` V `WHERE` G*, where* G *is the group graph pattern that contains (1) for every* $t(s, p, o) \in \mathbf{L}$*, the triple pattern* `s p o .` *and (2) for every* $\neg t(s, p, o) \in \mathbf{L}$*, the expression* `FILTER NOT EXISTS{ s p o . }`*.*

Combining triple patterns that refer to the same endpoint is a known method in SPARQL federation, where such combined patterns are known as *exclusive groups* [26]. However, blindly merging such groups may lead to inefficient queries if the resulting join patterns contain a Cartesian product.[10] We therefore group (body) literals into *components* based on origin *and* variable structure:

Definition 6. *For a set of atoms* $\mathbf{A}$*, let* $\sim$ *be the smallest transitive relation* $\sim \subseteq \mathbf{A} \times \mathbf{A}$ *such that* $A \sim B$ *holds if* $\mathsf{var}(A) \cap \mathsf{var}(B) \neq \emptyset$ *and* $\mathsf{orig}(A) = \mathsf{orig}(B)$*. The set* $\mathsf{Cmp}(\mathbf{A})$ *of* components of $\mathbf{A}$ *is the set of all* $\sim$ *equivalence classes.*

For a set of literals $\mathbf{L}$*, let* $\mathbf{L}^+$ *be the set of all non-negated atoms in* $\mathbf{L}$*, and let* $\mathbf{L}^-$ *be the set of all negated atoms in* $\mathbf{L}$*. For a subset* $\mathbf{M} \subseteq \mathbf{L}$*, we define the set* $\mathsf{LOC}_{\mathbf{L}}(\mathbf{M})$ *of* locally overlapping components *as*

$$\mathsf{LOC}_{\mathbf{L}}(\mathbf{M}) = \{\mathbf{D} \in \mathsf{Cmp}(\mathbf{L}^+) \mid \mathsf{orig}(\mathbf{D}) = \mathsf{loc}, \mathsf{var}(\mathbf{M}) \cap \mathsf{var}(\mathbf{D}) \neq \emptyset\}.$$

For a component $\mathbf{C} \in \mathsf{Cmp}(\mathbf{L}^+)$ *the* extended component $\mathsf{ec}_{\mathbf{L}}(\mathbf{C})$ *is the set*

$$\mathbf{C} \cup \{\neg B \in \mathbf{L}^- \mid \mathsf{orig}(\neg B) = \mathsf{orig}(\mathbf{C}), \mathsf{var}(\neg B) \subseteq \mathsf{var}(\mathbf{C} \cup \bigcup \mathsf{LOC}_{\mathbf{L}}(\mathbf{C}))\}.$$

The set of extended components of $\mathbf{L}$ *is denoted* $\mathsf{EC}(\mathbf{L})$*. Finally,* $\mathsf{nocmp}(\mathbf{L})$ *is the set* $\mathbf{L} \setminus \bigcup \mathsf{EC}(\mathbf{L})$ *of literals that are not in any extended component.*

[10] This is mostly ignored in SPARQL federation, possibly since typical SPARQL queries do not reach a size and complexity where products are likely.

Algorithm 1: Function $\texttt{sparqlImport}(\mathbf{L}, q[V], \mathbf{B}, \Pi, \Sigma)$

Input: literal set $\mathbf{L}$ with $\mathsf{orig}(\mathbf{L}) \neq \mathsf{loc}$, import atom $q[V]$, rule body $\mathbf{B}$, program Π, SPARQL import rules Σ
Output: extended program Π and SPARQL import rules Σ

1 $\mathbf{B}_{import} \coloneqq \mathsf{sparql}(\mathbf{L}, V)@\mathsf{orig}(\mathbf{L})$
2 **for** $\mathbf{D} \in \mathsf{LOC}_{\mathbf{B}}(\mathbf{L})$ **do**
3 $\quad W \coloneqq \mathsf{var}(\mathbf{D}) \cap \mathsf{var}(\mathbf{L})$
4 $\quad \mathbf{B}_{import} \coloneqq \mathbf{B}_{import} \wedge b[W]$ for a fresh predicate b of arity $|W|$
5 $\quad \Pi \coloneqq \Pi \cup \{b[W] \leftarrow \mathsf{ec}_{\mathbf{B}}(\mathbf{D})\}$
6 **return** $\langle \Pi, \Sigma \cup \{q[V] \leftarrow \mathbf{B}_{import}\}\rangle$

Intuitively, the components of Definition 6 are connected components of the "shares-some-variable" graph over a subset of atoms of the same origin. That $\sim$ is an equivalence relation is immediate from its definition. Locally binding components of $\mathbf{M}$ are components with origin loc that share some variable with $\mathbf{M}$. If $\mathbf{C} \in \mathsf{Cmp}(\mathbf{L}^+)$ with $\mathsf{orig}(\mathbf{C}) = \mathsf{loc}$, then this is simply $\mathsf{LOC}_{\mathbf{L}}(\mathbf{C}) = \{\mathbf{C}\}$. The extended component then adds all negated literals of the same origin that have all of their variables covered by the union of the binding components. Negative literals that do not satisfy this condition are collected in $\mathsf{nocmp}(\mathbf{L})$. Note that every positive literal is in some (extended) component by definition, so $\mathsf{nocmp}(\mathbf{L}) \subseteq \mathbf{L}^-$.

Example 4. Consider the set $\mathbf{L}$ of the following literals:

$$p(x,y) \qquad g(v,w) \qquad t(y,p_1,z) \qquad t(z,p_2,v) \qquad s(v,p_4,w)$$
$$\neg r(x,v) \qquad \neg a(w) \qquad \neg t(x,p_3,y) \qquad \neg s(x,p_5,w)$$

where $\mathsf{orig}(p) = \mathsf{orig}(g) = \mathsf{orig}(r) = \mathsf{orig}(a) = \mathsf{loc}$, $\mathsf{orig}(t) = e$, and $\mathsf{orig}(s) = f$; $x, y, z, v, w \in \mathbf{V}$; and $p_1, \ldots, p_5 \in \mathbf{C}$. The components $\mathsf{Cmp}(\mathbf{L}^+)$ are: $\mathbf{C}_1 = \{p(x,y)\}$, $\mathbf{C}_2 = \{g(v,w)\}$, $\mathbf{C}_3 = \{t(y,p_1,z), t(z,p_2,v)\}$, and $\mathbf{C}_4 = \{s(v,p_4,w)\}$. We find that $\mathsf{LOC}_{\mathbf{L}}(\mathbf{C}_3) = \{\mathbf{C}_1, \mathbf{C}_2\}$, and therefore $\mathsf{ec}_{\mathbf{L}}(\mathbf{C}_3) = \mathbf{C}_3 \cup \{\neg t(x,p_3,y)\}$. In contrast, $\mathsf{LOC}_{\mathbf{L}}(\mathbf{C}_4) = \{\mathbf{C}_2\}$, so $\neg s(x,p_5,w) \notin \mathsf{ec}_{\mathbf{L}}(\mathbf{C}_4)$ (since x is not covered) and $\neg a(w) \notin \mathsf{ec}_{\mathbf{L}}(\mathbf{C}_4)$ (since $\mathsf{orig}(a) \neq f$). However, $\neg a(w) \in \mathsf{ec}_{\mathbf{L}}(\mathbf{C}_2)$ and we obtain $\mathsf{nocmp}(\mathbf{L}) = \{\neg s(x,p_5,w), \neg r(x,v)\}$.

Algorithm 2 presents our procedure for extracting SPARQL import rules and bindings from a Datalog program with RDF imports, using some further notation as explained next. First, we often treat conjunctions of literals as sets of literals (e.g., in Line 10). Second, for a finite set $V \subseteq \mathbf{V}$ of variables and predicate $p \in \mathbf{P}$, we write $p[V]$ for the atom $p(\boldsymbol{v})$ where $\boldsymbol{v} = \langle v_1, \ldots, v_n\rangle$ is an arbitrary but fixed total order of V. The function `sparqlImport` of Algorithm 1 constructs the actual SPARQL import and the auxiliary rules needed to compute its bindings. Algorithm 2 transforms each rule of Π (L9) and eventually inserts a rewritten rule (L19). We explain the steps of rewriting one rule in an example.

Algorithm 2: SPARQL Import Extraction $\texttt{extractSparql}(\Pi)$

Input: program Π
Output: rewritten program Π' with SPARQL import rules Σ

7 $\Pi' \coloneqq \emptyset$
8 $\Sigma \coloneqq \emptyset$
9 **for** $H \leftarrow \mathbf{B} \in \Pi$ **do**
10 $\quad \mathbf{B}_{new} \coloneqq \bigwedge\{L \in \mathbf{B} \mid \mathsf{orig}(L) = \mathsf{loc}\}$
11 $\quad$ **for** $\mathbf{E} \in \mathsf{EC}(\mathbf{B})$ *and* $\mathsf{orig}(\mathbf{E}) \neq \mathsf{loc}$ **do**
12 $\quad\quad V \coloneqq \mathsf{var}(\mathbf{E}) \cap \mathsf{var}(\{H\} \cup \mathbf{B} \setminus \mathbf{E})$
13 $\quad\quad \mathbf{B}_{new} \coloneqq \mathbf{B}_{new} \wedge q[V]$ for a fresh predicate q of arity $|V|$
14 $\quad\quad \langle \Pi', \Sigma \rangle \coloneqq \texttt{sparqlImport}(\mathbf{E}, q[V], \mathbf{B}, \Pi', \Sigma)$
15 $\quad$ **for** $\neg A \in \mathsf{nocmp}(\mathbf{B})$ *and* $\mathsf{orig}(A) \neq \mathsf{loc}$ **do**
16 $\quad\quad V \coloneqq \mathsf{var}(\neg A)$
17 $\quad\quad \mathbf{B}_{new} \coloneqq \mathbf{B}_{new} \wedge \neg q[V]$ for a fresh predicate q of arity $|V|$
18 $\quad\quad \langle \Pi', \Sigma \rangle \coloneqq \texttt{sparqlImport}(\{A\}, q[V], \mathbf{B}, \Pi', \Sigma)$
19 $\quad \Pi' \coloneqq \Pi' \cup \{H \leftarrow \mathbf{B}_{new}\}$
20 **return** $\langle \Pi', \Sigma \rangle$

Example 5. Consider the set of literals $\mathbf{L}$ of Example 4, and a rule $\rho = h(x, w) \leftarrow \bigwedge \mathbf{L}$ that uses this body. Processing this rule in the loop L9, we first initialise $\mathbf{B}_{new} = p(x, y) \wedge g(v, w) \wedge \neg r(x, v) \wedge \neg a(w)$ (L10). Next, in L11, we consider each non-local extended component $\mathsf{ec}_{\mathbf{L}}(\mathbf{C}_3)$ and $\mathsf{ec}_{\mathbf{L}}(\mathbf{C}_4)$ (as given in Example 4).

For $\mathsf{ec}_{\mathbf{L}}(\mathbf{C}_3)$, we get $V = \{x, y, v\}$ (L12) and extend $\mathbf{B}_{new}$ by $q_1(x, y, v)$ (L13). Function `sparqlImport` initialises $\mathbf{B}_{import} = Q_1@e$ (L1) for the query Q_1:

```
SELECT ?x ?y ?v
WHERE { ?y <p1> ?z . ?z <p2> ?v . FILTER NOT EXISTS{ ?x <p3> ?y } }
```

where we use notation of Definition 5. We now iterate over $\mathsf{LOC}_{\mathbf{L}}(\mathbf{C}_3) = \{\mathbf{C}_1, \mathbf{C}_2\}$ (L2). Lines L3–L5 then produce rules $b_1(x, y) \leftarrow p(x, y)$ and $b_2(v) \leftarrow g(v, w) \wedge \neg a(w)$, and the SPARQL import rule $q_1(x, y, v) \leftarrow Q_1@e \wedge b_1(x, y) \wedge b_2(v)$ (L6).

Continuing with the loop L11 for $\mathsf{ec}_{\mathbf{L}}(\mathbf{C}_4)$, we similarly obtain a query $Q_2 =$ `SELECT ?v ?w WHERE { ?v <p4> ?w . }`, binding rule $b_3(v, w) \leftarrow g(v, w) \wedge \neg a(w)$, and import rule $q_2(v, w) \leftarrow Q_2@f \wedge b_3(v, w)$. At this point, $\mathbf{B}_{new} = p(x, y) \wedge g(v, w) \wedge \neg r(x, v) \wedge \neg a(w) \wedge q_1(x, y, v) \wedge q_2(v, w)$.

We continue with the loop L15 over the one literal $\neg A = \neg s(x, p_5, w)$, which we represent by fresh atom $\neg q_3(x, w)$ in $\mathbf{B}_{new}$ (L17). Algorithm 1 produces query $Q_3 =$ `SELECT ?x ?w WHERE { ?x <p5> ?w . }`, binding rules $b_4(x) \leftarrow p(x, y)$ and $b_5(w) \leftarrow g(v, w) \wedge \neg a(w)$, and import rule $q_3(x, w) \leftarrow Q_3@f \wedge b_4(x) \wedge b_5(w)$.

The final rewritten rule added in L19 then is $h(x, w) \leftarrow p(x, y) \wedge g(v, w) \wedge \neg r(x, v) \wedge \neg a(w) \wedge q_1(x, y, v) \wedge q_2(v, w) \wedge \neg q_3(x, w)$.

The following correctness result relies on the observation that our algorithms essentially implement a special variant of the *folding* transformation in logic programming. More details are found in the appendix.

Theorem 2. *For every Datalog program Π and input dataset $\mathcal{D}$, and rewriting $\langle \Pi', \Sigma \rangle = \mathtt{extractSparql}(\Pi)$ as per Algorithm 2,* $\mathsf{out}(\Pi, \mathcal{D}) = \mathsf{out}(\Pi', \Sigma, \mathcal{D})$.

Example 6. The separation of bindings into components in Algorithm 1 offers significant performance benefits if patterns in rule bodies are only connected by RDF import predicates, as in the following rule:

$$h(x, y) \leftarrow p(x) \wedge \mathsf{triple}(x, \mathsf{prop}, y) \wedge q(y) \tag{11}$$

Now suppose that we can infer 1000 facts each for p and q. Then Algorithm 2 produces a SPARQL import rule with two binding atoms, corresponding to $p(x)$ and $q(y)$, and the SPARQL query issued according to Definition 4 contains 2000 bindings in two `VALUES` clauses, which works well in modern RDF databases.

In contrast, a classical magic sets transformation [1] would proceed from left to right, and only pass bindings of $p(x)$ to the query, which would be at risk of greatly reducing selectivity (as it might happen for rule (3)). Reordering body atoms to have p and q before triple in turn would result in a single binary binding with 10^6 facts. A SPARQL query of corresponding size would need to be split into around 100 individual queries to be answered in practice.

5 Pushing Constants and Projections

The SPARQL queries created in Section 4 can be much more selective than imported triple facts, so that fewer results need to be transferred. However, they can also reduce the number of terms transferred in individual results, since generated SPARQL queries only return values for variables shared with other parts of the rule (Algorithm 2 L12 and L16). In this section, we introduce optimisations that strengthen these positive effects when applied before Algorithm 2.

Example 7. Among many other topics, Wikidata also holds data about series of sports events, and its RDF view contains, e.g., the following three triples:

$$\langle \overbrace{\mathtt{wd:Q535746}}^{\text{1903 Copa del Rey}}, \overbrace{\mathtt{p:P3450}}^{\text{in series (statement)}}, a \rangle \quad \langle a, \overbrace{\mathtt{ps:P3450}}^{\text{in series (value)}}, \overbrace{\mathtt{wd:Q483794}}^{\text{Copa del Rey}} \rangle \quad \langle a, \overbrace{\mathtt{pq:P156}}^{\text{followed by}}, \overbrace{\mathtt{wd:Q1130640}}^{\text{1904 Copa del Rey}} \rangle$$

which use Wikidata IRIs [9] and an auxiliary *statement* node a (a long IRI that we omit here) to encode that the *1903 Copa del Rey* was part of the series *Copa del Rey*, where it was followed by the 1904 edition. It is natural to import such meaningful structures into predicates, including the auxiliary node (which might have further properties), where we use wt for imported Wikidata triples:

$$\mathsf{event}(e, s, n, a) \leftarrow \mathsf{wt}(e, \mathtt{p:P3450}, a) \wedge \mathsf{wt}(a, \mathtt{ps:P3450}, s) \wedge \mathsf{wt}(a, \mathtt{pq:P156}, n) \tag{12}$$

A program that recursively retrieves all events that followed (directly or indirectly) after the 1903 event within that series might then be as follows:

$$\mathsf{evAfter}(e, f) \leftarrow \mathsf{event}(e, \mathtt{wd:Q483794}, f, a) \tag{13}$$

$$\mathsf{evAfter}(e, g) \leftarrow \mathsf{evAfter}(e, f) \wedge \mathsf{event}(f, \mathtt{wd:Q483794}, g, a) \tag{14}$$

$$\mathsf{out}(e) \leftarrow \mathsf{evAfter}(\mathtt{wd:Q535746}, e) \tag{15}$$

Algorithm 3: Pushing constant selections in program Π

for $p \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{in}}$ *and* $i \in \{1, \ldots, \mathsf{ar}(p)\}$ **do**
  if $p \in \mathbf{P}_{\mathsf{out}}$ **then** $\mathsf{flt}(p, i) \coloneqq \mathbf{C}$ **else** $\mathsf{flt}(p, i) \coloneqq \emptyset$
repeat
  for *all active* $\rho \in \Pi$, $b(\boldsymbol{s}) \in \mathsf{body}^+(\rho)$ *with* $b \notin \mathbf{P}_{\mathsf{in}}$, *and* $1 \leq i \leq \mathsf{ar}(b)$ **do**
    $\mathsf{flt}(b, i) \coloneqq \mathsf{flt}(b, i) \uplus \mathsf{hflt}(s_i, \mathsf{head}(\rho))$
until *all values of* flt *remain unchanged*
$\Pi \coloneqq \{\rho \in \Pi \mid \rho \text{ active under } \mathsf{flt}\}$
for *all* $\rho \in \Pi$ *and* $x \in \mathsf{var}(\mathsf{head}(\rho))$ *such that* $\mathsf{hflt}(x, \mathsf{head}(\rho)) = \{c\}$ **do**
  replace all occurrences of x in ρ by c
for *all* $\rho \in \Pi$, $b(\boldsymbol{s}) \in \mathsf{body}^+(\rho)$ *with* $b \notin \mathbf{P}_{\mathsf{in}}$, *and* $1 \leq i \leq \mathsf{ar}(b)$ **do**
  if $\mathsf{flt}(b, i) = \{s_i\}$ **then** replace s_i in $b(\boldsymbol{s})$ by a fresh variable z

Algorithm 2 turns (12) into an unselective SPARQL query without binding predicates, with a result of more than $250K$ RDF terms and 14MB (TSV format; Nov 2025).

To do better, we incorporate *static filtering*, a 40-year-old logic program optimisation that was mostly forgotten in recent decades [16,17,18], but that has recently been rediscovered and generalised [10]. The approach uses goal-oriented procedures to "push" selections and projections to earlier phases of the recursive computation. Selections and projections are handled in separate algorithms. In both cases, we store information about single *predicate positions*, which we denote as pairs $\langle p, i\rangle$ for $p \in \mathbf{P}$ and $1 \leq i \leq \mathsf{ar}(p)$.

For selection pushing, we focus on equality with constants, which can be beneficially incorporated into SPARQL. We consider the set of possible selection values $S = \{\emptyset, \mathbf{C}\} \cup \{\{c\} \mid c \in \mathbf{C}\}$, and a function flt from predicate positions to S. Intuitively, $\mathsf{flt}(p, i)$ indicates the set of values that might be relevant at this position, where we overapproximate sets of more than one value by $\mathbf{C}$ since only single values are relevant for us. For elements $s_1, s_2 \in S$, let $s_1 \uplus s_2$ be the smallest set $s \in S$ such that $s_1 \cup s_2 \subseteq s$, in particular $\{c\} \uplus \emptyset = \{c\}$ and $\{c\} \uplus \{d\} = \mathbf{C}$ for $c \neq d$. For atom $h(\boldsymbol{t})$, variable x, and constant c, let $\mathsf{hflt}(x, h(\boldsymbol{t})) = \bigcap_{1 \leq i \leq \mathsf{ar}(h); t_i = x} \mathsf{flt}(h, i)$ (which defaults to $\mathbf{C}$ if there is no $t_i = x$) and let $\mathsf{hflt}(c, h(\boldsymbol{t})) = \{c\}$. For example, if $\mathsf{flt}(h, 1) = \{c\}$, $\mathsf{flt}(h, 2) = \mathbf{C}$, and $\mathsf{flt}(h, 3) = \{d\}$ $(c \neq d)$, then $\mathsf{hflt}(x, h(x, x, y)) = \{c\}$ and $\mathsf{hflt}(x, h(x, y, x)) = \emptyset$. A rule $h(\boldsymbol{t}) \leftarrow B$ is *active* under flt if $\mathsf{flt}(h, j) \cap \mathsf{hflt}(t_j, h(\boldsymbol{t})) \neq \emptyset$ for all $1 \leq j \leq \mathsf{ar}(h)$.

Algorithm 3 shows our selection pushing procedure, which we next explain with an example. Our approach and notation are direct adaptations of the general case [10]. Note that we only push selections for non-input predicates, so we can generally restrict to non-negated body atoms here.

Example 8. Consider the program of Example 7. Line 22 initialises $\mathsf{flt}(\mathsf{out}, 1) = \mathbf{C}$ and $\mathsf{flt}(p, i) = \emptyset$ for all positions $\langle p, i\rangle$ of event and $\mathsf{evAfter}$. Therefore, only rule (15) is active in the first iteration of L24. Looping over its body positions,

we get $\mathsf{flt}(\mathsf{evAfter}, 1) = \{\texttt{wd:Q535746}\}$ and $\mathsf{flt}(\mathsf{evAfter}, 2) = \mathbf{C}$ (L25). This makes (13) and (14) active.

Repeating the process (L23), we first consider (13). Loop L24 now yields $\mathsf{flt}(\mathsf{event}, 1) = \{\texttt{wd:Q535746}\}$, $\mathsf{flt}(\mathsf{event}, 2) = \{\texttt{wd:Q483794}\}$, and $\mathsf{flt}(\mathsf{event}, 3) = \mathsf{flt}(\mathsf{event}, 4) = \mathbf{C}$. Continuing with rule (14) yields $\mathsf{flt}(\mathsf{evAfter}, 1) = \{\texttt{wd:Q535746}\}$ and $\mathsf{flt}(\mathsf{evAfter}, 2) = \mathbf{C}$, i.e., the same values as before. For event, we update $\mathsf{flt}(\mathsf{event}, 1)$ to $\mathbf{C}$, since $\mathsf{hflt}(f, \mathsf{evAfter}(e, g)) = \mathbf{C}$; values for all other positions of event remain unchanged. Now rule (12) is active, but this will not affect flt since its body contains only input predicates.

No further updates occur, so we continue in L27, which does not change Π since all rules are active. In L28, we replace s in (12) by $\texttt{wd:Q483794}$ and e in (13) and (14) by $\texttt{wd:Q535746}$. Finally, in L30, we replace $\texttt{wd:Q483794}$ in the bodies of (13) and (14), as well as $\texttt{wd:Q535746}$ in the body of (15) by fresh variables, which eliminates the equality check, since we know that those constants must be derived at the respective positions. The final rules are:

$$\begin{aligned} \mathsf{event}(e, \texttt{wd:Q483794}, n, a) \leftarrow\ & \mathsf{wt}(e, \texttt{p:P3450}, a) \wedge \\ & \mathsf{wt}(a, \texttt{ps:P3450}, \texttt{wd:Q483794}) \wedge \mathsf{wt}(a, \texttt{pq:P156}, n) \end{aligned} \tag{16}$$

$$\mathsf{evAfter}(\texttt{wd:Q535746}, f) \leftarrow \mathsf{event}(\texttt{wd:Q535746}, z, f, a) \tag{17}$$

$$\mathsf{evAfter}(\texttt{wd:Q535746}, g) \leftarrow \mathsf{evAfter}(z, f) \wedge \mathsf{event}(f, z', g, a) \tag{18}$$

$$\mathsf{out}(e) \leftarrow \mathsf{evAfter}(z, e) \tag{19}$$

Theorem 3. *For any Π with input dataset $\mathcal{D}$, and Π' the result of applying Algorithm 3 to Π, $\mathsf{out}(\Pi, \mathcal{D}) = \mathsf{out}(\Pi', \mathcal{D})$.*

Proof (sketch). Let $\mathcal{M}$ and $\mathcal{M}'$ be the output of Π and Π', respectively, over $\mathcal{D}$. The proof in the appendix proceeds by showing that $\mathcal{M}'$ is the subset of $\mathcal{M}$ that consists of all facts $p(\boldsymbol{t})$ for which $t_i \in \mathsf{flt}(p, i)$, for all $i \in \{1, \ldots, \mathsf{ar}(p)\}$. This shows the result, since output predicates are not filtered (L22). The proof then works by induction over the derivation sequences for Π and Π', carefully showing analoguous applicability of corresponding rules. □

Especially after pushing selections, programs often contain predicate parameters that are entirely unnecessary in the sense that their values are neither part of the output nor relevant to the applicability of any rule. To make this formal, we consider a set R of *relevant positions*. Initially, R is set to contain all positions of all output predicates. Now for all $\rho \in \Pi$, all $b(\boldsymbol{s}) \in \mathsf{body}^+(\rho)$ with $b \notin \mathbf{P}_{\mathsf{in}}$, and all $i \in \{1, \ldots, \mathsf{ar}(b)\}$, we add $\langle b, i\rangle$ to R if: (R1) s_i is a constant, or (R2) s_i is a variable that occurs more than once in $\mathsf{body}(\rho)$, or (R3) s_i is a variable that occurs in $\mathsf{head}(\rho)$ at a position $\langle h, j\rangle \in R$. We iterate this process until R is stable. Thereafter, all terms at predicate positions that are not in R are deleted from Π, and the arity of affected predicates is reduced accordingly.

Example 9. We continue from Example 8. Initially, $R = \{\langle \mathsf{out}, 1\rangle\}$. Iteratively, we then add $\langle \mathsf{event}, 1\rangle$ ((17)+(R1) or (18)+(R2)), $\langle \mathsf{evAfter}, 2\rangle$ ((18)+(R2) or

(19)+(R3)), and $\langle \mathsf{event}, 3\rangle$ ((17)+(R3)). Removing all other positions, we get:

$$\begin{aligned} \mathsf{event}(e, n) \leftarrow\ & \mathsf{wt}(e, \texttt{p:P3450}, a) \wedge \\ & \mathsf{wt}(a, \texttt{ps:P3450}, \texttt{wd:Q483794}) \wedge \mathsf{wt}(a, \texttt{pq:P156}, n) \end{aligned} \tag{20}$$

$$\mathsf{evAfter}(f) \leftarrow \mathsf{event}(\texttt{wd:Q535746}, f) \tag{21}$$

$$\mathsf{evAfter}(g) \leftarrow \mathsf{evAfter}(f) \wedge \mathsf{event}(f, g) \tag{22}$$

$$\mathsf{out}(e) \leftarrow \mathsf{evAfter}(e) \tag{23}$$

It is easy to verify that this computes the same out-facts as Example 7, but with significant simplifications. In particular, the SPARQL import rule for (20) omits two result columns of (12): s is fixed to a constant `wd:Q483794` and a is projected away. The result of the new query contains 246 RDF terms and is 11KB in size, i.e., three orders of magnitude smaller than before the optimisation.

Theorem 4. *For any Π with input dataset $\mathcal{D}$, and Π' the result of removing irrelevant predicate positions from Π, $\mathsf{out}(\Pi, \mathcal{D}) = \mathsf{out}(\Pi', \mathcal{D})$.*

The corresponding proof in the appendix is similar to the proof of Theorem 3, showing a correspondence between respective derivation sequences.

6 Empirical Evaluation

For evaluation, we have implemented semi-naive SPARQL imports (Section 3) and optimised SPARQL rule compilation (Section 4). These features have been integrated into *Nemo* [14], and are available since version 0.10.0.[11] Triple imports are represented in Nemo by using the SPARQL import directive with the query `SELECT ?s ?p ?o WHERE { ?s ?p ?o }`. To work with multi-lingual labels, we extend Definition 5 to include Nemo built-ins `LANG(?label) = "en"` into SPARQL queries (similar extensions are conceivable for other SPARQL filters). All code specific to our evaluation as well as the rule sets and individual runtimes are available in our repository.[12] We compare overall performance with Nemo v0.8.0 (last version with unoptimised SPARQL imports) and VLog v1.3.6 (via the Rulewerk library [6]). At the time of writing, RDFox did not support querying remote endpoints in Datalog reasoning, and was therefore not evaluated. We further study the relative impact of individual optimisations in our implementation.

Evaluation data We consider six programs that all use RDF triple imports from Wikidata and some further features, as shown in Table 1. Programs (anc), (win) and (dsj) are adapted from the Nemo example collection,[13] whereas (inv) and (non) are direct implementations of checks for Wikidata quality constraints. Finally, (oa) federates Wikidata with DBLP to find out which Open Access

[11] https://github.com/knowsys/nemo/releases/tag/v0.10.0, March 2026
[12] https://github.com/knowsys/nemo-examples/tree/main/evaluations/eswc2026
[13] https://www.wikidata.org/wiki/User:Markus_Kr%C3%B6tzsch/Nemo_examples

Table 1. Evaluation test cases; *all* means all of Wikidata or DBLP would be imported

	Test case	Extra features	#rules	#inferred facts	#imported triples	#imported results
anc	Common Ancestors	¬	7	1,946,133	*all*	1,964,815
win	Winning Streaks	¬, count	11	660	> 93M	713
dsj	Disjoint Classes	¬	16	172,226	*all*	176,328
inv	Inverse constraint	¬ (input)	2	6,021	*all*	6,021
non	None-of constraint	–	2	171,582	*all*	172,486
oa	Open Access	federation	5	26,786	*all*	54,107

journals researchers based in Oxford have published in. All programs already include the optimisations presented Section 5 (so no additional transformation is required). Table 1 also gives the number of Datalog inferences for user-defined predicates, the number of triples that would need to be imported if we were to load individual triple patterns, and the number of SPARQL query results that we actually load in the optimised version. For the imported triples, *all* denotes that all of Wikidata or DBLP would have been imported; at the time of writing, this totals approximately 8.69 billion[14] and 825 million triples, respectively. All data is based on the official Wikidata SPARQL Query Service (WDQS) and QLever DBLP service (only for (oa)).[15] For VLog/Rulewerk, some programs were simplified to adjust for missing features (counting aggregations were dropped, inferring only the facts that would have been aggregated; missing data-related filters were manually pushed into SPARQL queries).

Experimental setup We ran the experiments on a dedicated server (2×quad-core Intel Xeon E5-2637 v4@3.5 GHz, 768 GiB RAM, NixOS 25.05), but with a memory limit of 16GiB. We report average times taken across ten runs; the timeout was set to 10 min. To avoid server-side caching effects, our implementation was modified to include trivial syntactic modifications in each query. We did not do this for Nemo v0.8.0, which always times out, nor for VLog/Rulewerk, where no warm-up effects were observed (presumably since VLog uses POST requests, which are not HTTP-cached).

Performance comparison Figure 1 shows the median runtimes of successful runs of Nemo v0.8.0, VLog/Rulewerk, and our new implementation for Nemo v0.10.0 on the rule sets from Table 1. The error bars indicate the minimum and maximum runtime. Shaded bars mark experiments where all ten runs failed (due to server timeout or server error). Nemo v0.8.0 has no successful run, since its unfiltered triple import fails in all cases, which is unsurprising given the huge numbers of triples that would need to be imported. VLog in turn seems to use some restrictions on queries,[16] and only fails for (oa). We observe that

[14] For performance reasons, triples relating to scholarly articles have been split into a separate SPARQL endpoint: https://wikidata.org/wiki/Wikidata:SPARQL_query_service/WDQS_graph_split

[15] https://query.wikidata.org and https://qlever.dev/dblp/

[16] We couldn't find this published, but Carral et al. mention "on-demand" imports [6].

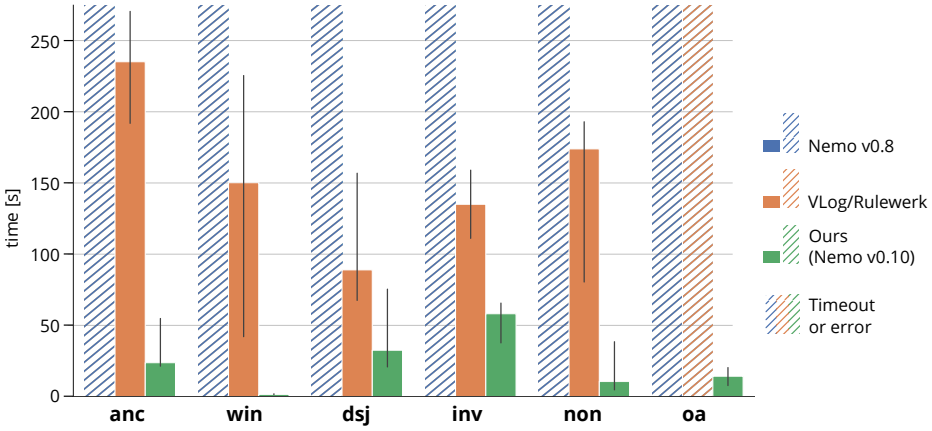

Figure 1. Performance of Nemo v0.8.0, VLog/Rulewerk, and our implementation (Nemo v0.10.0)

our implementation is consistently faster than VLog/Rulewerk, with at least one order of magnitude improvements in the median runtimes for (anc), (win), and (non). We note that the variance in query times between runs is quite high (individual runs can differ by a factor of up to 5.4). While variance is expected when conducting experiments on a public production server, the effect seems consistently more pronounced for VLog/Rulewerk than for Nemo. In spite of the variation, we obtain rather clear results: the maximum runtime of our implementation is always below the minimum runtime of VLog/Rulewerk, except for the case of a single outlier in (dsj).

Ablation study Figure 2 shows a detailed comparison of our implementation with variants that disable some optimisations, using the following abbreviations: (SN) semi-naive SPARQL evaluation for any binding pattern; (EC) multiple binding atoms based on extended components (Algorithm 1) vs. (no EC) single binding pattern that joins all local conditions; (MT) triple patterns merged into queries based on extended components vs. (no MT) triple patterns used as individual queries. Therefore (SN-EC-MT) corresponds to our maximal optimisation, and (none) is the case where we merely import triples as stated. As before, shaded bars indicate that all runs failed. Here, we divide the total runtime into the time taken for answering SPARQL queries (blue) and the time taken for Datalog reasoning (orange). Further, we show each individual run as a dot, to give a better picture of the variance in runtimes.

As we can see, each aspect of our optimisations can make a positive contribution at least in some cases, whereas the overhead of enabling them is generally not significant. Reasoning on the rule sets (dsj), (inv) and (non) times out when (MT) is disabled, but completes successfully once it is enabled. For (win), disabling (MT) does not lead to a timeout, but execution is significantly slower without this optimisation. The execution time of those rule sets is not further

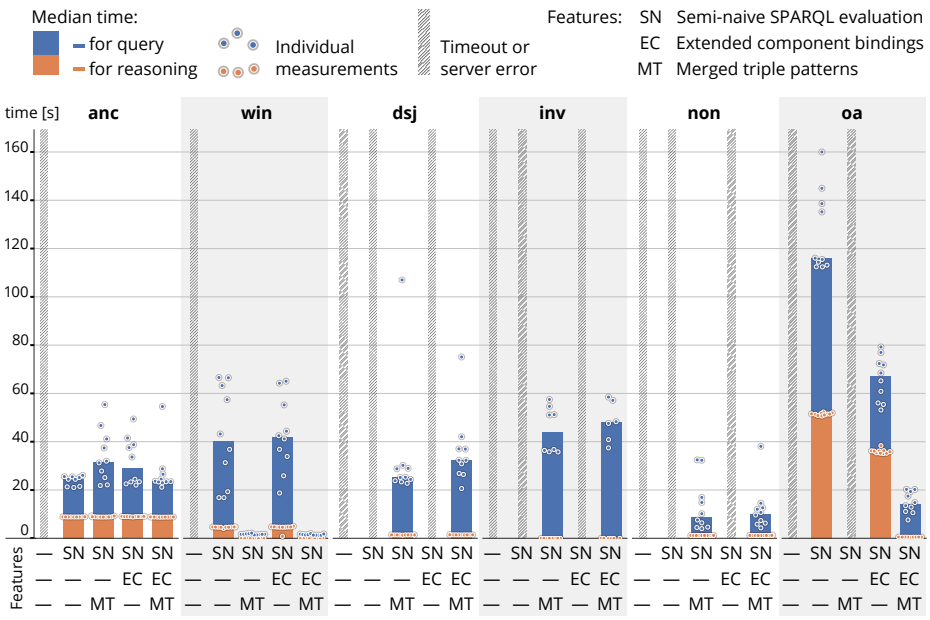

Figure 2. Runtime with different feature combinations

improved by enabling (EC), since no Cartesian products of local conditions occur in these cases. When evaluating (oa), we observe that both optimisations (EC) and (MT) individually and in combination improve the runtime. In the case of (anc), only (SN) had an impact.

7 Conclusions & Outlook

We have introduced a new approach to access large RDF-based knowledge graphs during rule reasoning, in a transparent fashion that does not require users to write SPARQL queries. Such self-optimising data access has advantages beyond the performance of a single case: it allows users to design reusable rule sets and program "modules" that are declarative and conceptually meaningful, rather than manually optimising the overall collection of rules and SPARQL queries for each specific scenario.

Many further research directions are possible along this path. The idea of Datalog-based federation of SPARQL sources has not been our focus, and many techniques from SPARQL federation could be explored in this context. Conversely, there are also further logic programming methods that could be beneficial, such as folding/unfolding of non-recursive rules. Indeed, recent works on static filtering discovered tractable methods for pushing much more general types of filter expressions in logic programs [10], which appears to be very promising in combination with our approach.

Acknowledgments. This work is supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) in project number 389792660 (TRR 248, Center for Perspicuous Systems), by the Bundesministerium für Forschung, Technologie und Raumfahrt (BMFTR, Federal Ministry of Research, Technology and Space) in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), and in DAAD project 57616814 (SECAI, School of Embedded Composite AI) as part of the program Konrad Zuse Schools of Excellence in Artificial Intelligence.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data availability statement. All source code, data and results are available at https://github.com/knowsys/nemo-examples/tree/main/evaluations/eswc2026.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1994)
2. Bellomarini, L., Sallinger, E., Gottlob, G.: The Vadalog system: Datalog-based reasoning for knowledge graphs. Proc. VLDB Endowment **11**(9), 975–987 (2018). https://doi.org/10.14778/3213880.3213888
3. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: Proc. 36th Symp. on Principles of Database Systems (PODS'17). pp. 37–52. ACM (2017). https://doi.org/10.1145/3034786.3034796
4. Bischoff, S., Krötzsch, M., Polleres, A., Rudolph, S.: Schema-agnostic query rewriting for SPARQL 1.1. In: Mika et al. [22], pp. 584–600. https://doi.org/10.1007/978-3-319-11964-9_37
5. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. J. Web Semant. **14**, 57–83 (2012). https://doi.org/10.1016/j.websem.2012.03.001
6. Carral, D., Dragoste, I., González, L., Jacobs, C., Krötzsch, M., Urbani, J.: VLog: A rule engine for knowledge graphs. In: Ghidini et al., C. (ed.) Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II). LNCS, vol. 11779, pp. 19–35. Springer (2019). https://doi.org/10.1007/978-3-030-30796-7_2
7. Cyganiak, R., Wood, D., Lanthaler, M. (eds.): RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (2014), available at http://www.w3.org/TR/rdf11-concepts/
8. Dachselt, R., Gerlach, L., Hanisch, P., Ivliev, A., Krötzsch, M., Marx, M., Méndez, J.: Declarative debugging for datalog with aggregation. In: Krause, A., Pimentel, J.F. (eds.) Proc. Workshops of the EDBT/ICDT 2026 Joint Conf. CEUR WS Proceedings, CEUR-WS (2026), to appear

9. Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., Vrandečić, D.: Introducing Wikidata to the linked data web. In: Mika et al. [22], pp. 50–65
10. Hanisch, P., Krötzsch, M.: Rule rewriting revisited: A fresh look at static filtering for Datalog and ASP. In: ten Cate, B., Funk, M. (eds.) Proc. 29th Int. Conf. on Database Theory (ICDT'2026). LIPIcs, vol. 365. Dagstuhl Publishing (2026). https://doi.org/10.4230/LIPIcs.ICDT.2026.5
11. Hogan, A., Pan, J.Z., Polleres, A., Decker, S.: SAOR: Template rule optimisations for distributed reasoning over 1 billion linked data triples. In: Patel-Schneider, P.F., Pan, Y., Glimm, B., Hitzler, P., Mika, P., Pan, J., Horrocks, I. (eds.) Proc. 9th Int. Semantic Web Conf. (ISWC'10). LNCS, vol. 6496, pp. 337–353. Springer (2010). https://doi.org/10.1007/978-3-642-17746-0_22
12. Horrocks, I., Patel-Schneider, P.F., Bechhofer, S., Tsarkov, D.: OWL Rules: A proposal and prototype implementation. J. of Web Semantics **3**(1), 23–40 (2005). https://doi.org/10.1016/j.websem.2005.05.003
13. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. J. of Web Semantics **3**(2–3), 79–115 (2005). https://doi.org/10.1016/j.websem.2005.06.001
14. Ivliev, A., Gerlach, L., Meusel, S., Steinberg, J., Krötzsch, M.: Nemo: Your friendly and versatile rule reasoning toolkit. In: Marquis, P., Ortiz, M., Pagnucco, M. (eds.) Proc. 21st Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'24). pp. 743–754. IJCAI Organization (2024). https://doi.org/10.24963/kr.2024/70
15. Kifer, M., Boley, H. (eds.): RIF Overview. W3C Working Group Note (22 June 2010), available at http://www.w3.org/TR/rif-overview/
16. Kifer, M., Lozinskii, E.L.: Filtering data flow in deductive databases. In: Ausiello, G., Atzeni, P. (eds.) Proc. 1st Int. Conf. on Database Theory (ICDT'86). LNCS, vol. 243, pp. 186–202. Springer (1986). https://doi.org/10.1007/3-540-17187-8_37
17. Kifer, M., Lozinskii, E.L.: Implementing logic programs as a database system. In: Proc. 3rd Int. Conf. on Data Engineering (ICDE'87). pp. 375–385. IEEE Computer Society (1987). https://doi.org/10.1109/ICDE.1987.7272403
18. Kifer, M., Lozinskii, E.L.: On compile-time query optimization in deductive databases by means of static filtering. ACM Trans. Database Syst. **15**(3), 385–426 (1990). https://doi.org/10.1145/88636.87121
19. Krötzsch, M.: Efficient rule-based inferencing for OWL EL. In: Walsh, T. (ed.) Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI'11). pp. 2668–2673. AAAI Press/IJCAI (2011). https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-444
20. Krötzsch, M.: The not-so-easy task of computing class subsumptions in OWL RL. In: Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A., Blomqvist, E. (eds.) Proc. 11th Int. Semantic Web Conf. (ISWC'12). LNCS, vol. 7649, pp. 279–294. Springer (2012). https://doi.org/10.1007/978-3-642-35176-1_18
21. Krötzsch, M.: Modern Datalog: Concepts, methods, applications. In: Artale, A., Bienvenu, M., García, Y.I., Murlak, F. (eds.) Joint Proceedings of the 20th and 21st Reasoning Web Summer Schools (RW 2024 & RW 2025). OASIcs, vol. 138. Dagstuhl Publishing (2025). https://doi.org/10.4230/OASIcs.RW.2024/2025.7
22. Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C.A., Vrandečić, D., Groth, P.T., Noy, N.F., Janowicz, K., Goble, C.A. (eds.): Proc. 13th Int. Semantic Web Conf. (ISWC'14), LNCS, vol. 8796. Springer (2014)

23. Mugnier, M., Thomazo, M.: An introduction to ontology-based query answering with existential rules. In: Koubarakis, M., Stamou, G.B., Stoilos, G., Horrocks, I., Kolaitis, P.G., Lausen, G., Weikum, G. (eds.) Proc. 10th Int. Reasoning Web Summer School (RW'14). LNCS, vol. 8714, pp. 245–278. Springer (2014). https://doi.org/10.1007/978-3-319-10587-1_6
24. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: A highly-scalable RDF store. In: et al., M.A. (ed.) Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II. LNCS, vol. 9367, pp. 3–20. Springer (2015). https://doi.org/10.1007/978-3-319-25010-6_1
25. Piro, R., Nenov, Y., Motik, B., Horrocks, I., Hendler, P., Kimberly, S., Rossman, M.: Semantic technologies for data analysis in health care. In: Groth, P.T., Simperl, E., Gray, A.J.G., Sabou, M., Krötzsch, M., Lécué, F., Flöck, F., Gil, Y. (eds.) Proc. 15th Int. Semantic Web Conf. (ISWC'16, Part II). LNCS, vol. 9982, pp. 400–417 (2016). https://doi.org/10.1007/978-3-319-46547-0_34
26. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization techniques for federated query processing on linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) Proc. 10th Int. Semantic Web Conf. (ISWC'11). LNCS, vol. 7032, pp. 601–616. Springer (2011). https://doi.org/10.1007/978-3-642-25073-6_38
27. Skvortsov, E.S., Xia, Y., Bowers, S., Ludäscher, B.: Logica-TGD: Transforming graph databases logically. In: Boehm, M., Daudjee, K. (eds.) Proc. Workshops of the EDBT/ICDT 2025 Joint Conf. CEUR WS Proceedings, vol. 3946. CEUR-WS (2025), https://ceur-ws.org/Vol-3946/TGD-4.pdf
28. Urbani, J., Jacobs, C., Krötzsch, M.: Column-oriented Datalog materialization for large knowledge graphs. In: Schuurmans, D., Wellman, M.P. (eds.) Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16). pp. 258–264. AAAI Press (2016). https://doi.org/10.1609/aaai.v30i1.9993
29. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: WebPIE: A Web-scale parallel inference engine using MapReduce. J. of Web Semantics **10**, 59–75 (2012). https://doi.org/10.1016/j.websem.2011.05.004
30. Van Woensel, W., Arndt, D., Champin, P.A., Tomaszuk, D., Kellogg, G. (eds.): Notation3 Language. W3C Draft Community Group Report (15 May 2024), available at https://w3c.github.io/N3/spec/

A Proofs for Section 4

Theorem 2. *For every Datalog program Π and input dataset $\mathcal{D}$, and rewriting $\langle \Pi', \Sigma \rangle = \texttt{extractSparql}(\Pi)$ as per Algorithm 2, $\mathsf{out}(\Pi, \mathcal{D}) = \mathsf{out}(\Pi', \Sigma, \mathcal{D})$.*

This result follows from several simple observations. The fundamental methods we use is the following *folding* operation.

Definition 7. *Consider a Datalog program Π and rule $\rho \in \Pi$ with $\mathsf{body}(\rho) = \mathbf{A} \wedge \mathbf{B} \wedge \mathbf{C}$ for arbitrary conjunctions of literals $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, such that all variables in negated atoms of $\mathbf{B} \wedge \mathbf{C}$ also occur in positive atoms of $\mathbf{B} \wedge \mathbf{C}$.*

The program $\Pi[\rho, \mathbf{B}, \mathbf{C}]$ is obtained from Π by replacing ρ with new rules

$$f[V] \leftarrow \mathbf{B} \wedge \mathbf{C} \tag{24}$$

$$\mathsf{head}(\rho) \leftarrow \mathbf{A} \wedge \mathbf{B} \wedge f[V] \tag{25}$$

where $V = \mathsf{var}(\mathbf{C}) \cap \mathsf{var}(\mathbf{A} \cup \mathbf{B} \cup \{\mathsf{head}(\rho)\})$ and f a fresh predicate of arity $|V|$.

If $\mathbf{B}$ is empty, this is standard folding. Adding $\mathbf{B}$ to rule (24) makes this rule more selective, but clearly without removing any inferences needed in (25).

Lemma 1. *For every dataset $\mathcal{D}$ for Π, and every $\Pi[\rho, \mathbf{B}, \mathbf{C}]$ as in Definition 7, $\mathsf{out}(\Pi, \mathcal{D}) = \mathsf{out}(\Pi[\rho, \mathbf{B}, \mathbf{C}], \mathcal{D})$.*

SPARQL import rules semantically correspond to normal rules if we replace the special SPARQL query atom by an import atom that represents the query result. With this view, we see that the replacement in Algorithm 2 L11–L14 corresponds to the transformation in Definition 7, with the extended components $\mathsf{ec}_{\mathbf{B}}(\mathbf{D})$ in Algorithm 1 playing the role of $\mathbf{B}$. The rules added in L3–L5 correspond to another use of Definition 7. Theorem 2 then follows from Lemma 1.

Note that variables in `FILTER NOT EXISTS` in the generated SPARQL queries are bound in positive query patterns if imports are evaluated as in Definition 4.

B Proofs for Section 5

Theorem 3. *For any Π with input dataset $\mathcal{D}$, and Π' the result of applying Algorithm 3 to Π, $\mathsf{out}(\Pi, \mathcal{D}) = \mathsf{out}(\Pi', \mathcal{D})$.*

Proof. Let flt be the mapping computed in Algorithm 3, and let $\mathcal{M}$ and $\mathcal{M}'$ be the set of consequences of Π and Π', respectively, over $\mathcal{D}$. For predicates $p \in \mathbf{P}_{\mathsf{in}}$ and $i \in \{1, \ldots, \mathsf{ar}(p)\}$, let $\mathsf{flt}(p, i) = \mathbf{C}$, and define $\mathsf{flt}(\mathcal{M}) = \{p(\boldsymbol{t}) \in \mathcal{M} \mid \text{for all } i \in \{1, \ldots, \mathsf{ar}(p)\} : t_i \in \mathsf{flt}(p, i)\}$ as the restriction of $\mathcal{M}$ to facts covered by flt. We claim that $\mathcal{M}' = \mathsf{flt}(\mathcal{M})$. This shows the theorem, since facts for predicates in $\mathbf{P}_{\mathsf{out}}$ are the same in $\mathcal{M}$ and $\mathsf{flt}(\mathcal{M})$ by L22.

We show the claim by induction on the step-wise computation of $\mathcal{D}^\infty$, where we write $\mathcal{D}^i$ and $\mathcal{D}'^i$ for the respective sequences for Π and Π'. Initially, $\mathcal{D}'^0 = \mathsf{flt}(\mathcal{D}^0) = \mathcal{D}$, since inputs are not filtered. For the induction step, assume that

$\mathcal{D}'^i = \mathsf{flt}(\mathcal{D}^i)$ for some $i \geq 0$ (IH). To show $\mathcal{M}' \supseteq \mathsf{flt}(\mathcal{M})$, consider a rule $\rho \in \Pi$ and substitution σ such that $\mathsf{body}^+(\rho)\sigma \subseteq \mathcal{D}^i$, $\mathsf{body}^-(\rho)\sigma \cap \mathcal{D}^i = \emptyset$, and $\mathsf{head}(\rho)\sigma \in \mathsf{flt}(\mathcal{M})$. Then ρ is active, and therefore (passing L27) has a corresponding rewritten rule in $\rho' \in \Pi'$. We extend σ to σ' that is defined on fresh variables z in ρ' as introduced in L31 for a position $\langle b, i\rangle$ by letting $\sigma'(z) = c$ for the c in $\mathsf{flt}(b, i) = \{c\}$. We claim that σ' lets us apply ρ' to $\mathcal{D}'^i$ to derive $\mathsf{head}(\rho)\sigma$.

- $\mathsf{body}^+(\rho')\sigma' \subseteq \mathcal{D}'^i$. Consider any $b(\boldsymbol{s}) \in \mathsf{body}^+(\rho)$, corresponding $b(\boldsymbol{s'}) \in \mathsf{body}^+(\rho')$, and $i \in \{1, \ldots, \mathsf{ar}(b)\}$. If $s_i \in \mathbf{C}$, then $\{c\} \subseteq \mathsf{flt}(b, i)$ by L25 and our definition of $\mathsf{hflt}(c, \mathsf{head}(\rho))$. Hence $s'_i\sigma' = c \in \mathsf{flt}(b, i)$ (if s'_i is a variable due to L31, this follows from how we defined σ'). If $s_i \in \mathbf{V}$, then $\mathsf{flt}(b, i) \supseteq \mathsf{hflt}(s_i, \mathsf{head}(\rho))$ (by L25 and since ρ is active). Since $\mathsf{head}(\rho)\sigma \in \mathsf{flt}(\mathcal{M})$ and by our definition of hflt, $s_i\sigma \in \mathsf{flt}(b, i)$, and therefore $s'_i\sigma' \in \mathsf{flt}(b, i)$ (this also holds true if s'_i is a constant used to replace s_i in L29). Therefore, $b(\boldsymbol{s'})\sigma' = b(\boldsymbol{s})\sigma \in \mathsf{flt}(\mathcal{D}^i)$, where the latter is $\mathcal{D}'^i$ (IH).
- $\mathsf{body}^-(\rho')\sigma' \cap \mathcal{D}'^i = \emptyset$. Using arguments as in the first case (and safety), we find that $\mathsf{body}^-(\rho')\sigma' = \mathsf{body}^-(\rho)\sigma$. The claim follows from $\mathsf{body}^-(\rho)\sigma \cap \mathcal{D}^i = \emptyset$ since $\mathcal{D}'^i \subseteq \mathcal{D}^i$ by (IH).
- $\mathsf{head}(\rho')\sigma' = \mathsf{head}(\rho)\sigma$. This is a direct consequence of $\mathsf{head}(\rho)\sigma \in \mathsf{flt}(\mathcal{M})$.

It remains to show $\mathcal{M}' \subseteq \mathsf{flt}(\mathcal{M})$. The only concern here is the relaxation at positions $\langle b, i\rangle$ in L31. However, by (IH), all facts in $\mathcal{D}'^i$ are covered by the filters, and therefore must contain the constant s_i of L31 at position $\langle b, i\rangle$. Therefore, in spite of the relaxation, rules only apply to cases that also satisfy the original rule in Π. □

Theorem 4. *For any Π with input dataset $\mathcal{D}$, and Π' the result of removing irrelevant predicate positions from Π, $\mathsf{out}(\Pi, \mathcal{D}) = \mathsf{out}(\Pi', \mathcal{D})$.*

Proof. First note that rules in Π' are safe: if a variable in a positive body is removed, then it does not occur in a negated atom (R2) nor in the head (R3). Now let $\mathcal{M}$ and $\mathcal{M}'$ be the set of consequences of Π and Π', respectively, over $\mathcal{D}$. We claim that $\mathcal{M}'$ can be obtained from $\mathcal{M}$ by deleting irrelevant positions from all non-input predicates. This shows the theorem, since predicates in $\mathbf{P}_{\mathsf{out}}$ only have relevant positions.

The claim can be shown by induction over the step-wise computation of $\mathcal{D}^\infty$, where we write $\mathcal{D}^i$ and $\mathcal{D}'^i$ for the respective sequences for Π and Π'. Initially, $\mathcal{D}^0 = \mathcal{D}'^0 = \mathcal{D}$. For the induction step, assume that $\mathcal{D}^i = \mathcal{D}'^i$ for some $i \geq 0$ (IH). Using (IH), every rule application on Π as in (5) corresponds to a rule application on Π' (using the same substitution), so $\mathcal{D}'^{i+1}$ does indeed contain all projected facts of $\mathcal{D}^{i+1}$. For the converse, consider any rule $\rho' \in \Pi'$ and substitution σ' such that $\mathsf{body}^+(\rho')\sigma' \subseteq \mathcal{D}'^i$ and $\mathsf{body}^-(\rho')\sigma' \cap \mathcal{D}'^i = \emptyset$. It is easy to extend σ' to a matching substitution σ over all variables of the original rule ρ: by the definition of R, each additional variable in ρ only occurs in one position of one positive body atom B; the projected atom B' satisfies $B'\sigma' \in \mathcal{D}'^i$, so by (IH) we find a corresponding $B\sigma \in \mathcal{D}^i$. □