SPARQLing Datalog for Rule-Based Reasoning over Large Knowledge Graphs (Extended Abstract)

Alex Ivliev[1,*], Markus Krötzsch[1] and Maximilian Marx[1]

[1]*Knowledge-Based Systems Group, TU Dresden, Germany*

Abstract

Rule languages such as Datalog are well-suited for reasoning over graph-structured data, typically represented as RDF triples. However, the largest knowledge graphs cannot be processed by existing in-memory rule engines. We have recently proposed an integration of rule engines with external RDF stores in which relevant data is fetched during reasoning through automatically generated SPARQL queries. Users write plain Datalog programs over RDF triples, without any knowledge of SPARQL, while the reasoner constructs and optimises the required queries. The optimisations adapt established techniques from logic programming, such as semi-naive evaluation, magic sets, and static filtering. In this extended abstract, we summarise our approach, implemented in the rule engine Nemo, and the empirical evaluation in which complex rule sets are executed against the public SPARQL services of Wikidata and DBLP.

Keywords

Datalog, SPARQL, Knowledge graphs, Query optimisation

1. Introduction

Datalog has seen renewed interest as a declarative language for large-scale data processing, supported by mature implementations and a growing range of applications that includes ontological reasoning [1, 2], query answering [3, 4], graph transformation [5, 6], and general data analysis [7, 8]. Knowledge graphs, commonly represented using the Resource Description Framework (RDF) [9], are a particularly good match, and several modern rule engines support RDF data natively [10, 11, 12]. However, the largest knowledge graphs, due to their scale and dynamic nature, still remain out of reach for such systems. The RDF export of Wikidata [13], for example, currently comprises more than 17.5 billion triples, with update rates above 100 triples per second.[1] Since rule engines are typically designed as in-memory systems, a local copy of this data would require main memory in the range of terabytes, and would quickly become outdated.

Dedicated RDF database systems, in contrast, are designed for data management at this scale, and make their contents available through the query language SPARQL. Many recursive computations, however, cannot be expressed in SPARQL: property paths, for instance, capture plain reachability, but cannot impose further conditions along the path. More fundamentally, SPARQL is limited to problems in NL, whereas Datalog is PTime-complete (both in data complexity). Some tasks that are possible in SPARQL in principle may still require impractically large queries [14]. 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.

In recent work [15], we have integrated Datalog reasoning with external RDF stores, automatically generating SPARQL queries to fetch relevant data during reasoning. Rules refer to RDF triples as if they were given as local facts, so no knowledge of SPARQL is needed. The optimisations that make this

Datalog 2.0 2026: 6th International Workshop on the Resurgence of Datalog in Academia and Industry, September 7, 2026, Klagenfurt, Austria

*Corresponding author.

alex.ivliev@tu-dresden.de (A. Ivliev); markus.kroetzsch@tu-dresden.de (M. Krötzsch); maximilian.marx@tu-dresden.de (M. Marx)

0000-0002-1604-6308 (A. Ivliev); 0000-0002-9172-2601 (M. Krötzsch); 0000-0003-1479-0341 (M. Marx)

[1]Source: grafana.wikimedia.org, sum of "main" and "scholarly" graphs, Jul 2026.

feasible adapt established techniques from logic programming to the new setting: semi-naive evaluation [16] restricts the data requested in each step to newly derived bindings; the generated queries are constrained by runtime bindings in the spirit of magic sets [16]; and static filtering [17, 18] propagates constants and projections into imports.

We have implemented this approach into the open source rule engine Nemo [12], which we use as the basis for an empirical evaluation. Compared to existing systems that support SPARQL data imports, we observe consistent performance improvements on complex rule sets executed on large knowledge graphs. For a more in-depth discussion, we refer the reader to the full paper, which includes proofs, detailed algorithms, and an ablation study [15].

2. Datalog over RDF

To describe our approach, we use a minimal rule language: plain Datalog with input negation, in addition to special RDF input predicates to access triples from RDF graphs. Practical extensions such as stratified negation, filter expressions, and aggregates [8] are compatible with the approach but not required in this paper.

We assume a set $\mathbf{C}$ of *constants* that includes all IRIs and RDF literals [9], and an infinite set $\mathbf{V}$ of *variables* disjoint from $\mathbf{C}$; *terms* are constants or variables. Blank nodes are not considered, since SPARQL cannot retrieve information about a specific blank node in the data, and since blank nodes in rules and queries can be replaced by variables. An *atom* is an expression $p(t_1, \ldots, t_k)$, where p is a *predicate* of arity k and $t_1, \ldots, t_k$ are terms. A *literal* is an atom A or a negated atom $\neg A$, and 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 the atom H is the *head* of ρ, denoted $\mathsf{head}(\rho)$, and the atoms $B_1, \ldots, B_\ell$ form its *body*, split into $\mathsf{body}^+(\rho) = \{B_1, \ldots, B_n\}$ and $\mathsf{body}^-(\rho) = \{B_{n+1}, \ldots, B_\ell\}$. Every variable of a rule must occur in some non-negated body atom (*safety*). A *fact* is a rule with empty body, written without $\leftarrow$.

A *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, such that input predicates do not occur in rule heads and only input predicates occur negated. A *dataset* $\mathcal{D}$ for the program is a finite set of facts over input predicates. 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.[2] 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 $\langle s, p, o \rangle \in G$.

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.

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

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

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

The semantics is defined as usual. A *substitution* σ maps finitely many variables to constants, and $\varphi\sigma$ denotes the result of applying σ to an expression φ. The consequences of Π over a set of facts $\mathcal{I}$ are $\Pi(\mathcal{I}) = \{\mathsf{head}(\rho)\sigma \mid \rho \in \Pi,\ \mathsf{body}^+(\rho)\sigma \subseteq \mathcal{I},\ \mathsf{body}^-(\rho)\sigma \cap \mathcal{I} = \emptyset\}$. Setting $\mathcal{D}^0 := \mathcal{D}$ and $\mathcal{D}^{i+1} := \mathcal{D}^i \cup \Pi(\mathcal{D}^i)$ yields a monotone sequence that reaches a fixpoint $\mathcal{D}^\infty$ after finitely many steps. The *output* $\mathsf{out}(\Pi, \mathcal{D})$ consists of all facts over output predicates in $\mathcal{D}^\infty$. In Example 1, for instance, a triple $\langle b, \mathsf{schema{:}name}, \text{"Bob"} \rangle$ leads to the output fact $\mathsf{label}(b, \text{"Bob"})$ whenever schema:name is a declared subproperty of rdfs:label.

[2] 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.

3. Generating SPARQL Queries during Reasoning

RDF import predicates allow users to write Datalog programs that use external RDF triples as if they were local facts. The most direct way of obtaining this data is to fetch the whole graph via the SPARQL query `SELECT ?s ?p ?o WHERE { ?s ?p ?o }`, but this is infeasible for graphs with billions of triples. In practice, only a small fraction of the graph is needed to evaluate a given program. We therefore produce selective SPARQL queries by matching only the triple patterns that occur in the rules, constraining them to the constants derived so far, and only fetching values relevant to the output of the program.

Incremental imports with bindings We fetch data through *SPARQL import rules* of the form

$$p(\boldsymbol{x}) \leftarrow Q@ep \wedge B_1 \wedge \ldots \wedge B_n, \tag{5}$$

where p is an input predicate, Q a SPARQL query with answer variables $\boldsymbol{x}$, *ep* an endpoint, and the *binding atoms* $B_1, \ldots, B_n$ only use constants and variables from $\boldsymbol{x}$. We write Q^{ep} for the set of answer tuples that Q returns over *ep*. Then, the SPARQL import rule derives, from the current facts $\mathcal{I}$, the set

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

That is, it keeps only the answers of Q that agree with derived facts for every binding atom. For the triple atom in rule (3), we restrict the generic query Q above through binding atoms:

$$\mathsf{spoTriple}(s, p, o) \leftarrow Q@ep \wedge p = \mathsf{rdfs{:}subPropertyOf} \wedge \mathsf{labelProp}(o). \tag{6}$$

Here, $p = \mathsf{rdfs{:}subPropertyOf}$ selects subproperty triples, and $\mathsf{labelProp}(o)$ keeps only those whose object is already known to be a label property.

To evaluate an import rule, we push the derived bindings into the query using the `VALUES` operator of SPARQL 1.1. For a binding atom $q(y_1, \ldots, y_m)$ with derived facts $q(\boldsymbol{c}^1), \ldots, q(\boldsymbol{c}^\ell)$, we extend the query with the clause

$$\texttt{VALUES}\ (y_1\ \cdots\ y_m)\ \{\ (\boldsymbol{c}^1)\ \cdots\ (\boldsymbol{c}^\ell)\ \},$$

so that the endpoint returns only the answers that are compatible with the derived facts.

Crucially, binding atoms may be defined recursively: as new labelProp facts are added, further subproperty triples become relevant. Import rules are therefore evaluated alongside ordinary rules of the program, sending updated queries as new bindings are derived.

A naive strategy that sends all current bindings in each step would still be impractical as the bindings from earlier steps would reappear in every query, leading to large requests and redundant answers. We prevent this by adapting *semi-naive* evaluation [16], which focuses each query on the newly derived bindings. For an import rule with n binding atoms, we send n queries where the j-th query restricts binding atom j to the newly derived facts, the earlier atoms to the strictly older facts, and the later atoms to all current facts.

Queries that are sent with this strategy can still become quite large. In practice, we observe that SPARQL services can handle 10^4–10^5 bindings per query. We therefore split larger binding sets over several requests.

Extracting import rules Import rules are not written by users but extracted from the program automatically. Every RDF import predicate is associated with the endpoint that provides its triples called its *origin*; all other predicates we refer to as *local*.

Within a rule body, all RDF atoms that have the same origin and are connected through shared variables are grouped and turned into a single SPARQL query as follows: positive atoms become triple patterns, negated atoms become `FILTER NOT EXISTS` patterns, and only those variables used elsewhere in the rule are selected for the query result. Groups of RDF atoms in the rule body are then replaced by a fresh import atom, defined by an import rule (5), where local body atoms that share a variable with the query become binding atoms.

Table 1
Evaluation test cases; *all* means that the complete contents of Wikidata (resp. DBLP) would be fetched

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

As an example, consider the rule

$$h(x,z) \leftarrow a(x) \wedge b(z) \wedge \mathsf{triple}(x,p_1,y) \wedge \mathsf{triple}(y,p_2,z) \wedge \neg\, \mathsf{triple}(x,p_3,z),$$

with a local predicates a and b, and RDF import predicate triple with endpoint *ep*. The two positive triples share y and form the query Q:

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

This query selects only relevant variables x and z, dropping the internal variable y, and turns the negated triple into a `FILTER NOT EXISTS` expression. The local atoms $a(x)$ and $b(z)$ become the binding atoms, resulting in the import rule $q(x,z) \leftarrow Q@ep \wedge a(x) \wedge b(z)$. The original rule is rewritten to $h(x,z) \leftarrow q(x,z)$. In order to compute the bindings, we do not evaluate the Cartesian product $a(x) \wedge b(z)$, but instead introduce two separate `VALUES` clauses, one for each binding atom.

Pushing constants and projections The extracted queries can still request more data than the program may need. As RDF frequently models n-ary relations through auxiliary nodes, such nodes and other values that are never inspected tend to end up in the imported results. Consider a relation of awards, each represented by a node a that links a recipient p and a year y, together with a rule that asks only for the recipients of the awards of one particular year c:

$$\mathsf{award}(p,y,a) \leftarrow \mathsf{triple}(a,\mathsf{recipient},p) \wedge \mathsf{triple}(a,\mathsf{year},y),$$
$$\mathsf{out}(p) \leftarrow \mathsf{award}(p,c,a).$$

As extracted, the import for award retrieves the recipient, year, and node of *every* award. We recover selectivity by preprocessing the program with *static filtering* [17, 18], which propagates constants and unused positions backwards from the output predicates. The constant c is pushed into the first rule, specialising its pattern to $\mathsf{triple}(a,\mathsf{year},c)$, so that the query returns only awards of year c. The node a is never used outside these rules and is projected away, and the year is fixed to c rather than returned, leaving a query that selects just the recipient p.

4. Evaluation

The semi-naive SPARQL imports and the optimised query extraction described above are available in the rule engine *Nemo* since version 0.10.0.[3] All rule sets, scripts, and measurements of our evaluation are available online.[4]

We evaluate the six Datalog programs shown in Table 1, all of which import triples from the public Wikidata Query Service.[5] Test case (oa) additionally accesses the QLever SPARQL endpoint for DBLP[6]

[3] https://github.com/knowsys/nemo/releases/tag/v0.10.0, March 2026
[4] https://github.com/knowsys/nemo-examples/tree/main/evaluations/eswc2026
[5] [19] https://query.wikidata.org; the "main" graph of Wikidata alone held ca. 8.69 billion triples at the time of the experiments.
[6] [20] https://qlever.dev/dblp/; ca. 825 million triples.

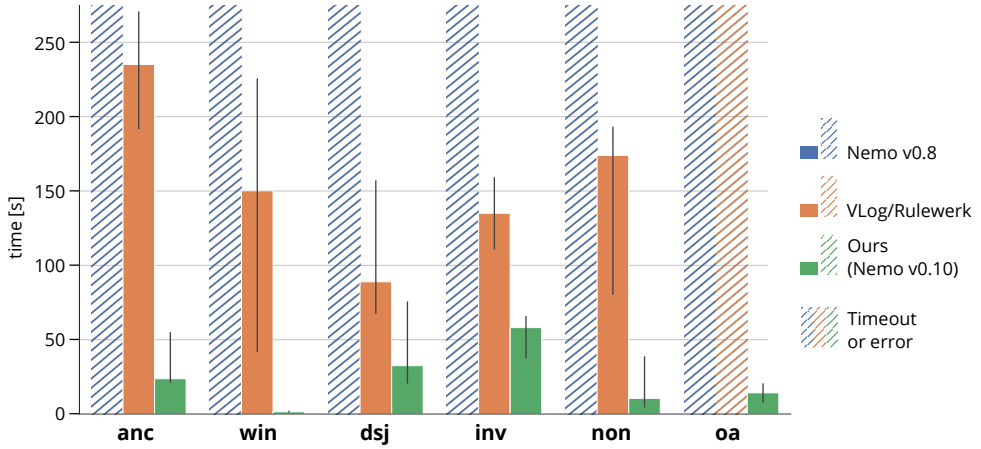

Figure 1: Median runtimes of Nemo v0.8.0, VLog/Rulewerk, and our implementation (Nemo v0.10.0); error bars span the fastest and slowest of ten runs; shaded bars indicate that all runs failed

to determine the Open Access journals in which Oxford-based researchers have published. Programs (anc), (win), and (dsj) are adapted from the Nemo example collection, whereas (inv) and (non) implement checks for Wikidata property constraints. Table 1 also contrasts the number of triples that a naive import of the individual triple patterns would fetch, which in four cases amounts to complete datasets, with the number of answers that the optimised imports transfer.

We compare with Nemo v0.8.0, the first version that introduces SPARQL imports, but without any of the discussed optimisations, and with VLog v1.3.6, used through the Rulewerk library [21]. RDFox did not support access to remote SPARQL endpoints during reasoning at the time of the experiments. Some programs were simplified for VLog to compensate for missing features. All experiments were run on a server (2× Intel Xeon E5-2637 v4, NixOS 25.05) with memory restricted to 16 GiB and a timeout of 10 min; we report median times over ten runs. To exclude server-side caching effects, every query sent by our implementation was modified by trivial syntactic changes.

Figure 1 shows the results. Nemo v0.8.0 does not complete any test case, which is expected given the size of its unrestricted imports. VLog seems to use some restrictions on the sent queries, and is therefore able to evaluate all cases except the federated test (oa) within the timeout. Our implementation completes all test cases and achieves the lowest median runtime throughout; for (anc), (win), and (non), the median improvement over VLog/Rulewerk exceeds one order of magnitude. We observe that runtimes may vary considerably between runs, by a factor of up to 5.4, with larger variation for VLog than for Nemo. Nevertheless, except for a single outlier run on (dsj), the slowest run of our implementation is faster than the fastest run of VLog/Rulewerk on every test case.

The full paper additionally contains an ablation study, which shows that each of the presented optimisations contributes to the overall performance, and that several test cases fail when optimisations are disabled [15].

5. Conclusions

We have presented a transparent way to access large RDF-based knowledge graphs during rule reasoning, without requiring users to write SPARQL queries. Our evaluation shows that the approach clearly outperforms existing systems with SPARQL imports. Beyond such performance gains, it also lets users design reusable and declarative rule sets, rather than manually tuning rules and queries for each scenario. Several directions remain open: techniques from SPARQL federation could be transferred to the multi-endpoint setting, and further logic programming methods, such as the unfolding of non-recursive rules or the more general forms of static filtering studied in recent work [18], may yield even more selective imports.

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.

Declaration on Generative AI

During the preparation of this work, the authors used Claude (Anthropic) in order to paraphrase and reword sentences. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] H. J. ter Horst, Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary, J. of Web Semantics 3 (2005) 79–115. doi:10.1016/j.websem.2005.06.001.

[2] M. Krötzsch, Efficient rule-based inferencing for OWL EL, in: T. Walsh (Ed.), Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI'11), AAAI Press/IJCAI, 2011, pp. 2668–2673. doi:10.5591/978-1-57735-516-8/IJCAI11-444.

[3] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, H. Bal, WebPIE: A Web-scale parallel inference engine using MapReduce, J. of Web Semantics 10 (2012) 59–75. doi:10.1016/j.websem.2011.05.004.

[4] A. Calì, G. Gottlob, T. Lukasiewicz, A general Datalog-based framework for tractable query answering over ontologies, J. Web Semant. 14 (2012) 57–83. doi:10.1016/j.websem.2012.03.001.

[5] E. S. Skvortsov, Y. Xia, S. Bowers, B. Ludäscher, Logica-TGD: Transforming graph databases logically, in: M. Boehm, K. Daudjee (Eds.), Proc. Workshops of the EDBT/ICDT 2025 Joint Conf., volume 3946 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2025. URL: https://ceur-ws.org/Vol-3946/TGD-4.pdf.

[6] R. Dachselt, L. Gerlach, P. Hanisch, A. Ivliev, M. Krötzsch, M. Marx, J. Méndez, Declarative debugging for Datalog with aggregation, in: A. Krause, J. F. Pimentel (Eds.), Proc. Workshops of the EDBT/ICDT 2026 Joint Conference, volume 4192 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2026. URL: https://ceur-ws.org/Vol-4192/TGD-paper4.pdf.

[7] R. Piro, Y. Nenov, B. Motik, I. Horrocks, P. Hendler, S. Kimberly, M. Rossman, Semantic technologies for data analysis in health care, in: P. T. Groth, E. Simperl, A. J. G. Gray, M. Sabou, M. Krötzsch, F. Lécué, F. Flöck, Y. Gil (Eds.), Proc. 15th Int. Semantic Web Conf. (ISWC'16, Part II), volume 9982 of *LNCS*, 2016, pp. 400–417. doi:10.1007/978-3-319-46547-0_34.

[8] M. Krötzsch, Modern Datalog: Concepts, methods, applications, in: A. Artale, M. Bienvenu, Y. I. García, F. Murlak (Eds.), Joint Proceedings of the 20th and 21st Reasoning Web Summer Schools (RW 2024 & RW 2025), volume 138 of *OASIcs*, Dagstuhl Publishing, 2025. doi:10.4230/OASIcs.RW.2024/2025.7.

[9] R. Cyganiak, D. Wood, M. Lanthaler (Eds.), RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation, 2014. Available at http://www.w3.org/TR/rdf11-concepts/.

[10] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, J. Banerjee, RDFox: A highly-scalable RDF store, in: M. A. et al. (Ed.), Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II, volume 9367 of *LNCS*, Springer, 2015, pp. 3–20. doi:10.1007/978-3-319-25010-6_1.

[11] J. Urbani, C. Jacobs, M. Krötzsch, Column-oriented Datalog materialization for large knowledge

graphs, in: D. Schuurmans, M. P. Wellman (Eds.), Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16), AAAI Press, 2016, pp. 258–264. doi:10.1609/aaai.v30i1.9993.

[12] A. Ivliev, L. Gerlach, S. Meusel, J. Steinberg, M. Krötzsch, Nemo: Your friendly and versatile rule reasoning toolkit, in: P. Marquis, M. Ortiz, M. Pagnucco (Eds.), Proc. 21st Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'24), IJCAI Organization, 2024, pp. 743–754. doi:10.24963/kr.2024/70.

[13] F. Erxleben, M. Günther, M. Krötzsch, J. Mendez, D. Vrandečić, Introducing Wikidata to the linked data web, in: [22], 2014, pp. 50–65.

[14] S. Bischoff, M. Krötzsch, A. Polleres, S. Rudolph, Schema-agnostic query rewriting for SPARQL 1.1, in: [22], 2014, pp. 584–600. doi:10.1007/978-3-319-11964-9_37.

[15] A. Ivliev, M. Krötzsch, M. Marx, SPARQLing Datalog for rule-based reasoning over large knowledge graphs, in: M. Acosta, M. van Erp, S. Rudolph, O. Hartig, B. Spahiu, A. Rula, D. Garijo, F. Osborne (Eds.), Proc. 23rd European Semantic Web Conf. , Part I (ESWC'26), volume 16549 of *LNCS*, Springer, 2026, pp. 518–536. doi:10.1007/978-3-032-25156-5_27.

[16] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison Wesley, 1994.

[17] M. Kifer, E. L. Lozinskii, Filtering data flow in deductive databases, in: G. Ausiello, P. Atzeni (Eds.), Proc. 1st Int. Conf. on Database Theory (ICDT'86), volume 243 of *LNCS*, Springer, 1986, pp. 186–202. doi:10.1007/3-540-17187-8_37.

[18] P. Hanisch, M. Krötzsch, Rule rewriting revisited: A fresh look at static filtering for Datalog and ASP, in: B. ten Cate, M. Funk (Eds.), Proc. 29th Int. Conf. on Database Theory (ICDT'2026), volume 365 of *LIPIcs*, Dagstuhl Publishing, 2026. doi:10.4230/LIPIcs.ICDT.2026.5.

[19] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, A. Bielefeldt, Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph, in: D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee, E. Simperl (Eds.), Proc. 17th Int. Semantic Web Conf. (ISWC'18), volume 11137 of *LNCS*, 2018, pp. 376–394.

[20] M. R. Ackermann, H. Bast, B. M. Beckermann, J. Kalmbach, P. Neises, S. Ollinger, The dblp knowledge graph and SPARQL endpoint, TGDK 2 (2024) 3:1–3:23. doi:10.4230/TGDK.2.2.3.

[21] D. Carral, I. Dragoste, L. González, C. Jacobs, M. Krötzsch, J. Urbani, VLog: A rule engine for knowledge graphs, in: C. Ghidini et al. (Ed.), Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II), volume 11779 of *LNCS*, Springer, 2019, pp. 19–35. doi:10.1007/978-3-030-30796-7_2.

[22] P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. A. Knoblock, D. Vrandečić, P. T. Groth, N. F. Noy, K. Janowicz, C. A. Goble (Eds.), Proc. 13th Int. Semantic Web Conf. (ISWC'14), volume 8796 of *LNCS*, Springer, 2014.