Nemo: First Glimpse of a New Rule Engine

Alex Ivliev[1] Stefan Ellmauthaler[1] Lukas Gerlach[1] Maximilian Marx[1]
Matthias Meißner Simon Meusel Markus Krötzsch[1]
Knowledge-Based Systems Group, Faculty of Computer Science / cfaed / CeTI / ScaDS.AI
TU Dresden, Germany
[1] {firstname.lastname}@tu-dresden.de

This system demonstration presents *Nemo*, a new logic programming engine with a focus on reliability and performance. Nemo is built for data-centric analytic computations, modelled in a fully declarative Datalog dialect. Its scalability for these tasks matches or exceeds that of leading Datalog systems. We demonstrate uses in reasoning with knowledge graphs and ontologies with $10^5 – 10^8$ input facts, all on a laptop. Nemo is written in Rust and available as a free and open source tool.

From the early days of logic programming, it has been clear that declarative rules can also be useful for data analysis and query answering. *Datalog* can either be viewed as the core of virtually every logic programming language, or as the generalisation of conjunctive queries with recursion [2]. This bridge between rule-based systems and databases has become even more important recently, since it fits well with the growing demand for data analytics, graph-based data management, and declarative processing.

Accordingly, there is a great number of Datalog-based rule engines, with widely different goals and features. The following overview of relevant system types is far from complete:

1. logic programming systems, esp. for ASP [8, 3] and Prolog [10],
2. knowledge graph and deductive database engines like RDFox [11], VLog [12], and Vadalog [5],
3. specialised data-analytics systems like Soufflé [9], LogicBlox [4], or EmptyHeaded [1], and
4. data management frameworks such as Datomic, Google Logica, and CozoDB.

Our new system *Nemo* is most closely related to tools of type (2) and (3). From tools of type (2), it inherits the focus on scalability (especially with regards to data size), compatibility with open data standards like RDF, and its support for *existential rules* (a.k.a. tuple-generating dependencies), which are important in databases and rule-based ontologies. At the same time, it aims to support a broader range of datatypes, built-in operations, and aggregates, as are typically used in data analytics tools of type (3). A commonality shared with most tools above (except those of type (4)) is that Nemo runs in memory, without a persistent database backend (though using input data from such databases is planned).

Nemo is still at an early stage of development, but already able to solve real and synthetic benchmarking tasks at speeds that can compete with other tools mentioned above. This system demonstration offers a first glimpse at the current functionalities and planned upcoming features. Nemo is developed in Rust. Its source code, releases, and documentation are at `https://github.com/knowsys/nemo/`. A live demo of Nemo can be tried online at `https://tools.iccl.inf.tu-dresden.de/nemo/`.

Supported Datalog Dialect Nemo works on a custom Datalog dialect that modifies common notation from logic programming to accommodate the more flexible and accurate data model from the W3C RDF and SPARQL standards. The syntax is widely compatible with that of Rulewerk [7, formerly *VLog4j*], and with the Datalog fragment of RDFox [11]. An example is shown in Figure 1.

The program in Figure 1 integrates two data sources – public `tree`s in Dresden and `taxon`omic information about plant species from Wikidata – to find old lime (linden) trees, i.e., trees of any subspecies of genus *Tilia*. The datasets are loaded in `@source` directives, where the comments note the

Submitted to:
ICLP 2023 System Demonstrations Track

```
@declare tree(any,any,integer,integer) .
@source tree[4]: load-csv("dresden-trees.csv") . % location,species,age,height
@source taxon[3]: load-csv("wikidata-taxons.csv.gz") . % taxon,label,supertaxon

lime(?id, "Tilia") :- taxon(?id, "Tilia", ?parentId) .
lime(?id, ?name)   :- taxon(?id, ?name, ?parentId), lime(?parentId, ?parentName) .
oldLime(?loc,?species,?age) :- tree(?loc,?species,?age,?height), ?age>200, lime(?id,?species) .
```

Figure 1: Finding Dresden's oldest lime (linden) trees in Nemo

	Doctors-1M	Ontology-256	LUBM-01k	Deep200	Galen EL	SNOMED CT
Inferred facts	792,500	5,674,201	186,742,694	725,457	1,858,810	24,117,991
Nemo (sec)	3.2	13.4	163.3	5.1	3.6	62.1
VLog (sec)	2.5	22.2	199.4	*timeout*	45.2	*oom*

Table 1: Selected benchmarking results (loading+reasoning); *timeout*: 60min; *oom*: out of memory

meaning of the parameters. For `tree`, we `@declare` specific datatypes, where *any* is the most general type that supports all data (the default if no declaration is given), whereas *integer* loads numeric values. The first two rules find all species of lime tree by recursively collecting all taxons below the genus *Tilia* in the tree of life. The third rule then finds trees of some such species and an age of over 200 years. Finding Dresden's seven old limes (the oldest a small-leaved lime of 337 years) from the >88,000 city trees with known age and >3.6M taxons takes about 7 sec on a laptop, of which 200 msec are used to apply rules (the rest is for data loading). The example data and program is available online at `https://github.com/knowsys/nemo-examples` in directory examples/lime-trees.

In addition to the features illustrated above, Nemo also supports stratified negation (denoted `~`), conjunctions in rule heads (denoted `,`), further datatypes and built-ins (esp. floating point numbers), and existentially quantified head variables (using `!` instead of `?` in front of the variable name).

System Overview The underlying reasoning procedure is based on materialisation (forward chaining of rules) using semi-naive evaluation [2] and the restricted chase [6]. Key to overall performance is a combination of columnar data structures (introduced for Datalog by Urbani et al. [12]), a multiway join algorithm based on *leapfrog triejoin* [13], and own new optimisation techniques based on careful computation planning. The columnar design also allows for efficient support for values of different types at the lowest level. The system aims at maximal declarativity and syntax-independent performance (e.g., the order or parameters in predicates or the order of atoms in rules has no effect on performance).

As a system, Nemo can be invoked through a command-line client `nmo`, which includes options for storing results. Various input and output formats are supported, currently CSV and TSV, RDF, and logic programming facts. We strive to support both the elaborate type system and data representation forms of RDF, and the more basic data schemes often found in CSV or classical logic programming, without a burden on the user. Nemo is implemented in Rust and can also be used as a Rust library (a *crate*).

Experiments We compare runtimes (loading and reasoning) of Nemo (v0.2.0) and VLog (v1.3.6) on established benchmarks and real-world tasks. Times were measured on a notebook (Dell XPS 13; Ubuntu Linux 22.04; Intel i7-1165G7@2.80GHz; 16GB RAM; 512 GB SSD). Table 1 presents an overview of the results obtained. The first four result columns are existential rule benchmarks from ChaseBench [6]; the final two columns used an unoptimised encoding of a well-known OWL EL reasoning calculus on two different ontologies. Nemo matched or outperformed VLog in all experiments, with notable advantages on hard cases (Deep200 is a synthetic stress test; SNOMED is one of the largest real-world ontologies). Full details are at `https://github.com/knowsys/nemo-examples/` under evaluations/iclp2023.

Outlook Nemo is still in its early stages, and many additional features are under development. They include further datatypes, built-in functions, and (stratified) aggregates; support for structured data (functional terms, sets, frames, etc.); and interface improvements (programming APIs, extended client functionality). Moreover, we are researching new optimisation and explanation approaches for rule reasoning.

Acknowledgements This work was supported in DFG grant 389792660 (TRR 248), by BMBF in grants ITEA-01IS21084 (InnoSale) and 13GW0552B (KIMEDS), and in DAAD grant 57616814 (SECAI).

References

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun & Christopher Ré (2016): *EmptyHeaded: A Relational Engine for Graph Processing*. In Fatma Özcan, Georgia Koutrika & Sam Madden, editors: *Proc. 2016 ACM SIGMOD Int. Conf. on Management of Data*, ACM, pp. 431–446, doi:10.1145/3129246.

[2] Serge Abiteboul, Richard Hull & Victor Vianu (1994): *Foundations of Databases*. Addison Wesley.

[3] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri & Jessica Zangari (2017): *The ASP System DLV2*. In Marcello Balduccini & Tomi Janhunen, editors: *Proc. 14th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, *LNCS* 10377, Springer, pp. 215–221, doi:10.1007/978-3-319-61660-5_19.

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen & Geoffrey Washburn (2015): *Design and Implementation of the LogicBlox System*. In T.K. Sellis, S.B. Davidson & Z.G. Ives, editors: *Proc. 2015 ACM SIGMOD Int. Conf. on Mngmt of Data*, pp. 1371–1382, doi:10.1145/2723372.2742796.

[5] Luigi Bellomarini, Emanuel Sallinger & Georg Gottlob (2018): *The Vadalog System: Datalog-based Reasoning for Knowledge Graphs*. *Proc. VLDB Endowment* 11(9), pp. 975–987, doi:10.14778/3213880.3213888.

[6] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro & Efthymia Tsamoura (2017): *Benchmarking the Chase*. In: *Proc. 36th Symp. on Principles of Database Systems (PODS'17)*, ACM, pp. 37–52, doi:10.1145/3034786.3034796.

[7] David Carral, Irina Dragoste, Larry González, Ceriel Jacobs, Markus Krötzsch & Jacopo Urbani (2019): *VLog: A Rule Engine for Knowledge Graphs*. In Chiara Ghidini et al., editor: *Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II)*, *LNCS* 11779, Springer, pp. 19–35, doi:10.1007/978-3-030-30796-7_2.

[8] Martin Gebser, Benjamin Kaufmann & Torsten Schaub (2012): *Conflict-driven answer set solving: From theory to practice*. *Artif. Intell.* 187, pp. 52–89, doi:10.1016/j.artint.2012.04.001.

[9] Herbert Jordan, Bernhard Scholz & Pavle Subotic (2016): *Soufflé: On Synthesis of Program Analyzers*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Proc. 28th Int. Conf. on Computer Aided Verification (CAV'16), Part II*, *LNCS* 9780, Springer, pp. 422–430, doi:10.1007/978-3-319-41540-6_23.

[10] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, José F. Morales, Jan Wielemaker, Daniel Diaz & Salvador Abreu (2022): *Fifty Years of Prolog and Beyond*. *Theory Pract. Log. Program.* 22(6), pp. 776–858, doi:10.1017/S1471068422000102.

[11] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu & Jay Banerjee (2015): *RDFox: A Highly-Scalable RDF Store*. In Marcelo Arenas et al., editor: *Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II*, *LNCS* 9367, Springer, pp. 3–20, doi:10.1007/978-3-319-25010-6_1.

[12] Jacopo Urbani, Ceriel Jacobs & Markus Krötzsch (2016): *Column-Oriented Datalog Materialization for Large Knowledge Graphs*. In Dale Schuurmans & Michael P. Wellman, editors: *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16)*, AAAI Press, pp. 258–264, doi:10.1609/aaai.v30i1.9993.

[13] Todd L. Veldhuizen (2014): *Triejoin: A Simple, Worst-Case Optimal Join Algorithm*. In N. Schweikardt, V. Christophides & V. Leroy, editors: *Proc. 17th Int. Conf. on Database Theory (ICDT'14)*, pp. 96–106, doi:10.5441/002/icdt.2014.13.