Nemo: A Scalable and Versatile Datalog Engine

Alex Ivliev[1,*], Lukas Gerlach[1], Simon Meusel[1], Jakob Steinberg[1] and Markus Krötzsch[1]

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

Abstract

Nemo is a toolkit for large-scale data analysis that emphasizes robustness and ease of use. Nemo's core is a scalable and efficient main-memory reasoner that supports an expressive extension of Datalog with support for data types, existential rules, aggregates, and (stratified) negation. Built around this core is a versatile system of libraries and applications for interfacing with several data formats and programming languages, use as a web application, and IDE integration. In this system description, we present this toolkit giving a high-level overview of the system architecture as well as its supported language features.

Keywords

Reasoning Engine, Existential Rules, Logic Programming

1. Introduction

Datalog is an important rule language [1] forming the basis for many more complex formalisms such as existential rules [2, 3] that are of interested in both theory and practice. Accordingly, many rule reasoning systems have been presented, which we can roughly classify in the following types [4]:

1. Answer set programming solvers [5, 6] and logic programming systems such as Prolog [7]
2. Knowledge graph and database engines such as RDFox [8], VLog [9], Vadalog [10], and Graal [11]
3. Data-analytics systems such as Soufflé [12], LogicBlox [13], SocialLite [14], or EmptyHeaded [15]
4. Data management frameworks such as Datomic, Google Logica, and CozoDB

Despite the wide variety of tools, many systems described in the literature may not be a viable choice in practice, due to discontinuation or access restrictions of closed-source commercial systems. In the field of logic programming, these problems have been overcome, and advanced open-source systems such as Clingo [16] are available to researchers, whereas the situation in other rule system categories is more precarious. Among the mentioned systems of types (2) and (3), the only open-source tool with a release in the past twelve months is Soufflé.

In this extended abstract, we summarize our recent paper [17] on our new rule reasoning toolkit *Nemo* built for applications of type (2) and (3), where the computation of logical entailments (or query results) from a variety of inputs is the main reasoning task. Nemo's rule language extends Datalog with various datatypes, negation, aggregates (both stratified), and many datatype-specific functions and operators known from query languages like SPARQL while also supporting existential rules with a fast implementation of the *restricted* (or *standard*) *chase* algorithm.

Thanks to Nemo's modular architecture, we can provide command-line clients for all major platforms, a public web application with a built-in rule editor, an IDE plugin for rule editing in VSCode, and APIs for integration in several programming languages. All components of Nemo are free and open source, and their development repositories public. Most of Nemo is written in Rust, a language that emphasizes type and memory safety, and code quality is an explicit concern. The source code is available at https://github.com/knowsys/nemo.

Datalog 2.0 2024: 5th International Workshop on the Resurgence of Datalog in Academia and Industry, October 11-14, 2024, Dallas, Texas, USA

*Corresponding author.

alex.ivliev@tu-dresden.de (A. Ivliev); lukas.gerlach@tu-dresden.de (L. Gerlach); simon.meusel@mailbox.tu-dresden.de (S. Meusel); jakob_maximilian.steinberg@mailbox.tu-dresden.de (J. Steinberg); markus.kroetzsch@tu-dresden.de (M. Krötzsch)

0000-0002-1604-6308 (A. Ivliev); 0000-0003-4566-0224 (L. Gerlach); 0000-0002-9172-2601 (M. Krötzsch)

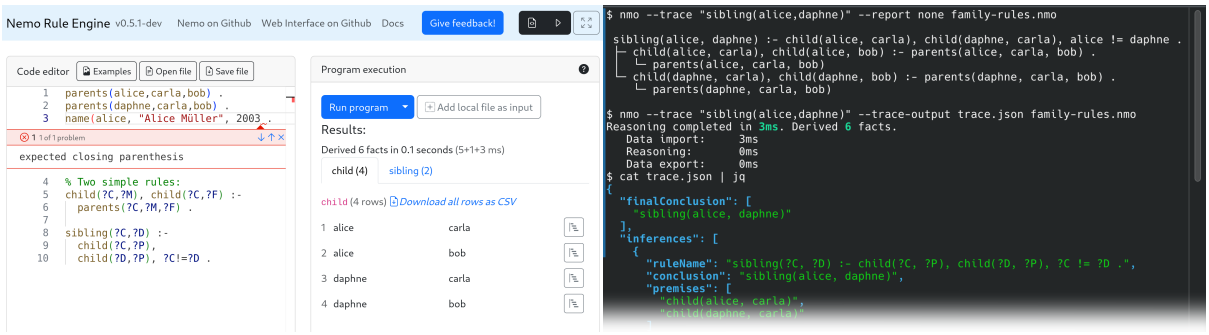

Figure 1: Rule reasoning in the browser with Nemo Web (left) and tracing on the console (right)

2. System Overview

The Nemo toolkit consists of several programs and libraries. This includes a command-line application called nmo, and a web application[1] shown in Figure 1 (left). The web application utilizes Nemo's WebAssembly interface, enabling it to run entirely in the browser without any installation. It features a powerful rule editor with syntax highlighting and auto-complete, which is made possible by Nemo's implementation of the open *language server protocol* [18]. This also enables advanced editor support for Nemo's rule language within compatible IDEs, such as VSCode. Nemo further provides a Python and Rust developer API, which expose import/export, the reasoner, and give access to the internal rule structure.

The core of Nemo is a fast and scalable reasoner that materializes logical consequences using semi-naive bottom-up evaluation [1]. For any inference that is computed, Nemo can also provide a computation trace that encodes a proof tree. This trace is available in machine-readable JSON format or in more visual representations. Figure 1 (right) shows an example use of this feature on the console.

Nemo keeps data in-memory, representing tables as hierarchically-sorted, column-based tables, following the design in VLog [9]. In contrast to VLog, our columns support domain elements of several types: fixed-size values (e.g., 32bit floats or 64bit signed integers) are stored directly, while variable-sized data (e.g., strings and IRIs) is first mapped to integer ids through a dictionary. Data in columns is further compressed using Run-Length-Encoding with increments. Tables in Nemo are accessible via trie iterators [19], which makes it possible to utilize the *leapfrog trie-join* [20], a multi-way, worst-case optimal join algorithm. In this setting, query plans are largely determined by the variable (column) order, which we heuristically determine by searching for orders that avoid inefficiencies like cross-products and excessive reordering of tables. The implementation for operations like union or difference also make use of this trie-based access. Projection and resorting of tables cannot be implemented efficiently in this way, however, and require row-based temporary tables.

Nemo's efficient storing mechanism combined with its use of state-of-the-art join algorithm and additional optimizations makes its performance competitive with other mature systems, often outperforming them on our benchmarks [17]. We were surprised that our browser-based Nemo Web often competed for second place in these evaluations. Our experiments with a version of the Wikidata knowledge graph [21] show that Nemo can reason with data sets of several billion triples. Further details on these evaluations can be found in Section 5 of [17].

3. Language Features

Nemo's rule language is based on Datalog [1], which is extended with many features of modern query and rule languages.[2] The syntax largely follows common logic programming conventions, representing rules as implications consisting of conjunctions of first-order atoms with all variables universally quantified

[1] The web application is available at: https://tools.iccl.inf.tu-dresden.de/nemo/
[2] A full documentation can be found here: https://knowsys.github.io/nemo-doc/

implicitly. Nemo further allows notation used in the query language SPARQL [22], which provides a robust encoding of identifiers, supporting special characters and data types, as illustrated next:

```
1 parents(<https://example.org/daphne>, carla, bob). parents(alice, carla, bob) .
2 nameAndYearOfBirth(alice, "Alice Müller", 2003) .
3
4 child(?C, ?M), child(?C, ?F) :- parents(?C, ?M, ?F) .
5 sibling(?C, ?D) :- child(?C, ?P), child(?D, ?P), ?C != ?D .
```

Lines 1–2 provide three facts, illustrating basic syntax for various kinds of data. In general, all syntactic forms available in RDF and SPARQL can be used, e.g., `"論理学"@ja` and `"3.1"^^<http://www.w3.org/2001/XMLSchema#float>`. Lines 4–5 show two simple rules, where `?` marks variables and `!=` denotes inequality. Nemo also supports non-monotonic negation using "~":

```
6 onlyChild(?C) :- child(?C, _), ~sibling(?C, _) .
```

As usual in logic programming, anonymous body variables are denoted by `_` and treated like different variables, i.e. variables `_` in Line 6 are distinct. Moreover, Nemo considers variables that occur only in negated body atoms to be existentially quantified beneath the negation. Hence, `~sibling(?C,_)` requires that *all* `sibling` facts do not match this pattern, whereas `child(?C,_)` requires *some* matching fact. A similar convention is used in the answer set programming, e.g., in current versions of Clingo [5].

Data Import and Export External data can be imported into Nemo using the following syntax:

```
7 @import parents :- csv { resource = "parents.csv.gz", limit = 100 } .
8 @import nameAndYearOfBirth :- csv { resource = "https://www.data.com/name_year.csv" } .
```

When providing an online resource through a URL as shown in Line 8, data will be downloaded upon program evaluation. Currently, Nemo is able to process data encoded as CSV (with user-set delimiter) or RDF. For RDF, triples (NTriples, Turtle, RDF/XML) and quads (TRiG, NQuads) are supported. All formats are available for export using analogous syntax via the `@export` directive. On the command-line, results may also be printed on the console. Various details of the import and export can be controlled through optional parameters, and data can be processed with GZip compression.

Datatypes and Functions Nemo natively supports values of many different data types, including integer, string, language-tagged string, single and double precision float, and Boolean. To manipulate such values, Nemo offers a wide range of built-in functions, which largely correspond with SPARQL `FILTER` expressions. Such functions include standard arithmetic operators and mathematical functions on numbers such as `SQRT`, functions for string manipulation such as `CONCAT`, Boolean operators, as well as comparison operators such as `<` or `!=`. Functions can be nested arbitrarily and may occur anywhere in the rule. Bindings for any variable used within a function must be sufficiently determined, similar to the standard safety condition used for Datalog rules.

Nemo is dynamically typed and therefore allows values of any type to be used in any position, without requiring a fixed schema. Most functions, such as `STRLEN` are not defined for all types of inputs, instead producing no result when given a value of unsupported type.

Aggregation Nemo supports common aggregates such as `#count`, `#sum`, or `#max`, as shown next:

```
9 childCount(?P, #count(?C)) :- child(?C, ?P) .
```

Rule 9 counts, for each value of `?P`, the number of distinct values of `?C` that satisfy the rule body. We distinguish three kinds of variables: *aggregate variables* occur in aggregate functions (in the head), *group-by variables* are the head variables that are not aggregate variables; and *body-only variables* that do not appear in the head. Aggregation is evaluated as follows: (1) find all rule matches, (2) project away

bindings of body-only variables and remove duplicates that agree on all remaining variables, (3) group the set of projected matches by distinct values of group-by variables, and (4) apply the aggregation function on each group. Duplicate elimination in (2) means we always aggregate over sets, corresponding to the keyword `DISTINCT` in many query languages. Users control the semantics through variables in the head:

```
10 sum1(?A, ?B, #sum(?N)) :- p(?A, ?B, ?N) .
11 sum2(?A, #sum(?N, ?B)) :- p(?A, ?B, ?N) .
12 sum3(?A,     #sum(?N)) :- p(?A, ?B, ?N) .
```

Rule 10 sums up the numbers N for each pair of As and Bs; rule 11 sums up the Ns from all pairs of Ns and Bs for each A (possibly having the same N value); and rule 12 sums up the distinct Ns for each A.

Existential Rules Existential rules, also known as tuple-generating dependencies, are an important formalism used to describe integrity constraints [23] or to formulate dependencies within a data exchange setting [24]. Syntactically, they are represented by introducing existentially quantified variables in the rule head, denote by `!` in Nemo, as shown in the next example:

```
13 child(?C, !P), child(!P, ?G) :- grandchild(?C, ?G) .
```

Nemo supports reasoning over existential rules by implementing the restricted (a.k.a. standard) chase [2, 3]. For each match of the body of rule 13, Nemo creates a fresh value for `!P`, provided that the head is not satisfied for any existing value yet. Fresh values are represented in Nemo by *named nulls*, which we treat as a separate type of domain element distinct from other elements. Named nulls are identified with RDF *blank nodes* in data import and export, and denoted accordingly (using notation like `_:42`).

4. Conclusion and Future Work

Nemo is a comprehensive framework for rule reasoning that includes a scalable rule engine, a feature-rich rule language, and advanced user interfaces with strong developer support. Further development of Nemo will focus on enhancing its expressive power by introducing native support for complex types such function terms and sets, and adding user-defined functions to increase flexibility and unlocking additional use cases. To further improve performance, we will transition Nemo's implementation to being able to utilize multi-threading. Next steps to improve developer tools include better highlighting of static analysis results to the user and better tracing support in the Nemo web application, which will simplify debugging and create a more robust and explainable reasoning system.

Acknowledgments

This work is in part supported by Deutsche Forschungsgemeinschaft in project number 389792660 (TRR 248, CPEC), by the Bundesministerium für Bildung und Forschung under European ITEA project 01IS21084 (InnoSale), in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), and by BMBF and DAAD in project 57616814 (SECAI).

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison Wesley, 1994.
[2] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, E. Tsamoura, Benchmarking the chase, in: Proc. 36th Symp. on Principles of Database Systems (PODS'17), ACM, 2017, pp. 37–52. doi:10.1145/3034786.3034796.
[3] M. Mugnier, M. Thomazo, An introduction to ontology-based query answering with existential rules, in: Reasoning Web: Reasoning on the Web in the Big Data Era – 10th Int. Summer School, volume 8714 of *LNCS*, Springer, 2014, pp. 245–278. doi:10.1007/978-3-319-10587-1_6.

[4] A. Ivliev, S. Ellmauthaler, L. Gerlach, M. Marx, M. Meißner, S. Meusel, M. Krötzsch, Nemo: First glimpse of a new rule engine, in: Proc. 39th Int. Conf. on Logic Programming (ICLP'23), volume 385 of *EPTCS*, 2023, pp. 333–335. doi:10.4204/EPTCS.385.35.

[5] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: From theory to practice, Artif. Intell. 187 (2012) 52–89. doi:10.1016/j.artint.2012.04.001.

[6] M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri, J. Zangari, The ASP system DLV2, in: Proc. 14th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'17), volume 10377 of *LNCS*, Springer, 2017, pp. 215–221. doi:10.1007/978-3-319-61660-5_19.

[7] P. Körner, M. Leuschel, J. Barbosa, V. S. Costa, V. Dahl, M. V. Hermenegildo, J. F. Morales, J. Wielemaker, D. Diaz, S. Abreu, Fifty years of Prolog and beyond, Theory Pract. Log. Program. 22 (2022) 776–858. doi:10.1017/S1471068422000102.

[8] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, J. Banerjee, RDFox: A highly-scalable RDF store, in: 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.

[9] J. Urbani, C. Jacobs, M. Krötzsch, Column-oriented Datalog materialization for large knowledge graphs, in: Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16), AAAI Press, 2016, pp. 258–264. doi:10.1609/aaai.v30i1.9993.

[10] L. Bellomarini, E. Sallinger, G. Gottlob, The Vadalog system: Datalog-based reasoning for knowledge graphs, Proc. VLDB Endowment 11 (2018) 975–987. doi:10.14778/3213880.3213888.

[11] J. Baget, M. Leclère, M. Mugnier, S. Rocher, C. Sipieter, Graal: A toolkit for query answering with existential rules, in: Proc. 9th Int. Web Rule Symposium (RuleML'15), volume 9202 of *LNCS*, Springer, 2015, pp. 328–344. doi:10.1007/978-3-319-21542-6_21.

[12] H. Jordan, B. Scholz, P. Subotic, Soufflé: On synthesis of program analyzers, in: Proc. 28th Int. Conf. on Computer Aided Verification (CAV'16), Part II, volume 9780 of *LNCS*, Springer, 2016, pp. 422–430. doi:10.1007/978-3-319-41540-6_23.

[13] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, G. Washburn, Design and implementation of the LogicBlox system, in: Proc. 2015 ACM SIGMOD Int. Conf. on Mngmt of Data, 2015, pp. 1371–1382. doi:10.1145/2723372.2742796.

[14] J. Seo, S. Guo, M. S. Lam, SociaLite: An efficient graph query language based on datalog, IEEE Trans. Knowl. Data Eng. 27 (2015) 1824–1837. doi:10.1109/TKDE.2015.2405562.

[15] C. R. Aberger, S. Tu, K. Olukotun, C. Ré, EmptyHeaded: A relational engine for graph processing, in: Proc. 2016 ACM SIGMOD Int. Conf. on Management of Data, ACM, 2016, pp. 431–446. doi:10.1145/3129246.

[16] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82. doi:doi.org/10.1017/S1471068418000054.

[17] A. Ivliev, L. Gerlach, S. Meusel, J. Steinberg, M. Krötzsch, Nemo: Your friendly and versatile rule reasoning toolkit, in: Proc. 21st Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'24), 2024.

[18] LSP, Official page for Language Server Protocol, Microsoft, 2024. https://microsoft.github.io/language-server-protocol/, accessed May 2024.

[19] E. Fredkin, Trie memory, Commun. ACM 3 (1960) 490–499. doi:10.1145/367390.367400.

[20] T. L. Veldhuizen, Triejoin: A simple, worst-case optimal join algorithm, in: Proc. 17th Int. Conf. on Database Theory (ICDT'14), 2014, pp. 96–106. doi:10.5441/002/icdt.2014.13.

[21] D. Vrandečić, M. Krötzsch, Wikidata: a free collaborative knowledgebase, Commun. ACM 57 (2014) 78–85. doi:10.1145/2629489.

[22] S. Harris, A. Seaborne (Eds.), SPARQL 1.1 Query Language, W3C Recommendation, 21 March 2013. Available at http://www.w3.org/TR/sparql11-query/.

[23] C. Beeri, M. Y. Vardi, A proof procedure for data dependencies, J. ACM 31 (1984) 718–741. doi:10.1145/1634.1636.

[24] R. Fagin, P. G. Kolaitis, R. J. Miller, L. Popa, Data exchange: semantics and query answering, Theoretical Computer Science 336 (2005) 89–124. doi:10.1016/j.tcs.2004.10.033.