SemReasoner - A High-Performance Knowledge Graph Store and Rule-Based Reasoner

Kevin Angele[1,3(✉)], Jürgen Angele[2], Umutcan Simsek[1], and Dieter Fensel[1]

[1] Semantic Technology Institute Innsbruck, University of Innsbruck, Technikerstrasse 21a, 6020 Innsbruck, Austria
{kevin.angele,umutcan.simsek,dieter.fensel}@sti2.at
[2] adesso, Competence Center Artificial Intelligence, Pariser Bogen 5, 44269 Dortmund, Germany
juergen.angele@adesso.de
[3] Onlim GmbH, Weintraubengasse 22, 1020 Vienna, Austria
kevin.angele@onlim.com

Abstract. Knowledge graphs have become essential for integrating data from heterogeneous sources powering intelligent applications. Integrating data from various sources often results in incomplete knowledge that needs to be enriched based on custom inference rules. Handling a large number of facts requires a scalable storage layer that must be seamlessly integrated into the reasoning algorithms to guarantee efficient evaluation of rules and query answering over the knowledge graph. To this end, we present SemReasoner, a comprehensive, scalable, high-performance knowledge graph store and rule-based reasoner. SemReasoner includes a deductive reasoning engine and fully supports document store functionality for JSON documents. SemReasoner's modular architecture is easy to extend and integrate into existing IT landscapes and applications. We evaluate SemReasoner against the state-of-the-art rule-based reasoning engines using test cases from OpenRuleBench. The results show that SemReasoner outperforms existing engines in most test cases.

Keywords: SemReasoner · OO-Logic · Rules · Knowledge Graph Store · Triple Store · Reasoner

1 Introduction

In recent years knowledge graphs have become essential for powering intelligent applications like Siri or Alexa. A knowledge graph is a vast semantic net representing entities and their relationships [11], integrating data from heterogeneous sources that are often incomplete. Therefore, in many applications (e.g., data integration or information extraction), it is essential to infer implicit knowledge based on the given statements using rules (so-called inference rules). Rules allow the decoupling of the domain logic from the underlying application code.

C. Pesquita et al. (Eds.): ESWC 2023, LNCS 13870, pp. 574–590, 2023.
https://doi.org/10.1007/978-3-031-33455-9_34

The logic not hardcoded within an application but represented by rules is easily exchangeable when needed allowing applications to be much more generic. Besides, the behavior of applications can be defined in a low-/no-code way using rules. Additionally, integrating the logic into the model makes the queries much shorter and more straightforward.

The importance of rules for knowledge graphs is also visible when following the developments in recent years. Many new rule engines [6,9,19] have been developed. For a rule engine to be broadly adopted, it is essential to provide simple and well-known APIs to access and manage the data stored within it. Developers unfamiliar with the W3C recommendations should be able to integrate a rule engine into their system architecture. But, a semantic web expert should also have full expressivity when using the rule engine. Building a bridge between developers' technologies, formats, query languages, and the W3C recommendations is essential.

This paper presents SemReasoner, a high-performance knowledge graph store and rule-based reasoner. SemReasoner is used successfully in several industrial projects powering intelligent applications.

SemReasoner provides a comprehensive, scalable, and high-performance deductive database. It stores the data in the form of triples and provides Horn logic with negation as well as OO-logic [1] (a successor of F-logic [15]) for defining rules. Additionally, SemReasoner fully supports document store functionality and allows returning JSON documents in their initial structure without recreating them. Based on SemReasoner, ontology-based applications can be developed which offer the following advantages:

- The shared meaning (semantics) of information in a knowledge model
- The capturing of complex relationships with the help of rules

Rules allow for modeling the know-how and the business logic separately from the execution logic. Hence, users can flexibly adapt and extend the application logic without modifying code. SemReasoner is mainly accessed using OO-logic queries, but GraphQL, a query language simplifying developers' access, is also supported. Furthermore, we are currently working on supporting SPARQL queries.

The remainder of the paper is structured as follows. Section 2 will lay the foundations to follow the presentation in this paper. Afterward, Sect. 3 highlights SemReasoner's key characteristics and gives insights into use cases relying on it. SemReasoner's modular architecture is presented in Sect. 4. Then, we present the evaluation against the state-of-the-art rule engines (Sect. 5). The related work section offers a comparison with those engines (Sect. 6). Finally, we conclude the paper and give an outlook on the future work in Sect. 7.

2 Background

SemReasoner provides two languages for querying, namely OO-logic and GraphQL, and for verifying the data, JSON Schema is supported.

OO-logic [1] is a successor of F-Logic [2,15]. It combines the advantages of conceptual modeling from object-oriented frame-based languages with the declarative style, compact and simple syntax, and the well-defined semantics of a logic-based language (based on first order logic). OO-logic supports typing, meta-reasoning, complex objects, methods, classes, inheritance, rules, queries, modularization, and scoped inference. It can be translated to Horn logic with non-monotonic negation, a subset of predicate logic with highly efficient reasoning algorithms. It is "Turing complete" (computationally universal), which means that everything that can be expressed by a computer can also be expressed in OO-logic. This is very important for industrial applications. For example, use cases requiring recursive functions cannot be directly expressed in a language like OWL. Further, OO-logic is more lightweight than F-Logic. For schema definition, no special constructs are available. Instead, schema definition is done precisely the same way as the definition of instances. For classes, properties are defined and used in their instances, and properties of classes are defined in the same way. This clears the syntax given for F-Logic without losing functionality and makes it much easier to describe ontologies and rules. Our commercial experience shows that OO-logic is an excellent language for industrial applications. For better support of knowledge graph use cases, OO-logic has been extended by powerful path expressions, for instance, to recursively follow edges in the graph and thus retrieve whole paths.

GraphQL. JSON objects may be queried using GraphQL[1]. GraphQL is a query language to retrieve data from a data store. Usually, GraphQL is used as an alternative to an ordinary REST call. It allows retrieving the relevant information only instead of a sometimes extensive JSON object. As its syntax is closely related to JSON, it fits very well with JSON. GraphQL allows simple queries but is less powerful than OO-logic queries. SemReasoner supports the GraphQL constructs like *aliases, unions, fragments, nested fragments, inline fragments, arguments, variables, default variables*, and *directives (@include, @skip)*. Furthermore, introspection and mutation are supported, and arguments can contain OO-logic paths. A GraphQL response is returned as a JSON document with the same structure as the query structure. Furthermore, only properties specified in the query are returned.

JSON Schema is used for verifying JSON objects. The relation between a JSON schema[2] and a JSON object is given by the class (`@type`) of the JSON object. The name of the class is the property value of the `@id` property of the schema. An extension to JSON Schema is the support for inheriting properties from super-schemas [3]. The relation of a sub-schema to its super-schema is expressed using the property `@subClassOf`. Additionally, constraints can be expressed using OO-logic by using the `@constraints` property. In SemReasoner schema objects are stored as JSON objects. Whenever a schema is available for a given type, every added instance of that type is verified against this schema.

[1] https://graphql.org.
[2] https://json-schema.org/.

3 Key Characteristics and Use Cases

We present the key characteristics (Sect. 3.1) together with industry-related use cases (Sect. 3.2) in the following.

3.1 Key Characteristics

Document Store. SemReasoner's JSON API stores added JSON documents with their initial structure. This allows retrieving the documents quickly without reconstructing them from triples. Even more important, the original structure is kept, which can not be guaranteed when recreating them from triples.

Persistent Storage. The persistent storage layer stores the triples in B+ trees. A shadow implementation of those B+ trees allows safe transactions. This implies facts added or deleted during a transaction only modify shadows of the B+ tree nodes. Further, SemReasoner partitions the extensional database vertically using the property names, i.e., each property has its relation. For instance, *abc[hasDollarPrice:5]* is stored in a B+ tree containing ternary tuples only. The member relation (member of a class) is partitioned using the class names, i.e., each class has its member relation. For instance, *abc:Product* is stored in a B+ tree containing binary tuples only. An additional horizontal partitioning splits the database into subgraphs. In that case, the B+ trees have tuples with an additional argument: the subgraph identifier.

Reasoning. Join operations are used instead of resolution for evaluating rules. Both *Merge*[3] joins and *Nested Loop*[4] joins are implemented and a heuristic chooses one of them for each join operation. The intermediate rule results are kept in the main memory, and the indices are created dynamically. For B+ relations, those are B+ trees as well. For main memory relations, those are realized either by specialized hash tables or AVL-trees, decided based on a heuristics. Cross products are managed differently, they are not executed. Relations are not joined and the cross product is forwarded to the next operator. The same holds for join operations concerning two triple sets, which are known to have a one-to-one relation. Constants in rule literals are used as additional filters for join operations (lazy filtering). Lazy projection means that projections are not applied immediately but during the next join operation. Rules with a single head literal and a single body literal do not create intermediate results. All those optimizations allow fast evaluation of rules with huge sets of facts in the extensional database.

Transactions. SemReasoner is transactional, i.e., it provides snapshot and long transactions. Snapshot transactions allow clients to add or remove facts and pose queries without influencing parallel accessing clients. The changes become

[3] https://sqlserverfast.com/epr/merge-join/.
[4] https://sqlserverfast.com/epr/nested-loops/.

visible to other clients only after committing the transaction. Such transactions can also be rolled back, leaving the original state in the extensional database as before the transaction. A rollback means that the changes in the shadow nodes are ignored, and the shadow nodes are freed up. A commit switches to the shadow nodes, and the original nodes are freed up. Long transactions are open for a long time, e.g., days, and they still allow parallel accessing clients to pose queries. Those queries are independent of the changes in the long transaction. The shadow implementation again accomplishes this. Long transactions create shadow nodes only, thus not affecting the use of the original nodes.

3.2 Use Cases

SemReasoner has been successfully used in industrial projects and applications. adesso SE[5], a listed consulting and IT service company with more than 8000 employees, employs SemReasoner in various projects and products. Some of those we describe in the following.

- **adesso insurance solutions**, a subsidiary of adesso, uses SemReasoner at the core of an *in/sure workflow* & *in/sure workplace* product. The business architecture is completely described with an ontology, and rules represent decisions. This is an excellent example of a no-code/low-code system, where ontologies describe all the concrete domain-specific issues.
- **Banking** is one line of business inside adesso using SemReasoner to analyze embargo restrictions. Large banks process millions of SWIFT payment messages daily with real-time response requirements. OO-logic rules describe conditions when a payment message violates an embargo, and SemReasoner evaluates those conditions in real-time. This use case shows the high performance of SemReasoner

Further, SemReasoner is used in Onlim[6], a leading conversational AI platform provider in the DACH-Region, to host knowledge graphs and power intelligent applications like chatbots and voice assistants.

4 SemReasoner's Architecture

SemReasoner combines a graph store with a deductive reasoning engine and comes with a search index for efficiently searching ontologies and documents. The architecture is optimized to use SemReasoner as a runtime system in semantic applications or at the core of ontology-based services. Therefore, SemReasoner comes with well-documented APIs, extension possibilities, and interfaces[7]. Figure 1 presents an overview of SemReasoner's architecture components.

[5] https://www.adesso.de/en/.
[6] https://onlim.com/.
[7] see https://kev-ang.github.io/SemReasoner/.

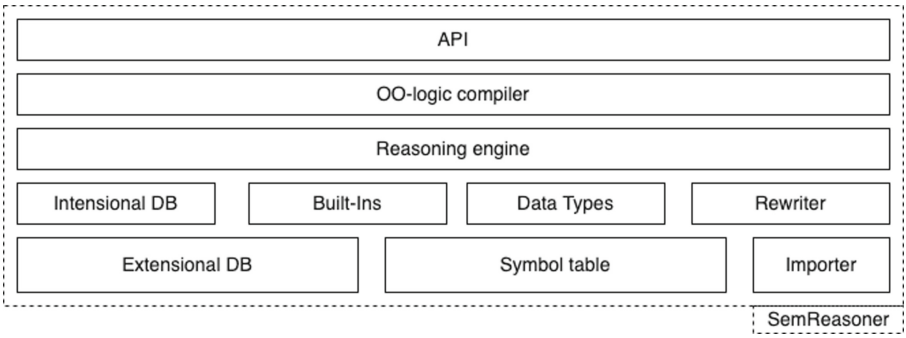

Fig. 1. SemReasoner's architecture

4.1 Storage Layer

The storage layer consists of three parts, the *Extensional Database* (EDB), *Symbol Table*, and *Importer*. While the first two are used for storing the data, the third imports the data from various formats into the EDB and Symbol Table.

Facts are separated from the rules and stored in the EDB. This EDB may either be configured to reside in the main memory (*memory mode*) or in its graph store (*persistent mode*)[8]. Additionally, a *mixed mode* keeps as much data as possible in the main memory and swaps (remove data from memory and load data from disk) whenever needed. Different data structures are used for the two modes. The graph store (persistent mode) is based on B+ trees [10]. Indices are generated and stored on disk for supporting joins and negations in reasoning. This persistent layer ensures rapid loading, rollback, parallel snapshot transactions, and backups. It has been tested for up to 34 billion triples while loading a billion facts in roughly 3.5 h on an ordinary PC. The loading time grows linearly with the number of triples. Binary trees (AVL trees) are used as the central data structure for the in-memory mode.

The storage of symbols is separated from the storage of triples. Symbols are encoded and stored within the *Symbol Table*. Therefore, the triples contain the codes of the symbols only. This allows fast comparisons of triples because only codes have to be compared. The encoding of symbols is done during the loading of the facts. The symbols are stored as B+ trees in the persistent mode, whereas hash tables are used for the in-memory mode.

Several importers are provided to load the data into memory or the graph store. SemReasoner currently supports the formats *JSON, OO-logic facts, Raw (files containing triples, with every element of a triple in a different line)* and all formats supported by the Jena[9] and Rio (RDF4J)[10]. In addition, SemReasoner provides an interface for implementing custom importers to add support for other formats.

[8] For performance reasons, the first configuration is preferable.
[9] https://jena.apache.org/documentation/io/.
[10] https://rdf4j.org/.

The storage layer is seamlessly integrated into the reasoning algorithms. It thus dramatically increases the performance compared to reasoners with a decoupled storage layer.

4.2 Logic Layer

Intensional DB, Built-Ins, Data Types, and *Rewriters* form the logic layer, containing the domain logic.

Intensional DB. Rules and queries are located in the *Intensional database*. They are arranged in a graph describing the dependency relation of the rules. A rule A is dependent on another rule B if the head atom of B unifies with one of the body literals of A. Cycles and double-connected components can be computed with the appropriate graph algorithms. Thus stratification of rules can be determined for non-monotonic negation. In addition, this allows deciding quickly which rule can contribute to the answer and which rules can be omitted.

Built-Ins. Some aspects cannot be easily described using logic. For example, complex mathematical algorithms should be described procedurally instead. SemReasoner can easily be extended with such procedural algorithms. Within OO-logic, these procedural attachments are called built-ins. They may be used inside rules or queries using predicate logic literals. For example, all mathematical built-ins are internally given in the same way. If we want to multiply numbers, we could use the multiply built-in (`_mult`) that can take two numbers as input and returns the output into a variable: `_mult(2,3,?Y)`, which results in `?Y=6`. Built-ins are identified by a leading underscore ("_"). The extension of such a built-in is not a given set of facts. Instead, the extension is computed by an algorithm. Built-ins are written in Java, are compiled against the SemReasoner code, and are registered as new built-ins. Then, they are subsequently available within queries and rules. Additionally, SemReasoner supports action built-ins. Action built-ins occur in the head of a rule and perform actions like writing a file or sending an e-mail. So far, SemReasoner comes with more than 200 built-ins, including the math and string functions from Java.

Data Types are assigned to facts and define how to interpret the data[11]. For example, adding two facts heavily depends on the assigned data type. Adding two integers results in the sum, whereas adding two strings is interpreted as concatenation. SemReasoner comes with a predefined set of data types consisting of *Boolean, Calendar, Double, Duration, Float, Integer, Long*, and *String*. As for the built-ins and the importers, extending the given set of data types is possible by implementing a given interface.

[11] https://foldoc.org/type.

Rewriter. Rewriters are used for optimizing the set of rules during the reasoning process. Optimizing the rules before evaluating them is essential for an efficient reasoning process. Without those optimizations, no real-time responses would be possible in many cases. The optimization is done by modifying the rules while ensuring that the revised rules evaluate the same answer as the original rules. An example is the magic set rewriter (see Sect. 4.3). SemReasoner comes with predefined rewriters, e.g., eliminating duplicate literals in rules or rewriting rules into SQL statements when integrating SQL databases.

4.3 Reasoning Engine

The reasoning engine operates on the existing facts stored in the Extensional DB. It uses the logic layer for inferring new knowledge or answering given queries. In the following, we will elaborate on the inference algorithms used. Afterward, the reasoning process is described, and finally, we briefly present SemReasoner's materialization abilities.

Inference Algorithms. In the kernel of SemReasoner, there are two reasoning methods available: a bottom-up reasoner (also called semi-naive evaluation, or forward chaining reasoner) [23], and a top-down reasoner based on the magic-set technology [7] (simulated backward-chaining reasoner).

A bottom-up reasoner (forward-chaining reasoner) takes the given facts, applies the rules, and creates derived facts. Afterward, the rules are applied again to derive more facts, including the derived facts of the first rule evaluation. This process continues until no new facts can be derived. Bottom-up reasoning is a simple method with the disadvantage that many (intermediate) facts are generated, which are generally unnecessary for answering the query. On the other hand, top-down reasoning sometimes provides so much overhead that this simple reasoning strategy performs best of all.

Magic set reasoning modifies the rules and processes them using a bottom-up reasoner. The rule transformation process creates a magic fact and a new rule from the given rule. This transformation directly brings a restricting ground term to the rule to be evaluated. The introduced ground term restricts the intermediate results at the bottom of the rule graph. Transforming the rules and processing the resulting rule set in a bottom-up way reduces the number of intermediate results that do not contribute to the answer. By creating these rules, we observe a trade-off between this reduction effect and the additional performance loss for magic set reasoning. We have seen queries better evaluated in a purely bottom-up fashion and others better evaluated by magic sets.

The Reasoning Process. A query is processed in several successive steps (see Fig. 2). First, the query is parsed and compiled (*OO-logic compiling)* into an internal data structure. Afterward, all rules that may contribute to the query are selected (*Selecting rules*) from all the rules stored in the Intensional DB. The resulting rules are then optimized by so-called rewriters (*Rule rewriting*).

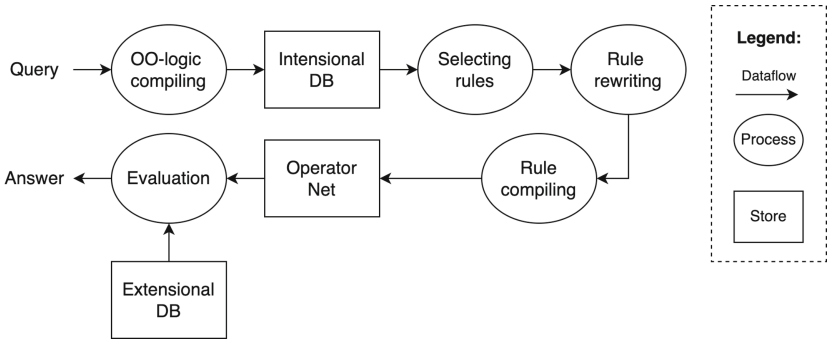

Fig. 2. Query processing in SemReasoner

Then a rule compiler creates a so-called operator net (*Rule compiling)*. An operator net is a low-level representation of the operations needed for processing the set of rules. Such an operator net contains operations like *join, match, access to the EDB, projection, operations for built-ins*, and *operations for connectors*. Move operations move tuples to another node, collectors store intermediate results, and distributors distribute tuples to several other nodes.

Such an operator net is purely data flow-oriented, with every operator performing its operation and sending the results to the successor nodes.

The whole query processing is multi-user-capable. This means multiple users can send queries simultaneously, which are processed in parallel.

Materialization. The same reasoning methods (bottom-up and top-down) can be used to materialize inferences. This means that rules are evaluated directly after loading. The results are stored in the internal triple store, which means that, during query evaluation, these facts have to be accessed and retrieved only so that no evaluation takes place during query time. This drastically improves response times but may also increase the amount of stored data. SemReasoner provides functions for incrementally materializing models. Suppose a new triple is added/deleted to/from the model. Then, the materialization process must only be repeated for a small subset of the model rather than for the entire model. Materialization for the SemReasoner can be enabled when needed. Typically, it is used without materialization.

4.4 OO-logic Compiler

The OO-logic compiler parses incoming rules and queries by using ANTLR[12] (ANother Tool for Language Recognition). This parser builds and walks parse trees. Based on the resulting parse tree, the internal Java representation of the given rules and queries is compiled. Compiling rules into the internal Java representation includes transformations for exceptional cases. Rules may contain

[12] https://www.antlr.org/.

several heads and an `OR` operation in the body. For such cases, the OO-logic compiler performs a lightweight version of Lloyd-Topor transformation [18]. This transformation compiles such a complex rule into several Horn rules. Another exceptional case is aggregations, usable in rules and queries, which are transformed into several intermediate rules.

4.5 API

SemReasoner provides three ways to integrate it into an application. Those Java APIs allow adding/removing facts and sending OO-logic queries to be evaluated. Furthermore, the APIs allow adding built-ins, data types, importers, and rules. In the following, we will briefly introduce the three APIs *Deductive Database*, *JSON Deductive Database*, and *Streaming Database*.

Deductive Database. The Deductive Database is all about OO-logic. OO-logic facts and OO-logic rules are managed, and OO-logic queries can be posted. OO-logic facts are compiled into internal triples or quads (if a context is provided). OO-logic rules and queries are stored in their internal Java representation. This API is transactional, i.e., it provides snapshot and long transactions (see Sect. 3.1 for details). The deductive database is a client to the data, i.e., several deductive databases can be used in parallel for the same core.

JSON Deductive Database. SemReasoner provides a particular API for storing and retrieving JSON objects. JSON objects can be used together with logic rules and queried as a whole or specific parts using OO-logic queries. A significant advantage of the JSON API is its document store ability. JSON objects added to SemReasoner are stored as a whole with their original structure. Therefore, the JSON object is encoded and stored in the Symbol Table, and the code is then stored in a separate JSON table. Further, the JSON Deductive Database converts the JSON objects into triples. It adds the symbols to the Symbol Table and the triples consisting of the symbol codes into the Extensional DB. For verifying JSON objects, SemReasoner supports JSON schemas.

Streaming Database. The standard way to use SemReasoner is to fill it with ontologies, ontology instances, and rules and pose queries answered by the system. This is similar to the way a database is used. For complex event processing (stream-based reasoning) the system is filled with ontologies, instances, rules, and queries. External events are streamed into SemReasoner, creating new instances which are added to the set of instances, and these events cause the stored queries to come up with new answers. SemReasoner does this incrementally. Adding a new instance does not mean that the full original query must be evaluated. Instead, the query evaluation considers the previously evaluated partial results. Only an incremental effort is necessary to derive the additional answers.

5 Evaluation

This section evaluates SemReasoner against state-of-the-art rule engines using the RUBEN [4] framework. RUBEN is a Rule Engine Benchmarking Framework with a predefined set of test cases from the OpenRuleBench [17]. Due to the limited space, we show only a selected test case for each category OpenRuleBench provides. This section first presents the experimental setup, then the methodology, and finally, the evaluation results.

Experimental Setup. RUBEN was hosted on a server with an *Intel® Core™ i9-9900K Octa-Core, 8 Cores/16 Threads, 3.60 GHz Base Frequency, 5.00 GHz Max Turbo Frequency* processor, 64 GB RAM, and Debian GNU/Linux 10. The memory was limited to 60GB to evaluate the various rule engines.

We selected rule-engines that purely evaluate rules during query evaluation time and rule-engines materializing the rules upfront[13].

- **No materialization** - Apache Jena (4.4), SemReasoner, and Stardog (8.2.2)
- **Materialization** - RDFox (6.1), SemReasoner (5.6.7), and VLog (0.8.0)

Apache Jena and Stardog are rule engines evaluating the rules during query time. Further, for those two rule engines, there is no possibility of materializing the rules upfront. Rule engines entirely relying on materialization are RDFox and VLog. In contrast to the selected engines supporting either materialization or no materialization, SemReasoner can be configured to operate in one or the other mode. Therefore, SemReasoner is placed in both categories and evaluated against both types of engines.

Methodology. RUBEN is a Java framework providing an interface to be implemented for evaluating rule engines. The data and rules need to be manually converted into the format of the particular rule engine. Each test case contains a file for the facts, a rule file, and a file with queries. Then, the materialization (if used) and afterward, the query is evaluated three times with a timeout of 15 min for each evaluation. The evaluation results are then used to calculate the average time for materialization and query response and the standard deviation.

For the scope of this paper, we have chosen the following three test cases from the OpenRuleBench[14]:

- **Large join tests - Join1** - non-recursive tree of binary joins relying on 250000 facts.
- **Datalog recursion - same-generation** - find all siblings in the same generation using cyclic and acyclic data with a data size of 24000.
- **Stratified negation - same-generation** - modified same-generation problem using cyclic data with a data size of 24000.

[13] see Sect. 6 for a detailed description of the rule-engines.
[14] For a full list and detailed description of those test cases consider [17].

Those test cases were selected based on the ability of the given rule engines. All rule engines can execute the selected test cases for the *large join tests* and *datalog recursion*. For example, some rule engines do not support n-ary predicates, which prevents them from evaluating the *Join2* test case from the OpenRuleBench. Jena and Stardog do not support negation. Therefore, no results are shown for the negation test case for both rule engines.

Results. For the presentation of the evaluation results, Table 1 shows the results of the engines without materialization. Further, in Table 2 a comparison of engines doing materialization is shown. The test cases for all engines were executed three times. After each run, the engines were restarted, and the data reloaded. Then, we calculated the average materialization time or query response time. For the non-materialized engines, we consider the query response time. In contrast, for engines running materialization, we only consider the materialization time as the query response time is the time for accessing the precomputed data, which does not give any essential insights. Besides the average of the materialization or query response time, we calculated the standard deviation. All times in the tables are given in milliseconds. The first time is the average materialization or query response time, and the standard deviation is given in brackets.

Table 1. Evaluation results not materialized (average query response time in ms, the standard deviation in ms in brackets, and the fastest times are marked bold)

Engine	**Join1**			Datalog Recursion	
	a	b1	b2	Cyc	No Cyc
Jena	timeout	timeout	51,134	timeout	timeout
SemReasoner (Memory)	**46,547.00** (1,420.65)	**18,638.67** (299.95)	**2,774.67** (104.51)	**2,330.67** (85.99)	**2,270.00** (89.37)
SemReasoner (Persistent)	96,903.67 (1,896.41)	37,542.33 (500.44)	8,103.67 (148.29)	3,325.67 (48.22)	3,257.00 (54.74)
Stardog	Exception	419,156.67 (25,884.61)	3,366.00 (253.15)	Error	Error

Jena and Stardog do not support negation in rules. Therefore, for the non-materialized engines, we left out the negation test. Due to the limited space, we only show the results of the queries without binding[15]. Only the bottom-up reasoning algorithm for SemReasoner was used, as the top-down approach only makes sense for queries with bindings. Comparing the results of the *Join1* test,

[15] for more results and the benchmarking data check https://github.com/kev-ang/SemReasoner and the ESWC branch in https://github.com/kev-ang/RUBEN.

SemReasoner is the fastest for the three given queries. A too large result causes the exception thrown by Stardog for the query `a`. For the *Datalog Recursion*, Stardog did not deliver a correct result, and the logs showed that the rule is not supported[16]. Out of the three rule engines, the open-source implementation of Jena either times out or is always the slowest. Jena delivers only a result for the `b2` query in one run. Therefore, no standard deviation can be calculated.

Table 2. Evaluation results materialized (average materialization time in ms, the standard deviation in ms in brackets, and the fastest times are marked bold)

Engine	Join1	Datalog Recursion		Negation
		Cyc	No Cyc	
RDFox	–	–	–	–
SemReasoner (Memory)	**18,157.33** (991.95)	**1,733.00** (99.06)	**1,611.00** (91.02)	45,353.67 (2,796.67)
VLog	518,676.22 (369.55)	24,245.67 (78.68)	21,467.00 (54.11)	**9,135.00** (7.94)

Besides evaluating rules during the query evaluation, SemReasoner supports materialization. SemReasoner was run in the `MEMORY` mode for this part of the evaluation using the bottom-up reasoning algorithm. This configuration was chosen as it is comparable to how the other engines run the reasoning. Table 2 presents the performance comparison between engines supporting materialization. SemReasoner supports parallel materialization and is, therefore, faster than VLog. SemReasoner's speed-up compared to VLog reaches a factor between 13 and 28 for the *Join1*, *Cyc* and *No Cyc* test of *Datalog Recursion*. However, VLog is nearly five times faster for the negation. This seems to be a terrible use case for our implementation of negation and must be investigated in detail. SemReasoner is the slowest of the evaluated engines for the negation test case. Unfortunately, for RDFox, we did not get approval for the benchmarking results before the submission deadline. Therefore, we replaced the numbers by dashes in Table 2.

6 Related Work

OO-logic is a successor of F-logic and comes with the same expressivity. In [14], the author elaborates on the relationship of F-logic and Description Logics (DLs). F-logic is computationally complete, not so DLs. While F-logic's expressivity allows simple specifications of many problems beyond the expressivity of DLs, F-logic knowledge bases can not provide computational guarantees. But, those problems can be neglected because the exponential complexity of problems in DLs provides little comfort in practice.

[16] https://docs.stardog.com/inference-engine/#known-issues.

Further, many computational problems in F-logic are decidable within polynomial time. Those are especially all queries without function symbols and a large subclass of queries beyond DLs' expressive power. Further, there are knowledge bases with decidable query answering, including function symbols. DLs are more flexible regarding representing existential information or admitting disjunctive information to the knowledge base [15].

The following briefly introduces and discusses the rule engines from the evaluation section and includes GraphDB and the Fact++ reasoner.

GraphDB uses rules in the format and semantics analogous to R-entailment[17]. R-entailment is less expressive than DLs but improves the complexity and adds meta-modeling expressivity [13]. The reasoning engine inside GraphDB is called TRREE, performing forward-chaining reasoning relying on total materialization. All data is stored in files in the storage directory.

FaCT++ is a DL reasoner based on the tableaux decision procedure [22]. It comes with a persistent and incremental reasoning mode [21]. The persistent mode stores the internal state, and precomputed inferences and reloads them when needed. Further, the incremental reasoning mode avoids reloading and reclassifying the ontology for a few changed axioms but approximates the affected subsumptions. However, it does not have an integrated persistency layer but does the reasoning in memory.

Jena is an open-source framework for Semantic Web applications[18]. Jena comes with a triple store and inference support by various reasoning engines. For this paper, only the general-purpose rule engine[19] is of interest. The reasoning engine provides forward chaining, backward chaining, and a hybrid mode. Further, the rule engine provides built-in functions like string and mathematical functions that can be extended by the user [20]. The rule engine comes with its own rules called *Jena rule*, which is comparable to Notation3. Explicitly stating the expressivity of N3Logic is difficult [8]. It is more expressive than Datalog but less expressive than FOL, and, unlike DL, it is not decidable [8].

RDFox is a main-memory RDF store supporting parallel Datalog reasoning relying on materialization [19]. It comes with a highly efficient parallel reasoning algorithm and efficient handling of owl:sameAs statements. Further, RDFox supports datalog rules, a subset of horn logic rules.

Stardog is an Enterprise Knowledge Graph Platform[20]. Stardog does not materialize inferences but evaluates rules at query time, allowing maximum flexibility. Further, it comes with its rule language based on SPARQL and supports

[17] https://graphdb.ontotext.com/documentation/10.0/reasoning.html#rule-format-and-semantics.
[18] https://jena.apache.org/.
[19] https://jena.apache.org/documentation/inference/#rules.
[20] https://www.stardog.com/.

SWRL rules [12]. The expressivity of SPARQL is equivalent to recursive safe Datalog with negation [5]. SWRL is a combination of OWL-DL and OWL-Lite, sublanguages of OWL and Unary/Binary Datalog RuleML [12], allowing positive, function-free horn clauses [16]. Additionally, built-in functions like string or mathematical functions are supported.

Focusing on the supported rules and especially the procedural extension of those, SemReasoner's rule language is more expressive than the rule languages of the presented rule engines. The presented rule engines rely on DL or Datalog, which are Horn Logic formulas without functions and are less expressive than SemReasoner's rule language. Besides SemReasoner, only GraphDB provides an integrated persistency layer. The other engines rely on a decoupled storage layer or operate only in memory. RDFox, for example, needs to have all data available in memory. SemReasoner provides forward- and (simulated) backward-chaining. From the other engines, only Jena also provides both. The others either rely on forward- or backward-chaining. Besides, SemReasoner supports materialization as well as reasoning during query-time. In contrast, GraphDB, RDFox, and VLog fully rely on materialization and Jena and Stardog on reasoning during query-time. Further, SemReasoner has an integrated document store, while others need to rely on external system integrations. Many use cases require retrieving the initial JSON document from the underlying knowledge graph. Here, storing the JSON document in its initial structure brings performance benefits and allows to return the document as it was entered. This is not possible when recreating the JSON from triples. Also, SemReasoner comes with an extensive number of built-ins.

7 Conclusion and Future Work

This paper presented SemReasoner, a comprehensive, scalable, high-performance knowledge graph store, and rule-based reasoner. SemReasoner has an integrated storage layer and a rule language based on Horn logic extended by non-monotonic negation. Further, rules can be extended by built-ins that are procedural attachments to the declarative rules.

We introduced SemReasoner's modular architecture and various APIs, allowing easy and quick integration into existing applications. SemReasoner's integrated document store functionality for JSON documents is a benefit. With SemReasoner's ability to handle a vast amount of data[21] and its reasoning performance, it outperforms other reasoning engines in most of the selected test cases.

We are currently working on integrating SPARQL as a query language to access the stored data. Besides, `sameAs` reasoning based on equivalence classes is currently being implemented. Additionally, a clustered version allows for efficiently storing and operating on larger datasets using a cluster network in the future. Further, we plan a more extended evaluation and discussion of the configuration options supported by SemReasoner.

[21] SemReasoner has been tested with up to 32B triples.

References

1. Angele, J., Angele, K.: OO-logic: a successor of F-logic. In: RuleML+ RR (Supplement) (2019)
2. Angele, J., Kifer, M., Lausen, G.: Ontologies in F-logic. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies. IHIS, pp. 45–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-92673-3_2
3. Angele, K., Angele, J.: JSON towards a simple ontology and rule (2021)
4. Angele, K., Angele, J., Şimşek, U., Fensel, D.: RUBEN: a rule engine benchmarking framework (2022)
5. Angles, R., Gutierrez, C.: The expressive power of SPARQL. In: Sheth, A., et al. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 114–129. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88564-1_8
6. Baget, J.-F., Leclère, M., Mugnier, M.-L., Rocher, S., Sipieter, C.: Graal: a toolkit for query answering with existential rules. In: Bassiliades, N., Gottlob, G., Sadri, F., Paschke, A., Roman, D. (eds.) RuleML 2015. LNCS, vol. 9202, pp. 328–344. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21542-6_21
7. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pp. 1–15 (1985)
8. Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., Hendler, J.: N3Logic: a logical framework for the World Wide Web. Theory Pract. Logic Program. **8**(3), 249–269 (2008). https://doi.org/10.1017/S1471068407003213
9. Carral, D., Dragoste, I., González, L., Jacobs, C., Krötzsch, M., Urbani, J.: VLog: a rule engine for knowledge graphs. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11779, pp. 19–35. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30796-7_2
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (2022)
11. Fensel, D., et al.: Knowledge Graphs. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-37439-6
12. Horrocks, I., et al.: SWRL: a semantic web rule language combining OWL and RuleML. W3C Member Submission **21**(79), 1–31 (2004)
13. Horst, H.J.: Combining RDF and part of OWL with rules: semantics, decidability, complexity. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 668–684. Springer, Heidelberg (2005). https://doi.org/10.1007/11574620_48
14. Kifer, M.: Rules and ontologies in F-logic. In: Eisinger, N., Małuszyński, J. (eds.) Reasoning Web. LNCS, vol. 3564, pp. 22–34. Springer, Heidelberg (2005). https://doi.org/10.1007/11526988_2
15. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. J. ACM (JACM) **42**(4), 741–843 (1995). https://doi.org/10.1145/210332.210335
16. Lawan, A., Rakib, A.: The semantic web rule language expressiveness extensions - a survey (2019). https://doi.org/10.48550/arXiv.1903.11723
17. Liang, S., Fodor, P., Wan, H., Kifer, M.: OpenRuleBench: an analysis of the performance of rule engines. In: Proceedings of the 18th International Conference on World Wide Web, pp. 601–610 (2009). https://doi.org/10.1145/1526709.1526790
18. Lloyd, J.W.: Foundations of Logic Programming. Springer, Heidelberg (2012)

19. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: a highly-scalable RDF store. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 3–20. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_1
20. Rattanasawad, T., Saikaew, K.R., Buranarach, M., Supnithi, T.: A review and comparison of rule languages and rule-based inference engines for the semantic web. In: 2013 International Computer Science and Engineering Conference (ICSEC), pp. 1–6. IEEE (2013). https://doi.org/10.1109/ICSEC.2013.6694743
21. Tsarkov, D.: Incremental and persistent reasoning in FaCT++. In: International Workshop on OWL Reasoner Evaluation, pp. 16–22 (2014)
22. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: system description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_26
23. Ullman, J.D.: Principles of database and knowledge-base systems, vol. I. Principles of computer science series, vol. 14. Computer Science Press (1988). https://www.worldcat.org/oclc/310956623. ISBN 0-7167-8069-0