Extending and Optimizing the rudof SHACL validator

Álvaro García-Fernández[1], Samuel Bustamante-Larriet[1], Jose Emilio Labra-Gayo[1] and Oscar Corcho-García[2]

[1] *Web Semantics Oviedo (WESO), University of Oviedo, Oviedo, Spain*
[2] *Ontology Engineering Group (OEG), Technical University of Madrid, Madrid, Spain*

Abstract

This paper extends the rudof SHACL validator to include SHACL-SPARQL constraints, achieving full compliance with the SHACL Core reference test suite. It also details performance optimizations that reduce validation times for large RDF graphs by orders of magnitude. These improvements aim to make rudof an alternative to established SHACL validators. To confirm this, we evaluate its throughput against six established validators across the ICDD and LUBM benchmarks. On LUBM, rudof achieves a 215× speedup over its previous version and ranks second overall, behind Jena and RDF4J and ahead of TopBraid. On ICDD, it leads all validators on the smallest building and remains competitive on mid-size cases, while the larger buildings expose the cost of complex SPARQL traversal in the current engine.

Keywords

SHACL, RDF validation, knowledge graphs, SPARQL constraints, semantic web, rudof

1. Introduction

The Shapes Constraint Language (SHACL) is a W3C recommendation [1] for validating RDF graphs against a set of conditions, known as shapes. Since its standardisation in 2017, SHACL has been widely adopted across domains such as knowledge graph curation, data quality assurance, and ontology-driven data integration. As RDF datasets grow in size and complexity, efficient SHACL validators become essential.

Several SHACL validators have been developed, including Apache Jena, TopBraid, pySHACL, Corese, RDF4J and RDFUnit (Section 2.3). Despite these options, existing tools often force a trade-off between performance, compliance and extensibility.

rudof [2] is an open source, Rust-based toolkit for RDF data that supports both SHACL and ShEx validation. However, its previous SHACL implementation had three limitations: its coverage of the SHACL Core specification was incomplete, it lacked support for SHACL-SPARQL constraints and its performance was inadequate for large-scale RDF graphs.

This paper details the optimisation and extension of the rudof SHACL validator. Our main contributions are:

- Performance optimisations for the SHACL engine that significantly reduce validation times on large RDF datasets.
- Completion of the SHACL Core specification and the introduction of SPARQL-based constraints.

Additionally, we compare the updated rudof engine against six SHACL validators, evaluating both correctness and throughput.

Section 3 outlines rudof's prior limitations and the redesign; Sections 4 and 5 present and analyse our evaluation; Section 6 concludes the paper and outlines future work.

DMKG'26: 2nd International Workshop on Data Management for Knowledge Graphs, October 2026, Bari, Italy
EMAIL: garciafalvaro@uniovi.es (Á. García-Fernández); bustamantesamuel@uniovi.es (S. Bustamante-Larriet); labra@uniovi.es (J. E. Labra-Gayo); ocorcho@fi.upm.es (O. Corcho-García)
ORCID: 0009-0008-9390-6210 (Á. García-Fernández); 0009-0005-8631-2682 (S. Bustamante-Larriet); 0000-0001-8907-5348 (J. E. Labra-Gayo); 0000-0002-9260-0753 (O. Corcho-García)

2. Background

Evaluating rudof's redesign requires understanding both the SHACL constraint model it implements and the architectural choices that determine how validators process large graphs.

2.1. The Shapes Constraint Language (SHACL)

SHACL [1] is a W3C Recommendation for validating RDF graphs via shapes graphs that declare node shapes, property shapes, target declarations, and constraint components. SHACL 1.0 defines SPARQL constraints (`sh:SPARQLConstraintComponent`), allowing arbitrary SPARQL `SELECT` queries to express conditions outside the built-in vocabulary.

2.2. The rudof SHACL Validator

rudof [2] is an open source Rust toolkit for RDF data modeling and validation. It supports SHACL [1], ShEx [3], DCTAP[1], PG-Schema [4] and conversions among these formalisms. It also provides reading, writing and conversion between different RDF serialization formats. The SHACL validator follows a multi-stage pipeline illustrated in Figure 1.

Parsing A SHACL parser traverses the shapes graph loaded into an in-memory RDF store and produces an abstract syntax tree (AST). The AST represents shapes as either `NodeShape` or `PropertyShape` and their constraints.

IR Compilation The AST is compiled into an intermediate representation (IR) that indexes shapes by integer handle for $O(1)$ lookup and resolves cross-shape references. Compilation also builds an inverted index from classes to their instances in a single $O(n)$ pass over the data graph's triples (n being the triple count), which is shared with the validation step to avoid repeated linear scans per `sh:targetClass` resolution.

Dependency Graph A dependency graph over shapes is constructed, with directed edges encoding positive and negative inter-shape references. Kahn's algorithm [5] then derives topological levels — groups of mutually independent shapes that can be validated concurrently.

Validation The processor iterates over topological levels sequentially, validating all shapes within each level in parallel. Focus and value nodes are drawn from the RDF data graph for each shape under evaluation. Each parallel worker consults the class index and a shared validation cache, ensuring each (node, shape) pair is evaluated at most once. Two validation engines are available: a native engine, which evaluates constraints directly against the in-memory store; and a SPARQL engine, which translates constraints into SPARQL queries to be executed against an in-memory store or a generic SPARQL endpoint.

The SHACL specification permits shapes to reference one another cyclically, leaving the semantics of such recursive schemas implementation-dependent [1]. rudof detects cycles at compile time and rejects validation with an error, preventing unbounded recursion. Properly handling recursive shapes — scheduling cyclic shapes and resolving inline recursive references via an in-progress cache sentinel — remains a subject for future work.

2.3. Related Work

The literature on SHACL validation covers a variety of theoretical and practical approaches aimed at handling the complexity of the language [6, 7]. Several works have explored the

[1] https://www.dublincore.org/specifications/dctap/

implementation of SHACL validators, focusing on the computational aspects of different constraint components [8, 9]. For instance, evaluating SHACL constraints over existing SPARQL endpoints has been studied to leverage mature graph database engines for validation tasks [10].

A significant portion of recent research is dedicated to addressing the challenges introduced by recursive SHACL shapes [11, 12]. The original SHACL specification [1] leaves the semantics of recursive references implementation-dependent, which has prompted the community to propose formal semantics to ensure predictable validation outcomes. Foundational work in this area established the semantics for recursive SHACL validation [13], which was subsequently extended with stable model semantics to handle negative cyclic dependencies [14] and well-founded semantics [15]. Ahmetaj et al. [16] introduced a common foundation for recursive shape languages, providing an experimental test suite and comprehensive formalizations that seek to unify validation semantics across different recursive shape implementations.

The SHACL validator ecosystem has grown substantially since the specification's publication in 2017, spanning multiple programming languages, integration strategies, and design priorities. Differences in SHACL-SPARQL support, conformance scope, and performance on large graphs mean that no single tool excels across all dimensions. Consequently, we compare rudof against the following SHACL validators:

- **Apache Jena**[2]: Java implementation using ARQ for SPARQL-based constraints.
- **TopBraid**[3]: Java reference implementation widely used as a conformance baseline.
- **pySHACL**[4]: Python implementation with broad SHACL 1.0 coverage including SHACL-SPARQL.
- **Corese**[5]: Java Semantic Web platform integrating SHACL with its native SPARQL engine.
- **RDF4J**[6]: Integrates SHACL at the storage layer for incremental validation.
- **RDFUnit**[7]: Java framework for SHACL and SPARQL-based RDF validation.

While all the validators listed above support SHACL-Core constraints, they vary significantly in their SHACL-SPARQL coverage, implementation languages, and throughput on large graphs.

3. Architecture and Implementation

The previous rudof SHACL validator suffered from several performance bottlenecks: inter-shape references were re-resolved on every constraint evaluation, the data graph was scanned linearly for each target node enumeration, mutually independent shapes were processed sequentially, intermediate structures were copied heavily across the different stages, and the abstract syntax tree (AST) and the intermediate representation (IR) coexisted in memory, with some constraint evaluations operating on the AST instead of the IR, which duplicated both work and memory. These bottlenecks were identified through profiling the old implementation with the benchmark workloads. Finally, one functional extension accompanies the performance work, support for inline SPARQL-based constraints.

3.1. Compiled intermediate representation

The IR predates this work but its use was inconsistent: some constraints evaluated against the AST, so both representations coexisted in memory and every inter-shape reference re-resolved

[2] https://jena.apache.org/documentation/shacl/
[3] https://github.com/topquadrant/shacl
[4] https://github.com/rdflib/pyshacl
[5] https://github.com/corese-stack/corese-core
[6] https://rdf4j.org/
[7] https://github.com/AKSW/RDFUnit

its target by IRI lookup. The redesigned validator ensures execution is restricted to IR. During compilation, every cross-shape reference is resolved to an integer handle, giving $O(1)$ access at validation time, and the AST is dropped from memory once the IR is built.

3.2. Inverted class index

The prior implementation evaluated class-targeted constraints — both `sh:targetClass` as a target declaration and `sh:class` as a value constraint — by scanning the data graph linearly to enumerate instances of the named class. With every referencing shape incurring its own scan, the cost grew with the product of shape count and triple count.

The inverted class index addresses this. For each class encountered in the data graph, it records two sets: the subjects asserted to be its direct `rdf:type` instances, and the classes related to it through `rdfs:subClassOf`. Both sets are derived in a single linear pass over the data graph and stored in indexed maps, so each subsequent class-targeted lookup becomes constant time instead of a fresh scan. The structure is shared by reference across all parallel workers, so this construction cost is paid exactly once per validation run.

3.3. Memoisation of validation results

Shape evaluation frequently revisits focus-node/shape pairs that have already been validated: a shape can be reached through several `sh:and` combinations, through nested `sh:node` references or simply by appearing as a target of multiple parent shapes. Each redundant evaluation not only repeats work but can also amplify it through downstream constraint chains.

The validation cache is implemented as a two-level concurrent map, keyed first by shape handle and then by focus-node identity. The outer level is partitioned, so workers operating on disjoint shape entries read and write without contention, and a cache hit needs no synchronisation at all. Earlier iterations of the cache stored results in a layout that incurred cloning overhead on every lookup. The present structure avoids this: it returns owned copies of the result list directly, keeping each cache hit free of memory allocation overhead.

3.4. Dependency-aware parallel evaluation

Shapes that share no dependency can be validated independently of one another. The prior implementation, however, evaluated all shapes sequentially in declaration order, leaving the inherent parallelism of most shapes graphs unexploited.

The dependency graph labels each inter-shape reference as positive or negative, capturing the semantic distinction that affects validation order. From this graph, Kahn's algorithm [5] produces a sequence of topological levels in which every shape's dependencies lie strictly in earlier levels. The processor then walks these levels sequentially, distributing the shapes within each level across worker threads via data-parallel iterations. Each worker thread has access to the class index and the cache. The cache is the only synchronisation point between workers, and its partitioned design avoids the lock conflicts that would otherwise arise on heavily-referenced shapes.

3.5. Reduction of memory usage

The redesigned engine shares large data structures (focus-node sets, value-node sets, the IR) by reference, transferring ownership only when modifications are necessary. The in-memory store initialises lazily on first access, graph traversal returns lazy iterators rather than materialised collections and a predicate allowlist short-circuits traversal over unrelated predicates. The in-memory store native indexes serve both traversal directions, making `sh:targetSubjectsOf` and `sh:targetObjectsOf` efficient without full-graph scans.

3.6. SPARQL-based constraint support

The prior validator implemented the SHACL-Core vocabulary but omitted its most relevant extension point, inline SPARQL constraints (`sh:SPARQLConstraintComponent`), used for conditions outside the built-in component set.

A new constraint variant was added to both the abstract syntax tree and the intermediate representation, capturing the inline `SELECT` query, the prefix declarations under which it must be parsed, and the associated constraint metadata. Evaluation follows the SHACL specification. For each focus node, the query is executed with the variable `$this` bound to the focus node; each result row of the SELECT corresponds to one validation failure, and bindings for `?value`, `?path` and `?message` — when present — populate the corresponding properties of the resulting validation result. The feature is gated at compile time so that deployments without a SPARQL engine can omit the dependency entirely, constraints declared in such a build are simply skipped.

Alongside this extension, a number of SHACL-Core constraint validators were repaired — including language-tag matching, cross-type numeric comparison in value range constraints, error propagation through `sh:or`, `sh:and` and similar components; as well as prefix-map propagation into validation reports — bringing the implementation to full SHACL-Core conformance.

3.7. Pluggable backends for large-scale validation

Validation workloads range from small shapes graphs to billion-triple knowledge graphs. The validator operates against a common RDF interface with three backends: an in-memory store, a generic SPARQL endpoint, and QLever[8] (a high-throughput SPARQL engine for very large knowledge graphs). Two validation engines are available: the native engine evaluates constraints directly against the in-memory store for maximum speed; the SPARQL engine translates constraints into SPARQL queries and can target any backend. The choice of backend is transparent to the core validation logic.

4. Methodology and results

We evaluate the redesigned rudof validator against other SHACL validators and quantify its improvements using dedicated micro-benchmarks.

The raw results can also be found in the benchmark repository[9].

4.1. Experimental setup

All benchmarks were executed on a single Ubuntu 24.04 virtual machine provisioned by a Xen Server host. The VM was allocated with 16 logical CPUs of an Intel Xeon Silver 4214 at 2.20 GHz, 98 GiB of RAM, and 4 GiB of swap.

Every evaluated validator is packaged into a dedicated Docker image and invoked through a uniform harness, providing a consistent and isolated execution environment with each tool's declared dependencies and ensuring reproducibility across runs. Each combination of validator and benchmark variant underwent 4 warm-up iterations—allowing JVM just-in-time compilation, native-extension loading, and per-tool dependency resolution to complete before measurement—followed by 10 timed iterations. These were capped by a 6 h wall-clock timeout per combination, runs failing to complete all timed iterations within this limit are reported as DNF (did not finish).

We evaluated nine validator configurations: Apache Jena (6.0.0), TopBraid (1.4.4), Corese (4.6.5), RDF4J (5.3.0-M2), RDFUnit (0.8.21), pySHACL (0.31.0)—in a Nuitka-compiled build

[8] https://github.com/ad-freiburg/qlever
[9] https://github.com/rudof-project/rudof-shacl-benchmarks/tree/master/results/iswc-2026

and in its native Python build (`pyshacl_n`)—and rudof in both its prior version (rudof v1 (0.1.146)) and its redesigned version (rudof v2 (0.3.7)).

For the dedicated micro-benchmarks, each case undergoes a 3 s warm-up phase followed by 100 timed iterations, with no per-combination timeout. In both setups, every iteration constructs a fresh validator instance and reloads the data and shapes graphs from disk, ensuring that cached state does not carry over between iterations.

4.2. Benchmark datasets

Two datasets were selected to cover complementary dimensions of validator performance.

(1) ICDD [17] is the SHACL benchmark distributed with the Information Container for linked Document Delivery, an ISO 21597 standard for cross-domain building information exchange. Four buildings of increasing size — Garage, Apartment Building, Duplex Building, and Office Building — are validated using the `binary` link variant. Each building is evaluated against two shapes graphs: **V1**, which operates over pre-materialised inference triples; and **V2**, which replaces all rules with complex multi-hop SPARQL queries that traverse the raw ICDD linkset at validation time. This yields 8 cases per validator, isolating two dimensions simultaneously: data graph scalability and SPARQL query complexity.

(2) LUBM [18], the Lehigh University Benchmark, generates synthetic university data from a fixed schema. Since no SHACL shapes are distributed with the benchmark, we authored a shapes graph for this study that encodes the structural invariants enforced by the LUBM data generator: 14 target classes, spanning organisational units to staff roles, are constrained through 17 node shapes. Cardinality constraints mirror the generator's own invariants, such as 2–4 courses taught per faculty member and 1–3 graduate courses per graduate student. Five scale factors — 5, 10, 50, 100 and 500 universities — were chosen to vary the data graph size while keeping the constraint workload constant across all cases.

4.3. Specification conformance

The report from each validator is collected alongside the timing measurements. Across all completed benchmark variants, the benchmark datasets conform to all evaluated shapes graphs. Throughout this comparison, TopBraid serves as the conformance reference, as it is the most complete SHACL implementation among the evaluated validators [19]. rudof v2 agrees with TopBraid on every completed variant, correctly asserting that the datasets conform to the shapes without yielding any validation results. This confirms full agreement with the SHACL-Core specification and the SPARQL-based constraint requirements.

4.4. Cross-engine throughput

The nine validator configurations are compared on both datasets: ICDD isolates the cost of constraint patterns by scaling shapes graph topology rather than data volume, while LUBM keeps the shapes graph fixed and scales only the data graph, isolating the cost attributable to data volume instead. Table 1 reports the per-case median iteration time for every validator configuration on ICDD, while Figure 2 shows the corresponding mean iteration time; Table 3 and Figure 3 report the same pair of statistics for LUBM.

4.5. In-process micro-benchmarks

Table 4 reports the per-case median iteration time for rudof v1 ($\widetilde{t}_{v1}$) and rudof v2 ($\widetilde{t}_{v2}$), as recorded by the in-process micro-benchmark suite. It also includes the median ratio ($\widetilde{t}_{v1}/\widetilde{t}_{v2}$).

Table 1
Per-case median iteration time (ms) of every evaluated validator on ICDD

Case	Jena	TopBraid	RDF4J	Corese	RDFUnit	pySHACL (N)	pySHACL	rudof v1	rudof v2
1-v1	15.206	23.949	27.905	38.003	60.951	774.533	574.121	7.107	4.511
2-v1	40.975	45.795	257.990	365.320	74.317	5337	4651	—	45.031
3-v1	52.399	68.103	397.140	503.861	92.271	8186	7387	—	84.160
4-v1	146.206	153.989	1693	1265	251.863	44 989	39 851	—	410.010
1-v2	18.736	27.099	18.810	48.996	51.377	1038	911.719	0.541	14.897
2-v2	68.352	76.245	2415	229.314	59.925	69 009	109 047	7.655	152.476
3-v2	209.632	229.171	118.709	385.455	73.566	—	—	16.334	522.141
4-v2	1926	1865	213 162	2016	195.278	—	—	76.410	10 160

Table 2
Per-case median iteration time of every evaluated validator on LUBM

Case	Jena	TopBraid	RDF4J	Corese	RDFUnit	pySHACL (N)	pySHACL	rudof v1	rudof v2
5	33.460	64.612	36.368	577.109	659.921	869.364	1137	15524	57.819
10	39.611	68.706	37.593	615.460	617.037	928.101	1241	17175	65.130
50	37.047	66.138	37.171	607.408	640.472	905.154	1185	17103	64.153
100	41.748	67.728	36.183	599.315	619.416	934.811	1205	17559	63.899
500	36.798	67.396	37.236	618.619	675.635	969.608	1223	17272	64.757

Table 3
Per-case median iteration time (ms) of every evaluated validator on LUBM

Case	Jena	TopBraid	RDF4J	Corese	RDFUnit	pySHACL (N)	pySHACL	rudof v1	rudof v2
5	33	64	36	577	659	869	1137	15524	57
10	39	68	37	615	617	928	1241	17175	65
50	37	66	37	607	640	905	1185	17103	64
100	41	67	36	599	619	934	1205	17559	63
500	36	67	37	618	675	969	1223	17272	64

5. Discussion

We analyze the conformance results, micro-benchmark improvements, and cross-engine rankings.

5.1. Specification conformance qualification

rudof v2 agrees with TopBraid [19] across all completed variants. Because the datasets are fully compliant, both validators return `sh:conforms true` and emit zero validation results. While this agreement is based solely on the final conformance verdict, it matters because rudof v2 now fully evaluates SPARQL constraints.

In contrast, the same `sh:conforms true` verdict from rudof v1 requires a caveat, it silently skipped `sh:SPARQLConstraintComponent`.

5.2. In-process improvement over rudof v1

Table 4 highlights the redesign's impact under the in-process micro-benchmark configuration. Here, process startup costs are amortised during warm-up. Furthermore, per-iteration setup steps—such as graph loading, validator construction, and IR compilation—are excluded from the timing window, ensuring we strictly measure the validation call itself.

Table 4
Per-case median iteration time (ms) of rudof v1 and rudof v2. (On the ICDD v2 cases, rudof v2 is slower than rudof v1 because it evaluates the SPARQL-based constraints, whereas rudof v1 silently skips them).

Dataset	Case	$\widetilde{t}_{v1}$ (ms)	$\widetilde{t}_{v2}$ (ms)	$\widetilde{t}_{v1}/\widetilde{t}_{v2}$
ICDD	1-v1	4.513	4.321	1.044
ICDD	2-v1	452.335	40.173	11.260
ICDD	3-v1	1525.586	70.928	21.509
ICDD	4-v1	34277.297	370.682	92.471
ICDD	1-v2	0.521	13.666	0.038
ICDD	2-v2	6.072	137.417	0.044
ICDD	3-v2	13.720	410.730	0.033
ICDD	4-v2	68.818	7848.270	0.009
LUBM	5	9083.341	42.980	211.340
LUBM	10	10400.407	46.794	222.261
LUBM	50	10412.016	48.101	216.460
LUBM	100	10312.429	45.290	227.697
LUBM	500	9898.456	45.053	219.708

5.2.1. ICDD

The ICDD benchmark reveals different behaviour depending on the shape variant. On V1 cases, rudof v2 and rudof v1 are nearly equal at building 1 (4.3 ms and 4.5 ms), but v2 significantly outperforms v1 as buildings grow: 11× faster at building 2, 21× at building 3, and 92× at building 4 (371 ms versus 34277 ms). The inverted class index is the dominant factor, since the ICDD shapes reference class-targeted constraints and each additional building compounds the cost of repeated linear scans in rudof v1. On V2 cases — all rules expressed as complex multi-hop SPARQL queries — rudof v1 appears fast (0.5–69 ms) because it silently skipped every `sh:SPARQLConstraintComponent`; rudof v2 evaluates them correctly, but the cost grows sharply with building size, reaching 7848 ms at building 4 and confirming that multi-hop traversal queries remain the current performance bottleneck of rudof's SPARQL engine.

5.2.2. LUBM

The LUBM benchmark yields the most consistent improvement, showing a speedup of roughly 215× across every scale factor (ranging from 211× to 228×). This stable ratio confirms that rudof v1's flat performance profile was bottlenecked by the repeated enumeration of class instances, not by data volume. In contrast, rudof v2 resolves these lookups in constant time using its inverted class index.

5.3. Cross-engine analysis

We compare nine validator configurations across ICDD and LUBM datasets.

5.3.1. ICDD

On V1 cases, rudof v2 is the fastest evaluated validator at building 1 (4.7 ms), matches Jena and TopBraid at building 2 (≈45 ms each), then falls progressively behind the leading pair at buildings 3 and 4, where Jena reaches 52 ms and 146 ms while rudof v2 reaches 84 ms and 410 ms. rudof v1 errored at buildings 2–4 in the harness. pySHACL is heavily penalised by SPARQL overhead even on V1 shapes, reaching ≈7–8 s at building 3 and ≈40–45 s at building 4. On V2 cases, rudof v2 leads at building 1 (14.8 ms), ahead of Jena (18.7 ms) and RDF4J (18.8 ms), but the cost grows steeply: RDFUnit proves the fastest evaluator at buildings 2–4 (61 ms, 73 ms, and

190 ms), well ahead of rudof v2 (152 ms, 549 ms, and 10177 ms). The divergence at building 4 is significant — RDF4J reaches ≈213 s, pySHACL does not finish (DNF), and Jena, TopBraid, and Corese cluster near 1865–2016 ms — confirming that the two shape variants stress entirely different parts of each validator's pipeline. Figure 2 shows the detailed per-case plots comparing all nine configurations.

5.3.2. LUBM

LUBM keeps the shapes graph fixed and scales only the data graph. rudof v1's execution time is flat at approximately 15–17 s regardless of scale, revealing that the bottleneck was repeated class-membership resolution rather than data volume. The inverted class index eliminates this with constant-time lookups (Section 3.2). As shown in Figure 3, rudof v2 achieves around 58–65 ms across all five scale factors, placing it ahead of TopBraid (around 64–69 ms) and behind Jena (around 33–42 ms) and RDF4J (around 36–38 ms).

6. Conclusion

This paper fulfills the objectives of optimising and extending the rudof SHACL validator. We present a complete redesign centred around four key architectural choices: consolidating execution on an intermediate representation (dropping the AST to reduce memory footprint), introducing an inverted class index, adding a memoisation cache for validation results and implementing a dependency-aware parallel evaluation strategy. This redesign extends the validator to support SPARQL-based constraints in full conformance with the corresponding reference test suite. Furthermore, it delivers an order of magnitude reduction in validation time on data-bound workloads, placing rudof ahead of TopBraid on LUBM and comparable to state-of-the-art implementations.

6.1. Contributions

Our primary contribution is the optimisation and extension of the rudof validator. The four design choices jointly deliver a roughly 215× speedup on LUBM. Furthermore, they bring rudof's SHACL-Core and SPARQL-based constraint coverage into full agreement with the reference test suite [19].

Our cross-engine benchmark comparison positions rudof v2 within the wider SHACL validator ecosystem.

6.2. Limitations

Several limitations drive our future work directions.

One of the most significant limitations concerns rudof's SPARQL validation engine, which serves both the QLever and generic SPARQL endpoint backends (Section 3.7). It is still lightly optimised, it works acceptably on small workloads but becomes impractical on larger ones due to two compounding effects. First, when targeting the QLever backend, the engine issues an independent HTTP request per constraint evaluation and this per-request network exchange wastes the bulk of the execution time, which is why the QLever backend was excluded from the cross-engine comparison. Second, even when this HTTP cost is amortised (such as against a generic SPARQL endpoint), the per-constraint work still scales poorly on queries requiring multi-hop traversal, as the ICDD comparison shows. Both issues are addressable by profiling the SPARQL engine against either backend.

Another limitation involves recursive shape definitions. The dependency graph construction detects cycles at compile time; when a cycle is found, rudof rejects validation with an error, preventing unbounded recursion. While SHACL 1.0 permits processors to reject recursion as long

as they indicate it [1]—meaning rudof satisfies the specification—proper cycle-aware scheduling and in-progress cache sentinels remain future work.

Finally, the cross-engine methodology itself has limitations. Time constraints restricted the evaluation to a single benchmark harness and six external validators. We did not have the scope to audit the methodology further—such as by reporting cold-start versus steady-state performance, evaluating non-conformant data, or by deeply characterising each validator's startup cost. A truly comprehensive comparison would benefit from reporting data-load and validation costs separately rather than as a single aggregate. Therefore, these results should be interpreted as positional rather than definitive.

6.3. Future Work

We identify several future work directions, ordered by priority:

The first and most pressing direction is tracking the SHACL 1.2 specification as it matures [20]. The present implementation already supports reification, but it does not yet cover the rest of the SHACL 1.2 surface—particularly the new constraints over RDF 1.2 triple terms. Once SHACL 1.2 stabilises as a W3C Recommendation, updating the AST, the IR, and the constraint evaluators will bring rudof into full compliance with the next major revision of the standard.

The second direction is fully supporting recursive shapes. rudof currently rejects validation when cyclic shape dependencies are detected. Addressing this properly requires adding an in-progress sentinel to the memoisation cache so that inline recursive references resolve safely, and implementing a cycle-aware variant of the topological scheduler so that cyclic shapes are scheduled and validated rather than rejected.

The third direction is improving the SPARQL engine and its QLever and generic endpoint backends, which represents the most significant performance limitation identified in this work. The per-constraint HTTP cost can be reduced by batching requests or replacing the HTTP transport with a native QLever client API; target-node resolution should likewise be batched to avoid per-shape query explosion. The multi-hop traversal cost exposed by the ICDD V2 cases calls for three changes: rewriting the current focus-node evaluation into a single batched query per shape, binding all focus nodes through a `VALUES` clause so the store's own indexes can seek them directly instead of the engine issuing a separate scan per node; caching the parsed and optimised query plan per shape, rather than reparsing and replanning identical SPARQL text on every focus-node evaluation; and reordering multi-hop triple patterns using selectivity statistics from the underlying store, so that highly selective patterns execute before open-ended traversal steps. Once optimised, the SPARQL engine can be benchmarked against very large datasets such as Wikidata or UniProt [21].

The fourth direction targets the data-load path itself, which currently dominates end-to-end runtime on heavier workloads. Improvements via a faster in-memory store, streaming parsers, or amortising load costs across repeated validations would directly lower wall-clock times for typical users.

The fifth direction is an ablation study to isolate each optimisation. The current study reflects the combined effect of all optimisations. Disabling them independently would quantify how much of the gain each change accounts for on its own. This study should also report peak memory usage: one of the optimisations explicitly targets memory footprint, yet the current benchmarking harness measures execution time only.

The sixth direction is broadening the cross-engine benchmark comparison along three axes: validator coverage (adding tools not yet packaged into the harness, such as maplib[10] and dotNetRDF[11]), methodology coverage (reporting cold-start versus steady-state metrics and using non-conformant data to evaluate each validator's error-handling behaviour) and dataset coverage. In particular, measurements on the ERA benchmark [22]—the linked-data publication of

[10] https://github.com/DataTreehouse/maplib
[11] https://github.com/dotnetrdf/dotnetrdf

the European Union Agency for Railways—are actively being collected at the time of writing and will be included in an extended evaluation.

Finally, the seventh and lowest-priority direction is continuous regression detection: wiring the in-process micro-benchmark suite into rudof's CI pipeline to flag per-case regressions automatically before they reach end users.

Acknowledgments

This work has been partially funded by the project *SHAKIGRA*: Shaping Knowledge and Interoperable Graphs, code: NAC-ES-PUB-ASV-2025 PID2024-157010OBI00, from the Spanish Research Agency, and the regional project with code SEK-25-GRU-GIC-24-089.

This publication is based upon work from COST Action CA23147 GOBLIN – Global Network on Large-Scale, Cross-domain and Multilingual Open Knowledge Graphs, supported by COST (European Cooperation in Science and Technology, https://www.cost.eu).

We also thank Magnus Bakken for his guidance and helpful comments about the benchmarks harness.

Declaration on Generative AI

During the preparation of this work, the author(s) used Claude and Gemini in order to check grammar and spelling, and to paraphrase and reword text. After using these tools/services, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] H. Knublauch, D. Kontokostas, Shapes Constraint Language (SHACL), 2017. URL: https://www.w3.org/TR/shacl/, w3C Recommendation, 20 July 2017.

[2] J.-E. Labra-Gayo, A. Iglesias-Préstamo, D. Martín-Fernández, M.-A. Arnaud, et al., rudof: A rust library for handling rdf data models and shapes, Proceedings of the 23rd ISWC: Posters and Demos (2024).

[3] E. Prud'hommeaux, J. E. Labra Gayo, H. Solbrig, Shape expressions: an rdf validation and transformation language, in: Proceedings of the 10th International Conference on Semantic Systems, 2014, pp. 32–40.

[4] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, A. Green, J. Hidders, B. Li, L. Libkin, V. Marsault, W. Martens, et al., Pg-schema: Schemas for property graphs, Proceedings of the ACM on Management of Data 1 (2023) 1–25.

[5] A. B. Kahn, Topological sorting of large networks, Communications of the ACM 5 (1962) 558–562.

[6] D. Tomaszuk, Rdf validation: A brief survey, in: International Conference: Beyond Databases, Architectures and Structures, Springer, 2017, pp. 344–355.

[7] I. Boneva, J. E. Labra Gayo, E. G. Prud'Hommeaux, Semantics and validation of shapes schemas for rdf, in: International semantic web conference, Springer, 2017, pp. 104–120.

[8] P. Pareti, G. Konstantinidis, F. Mogavero, T. J. Norman, Shacl satisfiability and containment, in: International Semantic Web Conference, Springer, 2020, pp. 474–493.

[9] M. Leinberger, P. Seifer, T. Rienstra, R. Lämmel, S. Staab, Deciding shacl shape containment through description logics reasoning, in: International Semantic Web Conference, Springer, 2020, pp. 366–383.

[10] J. Corman, F. Florenzano, J. L. Reutter, O. Savković, Validating shacl constraints over a sparql endpoint, in: International Semantic Web Conference, Springer, 2019, pp. 145–163.

[11] B. Bogaerts, M. Jakubowski, Fixpoint semantics for recursive shacl, arXiv preprint arXiv:2109.08285 (2021).
[12] P. Pareti, G. Konstantinidis, F. Mogavero, Satisfiability and containment of recursive shacl, Journal of Web Semantics 74 (2022) 100721.
[13] J. Corman, J. L. Reutter, O. Savković, Semantics and validation of recursive shacl, in: International Semantic Web Conference, Springer, 2018, pp. 318–336.
[14] M. Andresel, J. Corman, M. Ortiz, J. L. Reutter, O. Savkovic, M. Simkus, Stable model semantics for recursive shacl, in: Proceedings of The Web Conference 2020, 2020, pp. 1570–1580.
[15] C. Okulmus, M. Šimkus, Shacl validation under the well-founded semantics, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, volume 21, 2024, pp. 553–562.
[16] S. Ahmetaj, I. Boneva, J. Hidders, M. Jakubowski, J.-E. Labra-Gayo, W. Martens, F. Mogavero, F. Murlak, C. Okulmus, O. Savković, M. Šimkus, D. Tomaszuk, Common foundations for recursive shape languages, 2026. URL: https://arxiv.org/abs/2604.20946. arXiv:2604.20946.
[17] P. Hagedorn, P. Pauwels, M. König, Semantic rule checking of cross-domain building data in information containers for linked document delivery using the shapes constraint language, Automation in Construction 156 (2023) 105–106.
[18] Y. Guo, Z. Pan, J. Heflin, Lubm: A benchmark for owl knowledge base systems, Journal of Web Semantics 3 (2005) 158–182.
[19] J. E. Labra Gayo, H. Knublauch, D. Kontokostas, SHACL Test Suite and Implementation Report, 2026. URL: https://w3c.github.io/data-shapes/data-shapes-test-suite/, w3C Document, 23 June 2026.
[20] H. Knublauch, T. Bergwinkl, Y. Taghzouti, J. Wright, SHACL 1.2 Core, 2026. URL: https://www.w3.org/TR/shacl12-core/, w3C Working Draft, 19 June 2026.
[21] Uniprot: the universal protein knowledgebase in 2025, Nucleic acids research 53 (2025) D609–D617.
[22] E. Martınez-Sarmiento, E. Ruckhaus, J. Toledo, D. Dona, O. Corcho, Era-shacl-benchmark: A real-world benchmark to assess the performance and quality of in-memory shacl engines (2025).

Figures

This appendix contains additional supporting figures.

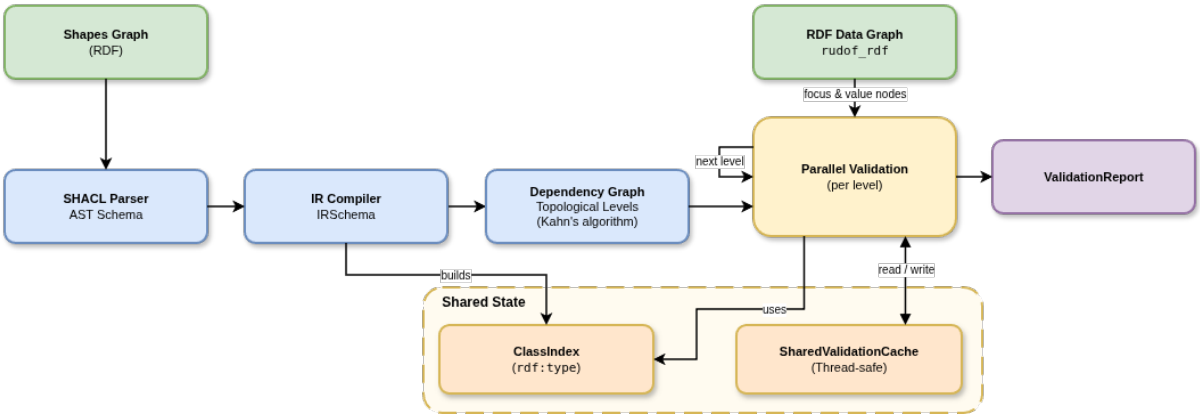

Figure 1: The rudof SHACL validation workflow.

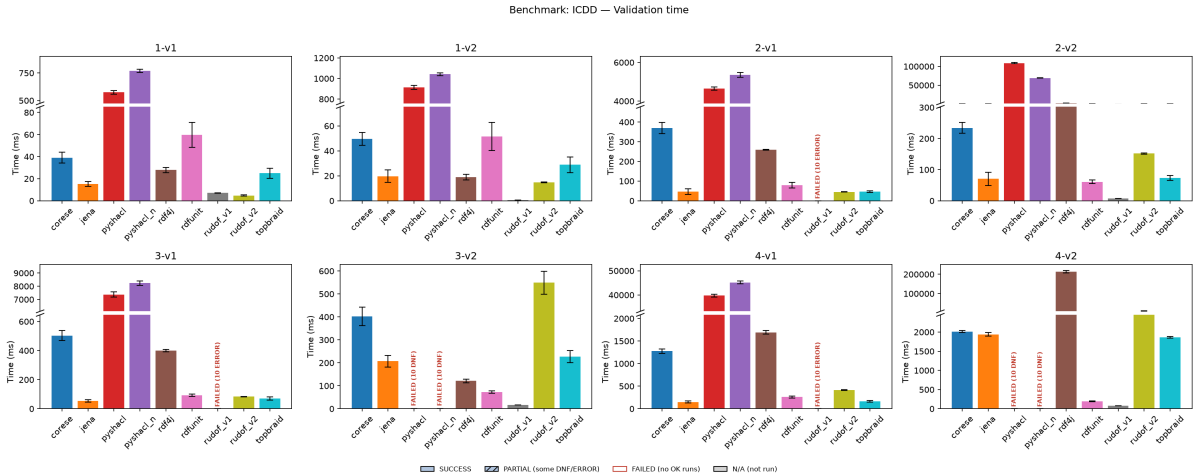

Figure 2: Mean validation time for the ICDD dataset.

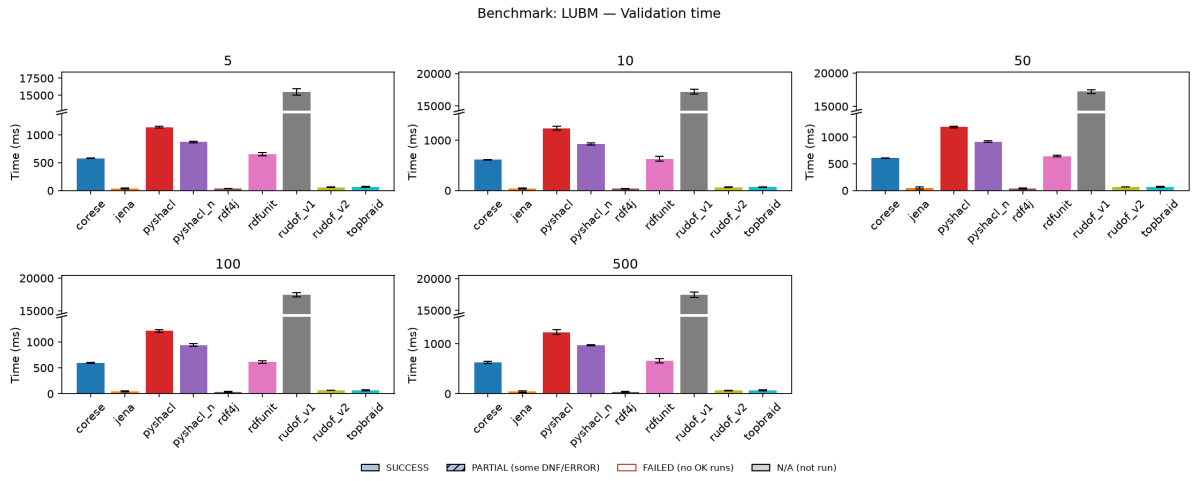

Figure 3: Mean validation time for the LUBM dataset.