Improving the ShExML engine through a profiling methodology

Herminio García-González
Kazerne Dossin, Mechelen, Belgium
E-mail: herminio.garciagonzalez@kazernedossin.eu

Abstract. The ShExML language was born as a more user-friendly approach for knowledge graph construction. However, recent studies have highlighted that its companion engine suffers from serious performance issues. Thus, in this paper I undertake the optimisation of the engine by means of a profiling methodology. The improvements are then measured as part of a performance evaluation whose results are statistically analysed. Upon this analysis, the effectiveness of each proposed enhancement is discussed. As a direct result of this work the ShExML engine offers a much more optimised version which can cope better with users' demands.

Keywords: declarative mapping rules, knowledge graph construction, data mapping languages, performance evaluation, profiling

1. Introduction

Declarative mapping rules have emerged as a more reusable, adaptable, shareable and understandable method of constructing knowledge graphs that supersedes *ad-hoc* ones [13]. While it all started with one-to-one transformations (e.g., R2RML[1]), the topic was soon shifted towards handling more heterogeneous data formats with one single representation [1], starting with RML [2].

From this point a myriad of languages and engines have emerged tackling different challenges and proposing different syntaxes and functionalities that could appeal to different users' profiles [3, 13]. Among the different solutions, ShExML was devised with usability in mind, trying to make the rules writing easy for users who are not always familiarised with Semantic Web technologies [4]. However, unlike other languages and specifications, ShExML only counts on one compliant engine[2] which can limit the possibilities for further optimisations [5, 6]. This aspect has been revealed in the recent SPARQL-Anything performance evaluation which shows how the ShExML engine performs drastically worse than other competitors [7].

Therefore, in this paper I undertake the task of improving the performance of the ShExML engine by means of a profiling analysis which reveals the performance bottlenecks present in the engine. The results of this analysis led to several improvements that had been gradually introduced in consecutive engine versions.[3] Therefore, in order to demonstrate the reliability of this analysis, a performance evaluation is conducted over the improved versions of the ShExML engine to study the real impact of these enhancements on the overall performance. The results yielded in this study could serve other practitioners who seek to enhance their engines or their knowledge graph construction workflows.

[1]https://www.w3.org/TR/r2rml/
[2]https://github.com/herminiogg/ShExML
[3]https://github.com/herminiogg/ShExML/releases

1570-0844/$35.00 © 0 – IOS Press. All rights reserved.

The rest of the paper is structured as follows: in Section 2 the related work is presented, Section 3 presents the ShExML engine algorithm and how it works while in Section 4 the introduced improvements are presented. Section 5 explains the experiments, and exposes and discusses the results. Finally, the future lines of work upon this paper are drafted in Section 6 and conclusions of this work are drawn in Section 7.

2. Related work

Some previous works have tackled the performance comparison of different mapping languages. For example, upon its launch, SPARQL-Generate was compared against RML for a set of size-increasing CSV documents [14]. Similarly, in [15] the authors compare a parallel-ready version of RML, RMLStreamer, against a set of size-increasing inputs in CSV, XML and JSON formats.

In a effort to bring some cohesion to this field, GTFS-Madrid-Bench, a benchmark based on transport data, was proposed [16] which has led to different evaluations using it (e.g., [17, 18]) and even a challenge[4] in a well-established workshop in the field leading to more consequent evaluations [19, 20].

More recently SPARQL-Anything was launched, comparing it against other existing declarative mapping languages (SPARQL-Generate, RML and ShExML) using two different sets for evaluation: 4 different JSON files and an increasing in size input in JSON [7]. Upon the delivered results it was demonstrated that the ShExML engine had serious performance problems.

However, a common issue of these evaluations is that all of them use a flat structure for the input files which cannot account for the possible differences and bottlenecks that processing hierarchical files may impose. As in the case of GTFS-Madrid-Bench, even though there are many different inputs that correlate to each other, the actual composition is delegated to a join function. Moreover, in many cases the results are not statistically contrasted nor analysed which can hinder and limit the assumptions made from them.

Therefore, this work tackles the optimisation of the ShExML engine using a profiling technique but taking a different approach for the evaluation of the results, including a hierarchical set, and delivering a statistical analysis of the produced results.

3. ShExML engine algorithm

The ShExML engine is implemented in Scala which allows for designing a purely functional algorithm which could potentially ease the future parallelisation of the engine. Therefore, the algorithm is merely conceived around this idea, even though some points do not strictly preserve the solely-functional behaviour as it will be detailed later.

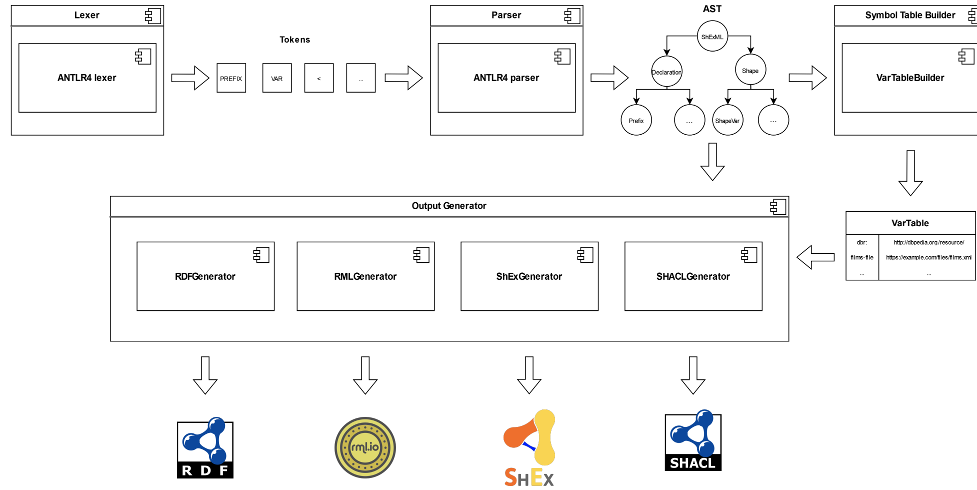
3.1. Engine architecture

The ShExML engine follows the pipes and filters architectural pattern based in the extended architectural design in which many compilers and interpreters are based on. Figure 1 shows how the engine architecture is composed and how the ShExML engine adapts it for its specific purpose.

The engine abstract workflow is as follows: (1) the lexical analyser produces a series of tokens from a given input, (2) the syntax analyser produces an Abstract Syntax Tree (AST) from the given set of tokens, (3) the symbol table is constructed upon the AST in order to be able to resolve the variables later, (4) the output generator receives the AST and the symbol table and generates an output. One particularity is that the engine does not implement a semantic analyser which is not strictly necessary for the functioning of the engine. However, this is a common problem in many declarative mapping rules engines as mentioned in [4], and it is envisioned to be implemented in the future to help users of this engine to better understand the thrown errors. This abstract workflow translates to the following specific implementations: (1) and (2) are implemented using ANTLR4[5] [8] which allows to build

[4]https://kg-construct.github.io/workshop/2023/#call
[5]https://www.antlr.org/

Fig. 1. Diagram of the pipes and filters architecture implemented in the ShExML engine.



a lexer[6] and a parser[7] using a grammar-based input. In ShExML, for maintanability reasons, the lexer and the parser grammars are separated. The result of the parser is then received by the ASTCreatorVisitor[8] that acts as an intermediary between the ANTLR4-generated visitor and the engine domain model[9] — decoupling the AST model from the lexer and parser used technology. The step (3) registers the symbols declared across the mapping rules and make them available for the output generators, generating a dictionary with the different symbols and their meaning. This dictionary is right now implemented as a mutable HashMap which could possibly prevent the parallelisation of the algorithm, however, this could be changed in the future by its thread-safe counterpart. Finally, in the final step (4), the output is generated by one of the output generators. Right now the ShExML engine counts on four different generators: `RDFGeneratorVisitor` (the main one that outputs RDF in the form of an Apache Jena model), `RMLGeneratorVisitor` (translates ShExML mapping rules to RML equivalent ones and outputs them as an Apache Jena model [9]), `ShExGenerationVisitor` (infers and outputs Shape Expressions that validate the data generated by the mapping rules), `SHACLGenerator` (similar behaviour to the ShEx generator but with SHACL shapes.) For the purpose of this study I will focus on the `RDFGeneratorVisitor` which has the greatest load and is responsible for the RDF generation being compared in the evaluation.

3.2. RDF generation algorithm

The RDF generation algorithm in the ShExML engine is broadly composed of two main operations. Firstly, a general operation processes each shape, iterates over the predicate-object tuples and correlates the results of each tuple to their corresponding subject results (see Algorithm 1.) Then, to generate the results and evaluate the expressions enclosed in the shapes actions, Algorithm 1 calls the Algorithm 2 which uses the symbol table to transpose the variables to the partial queries and then composes these partial queries into the final queries. Once the set of final queries is completed these are executed against the given file. For the sake of the explainability of the algorithm many specificities have been excluded but the whole algorithm implemented in Scala can be consulted in the engine source code repository.[10] Other details have been encapsulated in calls to functions which can be interpreted as follows:

[6]https://github.com/herminiogg/ShExML/blob/master/src/main/java/com/herminiogarcia/shexml/antlr/ShExMLLexer.g4
[7]https://github.com/herminiogg/ShExML/blob/master/src/main/java/com/herminiogarcia/shexml/antlr/ShExMLParser.g4
[8]https://github.com/herminiogg/ShExML/blob/master/src/main/scala/com/herminiogarcia/shexml/parser/ASTCreatorVisitor.scala
[9]https://github.com/herminiogg/ShExML/blob/master/src/main/scala/com/herminiogarcia/shexml/ast/AST.scala
[10]https://github.com/herminiogg/ShExML/blob/master/src/main/scala/com/herminiogarcia/shexml/visitor/RDFGeneratorVisitor.scala

```
PREFIX : <http://example.com/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX schema: <http://schema.org/>
SOURCE films_xml_file <https://shexml.herminiogarcia.com/files/films.xml>
ITERATOR film_xml <xpath: //film> {
    FIELD id <@id>
    FIELD name <name>
    ITERATOR actors <cast/node()[self::actor or self::actress]> {
        FIELD name <name>
    }
}

EXPRESSION films <films_xml_file.film_xml>

:Films :[films.id] {
    schema:name [films.name] ;
    :cast [films.actors.name] ;
}
```

Listing 1 Example ShExML file with a root iterator and a nested iterator with one field per iterator.

- *filterRelatedResults(POR)*: Filters the results from the predicate-object tuples that correspond with the subject results. For that if the id or one of the root ids (i.e., the roots of the upper iterator queries executed to reach the current one) of the predicate-object result matches the id of the subject result then the result is included. If the predicate-object result id does not exist and there are no root ids then it is also included as it concerns the generation of a literal value not an action.
- *composeIterationQuery(riq)*: Given the partial queries, this function composes an iterator query based on the used query language (i.e., JSONPath and XPath) which contains placeholders in the form of [*] for substitution in the next step. An example of expected result from this process can be seen in Listing 2 for the ShExML input defined in Listing 1.
- *generateAllQueries(iq)*: This function receives an iterator query, like the ones defined in Listing 2, and generates all the possible queries given the result of the partial queries evaluation. See Listing 3 for an example of the queries executed for the ShExML example in Listing 1.

Data: AST as AST, symbol table as ST
Result: set of triples as T
$T \leftarrow \emptyset$;
$S \leftarrow$ all shapes $\in AST$;
foreach *shape* $s \in S$ **do**
    $sa \leftarrow$ subject of s;
    $POA \leftarrow$ all predicate and object tuples of s;
    $SR \leftarrow resolveAction(sa)$;
    $POR \leftarrow \emptyset$;
    foreach *predicate and object tuple* $(pa, oa) \in POA$ **do**
        $OR \leftarrow resolveAction(oa, ST)$;
        $POR \leftarrow POR \cup (pa, OR)$;
    end
    foreach *subject result* $sr \in SR$ **do**
        $PORF \leftarrow filterRelatedResults(POR)$;
        foreach *filtered predicate-object result tuple* $(pa, or) \in PORF$ **do**
            $T \leftarrow T \cup (sr, pa, or)$;
        end
    end
end
return T;

Algorithm 1: General algorithm for the generation of RDF

Data: Action as a, symbol table as ST
Result: Result as R
$R \leftarrow \emptyset$;
$exp \leftarrow search(a, ST)$;
$EXP \leftarrow split(ext, ``.")$;
file as $f \leftarrow search(EXP[0], ST)$;
root iterator query as $riq \leftarrow search(EXP[1], ST)$;
tail queries as $TE \leftarrow \emptyset$;
foreach *tail expression* $te \in EXP[2:]$ **do**
    $tq \leftarrow search(te, ST)$;
    $TE \leftarrow TE \cup tq$;
end
$iq \leftarrow composeIterationQuery(riq)$;
$AQ \leftarrow generateAllQueries(iq)$;
foreach *query as q, iterator query as iq, index as i, root ids as RID* $\in AQ$ **do**
    $r \leftarrow performQuery(q, file)$;
    $id \leftarrow iq \cup i \cup file$;
    $R \leftarrow R \cup (r, id, RID)$;
end
return R;

Algorithm 2: Algorithm to evaluate the expressions and perform the queries, used in Algorithm 1 as *resolveAction(a)*

```
# id field iterator query
//film[*]/@id

# name field iterator query
//film[*]/name

# actors and actresses name field iterator query
//film[*]/cast/node()[self::actor or self::actress][*]/name
```

Listing 2 Iterator queries composed by the engine when executing the mapping rules contained in Listing 1.

```
# id field extraction
//film[1]/@id
//film[2]/@id

# name field extraction
//film[1]/name
//film[2]/name

# actors and actresses name field extraction
//film[1]/cast/node()[self::actor or self::actress][1]/name
//film[1]/cast/node()[self::actor or self::actress][2]/name
//film[1]/cast/node()[self::actor or self::actress][3]/name
//film[1]/cast/node()[self::actor or self::actress][4]/name
//film[1]/cast/node()[self::actor or self::actress][5]/name
//film[2]/cast/node()[self::actor or self::actress][1]/name
//film[2]/cast/node()[self::actor or self::actress][2]/name
//film[2]/cast/node()[self::actor or self::actress][3]/name
//film[2]/cast/node()[self::actor or self::actress][4]/name
//film[2]/cast/node()[self::actor or self::actress][5]/name
```

Listing 3 Final queries to be executed after expanding the iterator queries listed in Listing 2.

4. Profiling the ShExML engine and performance improvements

For discovering the bottlenecks of the ShExML engine and allowing to contrast the performance improvements, a profiling methodology was used. Then, the different improvements are introduced per version allowing for a modular assesment of their overall effect in the engine performance.

For the profiling methodology, the Java VisualVM[11] tool was used. This allowed for monitoring the different methods inside the engine and assessing their time consumption in relation with the whole execution time. The followed process consisted in starting the application to then activate the sampler on the CPU which allows to generate a faster overview than profiling the whole application and it was sufficient for the purpose of this work. However, as this process can take some seconds to set up I included a general delay of 20 seconds in the main method of the ShExML engine in order to ensure the capture of the relevant data. Afterwards, the method call tree was analysed in order to seek for the possible bottlenecks or methods that were taking more time than what was deemed appropriate.[12] This process was repeated through the different improvements and versions as once one specific bottleneck is solved, another one may become more evident than before. Finally, this process delivered a set of improvements distributed across the following ShExML engine versions:

Version 0.3.3

This version started the journey on performance improvements by adding a cache mechanism over two main features that were unnecessarily consuming a lot of time.[13] Namely, it was detected that the files were downloaded many times incurring in very costly IO operations due to the functional-based design of the algorithm. Therefore, the

[11]https://visualvm.github.io/
[12]The profiler results can be consulted on: https://github.com/herminiogg/shexml-performance-evaluation/tree/main/profiler-snapshots
[13]See the changelog: https://github.com/herminiogg/ShExML/compare/v0.3.2...v0.3.3

retrieval method was redesigned to cache the files contents, only retrieving them for the first time. This improvement was really evident when the IO operations involved downloading files over the internet. Similarly, and due to the algorithm design explained in Section 3, some of the queries were executed more than once creating an unnecessary time consumption, moreover in the cases of hierarchical data as explained before. Therefore, a cache system was also put into place for JSONPath and XPath queries.

Version 0.4.0

After the improvements on v0.3.3, other bottlenecks became more evident.[14] Firstly, the algorithm was naively using the file contents for the id generation — processed afterwards inside a hashcode function. This was not an evident bottleneck on small inputs, however, the longer the input the more this issue was a huge bottleneck. Thus, in order to improve the id generation function the file content was substituted by the file path. Besides, some functions were changed for more performant ones like `withFilter`[15] instead of `filter` and changed some used `List` collections for more performant equivalent ones according to their use case inside the algorithm.[16] However, the main improvement in this version is linked to the change of the filtering function, in charge of correlating the predicate-object results to the subject results, by a grouping function that distributes each result by its id and root ids. Ultimately, this changes the time complexity from a quadratic time $O(n^2)$ to a linear time $O(n)$. In addition, a cache system was introduced for the JSON parser in order to avoid parsing big files multiple times.

Version 0.4.2

This version includes some minor improvements on performance that were more easily identifiable after changing the filter function.[17] Since the very beginning ShExML incorporates a system for inferencing data types based on the obtained result [10] and a URI normalisation system that ensures that the engine generates compliant URIs. As these are not standard features across languages and they can affect the performance, it was deemed to make them optional letting users decide whether they need these systems.

Version 0.5.1

Once all the performance bottlenecks pertaining to the algorithm and implementation details were solved, the sampling results reflected that most of the time was used on executing the queries using the respective libraries. Even though this was totally out of the algorithm design and implementation details, the execution times seemed excessively high, even more in the case of XML files. In previous versions the engine was using kantan-xpath[18] and gatling-jsonpath[19], both libraries written in Scala and designed according to Scala idioms. For trying to improve the performance, the gatling-jsonpath library was substituted by Jayway JsonPath[20] already in use by other declarative mapping rules engines. In the case of XPath, firstly I tried to use the Javax Xpath API which delivered some improvements but very far from those of JSONPath. Finally, the Saxon-HE[21] library was incorporated delivering much better results. Apart from that, the VTD-XML[22] was also considered due to its very good performance results (potentially better than Saxon-HE), unfortunately it was not possible to incorporate this library in the ShExML engine due to a license collision. As it was done with the JSON parser in v0.4.0, a cache system was introduced in this version for the XML parser.[23]

5. Evaluation

In order to demonstrate the effectiveness of the analysis made and the consequent performance improvements, an experiment was conducted over the enlisted versions.

[14] See the changelog: https://github.com/herminiogg/ShExML/compare/v0.3.3...v0.4.0
[15] https://www.scala-lang.org/api/current/scala/collection/ArrayOps$$WithFilter.html
[16] For collections performance consult: https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html
[17] See the changelog: https://github.com/herminiogg/ShExML/compare/v0.4.0...v0.4.2
[18] https://nrinaudo.github.io/kantan.xpath/
[19] https://github.com/gatling/jsonpath
[20] https://github.com/json-path/JsonPath
[21] https://github.com/Saxonica/Saxon-HE
[22] https://vtd-xml.sourceforge.io/
[23] See the changelog: https://github.com/herminiogg/ShExML/compare/v0.4.2...v0.5.1

Input	Engine	n	$\bar{x}$	$\tilde{x}$	s	*min*	*max*
JSON Films 1000 entries	ShExML-v0.3.2.jar	30	30488.20000	30572.00000	2015.11590	26696.00000	34699.00000
	ShExML-v0.3.3.jar	30	19220.23333	18984.50000	1190.45815	17780.00000	22109.00000
	ShExML-v0.4.0.jar	30	2888.56667	2823.00000	202.72721	2670.00000	3418.00000
	ShExML-v0.4.2.jar	30	2935.83333	2843.00000	583.90600	2496.00000	5740.00000
	ShExML-v0.5.1.jar	30	2591.50000	2527.00000	242.94894	2271.00000	3220.00000
XML Films 1000 entries	ShExML-v0.3.2.jar	30	107352.33333	106644.50000	5239.70542	100762.00000	117625.00000
	ShExML-v0.3.3.jar	30	26724.26667	26358.00000	690.67777	26144.00000	28271.00000
	ShExML-v0.4.0.jar	30	22787.36667	22858.00000	1463.50730	20980.00000	26207.00000
	ShExML-v0.4.2.jar	30	21596.10000	21195.00000	906.07613	20770.00000	24324.00000
	ShExML-v0.5.1.jar	30	4867.26667	4825.50000	244.74800	4663.00000	5953.00000
EHRI institutions (JSON)	ShExML-v0.3.2.jar	1	2188032	2188032	0	2188032	2188032
	ShExML-v0.3.3.jar	30	157217.16667	154946.00000	5966.55622	149673.00000	171430.00000
	ShExML-v0.4.0.jar	30	7434.16667	7376.50000	302.49162	6757.00000	8132.00000
	ShExML-v0.4.2.jar	30	6228.93333	6230.50000	138.71279	5997.00000	6611.00000
	ShExML-v0.5.1.jar	30	5601.83333	5623.00000	197.05663	5283.00000	6124.00000

Table 1

This table shows the descriptive statistics for the execution time (measured in milliseconds) of the tested engine for each of the three inputs where n is the size of the sample, $\bar{x}$ is the mean, $\tilde{x}$ is the median, s is the standard deviation, *max* is the maximum value of the sample, and *min* is the minimum value of the sample.

5.1. Methodology

This methodology is based upon the one introduced in the SPARQL-Anything evaluation paper [7] but with some additions and changes to better assess the differences between the different engine versions.[24] Therefore, the experiment was composed of three different inputs: (1) a set of mapping rules for a single film shape encoded in a flat JSON file with 1000 entries, (2) another set for a single film shape encoded in a flat XML file containing 1000 entries, (3) a set of mapping rules used in a real case scenario [11] which converts multiple files in JSON format extracted from the EHRI project[25] with a deeper hierarchical structure. This combination should provide a double insight on the performance delivered by the engines, both in a synthetic environment and a real one. Each engine was run against each input 30 times in order to have a sample which can be considered statistically significant. The v0.3.2 was used as the baseline version and the others described versions containing performance improvements (i.e., v0.3.3, v0.4.0, v0.4.2 and v0.5.1) were contrasted between them and against the baseline method.

5.2. Results

The described methodology was executed using the Windows Subsystem for Linux under a steady Windows 11 environment in an 11th Generation Intel i7 at 2.80 GHz, with 16GB of RAM. The Java environment consisted on the OpenJDK Runtime Environment 17.0.9 (build 17.0.9+9).

The results of this experiment and the statistical analysis are openly available on Github.[26] For the statistical analysis R on its version 4.3.1 was used. The descriptive statistics for each of the inputs and engines can be consulted in Table 1.

With the aim to assess the differences between the engines in a statistically-significant manner, a statistical hypothesis testing was conducted. As not all the distributions follow a normal distribution upon Spahiro-Wilk testing, the Kruskal-Wallis test was used delivering the following results: ($H(4) = 123.41, p < .001$) for the JSON films, ($H(4) = 137.79, p < .001$) for the XML films and ($H(3) = 111.01, p < .001$) for the EHRI institutions case. These results demonstrate that there are significant differences between the engines for almost all the given inputs

[24]The resources and the raw results for this experiment can be found on https://github.com/herminiogg/shexml-performance-evaluation and under the following persistent DOI https://zenodo.org/doi/10.5281/zenodo.10890036

[25]https://www.ehri-project.eu/

[26]https://github.com/herminiogg/shexml-performance-evaluation/tree/main/statistics

Input	Comparison	p	r
JSON Films 1000 entries	ShExML-v0.3.2.jar - ShExML-v0.3.3.jar	< .05	.345
	ShExML-v0.3.3.jar - ShExML-v0.4.0.jar	< .001	.555
	ShExML-v0.4.0.jar - ShExML-v0.4.2.jar	.695	.0506
	ShExML-v0.4.2.jar - ShExML-v0.5.1.jar	< .05	.307
XML Films 1000 entries	ShExML-v0.3.2.jar - ShExML-v0.3.3.jar	< .05	.347
	ShExML-v0.3.3.jar - ShExML-v0.4.0.jar	< .05	.422
	ShExML-v0.4.0.jar - ShExML-v0.4.2.jar	.149	.186
	ShExML-v0.4.2.jar - ShExML-v0.5.1.jar	< .05	.425
EHRI institutions (JSON)	ShExML-v0.3.3.jar - ShExML-v0.4.0.jar	< .05	.431
	ShExML-v0.4.0.jar - ShExML-v0.4.2.jar	< .05	.437
	ShExML-v0.4.2.jar - ShExML-v0.5.1.jar	< .05	.420

Table 2

This table shows the results for the post hoc tests (p-value as p) and the effect size (as r) for each of the comparisons. Only the comparisons between consecutive versions are preserved in this table.

highlighting that the performance improvement has been achieved successfully. However, in order to analyse the different improvements and their degree of influence on the performance, the post hoc tests were run and the effect sizes calculated which can be consulted in Table 2. For the p-values the conventional significance levels are used ($p < .05$ significant differences and $p < .001$ very significant differences) and for the effect size, the Cohen's r suggested thresholds [12] (above .50 is considered a large effect size, above .30 a medium effect size and below .30 a small effect size.) This comparison between engines shows that there are significant differences between v0.3.2 and v0.3.3 for the three inputs with a medium effect size. Between v0.3.3 and v0.4.0 there are significant differences and a medium effect size for the XML and the institution entries, but the JSON input shows very significant differences and a large effect size. When comparing v0.4.0 and v0.4.2 only the institutions entry shows significant differences and a medium effect size. Finally, there are significant differences and a medium effect size between the v0.4.2 and the v0.5.1 for the three inputs. A closer look to the distributions allows to further explain these differences or lack of them. In the case of the JSON films entry the distributions (as it can be seen in Fig. 2) for the three most performant versions are quite close and they overlap among them, therefore explaining the lack of significant differences between v0.4.0 and v0.4.2 and the effect size very close to a small one between v0.4.2 and v0.5.1. For the XML input (represented in Fig. 3) the v0.5.1 is more performant but in the case of v0.4.2 and v0.4.0 (which do not present significant differences) both of them tend to have less performant iterations that fall under the v0.3.3 distribution values. Nevertheless, v0.4.2 shows a more unified shape suggesting a fairly improvement on performance over v0.4.0 (even though not quite appreciable.) Finally, when the EHRI institutions input is analysed (see Fig. 4) the three most performant versions (v0.5.1, v0.4.2 and v0.4.0) show very defined and almost not overlapping distributions.

5.3. Discussion

In the light of these results, I analyse the effects that each of the introduced modifications across the ShExML engine versions have in the overall performance given the three cases under study.

Firstly, the alleviation of IO operations and caching some of the iteration queries (from v0.3.2 to v.0.3.3) resulted consequently in a very good performance improvement in bigger inputs, like the EHRI institutions one. Moreover, when having a lot hierarchical information, caching these iteration queries (that tend to be repeated) also suposses a decent improvement over not doing so. However, this improvement might seem more limited when the inputs are not very long nor they contain hierarchical data, as it is the case for the JSON and XML films inputs.

As already anticipated earlier, the transition from v0.3.3 to v0.3.4 introduced a great deal of improvement coming mainly from the changes on the calculation of ids and the enhanced filtering function. Thus, this has translated to a significant and steady improvement over all kinds of inputs. The bigger effect size on the JSON films entries can be explained by the JSON parser cache also introduced in this version. Conversely, the EHRI institutions case which also uses an input in the JSON format, deals with many files in comparison with a single file entry in the films case.

Fig. 2. Plot of the three most performant engines distributions against the input JSON Films 1000 entries.

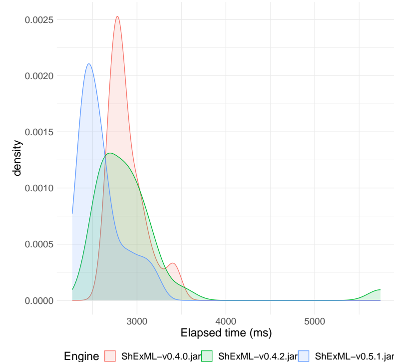


Fig. 3. Plot of the three closest (on performance) engines distributions against the input XML Films 1000 entries.

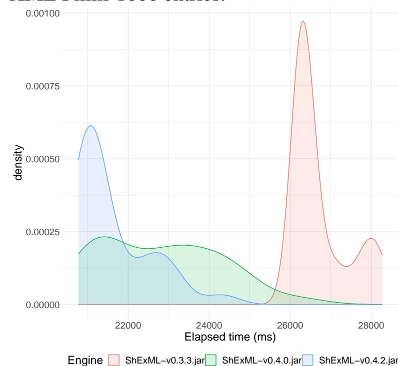
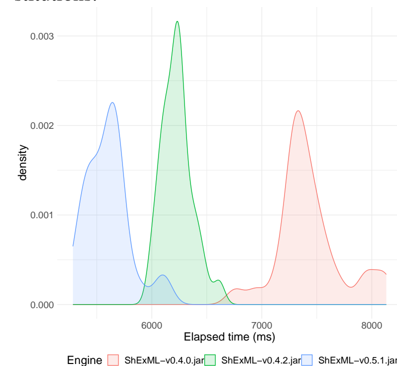


Fig. 4. Plot of the three most performant engines distributions against the input EHRI institutions.



Making the data types inference and URI normalisation systems optional did not have a very big impact on smaller inputs (like both films cases) but as the result of the EHRI institutions entry demonstrate, it can have a bigger impact when longer inputs are used. At the same time, from a usability perspective, having these system enabled by default can cause some confusion to users, so making them optional seemed a good alternative both from the performance and usability points of view.

Finally, changing the XPath and JSONPath libraries for more performant alternatives drove to a better response time in the ShExML engine. When other bottleneck problems are resolved the main core of these data mapping engines relies in the query system which is normally delegated to an external library. Therefore, a wise choice in this regard can have a great impact on the engines performance. Besides, the introduction of the XML document parser results cache system seems to improve the general XML processing time even for smaller inputs.

6. Future work

While a lot of improvements have been introduced with this work in the ShExML engine performance, there are still some points of improvement. The main aspect of improvement would be to allow for executing the engine in a parallel fashion. As introduced earlier, some parts of the engine are not thread-safe (e.g., the cache systems) so it would be necessary to change their implementation for their thread-safe counterpart. Similarly, it should be studied in which parts of the engine code the foreseen parallelisation could provide an improvement on the performance taking into account the additional overheads of managing different threads within an application.

Regarding the benchmark, I envisage the application of this methodology to evaluate the improvements performed in the ShExML engine against other declarative mapping rules engines to spot the current situation and differences among them. In this sense, the idea is to run a replication study over the results obtained by [7] but adding features present in my study, including the statistical analysis, different input formats, and the involvement of a hierarchical input. This should provide a better picture of mapping rules engines capabilities in relation to multiple — and heterogeneously-shaped — data.

7. Conclusions

In this work the optimisation of the ShExML engine has been tackled using a profiling methodology as a means for detecting the bottlenecks and verifying the delivered enhancements. In order to corroborate the success of this methodology, the consequent optimised versions of the ShExML engine were tested and analysed using a performance evaluation whose results were analysed using statistical methods. The contrast of the statistical analysis against the introduced improvements gives a set of watch points over the optimisation of such engines which can be wide-applicable for other practitioners whose engines suffer from similar problems. Moreover, as a result of this work ShExML users can now benefit from a much faster and reliable engine.

Funding

This work has been carried out in the context of the EHRI-3 project funded by the European Commission under the call H2020-INFRAIA-2018-2020, with grant agreement ID 871111 and DOI 10.3030/871111.

References

[1] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, B. De Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga and A. Dimou, The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF, in: *The Semantic Web – ISWC 2023*, T.R. Payne, V. Presutti, G. Qi, M. Poveda-Villalón, G. Stoilos, L. Hollink, Z. Kaoudi, G. Cheng and J. Li, eds, Springer Nature Switzerland, Cham, 2023, pp. 152–175.

[2] A. Dimou, M.V. Sande, P. Colpaert, R. Verborgh, E. Mannens and R.V. de Walle, RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, in: *Proceedings of the Workshop on Linked Data on the Web co-located with the 23rd International World Wide Web Conference (WWW 2014), Seoul, Korea, April 8, 2014.*, 2014.

[3] A. Iglesias-Molina, A. Cimmino, E. Ruckhaus, D. Chaves-Fraga, R. García-Castro and O. Corcho, An ontological approach for representing declarative mapping languages, *Semantic Web* (2022), 1–31.

[4] H. García-González, I. Boneva, S. Staworko, J.E.L. Gayo and J.M.C. Lovelle, ShExML: improving the usability of heterogeneous data mapping languages for first-time users, *PeerJ Compututer Science* **6** (2020), e318. doi:10.7717/peerj-cs.318.

[5] J. Arenas-Guerrero, M. Scrocca, A. Iglesias-Molina, J. Toledo, L. Pozo-Gilo, D. Doña, Ó. Corcho and D. Chaves-Fraga, Knowledge Graph Construction with R2RML and RML: An ETL System-based Overview, in: *Proceedings of the 2nd International Workshop on Knowledge Graph Construction co-located with 18th Extended Semantic Web Conference (ESWC 2021), Online, June 6, 2021*, D. Chaves-Fraga, A. Dimou, P. Heyvaert, F. Priyatna and J.F. Sequeda, eds, CEUR Workshop Proceedings, Vol. 2873, CEUR-WS.org, 2021. https://ceur-ws.org/Vol-2873/paper11.pdf.

[6] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, D. Collarana and M.-E. Vidal, SDM-RDFizer: An RML interpreter for the efficient creation of RDF knowledge graphs, in: *Proceedings of the 29th ACM international conference on Information & Knowledge Management*, 2020, pp. 3039–3046.

[7] L. Asprino, E. Daga, A. Gangemi and P. Mulholland, Knowledge Graph Construction with a façade: a unified method to access heterogeneous data sources on the Web, *ACM Transactions on Internet Technology* **23**(1) (2023), 1–31.

[8] T. Parr, The definitive ANTLR 4 reference, *The Definitive ANTLR 4 Reference* (2013), 1–326.

[9] H. García-González and A. Dimou, Why to tie to a single data mapping language? enabling a transformation from shexml to rml, in: *Proceedings of Poster and Demo Track and Workshop Track of the 18th International Conference on Semantic Systems co-located with 18th International Conference on Semantic Systems (SEMANTiCS 2022)*, Vol. 3235, 2022, p. paper–11.

[10] H. García-González, A ShExML Perspective on Mapping Challenges: Already Solved Ones, Language Modifications and Future Required Actions, in: *Proceedings of the 2nd International Workshop on Knowledge Graph Construction co-located with 18th Extended Semantic Web Conference (ESWC 2021), Online, June 6, 2021*, D. Chaves-Fraga, A. Dimou, P. Heyvaert, F. Priyatna and J.F. Sequeda, eds, CEUR Workshop Proceedings, Vol. 2873, CEUR-WS.org, 2021. http://ceur-ws.org/Vol-2873/paper2.pdf.

[11] H. García-González and M. Bryant, The Holocaust Archival Material Knowledge Graph, in: *The Semantic Web – ISWC 2023*, T.R. Payne, V. Presutti, G. Qi, M. Poveda-Villalón, G. Stoilos, L. Hollink, Z. Kaoudi, G. Cheng and J. Li, eds, Springer Nature Switzerland, Cham, 2023, pp. 362–379. ISBN 978-3-031-47243-5.

[12] J. Cohen, A power primer., *Psychological Bulletin* **112(1)** (1992), 155–159. doi:10.1037/0033-2909.112.1.155.

[13] D. Van Assche, T. Delva, G. Haesendonck, P. Heyvaert, B. De Meester and A. Dimou, Declarative RDF graph generation from heterogeneous (semi-) structured data: A systematic literature review, *Journal of Web Semantics* **75** (2023), 100753.

[14] M. Lefrançois, A. Zimmermann and N. Bakerally, A SPARQL extension for generating RDF from heterogeneous formats, in: *The Semantic Web: 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28–June 1, 2017, Proceedings, Part I 14*, Springer, 2017, pp. 35–50.

[15] G. Haesendonck, W. Maroy, P. Heyvaert, R. Verborgh and A. Dimou, Parallel RDF generation from heterogeneous big data, in: *Proceedings of the International Workshop on Semantic Big Data*, 2019, pp. 1–6.

[16] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus and O. Corcho, GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain, *Journal of Web Semantics* **65** (2020), 100596. doi:https://doi.org/10.1016/j.websem.2020.100596. https://www.sciencedirect.com/science/article/pii/S1570826820300354.

[17] M. Scrocca, A. Carenini, M. Comerio and I. Celino, Semantic Conversion of Transport Data Adopting Declarative Mappings: an Evaluation of Performance and Scalability, in: *Third International Workshop On Semantics And The Web For Transport*, 2021.

[18] C. Stadler, L. Bühmann, L.-P. Meyer and M. Martin, Scaling RML and SPARQL-based Knowledge Graph Construction with Apache Spark (2023).

[19] S. Bin, C. Stadler and L. Bühmann, KGCW2023 Challenge Report: RDFProcessingToolkit/Sansa, in: *Proceedings of the 4th International Workshop on Knowledge Graph Construction*, 2023.

[20] E. Iglesias and V. Maria-Esther, Knowledge Graph Creation Challenge: Results for SDM-RDFizer, in: *Proceedings of the 4th International Workshop on Knowledge Graph Construction*, 2023.