Optimising the ShExML engine through code profiling: from turtle's pace to state-of-the-art performance

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

Editor: Guest Editors KG Construction 2024
Solicited reviews: Jakub Klimek, Charles University, Czechia; Nuno Lopes, TopQuadrant, United Kingdom; Anonymous reviewer; Ana Iglesias-Molina, Universidad Politécnica de Madrid, Spain

Abstract. The ShExML language was born as a more user-friendly approach for knowledge graph construction. However, a recent study has 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 code profiling analysis. 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. Moreover, the optimised version of ShExML is compared against similar engines, delivering a comparable performance to its alternatives. 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 [1]. 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 [2], starting with RML [3].

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 [1, 4]. 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 [5]. However, unlike other languages and specifications, ShExML only counts on one compliant engine[2] which can limit the possibilities for further optimisations [6, 7]. This aspect has been revealed in the recent SPARQL-Anything performance evaluation which shows how the ShExML engine performs drastically worse than other competitors [8].

Building on this previous research this work defines the following research questions:

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

- RQ1: Is it possible to improve the performance of the ShExML engine, and if yes, what optimisations have a significant impact on the overall performance?
- RQ2: How the optimised ShExML engine stands with respect to other similar engines?
- RQ3: What impact do these improvements have on the the CPU and RAM usage?

Thus, 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] 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. Moreover, a replication study of the SPARQL-Anything experiment over the optimised version, and other mapping languages engines coetaneous versions, is conducted in order to reveal how the optimised version of ShExML stands in comparison with other similar engines. The results yielded in this study could serve other practitioners who seek to enhance their engines or their knowledge graph construction workflows.

The rest of the paper is structured as follows: in Section 2 the related work is presented, in Section 3 the ShExML language is briefly introduced, Section 4 presents the ShExML engine and how it works, while in Section 5 the introduced improvements are presented. Section 6 explains the experiments, and exposes and discusses the results. Finally, the future lines of work upon this paper are drafted in Section 7, and conclusions of this work are drawn in Section 8.

2. Related work

Some previous works have tackled the performance comparison of different mapping languages taking one specific engine implementation as the preferred one for establishing this comparison. This is specially true in the case of RML for which many different engines provide an implementation [1, 6], however, existing inter-language comparisons have mainly employed the RMLMapper engine[4]. For example, upon its launch, SPARQL-Generate was compared against RML for a set of size-increasing CSV documents [9]. More recently, SPARQL-Anything was launched, comparing it against other existing declarative mapping languages (SPARQL-Generate, RML and ShExML) using two different sets for the evaluation: 4 different JSON files and an increasing in size input in JSON [8]. Upon the delivered results it was demonstrated that the ShExML engine had serious performance problems.

While inter-languages comparisons are still in an incipient state, the emergence of many implementations for the RML language has made that intra-RML implementations comparisons are far more numerous, for example, in [7] SDM-RDFizer was compared against the RMLMapper and RocketRML engines or in [10] where the rmlmapper-node-js engine was compared to the legacy RML-Mapper[5] and the current RMLMapper. Similarly, in [11] the authors compare a parallel-ready version of RML, RMLStreamer, against a set of size-increasing inputs in CSV, XML and JSON formats.

In an effort to bring some cohesion to this field, GTFS-Madrid-Bench, a benchmark based on transport data, was proposed [12] which has led to different evaluations using it (e.g., [6, 13, 14]) and even a challenge[6] in a well-established workshop in the field, leading to more consequent evaluations (e.g., [15, 16]).

Recent optimisations have also been introduced and evaluated in some of the existing RML engines. SDM-RDFizer has improved its performance by introducing a reordering of mapping rules for a prioritised evaluation, alongside a compression technique that avoids the generation of duplicates [17]. In [18] the authors propose a mapping partition algorithm which is able to determine the best execution order for a set of mapping assertions, leading to an overall improve in the tested RML engines. A similar approach, using mapping partitions, has also

[3] https://github.com/herminiogg/ShExML/releases

[4] https://github.com/RMLio/rmlmapper-java

[5] https://github.com/RMLio/RML-Mapper

[6] The challenge counts to date with two editions:
2023: https://kg-construct.github.io/workshop/2023/#call
2024: https://kg-construct.github.io/workshop/2024/#call

been explored for the Morph-KGC engine, improving its execution time, decreasing the peak memory usage and surpassing other engines maximum processable data sizes [19].

However, a common issue of these evaluations is that all of them use a flat structure for the input files – stemming from the specific R2RML and RML tabular-oriented algorithm – 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. Introduction to the ShExML language

ShExML is a language designed for the integration of heterogeneous data sources into a single RDF output, taking special attention to its usability in an effort for improving the users' productivity [5]. The ShExML language specification[7] introduces two different concerns: how to extract data from heterogeneous data sources and how to load data into RDF. For the latter purpose, the syntax is based upon the Shape Expressions (ShEx) [20] one, adapting a subset of it to the specific task of integrating data. On its turn, the extraction of data is defined on the declarations part which follows a similar syntax style to prefixes in ShEx or SPARQL, but defining a new set of directives for the definition of the input files in order to: extract (relying in other query languages like: XPath, JSONPath, SQL and SPARQL), iterate, merge and transform the input data. This separation of concerns is intentionally enforced in the language to allow users to focus in one part, or the other, while defining their mappings without the need to conciliate both concerns under the same syntax constructions. This design choice, at the same time, allows for the use of a single set of shapes independently of the amount of input data sources, which enhances the readability and maintainability of the overall mapping solution.

Listing 3 shows an example ShExML file in which two input data sources (one in XML and defined in Listing 1 and the other in JSON and represented in Listing 2) are consolidated into a single RDF representation (see Listing 4). Following this example, the mapping file starts with the general prefixes definition (like other Semantic Web technologies syntaxes) to go forthwith to the declaration of the two main input files. Then, two iterators are defined, one for each format, which define how the data will be extracted from the sources. The main iterator defines the query language to be used, then the nested fields perform the extraction of the data, while the nested iterators continue the exploration process deeper in the hierarchical files (see Section 4.2 for a more detailed explanation of the query rewriting algorithm). It is worth noting that in the case of having multiple files under the same data format but with different data structures, different iterators are needed. Additionally, non-hierarchical data sources, like CSV files or relational databases do not require the definition of nested iterators. To conclude the declarations part, a expression defines a `films` variable holding the results of combining the two previously defined iterators, applied over the two input sources.

The shapes part defines how the data will be generated into RDF triples. As stated before, the ShEx syntax is borrowed to define these constructions but it has been adapted to the specific case of integrating data, therefore, one of the main changes resides in the substitution of the subject and object node constraints for generation expressions. A complete shape definition will contain a shape name, a subject generation and a set of predicate-object expressions which, at the same time, are divided into a predicate and an object generation expression. These generation expressions are constituted by an optional prefix definition (which will define if the node should be a literal or an IRI and it is, therefore, mandatory for subjects generation) and a variable navigation path (using the `.` symbol as the navigation operand). As an example, a user can define the path `films.actors.name` to retrieve the names of all the actors in both files for which the engine will resolve `films` to the defined expression, `actors` as the

[7] A draft specification of the language can be consulted on https://shexml.herminiogarcia.com/spec/

```
<films>
 <film id="1">
   <name>Dunkirk</name>
    <year>2017</year>
    <country>USA</country>
    <cast>
       <actor>
          <name>Fionn Whitehead</name>
          <role>Tommy</role>
       </actor>
       <actor>
          <name>Damien Bonnard</name>
          <role>French soldier</role>
       </actor>
    </cast>
 </film>
 <film id="2">
    <name>Interstellar</name>
    <year>2014</year>
    <country>USA</country>
    <cast>
       <actress>
          <name>Ellen Burstyn</name>
          <role>Murph (Older)</role>
       </actress>
       <actor>
          <name>Matthew McConaughey</name>
          <role>Cooper</role>
       </actor>
    </cast>
 </film>
</films>
```

Listing 1 Example of an input XML file containing data about films.

```
{
 "films": [
    {
       "id": 3,
       "name": "Inception",
       "year": "2010",
       "country": "USA",
       "cast": [
          {
             "name": "Leonardo DiCaprio",
             "role": "Cobb"
          },
          {
             "name": "Joseph Gordon-Levitt",
             "role": "Arthur"
          }
       ]
    },
    {
       "id": 4,
       "name": "The Prestige",
       "year": "2006",
       "country": "USA",
       "cast": [
          {
             "name": "Hugh Jackman",
             "role": "Robert Angier"
          },
          {
             "name": "Christian Bale",
             "role": "Alfred Borden"
          }
       ]
    }
 ]
}
```

Listing 2 Example of an input JSON file containing data about films.

nested iterators defined for JSON and XML, and finally `name` as the field to be extracted. Based on this substitution process, the engine will further compose the specific queries and retrieve the data from the defined sources as it is further detailed in Section 4.2. As a modularity affordance, and following the ShEx syntax, instead of an object generation, the user can define a link to another shape using its name. This will trigger the generation of triples from the linked shape whose subject IRI will be used as the object IRI in the invoking predicate-object expression.

Unlike other mapping languages, the ShExML specification is only implemented by a single engine whose optimisation is the main topic of this paper. Nevertheless, new research is being conducted around the topics of intermediate mapping representations [4, 21] and mapping languages translation [22, 23] which could potentially build crosswalks between different mapping languages and engines, facilitating the adoption of these tools by more users and the jointly improvement of languages and engines alike.

4. ShExML engine

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.

4.1. 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 [24]. In essence, it defines a pipeline in which the output

```
PREFIX : <http://example.com/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX schema: <http://schema.org/>
SOURCE films_xml_file <films.xml>
SOURCE films_json_file <films.json>
ITERATOR film_xml <xpath: //film> {
   FIELD id <@id>
   FIELD name <name>
   FIELD year <year>
   FIELD country <country>
   ITERATOR actors <cast/actor> {
      FIELD name <name>
      FIELD role <role>
   }
   ITERATOR actresses <cast/actress> {
      FIELD name <name>
      FIELD role <role>
   }
}
ITERATOR film_json <jsonpath: $.films[*]> {
   FIELD id <id>
   FIELD name <name>
   FIELD year <year>
   FIELD country <country>
   ITERATOR actors <cast[*]> {
      FIELD name <name>
      FIELD role <role>
   }
}
EXPRESSION films <films_xml_file.film_xml
      UNION films_json_file.film_json>

:Films :[films.id] {
   schema:name [films.name] ;
   :year dbr:[films.year] ;
   schema:countryOfOrigin dbr:[films.country] ;
   schema:actor @:Actor ;
   schema:actor @:Actress ;
}

:Actor dbr:[films.actors.name] {
   schema:name [films.actors.name] ;
   :role [films.actors.role] ;
}

:Actress dbr:[films.actresses.name] {
   schema:name [films.actresses.name] ;
   :role [films.actresses.role] ;
}
```

Listing 3 Example of a ShExML input which integrates the data from the XML file in Listing 1 and the JSON file in Listing 2 into the RDF representation shown in Listing 4.

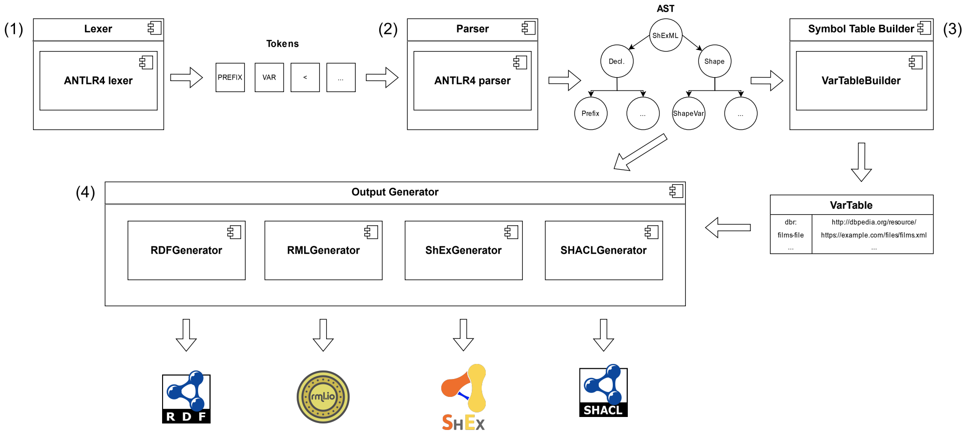
```
@prefix :  <http://example.com/> .
@prefix schema: <http://schema.org/> .
@prefix dbr: <http://dbpedia.org/resource/> .

:1  :year              dbr:2017 ;
    schema:countryOfOrigin dbr:USA ;
    schema:name        "Dunkirk" ;
    schema:actor       dbr:Fionn_Whitehead ,
                       dbr:Damien_Bonnard .

:2  :year              dbr:2014 ;
    schema:countryOfOrigin dbr:USA ;
    schema:name        "Interstellar" ;
    schema:actor       dbr:Matthew_McConaughey ,
                       dbr:Ellen_Burstyn .

:3  :year              dbr:2010 ;
    schema:countryOfOrigin dbr:USA ;
    schema:name        "Inception" ;
    schema:actor       dbr:Leonardo_DiCaprio ,
                       dbr:Joseph_Gordon-Levitt .

:4  :year              dbr:2006 ;
    schema:countryOfOrigin dbr:USA ;
    schema:name        "The Prestige" ;
    schema:actor       dbr:Hugh_Jackman ,
                       dbr:Christian_Bale .

dbr:Christian_Bale :role "Alfred Borden" ;
      schema:name "Christian Bale" .

dbr:Damien_Bonnard :role "French soldier" ;
      schema:name "Damien Bonnard" .

dbr:Fionn_Whitehead :role "Tommy" ;
      schema:name "Fionn Whitehead" .

dbr:Ellen_Burstyn :role "Murph (Older)" ;
      schema:name "Ellen Burstyn" .

dbr:Joseph_Gordon-Levitt :role "Arthur" ;
      schema:name "Joseph Gordon-Levitt" .

dbr:Leonardo_DiCaprio :role "Cobb" ;
      schema:name "Leonardo DiCaprio" .

dbr:Hugh_Jackman :role "Robert Angier" ;
      schema:name "Hugh Jackman" .

dbr:Matthew_McConaughey :role "Cooper" ;
      schema:name "Matthew McConaughey" .
```

Listing 4 Result yielded by the engine after applying the mapping defined in Listing 3, serialised in Turtle.

of a component is further transformed by the following one. This division of steps helps to encapsulate the specific functionality of each component and allows to build a rich output based on an incremental process. Fig. 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 mapping input in text format following the ShExML defined syntax (see Listing 3 for an input example), (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, and (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 [5], 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

Fig. 1. Diagram of the pipes and filters architecture implemented in the ShExML engine based on the UML components diagram. Some additional symbols were used for the clarity of the diagram, like the arrows to indicate the direction in which the data flows; the tokens, AST and VarTable representations which are represented graphically to help understand how the data is transformed; and the outputs which use the logos of the employed technologies. The numbers placed next to the main components denote the steps followed for an input processing in the ShExML engine and they are used as such in the explanation contained in Section 4.1.



to the following specific implementations: (1) and (2) are implemented using ANTLR4[8] [25], a tool for generating parsers using a grammar-based input. The grammar is then converted to a lexer, which converts a text-based input to tokens, and a parser, which based on the received tokens will compose a tree with the structure of the program. The result of the parser is then received by an AST creator which acts as an intermediary between the ANTLR4 output and the engine domain model – decoupling the AST model from the lexer and parser used technology. The step (3) registers the symbols declared across the mapping rules and makes them available for the output generators, generating a dictionary with the different symbols and their meaning. This dictionary is right now implemented using its mutable version 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 last step (4), the output is generated by one of the output generators. Right now, the ShExML engine counts on four different generators depending on the defined output: RDF (the main one that outputs an RDF graph in different serialisation formats), RML (translates ShExML mapping rules to RML equivalent ones [23]), ShEx (infers and outputs a set of ShEx shapes that validates the data generated by the mapping rules), and SHACL (having a similar behaviour to the ShEx generator but with SHACL shapes). For the purpose of this study, I will focus on the RDF one which has the greatest load and is responsible for the generation of RDF, being compared in the evaluation.

4.2. 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.[9] Other details have been encapsulated in calls to functions which can be interpreted as follows:

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

[9] https://github.com/herminiogg/ShExML/blob/master/src/main/scala/com/herminiogarcia/shexml/visitor/RDFGeneratorVisitor.scala

- *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 ids 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 the expected result from this process can be seen in Listing 6 for the ShExML input defined in Listing 5.
- *generateAllQueries(iq)*: This function receives an iterator query, like the ones defined in Listing 6, and generates all the possible queries given the result of the partial queries evaluation. See Listing 7 for an example of the queries executed for the ShExML example in Listing 5.

```
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 $s$;
    $SR \leftarrow resolveAction(sa, ST)$;
    $POR \leftarrow \emptyset$;
    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)*.

It is worth noting that for non-hierarchical files (like CSV and results from a SQL or SPARQL query), which will yield a set of tabular data, the *composeIterationQuery* function will not generate a query with placeholders and the *generateAllQueries* function will, on its turn, only generate a query per declared field. This difference, emanated from the need to keep the context while processing hierarchical files, explains the initial difference in performance when processing them, and why it was needed to specifically focus on the optimisation of this hierarchical processing mechanism, in contrast with their non-hierarchical counterparts.

5. Profiling the ShExML engine and performance improvements

For discovering the bottlenecks of the ShExML engine, and allowing to contrast the performance improvements, a profiling tool was used. Profilers are best suited for this task due to their ability to gather execution times for the functions inside the code, showing them in a call graph which allows to see the interdependencies between them [26]. Based on this analysis, the different improvements are introduced per version, allowing for a modular assessment of their overall effect in the engine performance.

The Java VisualVM[10] tool was used due to its specific support for profiling processes executed in the JVM. This allowed for monitoring the different methods inside the engine and assessing their time consumption in relation to the whole execution time. The workflow consisted in starting the application to, then, activate the sampler on the

[10] https://visualvm.github.io/

```
PREFIX : <http://example.com/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX schema: <http://schema.org/>
SOURCE films_xml_file <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 5 Example ShExML file with a root iterator and a nested iterator, and with one field per iterator. The input file `films.xml` is the one represented in Listing 1.

```
# 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 6 Iterator queries composed by the engine when executing the mapping rules contained in Listing 5.

```
# 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[2]/cast/node()[self::actor or self::actress][1]/name
//film[2]/cast/node()[self::actor or self::actress][2]/name
```

Listing 7 Final queries to be executed after expanding the iterator queries listed in Listing 6.

Bottleneck	Fix	Version implementing the fix
Same file downloaded many times	Cache system	ShExML-v0.3.3
Hierarchical files queries executed many times	Cache system for query results	ShExML-v0.3.3
File contents used for results id generation	Change the file contents for the path to generate the id	ShExML-v0.4.0
Functions being evaluated when it was not necessary (eager evaluation)	Change the possible functions to their lazy evaluation counterpart	ShExML-v0.4.0
Filtering function to assign the predicate-object results to the corresponding subjects $O(n^2)$	Grouping function $O(n)$	ShExML-v0.4.0
Parsing JSON data many times	Cache system for keeping already JSON parsed files	ShExML-v0.4.0
URI normalisation system by default	Making the URI normalisation system optional	ShExML-v0.4.2
Data type inference mechanism by default	Making the data type inference system optional	ShExML-v0.4.2
Parsing XML data many times	Cache system for keeping already XML parsed files	ShExML-v0.5.1
Slow JSONPath library (gatling-jsonpath)	Change the library to Jayway JsonPath	ShExML-v0.5.1
Slow XPath library (kantan-xpath)	Change the library to Saxon-HE	ShExML-v0.5.1

Table 1

This tables summarises the found bottlenecks and how they were fixed. A more detailed explanation can be found in Section 5 where more background information and examples are introduced.

CPU, which allows to generate a faster overview than instrumenting the whole application, and its results offered enough data for the purpose of this work. However, due to absence of the said instrumentation, it is not possible to start the sampling together with the process. Therefore, a general idle delay of 20 seconds had to be included 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, considering their specific weight on the general algorithm.[11] It is worth noting that this process requires the expertise of the tool developer to detect which code blocks are behaving abnormally slow. At

[11] The profiler results can be consulted on: https://github.com/herminiogg/shexml-performance-evaluation/tree/main/profiler-snapshots

the same time, the solution to the bottlenecks irremediable involves some trial-and-error modifications until finding the correct optimisation – or realising that it cannot be optimised as this is, in reality, the normal behaviour. 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 – which can be consulted per bottleneck in Table 1 – 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.[12] 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 retrieval method was redesigned to cache the files contents, only retrieving them for the first time and keeping them in memory for consecutive accesses. 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 4, 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.[13] 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`[14] instead of `filter` which performs a lazy evaluation instead of an eager one, delaying the evaluation until it is necessary and, therefore, avoiding unnecessary computation and memory usage [27, 28]. Also, some used collections of type `List` were changed for more performant equivalent ones according to their use case inside the algorithm.[15] For example, if a collection needs to be updated and randomly accessed, the immutable `List` type offers a linear performance while the immutable `Vector` type can offer an effective constant time. Nevertheless, 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. Instead of taking each subject result and filtering the concerning predicate-object results, these are grouped beforehand according to the relevant subject result. Ultimately, this changes the quadratic time complexity $O(n^2)$ to a linear one $O(n)$. 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.[16] Since the very beginning, ShExML incorporates a system for inferencing data types based on the obtained result and a URI normalisation system that ensures that the engine generates compliant URIs. The data type inference system works by trying to convert the obtained value to a list of prioritised data types, for example, if the engine receives the value `100` it will try to convert it to an integer and, when this succeeds, the acquired data type will be `xsd:integer`. A more detailed explanation of this mechanism can be consulted in [29]. The URI normalisation system receives a value and transforms it by omitting, or replacing, the non-valid characters within the URI specification[17] (e.g., white spaces). Following the example in Listing 4, if the engine receives the result `"Christian Bale"` and it must generate a URI under the `dbr:` prefix, it will normalise it to `dbr:Christian_Bale` so the entire result is a valid RDF file. As discussed later, this does not have a significant effect when processing small inputs but, when exposed to longer inputs, it can save some time, moreover when

[12] See the changelog: https://github.com/herminiogg/ShExML/compare/v0.3.2...v0.3.3
[13] See the changelog: https://github.com/herminiogg/ShExML/compare/v0.3.3...v0.4.0
[14] https://www.scala-lang.org/api/current/scala/collection/ArrayOps$$WithFilter.html
[15] For collections performance consult: https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html
[16] See the changelog: https://github.com/herminiogg/ShExML/compare/v0.4.0...v0.4.2
[17] https://datatracker.ietf.org/doc/html/rfc3986

they are not required in the requested RDF generation. Besides, these are not standard features across languages, therefore, it was deemed to make them optional letting users decide whether they need these systems for their use case.

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. While it is not the scope of this work to study the differences on performance of these libraries and why they happen, the reader can find more information and benchmarks about this in previous works [30, 31] and some other online benchmarks[18]. In previous versions, the engine was using kantan-xpath[19] and gatling-jsonpath[20], 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[21], 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[22] library was incorporated delivering much better results. Apart from that, the VTD-XML[23] 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 licence collision. As it was done with the JSON parser in v0.4.0, a cache system was also introduced in this version for the XML parser.[24]

6. Evaluation

In order to demonstrate the effectiveness of the previously described analysis and the consequent performance improvements, two experiments were conducted: (1) a personalised experiment over the enlisted versions, and (2) a replication study of the experiment performed in [8] adapted to the optimised ShExML engine version and the other engines coetaneous versions.

The reasoning behind introducing the custom-made experiment (1) is two-fold. First, the existing methodology is not suited for hierarchical data and only takes into account flat data in JSON (not in XML). This, ultimately, hinders the evaluation of many optimisations that can take place in ShExML, due to its hierarchical-based algorithm (see Section 4.2). Moreover, the existing study only performs three measures per case, falling short in statistical significance and hampering, hence, a more profound statistical analysis.

6.1. *ShExML optimisation experiment*

This section presents the custom-made evaluation of the ShExML engine optimisation, alongside its results and a discussion of them.

6.1.1. Methodology

This methodology is based upon the one introduced in the SPARQL-Anything evaluation paper [8] but with some additions and changes to better assess the differences between the different engine versions, taking care of reaching enough statistical evidence and evaluating hierarchical files access more thoroughly.[25] Therefore, the experiment was composed of three different inputs: (1) a set of mapping rules for a single film shape encoded in an almost-flat

[18] See for example: https://github.com/fabienrenaud/java-json-benchmark

[19] https://nrinaudo.github.io/kantan.xpath/

[20] https://github.com/gatling/jsonpath

[21] https://github.com/json-path/JsonPath

[22] https://github.com/Saxonica/Saxon-HE

[23] https://vtd-xml.sourceforge.io/

[24] See the changelog: https://github.com/herminiogg/ShExML/compare/v0.4.2...v0.5.1

[25] The resources and the raw results for this experiment can be found on: https://github.com/herminiogg/shexml-performance-evaluation. Additionally, they can also be consulted under the following persistent DOI: https://doi.org/10.5281/zenodo.13305712

JSON file with 1000 entries, (2) another set for a single film shape encoded in an almost-flat XML file containing 1000 entries, and (3) a set of mapping rules used in a real case scenario [32] which converts multiple files containing descriptive and contact data from 2260 institutions in JSON format, extracted from the EHRI project[26], with a deeper hierarchical structure. An example of the files format and aspect can be seen in Appendix A, where Listing 9 contains an extract from the films XML file, Listing 8 from the films JSON file and Listing 10 from the EHRI institutions JSON files. 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, and 5 different variables were captured: process elapsed time to perform the requested operation, the CPU kernel time which can be defined as the time that the process spends running OS code (this can be, for example, IO operations), the CPU user time which captures the time that the process uses to run its own code, the CPU usage that represents the percentage of CPU utilisation that this process is taking with respect to the whole CPU capacity – in the case of multi-core CPUs, the CPU usage can go over 100% as the process uses more than one core, and the maximum resident set size representing the peak memory consumed by the process. The v0.3.2 was used as the baseline version and the other 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.

6.1.2. 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.[27] 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 Appendix B. Moreover, visual representations of the distributions for the 5 measured variables can be found in: Fig. 2 for the elapsed time, Fig. 4 for the CPU kernel time, Fig. 5 for the CPU user time, Fig. 6 for the CPU usage and Fig. 3 for the peak memory usage.

Fig. 2. Violin plot comparing the distribution of the elapsed time results in milliseconds for the different inputs and engines under the test. Results for the ShExML-v0.3.2 under the EHRI institutions input are not shown as only one result was yielded whose value is 2188032 ms. The y axis has been adapted according to a log10 scale to increase the interpretability of this graphic.

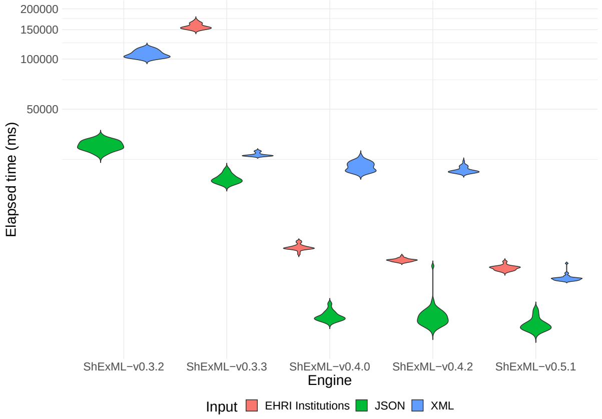
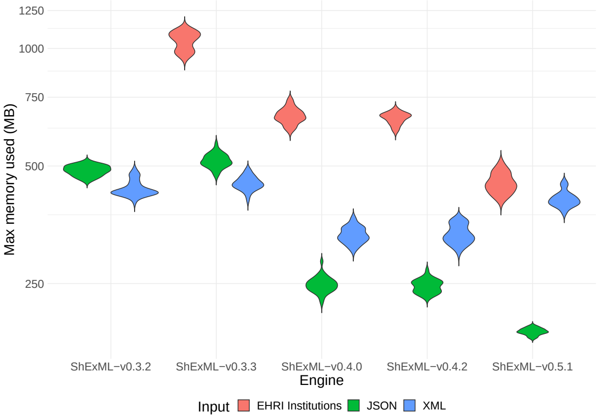


Fig. 3. Violin plot comparing the distribution of the peak memory used results in megabytes for the different inputs and engines under the test. Results for the ShExML-v0.3.2 under the EHRI institutions input are not shown as only one result was yielded whose value is 1926.30 MB. The y axis has been adapted according to a log10 scale to increase the interpretability of this graphic.



With the aim to assess the differences between the engines in a statistically-significant manner, a statistical hypothesis testing was conducted. In order to test the differences of means of two or more groups with respect to one

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

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

Fig. 4. Violin plot comparing the distribution of the CPU kernel time results in seconds for the different inputs and engines under the test. Results for the ShExML-v0.3.2 under the EHRI institutions input are not shown as only one result was yielded whose value is 45.53 s. The y axis has been adapted according to a log10 scale to increase the interpretability of this graphic.

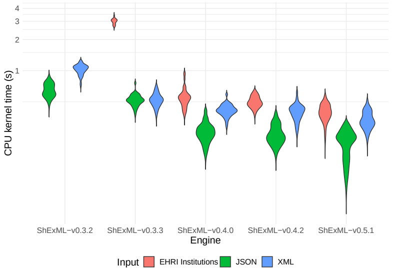


Fig. 5. Violin plot comparing the distribution of the CPU user time results in seconds for the different inputs and engines under the test. Results for the ShExML-v0.3.2 under the EHRI institutions input are not shown as only one result was yielded whose value is 2261.28 s. The y axis has been adapted according to a log10 scale to increase the interpretability of this graphic.

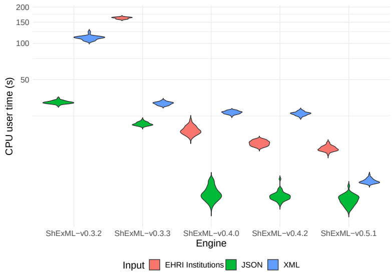
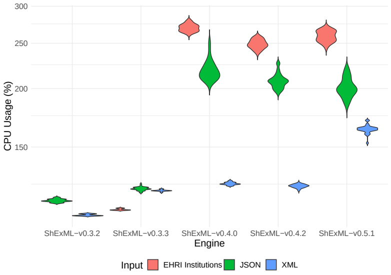


Fig. 6. Violin plot comparing the distribution of the CPU usage results in percentages for the different inputs and engines under the test. Results for the ShExML-v0.3.2 under the EHRI institutions input are not shown as only one result was yielded whose value is 105%. The y axis has been adapted according to a log10 scale to increase the interpretability of this graphic.



independent variable, the most suitable parametric test would be a One-Way ANOVA. However, it assumes the data to be distributed normally, having homogeneous variances and the observations being independent. Upon applying the Spahiro-Wilk test to verify the normality of the distributions, these are not guaranteed to be normally distributed which imposes the use of a non-parametric counterpart, that is, a Kruskal-Wallis test which delivered very significant differences ($p < .001$) for all the combinations of the measured variables, inputs and engines. Taking the elapsed time as the main variable to assess RQ1, I analysed the different improvements and their degree of influence on the performance by means of the post hoc tests (which determine the differences between groups) and the effect sizes (allowing to measure the meaningfulness of the discovered differences). The results for these can be consulted in Table 2. The analysis is based on the elapsed time variable and how it correlates with the differences seen on the other four variables (capturing CPU and RAM usage), answering RQ3. 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 [33] (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 in the elapsed time between v0.3.2 and v0.3.3 for the three inputs with a medium effect size, relating those results to the same medium effect size in the CPU user time and CPU usage, however, only a medium effect size is present on the XML input for the CPU kernel time and a small one on the JSON input for the peak memory usage.

Between v0.3.3 and v0.4.0 there are significant differences in the elapsed time, and a medium effect size, for the XML and the institutions entries, but the JSON input shows very significant differences and a large effect size which translates equally to the CPU user time. For the CPU kernel time, the JSON and EHRI institutions entries show a large effect size while the XML one has a medium effect size. The three inputs yield large effect sizes for the CPU usage and the peak memory usage.

When comparing v0.4.0 and v0.4.2 only the institutions entry shows significant differences and a medium effect size for the elapsed time and the CPU user variables.

Finally, there are significant differences in the elapsed times and a medium effect size between the v0.4.2 and the v0.5.1 for the three inputs. This also translates to the peak memory usage which shows a medium effect size for the JSON input and a large one for the other two. The CPU kernel time variable has a small effect size for the XML input and a medium one for the EHRI institutions entry, while the CPU user time variable shows a medium effect size for the XML and EHRI institutions entries alike.

As a general note, the CPU user time is normally related quite steadily to the results collected in the elapsed time variable. This is explained due to the reason that the engine process normally spends most of its time in the user mode. There is also a general increase on the CPU usage, as shown in Fig. 6, which can be explained by a better use of the CPU user time to resolve the same tasks in less time, avoiding idle time.

A closer look to the elapsed time distributions allows to further explain these differences or lack of them. In the case of the JSON films entry, the distributions for the three most performant versions (as it can be seen in Fig. 7) are quite close and they overlap among them, explaining, therefore, the lack of significant differences between v0.4.0 and v0.4.2 and the effect size being very close to a small one between v0.4.2 and v0.5.1.

For the XML input (represented in Fig. 8), 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 slight improvement on performance over v0.4.0 (even though not quite appreciable).

Finally, when the EHRI institutions input is analysed (see Fig. 9), the three most performant versions (v0.5.1, v0.4.2 and v0.4.0) show very defined and almost not overlapping distributions.

Fig. 7. Plot of the three most performant engines distributions against the input JSON Films 1000 entries.

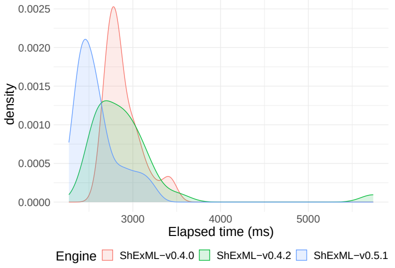


Fig. 8. Plot of the three closest (on performance) engines distributions against the input XML Films 1000 entries.

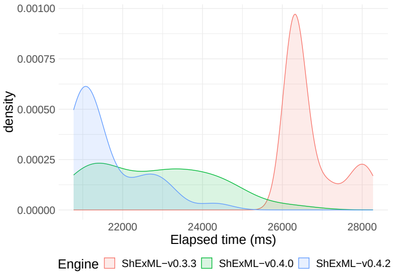
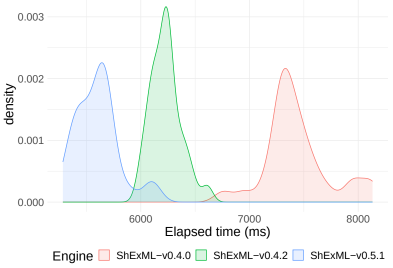


Fig. 9. Plot of the three most performant engines distributions against the input EHRI institutions.



6.1.3. *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 substantial performance improvement in bigger inputs, like the EHRI institutions one. This also reflects quite clearly on the decrease of the CPU kernel time for the XML input in opposition to the JSON input, as XML files tend to be larger and require longer IO operations. Moreover, when having a lot of 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. Special attention requires the caching of iteration queries for small and flat inputs as they can increase memory usage, like in the case of the JSON input.

As already anticipated earlier, the transition from v0.3.3 to v0.4.0 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 for the elapsed time can be explained by the JSON parser cache also introduced in this version which can be clearly seen in the CPU user effect sizes (large for the JSON input and medium for the other two inputs). Conversely, the EHRI institutions case, which also uses an input in the JSON format, deals with many files in comparison to a single file entry in the films case, mitigating partially this effect. In general, a big decrease on memory consumption is appreciated across inputs.

Making the data types inference and URI normalisation systems optional (from v0.4.0 to v0.4.2) did not have a very big impact on smaller inputs (like in both films entries) but, as the result of the EHRI institutions entry demonstrates, it can have a bigger impact when longer inputs are used. At the same time, from a usability perspective, having these systems 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 and memory usage in the ShExML engine for the three inputs. When other bottleneck problems are resolved, the main core of these data mapping engines relies on the query system which is normally delegated to an external

Input	Variable	Comparison	*p*	*p* int.	*r*	*r* int.
JSON Films 1000 entries	Elapsed time	ShExML-v0.3.2 - ShExML-v0.3.3	<.05	*	.345	++
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.555	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	.695		.0506	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.05	*	.307	++
	CPU Kernel	ShExML-v0.3.2 - ShExML-v0.3.3	.050		.272	
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.630	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	.618		.0643	
		ShExML-v0.4.2 - ShExML-v0.5.1	.235		.161	
	CPU User	ShExML-v0.3.2 - ShExML-v0.3.3	<.05	*	.345	++
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.545	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	.343		.122	
		ShExML-v0.4.2 - ShExML-v0.5.1	.152		.192	
	CPU Percentage	ShExML-v0.3.2 - ShExML-v0.3.3	<.05	*	.346	++
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.944	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	<.05	*	.302	++
		ShExML-v0.4.2 - ShExML-v0.5.1	.266		.156	
	Max Memory	ShExML-v0.3.2 - ShExML-v0.3.3	<.05	*	.261	+
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.801	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	.757		.0399	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.001	***	.498	++
XML Films 1000 entries	Elapsed time	ShExML-v0.3.2 - ShExML-v0.3.3	<.05	*	.347	++
		ShExML-v0.3.3 - ShExML-v0.4.0	<.05	*	.422	++
		ShExML-v0.4.0 - ShExML-v0.4.2	.149		.186	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.05	*	.425	++
	CPU Kernel	ShExML-v0.3.2 - ShExML-v0.3.3	<.05	*	.430	++
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.456	++
		ShExML-v0.4.0 - ShExML-v0.4.2	.736		.0436	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.05	*	.279	+
	CPU User	ShExML-v0.3.2 - ShExML-v0.3.3	<.05	*	.345	++
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.455	++
		ShExML-v0.4.0 - ShExML-v0.4.2	.333		.125	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.001	***	.455	++
	CPU Percentage	ShExML-v0.3.2 - ShExML-v0.3.3	<.05	*	.355	++
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.633	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	.053		.250	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.001	***	.650	+++
	Max Memory	ShExML-v0.3.2 - ShExML-v0.3.3	.211		.169	
		ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	1.09	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	.952		.0077	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.001	***	.589	+++
EHRI institutions (JSON)	Elapsed time	ShExML-v0.3.3 - ShExML-v0.4.0	<.05	*	.431	++
		ShExML-v0.4.0 - ShExML-v0.4.2	<.05	*	.437	++
		ShExML-v0.4.2 - ShExML-v0.5.1	<.05	*	.420	++
	CPU Kernel	ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.619	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	.194		.168	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.05	*	.396	++
	CPU User	ShExML-v0.3.3 - ShExML-v0.4.0	<.05	*	.431	++
		ShExML-v0.4.0 - ShExML-v0.4.2	<.001	***	.463	++
		ShExML-v0.4.2 - ShExML-v0.5.1	<.05	*	.368	++
	CPU Percentage	ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	1.25	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	<.001	***	.755	+++
		ShExML-v0.4.2 - ShExML-v0.5.1	<.05	*	.361	++
	Max Memory	ShExML-v0.3.3 - ShExML-v0.4.0	<.001	***	.612	+++
		ShExML-v0.4.0 - ShExML-v0.4.2	.593		.0690	
		ShExML-v0.4.2 - ShExML-v0.5.1	<.001	***	.612	+++

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. *p* int. refers to the interpretation of the p-value, where * means significant differences and *** very significant differences. *r* int. refers to the interpretation of the effect size where + means a small effect size, ++ a medium effect size, and +++ a large effect size.

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, at the cost, however, of an increase on the memory consumption.

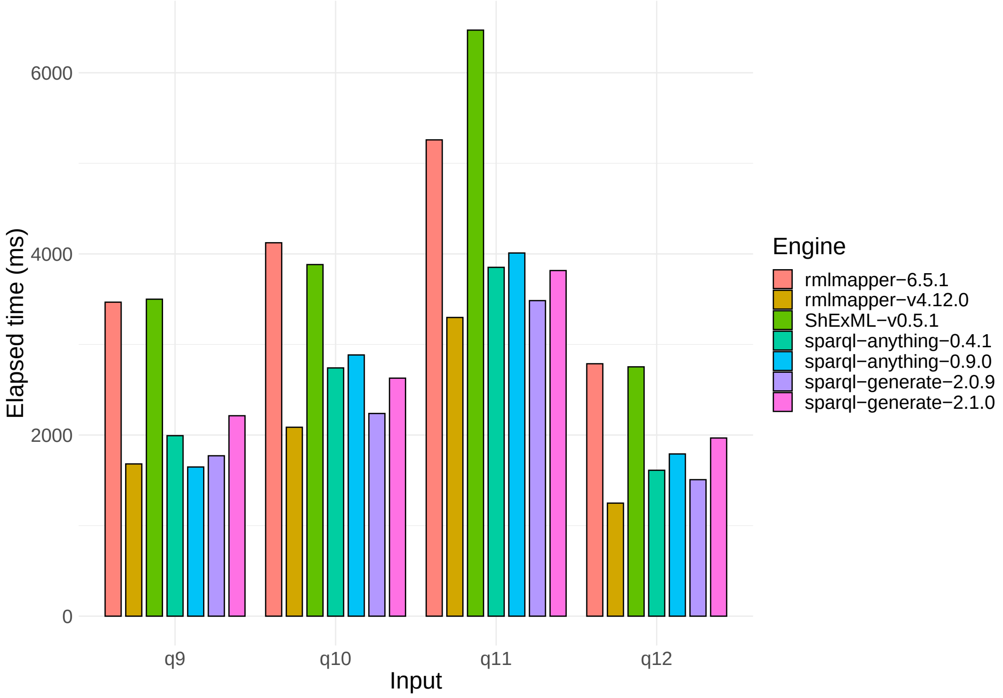
6.2. Replication study of the SPARQL-Anything experiment

In this section the methodology followed to replicate the SPARQL-Anything experiment is introduced. Afterwards, the results arisen from it are exposed and discussed.

6.2.1. Methodology

For this replication study, I based the methodology on the one followed by the authors in the original paper [8] which consists of two different tests: one containing 12 different integration cases (q1 to q12) and an incremental in size input based on the q12 data (containing 10, 100, 1000, 10000, 100000, 1000000 entities respectively). In the original experiment the inputs ranging from q1 to q8 were only run on the SPARQL-Anything and SPARQL-Generate engines, as they are intended for query evaluation retrieval, which RML and ShExML are not capable of doing, therefore, I omitted those from the experiment. Each input is executed against each engine thrice and the average execution time is preserved for comparison, as in the original experiment. In order to answer RQ2, not only the replication study, using the engines original versions, was run but also the same experiment using the version 0.5.1 of the ShExML engine and its coetaneous versions from the other engines (i.e., version 6.5.1 for RMLMapper, version 0.9.0 for SPARQL-Anything and 2.1.0 for SPARQL-Generate).

Fig. 10. Bar chart containing the execution time in milliseconds for the four common inputs under the SPARQL-Anything experiment. ShExML-v0.2.7 results are not included as they always exceeded the timeout of 3 minutes.



6.2.2. Results and discussion

The previously introduced methodology was run on the same environment described in Section 6.1.2 and the resources and results can be accessed on Github.[28]

[28] https://github.com/herminiogg/shexml-performance-evaluation/tree/main/sparql-anything-experiment/experiment

Fig. 11. Line chart comparing the results for the incremental input of SPARQL-Anything experiment executed over the versions of the engines used in the original experiment. The results on the y axis have been transformed using a log10 scale in order to make them more comparable. Results on the inputs 10000, 100000, 1000000 for the engine ShExML-v0.2.7 exceeded the timeout and are represented here as higher values only for representational purposes following the original graphic published in [8].

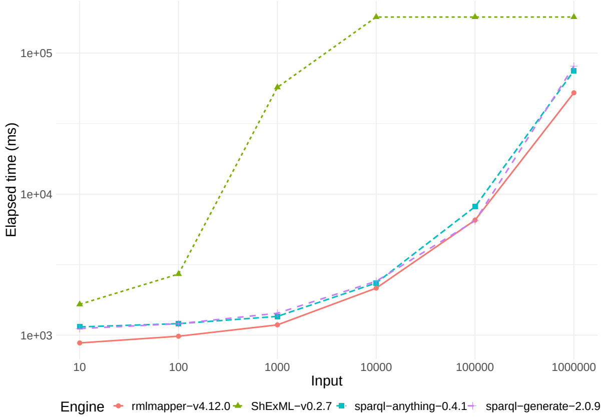
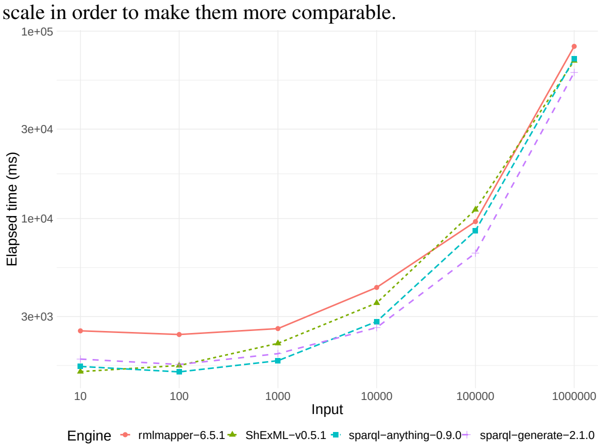


Fig. 12. Line chart comparing the results for the incremental input of SPARQL-Anything experiment executed over the new versions of the engines. The results on the y axis have been transformed using a log10 scale in order to make them more comparable.



Looking to the results of the four discrete inputs (q9, q10, q11 and q12), represented in Fig. 10, it can be seen how the optimised version of the ShExML engine shows a similar behaviour performance-wise to the other engines, even though in some cases, like q11, it still employs a bit more time. Interestingly, the newer version of the RMLMapper seems to perform rather bad in comparison with its predecesor, in some cases using double the time for the same task. This leads to, as a side effect, RMLMapper not being the most performant engine in most of the cases as it was before. On their turn, also SPARQL-Generate and SPARQL-Anything engines perform a bit worse for almost all the cases than their old versions, but in their case the difference is not so acute.

Focusing on the incremental input experiment, the optimised ShExML engine stands side to side to other engines, offering a similar performance (see Fig. 12). Moreover, the ShExML engine starts with a very good performance for the smallest case, it uses a bit more time for the middle range cases, but it seems to improve for the largest case. Here, it is also quite noticeable how the RML engine performs worse than the other three engines only showing a more comparable performance for the two larger inputs. This is a huge difference in comparison with the old versions where RMLMapper used to yield the best performance metrics (see Fig. 11). Between SPARQL-Generate and SPARQL-Anything, in the old experiment both engines showed a similar performance for almost all the cases, however, with the newer versions, SPARQL-Generate performs worse than SPARQL-Anything for the smaller inputs while, on bigger cases, SPARQL-Anything is performing worse than SPARQL-Generate.

7. Future work

While a lot of performance improvements have been introduced with this work in the ShExML engine, there is still some room for further refinement. The main enhancement relates to the possibility of executing the engine in parallel. 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.

Following the rationale behind the creation of the custom-made evaluation of this paper, it is envisaged to expand it to cover more cases, data mapping languages and engines. This statistically-based methodology can be used to better determine the differences between engines, providing a better picture of mapping rules engines capabilities in relation to multiple – and heterogeneously-shaped – data, and driving future optimisations in these kinds of tools.

8. Conclusions

In this work, the optimisation of the ShExML engine has been tackled using a profiling tool 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. Upon this analysis, the ShExML engine shows an improved performance, utilising the CPU cores more efficiently and reducing the consumed memory. Furthermore, as a result of the replication study, it has been demonstrated that due to the optimisations performed on the ShExML engine, it now manifests a comparable performance to other similar ones, letting ShExML users benefit from a much faster and reliable tool.

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] 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. doi:10.1016/j.websem.2022.100753.

[2] 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, Athens, Greece, 2023, pp. 152–175. doi:10.1007/978-3-031-47243-5_9.

[3] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens and R. Van de Walle, RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, in: *Proceedings of the 7th Workshop on Linked Data on the Web*, C. Bizer, T. Heath, S. Auer and T. Berners-Lee, eds, CEUR Workshop Proceedings, Vol. 1184, Seoul, Korea, 2014. ISSN 1613-0073. http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[4] 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* **15**(1) (2024), 191–221. doi:10.3233/SW-223224.

[5] 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 Computer Science* **6** (2020), e318. doi:10.7717/peerj-cs.318.

[6] 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), 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, Online, 2021. https://ceur-ws.org/Vol-2873/paper11.pdf.

[7] 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: *CIKM '20: Proceedings of the 29th ACM international conference on Information & Knowledge Management*, Association for Computing Machinery, Virtual Event, Ireland, 2020, pp. 3039–3046. doi:10.1145/3340531.3412881.

[8] 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. doi:10.1145/3555312.

[9] 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, May 28–June 1, 2017, Proceedings, Part I 14*, E. Blomqvist, D. Maynard, A. Gangemi, R. Hoekstra, P. Hitzler and O. Hartig, eds, Lecture Notes in Computer Science, Vol. 10249, Springer, Cham, Portorož, Slovenia, 2017, pp. 35–50. doi:10.1007/978-3-319-58068-5_3.

[10] U. Simsek, E. Kärle and D. Fensel, RocketRML - A NodeJS Implementation of a Use Case Specific RML Mapper, in: *Joint Proceedings of the 1st International Workshop on Knowledge Graph Building and 1st International Workshop on Large Scale RDF Analytics co-located with 16th Extended Semantic Web Conference (ESWC 2019), June 3, 2019*, D. Chaves-Fraga, P. Heyvaert, F. Priyatna, J.F. Sequeda, A. Dimou, H. Jabeen, D. Graux, G. Sejdiu, M. Saleem and J. Lehmann, eds, CEUR Workshop Proceedings, Vol. 2489, CEUR-WS.org, Portorož, Slovenia, 2019, pp. 46–53. https://ceur-ws.org/Vol-2489/paper5.pdf.

[11] G. Haesendonck, W. Maroy, P. Heyvaert, R. Verborgh and A. Dimou, Parallel RDF generation from heterogeneous big data, in: *SBD '19; Proceedings of the International Workshop on Semantic Big Data*, Association for Computing Machinery, Amsterdam, Netherlands, 2019, pp. 1–6. doi:10.1145/3323878.3325802.

[12] 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:10.1016/j.websem.2020.100596.

[13] M. Scrocca, A. Carenini, M. Comerio and I. Celino, Semantic Conversion of Transport Data Adopting Declarative Mappings: An Evaluation of Performance and Scalability, in: *Proceedings of the 3rd International Workshop Semantics And The Web For Transport co-located with Semantics Conference (SEMANTiCS 2021), September 6, 2021*, D. Chaves-Fraga, P. Colpaert, M. Sadeghi, M. Scrocca and M. Comerio, eds, CEUR Workshop Proceedings, Vol. 2939, CEUR-WS.org, Online, 2021. https://ceur-ws.org/Vol-2939/paper2.pdf.

[14] C. Stadler, L. Bühmann, L. Meyer and M. Martin, Scaling RML and SPARQL-based Knowledge Graph Construction with Apache Spark, in: *Proceedings of the 4th International Workshop on Knowledge Graph Construction co-located with 20th Extended Semantic Web Conference ESWC 2023, May 28, 2023*, D. Chaves-Fraga, A. Dimou, A. Iglesias-Molina, U. Serles and D.V. Assche, eds, CEUR Workshop Proceedings, Vol. 3471, CEUR-WS.org, Hersonissos, Greece, 2023. https://ceur-ws.org/Vol-3471/paper8.pdf.

[15] S. Bin, C. Stadler and L. Bühmann, KGCW2023 Challenge Report RDFProcessingToolkit / Sansa, in: *Proceedings of the 4th International Workshop on Knowledge Graph Construction co-located with 20th Extended Semantic Web Conference ESWC 2023, May 28, 2023*, D. Chaves-Fraga, A. Dimou, A. Iglesias-Molina, U. Serles and D.V. Assche, eds, CEUR Workshop Proceedings, Vol. 3471, CEUR-WS.org, Hersonissos, Greece, 2023. https://ceur-ws.org/Vol-3471/paper12.pdf.

[16] E. Iglesias and M. Vidal, Knowledge Graph Creation Challenge: Results for SDM-RDFizer, in: *Proceedings of the 4th International Workshop on Knowledge Graph Construction co-located with 20th Extended Semantic Web Conference ESWC 2023, May 28, 2023*, D. Chaves-Fraga, A. Dimou, A. Iglesias-Molina, U. Serles and D.V. Assche, eds, CEUR Workshop Proceedings, Vol. 3471, CEUR-WS.org, Hersonissos, Greece, 2023. https://ceur-ws.org/Vol-3471/paper13.pdf.

[17] E. Iglesias, M.-E. Vidal, D. Collarana and D. Chaves-Fraga, Empowering the SDM-RDFizer tool for scaling up to complex knowledge graph creation pipelines, *Semantic Web* **Pre-press**(Pre–press) (2024), 1–28. doi:10.3233/SW-243580.

[18] E. Iglesias, S. Jozashoori and M.-E. Vidal, Scaling up knowledge graph creation to large and heterogeneous data sources, *Journal of Web Semantics* **75** (2023), 100755. doi:10.1016/j.websem.2022.100755.

[19] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M.S. Pérez and O. Corcho, Morph-KGC: Scalable knowledge graph materialization with mapping partitions, *Semantic Web* **15**(1) (2024), 1–20. doi:10.3233/SW-223135.

[20] E. Prud'hommeaux, J.E. Labra Gayo and H. Solbrig, Shape expressions: an RDF validation and transformation language, in: *SEM '14: Proceedings of the 10th International Conference on Semantic Systems*, Association for Computing Machinery, Leipzig, Germany, 2014, pp. 32–40. doi:10.1145/2660517.2660523.

[21] S.M. Oo, B.D. Meester, R. Taelman and P. Colpaert, Towards Algebraic Mapping Operators for Knowledge Graph Construction, in: *Proceedings of the ISWC 2023 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice co-located with 22nd International Semantic Web Conference (ISWC 2023), November 6-10, 2023*, I. Fundulaki, K. Kozaki, D. Garijo and J.M. Gómez-Pérez, eds, CEUR Workshop Proceedings, Vol. 3632, CEUR-WS.org, Athens, Greece, 2023. https://ceur-ws.org/Vol-3632/ISWC2023_paper_412.pdf.

[22] A. Iglesias-Molina, A. Cimmino and Ó. Corcho, Devising Mapping Interoperability With Mapping Translation, in: *Proceedings of the 3rd International Workshop on Knowledge Graph Construction (KGCW 2022) co-located with 19th Extended Semantic Web Conference (ESWC 2022), May 30, 2022*, D. Chaves-Fraga, A. Dimou, P. Heyvaert, F. Priyatna and J. Sequeda, eds, CEUR Workshop Proceedings, Vol. 3141, CEUR-WS.org, 2022. https://ceur-ws.org/Vol-3141/paper6.pdf.

[23] 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)*, U. Simsek, D. Chaves-Fraga, T. Pellegrini and S. Vahdat, eds, CEUR Workshop Proceedings, Vol. 3235, CEUR-WS.org, Vienna, Austria, 2022, p. paper–11. https://ceur-ws.org/Vol-3235/paper11.pdf.

[24] A. Kumar, Software Architecture Styles a Survey, *International Journal of Computer Applications* **87**(9) (2014). doi:10.5120/15234-3768.

[25] T. Parr, *The definitive ANTLR 4 reference*, The Pragmatic Bookshelf, 2013, pp. 1–326. ISBN 1934356999.

[26] S.L. Graham, P.B. Kessler and M.K. McKusick, Gprof: A call graph execution profiler, *ACM Sigplan Notices* **17**(6) (1982), 120–126. doi:10.1145/872726.806987.

[27] P. Henderson and J.H. Morris Jr, A lazy evaluator, in: *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, Atlanta, Georgia, 1976, pp. 95–103. doi:10.1145/800168.811543.

[28] D.A. Turner, *Recursion Equations as a Programming Language*, in: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, S. Lindley, C. McBride, P. Trinder and D. Sannella, eds, Springer International Publishing, Cham, 2016, pp. 459–478. ISBN 978-3-319-30936-1. doi:10.1007/978-3-319-30936-1_24.

[29] 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)*, D. Chaves-Fraga, A. Dimou, P. Heyvaert, F. Priyatna and J.F. Sequeda, eds, CEUR Workshop Proceedings, Vol. 2873, CEUR-WS.org, Online, 2021. http://ceur-ws.org/Vol-2873/paper2.pdf.

[30] H.K. Dhalla, A Performance Analysis of Native JSON Parsers in Java, Python, MS.NET Core, JavaScript, and PHP, in: *16th International Conference on Network and Service Management (CNSM)*, IEEE, Online, 2020, pp. 1–5. doi:10.23919/CNSM50824.2020.9269101.

[31] B. Oliveira, V. Santos and O. Belo, Processing XML with Java–a performance benchmark, *International Journal of New Computer Architectures and their Applications (IJNCAA)* **3**(1) (2013), 72–85.

[32] H. García-González and M. Bryant, The Holocaust Archival Material Knowledge Graph, in: *The Semantic Web – ISWC 2023*, Vol. 14266, 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, Athens, Greece, 2023, pp. 362–379. ISBN 978-3-031-47243-5. doi:10.1007/978-3-031-47243-5_20.

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

Appendix A. Examples for the inputs used in the evaluation described in Section 6.1

```
{
  "films": [
    {
      "id": 1,
      "name": "Film-1",
      "year": "2007",
      "country": "UK",
      "director": [
        "Director-2",
        "Director-8"
      ]
    },
    {
      "id": 2,
      "name": "Film-2",
      "year": "2002",
      "country": "Germany",
      "director": [
        "Director-5",
        "Director-6"
      ]
    },
    ...
  ]
}
```

Listing 8 Extract of the 1000 films input in JSON used for the evaluation. `...` represents data that has been omitted from the extract for space limit reasons.

```
<films>
   <film id="1">
      <name>Film-1</name>
      <year>2000</year>
      <country>Germany</country>
      <directors>
         <director>Director-6</director>
      </directors>
   </film>
   <film id="2">
      <name>Film-2</name>
      <year>2010</year>
      <country>Germany</country>
      <directors>
         <director>Director-2</director>
         <director>Director-4</director>
         <director>Director-3</director>
      </directors>
   </film>
   ...
</films>
```

Listing 9 Extract of the 1000 films input in XML used for the evaluation. `...` represents data that has been omitted from the extract for space limit reasons.

```
{
  "data": [
    {
      "id": "be-002157",
      "type": "Repository",
      "attributes": {
        "name": "Kazerne Dossin: Memoriaal...",
        "parallelFormsOfName": [
          "Kazerne Dossin: Memorial..."
        ],
        "otherFormsOfName": [
          "Kazerne Dossin: Memoriaal..."
        ],
        "address": {
          "streetAddress": "Goswin de Stassartstraat 153",
          "city": "Mechelen",
          "region": "Flanders",
          "postalCode": "2800",
          "country": "Belgium",
          "countryCode": "BE",
          "email": [
            "info@kazernedossin.eu",
            "archives@kazernedossin.eu"
          ],
          "telephone": [
            "(+32) 15 29 06 60"
          ],
          "url": [
            "https://www.kazernedossin.eu/EN/"
          ]
        },
        "history": "Kazerne Dossin was built in 1756...",
        "holdings": "In 2003, the archives of the JMDR...",
        "openingTimes": "Kazerne Dossin Museum: Monday...",
        "conditions": "1) If you would like to consult the
            ↪ archive...",
        "reproductionServices": "If you have specific
            ↪ questions...",
        "geo": {
          "type": "Point",
          "coordinates": [
            51.03406,
            4.47928
          ]
        }
      },
      "relationships": {
        "country": {
          "data": {
            "id": "be",
            "type": "Country"
          }
        }
      },
      ...
    }
  ]
}
```

Listing 10 Extract of the EHRI institutions input in JSON used for the evaluation. `...` represents data that has been omitted from the extract for space limit reasons.

Appendix B. Descriptive statistics for the evaluation described in Section 6.1

Input	Engine	Variable	n	$\bar{x}$	$\tilde{x}$	s	min	max
JSON Films 1000 entries	ShExML-v0.3.2	Elapsed time (ms)	30	30488.20000	30572.00000	2015.11590	26696.00000	34699.00000
		CPU Kernel (s)	30	0.66933	0.67500	0.09801	0.49000	0.88000
		CPU Percentage	30	115.26667	115.00000	0.90719	114.00000	117.00000
		CPU User (s)	30	32.46033	32.36000	0.95310	30.48000	35.03000
		MaxMemory (KB)	30	499383.60000	499742.00000	15019.11828	470328.00000	526156.00000
	ShExML-v0.3.3	Elapsed time (ms)	30	19220.23333	18984.50000	1190.45815	17780.00000	22109.00000
		CPU Kernel (s)	30	0.52833	0.52000	0.06884	0.39000	0.76000
		CPU Percentage	30	122.16667	122.00000	1.08543	120.00000	125.00000
		CPU User (s)	30	21.49000	21.29000	0.70155	20.35000	23.26000
		MaxMemory (KB)	30	529061.33333	524394.00000	21785.55926	487264.00000	584120.00000
	ShExML-v0.4.0	Elapsed time (ms)	30	2888.56667	2823.00000	202.72721	2670.00000	3418.00000
		CPU Kernel (s)	30	0.26767	0.26000	0.05888	0.17000	0.42000
		CPU Percentage	30	219.30000	217.00000	10.87278	208.00000	253.00000
		CPU User (s)	30	5.80333	5.61500	0.76290	4.71000	7.48000
		MaxMemory (KB)	30	254791.86667	253972.00000	11966.66950	229668.00000	291692.00000
	ShExML-v0.4.2	Elapsed time (ms)	30	2935.83333	2843.00000	583.90600	2496.00000	5740.00000
		CPU Kernel (s)	30	0.25767	0.24000	0.05643	0.18000	0.39000
		CPU Percentage	30	207.86667	207.00000	7.54176	196.00000	227.00000
		CPU User (s)	30	5.45033	5.36000	0.53050	4.67000	7.49000
		MaxMemory (KB)	30	252594.66667	253438.00000	9881.96347	236272.00000	277340.00000
	ShExML-v0.5.1	Elapsed time (ms)	30	2591.50000	2527.00000	242.94894	2271.00000	3220.00000
		CPU Kernel (s)	30	0.21767	0.22500	0.05667	0.10000	0.31000
		CPU Percentage	30	201.70000	199.50000	9.42539	186.00000	221.00000
		CPU User (s)	30	5.08900	5.06000	0.50569	4.23000	6.54000
		MaxMemory (KB)	30	192980.93333	192632.00000	3794.64765	185368.00000	200164.00000

Table 3

This table shows the descriptive statistics for the execution time (measured in milliseconds) of the tested engine for the JSON films input 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.

Input	Engine	Variable	n	$\bar{x}$	$\tilde{x}$	s	min	max
XML Films 1000 entries	ShExML-v0.3.2	Elapsed time (ms)	30	107352.33333	106644.50000	5239.70542	100762.00000	117625.00000
		CPU Kernel (s)	30	1.04233	1.05500	0.11013	0.74000	1.23000
		CPU Percentage	30	107.36667	107.00000	0.49013	107.00000	108.00000
		CPU User (s)	30	111.59133	111.78500	4.65845	103.99000	125.72000
		MaxMemory (KB)	30	449475.06667	440568.00000	21540.67984	415636.00000	502536.00000
	ShExML-v0.3.3	Elapsed time (ms)	30	26724.26667	26358.00000	690.67777	26144.00000	28271.00000
		CPU Kernel (s)	30	0.53667	0.53000	0.07712	0.39000	0.72000
		CPU Percentage	30	121.06667	121.00000	0.52083	120.00000	122.00000
		CPU User (s)	30	31.78433	31.86500	0.93517	30.27000	33.73000
		MaxMemory (KB)	30	463119.20000	461158.00000	18609.17452	419384.00000	502620.00000
	ShExML-v0.4.0	Elapsed time (ms)	30	22787.36667	22858.00000	1463.50730	20980.00000	26207.00000
		CPU Kernel (s)	30	0.42267	0.42000	0.05552	0.30000	0.59000
		CPU Percentage	30	125.30000	125.00000	0.87691	124.00000	127.00000
		CPU User (s)	30	26.74133	26.87500	0.75558	25.07000	27.95000
		MaxMemory (KB)	30	341774.80000	339024.00000	15982.38534	313932.00000	376128.00000
	ShExML-v0.4.2	Elapsed time (ms)	30	21596.10000	21195.00000	906.07613	20770.00000	24324.00000
		CPU Kernel (s)	30	0.41333	0.41500	0.07341	0.27000	0.62000
		CPU Percentage	30	123.83333	124.00000	1.08543	121.00000	126.00000
		CPU User (s)	30	26.19167	26.17000	0.91638	24.21000	28.22000
		MaxMemory (KB)	30	343694.26667	337638.00000	18843.19582	309764.00000	375540.00000
	ShExML-v0.5.1	Elapsed time (ms)	30	4867.26667	4825.50000	244.74800	4663.00000	5953.00000
		CPU Kernel (s)	30	0.35167	0.33500	0.06608	0.24000	0.51000
		CPU Percentage	30	163.80000	164.00000	3.37741	153.00000	171.00000
		CPU User (s)	30	7.19900	7.09500	0.31453	6.80000	8.13000
		MaxMemory (KB)	30	421860.53333	418216.00000	19650.89765	387988.00000	469364.00000

Table 4

This table shows the descriptive statistics for the execution time (measured in milliseconds) of the tested engine for the XML films input 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.

Input	Engine	Variable	n	$\bar{x}$	$\tilde{x}$	s	*min*	*max*
EHRI institutions (JSON)	ShExML-v0.3.2	Elapsed time (ms)	1	2188032	2188032	0	2188032	2188032
		CPU Kernel (s)	1	45.53000	45.53000	0	45.53000	45.53000
		CPU Percentage	1	105.00000	105.00000	0	105.00000	105.00000
		CPU User (s)	1	2261.28000	2261.28000	0	2261.28000	2261.28000
		MaxMemory (KB)	1	1972528.00000	1972528.00000	0	1972528.00000	1972528.00000
	ShExML-v0.3.3	Elapsed time (ms)	30	157217.16667	154946.00000	5966.55622	149673.00000	171430.00000
		CPU Kernel (s)	30	3.00433	3.03500	0.18511	2.70000	3.41000
		CPU Percentage	30	110.30000	110.00000	0.46609	110.00000	111.00000
		CPU User (s)	30	162.60067	163.47000	2.63503	157.86000	167.26000
		MaxMemory (KB)	30	1074020.93333	1089692.00000	56317.43558	978308.00000	1158068.00000
	ShExML-v0.4.0	Elapsed time (ms)	30	7434.16667	7376.50000	302.49162	6757.00000	8132.00000
		CPU Kernel (s)	30	0.57833	0.55000	0.13432	0.40000	0.96000
		CPU Percentage	30	269.90000	270.00000	4.95740	261.00000	280.00000
		CPU User (s)	30	19.12867	18.85500	1.51962	16.75000	23.19000
		MaxMemory (KB)	30	689118.40000	686316.00000	32148.11219	629764.00000	761776.00000
	ShExML-v0.4.2	Elapsed time (ms)	30	6228.93333	6230.50000	138.71279	5997.00000	6611.00000
		CPU Kernel (s)	30	0.50900	0.49000	0.06261	0.39000	0.63000
		CPU Percentage	30	249.10000	248.50000	5.58539	238.00000	258.00000
		CPU User (s)	30	14.87800	14.90000	0.71282	13.71000	16.04000
		MaxMemory (KB)	30	679173.20000	686988.00000	22991.05882	627304.00000	718052.00000
	ShExML-v0.5.1	Elapsed time (ms)	30	5601.83333	5623.00000	197.05663	5283.00000	6124.00000
		CPU Kernel (s)	30	0.41333	0.41000	0.07194	0.24000	0.57000
		CPU Percentage	30	259.13333	259.50000	6.94179	245.00000	271.00000
		CPU User (s)	30	13.43867	13.24000	0.67281	12.06000	15.02000
		MaxMemory (KB)	30	464956.26667	460516.00000	27515.69036	421024.00000	523796.00000

Table 5

This table shows the descriptive statistics for the execution time (measured in milliseconds) of the tested engine for the EHRI institutions input 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.

Appendix C. Results for the evaluation described in Section 6.2

Engine	Input	Elapsed time (ms)
ShExML-v0.2.7	q9	180017
ShExML-v0.2.7	q10	180014
ShExML-v0.2.7	q11	180016
ShExML-v0.2.7	q12	180016
sparql-generate-2.0.9	q9	1771
sparql-generate-2.0.9	q10	2238
sparql-generate-2.0.9	q11	3484
sparql-generate-2.0.9	q12	1507
rmlmapper-v4.12.0	q9	1681
rmlmapper-v4.12.0	q10	2086
rmlmapper-v4.12.0	q11	3298
rmlmapper-v4.12.0	q12	1248
sparql-anything-0.4.1	q9	1993
sparql-anything-0.4.1	q10	2741
sparql-anything-0.4.1	q11	3852
sparql-anything-0.4.1	q12	1611
ShExML-v0.5.1	q9	3500
ShExML-v0.5.1	q10	3882
ShExML-v0.5.1	q11	6471
ShExML-v0.5.1	q12	2753
sparql-generate-2.1.0	q9	2213
sparql-generate-2.1.0	q10	2628
sparql-generate-2.1.0	q11	3816
sparql-generate-2.1.0	q12	1967
rmlmapper-6.5.1	q9	3467
rmlmapper-6.5.1	q10	4123
rmlmapper-6.5.1	q11	5259
rmlmapper-6.5.1	q12	2787
sparql-anything-0.9.0	q9	1647
sparql-anything-0.9.0	q10	2884
sparql-anything-0.9.0	q11	4010
sparql-anything-0.9.0	q12	1791

Table 6

This table shows the average results for the elapsed time in milliseconds of the discrete inputs experiment obtained after running each input against each engine three times. Results for the engine ShExML-v0.2.7 under the four inputs reached the timeout of 3 minutes.

Engine	Input	Elapsed time (ms)
ShExML-v0.2.7	10	1654
ShExML-v0.2.7	100	2713
ShExML-v0.2.7	1000	57127
ShExML-v0.2.7	10000	180017
ShExML-v0.2.7	100000	180024
ShExML-v0.2.7	1000000	180064
sparql-anything-0.4.1	10	1145
sparql-anything-0.4.1	100	1207
sparql-anything-0.4.1	1000	1359
sparql-anything-0.4.1	10000	2346
sparql-anything-0.4.1	100000	8140
sparql-anything-0.4.1	1000000	74924
rmlmapper-v4.12.0	10	881
rmlmapper-v4.12.0	100	983
rmlmapper-v4.12.0	1000	1183
rmlmapper-v4.12.0	10000	2156
rmlmapper-v4.12.0	100000	6542
rmlmapper-v4.12.0	1000000	52285
sparql-generate-2.0.9	10	1114
sparql-generate-2.0.9	100	1206
sparql-generate-2.0.9	1000	1431
sparql-generate-2.0.9	10000	2412
sparql-generate-2.0.9	100000	6469
sparql-generate-2.0.9	1000000	80734
ShExML-v0.5.1	10	1525
ShExML-v0.5.1	100	1644
ShExML-v0.5.1	1000	2158
ShExML-v0.5.1	10000	3543
ShExML-v0.5.1	100000	11158
ShExML-v0.5.1	1000000	69564
sparql-anything-0.9.0	10	1625
sparql-anything-0.9.0	100	1520
sparql-anything-0.9.0	1000	1744
sparql-anything-0.9.0	10000	2821
sparql-anything-0.9.0	100000	8624
sparql-anything-0.9.0	1000000	71299
rmlmapper-6.5.1	10	2517
rmlmapper-6.5.1	100	2406
rmlmapper-6.5.1	1000	2587
rmlmapper-6.5.1	10000	4289
rmlmapper-6.5.1	100000	9632
rmlmapper-6.5.1	1000000	82971
sparql-generate-2.1.0	10	1779
sparql-generate-2.1.0	100	1665
sparql-generate-2.1.0	1000	1901
sparql-generate-2.1.0	10000	2620
sparql-generate-2.1.0	100000	6543
sparql-generate-2.1.0	1000000	60203

Table 7

This table shows the average results for the elapsed time in milliseconds of the incremental input experiment obtained after running each input against each engine three times. Results for the engine ShExML-v0.2.7 under input sizes 10000, 100000, 1000000 reached the timeout of 3 minutes.