RMLWeaver-JS: An algebraic mapping engine in the KGCW Challenge 2024

Sitt Min Oo[1,*], Tristan Verbeken[1] and Ben De Meester[1]

[1]IDLab, Dept. Electronics & Information Systems, Ghent University – imec, Belgium

Abstract

We present the Knowledge Graph Construction Workshop (KGCW) Challenge 2024 results of our proof of concept mapping engine, RMLWeaver-JS, implemented in JavaScript and based on the reactive programming paradigm. RML documents are translated into a mapping plan consisting of algebraic mapping operators, which RMLWeaver-JS uses to execute the mapping workload. RMLWeaver-JS is evaluated for Track 2 on performance for the Knowledge Graph Construction Challenge for CSV files. The results of the challenge showed that RMLWeaver-JS has a constant memory usage across different workloads, and scales linearly regarding CPU usage and execution time. However, the results also show that the execution time of RMLWeaver-JS greatly depends on the generated mapping plan. As future works, we will focus on the optimizations of the generated mapping plan.

Keywords

Mapping engine, mapping algebra, RML, knowledge graph construction

1. Introduction

The Knowledge Graph Construction Workshop (KGCW) Challenge 2024 [1] presents 2 tracks of challenges measuring the different aspect of mapping engines implementing RML: 1) feature compliance with the RML specification, and 2) the performance of the mapping engine implementation based on different workloads.

For this challenge, we implemented an algebraic mapping engine, RMLWeaver-JS[1], as a proof of concept implementation utilizing algebraic mapping plans while focusing on the performance of the mapping engine. The implementation is based on RML.io's RML specification v1.1.1 [2] with limited support for logical sources and reference iterators. Therefore, we did not participate in the Track 1 challenge, which evaluates the mapping engine's conformance to W3C Knowledge Graph Construction Community Group RML specification [3]. The engine is evaluated in the Track 2 challenge on the performance of the mapping engines, measuring memory usage, CPU usage, and execution time. However, due to the limitations of the engine implementation, we

ESWC 2024: Extended Semantic Web Conference May 26–30, 2024, Hersonissos, Greece

*Corresponding author.

x.sittminoo@ugent.be (Sitt Min Oo); tristan.verbeken@ugent.be (T. Verbeken); ben.demeester@ugent.be (B. De Meester)

https://ben.de-meester.org/#me (B. De Meester)

0000-0001-9157-7507 (Sitt Min Oo); 0000-0003-0248-0987 (B. De Meester)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]https://github.com/RMLio/rmlweaver-js

[2]RML specification 2022: https://rml.io/specs/rml/v/1.1.1/

[3]https://w3id.org/rml/portal

did not evaluate RMLWeaver-JS for the GTFS [2] test cases.

The evaluation results show that RMLWeaver-JS maintains a constant memory usage for mapping workloads without joins, with a linear increase in execution time and full CPU usage for increasing input data size. For mapping workloads with joins, RMLWeaver-JS has a linear increase in memory usage, CPU usage and execution time.

Section 2 discusses the mapping pipeline, where an RML document is first translated to a mapping plan, which RMLWeaver-JS uses to map heterogeneous data to RDF data. Section 3 presents the evaluation set-up. Section 4 presents the result of the evaluation of RMLWeaver-JS in Track 2 of KGCW challenge, and finally, we conclude in Section 5, including a discussion for future work on algebraic mapping engines.

2. Algebraic Mapping Engine Pipeline

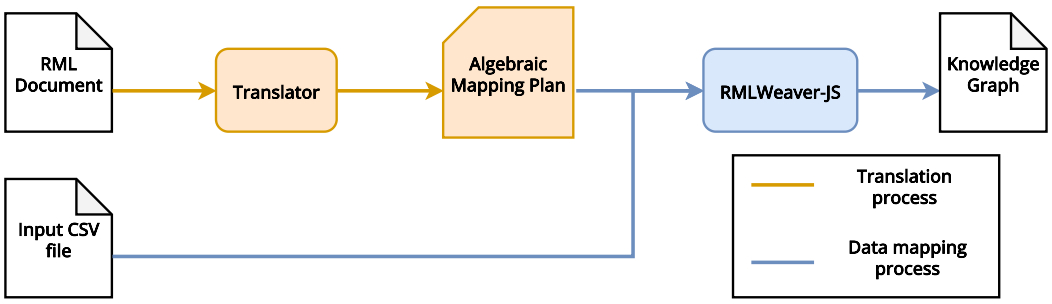

Figure 1: Algebraic mapping engine pipeline where an RML document is first translated into an algebraic mapping plan which is used to generate the KG.

The algebraic mapping engine pipeline consists of two stages: 1) translate RML documents to a mapping plan consisting of algebraic operators, and 2) execute the generated mapping plan.

The separation of the *interpretation* and the *execution* step allows reusing the interpretation step in different development contexts (i.e. different programming languages and frameworks can execute the same mapping plan). Furthermore, depending on the structure of the generated mapping plan, the performance of the mapping engine can be improved.

In order to show the correlation between the structure of the mapping plan and the performance of the mapping engine, we made the following choices in the implementation languages used for the *interpretation* and the *execution* engines.

The *interpretation* engine, which translates RML documents to mapping plans, is written in Rust [4]. We decided to use Rust for the *interpretation* engine due to (i) its ability to be compiled for a multitude of runtimes (e.g. into WebAssembly to be used by JavaScript); (ii) its robust feature support in writing CLI [5] applications; and (iii) low memory footprint as it does not have a memory garbage collector. This ensures that the whole pipeline of knowledge graph

[4]Rust: https://www.rust-lang.org/
[5]Rust Clap: https://github.com/clap-rs/clap

construction does not suffer from high memory usage for fast mapping plan translation. The mapping process described by the RML document is translated to a mapping plan consisting of several algebraic operators, as described in our previous work on algebraic mapping operators [3]. For the KGCW challenge, the interpretation engine only supports RML.io's RML specification, v1.1.1.

The *execution* engine, which executes the mapping plan and generates the KG from heterogeneous data sources, is implemented in JavaScript with a reactive programming paradigm. We utilized RxJS [6] to enable reactive programming for the mapping plan execution. Reactive programming ensures that we process the input data one record at a time in streaming fashion, which lowers the memory usage throughout the mapping process. We used JavaScript to implement the *execution* engine to show that even engine implementations in interpreted languages could execute KG construction workloads with reasonable performance, as shown in Section 4.

As it is a proof-of-concept implementation, the *execution* engine has the following limitations for this challenge: i) it can only process CSV input files, ii) it does not ignore empty values, and iii) it cannot deduplicate the generated KG triples. Thus, due to the aforementioned limitations, GTFS test cases could not be run since they test the engine's ability to process a combination of different input data formats.

Figure 1 illustrates the mapping pipeline deployed for the challenge. The *interpretation* engine is used to first translate RML documents into a mapping plan. Afterwards, the *execution* engine uses the generated mapping plan to execute the mapping process described in the RML document.

3. Experiment

The experiment environment for KGCW challenge 2024 is a virtual machine provided by Orange [7]. The machine has an 64 bit architecture, and it is configured with an Intel(R) Xeon(R) Gold 6161 CPU at 2.20GHz with 4 cores, 16765 MB of RAM memory, and 150 GB of storage space. The operating system of the machine is Ubuntu 22.04.03 LTS.

The pipeline as described in Section 2 is benchmarked for part 1 of the Track 2 challenge, measuring the performance of RMLWeaver-JS based on Knowledge Graph Construction Parameters. For the experiment, we execute the algebraic mapping pipeline with the execution tools provided by the KGCW challenge [1], isolated by using a Docker container[8].

We use the default settings of 3 runs per experiment and collected the median measurements as the results. Since the engine generates multiple output files to write the generated triples, the generated results are aggregated into a single file first before comparison with the baseline results of the challenge. We compare the aggregated output results with the baseline results provided by the challenge to ensure the correctness of our algebraic mapping engine.

[6] RxJS: https://rxjs.dev/
[7] Orange Telecom: https://www.orange.be/
[8] Docker: https://www.docker.com/

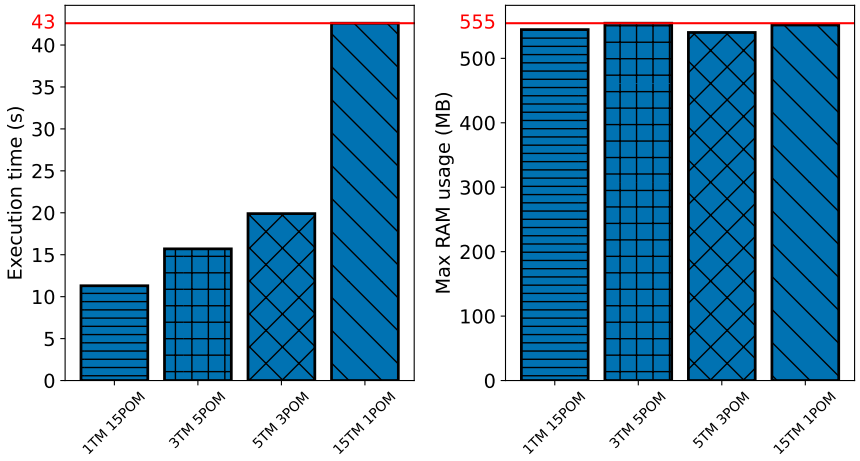

Figure 2: RMLWeaver-JS has a significant increase in execution time, at 43 seconds, for the test case with 15 triples maps and 3 predicate object maps despite almost constant memory usage across the different test cases.

4. Results

The metrics are collected and uploaded to Zenodo[9]. The performance results of RMLWeaver-JS for Track 2 on Knowledge Graph parameters benchmarks are as we expected for a mapping engine implemented in JavaScript. To summarize, RMLWeaver-JS has a constant memory usage even when the parameter values are changed for each groups of test cases. CPU usage is only 25% (100% CPU usage is achieved by multiplying execution time with the number of cores available on the system) despite the presence of four cores on the virtual machine provided for the challenge. This limitation is due to RMLWeaver-JS not utilizing all four cores of the CPU on the virtual machine, since JavaScript is single-threaded (when not using Web Workers[10]). Thus, we can still conclude that RMLWeaver-JS has a 100% CPU utilization on a single core. We plotted a bar graph in Figure 2 where RMLWeaver-JS performed poorly in execution time despite the usage of only a single data source in the test cases. Table 1 shall be used to explain the general behaviour of RMLWeaver-JS across the different test cases.

Figure 2 shows the test case where the number of triples maps and predicate object maps are the independent variables of the experiment. RMLWeaver-JS has a significant increase in execution time of 43 seconds for the test case with 15 triples maps and 3 predicate-object maps despite the constant memory usage. The constant memory usage is explainable by the reactive programming paradigm implemented by RMLWeaver-JS where the data input is processed one at a time in a streaming manner. The increase in execution time is due to the way the interpreter engine translates RML documents into mapping plans. The RML document provided with the challenge contains several definitions of logical sources, despite all the logical sources referencing the same *data.csv* file, with the same iterator. The interpreter engine generates a

[9]https://zenodo.org/doi/10.5281/zenodo.11209233
[10]Web workers: https://html.spec.whatwg.org/multipage/workers.html

Table 1
Median measurements of RMLWeaver-JS's performance in terms of execution time (s), CPU (s), and maximum RAM (MB) usage. The output is also checked with the ground truth provided by the challenge.

Test cases	Execution time (s)	CPU usage (s)	Peak RAM (MB)	Output correct? (Y/N)
Records (20 col)				
10K rows	2.56	3.34	476	Yes
100K rows	13.51	14.76	485	Yes
1M rows	112.05	125.52	505	Yes
11M rows	1116.40	1249.97	544	Yes
Duplicates				
0 percent	12.90	14.70	534	Yes
100 percent	12.70	14.60	531	No
Empty				
0 percent	12.70	14.50	530	Yes
100 percent	12.60	14.40	531	No
Joins 1-N				
1-10 0 percent	15.60	17.60	664	Yes
1-10 100 percent	30.00	33.60	638	Yes
Joins N-1				
10-1 0 percent	15.60	17.80	656	Yes
10-1 100 percent	30.00	33.80	636	Yes
Joins N-M				
5-5 25 percent	35.16	38.80	673	Yes
5-5 100 percent	82.00	90.90	814	Yes
10-5 25 percent	35.20	39.50	661	Yes
10-5 100 percent	82.30	91.00	781	Yes

new source operator for each of the logical sources encountered in the RML document. This results in RMLWeaver-JS reading the input file N times for the N number of triples maps defined in the RML document. We can potentially improve the performance by enabling the detection of semantically similar logical sources in the interpreter engine and generating only *unique* source operators.

Table 1 contains the median measurement of each group of test cases, some results are omitted since they do not show interesting trends to explain the behaviour of RMLWeaver-JS for the performance measured.

With the linearly increasing number of records, with 20 columns, RMLWeaver-JS experience a linear increase in execution time. A 10 times increase in the number of rows results in a 10 times increase in execution time, approximately.

For the test cases testing on *empty* and *duplicate* values, RMLWeaver-JS does not produce the correct RDF triples output compared to the ground truth. This limitation arises due to RMLWeaver-JS not supporting the handling of *empty* values, nor does it *deduplicate* the generated triples.

For test cases on joins, RMLWeaver-JS have the same performance across all metrics depending on the $min(N, M)$ in test case Join N-M. This can be explained as follows: A test case Join N-M has two data sources with S_1 and S_2, where N records in S_1 has M records from S_2, with which it matches to be joined. RMLWeaver-JS employs a hash-join approach when joining data from

two different sources. Two hash maps, one for each of the two sources, are used internally by RMLWeaver-JS for bookkeeping when joining records from the two different data sources. For example, when joining on attribute A, and provided $N < M$ and N records come from S_1 and M records come from S_2. The following condition is necessary for RMLWeaver-JS to have similar performance as shown in the table: N records for a specific attribute A need to arrive first at the join operator and be stored in the hash map $HashMap_{S_1}(A)$ from source S_1. This results in a lower amortized cost of having to only loop through N times to join the records arriving from source S_2. Otherwise, if M records from S_2, for attribute A, arrives first with $M > N$, it will take a longer amortized time to produce the join results where each new arriving records from S_1 will have to loop through at least M times. This explains the similar performance of RMLWeaver-JS for both Join 5-5 and Join 10-5 in terms of maximum RAM and CPU usage, and the execution time.

5. Conclusion

In this work, we participated in KGCW Challenge 2024, for Track 2 on Knowledge Graph Construction parameters, to evaluate the performance of RMLWeaver-JS in terms of execution time, maximum RAM and CPU usage. The results demonstrated that RMLWeaver-JS has a constant memory usage of around 500 MB despite the increasing number of rows in the input data source. RMLWeaver-JS also has an efficient usage of CPU by maintaining a 100% usage for a single core since it is implemented in JavaScript, which is single-threaded.

The constant memory usage, despite the increase in the size of the input data, and an efficient usage of CPU demonstrates the viability of implementing mapping engines in interpreted language like JavaScript in web browsers. Implementation of mapping engines in JavaScript could potentially empower both the client and the server web applications with semantic knowledge from heterogeneous data sources.

Furthermore, we also identified a potential solution to the performance bottleneck suffered as a result of badly generated mapping plans by the interpreter engine. The interpreter engine could generate unique source operators (instead of its current behaviour of creating duplicate operators for each identical source), thus, reducing the number of data sources being iterated over, which leads to increase in performance by RMLWeaver-JS by reducing execution time. This also shows the potential of utilizing algebraic mapping plans, where the mapping engines benefit from the optimizations done on the algebraic mapping plans, without any changes to the implementation of the mapping engine.

As a future work, the interpreter engine could be improved with mappings partition as done by Morph-KGC [4], and employing heuristic-based planning just like SDM-RDFizer [5]. These optimization techniques could improve the performance of RMLWeaver-JS without changing its implementation, as the mapping plan optimizations could be done at the interpretation stage.

References

[1] D. Van Assche, D. Chaves-Fraga, A. Dimou, U. Serles, A. Iglesias, KGCW 2024 Challenge @ ESWC 2024, 2024. doi:10.5281/zenodo.10973433.

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

[3] Sitt Min Oo, B. De Meester, R. Taelman, P. Colpaert, Towards Algebraic Mapping Operators for Knowledge Graph Construction, in: I. Fundulaki, K. Kouji, D. Garijo, J. M. Gomez-Perez (Eds.), 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), 2023. URL: https://ceur-ws.org/Vol-3632/ISWC2023_paper_412.pdf.

[4] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M. Pérez, O. Corcho, Morph-kgc: Scalable knowledge graph materialization with mapping partitions, Semantic Web 15 (2022) 1–20. doi:10.3233/SW-223135.

[5] E. Iglesias, M. Vidal, S. Jozashoori, D. Collarana, D. Chaves-Fraga, Empowering the SDM-RDFizer Tool for Scaling Up to Complex Knowledge Graph Creation Pipelines, Semantic Web Journal (2024) 1–28. doi:10.3233/sw-243580.