Performance Results of FlexRML in the KGCW Challenge 2024

Michael Freund[1,*], Sebastian Schmid[2], Rene Dorsch[1] and Andreas Harth[1,2]

[1]*Fraunhofer Institute for Integrated Circuits IIS, Nürnberg, Germany*
[2]*Friedrich-Alexander-Universität Erlangen-Nürnberg, Nürnberg, Germany*

Abstract
The Knowledge Graph Construction Workshop introduced a challenge to evaluate the performance metrics of different RML interpreters using a set of standardized benchmarks. We participated in the challenge's performance track with our RML interpreter, FlexRML, and report the median execution time and peak memory consumption over five runs on the provided virtual machine using the challenge tool. Through this challenge, we were able to identify weaknesses in FlexRML, such as its current support for CSV data only, lack of support for the latest RML vocabulary, and a crash that occurs when the system executing the mapping runs out of memory. These are issues that we plan to address in future releases of FlexRML.

Keywords
Knowledge Graph Construction, RDF Mapping Language, KGCW Challenge

1. Introduction

The construction of Knowledge Graphs (KGs) by creating Resource Description Framework (RDF) data from various sources such as CSV, JSON, or relational databases is becoming increasingly important. This growth is driven by the overall increase in data volumes and the need to integrate heterogeneous data sources [1]. The most common method of mapping non-RDF data to RDF is a declarative approach using the RDF Mapping Language (RML) [2]. RML mappings specify the input data sources, their encodings, the ontologies to be used, and the overall structure of the output RDF triples. Using these mappings, RML interpreters then transform the data sources into the specified RDF. The RML method is widely used due to the availability of many well-maintained RML interpreters, including the RMLMapper[1], RocketRML [3], SDM-RDFizer [4], and Morph-KGC [5].

Depending on the use case, the constraints of the machine performing the mapping, and the structure and parameters of the source data, such as overall size, number of duplicates, and complexity of the RML mappings, some RML interpreters may be more suitable than others. The suitability of RML interpreters is influenced by factors such as the programming language used for implementation and the internal mapping and data transformation algorithms. To

KGCW'24: 5th International Workshop on Knowledge Graph Construction, May 27, 2024, Crete, Greece
*Corresponding author.
michael.freund@iis.fraunhofer.de (M. Freund)
0000-0003-1601-9331 (M. Freund); 0000-0002-5836-3029 (S. Schmid); 0000-0001-6857-7314 (R. Dorsch); 0000-0002-0702-510X (A. Harth)
© 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/rmlmapper-java

help practitioners choose the right RML interpreter for their project, the Knowledge Graph Construction Workshop (KGCW) has introduced the KGCW Challenge to enable fair comparison. The KGCW Challenge consists of a dataset and corresponding RML mappings divided into two tracks. The first track covers conformance of the RML interpreters to the new RML specifications, while the second track focuses on mapping performance by evaluating execution time and peak memory consumption.

This paper reports on the results of FlexRML [6], a new RML interpreter built from the ground-up, in the performance track of the KGCW Challenge 2024. The rest of the paper is divided into three sections. Section 2 gives a brief overview of FlexRML in its current state and the technical improvements planned for the next releases. Section 3 reports the performance metrics of FlexRML in the empirical evaluation and a discussion of the results. Finally, Section 4 concludes the paper and outlines future research.

2. FlexRML

FlexRML is a flexible RML interpreter written in C++ that is designed to be usable across the entire network architecture. This means that FlexRML is designed to operate in unconstrained cloud environments, moderately constrained industrial PCs and single-board computers at the network edge, and extremely resource-constrained Internet of Things (IoT) devices and microcontrollers running real-time operating systems. This flexibility sets FlexRML apart from other RML interpreters, which typically focus only on unconstrained cloud environments and are therefore written in high-level programming languages such as Java, JavaScript, or Python. In the following, we provide a brief overview of FlexRML's architecture, current supported features, and planned features for future releases.

2.1. Architecture

When mapping non-RDF data to RDF, FlexRML performs two main steps: the preprocessing step, which optimizes the RML mappings for speed, and the actual mapping step, which executes the optimized RML mappings and generates the RDF data.

Preprocessing Step FlexRML applies well-established mapping optimization strategies to improve overall mapping performance, such as self-join elimination or mapping normalization, and replaces joins with reference conditions whenever possible. In addition, FlexRML uses a result size estimation algorithm based on independent Bernoulli sampling. The algorithm generates a small sample from the original non-RDF data sources using a simple random sampling approach, performs the mapping on the sample, enumerates all unique RDF triples generated, and based on the result estimates the number of unique RDF triples that will be generated when all source files are mapped. The estimated number of unique RDF triples is used to select correct bit sizes for hash functions and data structures in the rest of the mapping process, which allows to save memory.

Mapping Step The mapping process itself can take advantage of multiple cores, if available, by using the producer-consumer design pattern. In the mapping process, FlexRML generates all triples, uses a hash function with bit sizes of 32, 64, or 128 bits depending on the result of the

estimation process to hash the generated triple, and compares the result to a hash set containing all hashes of RDF triples generated up to that point to remove duplicates. If the generated triple is a duplicate, the triple is discarded, otherwise the triple is written to the output and the hash is added to the hash set. This duplicate removal approach is memory efficient, but carries the risk of hash collisions, which can result in missing RDF triples in the output data. By choosing appropriate bit sizes, the risk of missing output RDF triples can be minimized and has never occurred in the KGCW challenge dataset.

2.2. Current State and Planned Features

The current release, FlexRML 1.0, is available in easy-to-use, pre-built binaries for Debian-based 64 bit systems, arm-based 64 bit systems such as Raspberry Pis[2], and offers a GitHub Repository[3] usable on microcontrollers such as the Arduino Nano 33 IoT and ESP32 via the Arduino IDE. Because the code is open source, FlexRML can also be built locally, giving users access to the latest development features.

The main drawback of FlexRML is that currently only the mapping of CSV source data to RDF is fully supported. This is because we want to be able to run FlexRML across the entire network architecture, and some microcontrollers only support a subset of the C++ standard library, which limits our ability to reuse existing libraries for input file handling. This forces us to implement the file handling logic ourselves, which is complicated and time-consuming. But we are making progress, in the current development build we already partially support mapping JSON data with a subset of JSONPath expressions. In addition, we currently do not fully support the new RML vocabulary terms, only those of the RML-IO specification[4].

In the next releases of FlexRML, we will fully integrate JSON by enhancing our implementation of a JSONPath parser to cover all features. Additionally, we plan to make FlexRML available in web browsers using WebAssembly, which will allow us to extend FlexRML beyond the cloud and directly into user applications. We are also aware that the performance of joins that cannot be substituted by reference conditions is not optimal, and we plan to apply optimization strategies to address this. A full list of planned features and the implementation progress can be found on our roadmap on GitHub[5].

3. Empirical Evaluation

In the following, we discuss the hardware and software used, report the results of the empirical evaluation, and discuss the results.

3.1. Experimental Setup

For the empirical evaluation, we used the virtual machine provided by the organizers of the KGCW Challenge. To execute the mappings, we used the recommended challenge tool, for

[2]https://github.com/wintechis/flex-rml/releases
[3]https://github.com/wintechis/flex-rml-esp32/tree/main
[4]https://github.com/kg-construct/rml-io
[5]https://github.com/wintechis/flex-rml?tab=readme-ov-file#planned-features-for-flexrml

which we included a Dockerfile for FlexRML. The non-RDF data to be transformed into RDF was available in CSV format, and in the default process, it is loaded into a database from which the mapping is performed. Since FlexRML does not support mapping from databases, we adjusted the pipeline to directly map the CSV data. Additionally, FlexRML does not support the newest vocabulary terms, so we adjusted the RML mapping rules accordingly. The challenge tool with FlexRML integrated, the adjusted `metadata.json` files describing the new pipeline, and the adjusted RML mappings used for the evaluation can be found on GitHub[6]. To allow for easy reproducibility, we also included a simple shell script that copies all the adjusted data into the correct directories and needs to be run once the benchmark data has been downloaded. We used a simple Python script to verify the correctness of FlexRML's output against the reference output provided[7]. All performance metrics reported in the following are collected by the challenge tool over five runs.

3.2. Results

The challenge tool evaluated the performance of FlexRML in duplicate values, empty values, the GTFS-Madrid-Bench, joins, mappings, properties, and records.

Duplicated Values The *Duplicated Values* dataset comes in five variants with different percentages of duplicates, ranging from 0 percent to 100 percent in steps of 25 percent. The median runtime and peak memory usage is reported in table 1. All outputs of FlexRML match the expected reference output.

Dataset	Execution Time (sec)	Peak Memory (MiB)
Duplicated Values 0%	8.59	454.48
Duplicated Values 25%	7.46	456.40
Duplicated Values 50%	6.75	417.48
Duplicated Values 75%	6.23	406.35
Duplicated Values 100%	5.73	393.68

Table 1
Performance metrics of FlexRML mapping the *Duplicated Values* dataset.

The results show that FlexRML's runtime and peak memory usage during the mapping process continuously decrease as the number of duplicates increases. This is because the number of unique output RDF triples also decreases, resulting in fewer write operations to disk and thus a reduction in runtime. In addition, fewer hashes need to be stored in the hash set, resulting in lower memory consumption.

Empty Values The *Empty Values* dataset is available in the same variations as the previous dataset, containing empty values ranging from 0 percent to 100 percent of the total dataset size in steps of 25 percent. The median runtime and peak memory consumption are reported in Table 2. The output produced by FlexRML again matches the expected reference output.

The results of the *Empty Values* dataset mirror those of the *Duplicated Values* dataset. As the overall number of empty values increases, the required disk writes are reduced, resulting in

[6]https://github.com/FreuMi/challenge-tool
[7]https://zenodo.org/records/10973433

Dataset	Execution Time (sec)	Peak Memory (MiB)
Empty Values 0%	7.98	458.34
Empty Values 25%	7.06	462.36
Empty Values 50%	5.83	436.75
Empty Values 75%	4.91	415.30
Empty Values 100%	3.80	405.78

Table 2
Performance metrics of FlexRML mapping the *Empty Values* dataset.

faster execution times. In addition, the size of the internal hash set decreases, resulting in less memory consumption.

GTFS-Madrid-Benchmark The *GTFS-Madrid-Benchmark* [7] is used to evaluate the mapping performance with increasing dataset sizes. The benchmark is available in four scales, scale 1, scale 10, scale 100, and scale 1000. The benchmarks testing mixed content mapping, i.e., mapping JSON and XML data, could not be run because FlexRML currently only supports mapping CSV data. The median runtime and peak memory consumption are reported in table 3.

Dataset	Execution Time (sec)	Peak Memory (MiB)
GTFS-Madrid Scale 1	2.97	414.10
GTFS-Madrid Scale 10	24.14	599.30
GTFS-Madrid Scale 100	251.88	2332.76
GTFS-Madrid Scale 1000	-	-

Table 3
Performance metrics of FlexRML mapping the *GTFS-Madrid-Benchmark* dataset.

While running the *GTFS-Madrid-Benchmark* with a scale factor of 1000, FlexRML crashed because the virtual machine doing the mapping ran out of memory. We plan to address this issue in future releases of FlexRML, as it requires us to change the way we store the hashes of generated RDF triples. Specifically, we need to monitor the system's RAM and, when approaching the maximum available RAM, store the additional hashes on disk to avoid a crash.

Additionally, we noticed that the provided reference data does not match the output generated by FlexRML. A closer examination of the data revealed that most of the numerical values are set to 999.999, and additional data types were expected but not declared in the mappings. This issue was also identified in the KGCW Challenge 2023 [8]. When using the mappings and data sources with multiple other RML interpreters, the results match those of FlexRML and show the same mismatches with the reference.

Otherwise, the performance results are as expected. As the size of the source data increases, both execution time and peak memory usage increase.

Joins The *Join* dataset evaluates the performance of RML interpreters when mapping definitions containing different types of joins, namely 1-to-1 joins, 1-to-N joins, N-to-1 joins, and N-to-M joins. The Join dataset contains 33 different variations, varying in the number of joined datasets and the percentage of data in each dataset that needs to be joined. Due to the large number of datasets, we report only selected results in Table 4. All outputs are in line with the reference.

Dataset	Execution Time (sec)	Peak Memory (MiB)
Join 1-1 50%	14.52	526.20
Join 1-10 50%	14.11	489.24
Join 10-1 50%	14.42	514.92
Join 5-10 50%	18.20	526.59
Join 10-5 50%	18.21	543.96

Table 4
Performance metrics of FlexRML mapping selected elements of the *Join* dataset.

The performance results for the *Join* dataset show that FlexRML handles different types of joins with fairly consistent execution times and peak memory usage. For datasets with 50% of the data to be joined, the execution times range from 14.11 seconds to 18.21 seconds, indicating that the join type does not drastically affect the processing time. Peak memory usage shows a small variation, from 489.24 MiB to 543.96 MiB, indicating that memory use is relatively stable across different join variants.

Mappings The *Mappings* dataset is used to evaluate the impact of different structures in the RML mapping rules. Specifically, the mappings vary the number of TripleMaps (TMs) and PredicateObjectMaps (POMs). TMs specify rules for transforming the input data into RDF triples and consist of POMs, which define how the predicate and object of the predefined subject must be generated. The *Mappings* dataset consists of mappings with variants of 1 TM with 15 POMs, 15 TMs with 1 POM each, 3 TMs with 5 POMs each, and 5 TMs with 3 POMs each. The resulting median execution time and peak memory consumption are shown in Table 5. All outputs of FlexRML match the reference output data.

Dataset	Execution Time (sec)	Peak Memory (MiB)
Mappings 1TM / 15POM	6.54	444.65
Mappings 15TM / 1POM	6.34	434.35
Mappings 3TM / 5POM	3.79	486.92
Mappings 5TM / 3POM	4.60	453.81

Table 5
Performance metrics of FlexRML mapping the *Mappings* dataset.

The performance results for the *Mappings* dataset reveal that mappings with fewer TMs but more POMs per TM (1 TM with 15 POMs) have slightly higher execution times and memory usage compared to variants with more TMs but fewer POMs per TM (15 TMs with 1 POM each). The variant with 3 TMs and 5 POMs each achieves the lowest execution time of 3.79 seconds but the highest peak memory usage of 486.92 MiB. This is due to the way multithreading is implemented in FlexRML, where each TM is mapped in a separate thread. This increases memory consumption due to threading overhead but also reduces execution time.

Properties The *Properties* dataset increases the number of columns while keeping the number of rows constant. This means, that the *Properties* dataset evaluates RML interpreters for their ability to handle horizontally scaled dataset sizes. The *Properties* dataset is available in the variants 1M rows with 1 column, 1M rows with 10 columns, 1M rows with 20 columns, and 1M

rows with 30 columns. The results are shown in Table 6. All outputs of FlexRML are verified to match the expected output.

Dataset	Execution Time (sec)	Peak Memory (MiB)
Properties 1M rows / 1 column	5.63	403.58
Properties 1M rows / 10 columns	43.66	750.66
Properties 1M rows / 20 columns	87.01	1141.77
Properties 1M rows / 30 columns	137.30	1658.16

Table 6
Performance metrics of FlexRML mapping the *Properties* dataset.

The performance results for the *Properties* dataset show that both FlexRML's execution time and peak memory consumption increase as the number of columns in the dataset increases, while keeping the number of rows constant at 1 million. The execution time increases from 5.63 seconds for a dataset with 1 column to 137.30 seconds for a dataset with 30 columns, an increase of approximately 25 times. Similarly, the peak memory usage increases from 403.58 MiB to 1658.16 MiB over the same range, an increase of about a factor of 4. These results show that FlexRML's mapping performance is directly affected by horizontally scaled data, with more columns leading to higher computational requirements, more hashes to compute and more RDF triples to write to disk, and increased memory usage due to a larger in-memory hash set. However, the scaling is sublinear, as a 30-fold increase in the number of columns results in only a 25-fold increase in execution time and a 4-fold increase in memory consumption.

Records The *Records* dataset is complementary to the *Properties* dataset, as it evaluates the performance of RML interpreters in handling vertically scaled data. The *Records* dataset keeps the number of columns constant while systematically increasing the number of rows with each variant. The dataset consists of the variants 10k rows with 20 columns, 100k rows with 20 columns, 1M rows with 20 columns, and 10M rows with 20 columns, as shown in Table 7. All outputs of FlexRML again match the expected reference output.

Dataset	Execution Time (sec)	Peak Memory (MiB)
Records 10k rows / 20 columns	1.23	381.51
Records 100k rows / 20 columns	8.28	434.05
Records 1M rows / 20 columns	85.79	1154.07
Records 10M rows / 20 columns	943.34	11275.53

Table 7
Performance metrics of FlexRML mapping the *Records* dataset.

The performance results for the *Records* dataset show that both the execution time and peak memory usage of FlexRML increase as the number of rows in the dataset increases, while the number of columns remains constant at 20. The execution time increases from 1.23 seconds for 10k rows to 943.34 seconds for 10M rows, and the peak memory usage increases from 381.51 MiB to 11275.53 MiB over the same range. While these increases are significant, they are not proportional, with the execution time increasing by a factor of about 767 and memory usage increasing by a factor of about 30 for a 1000-fold increase in dataset size. This indicates that

FlexRML scales more efficiently than a direct linear relationship would suggest, especially in terms of memory usage.

4. Conclusion

The results of FlexRML in the KGCW Challenge 2024 show that the performance metrics for handling the *Duplicated Values* dataset, the *Empty Values* dataset, and the *GTFS-Madrid-Benchmark* are as expected. The number of unique output RDF triples mainly affects memory consumption due to the duplicate removal hash set and execution time due to disk write operations. The *Join* dataset shows stable performance regardless of join type. The *Mappings* dataset shows that multithreading increases memory consumption due to overhead, but reduces execution times. The *Properties* and *Records* datasets show that FlexRML's execution time and memory consumption increase with larger dataset sizes, but only sublinearly, with memory consumption growing much slower than execution time and both less than the increase in dataset size.

Future research plans include combining our result size estimation algorithm with mapping partitioning to further reduce memory consumption, and implementing additional features discussed in this paper and our roadmap.

Acknowledgments

This work was funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) through the Antrieb 4.0 project (Grant No. 13IK015B).

References

[1] L. Wang, Heterogeneous data and big data analytics, Automatic Control and Information Sciences 3 (2017).
[2] A. Dimou, M. Vander Sande, P. Colpaert, et al., RML: A generic language for integrated RDF mappings of heterogeneous data., 7th Workshop on Linked Data on the Web 1184 (2014).
[3] U. Şimşek, E. Kärle, D. Fensel, RocketRML - A NodeJS implementation of a use-case specific RML mapper, arXiv preprint arXiv:1903.04969 (2019).
[4] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, et al., SDM-RDFizer: An RML interpreter for the efficient creation of RDF knowledge graphs, in: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, 2020.
[5] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, et al., Morph-KGC: Scalable knowledge graph materialization with mapping partitions, Semantic Web (2022).
[6] M. Freund, S. Schmid, R. Dorsch, et al., FlexRML: A Flexible and Memory Efficient Knowledge Graph Materializer, in: Extended Semantic Web Conference, Springer, 2024.
[7] D. Chaves-Fraga, F. Priyatna, A. Cimmino, et al., GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain, Journal of Web Semantics 65 (2020).
[8] S. Bin, C. Stadler, KGCW2023 Challenge Report RDFProcessingToolkit / Sansa (2023).