FlexRML: A Flexible and Memory Efficient Knowledge Graph Materializer

Michael Freund[1(✉)], Sebastian Schmid[2], Rene Dorsch[1], and Andreas Harth[1,2]

[1] Fraunhofer Institute for Integrated Circuits IIS, Nürnberg, Germany
{michael.freund,rene.dorsch,andreas.harth}@iis.fraunhofer.de
[2] Friedrich-Alexander-Universität Erlangen-Nürnberg, Nürnberg, Germany

Abstract. We present FlexRML, a flexible and memory efficient software resource for interpreting and executing RML mappings. As a knowledge graph materializer, FlexRML can operate on a wide range of systems, from cloud-based environments to edge devices, as well as resource-constrained IoT devices and real-time microcontrollers. The primary goal of FlexRML is to balance memory efficiency with fast mapping execution. This is achieved by using C++ for the implementation and a result size estimation algorithm that approximates the number of N-Quads generated and, based on the estimate, optimizes bit sizes and data structures used to save memory in preparation for mapping execution. Our evaluation shows that FlexRML's adaptive bit size and data structure selection results in higher memory efficiency compared to conventional methods. When benchmarked against state-of-the-art RML processors, FlexRML consistently shows lower peak memory consumption across different datasets while delivering faster or comparable execution times.

Resource type: RML Processor
License: GNU AGPLv3
DOI: https://doi.org/10.5281/zenodo.10256148
URL: https://github.com/wintechis/flex-rml

Keywords: Knowledge Graph Construction · RML · Internet of Things

1 Introduction

Knowledge graphs (KGs) [13] have become increasingly popular in both industry and academia. For instance, KGs are used to enhance Large Language Models (LLMs) by supplementing them with additional factual information, thereby countering issues like hallucination [20]. In the context of Industry 4.0, KGs play an important role in enabling semantic interoperability between devices and services [2]. Additionally, as Internet of Things (IoT) devices generate large volumes of raw data, the application of Semantic Web technologies, such as RDF and ontologies, can add context and meaning to the data, which significantly increases its value [29].

A. Meroño Peñuela et al. (Eds.): ESWC 2024, LNCS 14665, pp. 40–56, 2024.
https://doi.org/10.1007/978-3-031-60635-9_3

One approach to constructing KGs, i.e., mapping non-RDF data such as CSV, JSON, or XML to RDF, is to use a generic mapping language such as the RDF Mapping Language (RML) [9]. RML mappings are represented as RDF graphs and outline the tasks for transforming a non-RDF data source, called *logicalSource*, into RDF format. The transformation process is organized by a construct called *triplesMap*. Within a triplesMap, the generation of RDF data is described by two main mappings: the *subjectMap* and the *predicateObjectMap*. The subjectMap is responsible for mapping elements from the logical source to RDF subjects. In turn, the predicateObjectMap handles the creation of predicates and objects for the RDF data by integrating a *predicateMap* that maps to RDF predicates and an *objectMap* that maps to RDF objects. If objects are generated using a join, the objectMap also specifies a *joinCondition*. The RML mappings are used in conjunction with RML processors, which are engines that interpret the mappings and transform non-RDF input data to output RDF data in serializations such as N-Quads. Well known RML processors are the RMLMapper[1], Morph-KGC [3], or the SDM-RDFizer [15].

The current generation of RML processors is implemented using high-level languages such as Java, JavaScript, and Python. The design choice to use high-level languages makes them well-suited for operation in unconstrained environments, such as cloud platforms. However, it also restricts their ability to function effectively in environments with limited memory, such as single-board computers or highly constrained microcontrollers. Additionally, these RML processors may not be optimal for real-time applications that are crucial in Industry 4.0 contexts, where processing needs to be reliably completed within strict time limits [14].

With the emergence of new and evolving domains such as LLMs and Industry 4.0, there is a growing demand for access to both semantic and factual data. To meet this demand, there is a need for an RML processor that is not only fast and memory efficient, but also versatile enough to operate in a variety of environments, whether constrained, unconstrained, or requiring real-time guarantees. The main challenge in developing such a framework is to ensure efficient management of available processing power and memory.

Our resource, FlexRML, is a flexible and memory efficient RML processor developed in C++ to ensure adaptability to various environments with different resource constraints. FlexRML uses an algorithm based on Bernoulli sampling to estimate the result size, specifically the number of unique RDF N-Quads to be generated. This feature allows FlexRML to select the most optimal hash functions for different data structures, balancing improved memory efficiency and increased processing time for KG materialization. From an implementation perspective, FlexRML optimizes at the mapping level using techniques such as self-join elimination, RML mapping normalization, and, where possible, replaces join operations with reference conditions. In cases where joins cannot be replaced, FlexRML uses a hash join algorithm to perform them. To increase memory efficiency, duplicate removal is handled by storing the hash of already generated RDF data, which carries the risk of hash collisions that can result in missing

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

output N-Quads. But the risk can be mitigated by choosing an appropriate bit size when computing hashes.

The key contributions of our work are as follows:

- Introduction of FlexRML, a memory-efficient RML processor capable of mapping data in cloud, edge, and IoT environments.
- Usage of a sampling-based algorithm to estimate N-Quads, enabling optimal hash function selection for improved memory efficiency.
- Comprehensive evaluation of FlexRML's performance against existing RML processors across cloud, edge, and IoT platforms, accompanied by the introduction of a dataset specifically tailored for evaluating IoT devices.

2 Related Work

There are several languages and techniques for building KGs from non-RDF data, notably SPARQL-Generate [19], SPARQL Anything [6], and D-REPR [28]. However, the dominant method relies on R2RML [7], a W3C recommendation, and its extension RML, which supports non-relational data sources such as CSV, JSON, and XML. The adoption of RML is underscored by the number of RML processors available [3,15,23,25], each with different conformance[2] and features, such as mapping partitions. These implementations in high-level languages are suitable for cloud environments, but are less optimal for resource-constrained devices. In contrast, we propose an RML processor designed for a wide range of devices, from cloud and consumer hardware to single-board computers and resource-constrained microcontrollers and IoT devices.

Semantic Web technologies such as RDF, RDFS, and OWL are considered key to semantic interoperability in IoT, as proposed by various authors [12,17]. Many proposed IoT architectures [1,18,22] share a common need to map IoT data to RDF for cloud storage and application use, typically using an edge device such as a gateway for mapping. However, FlexRML offers a simpler alternative by enabling direct data mapping on Internet-connected, constrained devices, enabling direct cloud uploads and bypassing the need for edge RML processors. Nevertheless, in scenarios where edge-based RML processors are required, FlexRML's memory efficiency makes it a viable option for edge deployment as well. The availability of RDF data directly on IoT devices is a prerequisite for device-level semantic interoperability, which can be used to enable data exchange in distributed IoT systems.

3 FlexRML: Architecture and Implementation

The FlexRML architecture integrates both high-level design and specific implementation details, providing a complete picture of how each component works within the system.

FlexRML performs two main steps: the *Preprocessing Step* and the *Mapping Step*, as shown in Fig. 1.

[2] https://rml.io/implementation-report/.

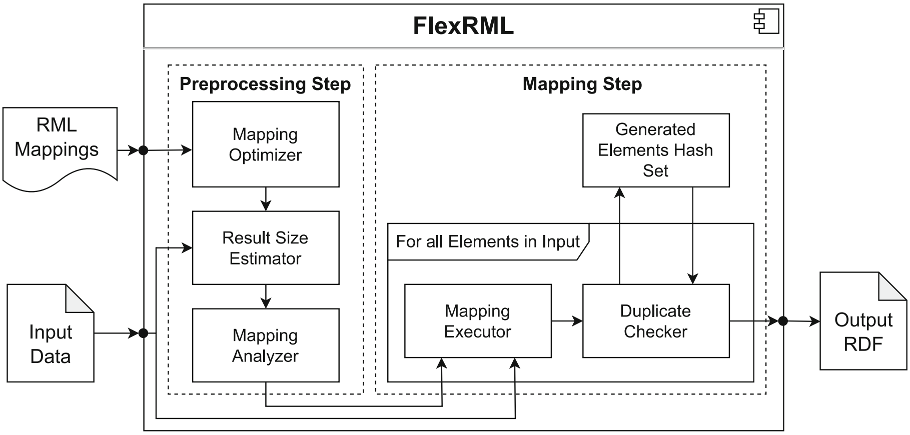

Fig. 1. The Architecture of FlexRML.

3.1 Preprocessing Step

The *Preprocessing Step* is designed to optimize input RML mappings for efficient processing. This involves preparing the correct physical data structures and hash functions, and extracting necessary processing steps from the RML mappings.

Mapping Optimizer. The Mapping Optimizer is designed to increase the efficiency of input RML mappings. This is achieved through a multi-step process: First, the component analyzes the input RML mapping and applies strategies such as self-join elimination [3] or mapping normalization [24]. In addition, the component evaluates the potential to replace join operations with reference conditions [27]. From a technical point of view, the optimizer uses the Serd[3] C library for RDF parsing. The component's functionality includes replacing self-joins with corresponding objectMaps, expanding constant shortcuts or classes, and decomposing complex predicateObjectMaps that may consist of multiple predicateMaps or objectMaps. Where appropriate, the optimizer also replaces standard joins with reference conditions in the RML mapping to improve materialization speed.

Result Size Estimator. The Result Size Estimator uses sampling methods described in Sect. 4 to estimate the number of N-Quads generated by the RML mapping. Based on the estimated total $\hat{T}$, the component determines the appropriate bit size for hash functions. The bit size corresponds to the number of possible unique hash values N and must be chosen according to $\hat{T}$. The decision which hash size to use is based on pre-calculated thresholds derived from the birthday problem in probability theory (see Eq. 1).

$$P = 1 - \frac{N!}{(N - \hat{T})! \cdot N^{\hat{T}}} \approx 1 - e^{-\frac{\hat{T}^2}{2N}} \tag{1}$$

[3] https://github.com/drobilla/serd.

The thresholds are computed to allow a worst-case maximum hash collision probability P of 0.05 percent, which has proven sufficient in our experiments to mitigate missing output RDF statements. Therefore, we have set the thresholds at 2,073 N-Quads for 32-bit hash functions, and 135,835,773 N-Quads for 64-bit hash functions. For estimates exceeding these thresholds, 128-bit hash functions are used. FlexRML uses the CityHash algorithm[4] to perform string hashing operations due to its high hashing speed. The CityHash algorithm is available in 32-bit, 64-bit and 128-bit variants. Using smaller bit sizes for hash computations not only speeds up the process, but also reduces the amount of memory needed to store hashes. The goal is to choose the smallest bit size for the hash function that keeps the risk of hash collisions below the predefined probability threshold. Choosing an optimal bit size is critical, as this will significantly reduce the probability of hash collisions.

Mapping Analyzer. The Mapping Analyzer component processes the normalized input RML mappings and extracts key information. The extracted information includes the number of predicateObjectMaps in each triplesMap, the classes corresponding to the subjectMap, and the details of objectMaps including the joinCondition. The data extracted by the Mapping Analyzer is organized into several arrays. These arrays form the basis of the mapping process and serve as directives to the Mapping Executor.

3.2 Mapping Step

The *Mapping Step* performs the actual mapping of the logical sources to RDF using the Mapping Executor, which interprets the extracted information generated by the Mapping Analyzer, creates N-Quads, and removes duplicates using a Duplicate Checker and a hash set to track hashes of already generated N-Quads.

The Mapping Step is parallelized in FlexRML using the Producer-Consumer design pattern. In this setup, each triplesMap is processed in a separate thread by a Mapping Executor acting as a producer, while the Duplicate Checker operates as a consumer in a separate thread. This design ensures that all generated N-Quads are checked before they are written to the output file, while allowing FlexRML to use multiple threads to speed up the materialization process.

Mapping Executor. The Mapping Executor component maps input data sources into N-Quads. The component processes each data element from the specified inputs, e.g., from a file on the local drive or from in-memory data structures such as the std::string class [8], converts the read element into the internal vector-based data representation, and generates a set of N-Quads for each element. Note that FlexRML currently supports only CSV data as input, due to limited support for the C++ standard library in some microcontroller environments, which restricts the use of existing parsing libraries. The N-Quad generation uses the extracted information from the RML mapping gathered by the Mapping Analyzer. If join operations cannot be substituted with a reference condition at the mapping level, the Mapping Executor performs a join using a hash join algorithm, and therefore must create an index on the data beforehand.

[4] https://opensource.googleblog.com/2011/04/introducing-cityhash.html.

Duplicate Checker. Each generated set of N-Quads is passed to the Duplicate Checker. This component uses a hash function with a bit size of either 32, 64, or 128, as specified by the Result Size Estimator. Each N-Quad is hashed, and the hash value is then evaluated for uniqueness. The evaluation is done by attempting to insert the hash into a hash set. If the hash already exists in the set, indicating a potential duplicate, the corresponding N-Quad is discarded. Otherwise the N-Quad is considered unique and written to the output file. Through this mechanism, the output RDF data is continuously and incrementally built. As new N-Quads are generated, they undergo the same duplication check. However, a risk of this method is the possibility of hash collisions where an N-Quad is mistakenly marked as a duplicate, resulting in its absence in the output. The risk can be mitigated by choosing an appropriate bit size.

4 Estimation of Generated RDF Elements

Estimating the number of RDF elements, such as N-Quads, that will be generated before executing an RML mapping is beneficial. For instance, in FlexRML, the estimate is used to select the appropriate bit size for hash functions. Generally, the estimation can serve other purposes similar to those in classical relational database systems, such as query optimization or the selection of efficient algorithms and access methods [16].

Independent Bernoulli sampling is an established technique for estimating result sizes [11, p. 348], especially for join operations [26]. In general the algorithm consits of following steps:

1. Generate a sample from the original dataset through simple random sampling, where each record is independently and randomly chosen.
2. Count the elements in the sample that exhibit the desired characteristic. The proportion of these elements in the sample is an estimate of their proportion in the entire dataset.
3. Use the identified proportion to estimate the number of elements with the desired characteristic in the entire dataset.

The method allows quick estimations by analyzing the distribution and properties of the data without processing the entire dataset, assuming sample sizes are large [21]. In KG construction, we sample the logical source data, perform the RML mapping on the sample, and enumerate the unique results, before estimating the unique results in the entire dataset.

When performing the estimation, we differentiate between two categories of triplesMaps and their corresponding objectMaps: those that require a join operation and those that do not.

4.1 Estimation of triplesMaps Without Join

To estimate the output size produced by triplesMaps consisting of objectMaps without join we assess the number of unique N-Quads U that will be generated when mapping each element of the logical data source A. Elements in the logical source can be, for example, rows in CSV files or JSON objects in JSON arrays.

To estimate U, we generate a Bernoulli sample S_A drawn from the original dataset A. The sampling process is performed with a given probability p. Using S_A, we perform the mapping and enumerate the generated unique N-Quads U'. Given that each element's inclusion in our sample is defined by p, the expected count of U' is U multiplied by p, or $E[U'] = U \cdot p$. From this, we derive our estimator $\hat{U}$, which represents the estimated total number of unique N-Quads that will be generated when mapping the entire logical source. To obtain $\hat{U}$, we scale U' by the inverse of the sampling probability, resulting in $\hat{U} = \frac{U'}{p}$. To calculate the expected value of $\hat{U}$, we consider the scaling of U', resulting in $E[\hat{U}] = E[\frac{U'}{p}]$. Substituting the expression for U', we further show that

$$E[\hat{U}] = E[\frac{U'}{p}] = E[\frac{U \cdot p}{p}] = E[U] \tag{2}$$

which demonstrates that $\hat{U}$ serves as an estimator for U.

To estimate the total number of N-Quads, $\hat{U}_{total}$, generated by all triplesMaps with non-join objectMaps, we sum the individual estimates $\hat{U}$:

$$\hat{U}_{total} = \sum \hat{U} \tag{3}$$

4.2 Estimation of triplesMaps with Join

When estimating the number of unique N-Quads J when mapping triplesMaps consisting of objectMaps with joins, it must be taken into account that there are two logical sources, A and B, instead of just one. Therefore, the estimation operation requires the creation of two samples using independent Bernoulli sampling: sample S_A from logical source A sampled with probability p_A, and sample S_B from logical source B sampled with probability p_B. The RML mapping is performed on the two samples, and the unique N-Quads J' generated are enumerated. To obtain the estimated number of unique N-Quads $\hat{J}$ when mapping the two entire logical sources, we again scale up J' by using the inverse of the two sampling probabilities p_A and p_B [26]. The inverse is used because the sampling probability represents the fraction of the logical source that is included in the sample. Thus, to estimate the number of unique RDF elements when mapping the entire content of the logical sources, we use $\hat{J} = \frac{J'}{p_A \cdot p_B}$.

To estimate the total number of unique N-Quads generated by triplesMaps with joins $\hat{J}_{total}$, we sum the scaled estimates of generated unique N-Quads:

$$\hat{J}_{total} = \sum \hat{J} \tag{4}$$

4.3 Estimation of All triplesMaps

The estimate of the total number of unique N-Quads generated from all triplesMaps, $\hat{T}$, is therefore given by

$$\hat{T} = \hat{U}_{total} + \hat{J}_{total} \tag{5}$$

By adding the unique estimates from predicateObjectMaps without joins to the unique estimates of predicateObjectMaps with joins, we arrive at an estimation of the total unique RDF elements generated through an RML mapping.

5 Empirical Evaluation

In the evaluation we aim to answer following three research questions:

- **R1:** How accurate is the estimation using the Result Size Estimator?
- **R2:** How does adaptive selection of the appropriate bit size for hash algorithms impact execution time and memory consumption?
- **R3:** How does FlexRML compare to state-of-the-art KG materializers in terms of execution time and memory consumption?

All results and scripts to run the benchmark can be found on GitHub[5].

Datasets. We evaluated FlexRML using three benchmarks. In a cloud and edge environment we used the GTFS Madrid benchmark [5] with scale factors of 10, 100, and 500, and the SDM Genomic Testbed[6] with dataset sizes of 10K, 100K, and 1M entries and a duplicate rate of 75 percent, with each element repeated 20 times. To the best of our knowledge, there are no datasets containing sensor data that are small enough to be mapped on IoT devices, so for this purpose we introduce a new RML-SENSOR benchmark[7] (SENS). The RML-SENSOR benchmark simulates data produced by two sensors and includes one metadata file. The data can be generated using small scale factors, which allows for the evaluation of the performance of RML processors on memory-constrained devices. All datasets used in the evaluation are in CSV format.

The SDM Genomic Testbed provides RML mappings[8] with different types of objectMaps: simple objectMaps without joins (POM), objectMaps with a self-join (REF), and objectMaps with a join (JOIN), the abbreviations are reused form [3]. The number of mappings of each type used is indicated by a number followed by the type, e.g. 1POM means that the mapping consists of one simple objectMap without join.

Metrics. For R1, we evaluate the Result Size Estimator by comparing the estimated number of generated N-Quads, obtained using various sampling probabilities, against the true value of generated N-Quads and the required execution time using the built-in chrono module. To address R2 and R3, we assess the KG materialization performance using two parameters: elapsed wall time and maximum memory usage. The latter was measured by the maximum resident set size. These parameters were evaluated on three systems with different levels of constraint. Both parameters, elapsed wall time and maximum resident set size, were obtained using the time command from the GNU time package[9]. Each

[5] https://github.com/wintechis/flex-rml-evaluation/.

[6] https://figshare.com/articles/dataset/SDM-Genomic-Datasets/14838342/1.

[7] https://github.com/wintechis/rml-sensor-benchmark/.

[8] https://github.com/SDM-TIB/SDM-RDFizer-Experiments/tree/master/cikm2020/experiments.

[9] https://www.gnu.org/software/time/.

experiment was conducted three times, and the average values for elapsed wall time and peak memory usage are reported. The exact hardware and software configurations of each tested device are detailed in Table 1. Better results are indicated by lower values in the diagram in both metrics.

Table 1. Evaluated Hardware and Systems.

Name	Type	Processor	Memory	Operating System
ESP32	IoT Device	2 × 240 MHz	512 KB	ESP-IDF FreeRTOS
Pi 4	Edge Node	4 × 1.5 GHz	4 GB	Raspberry Pi OS
VM	Cloud	8 × 2.0 GHz	64 GB	Ubuntu 22.04.3 LTS

5.1 Accuracy of Result Size Estimator

To evaluate R1, we performed experiments on the 100k SDM Genomic dataset using the combination of 4POM and 5JOIN mapping rules (Fig. 2), allowing us to evaluate the accuracy in a complex scenario combining objectMaps with and without joins. The evaluation is performed on the cloud based virtual machine.

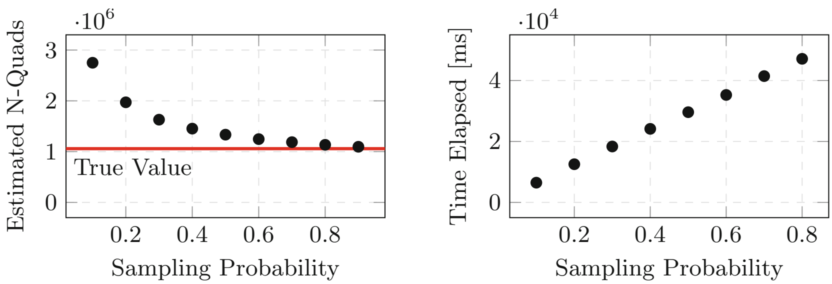

Fig. 2. Accuracy of estimated N-Quads compared to the true value, indicated by a red line, at 1,058,181 N-Quads (left) and elapsed time (right) of the Result Size Estimator on the 100k Genomic dataset using mapping rule 4POM combined with 5JOIN. (Color figure online)

Our results show that the estimated N-Quads follow an exponential decay as a function of the sampling probability and converge to the true value. The exponential decay is a result of diminishing returns in accuracy gains as the probability of sampling, and therefore the representativeness of the sample, increases; initially, small increases in probability yield significant improvements in estimation accuracy. However, as the probability and representativeness continue to increase, the incremental accuracy gains slow down. On the other hand, the execution time increases linearly with the sampling probability, which is to be expected since a doubling of the sampling probability results in twice as much data in the sample and thus requires roughly twice as much processing time.

In general, a higher sampling probability increases the accuracy of the estimated N-Quads, but at the cost of increased computational time. Conversely, lower sampling probabilities provide the benefit of faster estimation, but at the cost of reduced accuracy. Balancing these tradeoffs, FlexRML chooses a default sampling probability of 0.2.

5.2 Performance of Adaptive Hash Function Selection

To address R2, we evaluated FlexRML using an adaptive bit size selection for the hash algorithm and compared it to FlexRML$_{128}$, which represents a naive approach where a 128-bit hash function is used uniformly for all hash algorithms. The evaluation is performed on the cloud platform described in Table 1. For the SDM Genomic dataset, the RML mapping for 1POM is used. All mapped datasets produce the same output using both implementations, i.e. there are no hash collisions.

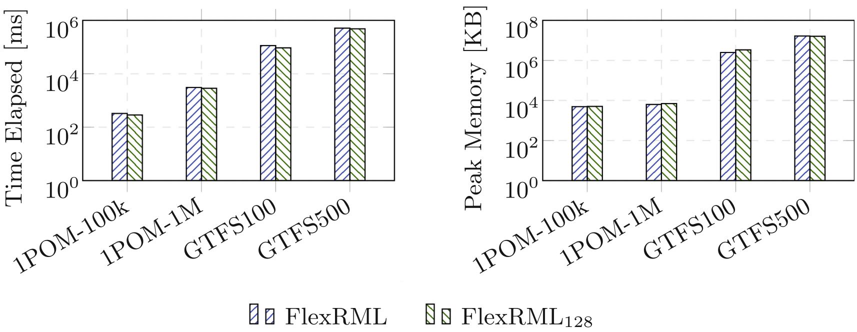

Fig. 3. Elapsed time (left) and peak memory usage (right) for FlexRML with Adaptive Hash Size (Blue) vs. FlexRML with Fixed 128-bit Hash Size (Red). (Color figure online)

The results show that the execution time of FlexRML$_{128}$ is consistently shorter compared to that of FlexRML (see Fig. 3, left diagram). This outcome is expected, as the algorithms used in the Result Size Estimator require additional processing time. However, as depicted in the right diagram of Fig. 3, FlexRML shows lower memory consumption when performing the mapping on all datasets, except in the case of the GTFS500 dataset, where both approaches require a 128-bit hash function. For instance, when mapping the GTFS100 dataset FlexRML requires about 1 GB less memory.

The evaluation shows that when a 128-bit hash function is unnecessary, the use of the Result Size Estimator can reduce the peak memory usage. However, this benefit is accompanied by an increase in processing time.

5.3 Comparison with Existing RML Processors

We perform a comparative analysis of FlexRML on three different platforms, as shown in Table 1: a virtual machine representing a cloud environment, a Raspberry Pi 4 representing an edge node, and an ESP32 microcontroller representing an embedded real-time IoT device. Note that the ESP32 has 512 KB of RAM, of which 300 KB are accessible to the heap. The comparison is made against other state-of-the-art RML processors, specifically Morph-KGC [4] and SDM-RDFizer [10]. Both RML processors are primarily implemented in Python, but use libraries written in C/C++ or Rust for performance-critical parts. When comparing FlexRML to the two RML processors, we took an outcome-based approach, focusing on comparing results using two metrics: materialization speed and memory usage, to evaluate KG materializers regardless of their implementation language. In the following, all figures have a logarithmic scale.

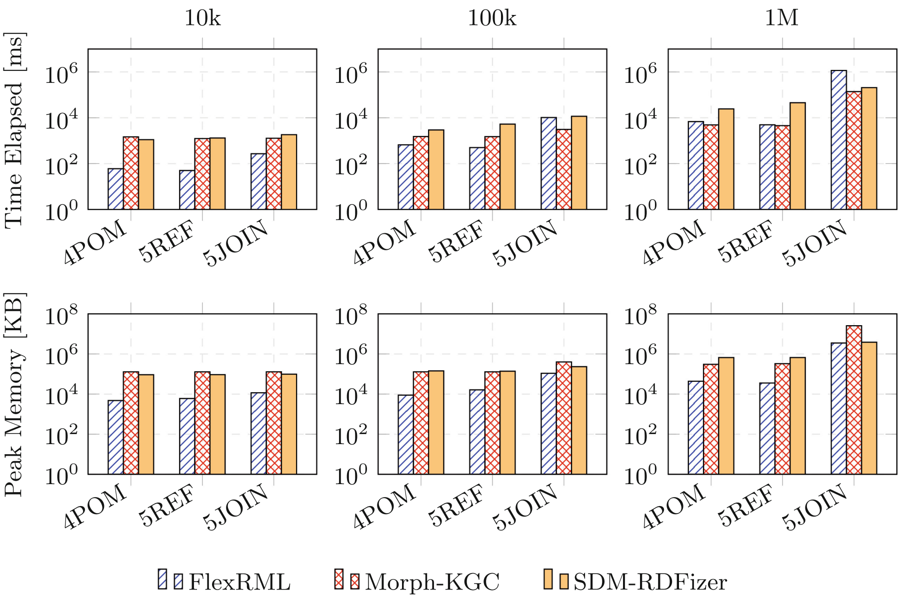

Fig. 4. Cloud: Elapsed time (top) and peak memory usage (bottom) for the SDM genomic dataset over data sizes of 10k, 100k, and 1M.

Cloud Environment. The data on elapsed time and peak memory usage for the SDM Genomic dataset (Fig. 4), shows FlexRML's better performance than Morph-KGC and SDM-RDFizer in processing predicateObjectMaps without join (POMs) or with self-join (REFs), and smaller datasets. However, for larger datasets, FlexRML is slower than Morph-KGC, and it is slower than both Morph-KGC and SDM-RDFizer when processing large datasets with JOIN mapping rules. Despite this, FlexRML's overall lower memory consumption across all datasets is a considerable compensating factor.

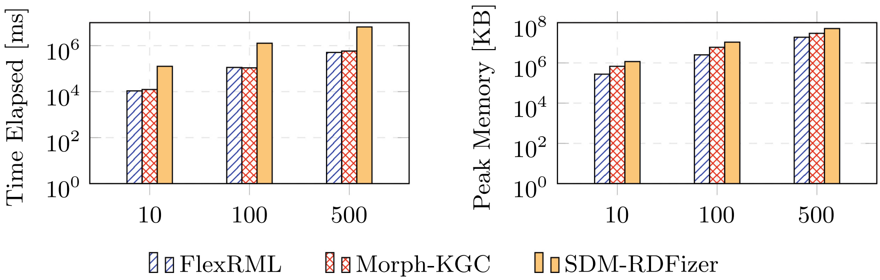

Fig. 5. Cloud: Elapsed time (left) and peak memory usage (right) for the GTFS Madrid benchmark using scale sizes of 10, 100, and 500.

In the GTFS Madrid benchmark (Fig. 5), FlexRML shows comparable performance to Morph-KGC in terms of processing time, while outperforming SDM-RDFizer. In addition, FlexRML demonstrates higher memory efficiency than the two other RML processors. The benchmarks show that FlexRML consistently outperforms other leading RML processors in memory efficiency across all datasets on cloud platforms. In situations where mappings consist entirely of POMs, REFs, or joins that are substitutable by reference conditions, FlexRML using the Result Size Estimator demonstrates processing speeds that are either superior to or on par with competing processors.

Edge Environment. The results from the SDM genomic dataset (Fig. 6) evaluation at the edge are consistent with those observed in the cloud. FlexRML shows greater memory efficiency and faster performance in mapping POMs or REFs. But for JOIN mappings, FlexRML's execution time exceeds that of the other two RML processors. Notably, Morph-KGC was not able to process the 5JOIN operation on the 1M dataset, due to insufficient memory of the edge.

In the case of the GTFS benchmark (Fig. 7), FlexRML again stands out, showcasing the shortest execution time and the least peak memory usage. This memory efficiency is significant in the context of the GTFS100 dataset, where FlexRML is the only RML processor evaluated that successfully maps the dataset within the 4 GB RAM limitation of the Pi.

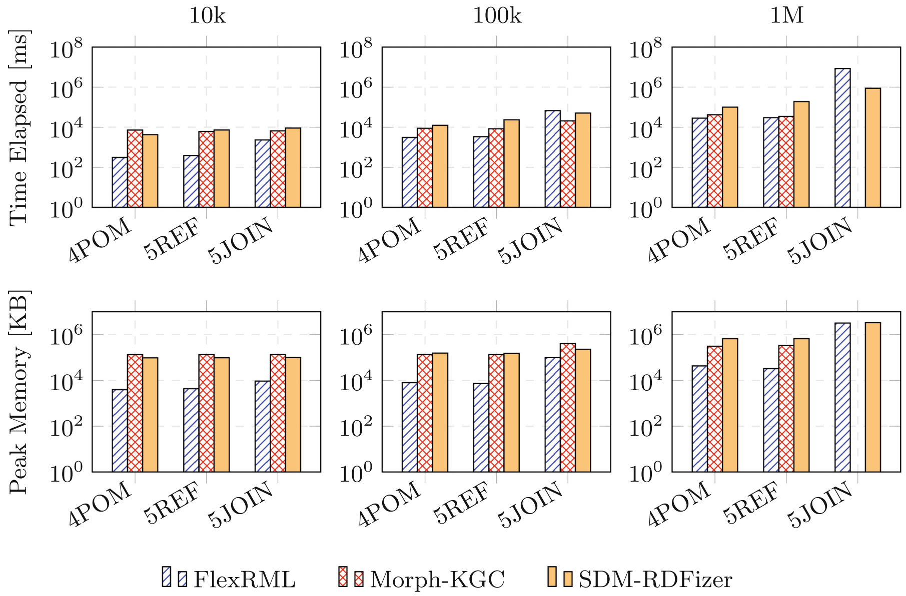

Fig. 6. Edge Node: Elapsed time (top) and peak memory usage (bottom) for the SDM genomic dataset. No bars indicate memory exhaustion and a process crash.

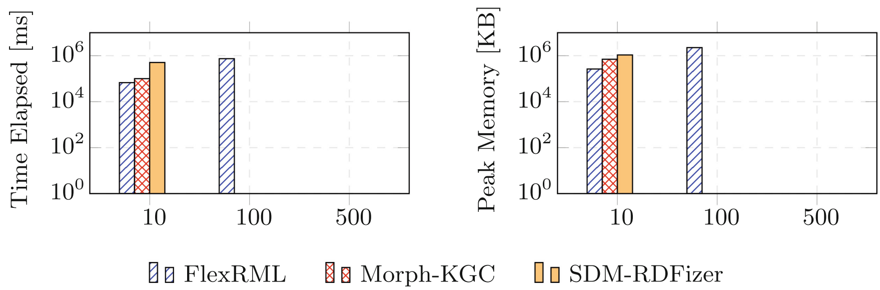

Fig. 7. Edge Node: Elapsed time (left) and peak memory usage (right) for the GTFS Madrid benchmark. No bars indicate memory exhaustion and a process crash.

Device Environment. FlexRML is the only RML processor capable of running on real-time microcontrollers due to its implementation in C++, so only FlexRML can be evaluated. In the device environment, the RML processor can only use in-memory data structures because the ESP32 microcontroller has no local storage, therefore both the RML mappings and the data to be mapped were stored in memory. Peak memory was measured by evaluating the heap memory used.

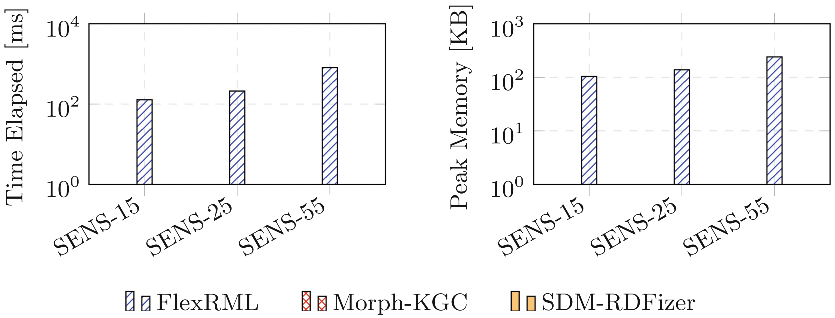

Fig. 8. IoT Device: Elapsed time (left) and peak memory usage (right) for the RML-SENSOR benchmark. Higher scales could not be mapped due to memory limitations.

The results in Fig. 8 show that FlexRML is capable of processing complex RML mappings, including joins, in a simple microcontroller environment. FlexRML can map up to 550 N-Quads, using the RML-SENSOR benchmark and a scale factor of 55, with only about 250 KB of memory. Such performance is considered sufficient and has the potential to enable semantic interoperability at the device level.

6 The Resource FlexRML

FlexRML is designed with several key aspects in mind to maximize usability, community adoption, and extensibility. These aspects include:

Availability. The FlexRML source code is actively maintained and publicly available in a GitHub[10] repository. FlexRML is released under the open source GNU AGPLv3 license. To ensure long-term accessibility and to support reproducibility, all releases of FlexRML are archived on Zenodo. Our future plans for FlexRML can also be found on the GitHub page. The roadmap outlines our upcoming features and reflects our commitment to continuous improvement and community engagement.

Novelty. FlexRML introduces a result size estimation technique for constructing KGs. The result size estimation feature is particularly important for devices with limited memory. Through this optimization and the implementation in C++, FlexRML expands the range of devices capable of mapping non-RDF data to RDF by focusing not only on unconstrained cloud environments, but also on resource-constrained devices and microcontrollers, a domain not previously explored by existing RML processors.

Reusability. FlexRML is designed with accessibility and ease of use in mind. Documentation, usage examples, and configuration files for various use cases are available in the GitHub repository, making it easy to use the resource. To further

[10] https://github.com/wintechis/flex-rml.

enhance user experience, prebuilt executables for multiple platforms are available. These executables align with the familiar process of installing or running programs for many users. By offering prebuilt versions, we eliminate the need to set up a build environment or install a specific language interpreter, tasks that require technical expertise. Each release of FlexRML is validated against the latest RML test cases to ensure full compliance with the RML specification.

Versatility. FlexRML uses a modular design that allows future enhancements, including the ability to handle additional data types such as JSON and XML. To handle each data encoding, a dedicated reader must be implemented. The reader is responsible for converting the read data to FlexRML's internal vector-based data representation. The FlexRML core can also be adapted to specific needs, as we have done with the CLI and ESP32 versions. The CLI version has an additional C++ file that handles command line arguments and calls the correct FlexRML functions based on the parameters, while the ESP32 version does not need CLI argument processing, but rather uses microcontroller specific libraries enabling, for example, serial communication.

7 Conclusion and Future Work

We introduced FlexRML, a flexible and memory-efficient resource for executing RML mappings to materialize KGs on a variety of devices, spanning different levels of constraints. FlexRML expands the range of devices capable of producing RDF, including those previously unable to run an RML processor. Our empirical evaluation shows that using the result size estimation algorithm reduces memory consumption compared to a naive fixed bit size approach, by allowing the use of smaller bit sizes in the hash function for duplicate removal without losing output quads. Additionally we found that FlexRML, using the adaptive bit size selection and memory advantages of C++, consistently shows lower peak memory consumption across different datasets when benchmarked against leading RML processors, and offers faster or comparable execution times. Looking ahead, our future development efforts will focus on implementing support for additional data formats, with a particular focus on JSON, incorporating compliance with the new RML specification, and reducing the execution time of non-substitutable joins by implementing join optimizations. In addition, an interesting future research direction we want to explore is the combination of mapping partitions and result size estimation to further improve memory efficiency.

Acknowledgements. 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. Ahamed, J., Mir, R.N., Chishti, M.A.: RML based ontology development approach in internet of things for healthcare domain. Int. J. Pervasive Comput. Commun. **17**(4), 377–389 (2021)
2. Al-Osta, M., Ahmed, B., Abdelouahed, G.: A lightweight semantic web-based approach for data annotation on IoT gateways. Procedia Comput. Sci. **113**, 186–193 (2017)
3. Arenas-Guerrero, J., Chaves-Fraga, D., Toledo, J., Pérez, M.S., Corcho, O.: Morph-KGC: scalable knowledge graph materialization with mapping partitions. Semant. Web (Preprint) 1–20 (2022)
4. Arenas-Guerrero, J.: morph-kgc/morph-kgc: 2.6.4 (2023). https://doi.org/10.5281/zenodo.10171377
5. Chaves-Fraga, D., Priyatna, F., Cimmino, A., Toledo, J., Ruckhaus, E., Corcho, O.: GTFS-Madrid-Bench: a benchmark for virtual knowledge graph access in the transport domain. J. Web Semant. **65**, 100596 (2020). https://doi.org/10.1016/j.websem.2020.100596
6. Daga, E., Asprino, L., Mulholland, P., Gangemi, A., et al.: Facade-X: an opinionated approach to SPARQL anything. Stud. Semant. Web **53**, 58–73 (2021)
7. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF Mapping Language (2012). https://www.w3.org/TR/r2rml/
8. Dasoulas, I., Chaves-Fraga, D., Garijo, D., Dimou, A.: Declarative RDF construction from in-memory data structures with RML (2023)
9. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: a generic language for integrated RDF mappings of heterogeneous data. In: 7th Workshop on Linked Data on the Web, vol. 1184 (2014)
10. eiglesias34, Chaves, D., et al.: SDM-TIB/SDM-RDFizer: v4.7.2.7 (2023). https://doi.org/10.5281/zenodo.10101405
11. Freedman, D., Pisani, R., Purves, R.: Statistics. 4th edn. W. W. Norton & Co (2007)
12. Ganzha, M., Paprzycki, M., Pawłowski, W., Szmeja, P., Wasielewska, K.: Towards semantic interoperability between Internet of Things platforms. In: Integration, Interconnection, and Interoperability of IoT Systems, pp. 103–127 (2018)
13. Hogan, A., et al.: Knowledge graphs. ACM Comput. Surv. **54**(4) (2021). https://doi.org/10.1145/3447772
14. Hozdić, E.: Smart factory for industry 4.0: a review. Int. J. Mod. Manuf. Technol. **7**(1), 28–35 (2015)
15. Iglesias, E., Jozashoori, S., Chaves-Fraga, D., Collarana, D., Vidal, M.E.: 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, pp. 3039–3046 (2020)
16. Ioannidis, Y.E.: Query optimization. ACM Comput. Surv. (CSUR) **28**(1), 121–123 (1996)
17. Jabbar, S., Ullah, F., Khalid, S., Khan, M., Han, K., et al.: Semantic interoperability in heterogeneous IoT infrastructure for healthcare. Wirel. Commun. Mob. Comput. **2017** (2017)
18. Lakka, E., et al.: End-to-end semantic interoperability mechanisms for IoT. In: 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), pp. 1–6. IEEE (2019)

19. Lefrançois, M., Zimmermann, A., Bakerally, N.: A SPARQL extension for generating RDF from heterogeneous formats. In: Blomqvist, E., Maynard, D., Gangemi, A., Hoekstra, R., Hitzler, P., Hartig, O. (eds.) ESWC 2017. LNCS, pp. 35–50. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58068-5_3
20. Martino, A., Iannelli, M., Truong, C.: Knowledge injection to counter large language model (LLM) hallucination. In: Pesquita, C., et al. (eds.) ESWC 2023. LNCS, vol. 13998, pp. 182–185. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-43458-7_34
21. Megill, N.D., Pavicic, M.: Estimating Bernoulli trial probability from a small sample. arXiv preprint arXiv:1105.1486 (2011)
22. Moons, B., Sanders, F., Paelman, T., Hoebeke, J.: Decentralized linked open data in constrained wireless sensor networks. In: 2020 7th International Conference on Internet of Things: Systems, Management and Security (IOTSMS), pp. 1–6. IEEE (2020)
23. Oo, S.M., Haesendonck, G., De Meester, B., Dimou, A.: RMLStreamer-SISO: an RDF stream generator from streaming heterogeneous data. In: Sattler, U., et al. (eds.) ISWC 2022. LNCS, vol. 13489, pp. 697–713. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19433-7_40
24. Rodriguez-Muro, M., Rezk, M.: Efficient SPARQL-to-SQL with R2RML mappings. J. Web Semant. **33**, 141–169 (2015)
25. Şimşek, U., Kärle, E., Fensel, D.: RocketRML - a NodeJS implementation of a use-case specific RML mapper. arXiv preprint arXiv:1903.04969 (2019)
26. Vengerov, D., Menck, A.C., Zait, M., Chakkappen, S.P.: Join size estimation subject to filter conditions. Proc. VLDB Endow. **8**(12), 1530–1541 (2015)
27. de Vleeschauwer, E., Min Oo, S., De Meester, B., Colpaert, P.: Reference conditions: relating mapping rules without joining. In: KGCW 2023, the 4th International Workshop on Knowledge Graph Construction (2023)
28. Vu, B., Pujara, J., Knoblock, C.A.: D-REPR: a language for describing and mapping diversely-structured data sources to RDF. In: Proceedings of the 10th International Conference on Knowledge Capture, pp. 189–196 (2019)
29. Wang, L.: Heterogeneous data and big data analytics. Autom. Control Inf. Sci. **3**(1), 8–15 (2017)