

A Functional and Distributed RML Processor

Corentin Michaux^{1,*}, Christophe Debruyne^{1,*}

¹Montefiore Institute, University of Liège, 4000 Liège, Belgium

Abstract

Distributed RML processing remains comparatively underexplored, especially for batch-oriented knowledge graph construction over large, bounded datasets. This paper presents FPMapper, a functional RML processor implemented in Scala on Apache Spark. By modeling RML constructs as composable functions, FPMapper can execute the same mapping pipeline locally or over Spark partitions. It passes 154 conformance tests across RML-Core, FNML, Logical Views, and RML-star. Preliminary KROWN benchmark results show that, despite Spark's startup overhead, FPMapper becomes the fastest evaluated engine at one million rows, highlighting the potential of functional Spark-based RML processing when data becomes too large to process (efficiently) on a single machine.

Keywords

Knowledge Graph Construction, RDF Mapping Language, RML

1. Introduction

RML [1] is a declarative mapping language for describing how to transform non-RDF resources into RDF. Reference implementations of older and newer RML versions, such as RMLMapper¹ and BURP [2], both generate RDF sequentially, and neither parallelizes the process. Many existing RML engines often focus on specific aspects of RML processing: algebraic execution [3, 4], specification-oriented conformance [2], optimization [5], or distributed and streaming execution [6].

Distributed approaches for RML generation remain scarce. RMLStreamer [6] executes RML mappings on Apache Flink using a streaming execution model, making it well suited to scenarios where data arrives continuously and RDF must be generated incrementally. Closer to our setting, [7] targets bounded, batch-oriented sources on Apache Spark by translating RML mappings into a workload of extended SPARQL CONSTRUCT queries and evaluating them through a generic SPARQL-to-RDD algebra layer. However, many knowledge graph construction tasks are batch-oriented: large but bounded data sources are transformed into RDF as part of an offline data integration pipeline. This raises the question of whether RML constructs can instead be executed directly against Spark's distributed data model, without an intermediate SPARQL rewriting and query-algebra evaluation step.

The question we address in this paper is thus: *To what extent can a distributed RML processor, implemented for bounded batches, achieve conformance with the RML specification while enabling scalable execution on large, bounded datasets?*

2. Related Work

As stated in the introduction, [6] supports distributed RML processing on Apache Flink, but its streaming model targets continuously arriving data rather than large bounded batches. The closest work to ours is [7], which also targets batch-oriented RML processing on Apache Spark, but takes a rewriting approach: RML mappings are translated into extended SPARQL CONSTRUCT queries (introducing SPARQL extensions for CSV/JSON/XML access) and executed by mapping SPARQL algebra operators, including joins, onto RDD operations via the SANSA framework, with additional query-level optimizations such

Posters, Demos, Blue Sky, and Tutorials at SEMANTICS 2026, Sep 2026, Ghent, Belgium

*Corresponding authors

✉ corentin.michaux@student.uliege.be (C. Michaux); c.debruyne@uliege.be (C. Debruyne)

id 0000-0003-4734-3847 (C. Debruyne)



© 2026 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>, last accessed July 25, 2026

as CONSTRUCT-query merging and self-join elimination. FPMapper instead models RML constructs directly as composable Spark-native functions, without an intermediate SPARQL rewriting and query-algebra layer, and reports conformance against the RML-Core, FNML, Logical Views, and RML-star test suites, which [7] does not evaluate. Their evaluation also uses GTFS-Madrid-Bench and SDM-Genomic-Datasets rather than KROWN, so direct performance numbers are not comparable in the evaluation section. Morph-KGC [8] and SDM-RDFizer [9] support parallel RDF generation on a single machine through multiprocessing in Morph-KGC and multi-thread-safe execution in SDM-RDFizer. Therefore, scaling large materialization jobs with these processors primarily requires sufficient CPU, memory, and I/O resources on one host. We therefore argue that the gap is worth exploring, and we developed an RML engine built on top of a distributed computing framework.

3. Design and Implementation

We present FPMapper², an RML processor implemented in Scala³ on top of Apache Spark⁴. FPMapper’s design models RML constructs as composable functions, allowing the same mapping pipeline to run locally or over distributed Spark partitions. In this work, Spark is useful because independent RML logical iterations can be distributed across partitions while reusing the same functional mapping pipeline.

A pure functional implementation is a natural paradigm to adopt for RML because RML itself is declarative. By representing RML constructs as pure, composable functions, the implementation remains close to the specification and avoids shared mutable state. This is particularly relevant for distributed execution, since independent logical iterations can be evaluated on different workers without changing the mapping semantics.

FPMapper is built around a functional mapping pipeline. RML constructs are modeled as composable pure functions, so logical sources, term maps, expression maps, and triple maps can be evaluated independently and distributed over Spark RDD partitions. The implementation separates pure RDF-generation logic from effectful operations such as file I/O, using Cats Effect IO, and relies on Apache Jena for RDF parsing and serialization.

For a TriplesMap without joins or RML-star, the logical iterations of the source are parallelized into an RDD⁵, which can be seen as some sort of “row”, and the term-map evaluation for that iteration or “row” runs independently inside a single `mapPartitionsWithIndex` pass. Blank-node identifiers are minted locally within each partition and disambiguated with a partition-scoped prefix, avoiding any cross-partition coordination; the resulting quads are grouped by named graph within the partition and written directly, without a Spark-level shuffle.

Join resolution is currently performed once per TriplesMap before the distributed term-map evaluation step, rather than as a distributed Spark-level operation: parent and child iterations are matched once, and the resulting “row” combinations are partitioned across the cluster for RDF term generation. This keeps the core per-row pipeline shuffle-free, but does bound join cost to what fits in driver memory. Expressing join resolution itself as a distributed Spark operation (e.g., a partitioned RDD join keyed on the join attributes) is a natural next step, left to future work.

Duplicate RDF output is not eliminated globally in the default file-output mode; only whole-graph serializations that are byte-identical across TriplesMaps are dropped. A canonical, fully deduplicated and sorted output is available via a `--stdout` mode, which merges and deduplicates at the level of individual N-Quads lines, but this at the cost of collecting all output back to the driver.

²The source code of FPMapper is available at <https://gitlab.uliege.be/Corentin.Michaux/tfe>.

³In this work, Scala is useful because it makes it natural to model RML constructs as composable functions while still integrating with Apache Spark.

⁴Apache Spark is a distributed data-processing framework designed to run large-scale computations across a cluster of machines. It provides abstractions such as RDDs (Resilient Distributed Datasets) and DataFrames, allowing data to be partitioned and processed in parallel.

⁵Partitioned at four times the cluster’s default parallelism (floor of 20, absorbing skew across differently sized sources).

Properties	FPMapper	RMLMapper	RMLStreamer	SDM-RDFizer
1	27.9	5.0	22.0	4.8
10	38.1	13.2	37.4	16.0
20	46.4	22.1	51.9	28.5
30	54.2	31.1	68.8	40.0

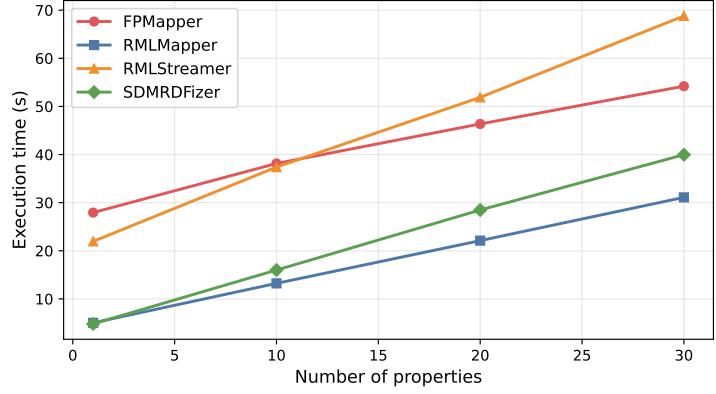


Figure 1: Median execution time (s) by engine for varying number of properties (100K rows, no value padding) based on three runs.

4. Evaluation

FPMapper was evaluated for both correctness and scalability.⁶ Conformance was tested on 154 RML test cases across RML-Core, FNML, Logical Views, and RML-star, all of which were passed. FPMapper does not support RML-CC and only partly supports RML-IO. This will be discussed in the limitations.

Scalability was measured with KROWN-benchmark [10] generated CSV workloads and compared against RMLMapper, SDM-RDFizer, and RMLStreamer. First, we analyze the impact of increasing the number of properties. All four engines were evaluated in the same Azure environment⁷, using the KROWN execution framework and the same generated benchmark scenarios, so that the reported differences reflect the processors rather than changes in the experimental setup. Since FPMapper is distributed, it was submitted as a Spark job to a standalone Spark cluster composed of one coordinator node (the same as the others) and two worker nodes⁸, with the master coordinating execution and the workers processing the mapping workload over Spark partitions.

In Figure 1, we fix the dataset at 100K rows with no value padding and vary the number of mapped properties from 1 to 30, measuring how each engine’s execution time scales with mapping complexity. All four engines exhibit roughly linear growth with respect to the number of properties. RMLStreamer is markedly slower than RMLMapper throughout, though the gap becomes smaller as properties increase: from 4.4× at 1 property down to roughly 2-2.8× from 10 properties onward. FPMapper starts with the largest relative overhead at 1 property, but its gap over RMLMapper stays roughly constant in absolute terms (around 23s) rather than growing with the number of properties, so its relative slowdown shrinks to about 1.7× at 30 properties.

We aim to assess this in Figure 2. In this experiment, we set the mapping to 20 properties with no value padding and vary the number of input rows from 1000 to 1M, testing how engines scale with data volume. We notice the high fixed startup cost of FPMapper and RMLStreamer, which take around 20 seconds even for just 1000 rows, while RMLMapper and SDM-RDFizer start in roughly 2 seconds. As a result, at small to moderate sizes (up to ±10K rows), RMLMapper and SDM-RDFizer are preferable. However, at 1M rows the ranking reverses: FPMapper becomes the fastest of the four evaluated engines, suggesting it has better per-row throughput that only becomes apparent at large scale; this ranking is with respect to RMLMapper, SDM-RDFizer, and RMLStreamer only, and may not hold against engines not included in this comparison.

FPMapper’s advantage at 1M rows, in the context of KROWN with default values, can be explained. Spark’s maxPartitionBytes is 128 MB by default. Once data exceeds that threshold, the data is partitioned.

⁶The data is available at https://gitlab.uliege.be/Coentn.Michaux/tfe_experiments

⁷The coordinator node was an Azure Standard_E2_v3 virtual machine with 2 vCPUs and 16 GB of RAM. It hosted the Spark master process and also ran the KROWN execution framework.

⁸The distributed part of FPMapper used two worker nodes, each an Azure Standard_B2s_v2 virtual machine with 2 vCPUs and 4 GB of RAM. Together, the workers provided 4 vCPUs and 8 GB of worker memory for processing the Spark partitions.

Rows	FPMapper	RMLMapper	RMLStreamer	SDM-RDFizer
1K	25.4	3.1	18.1	2.5
5K	27.0	4.1	20.7	3.5
10K	28.6	5.1	23.6	4.8
100K	46.4	22.1	51.9	28.5
1M	205.9	242.2	294.2	266.0

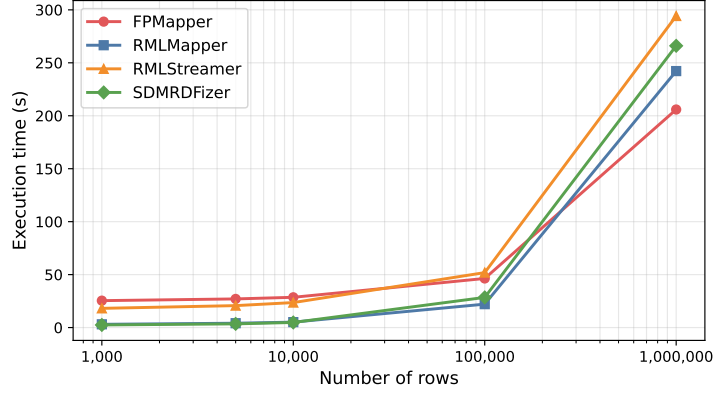


Figure 2: Median execution time (s) per engine as a function of the number of input rows (20 properties, no value padding) based on three runs.

Rows	FPMapper	RMLMapper	RMLStreamer	SDM-RDFizer
1K	1604	1192	2028	1051
5K	1675	1425	2028	1077
10K	1693	1717	2095	1111
100K	2749	4399	6641	1870
1M	10200	9561	6670	8923

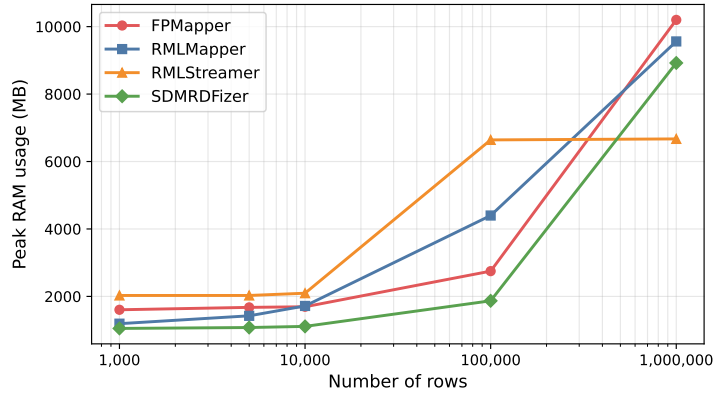


Figure 3: Peak RAM per engine as a function of the number of input rows (20 properties, no value padding) based on three runs.

Each worker has two cores and can thus hold two partitions. At 1M rows, however, the data exceeds the threshold and is partitioned across workers or cores, thereby parallelizing processing.

We also investigate (peak) RAM usage, shown in Figure 3. All engines have a fairly flat RAM baseline about 1-2 GB up to about 10k rows, but then diverge. FPMapper uses the most RAM at scale (10.2 GB at 1M rows). SDM-RDFizer is the most memory-efficient up to 100K rows, but RMLStreamer overtakes it at 1M rows (6.7 GB vs. 8.9 GB), since RMLStreamer's RAM usage stays nearly flat from 100K rows onward while SDM-RDFizer's continues to grow with row count. RMLStreamer plateaus around 6 GB, suggesting that it may load data in chunks rather than all at once; this needs further investigation. The figure does not represent the total cluster memory consumed by FPMapper: the executor JVMs on the worker nodes perform the actual partition processing, and their RAM is not captured by KROWN's monitor (which runs on the master). While likely smaller, one must thus take into account the RAM used by the worker nodes.

The experiments varied both the number of rows and the number of properties. While Spark introduces overhead for small datasets, FPMapper scales well and becomes the fastest engine at 1M rows. However, the experiments are still preliminary. We can tentatively conclude that the benchmark reveals a speed-vs-memory trade-off and a volume-dependent crossover in engine rankings. The practical recommendation would be: use RMLMapper or SDM-RDFizer for typical workloads, FPMapper for very large batch jobs where speed matters or data does not fit into the main memory of a single node, and RMLStreamer when data can be streamed or processed in chunks.

4.1. Limitations

RML-IO was only partially implemented because the required conformance tests mainly needed CSV and JSON input handling, while full RML-IO target/source descriptor support was outside the scope of this study: RML processing and distributed execution.

RML-CC was not implemented in the current prototype because full support for RDF collections and containers raises additional challenges in a distributed setting. In particular, collections whose elements are *spread across multiple logical iterations* require grouping, ordering, and blank-node coordination across Spark partitions, which introduces wide transformations and shuffle operations. This challenge is not specific to FPMapper, as [11] also noted that support for collections and containers across iterations would require significant changes for Morph-KGC.

5. Conclusions

The main lesson learned is that a Spark-based distributed architecture is viable for large bounded RML workloads, but not universally preferable. For small and medium datasets, the overhead of Spark makes single-machine engines such as RMLMapper or SDM-RDFizer more practical. However, as data volume increases, FPMapper’s distributed execution becomes advantageous, suggesting that the research question can be answered positively but conditionally: a distributed RML processor for bounded batches can combine conformance with scalable execution, provided that the workload is large enough to justify the overhead of using Spark. The functional design also proved useful because it keeps the implementation close to the declarative nature of RML and makes distribution over independent logical iterations natural.

Future work should look into how the current join resolution strategy could be improved by expressing it as a distributed Spark operation, which would allow for better scalability and performance on larger datasets. We also need to expand the set of supported RML profiles and investigate the behavior of different engines in greater depth across different scenarios. Further experiments should evaluate larger clusters, additional datasets, different Spark configurations, memory usage across both coordinator and worker nodes, and comparisons with streaming approaches such as RMLStreamer in scenarios where batch and stream processing overlap. This last point is currently constrained by KROWN itself, which gathers information only from the node on which the engine runs and does not capture metrics from worker nodes in a distributed setting; extending KROWN with such support would be valuable to the community. The evaluation should also be extended to single-machine parallel engines such as Morph-KGC, which were not part of the present comparison, to determine how FPMapper’s distributed execution model fares against optimized single-node parallelism.

Declaration on Generative AI

Most of the code was written manually, except for the triples map parsing, which was developed using Claude. We note that parsing RML (triple maps) was not central to this thesis, as this step merely allows us to instantiate our Algebraic Data Types (ADTs). Some debugging and documentation were also done with Claude. The MSc thesis from which this paper is distilled was written from manually drafted text, but Claude, GPT, and DeepL were used for LaTeX layouting, to improve sentence phrasing, and to translate from French to English. Claude was also used to generate Python scripts for generating LaTeX tables and plots for the images. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

- [1] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, B. De Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga, 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 - 22nd International Semantic Web Conference*, Athens, Greece, November 6-10, 2023, *Proceedings, Part II, Lecture Notes in Computer Science*, Springer, 2023, pp. 152–175. doi:10.1007/978-3-031-47243-5_9.
- [2] D. Van Assche, C. Debruyne, Burping through RML test cases, in: *Proceedings of the 5th International Workshop on Knowledge Graph Construction co-located with 21th Extended Semantic Web Conference (ESWC 2024)*, Hersionissos, Greece, May 27, 2024, *CEUR Workshop Proceedings*, CEUR-WS.org, 2024.
 - [3] S. Min Oo, B. De Meester, R. Taelman, P. Colpaert, Algebraic mapping operators for knowledge graph generation, *Semantic Web 16* (2025). doi:10.1177/22104968251361350.
 - [4] S. Min Oo, T. Verbeken, B. De Meester, Rmlweaver-js: An algebraic mapping engine in the KGCW challenge 2024, in: *Proceedings of the 5th International Workshop on Knowledge Graph Construction co-located with 21th Extended Semantic Web Conference (ESWC 2024)*, Hersionissos, Greece, May 27, 2024, *CEUR Workshop Proceedings*, CEUR-WS.org, 2024.
 - [5] M. Freund, S. Schmid, A. Harth, Efficient knowledge graph construction based on optimized plans, in: *Linking Meaning: Semantic Technologies Shaping the Future of AI - Proceedings of the 21st International Conference on Semantic Systems, SEMANTiCS 2025*, Vienna, Austria, September 3-5, 2025, *Studies on the Semantic Web*, IOS Press, 2025, pp. 2–18. doi:10.3233/SSW250005.
 - [6] S. Min Oo, G. Haesendonck, B. De Meester, A. Dimou, Rmlstreamer-siso: An RDF stream generator from streaming heterogeneous data, in: *The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference*, Virtual Event, October 23-27, 2022, *Proceedings, Lecture Notes in Computer Science*, Springer, 2022, pp. 697–713. doi:10.1007/978-3-031-19433-7_40.
 - [7] C. Stadler, L. Bühmann, L. Meyer, 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*, Hersionissos, Greece, May 28, 2023, *CEUR Workshop Proceedings*, CEUR-WS.org, 2023.
 - [8] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M. S. Pérez, Ó. Corcho, Morph-kgc: Scalable knowledge graph materialization with mapping partitions, *Semantic Web 15* (2024) 1–20. URL: <https://doi.org/10.3233/SW-223135>. doi:10.3233/SW-223135.
 - [9] E. A. Iglesias, M. Vidal, Results for knowledge graph creation challenge 2025: Sdm-rdfizer, in: *Proceedings of the 6th International Workshop on Knowledge Graph Construction co-located with 22nd Extended Semantic Web Conference (ESWC 2025)*, Portorož, Slovenia, June 1, 2025, *CEUR Workshop Proceedings*, CEUR-WS.org, 2025. URL: <https://ceur-ws.org/Vol-3999/short3.pdf>.
 - [10] D. Van Assche, D. Chaves-Fraga, A. Dimou, KROWN: A benchmark for RDF graph materialisation, in: *The Semantic Web - ISWC 2024 - 23rd International Semantic Web Conference*, Baltimore, MD, USA, November 11-15, 2024, *Proceedings, Part III, Lecture Notes in Computer Science*, Springer, 2024, pp. 20–39. doi:10.1007/978-3-031-77847-6_2.
 - [11] C. Debruyne, S. Jaadari, D. Chaves-Fraga, Declarative generation of RDF collections and containers from heterogeneous data, in: *Linking Meaning: Semantic Technologies Shaping the Future of AI - Proceedings of the 21st International Conference on Semantic Systems, SEMANTiCS 2025*, Vienna, Austria, September 3-5, 2025, *Studies on the Semantic Web*, IOS Press, 2025, pp. 28–43. doi:10.3233/SSW250007.