Parallel RDF Generation from Heterogeneous Big Data

Gerald Haesendonck
gerald.haesendonck@ugent.be
IDLab, Dep. of Electronics and Information Systems, Ghent University – imec
Zwijnaarde, Belgium

Wouter Maroy
wouter.maroy@ugent.be
IDLab, Dep. of Electronics and Information Systems, Ghent University – imec
Zwijnaarde, Belgium

Pieter Heyvaert
pheyvaer.heyvaert@ugent.be
IDLab, Dep. of Electronics and Information Systems, Ghent University – imec
Zwijnaarde, Belgium

Ruben Verborgh
ruben.verborgh@ugent.be
IDLab, Dep. of Electronics and Information Systems, Ghent University – imec
Zwijnaarde, Belgium

Anastasia Dimou
anastasia.dimou@ugent.be
IDLab, Dep. of Electronics and Information Systems, Ghent University – imec
Zwijnaarde, Belgium

ABSTRACT

To unlock the value of increasingly available data in high volumes, we need flexible ways to integrate data across different sources. While semantic integration can be provided through RDF generation, current generators insufficiently scale in terms of volume. Generators are limited by memory constraints. Therefore, we developed the RMLStreamer, a generator that parallelizes the ingestion and mapping tasks of RDF generation across multiple instances. In this paper, we analyze what aspects are parallelizable and we introduce an approach for parallel RDF generation. We describe how we implemented our proposed approach, in the frame of the RMLStreamer, and how the resulting scaling behavior compares to other RDF generators. The RMLStreamer ingests data at 50% faster rate than existing generators through parallel ingestion.

CCS CONCEPTS

• **Information systems → Semantic web description languages**.

KEYWORDS

RDF generation, big data, linked data, semantic web

ACM Reference Format:
Gerald Haesendonck, Wouter Maroy, Pieter Heyvaert, Ruben Verborgh, and Anastasia Dimou. 2019. Parallel RDF Generation from Heterogeneous Big Data. In *The International Workshop on Semantic Big Data (SBD'19), July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3323878.3325802

1 INTRODUCTION

The massively increasing volume of data has become a global phenomenon. However, flexible ways to integrate data across different sources are needed to unlock the value of such high volumes of data. While semantic integration can be achieved through RDF generation, current RDF generators insufficiently scale in terms of volume due to *memory constraints*. However, since data can far exceed the amount of available memory, RDF generators are not able to process data in high volumes so far.

Traditional RDF generators do not consider parallelizations, limiting the *volume* of data sources that can be supported and eventually even reflecting on the overall performance with respect to *speed*. Existing RDF generators, e.g. the RMLMapper[1] or SPARQL-Generate[2] *sequentially ingest* multiple data sources, even though not only the ingestion, but also the processing, can be parallelized. They load all data *in memory* during ingestion, before the RDF generation starts. However, this severely *limits the amount of data* that can be ingested, since the size of the data must be smaller than the available memory. Other solutions, e.g., CARML[3], *ingest through streaming*. Instead of loading all data in memory, CARML iterate through the data, so only smaller parts that fit in memory are processed at any time. Even though this solves the data volume problem, it *does not improve the speed*, because the data is still sequentially processed. Last, other solutions, e.g. RocketRML[4], *limit the support* of data sources or rules range to optimize their speed.

Therefore, we propose an approach that parallelizes and distributes the *ingestion* tasks of the RDF generation process over multiple nodes to scale with data volume. Scaling out the RDF generation process allows generating RDF in far higher volumes than was feasible before. In this paper, we (i) demonstrate how we implemented such a methodology, and (ii) compare the resulting scaling behavior of such an implementation to other RDF generators. To validate our methodology, we developed and evaluated the RMLStreamer[5], a generator that parallelizes the ingestion task of RDF generation across multiple instances. The RMLStreamer ingests unlimited volumes of data at a faster rate than existing RDF generators through parallel ingestion.

[1]RMLMapper, http://github.com/RMLio/RML-Mapper
[2]SPARQ:L-Generate, https://ci.mines-stetienne.fr/sparql-generate/
[3]CARML, https://github.com/carml/carml
[4]RocketRML, https://github.com/semantifyit/RML-mapper
[5]RMLStreamer, https://github.com/RMLio/RMLStreamer

The remainder of the paper is structured as follows: In Section 2, we outline the current state of the art with respect to Linked Data generators and distributed processing frameworks. In Section 3, we introduce our proposed approach. In Section 4, we explain the implementation. In Section 5 we describe the evaluation we performed and its results. Last, in Section 6, we outline our conclusions.

2 RELATED WORK

In this section, we discuss the state of the art regarding RDF generation from multiple heterogeneous data sources (Section 2.1) and distributed processing (Section 2.2).

2.1 RDF generation tools

Several RDF generators were implemented so far, but only few can generate RDF from multiple data formats, employing though separate source-centric approaches for each format. We outline the most well-known RDF generators for data in different formats.

The *RMLMapper* [4] is an RML engine: it generates RDF from (semi-)structured data sources in various structures, formats, and serializations using rules in RML [4], but it is not optimized for high volumes. The RMLMapper is Java-based built on top of RDF4J[6].

XSPARQL[1] performs dynamic query translation to generate RDF from XML, combining XQuery[7] and SPARQL. This way, it allows querying data in XML and RDF, using the same framework, and transform data from one format to the other, i.e. generating RDF from XML and vice versa. Besides data in XML, XSPARQL was also used for data in relational databases. However, it was never extended to be used beyond data in databases or XML format.

xR2RML[8] [8] extends R2RML and RML to generate RDF from data in NoSQL, clarifying RML's extension over access interfaces for NoSQL databases. Its processor extends Morph with a MongoDB implementation that relies on the MongoDB API and the Jongo API for the management of MongoDB shell queries. xR2RML follows the object factory design pattern to deal with heterogeneity.

CARML[9] is an RML processor that generates RDF using RML rules. It is implemented along the lines of the RMLMapper with respect to schema and data transformations. It differs because it turns a data source into a stream to generate the corresponding RDF overcoming out of memory issues.

SPARQL-Generate [6] is an RDF generator that considers the SPARQL-Generate language for specifying rules to generate RDF from various data structures and formats. It is a Java-based implementation on top of Apache Jena [10]. However, SPARQL-Generate is not optimized for high volumes.

2.2 Distributed processing frameworks

We present the most well-known distributed processing frameworks and discuss their scaling behavior.

[6]RDF4J, http://rdf4j.org

[7]XQuery, https://www.w3.org/TR/xquery/all/

[8]xR2RML, https://github.com/frmichel/morph-xr2rml

[9]CARML, https://github.com/carml/carml

[10]Apach Jena, https://jena.apache.org

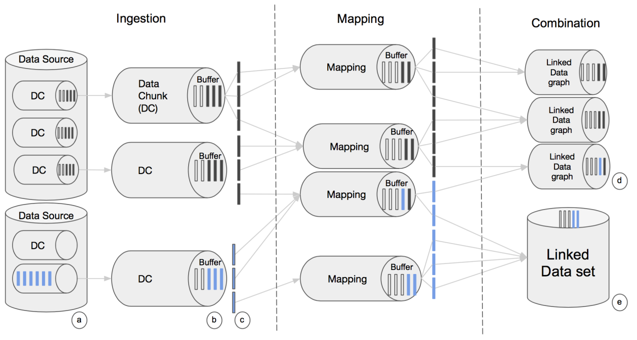

Figure 1: Parallel RDF generation. In parallel, (i) multiple Data Sources [DS] can be ingested, (ii) its data can be split in Data Chunks [DC], (iii) their Data Records [DR] can be processed, and (iv) their mapping can be performed to generate RDF in separate graphs or a complete RDF set.

Akka[11] is an actor-based concurrency library, written in Scala, which is designed for low-level streaming applications implemented conforming to the Reactive Streams standard[12].

Apache Spark Streaming[13] is a distributed data streaming processing framework, written in Scala. The execution of algorithms is split in different partitions run in parallel over different configured machines. The Spark Streaming API is an extension of the Apache Spark framework[14], a batched processing framework. Thus, Spark Streaming uses micro-batches to process streams [9]. Apache Spark Streaming is a mature library for Big Data processing.

Apache Storm[15] is a distributed real-time data processing framework, written in Java. Apache Storm processes data streams, doing for realtime processing what Hadoop did for batch processing.

Apache Flink [2] is an open-source distributed processing framework for streaming and batch data, written in Java. Flink is a more recent frameworks and the only framework that supports event time and out-of-order processing, provides consistent managed state with exactly-once guarantees, and achieves high throughput and low latency, serving both big static and streaming data.

3 PARALLELIZABLE RDF GENERATION

We propose an approach to incorporate distribution and parallelization in the RDF generation process. Our proposed approach is driven by observations of workloads from our existing RDF generator, i.e., the RMLMapper, and our real-case experiences, e.g., with the DBpedia Extraction Framework [7], which led us to reexamine traditional choices and explore a radically different design.

Our proposed approach considers three main tasks: (i) *ingestion*, (ii) *mapping*, and (iii) *combination*. These tasks are aligned in the aforementioned order, following the producer–consumer paradigm [5]. The producer and consumer are two concurrent processes which use a common buffer as a queue. The producer generates data into the buffer and the consumer takes data out of the buffer.

[11]Akka, https://akka.io/

[12]Reactive Streams, https://goo.gl/FoNs7h

[13]Spark streaming, https://spark.apache.org/streaming/

[14]Apache Spark, https://spark.apache.org

[15]Apache Storm, http://storm.apache.org

Each instance of a task produces data in buffers in memory for consumption for the next task, while multiple instances of each task can exist in parallel (Figure 1). The *ingestion* consumes data from data sources and produces data records. The *mapping* then consumes these data records and generates RDF according to the specified rules. The *combination* consumes the results from all mapping tasks and reduces to a single RDF dataset. In details:

Ingestion. The ingestion task parallelizes on (i) data source (DS, Figure 1 ⓐ), (ii) data chunk (DC, Figure 1 ⓑ), and (iii) data record level (DR, Figure 1 ⓒ). Multiple data sources are ingested *in parallel*. Then data from data sources are retrieved and split in smaller data chunks of predefined size. The ingestion task fetches and deserializes data records from each chunk in memory (consumer, Figure 1 ⓑ). A data record can be, for instance, a CSV row, a JSON object, or an XML element. Several data records are ingested in a buffer (the ingestion task becomes a producer, Figure 1 ⓒ) to be consumed in the next task, i.e. mapping. As a result, data records are available in buffers of parallel instance.

Mapping. The mapping task reads data records from the ingestion task buffers, and generates RDF in its own buffer from these independent records, according to the specified rules. By default mapping rules allow the records to be processed in parallel, however in some specific situations the parallelism might be reduced, for instance, when relations among different data sources are defined *and* the order of the records to merge differs significantly.

Combination. The combination task reads all RDF from all buffers of the mapping task and reduces this to separate graphs (RDF graph, Figure 1 ⓓ) or an RDF set (RDF set, Figure 1 ⓔ). The RDF that resides in the buffers from all instances of the mapping task is merged (a union on Flink DataStreams) into a single RDF set by concurrently emptying the buffers and writing into a final source.

4 IMPLEMENTATION

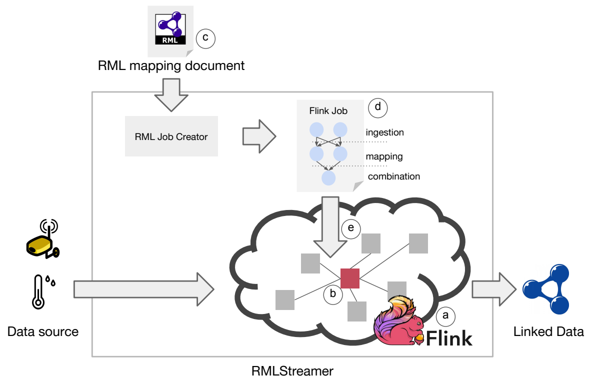

Figure 2: RMLStreamer's execution workflow

We propose a Scala implementation of the aforementioned proposed approach for parallel RDF generation. It is built on top of the distributed processing framework Apache Flink, for handling the parallel execution of each task of the RDF generation process over multiple (distributed) instances. RML [4] is used to define the rules to generate RDF, but other languages can be used as well. We chose RML because it extends the W3C recommended R2RML [3], and supports data from multiple heterogeneous sources.

We introduce our implementation of the proposed approach (Section 4.1) and our workflow for RDF generation (Section 4.2).

4.1 Tasks in a Flink pipeline

The tasks of our proposed approach are implemented as part of a Flink pipeline. Rules in RML determine how the pipeline is configured for the RDF generation process. Flink defines its execution process as *jobs* that exist out of *pipelines*. A job is executed by a running *instance*, a task manager that orchestrates tasks on a local node, or *a cluster of multiple nodes*. Flink pipelines are defined by several operators that handle input, transformations and output. A pipeline consists of consumers and producers: one operator is the producer for the next operator that consumes as input the output of the previous operator. If an operator is parallelizable, Flink's task managers distribute the execution of this operator over multiple (distributed) instances. The tasks implementation are outlined here:

Ingestion. The ingestion is implemented as an *input operator* in a Flink pipeline. Different data formats (CSV, JSON and XML are indicatively supported) require dedicated input operators. Currently a data source in CSV format is implemented as a *parallel* input operator, i.e. it can be parallelized over multiple (distributed) nodes, while data sources in JSON and XML as *sequential* input operators. This occurs because data in CSV format consists of independent rows which can be split and consumed in parallel. Namely, its data is split in *data chunks*, which, in turn, can be handled *in parallel*, as well as its *data records*. If the structure of the data does not opt for parallel consumption, the data will be read sequentially by a single node, but still without loading everything in memory.

Mapping. The mapping is implemented as a custom *transformation operator* in a Flink pipeline. It contains an engine that supports rules in RML to specify how RDF terms and triples are generated. Transformation operators allow RDF to be generated as output, from a certain input, according to a given function. RML rules can be given to the transformation operator as a side parameter.

Combination. The combination is implemented as an *output operator* in a Flink pipeline. Output operators merge all intermediary results from a previous operator in the pipeline to a final output, namely the results of the different mapping operators are merged.

4.2 Workflow

The RMLStreamer workflow consists of the following steps: (i) **Flink runtime setup**, a Flink runtime must be configured and start running; (ii) **mapping configuration**, a set of rules must be provided which are used to execute on the Flink runtime; and (iii) **RMLStreamer execution**, the three subtasks (*ingestion, mapping, combination*) of our proposed approach are performed.

Flink runtime setup. A Flink runtime is a configuration of a local node or a cluster of nodes (Flink runtime, Figure 2 ⓐ) managed by task managers. Flink's task managers can execute and distribute created Flink jobs (Flink task manager, Figure 2 ⓑ). Flink jobs are executables that contain all tasks of a pipeline to be executed.

Mapping configuration. The execution is driven by rules in RML (RML rules, Figure 2 ⓒ). These rules define the semantic annotations to be applied to certain data. Different rules refer and, thus, are applied to data records from different data sources. Therefore, this influences the execution's parallelization, because both ingesting the data sources and applying the rules to the records of those data sources are executed in parallel.

Execution. When the RML rules are provided as input, an executable is created that contains a Flink job (Flink job, Figure 2 ⓓ). The three subtasks: *ingestion*, *mapping* and *combination* that were discussed before as part of a Flink pipeline, are embedded in this job as tasks. This job is executed on a Flink runtime (Flink runtime, Figure 2 ⓐ). After the job is created, it is given to the Flink runtime for execution (Flink job given to Flink runtime, Figure 2 ⓔ). Flink's task managers distribute the execution of all tasks in the job's pipeline over all available nodes of the Flink runtime. Namely, the ingestion and mapping tasks are distributed and parallelized over different nodes managed by the Flink runtime.

5 EVALUATION

We conducted a comparative study focused on data with high volume and compared our approach to SPARQL-Generate, as it supports multiple data formats, including CSV, JSON, and XML. We elaborate on the applied methodology and discuss the results.

Methodology. We generate RDF from artificial datasets with person details (incl. id, name, phone, email, birth date, height, weight, and company) in different formats and measure the duration of this task for both the RMLStreamer and SPARQL-Generate. We created 15 datasets[16] with their number of records ranging from 1,000,000 till 5,000,000 and with CSV, XML, or JSON as their data format, together with the corresponding mapping files[17]. The records in the JSON and XML files have a flat structure (no hierarchy) to reflect the same data records of the CSV files, which are flat by definition. Note that the RMLStreamer is not limited to handling flat records. For each data source, the corresponding RDF is generated. For every person, an entity is created together with 6 attributes.

We measure the time it takes to (i) start the tool, (ii) generate the corresponding RDF, and (iii) stop the tool. The starting and stopping time of the tool is included, as it might have an impact on the total duration to generate the RDF. We accomplished this by creating a Docker container for both the RMLStreamer and SPARQL-Generate. For each dataset, the duration between when the container starts and when it stops is recorded. With the RMLStreamer we generated RDF four times, because Flink is able to use 1, 2, 3, or 4 nodes (or more), but one with SPARQL-Generate because this cannot be configured. This results in four different measurements for a single dataset for the RMLStreamer. The scripts to orchestrate the evaluation are available at https://doi.org/10.6084/m9.figshare.6106706.v1.

The evaluation was executed on a machine with 24 cores (Intel Xeon CPU E5-2620 v3 @ 2.40GHz) and 128GB Random Access Memory (RAM). Each Docker container could use as many cores as desired, but the total amount of memory was limited to 4GB.

[16]http://rml.io/data/sbd2019/boundeddata/data
[17]https://doi.org/10.6084/m9.figshare.6108623.v1

Results. The RMLStreamer outperforms SPARQL-Generate regardless of the data format and number of nodes used. Furthermore, SPARQL-Generate runs out of memory when dealing with larger datasets, which is not the case for the RMLStreamer. In the following paragraph we provide detailed results. The duration of the RDF generation process from the different datasets is available in Figures 3 to 5 for data in CSV, XML, and JSON format.

The RMLStreamer outperforms SPARQL-Generate for data in CSV format, regardless of the number of nodes used by Flink, reaching to a performance increase of 75% when using 4 nodes. Even more, for 2,000,000 records and more, SPARQL-Generate either exceeds the limit of Java's default garbage collection or runs out of memory. The RMLStreamer does not have these issues, thanks to the efficient memory management of Flink.

The overall trend for the RMLStreamer is linear in function of the number of records, as it is indicate on Figure 3. The improvement on duration gets reduced as the number of nodes increases. For instance, when using 1 and 2 nodes, the durations for 1,000,000 records are 97s and 62s respectively, resulting in a difference of 36%. However, when using 3 and 4 nodes, the durations are 48s and 41s, resulting in a difference of only 15%.

For data with a one-level hierarchical depth, i.e., XML and JSON, the results are similar. The RMLStreamer outperforms SPARQL-Generate, regardless of the number of nodes used by Flink. There is a performance increase of 62%. Even more, for 2,000,000 records and more, SPARQL-Generate runs out of memory. The RMLStreamer does not have this issue, due to the efficient memory management of Flink. According to Figures 4 and 5, as with CSV, the overall trend of the RMLStreamer is linear.

Insights. The RMLStreamer outperforms SPARQL-Generate when dealing with bounded data in different data formats, which is attributable to the parallelization. However, an increase of the number of nodes does not necessarily result in an increase of performance. The specifics of each use case determine the appropriate number.

6 CONCLUSIONS

In this paper, we identified the aspects of RDF generation that can be parallelized to scale with data volume. We observed that parallelism can be introduced in (i) the ingestion of both data sources and their data records, and (ii) the mapping task. Thus, we introduced an approach that incorporates parallelization in the RDF generation process by aligning three subtasks (*ingestion, mapping, combination*), following the producer–consumer paradigm.

We presented as a proof-of-concept our implementation, the RMLStreamer, which follows our proposed approach. The RMLStreamer builds on top of Flink for handling the parallel execution. Our evaluation shows that the RMLStreamer is able to handle high volumes of data that other RDF generators cannot. Furthermore, when the data format allows parallelism, the generation speed increases proportionally with the parallelism.

In the future, we will continue applying parallelism to join multiple data sources even more efficiently and ingest more complicated data sources, e.g. distributed file systems.

7 ACKNOWLEDGMENTS

The described research activities were funded by Ghent University, imec, Flanders Innovation & Entrepreneurship (VLAIO), and the European Union. Ruben Verborgh is a postdoctoral fellow of the Research Foundation – Flanders (FWO).

REFERENCES

[1] Stefan Bischof, Stefan Decker, Thomas Krennwallner, Nuno Lopes, and Axel Polleres. 2012. Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics* (2012). https://doi.org/10.1007/s13740-012-0008-7

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[3] Souripriya Das, Seema Sundara, and Richard Cyganiak. 2012. *R2RML: RDB to RDF Mapping Language*. Working Group Recommendation. W3C. http://www.w3.org/TR/r2rml/.

[4] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. 2014. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Workshop on Linked Data on the Web*.

[5] Kevin Jeffay. 1993. The Real-time Producer/Consumer Paradigm: A Paradigm for the Construction of Efficient, Predictable Real-time Systems. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice (SAC '93)*. ACM, New York, NY, USA, 796–804. https://doi.org/10.1145/162754.168703

[6] Maxime Lefrançois, Antoine Zimmermann, and Noorani Bakerally. 2017. A SPARQL Extension for Generating RDF from Heterogeneous Formats. In *The Semantic Web: 14th International Conference, ESWC 2017, PortoroÅ¿, Slovenia, May 28 âĂŞ June 1, 2017, Proceedings*. Springer International Publishing, Portoroz, Slovenia, 35–50. https://doi.org/10.1007/978-3-319-58068-5_3

[7] Wouter Maroy, Anastasia Dimou, Dimitris Kontokostas, Ben De Meester, Ruben Verborgh, Jens Lehmann, Erik Mannens, and Sebastian Hellmann. 2017. Sustainable Linked Data Generation: The Case of DBpedia. In *The Semantic Web – ISWC 2017*, Claudia d'Amato, Miriam Fernandez, Valentina Tamma, Freddy Lecue, Philippe Cudré-Mauroux, Juan Sequeda, Christoph Lange, and Jeff Heflin (Eds.). Springer International Publishing, Cham, 297–313.

[8] Franck Michel, Loïc Djimenou, Catherine Faron-Zucker, and Johan Montagnat. 2015. Translation of Relational and Non-relational Databases into RDF with xR2RML. In *WEBIST*.

[9] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

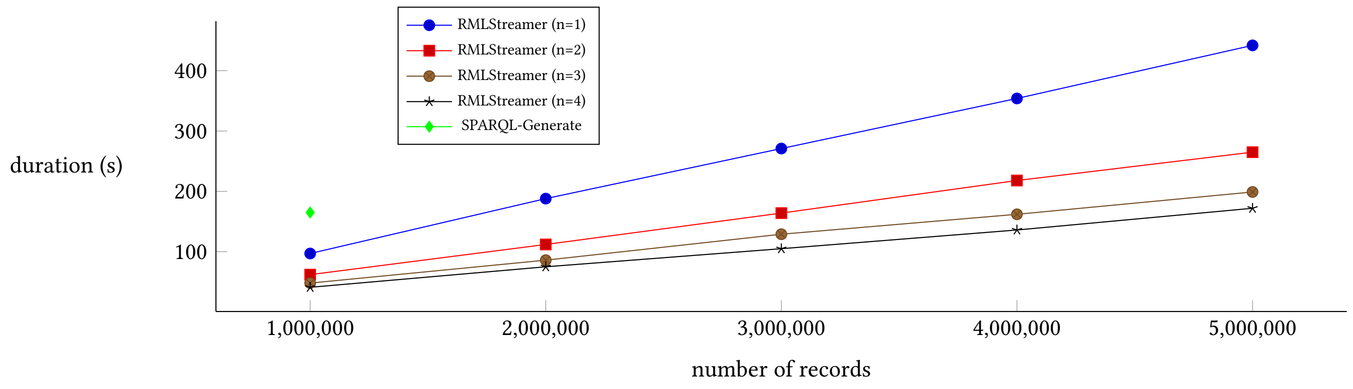

Figure 3: The processing time of RMLStreamer scales linearly with the number of records, whereas SPARQL-Generate only handles CSV datasets of 1,000,000 records. Using more nodes reduces the needed time, as the input is ingested in parallel.

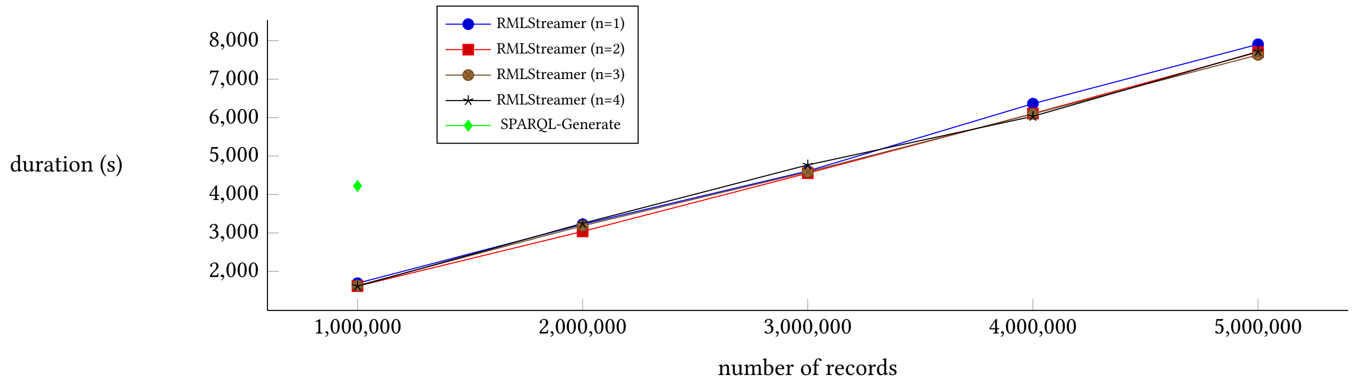

Figure 4: The RMLStreamer's processing time scales linearly with the number of records, whereas SPARQL-Generate only handles XML datasets of 1,000,000 records.

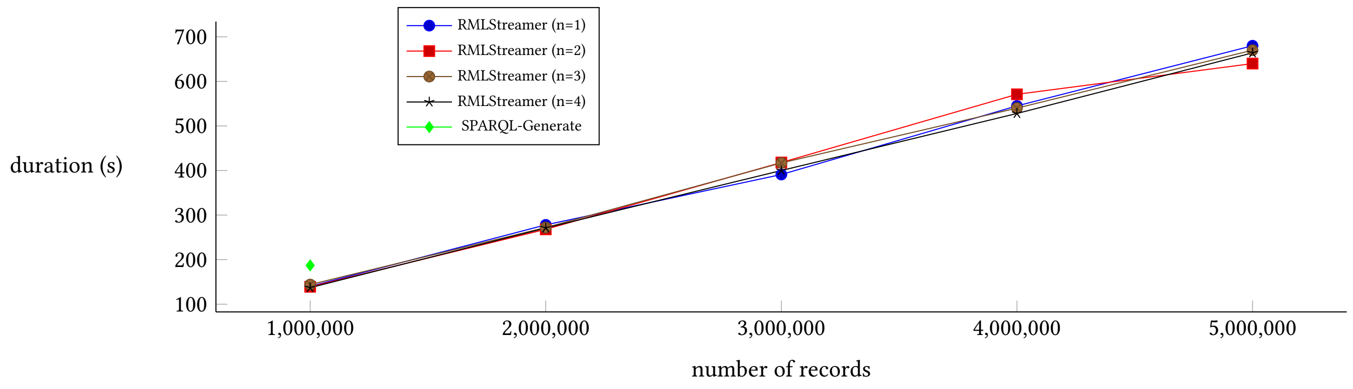

Figure 5: The RMLStreamer's processing time scales linearly with the number of records, whereas SPARQL-Generate only handles JSON datasets of 1,000,000 records.