Demonstrating the Power of Streaming and Sorting for Non-distributed RDF Processing: RDF_{pro}*

Francesco Corcoglioniti, Alessio Palmero Aprosio, and Marco Rospocher

Fondazione Bruno Kessler – IRST, Via Sommarive 18, Trento, I-38123, Italy
{corcoglio,aprosio,rospocher}@fbk.eu

Abstract. We demonstrate RDF_{pro} (RDF Processor), an extensible, general-purpose, open source tool for processing large RDF datasets on a commodity machine leveraging streaming and sorting techniques. RDF_{pro} provides out-of-the-box implementations – called *processors* – of common tasks such as data filtering, rule-based inference, smushing, and statistics extraction, as well as easy ways to add new processors and arbitrarily compose processors in complex pipelines.

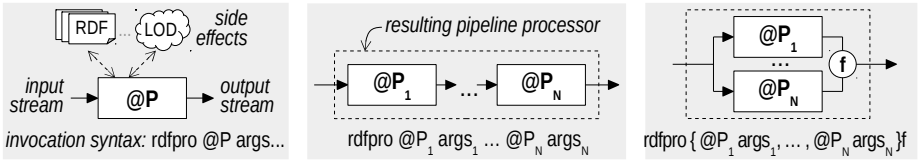
1 Introduction

Processing of RDF data – e.g., Linked Open Data (LOD) – often requires performing a number of common processing tasks such as triple-level filtering and/or transformation, inference materialization, owl:sameAs smushing (i.e., replacing URI aliases with a "canonical" URI), and statistics extraction. Although tools do exist for these tasks, a Semantic Web practitioner typically faces two challenges. First, tool support is fragmented, often forcing a user to integrate many heterogeneous tools even for simple processing workflows. Second, tools coping with LOD dataset sizes in the range of millions to billions of triples often require distributed infrastructures such as *Hadoop* (e.g., *WebPIE* [1] for forward-chaining inference, *voidGen* [2] for VoID statistics extraction), which are complex to set up for the user and cannot be used efficiently – if not at all – on a single machine due to their inherent complexity and overhead.

Given these premises we demonstrate RDF_{pro} (RDF Processor) [3,4], a tool and Java library addressing these shortcomings. On the one hand, RDF_{pro} reduces integration efforts by providing out-of-the-box implementations – called *processors* – of common RDF processing tasks, as well as easy ways to add new processors and compose processors in complex processing pipelines. On the other hand, RDF_{pro} targets local processing of large datasets without requiring clusters and complex computing infrastructures. Vertical scalability is achieved with multi-threading and a processing model based on *streaming* and *sorting*, two scalable techniques well-known in the literature [5]. Streaming consists in processing one triple at a time, translates to efficient sequential I/O, and is at the basis of tools such as *LODStats* [6] for scalable approximate statistics extraction and *Jena RIOT*[1] for partial RDFS inference. Sorting overcomes many of the limitations of a pure streaming model, supporting tasks such as duplicate removal, set operations, and grouping of data that must be processed together (e.g., all the data about an entity). We describe RDF_{pro} in Section 2 and discuss its use and demonstration in Section 3.

* Partially funded by the European Union's FP7 via the NewsReader Project (ICT-316404).

[1] https://jena.apache.org/documentation/io/



(a) (b) (c)

Fig. 1: RDF processor (a); sequential composition (b); and parallel composition (c).

2 The RDF$_{pro}$ Tool

At its core, RDF$_{pro}$ is a Java multi-threaded engine based on the Sesame library[2] that executes *RDF processors*, supports their *sequential and parallel composition* in processing pipelines, and provides several *builtin processors* for common processing tasks.

RDF Processor A processor @P (Figure 1a) is a software component that consumes RDF *quads* from an input stream in one or more passes, produces an output stream of quads, and may have an internal state as well as side effects like writing RDF data. An RDF quad is a triple with an optional fourth named graph component, which is unspecified for plain triples and triples in the default graph of the RDF dataset. Streaming characterizes the way quads are processed: one at a time, with no possibility for the processor to "go back" in the input stream and recover previously seen quads. Sorting is offered to processors as a primitive to arbitrarily sort selected data during a pass. This primitive is realized on top of the native `sort` Unix utility that support *external sorting*, using *dictionary encoding* techniques to compactly encode frequently used RDF terms (e.g., TBox ones) in order to reduce the size of sorted data and improve performances.

Sequential and Parallel Composition Composition can be applied recursively to build *pipeline processors* starting from a fixed set of basic processors. In a *sequential composition* (Figure 1b), two or more processors @P$_i$ are chained so that the output stream of @P$_i$ becomes the input stream of @P$_{i+1}$. In a *parallel composition* (Figure 1c), the input stream is sent concurrently to several processors @P$_i$, whose output streams are merged into a resulting stream using one of several possible set operators, such as union with/without duplicates. Composition supports complex processing tasks that cannot be tackled with a single processor. Moreover, executing a pipeline processor is often faster than executing the processors it is composed of separately (if separate execution is possible), as input data is parsed once and I/O costs for intermediate files are eliminated.

Builtin Processors The basic processors included in RDF$_{pro}$ are listed below:

@read Reads RDF file(s), emitting their quads together with the input stream. Files are read in parallel and, where possible, split in chunks that are parsed concurrently.

@write Writes quads to one RDF file or splits them to multiple files evenly; quads are also propagated in output. Parallel, chunk-based writing is supported as for @read.

@download Emits data downloaded from a SPARQL endpoint using a query.

@upload Uploads input data to an RDF store using SPARQL INSERT DATA calls.

@tbox Filters the input stream by emitting only quads belonging to TBox axioms.

[2] http://rdf4j.org/

@transform Discards or rewrites input quads one at a time, either based on simple matching criteria or based on an arbitrarily complex JavaScript or Groovy[3] script.

@smush Performs *smushing*, replacing the members of each owl:sameAs equivalence class with a canonical URI selected based on a ranked namespace list.

@rdfs Computes the RDFS deductive closure of an input stream consisting only of ABox quads. A fast, hard-coded implementation loads the TBox from a file and computes its closure first, using the resulting domain, range, sub-class, and sub-property axioms to perform inference on quads of the input stream one at a time.

@rules Emits the closure of input quads using a customizable set of rules. Rules heads and bodies are SPARQL graph patterns, with FILTER, BIND, and UNION constructs allowed in the body. The current implementation is based on Drools.[4]

@mapreduce Applies a custom *map* script (JavaScript or Groovy) to label and group input quads into partitions, each one reduced with a *reduce* script. A multi-threaded, non-distributed MapReduce implementation based on the sort primitive is used.

@stats Computes VoID [7] structural statistics for its input, plus additional metadata and informative labels for TBox terms that can be shown in tools such as Protégé.

@unique Discards duplicate quads in the input stream.

3 Using RDF$_{pro}$

RDF$_{pro}$ binaries and public domain sources are available on its website.[5] RDF$_{pro}$ can be used in three ways: (i) as a command line tool able to process large datasets; (ii) as a web tool suited to smaller amounts of data uploaded/downloaded with the browser; and (iii) as a Java library[6] embedded in applications. Users can extend RDF$_{pro}$ via custom scripts and rulesets, while developers can create new processors by implementing a simple Java API and focusing on the specific task at hand, as efficient streaming, sorting, I/O, thread management, scripting, and composition facilities are already provided.

Examples of using RDF$_{pro}$ as a command line and web tool are shown in Figures 2a and 2b, where a pipeline is executed to compute the RDFS closure of some DBpedia data (70M triples) and return only rdfs:label triples of entities of type dbo:Company. The pipeline performs 6 tasks: (i) read data; (ii) compute RDFS closure using DBpedia TBox; (iii) keep rdf:type and rdfs:label quads; (iv) partition quads by subject, keeping partitions with object dbo:Company; (v) retain rdfs:label quads; (vi) write results. Examples of applications using RDF$_{pro}$ as an embedded Java library for RDF I/O, filtering, smushing, RDFS and rule-based inference are the KnowledgeStore [8] and PIKES [9] (code publicly available); instructions for using the library are provided on the website.

A video showing the usage of RDF$_{pro}$ is available on the website, together with a fully-working installation of the RDF$_{pro}$ web interface, where users can try arbitrary commands and processing tasks. The demo will mainly focus on using this web interface on suitable examples, to demonstrate the usability of RDF$_{pro}$ in tasks such as the ones considered in [3,4]. Use of RDF$_{pro}$ as a command line tool on large datasets or as a library in a sample application will also be demonstrated to interested attendees.

[3] Groovy is a scripting language based on Java and its libraries. See http://groovy.codehaus.org/

[4] Drools is a rule engine implementing the RETE algorithm. See http://www.drools.org/

[5] http://rdfpro.fbk.eu/

[6] Available on Maven Central: http://repo1.maven.org/maven2/eu/fbk/rdfpro/.

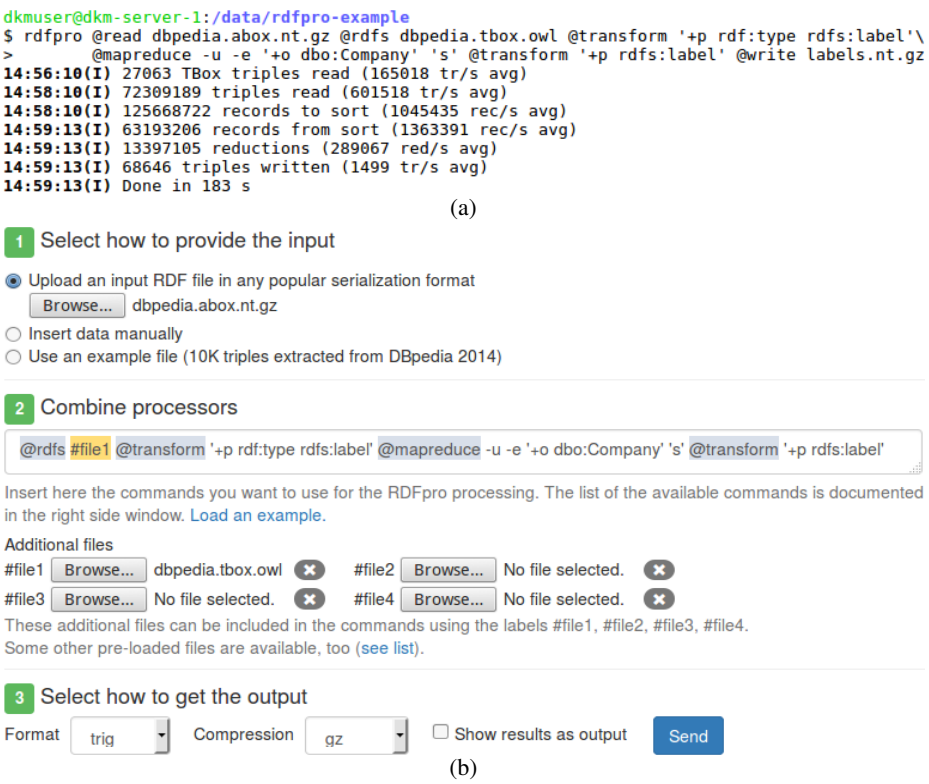


Fig. 2: Using the command line (a) and web (b) interfaces of RDF_{pro}.

References

1. Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., Bal, H.: WebPIE: A web-scale parallel inference engine using MapReduce. J. Web Semant **10** (2012) 59–75
2. Böhm, C., Lorey, J., Naumann, F.: Creating VoID descriptions for web-scale data. J. Web Semant. **9**(3) (September 2011) 339–345
3. Corcoglioniti, F., Rospocher, M., Amadori, M., Mostarda, M.: RDFpro: an extensible tool for building stream-oriented RDF processing pipelines. In: Proc of ISWC Developers Workshop 2014. Volume 1268 of CEUR Workshop Proceedings., CEUR-WS.org (2014) 49–54
4. Corcoglioniti, F., Rospocher, M., Mostarda, M., Amadori, M.: Processing billions of RDF triples on a single machine using streaming and sorting. In: ACM SAC. (2015) 368–375
5. O'Connell, T.: A survey of graph algorithms under extended streaming models of computation. In: Fundamental Problems in Computing. Springer Netherlands (2009) 455–476
6. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODStats - an extensible framework for high-performance dataset analytics. In: EKAW. (2012) 353–362
7. Cyganiak, R., Zhao, J., Hausenblas, M., Alexander, K.: Describing linked datasets with the VoID vocabulary. W3C note, W3C (2011)
8. Corcoglioniti, F., Rospocher, M., Cattoni, R., Magnini, B., Serafini, L.: The KnowledgeStore: a Storage Framework for Interlinking Unstructured and Structured Knowledge. Int. J. Semantic Web Inf. Syst. (to appear)
9. Corcoglioniti, F., Rospocher, M., Palmero Aprosio, A.: Extracting Knowledge from Text with PIKES. In: ISWC 2015 Posters & Demonstrations Track. (to appear)