RDF$_{PRO}$: an Extensible Tool for Building Stream-Oriented RDF Processing Pipelines*

Francesco Corcoglioniti, Marco Rospocher, Marco Amadori, and Michele Mostarda

Fondazione Bruno Kessler, Via Sommarive 18, 38123 Trento, Italy
{corcoglio,rospocher,amadori,mostarda}@fbk.eu

Abstract. We present RDF$_{PRO}$ (RDF Processor), an open source Java command line tool and embeddable library that offers a suite of stream-oriented, highly optimized processors for common tasks such as data filtering, RDFS inference, smushing and statistics extraction. RDF$_{PRO}$ processors are extensible by users and can be freely composed to form complex pipelines to efficiently process RDF data in one or more passes. We show how RDF$_{PRO}$ model and multi-threaded design allow processing billions of triples in few hours in a typical Linked Open Data integration scenario, and discuss relevant implementation aspects and lessons learnt.

1 Introduction

The amount of RDF data available for application consumption is steadily increasing, thanks also to the Linked Open Data (LOD) initiative. Although remote and on-the-fly RDF consumption is possible via SPARQL and URI dereferencing, many applications must collect and pre-process data locally before using it. A typical motivating scenario is the creation of a specifically-purposed dataset by integrating data of popular LOD sources such as DBpedia, Freebase or GeoNames, e.g., to support knowledge-intensive applications such as content-enrichment. This scenario involves a number of processing tasks that are often fine-tuned in an iterative and exploratory process, as a better understanding of sources and their "idiosyncrasies" is acquired. Common tasks are:

- *filtering* source data, removing redundant or otherwise unwanted triples;
- *merging* data of different datasets, removing duplicates and tracking provenance;
- performing *inference*, materializing the deductive closure of data to avoid the need for expensive (and often unsupported) query-time inference when accessing it;
- performing *smushing*[1], i.e., select and use unique "canonical" URIs for entities with multiple aliases, easing data usage as owl:sameAs links may then be ignored;
- computing *statistics* on classes and predicates usage, to summarize dataset contents.

Although many of the tasks above have received considerable attention in the literature, tool support is limited and fragmented, with users often forced to integrate and complement existing tools in a time-consuming and error-prone process requiring software development skills. Moreover, tools based on *MapReduce* or other distributed paradigms, such as *voidGen* [8], *LDIF* [11] and *Infovore* [2], require the availability of a computer cluster, while tools based on SPARQL data manipulation on top of triple stores, such as *make-void* [3] and *RDFStats* [10], require very powerful machines for

* The work described in this paper has been supported by the European Union's 7th Framework Programme via the NewsReader Project (ICT-316404, http://www.newsreader-project.eu/)

[1] http://patterns.dataincubator.org/book/smushing.html

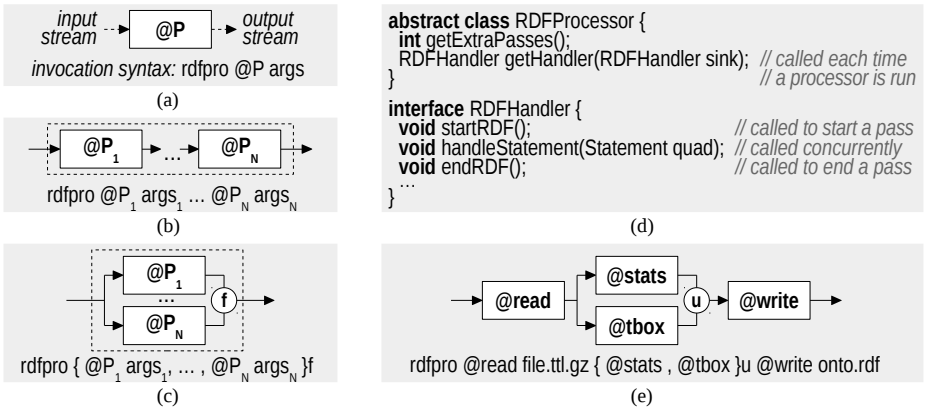

Fig. 1. RDF processor (a); sequence (b) and parallel (c) composition; Java API (d); example (e).

typical LOD dataset sizes (hundreds to thousands millions triples). Tools processing large datasets on a commodity machine exist and are often based on a *streaming* computation model, but they deal essentially with RDF format conversion, such as *rapper* [4] and *rdfpipe* [5], with the notable exceptions of *Jena riot* [1] and *LODStats* [7] providing partial RDFS inference and (approximate) statistics extraction. In this setting, processing large datasets is a challenge, especially if limited hardware resources are available.

On these premises we propose RDF$_{\text{PRO}}$ (RDF Processor)[2], an open source Java library and command line tool for stream-oriented RDF processing on a single machine. RDF$_{\text{PRO}}$ is based on Sesame[3] and offers a suite of *processors* for common tasks, which can be composed in complex pipelines to efficiently process RDF data in one or more passes. These features allow RDF$_{\text{PRO}}$ to address a variety of processing needs with a single tool usable by casual users and not just developers, making it a sort of "swiss-army-knife" for exploring and manipulating RDF datasets. At the same time, RDF$_{\text{PRO}}$ provides an extensible processing model that allows developers to create new processors by focusing on the specific task at hand, as efficient I/O, thread management and pipeline integration are provided. We describe RDF$_{\text{PRO}}$ in Section 2 and apply it to the motivating integration scenario in Section 3; we discuss relevant implementation aspects and lessons learnt in Section 4 and report some concluding remarks in Section 5.

2 Tool Description

RDF$_{\text{PRO}}$ processing model is centred around the concept of *RDF processor* (Figure 1a), a Java component that consumes an *input stream* of RDF quads—i.e., RDF triples with an *optional* fourth named graph component[4]—in one or more passes, produces an *output stream* of quads and may have side effects like writing RDF data. Technically, a processor extends the `RDFProcessor` class (Figure 1d), declaring how many passes it needs on its input and producing an `RDFHandler` (Sesame interface) where quads can be fed and handled *concurrently* by multiple threads, the result sent to a sink `RDFHandler`.

[2] http://fracor.bitbucket.org/rdfpro/

[3] http://www.openrdf.org/

[4] The graph component is unspecified for triples in the *default graph* of the RDF dataset (see RDF 1.1 and SPARQL specifications); this allows using RDF$_{\text{PRO}}$ on plain triple data.

RDF$_{PRO}$ offers processors for common tasks that can be easily extended by users. Importantly, RDF$_{PRO}$ allows to derive new processors by (recursively) applying sequential and parallel compositions. 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 more processors @P$_i$, whose output streams are merged into a resulting stream based on one of several possible *merge criteria* (given by flag `f`): union with and without duplicates (flags `a`, `u`), intersection (`i`) and difference (`d`) of quads from different branches.

An example of composition is shown in Figure 1e, where a Turtle+gzip file is read (`file.ttl.gz`), TBox and VOID [6] statistics are extracted in parallel and their union is written to an RDF/XML file (`onto.rdf`). This example shows how I/O can be done also using specific @read and @write processors that augment or dump the stream at any point of the pipeline, removing the limit of single input and output streams. Indeed, the RDF$_{PRO}$ tool relies on these processors for all the I/O, ignoring the global input and output streams that are instead accessible when using RDF$_{PRO}$ as a library.

The following processors are included in RDF$_{PRO}$:

@read Reads one or more RDF files in parallel, emitting the input stream augmented with parsed quads. May rewrite bnodes on a per-file basis to avoid clashes.

@write Writes input quads to one RDF file or divides them evenly among multiple RDF files, so to allow splitting large datasets; quads are also propagated in output.

@download Retrieves quads from a SPARQL endpoint and emits them together with the input stream. CONSTRUCT queries return quads in the default graph (i.e., triples). SELECT queries produce quads based on the bindings of variables `s`, `p`, `o`, `c`.

@upload Uploads quads in the input stream to a triple store using SPARQL Update calls, in chunks of a specified size; quads are also propagated in the output stream.

@transform Processes each input quad with a *Groovy*[5] expression that can either discard the quad, propagate it or transform it into one or more output quads. The expression can include a mix of Groovy and Java code and SPARQL 1.1 functions.

@smush Performs *smushing* in two passes: the first extracting the owl:sameAs graph; the second replacing URIs. Canonical URIs are chosen based on a ranked namespace list and are linked in output to coreferring URI aliases via owl:sameAs quads.

@infer Computes the RDFS closure of its input. The TBox, read from a file, is closed and emitted first. Domain, range, sub-class and sub-property axioms are then used to do inference on input quads one at a time, placing inferences in the graph of the input quad.[6] Specific RDFS rules can be disabled to avoid unwanted inferences.

@tbox Filters the input stream by emitting only quads of TBox axioms. Both RDFS and OWL axioms are extracted, even if the latter are not used by @infer.

[5] Groovy is a scripting language reusing Java syntax and libraries: http://groovy.codehaus.org/

[6] This scheme avoids join operations and works with arbitrarily large datasets whose TBox fits into memory. Inference is complete if: (i) domain, range, sub-class and sub-property axioms in the input stream are also in the TBox; and (ii) the TBox has no quad matching patterns:
- X rdfs:subPropertyOf {rdfs:domain|rdfs:range|rdfs:subPropertyOf|rdfs:subClassOf}
- X {rdf:type|rdfs:domain|rdfs:range|rdfs:subClassOf} rdfs:ContainerMembershipProp.
- X {rdf:type|rdfs:domain|rdfs:range|rdfs:subClassOf} rdfs:Datatype.

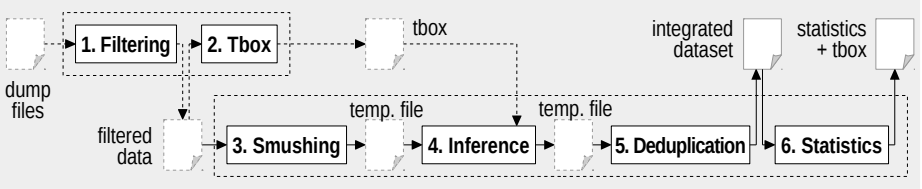

Fig. 2. Processing steps for the motivating scenario of Section 1.

@stats Emits VOID structural statistics for its input. A VOID dataset is associated to the whole input and to each set of graphs annotated with the same *source* URI; class and property partitions are produced for each dataset. Additional terms extend VOID to express the number of TBox, ABox, rdf:type and owl:sameAs quads, the average number of properties per entity and informative labels for TBox terms.

@unique Discards duplicates in the input stream. Optionally, it merges quads with the same subject, predicate and object but different graphs in a unique quad. To track provenance, this quad is placed in a "fusion" graph linked to all the original graphs.

3 Using the Tool

We apply RDF$_{\text{PRO}}$ to the scenario of Section 1, using a small workstation[7] and assuming as a proof-of-concept that the user wants to integrate multi-lingual data from Freebase, GeoNames and DBpedia in the languages EN, ES, IT and NL.[8] Processing with RDF$_{\text{PRO}}$ involves the six steps reported below and in Figure 2 (key processors in parenthesis):

Step 1 *Filtering* (@transform). Downloaded files are filtered to extract desired RDF quads and place them in separate graphs to track provenance. A metadata file is added to link each graph to the URI of the associated *source* (e.g., Freebase).

Step 2 *TBox extraction* (@tbox, @transform). TBox quads are extracted from filtered data and stored, dropping unwanted top level classes and vocabulary alignments.

Step 3 *Smushing* (@smush). Filtered data is smushed so to use canonical URIs for each owl:sameAs equivalence class, producing an intermediate smushed file.

Step 4 *Inference* (@infer). The deductive closure of smushed data is computed and saved, using the extracted TBox and excluding RDFS rules *rdfs4a*, *rdfs4b* and *rdfs8* to avoid inferring uninformative ⟨ X rdf:type rdfs:Resource ⟩ quads.

Step 5 *Deduplication* (@unique). Quads with the same subject, predicate and object are fused and placed in a graph linked to the original sources to track provenance.

Step 6 *Statistics extraction* (@stats). VOID statistics are extracted and merged with TBox data, forming an annotated ontology that documents the produced dataset.

These steps can be executed separately by calling RDF$_{\text{PRO}}$ six times, leading to the execution times, throughputs, input and output sizes (quads and compressed bytes) reported in the upper part of Table 1. TBox extraction and filtering are fast (the latter only for certain files), while smushing and inference add duplicates that are removed in the deduplication step. Steps 1-2 and 3-6 can also be aggregated as shown with dotted boxes

[7] Intel Core I7 860 CPU (4 cores), 16 GB RAM, 500 GB 7200 RPM hard disk, Linux 2.6.32.

[8] Data selection details are omitted but can be found on the RDF$_{\text{PRO}}$ web site (*Example* page), together with scripts for downloading the data and processing it as described in this section.

Table 1. Input and output size, throughput (w.r.t. input) and execution time of processing steps.

Processing step	Input size		Output size		Throughput		Time
	[Mquads]	[GB]	[Mquads]	[GB]	[Mquads/s]	[MB/s]	[s]
Step 1 - Filtering	3019.89	29.31	750.78	9.68	0.57	5.70	5266
Step 2 - TBox extraction	750.78	9.68	0.15	0.01	1.36	18.00	551
Step 3 - Smushing	750.78	9.68	780.86	10.33	0.31	4.04	2453
Step 4 - Inference	781.01	10.34	1693.59	15.56	0.22	2.91	3630
Step 5 - Deduplication	1693.59	15.56	954.91	7.77	0.38	3.61	4413
Step 6 - Statistics extract.	954.91	7.77	0.32	0.01	0.36	3.02	2640
Steps 1-2 aggregated	3019.89	29.31	750.92	9.69	0.56	5.60	5363
Steps 3-6 aggregated	750.92	9.69	955.23	7.78	0.09	1.21	8168

in Figure 2, exploiting RDF_{PRO} composition capabilities. The resulting performance figures, reported in the lower part of Table 1, show a marked 28% reduction of the total processing time (from 18953 s to 13531 s) due to reduced I/O for temporary files.

4 Implementation Notes

From an implementation perspective, the distinctive feature of RDF_{PRO} is its streaming, multi-threaded processing model, which is embodied in the RDF processor definition of Section 2 and enables the full utilization of available CPU resources. Indeed, one of RDF_{PRO} goals is to read data as fast as possible and involve all the available CPU cores in its processing. This is achieved by parsing multiple RDF files in parallel and, for line-oriented RDF formats, by splitting them in newline-terminated chunks that are processed concurrently, achieving substantial speed improvements (e.g., from 610K quad/s to 1450K quad/s for Freebase NTriples+gzip data); data writing is performed similarly to avoid bottlenecks. Another mechanism for introducing parallelism is the use of a special queue in front of each processor. The queue collects a fraction of incoming quads and triggers their processing in a separate thread when full; the fraction is adapted at runtime using heuristics trying to ensure that all CPU cores are exploited.

Another relevant aspect of RDF_{PRO} is its use of *external sorting* (using the native `sort` utility and compact data encoding) for tasks that cannot be done one quad at a time, enabling their execution on arbitrarily large inputs at the price of some throughput reduction and temporary disk space usage (we measured $\sim$40 bytes/quad on real-world data, which easily translates to many GBs of data when processing large datasets). Sorting is used with @unique and the parallel composition, with intersection and difference implemented by appending a label with the operand index to each quad sent to `sort`, and then gathering all the labels of a sorted quad to decide whether to emit it. Sorting is used also with @stats, by (conceptually) sorting the quad stream twice: first based on the subject, to group quads about the same entity and compute entity-based and distinct subjects statistics; then based on the object, to compute distinct objects statistics.

Quads in RDF_{PRO} are processed one at a time and few data must be retained in memory, which is then exploited for I/O buffers. However, the @stats and @smush processors may need a lot of memory for tracking statistics and owl:sameAs equivalence classes, and the design of specialized in-memory data structures that are fast and compact at the same time proved to be a crucial and challenging task. To give an ex-

ample, @smush uses raw buffers to store URIs, which are indexed using a custom hash table with an open addressing scheme; table entries contain also a *next pointer* that organizes URIs of an owl:sameAs class in a circular linked list, which expands as new owl:sameAs quads are encountered. This 'low level' structure grows linearly with the number of URIs and presents a very limited overhead (differently from a solution based on Java `String`s and `HashMap`s), making it possible to smush an owl:sameAs graph of ∼38M URIs and ∼8M equivalence classes using ∼2 GB of RAM (∼56 bytes/URI).

A final note concerns data formats and compression. Use of uncompressed data is inefficient, while throughputs are better for Freebase NTriples+gzip data (939K quad/s in the filtering task) and worse for DBpedia Turtle+bzip2 (253K quad/s) and GeoNames RDF/XML+zip data (68K quad/s), showing the impact of format and compression on processing speed. Using native compression utilities is also beneficial, especially if their parallel variants are employed (e.g., `pigz` and `pbzip2`). While developing RDF$_{\text{PRO}}$ we also had problems with handling Turtle and TriG data, as "unusual" URIs ending with a period were incorrectly serialized by Sesame but then could not be parsed. The issue is related to the migration to RDF 1.1 and exemplifies the difficulties a SW developer may encounter when building and using libraries due to evolution of standards.

5 Conclusions

We presented RDF$_{\text{PRO}}$, a tool for processing RDF data in stream-oriented pipelines, and described its practical use in an integration scenario involving large amounts of data and non-trivial processing tasks. RDF$_{\text{PRO}}$ has been developed in the NewsReader project, where it is used to process generated RDF data and build background knowledge datasets (linked on RDF$_{\text{PRO}}$ web site) from multi-lingual LOD sources. Future work include better entity-based filtering (vs. quad-based) and better inference support (e.g., OWL-LD [9]). We released RDF$_{\text{PRO}}$ code in the Public Domain to promote its reuse.

References

1. Jena `riot`. https://jena.apache.org/documentation/io/, visited 2014-09-30
2. `Infovore`. https://github.com/paulhoule/infovore, visited 2014-09-30
3. `make-void`. https://github.com/cygri/make-void, visited 2014-09-30
4. `rapper`. http://librdf.org/raptor/rapper.html, visited 2014-09-30
5. `rdfpipe`. http://rdfextras.readthedocs.org/en/latest/tools/rdfpipe.html, visited 2014-09-30
6. Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing linked datasets. In: WWW Workshop on Linked Data on the Web (LDOW). vol. 538. CEUR-WS.org (2009)
7. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODStats - an extensible framework for high-performance dataset analytics. In: EKAW. pp. 353–362. Springer-Verlag (2012)
8. Böhm, C., Lorey, J., Naumann, F.: Creating voiD descriptions for Web-scale data. Web Semant. 9(3), 339–345 (Sep 2011), http://dx.doi.org/10.1016/j.websem.2011.06.001
9. Glimm, B., Hogan, A., Krötzsch, M., Polleres, A.: OWL: yet to arrive on the Web of Data? In: WWW Workshop on Linked Data on the Web (LDOW). vol. 937. CEUR-WS.org (2012)
10. Langegger, A., Woss, W.: RDFStats - an extensible RDF statistics generator and library. In: Int. Workshop on Database and Expert Systems Application, DEXA'09. pp. 79–83 (2009)
11. Schultz, A., Matteini, A., Isele, R., Mendes, P.N., Bizer, C., Becker, C.: LDIF - a framework for large-scale Linked Data integration. In: WWW Developers Track (2012)

