Processing Billions of RDF Triples on a Single Machine using Streaming and Sorting

Francesco Corcoglioniti, Marco Rospocher, Michele Mostarda, Marco Amadori
Fondazione Bruno Kessler
Via Sommarive 18, 38123 Trento, Italy
{corcoglio,rospocher,mostarda,amadori@fbk.eu}

ABSTRACT

We consider the feasibility of processing billions of RDF triples on a single commodity machine using streaming and sorting techniques and focusing on RDF processing tasks relevant for Linked Data consumption: data filtering and transformation, RDFS inference, owl:sameAs smushing and statistics extraction. To investigate this research question we built RDF$_{\text{PRO}}$ (RDF Processor), an open source tool that provides streaming and sorting-based processors for the considered tasks and allows their sequential and parallel composition in complex pipelines. An empirical evaluation of RDF$_{\text{PRO}}$ in four application scenario—dataset analysis, filtering, merging and massaging—shows the effectiveness of the tool and allows to positively answer our research question.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods; H.2.4 [**Database Management**]: Systems

General Terms

Design, Algorithms and Experimentation.

Keywords

RDF Processing, Streaming, Sorting, Linked Data, RDF$_{\text{PRO}}$

1. INTRODUCTION

With Semantic Web (SW) technologies coming of age and the public acclaim of the Linked Open Data (LOD) initiative, the last few years have seen the massive proliferation of structured data, especially in RDF form. Although remote and on-the-fly RDF consumption may be possible via SPARQL and URI dereferencing (where reliably implemented), many applications have to collect and pre-process data locally before using it. While the scope of such processing cannot be restricted in principle, there are some relevant RDF processing tasks that are common in practice, especially when preparing LOD data for application consumption [17, 18]: (i) *TBox and statistics extraction*, to better understand the contents of a dataset, how they can be used and what pre-processing is needed; (ii) *data filtering*, to remove redundant, unreliable or otherwise unwanted data; (iii) *data transformation*, e.g., to normalize or sanitize values or map between vocabularies; (iv) *inference materialization* and (v) *smushing*, i.e., the use of owl:sameAs links to detect URI aliases of the same entity and replace them with a "canonical" URI, to ease and speed-up data querying.

Although many of the tasks mentioned above have received considerable attention in the literature, tool support is limited and fragmented. Tools scaling to large datasets typically employ parallel, distributed computation models such as *MapReduce* (e.g., [14, 24, 25, 1]) and cannot be used efficiently—if not at all—on a single commodity machine. Tools targeting local computation, on the other hand, are often based on some form of data indexing (e.g., [19, 12, 24]), such as a *triple store* or a similar index [16]; these approaches typically scale as long as the index can be kept in main memory and incur a severe performance hit when data has to be accessed on disk. Only few tools can process large datasets locally, but they mainly target RDF syntax conversion [4, 7, 8, 5], with notable exceptions addressing RDF filtering [22], (possibly approximate) statistics extraction [11] and partial RDFS inference [2]. Interestingly, all these tools adopt a *streaming* processing model where data is processed one triple at a time, possibly in multiple passes.

Indeed, it is known from the literature that many processing tasks, including tasks on graph data like RDF, can be efficiently applied to large datasets on a single machine using the streaming model or one of its extensions [23], and in particular the extension where streaming is coupled with sorting [9]. This leads to an interesting research question:

> *Are relevant RDF processing tasks practically feasible on large datasets by using streaming and sorting techniques on a single commodity machine?*

Here, for "relevant" we intend the tasks previously listed and commonly associated to the processing of LOD data [17, 18]. For "large" we intend typical LOD dataset sizes, i.e., billions of triples. For "feasible", we mean the successful completion of tasks with an execution time that is "reasonable" if compared to competing single-machine approaches (where available), and to the time needed for consuming resulting data, which often consists in loading it in a production triple store where data can be queried by applications and users.

In this paper, we investigate the research question above by proposing implementations of the considered RDF pro-

.

cessing tasks based on streaming and sorting techniques. We call *processors* these implementations and collect them in an open source tool, called RDF_{PRO} (RDF Processor) [6], that allows composing processors in complex pipelines to efficiently perform multiple (parallel, sequential) computations on RDF data in one or more passes. We evaluate RDF_{PRO} in four broad and representative RDF processing scenarios—dataset *analysis*, *filtering*, *merging* and *massaging*[1]—using a commodity workstation and processing billions of triples of popular datasets (DBpedia, Freebase, GeoNames) whose contents and sizes are representative of the ones typically faced by applications dealing with LOD data. Experimental results allow us to positively answer the research question and show the practical applicability of our approach. Summing up, our contribution is threefold:

1. we address and answer the presented research question;
2. we present RDF_{PRO}, a general-purpose, open source tool that using sorting and streaming techniques is able to process billions of triples on a commodity machine;
3. we discuss four concrete RDF processing scenarios, showing how users can address them with RDF_{PRO}.

The paper is organized as follows. Section 2 describes RDF_{PRO}, presenting its processing model, processors and implementation. Section 3 evaluates the tool in the four scenarios, reporting experimental results, answering the research question and reporting relevant findings. Section 4 compares our approach with related works while Section 5 concludes.

2. RDF_{PRO} TOOL

This section describes how streaming and sorting are combined in RDF_{PRO}—the tool built to investigate our research question. We present RDF_{PRO} model in Section 2.1, its processors in Section 2.2 and its implementation in Section 2.3.

2.1 Processing model

The processing model of RDF_{PRO} is centered around the concept of *RDF processor*. A processor @P[2] (Figure 1a) is a software component that consumes an *input stream* of RDF quads—i.e., RDF triples with an *optional* fourth named graph component[3]—in *one or more passes*, produces an *output stream* of quads and may have an internal state as well as side effects like writing or uploading RDF data.

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. Specifically, a processor declares the number $n \geq 1$ of passes it needs, and may be asked to perform $m \geq n$ passes on its input (e.g., to support multiple downstream passes). In the first $n-1$ passes (if any), the processor reads the input and updates its state. At pass n the input is read again and output quads are emitted for the first time. In the next $m-n$ passes (if any), the processor is given again the input and must emit the same quads of pass n (the order may change).

Sorting is offered to processors as a primitive to arbitrarily sort selected data—possibly (a subset of) input quads—during a pass. Sorting is often combined to streaming in

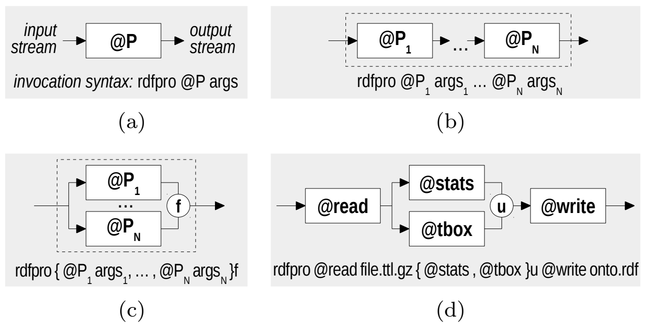

input stream
@P
output stream
invocation syntax: rdfpro @P args
(a)
@P1
...
@PN
rdfpro @P1 args1 ... @PN argsN
(b)
@P1
...
@PN
f
rdfpro { @P1 args1, ... , @PN argsN }f
(c)
@read
@stats
@tbox
u
@write
rdfpro @read file.ttl.gz { @stats , @tbox }u @write onto.rdf
(d)

Figure 1: Processor (a); sequential (b) & parallel (c) composition; example (d) – full syntax on web site.

the literature as it overcomes many of the limitations of a pure streaming model [9, 23]. In particular, it enables duplicates removal and set operations and provides the capability to group together information that may be scattered in the stream but must be processed together (e.g., all the quads about an entity when computing statistics). At the same time, most platforms provide an highly-optimized sorting utility that fully exploits available hardware resources: multiple CPU cores to parallelize the sorting algorithm, disk space to manage large datasets via (disk-based) *external sorting* and memory space to speed up processing.

Starting from the processors supplied with RDF_{PRO} (Section 2.2) or implemented by users, new *pipeline* processors can be derived by (recursively) applying sequential and parallel compositions. In a *sequential composition* (Figure 1b), two or more processors $@\mathsf{P}_\mathsf{i}$ are chained so that the output stream of $@\mathsf{P}_\mathsf{i}$ becomes the input stream of $@\mathsf{P}_{\mathsf{i}+1}$. In a *parallel composition* (Figure 1c), the input stream is sent concurrently to several processors $@\mathsf{P}_\mathsf{i}$, whose output streams are merged into a resulting stream using one of several possible *set operators* (specified with a flag `f` in figure and syntax): *union with duplicates* (flag `a`), *union without duplicates* (`u`), *intersection* (`i`) and *difference* (`d`) of quads from different branches. The number and orchestration of passes resulting from composition are automatically managed.

An example of composition is shown in Figure 1d, where a Turtle+gzip file (`file.ttl.gz`) is read, TBox and VOID [10] statistics are extracted in parallel and their union is written to an RDF/XML file (`onto.rdf`). Notably, I/O in the example do not use the input and output streams of the pipeline processor (dotted box in the figure), but relies on specific @write and @read processors whose side effects are dumping and augmenting the stream with the contents of external files. These I/O processors provide a lot of flexibility in how data is read and written, as they can be placed at any point of a 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 global input and output streams that are instead accessible when using RDF_{PRO} as a library).

2.2 Processors

In order to address the processing tasks considered in Section 1, we implemented the following processors in RDF_{PRO}:

@read Reads RDF file(s), emitting their quads together with the input stream; rewrites bnodes to avoid clashes.

@write Writes quads to one RDF file or splits them to multiple files evenly; quads are also propagated in output.

@download Sends a query to a SPARQL endpoint to down-

[1]Data *massaging* informally denotes all the ad-hoc transformation tasks necessary to make data better suited to a particular use (e.g., format conversion, value normalization).

[2]Processors are denoted by '@' in RDF_{PRO} syntax.

[3]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_{PRO} on plain triple data.

load quads that are emitted by augmenting the input stream. Both CONSTRUCT and SELECT queries are supported: the first can only return triples in the default graph; the latter produces bindings for specific variables `s`, `p`, `o`, `c` that are used to build output quads.

@upload Uploads quads from the input stream to an RDF store using SPARQL INSERT DATA calls, in chunks of a specified size; quads are also propagated in output.

@transform Processes each input quad with a user-supplied Groovy[4] script that can either discard the quad, propagate it or transform it into one or more output quads.

@smush Performs *smushing*, using a ranked namespace list to select canonical URIs that are linked in output to coreferring URIs (aliases) via owl:sameAs quads.

@infer Computes the RDFS deductive 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 same graph of the input quad.[5] Specific RDFS rules may be optionally 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.

@stats Emits VOID structural statistics for its input. A VOID dataset is associated to the whole input and to each *source* URI linked to named graphs in the data by a configurable property; 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 and examples for TBox terms, viewable in tools such as Protégé.

@unique Discards duplicates in the input stream. Optionally, it merges quads with same subject, predicate and object but different graphs in a unique quad. To track provenance, this quad is placed in a graph inheriting the descriptions of source graphs (i.e., the quads having them as subject) and representing their "fusion".

2.3 Implementation

RDF$_{\text{PRO}}$ is implemented in Java on top of the open source Sesame RDF library.[6] It consists of a *runtime* where multiple *processor implementations* can be plugged in, assembled using sequential and parallel composition and executed.

Runtime implementation The runtime defines the API of RDF processors and manages their lifecycle. Processors are Java classes extending `RDFProcessor` and declaring the number of passes they need. Each processor is attached to an input *quad queue* and an output *quad sink*. Input quads from the queue are "pushed" to the processor by invoking a callback method, using multiple threads from a common pool to process quads in parallel; other callbacks are invoked at the beginning and end of each pass to allow for initialization and completion of stateful computations. Output quads are emitted to the quad sink (a Sesame `RDFHandler`), with the runtime taking care of their downstream processing (if any). The design is inspired to the *Staged Event Driven Architecture* (SEDA) [27], with processors playing a passive role and all the queue and thread management handled by the runtime with the goal of maximizing CPU usage.

Within the runtime, streaming is embodied in the `RDFProcessor` API and in the management of input and output streams. Sorting, instead, is realized as a reusable primitive that can be invoked by the runtime and by processor implementations. This primitive is realized on top of the native, highly-optimized `sort` Unix utility, using *dictionary encoding techniques*[7] to compactly encode frequent RDF terms in sorted data, reducing its size (we measured ∼40 bytes per quad on real-world data) and improving execution times at the price of some memory consumption for the dictionary.

Processor composition is also managed by the runtime. Sequential composition and union with duplicates are computationally cheap, while the other forms of parallel composition are more expensive due to their use of sorting. In particular, intersection and difference are implemented by appending a label identifying the source branch to each quad sent to `sort`, and then gathering and checking all the labels of a sorted quad to decide if it can be emitted.

Processor implementation Due to their central role, the @read and @write processors feature multi-thread implementations aiming at transferring data as fast as possible to avoid I/O bottlenecks. Multiple RDF files can be parsed and written in parallel and, for line-oriented RDF formats, a single file can be split in newline-terminated chunks that are processed concurrently to increase the data throughput.

The @smush processor performs two passes: the first to extract the owl:sameAs graph which is kept in memory; the second to replace URIs based on detected equivalence classes. Efficient memory consumption is achieved with a specialized data structure that uses a custom hash table with an open addressing scheme to index URIs; table entries contain also a *next pointer* that organizes URIs of an owl:sameAs equivalence class in a circular linked list, which expands as new owl:sameAs quads are encountered and allows the structure to grow linearly with the number of URI aliases.

The @infer processor performs TBox inference using an in-memory, *semi-naive forward-chaining* algorithm [15]. ABox inference is done one quad at a time, using multiple threads and special deduplication logic and data structures for removing as many duplicate inferred quads as possible, so to avoid an artificial "explosion" of the number of output quads.

The @stats processor is implemented by sorting the input stream twice (simultaneously within a single pass): based on the subject, to group quads about the same entity and compute entity-based and distinct subjects statistics; and based on the object, to compute distinct objects statistics. Partial statistics are kept in memory during the processing.

[4] Groovy is a scripting language well integrated with Java and reusing its libraries. See http://groovy.codehaus.org/

[5] This scheme avoids expensive 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:subClassOf | rdfs:domain | rdfs:range | rdfs:subPropertyOf}
- X {rdf:type | rdfs:domain | rdfs:range | rdfs:subClassOf} {rdfs:Datatype | rdfs:ContainerMembershipProperty}

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

[7] We encode TBox URIs of known vocabularies (from `prefix.cc`) with integers. Namespaces and local names of other URIs are separately encoded until the encoding tables are full, after which they are emitted unchanged. For literals we encode the language and datatype tags, but not their labels.

3. EMPIRICAL EVALUATION

To answer our research question, we perform an empirical evaluation of RDF_{PRO} in four broad, relevant usage scenarios that exemplify the considered RDF processing tasks. In the first three scenarios—*dataset analysis* (Section 3.1), *filtering* (Section 3.2) and *merging* (Section 3.3)—we conduct practical experiments using a commodity workstation[8] and popular datasets (Freebase, DBpedia, GeoNames) whose contents and sizes are representative of the ones typically encountered by LOD applications. In the fourth scenario—*dataset massaging* (Section 3.4)—we categorize miscellaneous data massaging tasks that can be addressed with our approach and show its larger applicability; due to the simple processing involved we do not conduct experiments here. An extended description of the scenarios, including scripts for reproducing the experiments, is reported on RDFPRO web site [6].

3.1 Dataset Analysis

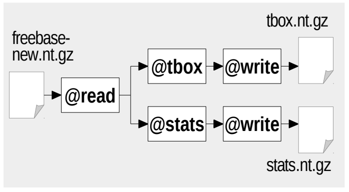
Dataset analysis comprises all the tasks aimed at providing a qualitative and quantitative characterization of the contents of an RDF dataset, such as the extraction of the data TBox or of instance-level ABox data statistics (e.g., VOID). When processing RDF, dataset analysis can be applied both to input and output data. In the first case, it helps identifying relevant data and required pre-processing tasks, especially when the dataset scope is broad (as occurs with many LOD datasets) or its documentation is poor. In the second case, it provides a characterization of output data that is useful for validation and documentation purposes.

Experiment As a representative example of large-scale dataset analysis, we consider the tasks of extracting TBox and VOID statistics from Freebase data (2014/09/10 dump, 2863 MQ – millions of quads), whose schema and statistics are not available online, and the task of comparing this Freebase release with a previous release (2014/07/10 dump, 2623 MQ) in order to identify newly added triples.[9]

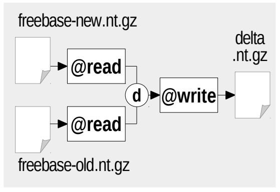
We use the @tbox and @stats processors to extract TBox and VOID statistics, invoked both separately and aggregated in a pipeline processor as shown in Figure 2a. To extract new triples, we read both dataset releases and use parallel composition with the *difference* set operator (Section 2.1) to combine quads, as shown in Figure 2b.

Table 2c reports the tasks execution times, throughputs, input and output sizes both in quads and compressed (gzip) bytes as measured on our test machine. Additionally, when running the comparison task we measured a disk usage of 92.8 GB for the temporary files produced by the sorting-based *difference* set operator (∼18 bytes per input triple).

Comment Comparing the two Freebase releases resulted the most expensive task due to sorting and involved input size. When performed jointly, TBox and statistics extraction present performance figures close to statistics extraction alone, as data parsing is performed once and the cost of TBox extraction (excluded parsing) is negligible. This is an example of how the aggregation of multiple processing tasks in a single computation, enabled by RDFPRO streaming model and composition facilities, can generally lead to better performances due to a reduction of I/O overheads.



(a)



(b)

Task	Input [MQ]	Input [MB]	Output [MQ]	Output [MB]	Throughput [MQ/s]	Throughput [MB/s]	Time [s]
1. TBox	2863	28339	0.23	3.01	1.43	14.12	2006
2. Statistics	2863	28339	0.13	1.36	0.34	3.36	8443
1-2 Aggregated	2863	28339	0.36	4.35	0.34	3.36	8426
3. Comparison	5486	55093	260	1894	0.42	4.25	12955

(c)

Figure 2: Dataset analysis flows (a, b) & results (c).

To provide an idea of the how analysis results can be used to explore the dataset, Figure 3 shows the joint browsing of TBox and statistics in Protégé, exploiting the specific concept annotations emitted by @stats. The class and property hierarchies are augmented with the number of entities and property triples (marked as 1 in the figure), as well as with the detected property usage (2), e.g., O for object property, I for inverse functional; each concept is annotated with an example instance and a VOID partition individual (3), which provides numeric statistics about the concept (4).

3.2 Dataset Filtering

When dealing with large RDF datasets, *dataset filtering* (or *slicing* [22]) is often required to extract a small subset of interesting data, identified, e.g., based on a previous dataset analysis (Section 3.1). Dataset filtering typically consists in (i) identifying the entities of interest in the dataset, based on selection conditions on their URIs, rdf:type or other properties; and (ii) extracting all the quads about these entities expressing selected RDF properties. These two operations can be implemented using multiple streaming passes.

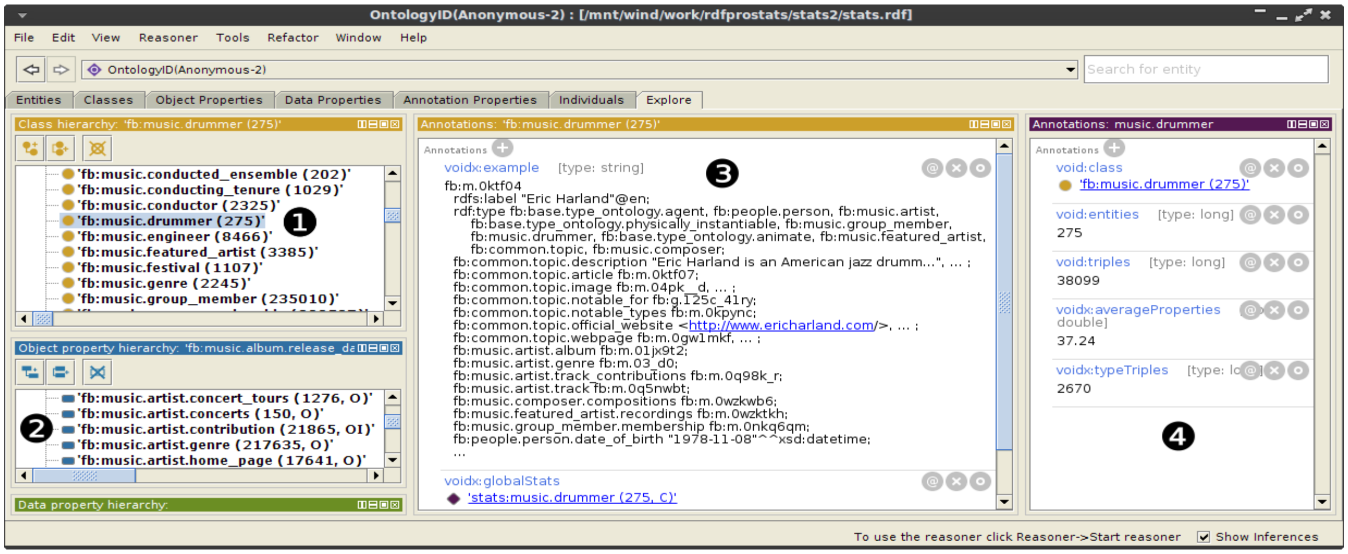
Experiment We consider a concrete dataset filtering example where the dataset is Freebase (2014/09/10 dump, 2863 MQ – millions of quads), the entities of interest are musical groups (i.e., their rdf:type is fb:music.musical_group) that are still active (i.e., there is no associated property fb:music.artist.active_end), and the properties to extract are the group name, genre and place of origin (respectively, rdfs:label, fb:music.artist.genre and fb:music.artist.origin).

We implement the task with two invocations of RDFPRO as in Figure 4a. The first invocation (marked as 1) generates an RDF file listing as subjects the URIs of the entities of interest; this is done with two parallel @transform processors, extracting respectively musical groups and no more active musical entities, whose outputs are combined using the *difference* set operator. The second invocation (marked as 2) uses another @transform processor to extract desired quads, testing predicates and requiring subjects to be contained in the previously extracted file (whose URIs are indexed in memory by a specific function in the @transform expression).

Table 4b reports the execution times, throughputs, input and output sizes of the two invocations on the test machine.

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

[9]From this delta TBox and VOID statistics can be extracted to get a concise summary of what has been added. This analysis is analogous to (and computationally cheaper than) the one done on the whole Freebase and is thus omitted.


OntologyID(Anonymous-2) : [/mnt/wind/work/rdfprostats/stats2/stats.rdf]

File Edit View Reasoner Tools Refactor Window Help

OntologyID(Anonymous-2)

Search for entity

Entities | Classes | Object Properties | Data Properties | Annotation Properties | Individuals | Explore

Class hierarchy: 'fb:music.drummer (275)'

- 'fb:music.conducted_ensemble (202)'
- 'fb:music.conducting_tenure (1029)'
- 'fb:music.conductor (2325)'
- 'fb:music.drummer (275)' ❶
- 'fb:music.engineer (8466)'
- 'fb:music.featured_artist (3385)'
- 'fb:music.festival (1107)'
- 'fb:music.genre (2245)'
- 'fb:music.group_member (235010)'

Object property hierarchy: 'fb:music.album.release_da

❷
- 'fb:music.artist.concert_tours (1276, O)'
- 'fb:music.artist.concerts (150, O)'
- 'fb:music.artist.contribution (21865, OI)'
- 'fb:music.artist.genre (217635, O)'
- 'fb:music.artist.home_page (17641, O)'

Data property hierarchy:

Annotations: 'fb:music.drummer (275)'

Annotations

voidx:example [type: string] ❸

```
fb:m.0ktf04
  rdfs:label "Eric Harland"@en;
  rdf:type fb:base.type_ontology.agent, fb:people.person, fb:music.artist,
     fb:base.type_ontology.physically_instantiable, fb:music.group_member,
     fb:music.drummer, fb:base.type_ontology.animate, fb:music.featured_artist,
     fb:common.topic, fb:music.composer;
  fb:common.topic.description "Eric Harland is an American jazz drumm...", ... ;
  fb:common.topic.article fb:m.0ktf07;
  fb:common.topic.image fb:m.04pk__d, ... ;
  fb:common.topic.notable_for fb:g.125c_41ry;
  fb:common.topic.notable_types fb:m.0kpync;
  fb:common.topic.official_website <http://www.ericharland.com/>, ... ;
  fb:common.topic.webpage fb:m.0gw1mkf, ... ;
  fb:music.artist.album fb:m.01jx9t2;
  fb:music.artist.genre fb:m.03_d0;
  fb:music.artist.track_contributions fb:m.0q98k_r;
  fb:music.artist.track fb:m.0q5nwbt;
  fb:music.composer.compositions fb:m.0wzkwb6;
  fb:music.featured_artist.recordings fb:m.0wzktkh;
  fb:music.group_member.membership fb:m.0nkq6qm;
  fb:people.person.date_of_birth "1978-11-08"^^xsd:datetime;
  ...
```

voidx:globalStats
◆ 'stats:music.drummer (275, C)'

Annotations: music.drummer

Annotations

void:class
● 'fb:music.drummer (275)'

void:entities [type: long]
275

void:triples [type: long]
38099

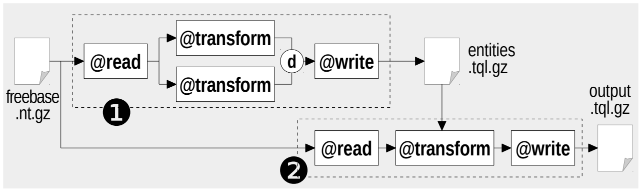
voidx:averageProperties double]
37.24

voidx:typeTriples [type: lo
2670

❹

To use the reasoner click Reasoner->Start reasoner ✓ Show Inferences

Figure 3: Visualization of dataset TBox and statistics in Protégé.



(a)

Task	Input [MQ]	Input [MB]	Output [MQ]	Output [MB]	Throughput [MQ/s]	Throughput [MB/s]	Time [s]
1. Select entities	2863	28339	0.20	0.73	1.36	13.4	2111
2. Extract quads	2863	28339	0.42	5.17	1.15	11.4	2481

(b)

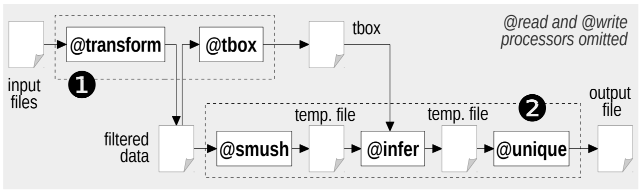
Figure 4: Dataset filtering flow (a) and results (b).

Comment Although simple, the experiment shows how practical, large-scale filtering tasks are feasible using the streaming and sorting approach of RDFpro. Performances are worse than the ones obtainable using SPARQL in a triple store, but competitive if one considers also the time needed for indexing data in the triple store (see Section 3.5).

More complex filtering scenarios can be addressed using set operations for implementing conjunction, disjunction and negation of selection conditions, and with additional invocations of RDFpro that progressively augment the result (e.g., a third invocation can identify albums of selected artists, while a fourth invocation can extract the quads describing them). In cases where RDFpro model is insufficient (e.g., due to the need for aggregations or joins), the tool can still be used to perform a first coarse-grained filtering that reduces the number of quads and eases their downstream processing.

3.3 Dataset Merging

A common usage scenario is *dataset merging*, where multiple RDF datasets are integrated and prepared for application consumption. Data preparation typically comprises smushing, inference materialization and data deduplication (possibly with provenance tracking). These tasks make the use of the resulting dataset more easy and efficient, as reasoning and entity aliasing have been already accounted for.



(a)

Step	Input [MQ]	Input [MB]	Output [MQ]	Output [MB]	Throughput [MQ/s]	Throughput [MB/s]	Time [s]
1. Transform	3394	33524	3394	36903	0.42	4.12	8137
2. TBox extract.	3394	36903	<1	4	1.28	13.9	2656
3. Smushing	3394	36903	3424	38823	0.37	3.98	9265
4. Inference	3424	38823	5615	51927	0.32	3.66	10612
5. Deduplication	5615	51927	4085	31297	0.33	3.03	17133
1-2 Aggregated	3394	33524	3394	36903	0.41	4.06	8247
3-5 Aggregated	3394	36903	4085	31446	0.14	1.56	23734

(b)

Figure 5: Dataset merging flow (a) and results (b).

Experiment We consider a concrete dataset merging scenario with data from Freebase (2014/09/10 dump, 2863 MQ – millions quads), GeoNames (2013/08/27 dump, 125 MQ) and DBpedia in the four languages EN, ES, IT and NL (version 3.9, 406 MQ without redirects, disambiguation, pages and revisions metadata), for a total of 3394 MQ.

Figure 5a shows the required processing steps. A preliminary processing phase (marked as 1) is required to transform input data and extract the TBox axioms required for inference. Data transformation serves (i) to track provenance, by placing quads in different named graphs based on the source dataset; and (ii) to adopt optimal serialization format (Turtle Quads) and compression scheme (gzip) that speed up further processing. The main processing phase (marked as 2) consists in the cascaded execution of smushing, RDFS inference and deduplication to produce the merged dataset. Smushing identifies owl:sameAs equivalence classes and assigns a canonical URI to each of them. RDFS inference excludes rules *rdfs4a*, *rdfs4b* and *rdfs8* to avoid materializing uninformative ⟨ X rdf:type rdfs:Resource ⟩ quads. Deduplication takes quads with the same subject, predicate and object (possibly produced by previous steps) and merges them in a single quad inside a graph linked to all the original sources.

Table 5b reports the execution times, throughputs and input and output sizes of each step, covering both the cases where steps are performed separately via intermediate files and multiple invocations of RDF$_{\text{PRO}}$ (upper part of the table), or aggregated per processing phase using composition capabilities (lower part). Additionally, RDF$_{\text{PRO}}$ reported the use of ∼2 GB of memory for smushing an owl:sameAs graph of ∼38M URIs and ∼8M equivalence classes (∼56 bytes/URI).

Comment Also in this scenario, the aggregation of multiple processing tasks leads to a marked reduction of the total processing time (33% reduction from 47803 s to 31981 s) due to the elimination of the I/O overhead for intermediate files.

While addressed separately, the three scenarios of dataset analysis, filtering and merging are often combined in practice, e.g., to remove unwanted ABox and TBox quads from input data, merge remaining quads and analyze the result producing statistics that describe and document it; an example of such combination is reported on RDF$_{\text{PRO}}$ web site [6].

3.4 Dataset Massaging

We categorize three relevant, broad classes of *dataset massaging* tasks that are supported by RDF$_{\text{PRO}}$ processing model: data repackaging, data sanitization and data derivation.

Data repackaging comprises all the modifications that preserve data contents, i.e., the quads, and just affect the way data is packaged, i.e., the choices of RDF syntax, compression scheme and number of files. These modifications are often necessary to comply with restrictions of existing tools and systems, or to distribute data in a form that is best suited to the intended use (e.g., machine vs human consumption). Data repackaging operations are all supported by RDF$_{\text{PRO}}$ and are best performed in a streaming model, which thus represent the most common choice for this task.

Data sanitization consists in fixing or removing the RDF terms or quads that prevent further processing of data. An example consists in the conversion of literals of a datatype to literals of an alternative datatype, because the former is not (properly) supported by the target system.[10] Other tasks supported in a streaming model include the rewriting of URIs (e.g., to change namespace), the normalization of literals (e.g., to ensure that rdfs:label strings obey certain capitalization patterns) and the removal of quads whose literal object has an excessive length.[11] These and similar tasks are supported by the RDF$_{\text{PRO}}$ @transform processor.

Data derivation consists in augmenting a dataset with quads computed from original data. Two broad classes of derivations supported in a streaming model are (i) quad-level derivations, and (ii) aggregations of quad-level information with emission of aggregate results at the end. Examples of the first kind include the conversion of a numeric value from a unit of measurement to another, as done in DBpedia "Mapping-based Properties (Specific)" dataset, or the computation of the age of persons starting from their birth dates. Examples of the second kind include counting the occurrences of a certain property for an entity (e.g., the number of person he/she foaf:knows). All these derivations are supported by @transform, while more complex derivations (e.g., involving joins) may in principle be implemented in new processors by exploiting the sorting primitive.

While we do not conduct experiments here, we note that the tasks described can be all implemented in a single pass without sorting. Assuming similar input and output sizes, performances roughly amounts to the ones of reading and writing data in a pass (∼0.4-0.5 MQ/s on the test machine).

3.5 Discussion

The experimental results and the applicability of RDF$_{\text{PRO}}$ in relevant scenarios allow to answer our research question and provide interesting findings on the use of our approach.

Research question assessment Two results emerge from the experiments: (i) RDF$_{\text{PRO}}$ implementation of the processing tasks of Section 1 succeeds in managing billions of RDF triples on a commodity machine; and (ii) execution times are in the order of hours (1h 16' for filtering 2.86 BQ, 5h 56' for analyzing 5.49 BQ, 8h 53' for merging 3.39 BQ).

The first result, which is trivial for tasks inherently expressible at a quad-level such as TBox extraction and some kinds of filtering, is not obvious for other tasks such as RDFS inference, smushing, statistics extraction, deduplication and set operations, for which we provide specialized implementations based on a mix of streaming and sorting techniques.

The second result can be put into perspective by comparing it with the time needed to load data in a triple store. On Virtuoso 7, a state-of-the-art triple store, the load time for one billion of quads is 9h 08' on our test machine and 27' on the very powerful machine used in the latest Berlin SPARQL Benchmark (BSBM) experiment.[12] Assuming that these rates hold for larger amounts of data, the comparison between these times and our processing times leads to two conclusions. First, given the same hardware, any *one-time* processing based on the use of a triple store—a common approach to RDF processing—is not competitive with our approach, as just the loading of input data would take longer than our processing time in the considered scenarios.[13] Second, our processing times are negligible if compared to load times on the same machine, and have the same order of magnitude of load times in the BSBM machine, overall meaning that RDF processing based on RDF$_{\text{PRO}}$ approach would not slow down (and is thus compatible with) a typical *Extract, Transform, Load* (ETL) procedure where resulting RDF data is put in a production triple store.

Based on these results, a positive answer can be given to the research question *"Are relevant RDF processing tasks practically feasible on large datasets by using streaming and sorting techniques on a single commodity machine?"*.

Other findings The empirical evaluation also highlights the importance of task aggregation and allows us to analyze the factors impacting streaming and sorting performances.

Aggregation of multiple processing tasks in a single RDF$_{\text{PRO}}$ invocation provides better performances as input data is parsed once and I/O costs for accessing intermediate files

[10]E.g., the Community Edition of Virtuoso 7.1 normalizes xsd:gDay, xsd:gMonth and xsd:gYear values to xsd:datetime altering their semantics; changing to xsd:int is a partial fix.

[11]E.g., the OWLIM Lite 5.4.6486 triple store cannot store very long literals (e.g., 20M chars of GML geographic data).

[12]2x Intel Xeon E5-2650 CPU (8 cores, hyper-threading), 256 GB RAM, 3x 1.8 TB 7200 RPM disks RAID 0, Linux 3.3.4; see http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/index.html

[13]Of course, using a triple store may pay off in scenarios where data is loaded once and processed many times, or when the triple store is also the final destination of data.

are eliminated, as shown in Section 3.1 and 3.3. Task aggregation requires composition primitives and the support for reading and writing data at any point of the pipeline, two features of RDF$_{\text{PRO}}$ that are relevant to any similar tool.

Streaming performances within a pipeline are highly dependent on the file compression and RDF format used. No compression and best compression (e.g., `bzip2` used in DBpedia dumps) are inefficient, with `gzip` representing a good trade-off; using native compression utilities (and especially their parallel versions, e.g., `pigz` and `pbzip2`) is also beneficial. Line-oriented RDF formats such as NTriples and NQuads provide better performances as they allow multi-thread parsing and serialization (e.g., from 0.61 MQ/s to 1.45 MQ/s for Freebase NTriples+gzip data with multiple threads). We also experimented with the HDT binary format [16], but writing HDT is very expensive while reading HDT is not faster than reading other formats (unless lookup of RDF terms in the HDT dictionary is skipped).

Sorting performances depend on a number of factors. In our experience, performances can be improved by allocating a large amount of memory for sorting (we gave 8 GB out of the available 16 GB to the `sort` utility in our experiments), by using a parallel sort implementation and by configuring the compression of temporary files. Dictionary encoding of frequent RDF terms also helps to improve throughput.

4. RELATED WORK

Most RDF data (perhaps the most relevant) is available as LOD data. An in-depth account of the LOD initiative, including a reference architecture for LOD data consumption, is presented in [17], while [18] reports the findings of a survey on 124 LOD applications (2003-2009). Our selection of RDF processing tasks, usage scenario and evaluation criteria for processing times can be justified based on these works. Concerning the processing tasks, data filtering, transformation and smushing are at the core of the *vocabulary mapping*, *identity resolution* and *quality evaluation* modules in the reference architecture of [17]. Inference—optional task in the architecture—is relevant to 58% of the applications surveyed in [18], while VOID [10] statistics extraction is relevant due to VOID being acknowledged as the standard tool to describe LOD datasets [17]. Concerning the usage scenarios, we already noted that dataset analysis, filtering and merging are part of a larger integration workflow that is the *raison d'être* of the reference architecture and is a required activity in 80% of the surveyed application. Finally, our comparison of processing times to triple store loading times is grounded in the use of triple stores in 88% of the surveyed applications.

All the processing tasks we considered can be implemented on a single machine using some kind of data index—typically a triple store but also an RDF file indexed in the *HDT* format [16]—and then exploiting the index querying and manipulation primitives—typically SPARQL. Data filtering and transformation are straightforward in this setting and smushing is not complex to implement. Statistics extraction can use queries, as done using SPARQL in *make-void* [3] to compute VOID statistics, and in *RDFStats* [19] to compute instance counts and class and property histograms. Finally, inference is possible using forward-chaining algorithms, as done in the *OWLIM* triple store [12] for RDFS and various OWL fragments. While supporting flexible processing and many RDF processing tools, the data index must reside entirely in memory for these tools to operate efficiently—if at all—with performances decreasing quickly if disk access is involved, both for creating the index and accessing it during processing. In comparison, a streaming approach also hits the disk, but data access is sequential and faster, leading to situations where total processing time with streaming is faster than index creation alone, as reported in Section 3.5.

Scalability with respect to data size is generally achieved using distribution. In particular, the MapReduce paradigm and its Hadoop[14] implementation have been used for forward-chaining RDFS and OWL Horst inference in *WebPIE* [25], for VOID statistics extraction in *voidGen* [14] and for general RDF processing (with special focus on Freebase data) in *Infovore* [1]. Although a MapReduce engine can be deployed on a single machine, its natural execution environment is a cluster of many machines where coordination costs and the overhead associated to its distributed nature are negligible.

A hybrid tool using either a local triple store or a Hadoop cluster is the *Linked Data Integration Framework* (LDIF) [24]. LDIF implements the architecture of [17] and, in particular, data filtering based on simple rules, data transformation for vocabulary mapping with *R2R* rules [13] and smushing and entity resolution with *Silk* [26]. LDIF partitions data in per-entity graphs processed by distinct threads and machines. This approach scales to hundreds of millions of triples on single machines, and to billions of triples on Hadoop clusters. Its downsides are that computations spanning multiple entities may be infeasible and data partitioning is not for free (especially with a triple store), which could be the limit to LDIF scalability on single machines.

The streaming computation model has been studied for a long time. A good survey of streaming algorithms on graph data such as RDF is [23]. Here, different extensions to the base streaming model are analyzed and the one with a sorting primitive, formalized in [9], results the most expressive. Sorting is an highly-efficient operation on today machines and allows to perform set operations and to group together information scattered in the stream that must be processed together (e.g., quads of an entity), in this way overcoming many of the limitations of a pure streaming model.

Our use of streaming for RDF processing differentiates from *Stream Reasoning* [21], as we focus on processing large but *finite* amounts of data using a streaming computation model, whereas a Stream Reasoning engine deals with the processing of possibly *time-windowed* and *infinite* streams of temporally-tagged triples; nevertheless, such an engine can be included in our framework as a specialized processor. Tools using a streaming model for RDF processing like us are often limited to syntax conversion, e.g., *rapper* [4], *rdfpipe* [7], *Sesame RDFConvert* [8] and *DotNetRDF rdfConvert* [5]. The few exceptions include Jena riot, LODStats and SliceSPARQL. *Jena riot* [2] supports rdfs:domain, rdfs:range, rdfs:subClassOf and rdfs:subPropertyOf inference after preliminary indexing of TBox data. This approach is similar to ours, although we cover complete RDFS inference with very loose ABox restrictions. *LODStats* [11] extracts 32 statistics (a superset of VOID) in a single streaming pass, but relies heavily on in-memory data structures whose content is dropped when full, leading to approximate statistics computation. In comparison, our implementation employs sorting and lightweight data structures (size linear in number of classes and properties, rather than entities). *Slice-*

[14]http://hadoop.apache.org/

SPARQL [22] is a SPARQL fragment for data filtering. It includes UNION, FILTER and graph patterns with one join variable, and can be evaluated in (max) three streaming passes with an auxiliary disk index for join candidates. Our @transform processor supports the same filtering, but passes and join candidates must be explicitly managed by users.

We conclude noting that our sequential and parallel composition mechanisms are in line with the workflow primitives for RDF processing studied in the literature, especially in the scope of semantic mash-ups. Here, *Semantic Web Pipes* (SWP) [20] allow composing processing operators in a DAG similarly to us, although their focus is on easy visual composition rather than data streaming and large-scale processing.

5. CONCLUSIONS

With RDF$_{\text{PRO}}$, we showed that many RDF processing tasks can be performed on billions of triples on a single machine using streaming and sorting, with execution times compatible with batch RDF processing and better than the times of approaches using a local data index (e.g., a triple store).

Of course, tasks such as OWL 2 inference, SPARQL query answering and SPARQL-based data massaging are not supported in RDF$_{\text{PRO}}$. These tasks involve joining of data and are thus hard to implement and possibly inefficient in a streaming and sorting setting, although specialized algorithms might be devised to implement restricted versions of them (a direction we would like to investigate).

Even with these limitations, we consider RDF$_{\text{PRO}}$ a practically useful tool. RDF$_{\text{PRO}}$ processors and composition facilities allow addressing a variety of processing needs with a single tool that can be used by casual users and not just by developers, making it a sort of "swiss-army-knife" for exploring and manipulating RDF datasets. At the same time, RDF$_{\text{PRO}}$ can be extended by developers with new processors by just focusing on the specific task at hand, as efficient streaming, sorting, I/O, thread management and composition facilities are already provided by RDF$_{\text{PRO}}$ runtime.

RDF$_{\text{PRO}}$ is actively used in the NewsReader project to process generated RDF data and build background knowledge datasets out of multi-lingual LOD data sources. We released RDF$_{\text{PRO}}$ source code in the public domain, and we plan to extend its capabilities in follow-ups of this work.

Acknowledgements The research leading to this paper was supported by the European Union's 7th Framework Programme via the NewsReader Project (ICT-316404).

6. REFERENCES

[1] Infovore. https://github.com/paulhoule/infovore.

[2] Jena riot. https://jena.apache.org/documentation/io/.

[3] make-void. https://github.com/cygri/make-void.

[4] rapper. http://librdf.org/raptor/rapper.html.

[5] rdfConvert. https://bitbucket.org/dotnetrdf/dotnetrdf/wiki/UserGuide/Tools/rdfConvert.

[6] RDF$_{\text{PRO}}$. http://fracor.bitbucket.org/rdfpro/.

[7] rdfpipe. http://rdfextras.readthedocs.org/en/latest/tools/rdfpipe.html.

[8] Sesame RDFConverter. http://sourceforge.net/projects/rdfconvert.

[9] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 540–549, 2004.

[10] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets. In *Workshop on Linked Data on the Web (LDOW)*, 2009.

[11] S. Auer, J. Demter, M. Martin, and J. Lehmann. LODStats - an extensible framework for high-performance dataset analytics. In *EKAW*, pages 353–362, 2012.

[12] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. OWLIM: A family of scalable semantic repositories. *Semant. Web*, 2(1):33–42, 2011.

[13] C. Bizer and A. Schultz. The R2R framework: Publishing and discovering mappings on the Web. In *Int. Workshop on Consuming Linked Data (COLD)*, 2010.

[14] C. Böhm, J. Lorey, and F. Naumann. Creating voiD descriptions for Web-scale data. *Web Semant.*, 9(3):339–345, Sept. 2011.

[15] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Knowl. Data Eng.*, 1(1):146–166, 1989.

[16] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF representation for publication and exchange (HDT). *Web Semant.*, 19:22–41, 2013.

[17] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.

[18] B. Heitmann, R. Cyganiak, C. Hayes, and S. Decker. Architecture of Linked Data applications. In *Linked Data Management: Principles and Techniques*. CRC Press, 2013.

[19] A. Langegger and W. Woss. RDFStats - an extensible RDF statistics generator and library. In *Int. Workshop on Database and Expert Systems Application, DEXA'09*, pages 79–83, 2009.

[20] D. Le-Phuoc, A. Polleres, M. Hauswirth, G. Tummarello, and C. Morbidoni. Rapid prototyping of semantic mash-ups through Semantic Web Pipes. In *WWW*, pages 581–590, 2009.

[21] A. Margara, J. Urbani, F. van Harmelen, and H. Bal. Streaming the Web: Reasoning over dynamic data. *Web Semant.*, 25(0):24 – 44, 2014.

[22] E. Marx, S. Shekarpour, S. Auer, and A.-C. Ngomo. Large-scale RDF dataset slicing. In *IEEE Int. Conf. on Semantic Computing (ICSC)*, pages 228–235, 2013.

[23] T. O'Connell. A survey of graph algorithms under extended streaming models of computation. In *Fundamental Problems in Computing*, pages 455–476. Springer Netherlands, 2009.

[24] A. Schultz, A. Matteini, R. Isele, P. N. Mendes, C. Bizer, and C. Becker. LDIF - a framework for large-scale Linked Data integration. In *WWW Developers Track*, 2012.

[25] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal. WebPIE: A web-scale parallel inference engine using MapReduce. *J. Web Semant*, 10:59–75, 2012.

[26] J. Volz, C. Bizer, M. Gaedke, and G. Kobilarov. Silk - A link discovery framework for the Web of Data. In *Workshop on Linked Data on the Web (LDOW)*, 2009.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.