Efficient RDF Interchange (ERI) Format for RDF Data Streams

Javier D. Fernández, Alejandro Llaves, Oscar Corcho

Ontology Engineering Group (OEG), Univ. Politécnica de Madrid (Spain)
{jdfernandez,allaves,ocorcho}@fi.upm.es

Abstract. RDF streams are sequences of timestamped RDF statements or graphs, which can be generated by several types of data sources (sensors, social networks, etc.). They may provide data at high volumes and rates, and be consumed by applications that require real-time responses. Hence it is important to publish and interchange them efficiently. In this paper, we exploit a key feature of RDF data streams, which is the regularity of their structure and data values, proposing a compressed, efficient RDF interchange (ERI) format, which can reduce the amount of data transmitted when processing RDF streams. Our experimental evaluation shows that our format produces state-of-the-art streaming compression, remaining efficient in performance.

1 Introduction

Most of the largest RDF datasets available so far (*e.g.* Bio2RDF,[1] LinkedGeoData,[2] DBpedia[3]) are released as static snapshots of data coming from one or several data sources, generated with some ETL (Extract-Transform-Load) processes according to scheduled periodical releases. That is, data are mostly static, even when they contain temporal references (*e.g.* the Linked Sensor Data dataset, which contains an historical archive of data measured by environmental sensors). Typical applications that make use of such datasets include those performing simulations or those training numerical models.

In contrast, some applications only require access to the most recent data, or combine real-time and historical data for different purposes. In these cases a different approach has to be followed for RDF data management, and RDF streams come into play. RDF streams are defined as potentially unbounded sequences of time varying RDF statements or graphs, which may be generated from any type of data stream, from social networks to environmental sensors.

Several research areas have emerged around RDF streams, *e.g.* temporal representation in RDF [12, 6, 11], or RDF stream query languages and processing engines (C-SPARQL [2], SPARQLStream and morph-streams [4], CQELS Cloud [15], Ztreamy [1]). The recently-created W3C community group on RDF Stream

[1] http://bio2rdf.org/.
[2] http://linkedgeodata.org/.
[3] http://www.dbpedia.org/.

Processing is working on the provision of "a common model for producing, transmitting and continuously querying RDF Streams".[4]

In this paper, we focus on the efficient transmission of RDF streams, a necessary step to ensure higher throughput for RDF Stream processors. Previous work on RDF compression [9, 14] shows important size reductions of large RDF datasets, hence enabling an efficient RDF exchange. However, these solutions consider static RDF datasets, and need to read the whole dataset to take advantage of data regularities. A recent proposal, RDSZ [10], shows the benefits of applying the general-purpose stream compressor Zlib [8] to RDF streams, and provides a compression algorithm based on the difference of subject groups (provided in Turtle [17]), with some gains in compression (up to 31% *w.r.t.* Zlib).

Our work sets on the basis of RDSZ and exploits the fact that in most RDF streams the structural information is well-known by the data provider, and the number of variations in the structure are limited. For instance, the information provided by a sensor network is restricted to the number of different measured properties, and in an RDF context the SSN ontology [5] will be probably used for representing such data. Furthermore, given that "regularities" are also present in very structured static datasets (*e.g.* statistical data using the RDF Data Cube Vocabulary [7]), our approach may be also applicable to those datasets. Thus, our **preliminary hypothesis** states that our proposed RDF interchange format can optimize the space and time required for representing, exchanging, and parsing RDF data streams and regularly-structured static RDF datasets.

In this paper, we propose a complete efficient RDF interchange (`ERI`) format for RDF streams. `ERI` considers an RDF stream as a continuous flow of blocks (with predefined maximum size) of triples. Each block is modularized into two main sets of channels to achieve large spatial savings:

- Structural channels: They encode the subjects in each block and, for each one, the structural properties of the related triples, using a dynamic dictionary of structures.
- Value channels: They encode the concrete data values held by each predicate in the block in a compact fashion.

We also provide a first practical implementation with some decisions regarding the specific compression used in each channel. An empirical evaluation over a heterogeneous corpora of RDF streaming datasets shows that `ERI` produces state-of-the-art compression, remaining competitive in processing time. Similar conclusions can be drawn for very regular datasets (such as statistical data) and general datasets in which the information is strongly structured.

Our main **contributions** are: (i) the design of an efficient RDF interchange (`ERI`) format as a flexible, modular and extensible representation of RDF streams; (ii) a practical implementation for `ERI` which can be tuned to cope with specific dataset regularities; and (iii) an evaluation that shows our gains in compactness *w.r.t.* current compressors, with low processing overheads.

The rest of the paper is organized as follows. Section 2 reviews basic foundations of RDF streaming and compression. Our approach for efficient RDF

[4] http://www.w3.org/community/rsp/.

interchange (ERI) is presented in Section 3, as well as a practical deployment for ERI encoding and decoding. Section 4 provides an empirical evaluation analyzing compactness and processing efficiency. Finally, Section 5 concludes and devises future work and application scenarios.

2 Background and Related Work

A key challenge for stream processing systems is the ability to consume large volumes of data with varying and potentially large input rates. Distributed stream processing systems are a possible architectural solution. In these systems, the circulation of data between nodes takes an amount of time that depends on parameters like data size, network bandwidth, or network usage, among others. Hence it is crucial to minimize data transmission time among processing nodes.

To reach this goal, our work focuses on RDF stream compression techniques. RDF compression is an alternative to standard compression such as gzip. It leverages the skewed structure of RDF graphs to get large spatial savings. The most prominent approach is HDT [9], a binary format that splits and succinctly represents an RDF dataset with two main components: the *Dictionary* assigns an identifier (ID) to all terms in the RDF graph with high levels of compression, and the *Triples* uses the previous mapping and encodes the pure structure of the underlying RDF graph. HDT achieves good compression figures while providing retrieving features to the compressed data [9]. However, these are at the cost of processing the complete dataset and spending non-negligible processing time. The same applies to other recent RDF compression approaches based on inferring a grammar generating the data [14] or providing other dictionary-based compression on top of MapReduce [19].

Streaming HDT [13] is a deviation from HDT that simplifies the associated metadata and restricts the range of available dictionary IDs. Thus, the scheme is a simple dictionary-based replacement which does not compete in compression but allows operating in constrained devices. RDSZ [10] is the first specific approach for RDF streaming compression. RDSZ takes advantage of the fact that items in an RDF stream usually follow a common schema and, thus, have structural similarities. Hence it uses differential encoding to take advantage of these similarities, and the results of this process are compressed with Zlib to exploit additional redundancies. Experiments show that RDSZ produces gains in compression (17% on average) at the cost of increasing the processing time.

The increasing interest on RDF compression over streaming data has also been recently highlighted by RDF stream processing systems such as CQELS Cloud [15] and Ztreamy [1]. The first one uses a basic dictionary-based approach to process and move fixed-size integers between nodes. The latter exploits the Zlib compressor with similar purposes. In addition, it is also relevant to detect trends in data, extract statistics, or compare historic data with current data to identify anomalies, although historical data management is not considered in most of stream processing systems [16]. A potential use case of RDF compression may be the integration of historical data and real-time data streams.

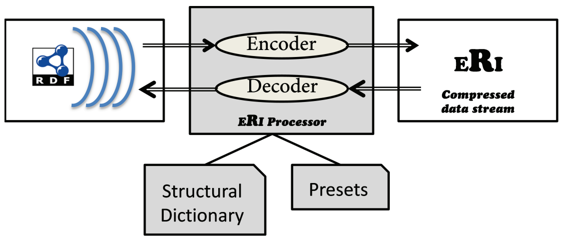


Fig. 1. ERI processing model.

3 Efficient RDF Interchange (ERI) Format

The ERI format is a compact RDF representation designed to leverage the inherent structural and data redundancy in RDF streams. In the following, we introduce the basic concepts behind the ERI format, and we present a practical implementation for encoding and decoding RDF data.

3.1 Basic Concepts

In ERI, we consider the generic processing model depicted in Figure 1. In this scheme, RDF data, potentially in the form of a data stream, is encoded or decoded to ERI, resulting in a compressed data stream. We refer to an **ERI processor** as any application able to encode such RDF data to ERI or to decode the ERI compressed stream (defined below) to make the RDF data accessible. A processor mainly leverages on two information sets to improve compactness: (i) the Structural Dictionary and (ii) the Presets, defined as follows.

The **Structural Dictionary** holds a dynamic catalog of all different structural patterns found for a given set of triples called **Molecules**.

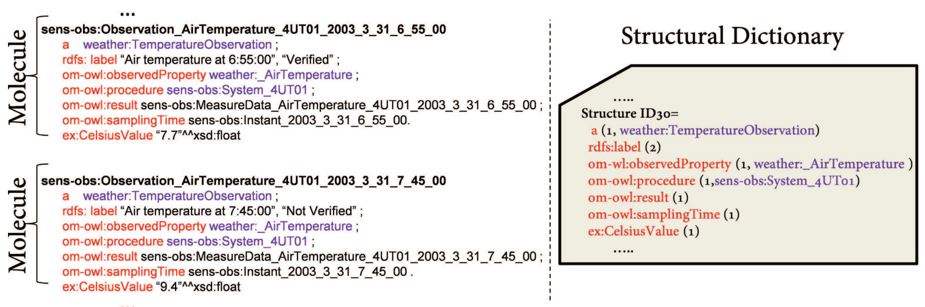
Definition 1 (RDF (general) molecule). *Given an RDF graph G, an RDF molecule $M \subseteq G$ is a set of triples $\{t_1, t_2, \cdots, t_n\}$.*

Molecules are the unit elements for encoding; each molecule will be codified as its corresponding identifier (ID) in the dictionary of structures and the concrete data values held by each predicate.

The most basic (but inefficient) kind of grouping is at the level of triples (one group per triple), *i.e.* having as many molecules as the total number of triples in the RDF data. In this case, the Structural Dictionary will assign an ID to each structure which is just the predicate in the triple. Trying to set up larger groups sharing regularities is much more appropriate.

A straightforward approach is to consider the list of all triples with the same subject (similar to abbreviated triple groups in Turtle [17]). We take this grouping as the method by default, then managing RDF subject-molecules:

Definition 2 (RDF subject-molecule). *Given an RDF graph G, an RDF subject-molecule $M \subseteq G$ is a set of triples $\{t_1, t_2, \cdots, t_n\}$ in which $subject(t_1) = subject(t_2) = \cdots = subject(t_n)$.*



```
...
sens-obs:Observation_AirTemperature_4UT01_2003_3_31_6_55_00
    a    weather:TemperatureObservation ;
    rdfs: label "Air temperature at 6:55:00", "Verified" ;
    om-owl:observedProperty weather:_AirTemperature ;
    om-owl:procedure sens-obs:System_4UT01 ;
    om-owl:result sens-obs:MeasureData_AirTemperature_4UT01_2003_3_31_6_55_00 ;
    om-owl:samplingTime sens-obs:Instant_2003_3_31_6_55_00.
    ex:CelsiusValue "7.7"^^xsd:float

sens-obs:Observation_AirTemperature_4UT01_2003_3_31_7_45_00
    a    weather:TemperatureObservation ;
    rdfs: label "Air temperature at 7:45:00", "Not Verified" ;
    om-owl:observedProperty weather:_AirTemperature ;
    om-owl:procedure sens-obs:System_4UT01 ;
    om-owl:result sens-obs:MeasureData_AirTemperature_4UT01_2003_3_31_7_45_00 ;
    om-owl:samplingTime sens-obs:Instant_2003_3_31_7_45_00 .
    ex:CelsiusValue "9.4"^^xsd:float
...
```

```
.....
Structure ID30=
  a (1, weather:TemperatureObservation)
 rdfs:label (2)
 om-wl:observedProperty (1, weather:_AirTemperature )
 om-owl:procedure (1,sens-obs:System_4UT01)
 om-owl:result (1)
 om-owl:samplingTime (1)
 ex:CelsiusValue (1)
.....
```

Fig. 2. Example of molecules and their Structural Dictionary.

Note that an RDF stream can be seen as a sequence of (potentially not disjoint) RDF subject-molecules[5]. Figure 2 illustrates two molecules in a sequence of weather sensor data and their entry in the Structural Dictionary. This data excerpt (inspired by the data examples in SRBench [20]) represents temperature measures of a sensor at two sampling times. As can be seen, the lists of predicates is exactly the same for both molecules. In addition, we can observe regularities in certain property values (in different color). In particular, the values for *rdf:type*, *om-owl:observedProperty* and *om-owl:procedure* are exactly the same, and will be repeated throughout the data stream for all the air temperature observations of the same sensor. We call this type of predicates producing massive data repetitions, **discrete predicates**. Thus, we avoid these repetitions and save space codifying the concrete values for discrete predicates as part of the structural patterns, as shown in Figure 2 (right). In this example, the structure in the dictionary is encoded as the list of related predicates and, for each one, it counts the number of objects for the predicate and the aforementioned fixed property value if the predicate is discrete.

We assume that discrete predicates can be easily identified by streaming data providers and set up before encoding, or they can be statistically detected at run time. In any case, this kind of information that must be shared between encoders and decoders, is kept in the information set called **Presets**. Presets include all the configuration and compression-oriented metadata supplied by the data provider or inferred at run time. We distinguish between (a) mandatory features in Presets, which include the aforementioned set of discrete predicates and the selected policy for grouping triples into molecules, and (b) application-specific configurations. The latter opens up the format for extensions as long as the concrete application clearly states the decisions and available configurations. For instance, specific features could include common prefixes, suffixes or infixes in URIs and BNodes, or a set of common datatypes in some concrete predicates.

[5] For simplicity, we will use the term molecules hereinafter, assuming that they are subject-molecules by default.

3.2 ERI Streams

At a high level, an **ERI Stream** is a sequence of contiguous blocks of molecules, as depicted in Figure 3. That is, `ERI` first splits the incoming RDF data into contiguous blocks of a maximum predefined *blockSize*, measured in number of triples and set up in the encoder. Then, the molecules (groups) within each block are identified according to the established grouping policy. Note that the grouping by default could slightly alter the original order of triples once it groups triples by subject. Other grouping policies may be established for those scenarios with specific ordering needs.

`ERI` follows an encoding procedure similar to that of the Efficient XML Interchange (EXI) format [18]: each molecule is multiplexed into **channels**:

Definition 3 ((general) Channel). *A channel is a list of lower entropy items (similar values), which is well suited for standard compression algorithms.*

The idea is to maintain a channel per different type of information, so that a standard compressor can be used in each channel, leveraging its data regularities to produce better compression results. In `ERI` we distinguish between two types of channels: (i) *structural channels* and (ii) *value channels*.

Structural channels hold the information of the structure of the molecules in the block and keep the Structural Dictionary updated. We define the following high-level minimum channels:

– *Main Terms of molecules*: In the grouping by default, it states the subject of the grouping. Other kinds of groupings may assign different values.
– *ID-Structures*: It lists the ID of the structure of each molecule in the block. The ID points to the associated structural entry in the Structural Dictionary.
– *New Structures*: It includes new entries in the Structural Dictionary.

Value channels organize the concrete values in the molecules of the block for each non-discrete predicate. In short, `ERI` mainly considers one channel per different predicate, listing all objects with occurrences in the molecules related to it. Having property values of a predicate grouped together may help parsers to directly retrieve information corresponding to specific predicates.

The complete `ERI` stream consists of an `ERI` header followed by an `ERI` body, as shown in Figure 3 (bottom). The `ERI` header includes the identification of the stream and the initial Presets, as previously described. The `ERI` body carries the content of the streaming representing each block as (i) a set of metadata identifying the block and updating potential changes in Presets, and (ii) its compressed channels, using standard compression for each specific channel.

The decoding process is the exact opposite: the stream body is decompressed by channels, and demultiplexed into blocks containing the molecules.

3.3 Practical ERI Encoding and Decoding

Now we describe our current deployment for `ERI` encoding and decoding. For simplicity, we obviate citing the representation of metadata as it is relatively easy to define a key set of keywords, and we focus on channel representations.

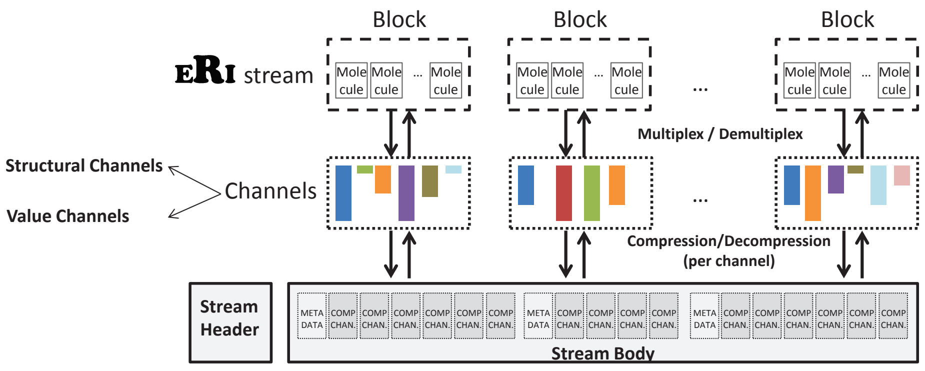


Fig. 3. Overview of the ERI format.

Figure 4 illustrates our practical decisions over the previous example in Figure 2. The figure represents the structural and value channels of the data excerpt, as well as the potential standard compression that could be used to produce each compressed channel. Regarding structural channels, we first follow a straightforward approach for *Main Terms of Molecules* and list main terms (subjects) in plain. Advanced representations could consider the usage of an additional dictionary mapping terms to identifiers, and using the corresponding identifier to refer to a main term previously seen in the input streaming data. However, our experience with streaming data suggests that main terms are rarely repeated because they refer to a certain timestamp.

The *ID-Structures* channel lists integer IDs representing the entry in the Structural Dictionary. New entries are identified by means of an additional channel called *New Structure Marker*. This channel has a bit for each ID in the *ID-Structures* channel: a 0-bit states that the corresponding ID is already in the Structural Dictionary, whereas a 1-bit shows that the ID points to a new entry that is retrieved in the *New Structures* channel. In Figure 4, the first molecule is related to the structure having the ID-30, which is marked as new. Then, the concrete structure can be found in *New Structures*. Similarly to the example in Figure 2, we codify each dictionary entry as the predicates in the structure, the number of objects for each predicate and the concrete property values for discrete predicates. To improve compactness in the representation we use a dictionary of predicates, hence the predicate in the structure is not a term but an ID pointing to the predicate entry in this dictionary. If there is a new predicate never seen before in a block, it is listed in an additional *New Predicates* channel, as shown in Figure 4.

The decoder will maintain a pointer to the next entry to be read in *New Structures* (and increment it after reading), and to hold and update the dictionary of structures and predicates. Given that the number of predicates is relatively low in RDF datasets, we consider a consecutive list of IDs in the predicate dictionary for the encoder and decoder. For the dictionary of structures, we use a *Least Recently Used* (LRU) policy for the dictionary in the encoder. That is, whenever the maximum capacity is reached, the LRU entry is erased

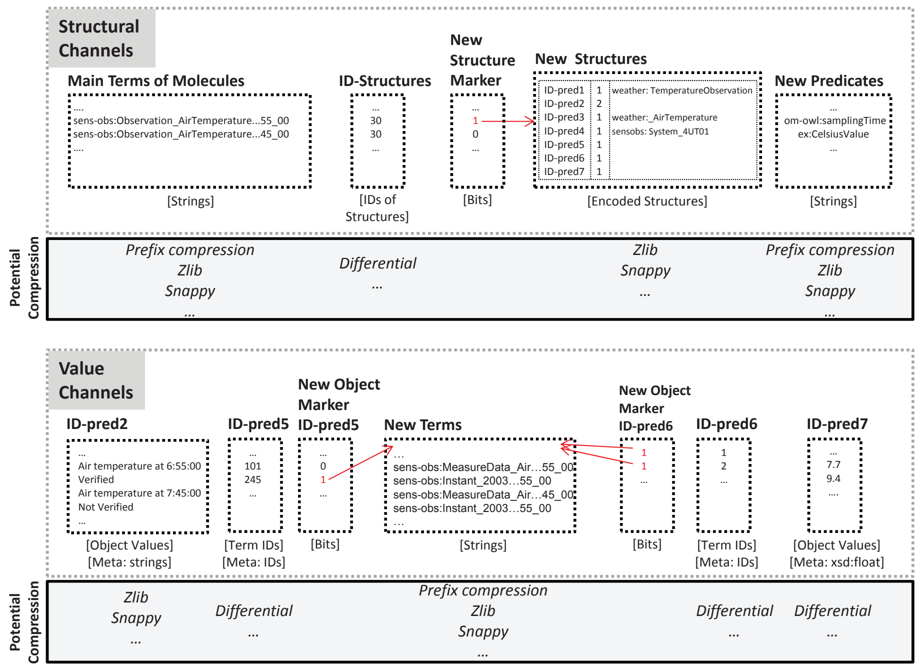


Fig. 4. Example of ERI channels.

and the ID is available to encode the next entry, which must be marked as new in the *New Structure Marker* channel. Therefore, the decoder can make use of simple hashing for the dictionary, as it always knows if the entry is new.

Regarding value channels, Figure 4 illustrates several options. The channel *ID-pred2* storing the values of the second predicate (*rdfs:label*) simply lists all the values. In contrast, *ID-pred5* and *ID-pred6* make use of a dictionary of objects. This case is similar to the dictionary of structures: the channels hold the ID of the entry, and an associated list of bits (*New Object Marker ID-pred5* and *New Object Marker ID-pred6*, respectively) describes if this corresponds to a new entry in the dictionary. The ERI processor maintains an object dictionary per predicate. This decision produces shorter IDs per predicate *w.r.t.* maintaining one general dictionary of objects. In contrast, the processor manages more dictionaries, although the number of different predicates always remains proportionally low, and so the number of dictionaries. In our implementation we maintain one channel (*New Terms*) with all the new terms in the dictionaries. As this list is coordinated with the IDs, there are no overlaps; the decoder must keep a pointer to the next entry to be decoded when a 1-bit in a marker indicates that there is a new term.

Finally, *ID-pred7* also holds the object values directly, as in *ID-pred2*. However, as shown in the figure, it extracts the datatype of all values (*xsd:float*). We assume that all property values for a given predicate are of the same kind. In practice, this means that (i) every channel holds whether URIs/BNodes or liter-

Category	Dataset	Triples	Nt Size (MB)	Subjects	Predicates	Objects
Streaming	Mix	93,048	12	17,153	89	36,279
	Identica	234,000	25	56,967	12	116,065
	Wikipedia	359,028	33	119,676	3	215,382
	AEMET-1	1,018,815	133	33,095	59	5,812
	AEMET-2	2,788,429	494	398,347	7	403,824
	Petrol	3,356,616	485	419,577	8	355,122
	Flickr_Event_Media	49,107,168	6,714	5,490,007	23	15,041,664
	LOD_Nevada	36,767,060	7,494.5	8,188,411	10	8,201,935
	LOD_Charley	104,737,213	21,470	23,306,816	10	23,325,858
	LOD_Katrina	172,997,931	35,548	38,479,105	10	38,503,088
Statistics	Eurostat_migr_reschange	2,570,652	467	285,629	16	2,376
	Eurostat_tour_cap_nuts3	2,849,187	519	316,576	17	47,473
	Eurostat_avia_paexac	66,023,172	12,785	7,335,909	16	235,253
General	LinkedMDB	6,147,996	850	694,400	222	2,052,959
	Faceted DBLP	60,139,734	9,799	3,591,091	27	25,154,979
	Dbpedia 3-8	431,440,396	63,053	24,791,728	57,986	108,927,201

Table 1. Description of the evaluation framework.

als and (ii) all literal values of a given predicate are of the same data type (float, string, dateTime, etc.). We refer to this assumption as *consistent predicates*. Although this is common for data streams and other well-structured datasets, it is more difficult to find general datasets in which this assumption remains true. Thus, we set a parameter in Presets to allow or disallow *consistent predicates*.

Regarding potential channel compressions, Figure 4 includes some standard compression techniques and tools for each type of data. In practice, to simplify the following evaluation, our `ERI` processor uses Zlib whenever textual information is present, *i.e.* in the main terms of molecules, new structures, new predicates and new terms channels. As for those channels managing IDs, each ID is encoded with $log(n)$ bits, n being the maximum ID in the current channel.

4 Evaluation

We implemented a first prototype of an `ERI` processor in Java, following the aforementioned practical decisions. We used some tools provided by the HDT-Java library 1.1.2[6], and the default Deflater compressor provided by Zlib. Tests were performed on a computer with an Intel Xeon X5675 processor at 3.07 GHz and 48 GB of RAM, running Ubuntu/Precise 12.04.2 LTS. The network is regarded as an ideal communication channel for a fair comparison.

4.1 Datasets

Table 1 lists our experimental datasets[7], reporting: number of triples, size in N-Triples (Nt herinafter) format, and the different number of subjects, predicates and objects. We choose representative datasets based on the number of triples, topic coverage, availability and, if possible, previous uses in benchmarking.

[6] https://code.google.com/p/hdt-java.

[7] We have set a Research Object with all the datasets as well as the prototype source code at http://purl.org/net/ro-eri-ISWC14.

We define three different categories of datasets: *streaming* (10), *statistics* (3) and *general* (3). Obviously, **Streaming datasets** are our main application focus; the first six datasets in Table 1 have been already used in the evaluation of RDSZ [10] and correspond to RDF messages in the public streamline of a microblogging site (*Identica*), Wikipedia edition monitoring (*Wikipedia*), information from weather stations in Spain (*AEMET-1* and *AEMET-2*), credit card transactions in petrol stations (*Petrol*) and a random mix of these datasets (*Mix*). We complete the corpora with information of media events (*e.g.* concerts and other performances) in Flickr (*Flickr_Event_Media*), and weather measurements of a blizzard (*LOD_Nevada*) and two hurricanes (*LOD_Charley* and *LOD_Katrina*) extracted from the Linked Observation Data dataset which is the core of SRBench [20].

Statistical datasets are the prototypical case of other (non-streaming) data presenting clear regularities that `ERI` can take advantage of. We consider three datasets[8] (*Eurostat_migr-reschange*, *Eurostat_tour_cap_nuts3* and *Eurostat_avia_paexac*) using the RDF Data Cube Vocabulary [7], providing population, tourism and transport statistics respectively.

Finally, we experiment with **general static datasets**, without prior assumptions on data regularities. We use well-known datasets in the domains of films (*LinkedMDB*) and bibliography (*Faceted DBLP*), as well as *Dbpedia 3-8*.

4.2 Compactness Results

`ERI` allows multiple configurations for encoding, providing different space/time tradeoffs for different scenarios. In this section we focus on evaluating three different configurations: `ERI-1K` (blocksize - 1024), `ERI-4k` (blocksize - 4096) and `ERI-4k-Nodict` (blocksize - 4096). `ERI-1K` and `ERI-4K` include a LRU dictionary for each value channel whereas `ERI-4k-Nodict` does not. We allow the *consistent predicates* option (*i.e.* we save datatype tag repetitions) in all datasets except for the *mix* dataset and all the *general* category in which the aforementioned assumption is not satisfied. In turn, we manually define a set of common discrete predicates in Presets. Finally, according to previous experiences [10], the *blockSize* selection introduces a tradeoff between space and delays: the bigger the blocks, the more regular structures can be found. This implies better compression results, but with longer waiting times in the decoder. Based on this, we select two configurations, 1K and 4K triples providing different tradeoffs.

We compare our proposal with diverse streaming compression techniques. Table 2 analyzes the compression performance providing compression ratios as $\frac{Compressed\_size}{Original\_size}$, taking Nt as the *Original size*. First, we test standard deflate over Nt (*Nt Deflate-4K*), flushing the compression internal buffer each 4096 triples, and over the Turtle (*TTL Deflate*) serialization[9] in the best scenario of compressing the complete dataset at once. We also test the RDSZ approach, which is focused on compressing streams of RDF graphs, whereas `ERI` considers continuous flows of RDF triples. Thus, the evaluation with RDSZ is limited to

[8] Taken from Eurostat-Linked Data, http://eurostat.linked-statistics.org/.

[9] For the conversion process we use Any23 0.9.0, http://any23.apache.org/.

Dataset	Compression Ratio						
	Nt Deflate-4K	TTL Deflate	ERI-4k	ERI-4k-Nodict	RDSZ	HDT-4K	HDT
Mix	8.2%	5.1%	5.2%	5.1%	**4.9%**	10.6%	7.6%
Identica	11.0%	8.5%	8.4%	**8.0%**	8.7%	16.4%	13.6%
Wikipedia	10.5%	7.5%	7.5%	7.7%	**7.2%**	13.4%	10.9%
AEMET-1	4.1%	1.5%	1.2%	**0.8%**	1.3%	4.4%	2.9%
AEMET-2	2.8%	**1.1%**	**1.1%**	**1.1%**	**1.1%**	3.8%	3.8%
Petrol	6.5%	3.8%	2.9%	**2.6%**	3.9%	9.9%	5.2%
Flickr_Event_Media	9.0%	6.9%	6.6%	**6.3%**	6.6%	14.4%	7.2%
LOD_Nevada	3.2%	1.3%	1.5%	1.3%	**1.2%**	4.9%	3.2%
LOD_Charley	3.1%	1.3%	1.4%	**1.2%**	**1.2%**	4.9%	3.2%
LOD_Katrina	3.1%	1.3%	1.4%	**1.2%**	**1.2%**	5.0%	3.2%
Eurostat_migr.	2.1%	**0.5%**	**0.5%**	**0.5%**	-	2.6%	2.5%
Eurostat_tour.	2.2%	0.6%	**0.5%**	0.6%	-	2.6%	2.5%
Eurostat_avia_paexac	2.2%	**0.6%**	**0.6%**	**0.6%**	-	3.2%	2.6%
LinkedMDB	4.7%	2.9%	3.1%	**2.6%**	-	9.5%	5.9%
Faceted DBLP	5.4%	3.7%	4.0%	**3.5%**	-	11.3%	9.2%
Dbpedia 3-8	8.0%	**6.4%**	8.0%	7.5%	-	16.0%	8.0%

Table 2. Compression results on the different datasets.

streaming datasets (the first category in the table), for which we can configure the RDSZ input as a set of Turtle graphs merged together (the input expected by the RDSZ prototype), one per original graph in the dataset. The results of RDSZ depend on two configuration parameters: we use batchSize=5 and cacheSize=100, the default configuration in [10]. For a fair comparison, we consider the original size in Nt in the reported RDSZ compression results. To complete the comparison, we evaluate the `HDT` serialization, although it works on complete datasets. Thus, we also analyze `HDT` on partitions of 4096 triples (`HDT-4k`).

The results show the varied compression ratio between categories and different datasets. The considered statistical datasets are much more compressive than the rest. Besides structural repetitions, they are highly compressible because they include few objects (see Table 1) repeated throughout the dataset.

As can be seen, `ERI` excels in space for streaming and statistical datasets. As expected, it clearly outperforms Nt compression (up to 5 times) thanks to the molecule grouping. This grouping is somehow also exploited by Turtle, which natively groups items with the same subject. Thus, the deflate compression over Turtle can also take advantage of datasets in which predicates and values are repeated within the same compression context. In turn, `ERI` clearly outperforms Turtle compression (up to 1.9 times) in those datasets in which the repetitions in structures and values are distributed across the stream (*e.g. Petrol* and *Identica*).

Similar reasoning can be made for the slightly different results reported by `ERI-4k` and `ERI-4k-Nodict`. As can be seen, the presence of the object dictionary can overload the representation, although it always obtains comparable compression ratios. Note that, since `ERI` groups the objects by predicate within each block, `ERI-4k-Nodict` using Zlib can already take advantage of the redundancies in objects whenever these repetitions are present in the same block. In turn, `ERI-4k` slightly improves `ERI-4k-Nodict` in those cases (such as statistical datasets) in which the object repetitions are distributed across different blocks.

RDSZ remains comparable to our approach. ERI outperforms RDSZ in those datasets in which the division in graphs of the input fails to group redundancies in compression contexts. In contrast, the RDSZ compression slightly outper-

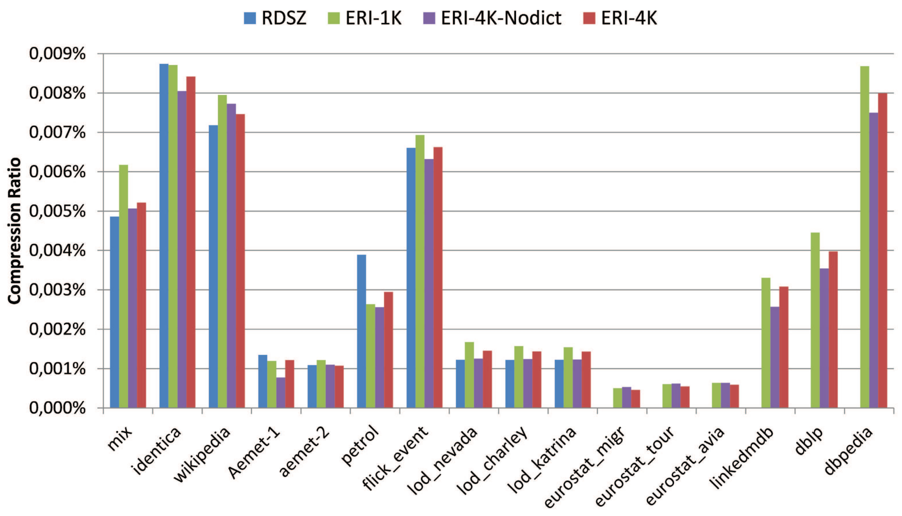


Fig. 5. Analysis of compression results of the considered `ERI` configurations.

forms ERI in two particular cases of interest: *Mix*, where the information is randomly distributed, and the simple *Wikipedia* dataset, where only 3 predicates are present. In such cases, `ERI` pays the cost of having several compression channels and thus flushing the metadata of several compressors (in contrast to one compressor in RDSZ). An alternative, which is not exploited in the present proposal, is to group channels and use one compressor per group of channels. This kind of decision has also been taken by EXI [18].

As for general data, *LinkedMDB* and *Faceted DBLP* datasets provide well-structured information and thus `ERI` can also take advantage of repetitive structures of predicates, obtaining the best compression as well. As expected, `ERI` losses efficiency in a dataset with very diverse information and structures such as *Dbpedia*. Nonetheless, Turtle compression is the only competitor in this case.

As expected, `HDT` is built for a different type of scenario and the results are not competitive *w.r.t.* `ERI`. Although the compression of the full dataset with `HDT` improves the compression by blocks (`HDT-4k`), it remains far from `ERI` efficiency.

Figure 5 compares the compression ratio of the three `ERI` configurations that have been considered. As expected, a smaller buffer in `ERI-1k` slightly affects the efficiency; the more blocks, the more additional control information and smaller regularities can be obtained and compressed. The comparison between `ERI-4k` and `ERI-4k-Nodict` corresponds with the results in Table 2 and the aforementioned analysis denoting the object dictionary overhead.

4.3 Processing Efficiency Results

In this section we measure the processing time of `ERI`, reporting elapsed times (in seconds) for all experiments, averaging five independent executions.

First, we compare compression and decompression times of `ERI` against RDSZ. Table 3 reports the results for the streaming datasets (in which RDSZ applies),

Dataset	Compression Time (sec.)			Decompression Time (sec.)		
	RDSZ	ERI-4k	ERI-4k-Nodict	RDSZ	ERI-4k	ERI-4k-Nodict
Mix	2.5	1.8	**1.2**	**0.5**	1.4	1.2
Identica	8.4	3.1	**2.1**	**0.8**	2.9	2.1
Wikipedia	3.8	2.8	**2.2**	**2.7**	3.2	**2.7**
AEMET-1	17.9	**4.3**	4.6	**3.7**	6.3	5.1
AEMET-2	95.7	15.7	**12.5**	**4.8**	20.3	16.8
Petrol	149.9	13.4	**11.8**	**6.7**	16.8	20.4
Flickr_Event_Media	1,141.8	262.4	**207.2**	**204.0**	311.7	388.2
LOD_Nevada	534.7	329.9	**208.3**	428.2	**191.8**	218.3
LOD_Charley	1,388.9	663.6	**501.4**	1,115.7	**600.6**	611.1
LOD_Katrina	2,315.7	1,002.5	**822.0**	1,869.6	1,038.0	**890.0**

Table 3. Compression and decompression times comparing `ERI` and RDSZ.

comparing `ERI-4k` and `ERI-4k-Nodict`. As can be seen, `ERI` always outperforms the RDSZ compression time (3 and 3.8 times on average for `ERI-4k` and `ERI-4k-Nodict`, respectively). In contrast, `ERI` decompression is commonly slower (1.4 times on average in both `ERI` configurations). Note again that RDSZ processes and outputs streams of graphs, whereas `ERI` manages a stream of triples. Thus, RDSZ compression can be affected by the fact that it has to potentially process large graphs with many triples (as is the case of the LOD datasets), hence the differential encoding process takes a longer time. In contrast, `ERI` compresses blocks of the same size. In turn, `ERI` decompression is generally slower as it decompresses several channels and outputs all triples in the block. In those datasets in which the number of value channels is very small (*Wikipedia* with three predicates and LOD datasets with many discrete predicates), `ERI` decompression is comparable or even slightly better than RDSZ.

As expected, the object dictionary in `ERI-4k` deteriorates the performance against `ERI-4k-Nodict`, once the dictionary has to be created in compression and continuously updated in decompression. The decompression is faster in `ERI-4k` when there are series of triples in which the dictionary does not change.

Then, we test an application managing `ERI` for compression, exchange and consumption processes. We assume hereinafter an average unicast transmission speed of 1MByte/s. Although applications could work on faster channels, we assume that there is a wide range of scenarios, such as sensor networks, where the transmission is much poorer, limited, or costly. In the following, we only focus on streams of RDF triples. Thus, we obviate RDSZ (managing streams of graphs) and Turtle (grouping human-readable triples by default), and we establish compressed Nt as the baseline for a fair comparison.

We first focus on transmission and decompression, without considering at this stage the compression process as many scenarios allow the information to be compressed beforehand. Thus, we measure the **parsing throughput** provided to the client of the transmission, *i.e.* the number of triples parsed per time unit. In turn, the total time includes the exchange time of the considered network, the decompression time and the parsing process to obtain the components (subject, predicate and object) of each triple. Figure 6 reports the average results over the corpora, in triples per second, comparing `ERI-4k` and `ERI-4k-Nodict` against the baseline *NT-Deflate-Blocks-4K*. As can be seen, both `ERI-4k` and `ERI-4k-Nodict` outperform the baseline in most cases except for those datasets

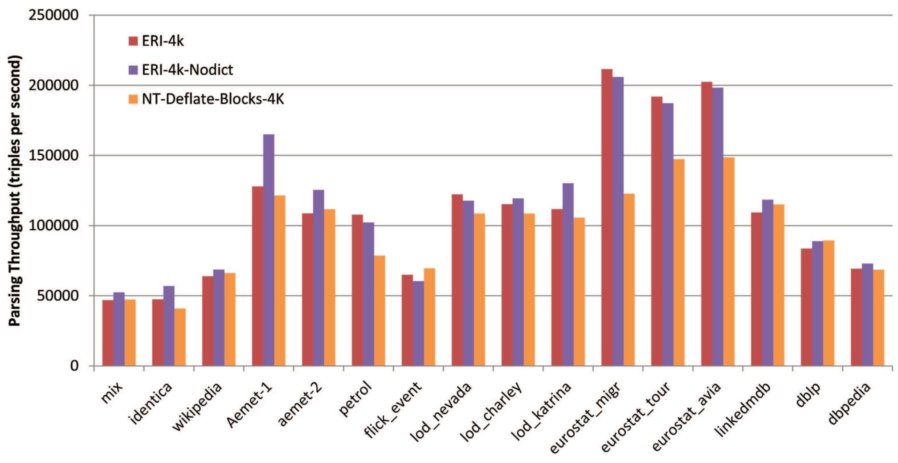


Fig. 6. Analysis of Parsing (Exchange+Decompressing) throughput.

with less regularities in the structure or the data values, which is in line with the previous results for compression. This is the case of *general datasets* as well as two streaming datasets (*Wikipedia* and *Flickr_Event_Media* in which most objects are unrepeated (as can be seen in Table 1). On average, the gains in parsing throughput for both ERI configurations are 110.64% and 142.36% for streaming and statistical datasets respectively, whereas they only decrease to 99.65% for general datasets.

Finally, we address a scenario where compression is subsequently followed by transmission and decompression (including parsing the final output to obtain each triple). Figure 7 compares the resulting times (in logarithmic scale) of *ERI-4k* against the baseline *NT-Deflate-Blocks-4K*. We choose *ERI-4k* over *ERI-4k-Nodict* because the first one produces bigger sizes and worse parsing throughput, hence we are comparing our worst case over the baseline. Under these conditions, *ERI-4k* suffers an expected overhead, given that we are always including the time to process and compress the information in ERI whereas the baseline directly compresses the information. Nevertheless, the time in which the client receives all data in ERI is comparable to the baseline even in this worst case (*ERI-4k-Nodict* performs better than *ERI-4k* as stated above), which the aforementioned improvement in parsing throughput (as shown in Figure 6). In turn, the huge savings in the statistical dataset make ERI slightly faster than the baseline.

5 Conclusions and Future Work

In this paper we have focused on compression as a way to minimize transmission costs in RDF stream processing. In particular, we propose the ERI format, which leverages inherent structural and data redundancy, which is common on RDF streams, especially those using the W3C Semantic Sensor Network Ontology. ERI groups triples into information units called molecules, which are encoded into two type of channels: structural channels referencing the structure of each

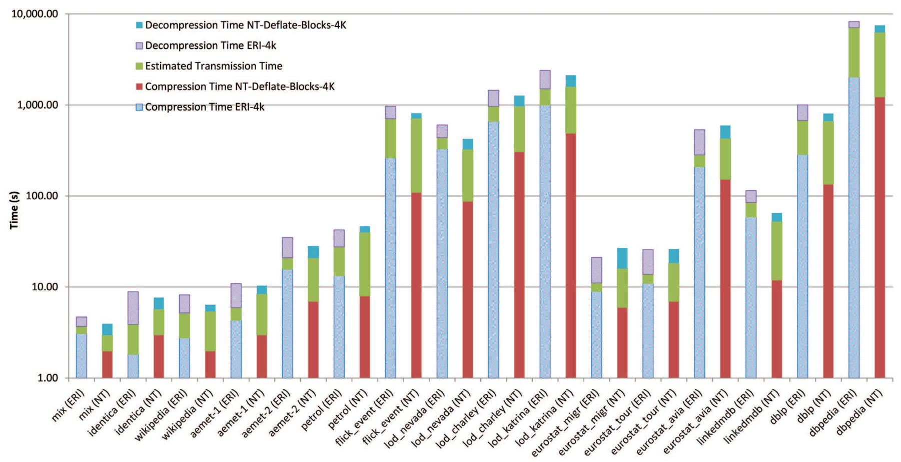


Fig. 7. Comparison of processing performance, *ERI-4K* against *NT-Deflate-Blocks-4k*.

molecule by means of a dynamic dictionary of structures, and value channels grouping and compressing together all the property values by predicate. We provide insights on the flexible and extensible `ERI` configurations and present a practical implementation that is empirically evaluated.

Experiments show that `ERI` produces state-of-the-art compression for RDF streams and it excels for regularly-structured static RDF datasets (*e.g.*, statistical datasets), remaining competitive in general datasets. Time overheads for `ERI` processing are relatively low and can be assumed in many scenarios.

Our next plans focus on integrating `ERI` within the next version of morph-streams [3] , with the purpose of scaling to higher input data rates, minimizing data exchange among processing nodes and serving a small set of retrieving features on the compressed data. This will come together with other new features, including an adaptive query processor aware of the compression dimension during the application of optimization strategies. Besides, we expect to improve performance of `ERI` management by exploring parallel compression/decompression and the use of caches and other fast compressors besides Zlib.

Acknowledgments

This research has been funded by the European Commission under the grant PlanetData (FP7-257641) and by Ministerio de Economía y Competitividad (Spain) under the project "4V: Volumen, Velocidad, Variedad y Validez en la Gestión Innovadora de Datos" (TIN2013-46238-C4-2-R). We are thankful for discussions with authors of the RDSZ approach, specially with Norberto Fernández.

References

1. J. Arias, N. Fernandez, L. Sanchez, and D. Fuentes-Lorenzo. Ztreamy: A middleware for publishing semantic streams on the Web. *Web Semantics: Science,*

Services and Agents on the World Wide Web, 25:16–23, 2014.
2. D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *Proc. of the International Conference on Extending Database Technology (EDBT)*, pages 441–452. ACM, 2010.
3. J. P. Calbimonte, H. Jeung, O. Corcho, and K. Aberer. Enabling query technologies for the semantic sensor web. *International Journal on Semantic Web and Information Systems*, 8(1):43–63, 2012.
4. J.P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 96–111. Springer, 2010.
5. M. Compton, P Barnaghi, L. Bermudez, et al. The SSN Ontology of the W3C Semantic Sensor Network Incubator Group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17(0), 2012.
6. G. Correndo, M. Salvadores, I. Millard, and N. Shadbolt. Linked Timelines: Temporal Representation and Management in Linked Data. In *Proc. of Workshop on Consuming Linked Data (COLD)*, volume CEUR-WS 665, paper 7. 2010.
7. R. Cyganiak, D. Reynolds, and J. Tennison. The RDF Data Cube Vocabulary. *W3C Recommendation*, 16 January 2014.
8. P. Deutsch and J-L Gailly. Zlib compressed data format specification version 3.3. May 1996. Internet RFC 1950.
9. J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange. *Journal of Web Semantics*, 19:22–41, 2013.
10. N. Fernández, J. Arias, L. Sanchez, D. Fuentes-Lorenzo, and O. Corcho. RDSZ: An approach for lossless RDF stream compression. In *Proc. of the Extended Semantic Web Conference (ESWC)*, volume LNCS 8465, pages 52–67, 2014.
11. F. Grandi. Introducing an Annotated Bibliography on Temporal and Evolution Aspects in the Semantic Web. In *SIGMOD Rec.*, volume 41, pages 18–21. 2013.
12. C. Gutiérrez, C. Hurtado, and A. Vaisman. Temporal rdf. In *The Semantic Web: Research and Applications*, volume LNCS 3532, pages 93–107. Springer Berlin Heidelberg, 2005.
13. H. Hasemann, A. Kroller, and M. Pagel. RDF Provisioning for the Internet of Things. In *Proc. of the International Conference on the Internet of Things (IOT)*, pages 143–150. IEEE, 2012.
14. A. K. Joshi, P. Hitzler, and G. Dong. Logical Linked Data Compression. In *The Semantic Web: Semantics and Big Data*, volume LNCS 7882, pages 170–184. 2013.
15. D. Le-Phuoc, H. Q. Nguyen-Mau, C. Le Van, and M. Hauswirth. Elastic and Scalable Processing of Linked Stream Data in the Cloud. In *The Semantic Web - ISWC 2013*, volume LNCS 8218, pages 280–297. Springer Berlin Heidelberg, 2013.
16. A. Margara, J. Urbani, F. Harmelen, and H. Bal. Streaming the Web: Reasoning over Dynamic Data. *Web Semantics*, 25, 2014.
17. E. Prud'hommeaux and Carothers. RDF 1.1 Turtle-Terse RDF Triple Language. *W3C Recommendation*, 25 February 2014.
18. J. Schneider, T. Kamiya, D. Peintner, and R. Kyusakov. *Efficient XML Interchange (EXI) Format 1.0 (Second Edition).* W3C Recommendation, 11 February 2014.
19. J. Urbani, J. Maassen, N. Drost, F. Seinstra, and H. Bal. Scalable RDF data compression with MapReduce. *Concurrency and Computation: Practice and Experience*, 25(1):24–39, 2013.
20. Y. Zhang, P. M. Duc, O. Corcho, and J. P. Calbimonte. SRBench: a streaming RDF/SPARQL benchmark. In *The Semantic Web–ISWC 2012*, pages 641–657. Springer, 2012.