RDSZ: An approach for lossless RDF stream compression

Norberto Fernández[1] and Jesús Arias[1] and Luis Sánchez[1] and Damaris Fuentes-Lorenzo[1] and Óscar Corcho[2]

[1] Dpto. Ing. Telemática, Universidad Carlos III de Madrid, Spain
{berto, jaf, luiss, dfuentes}@it.uc3m.es
[2] Ontology Engineering Group, Universidad Politécnica de Madrid, Spain
ocorcho@fi.upm.es

Abstract. In many applications (like social or sensor networks) the information generated can be represented as a continuous stream of RDF items, where each item describes an application event (social network post, sensor measurement, etc). In this paper we focus on compressing RDF streams. In particular, we propose an approach for lossless RDF stream compression, named RDSZ (RDF Differential Stream compressor based on Zlib). This approach takes advantage of the structural similarities among items in a stream by combining a differential item encoding mechanism with the general purpose stream compressor Zlib. Empirical evaluation using several RDF stream datasets shows that this combination produces gains in compression ratios with respect to using Zlib alone.

Keywords: #eswc2014Garcia

1 Introduction

The popularization of streaming data on the Web has fostered the interest of the Semantic Web community on this kind of data. Some evidences of this interest are, for instance, proposals like C-SPARQL [4] or $\text{SPARQL}_{\text{Stream}}$ [5], which aim to define query languages for RDF streams, work like [13], centered on stream reasoning, CQELS Cloud [9], which addresses the problem of scalable stream processing, or Ztreamy [2], which presents a scalable middleware for stream publishing. As a result of this interest, a W3C community group on RDF Stream Processing[3] has recently started. It is focused on defining a common model for producing, transmitting and continuously querying RDF Streams.

Recent work, particularly CQELS Cloud [9] and Ztreamy [2] has pointed out the importance of compression to reduce communication overheads when transmitting RDF streams. Though the problem of RDF compression has been previously addressed, notably by [1, 6–8, 12], these approaches are mostly centered on compressing static RDF files and datasets. As streams cannot normally be stored in their entirety, compressing streaming data requires different techniques than compressing files. In particular, it requires keeping state information

[3] http://www.w3.org/community/rsp/ (January 13th, 2014)

```
class RDSZCompressor {
  CONSTRUCTOR(cacheSize)
  COMPRESS(RDFgraph)
  data FLUSH()
}
```

```
class RDSZDecompressor {
  CONSTRUCTOR(cacheSize)
  RDFgraph[] DECOMPRESS(data)
}
```

Fig. 1: RDSZCompressor (left) and RDSZDecompressor (right) APIs.

about past data in order to compress future items in the stream, an aspect not covered by the aforementioned approaches.

Taking this into account, in this paper we present an algorithm for lossless RDF stream compression, named RDSZ (RDF Differential Stream compressor based on Zlib). Our approach takes advantage of the fact that, in many cases, RDF streams are constituted by items built automatically by software components according to a single RDF schema (or a small set of them). Due to this, these items have structural similarities that can be exploited by a differential item encoding mechanism, so that new items in the stream can be represented on the basis of the previously processed items. To take advantage of additional redundancies, the results of this differential encoding process are later on compressed using a general-purpose streaming compressor. In particular, due to its popularity, we selected Zlib [10], which implements DEFLATE [11].

Empirical evaluation using several heterogeneous datasets shows that the combination of differential item encoding with Zlib outperforms the usage of Zlib alone. Despite its simplicity, our approach achieves significant improvements, with gains around 9%-31% on the compressed size of some datasets.

The rest of this paper is organized as follows: Section 2 describes the RDSZ algorithm. The empirical evaluation of the algorithm is reported in Section 3. Section 4 offers a discussion on related work. Finally, Section 5 presents some conclusions and future lines of development of this work.

2 The RDSZ algorithm

This section describes the RDSZ algorithm. Section 2.1 introduces its programming interface and intended usage by applications, whereas sections 2.2 and 2.3 describe the algorithm compression and decompression stages.

2.1 Programming interface and usage

The RDSZ API, shown in Figure 1, is based on the Zlib API. When an application needs to compress an RDF stream, it instantiates an RDSZCompressor object. Then, it calls the other two methods in the interface. The method *compress* is called to provide the compressor with a new RDF item in the stream to be compressed. This item may have been obtained, for instance, from a continuous *CONSTRUCT* query in an RDF stream processing engine. The method receives as input a memory data structure that represents the RDF graph of

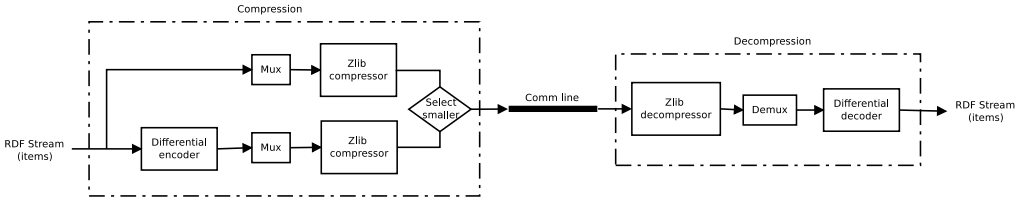

Fig. 2: Processing blocks of RDSZ.

the item[4], containing one or several triples. This RDF graph is buffered by the RDSZ compressor. When a certain number of items have been buffered or after a certain period of time, the application calls *flush*. Then, the compressor flushes its internal buffer, processing the buffered RDF graphs and producing a binary output ready to be sent to an output stream (for instance, a network socket).

The decompression process takes as input a buffer of binary data read from an input stream. To decompress this data, applications should instantiate an RDSZDecompressor object. Later on, they call the *decompress* method, providing the binary data as input and obtaining as result a list of RDF graphs, each of them representing an item in the RDF stream.

The RDSZ compressor and decompressor and all their methods are described with more detail in the following sections.

2.2 Compression

The pseudocode of the RDSZ compressor is provided in Algorithm 1. The main processing blocks of this compressor (shown in Figure 2) are:

- **Differential encoder** (*DiffEncoder* in Algorithm 1): it carries out the differential encoding of each stream item at the input, on the basis of previously processed items. For each input item, it produces as output an encoded item, in a text-based format.
- **Multiplexer** (*Mux* in Figure 2): it takes as input a sequence of items (encoded or not) and converts it into a single string by concatenating the text serialization of the items. A special delimiter is used to mark the limits of each item, so that the decompressor can separate them again.
- **Zlib compressor**: it takes as input the string generated by the multiplexer and compresses that string using Zlib to exploit additional redundancies.

As shown in Algorithm 1, RDSZ compresses the items in the RDF stream in two different ways: (1) using only Zlib, and (2) using the differential encoder followed by Zlib. Later on, when the results of both mechanisms are obtained, the smaller option is selected. Note that, in principle, it is possible to run in parallel both alternative mechanisms (to take advantage of multi-core processors). However, our current implementation does not exploit this possibility.

[4] In our Python implementation, this RDF graph is an *rdflib.Graph* object.

Algorithm 1 RDSZ compressor

```
    /* Build a RDSZ compressor object */
 1: function Constructor(cacheSize)
 2:     compressor ← ZlibCompressor()
 3:     mux ← Multiplexer()
 4:     encoder ← DiffEncoder(cacheSize)
 5:     items ← []
 6: end function

    /* Append an RDF item to the compressor buffer */
 7: function Compress(RDFgraph)
 8:     items.append(RDFgraph)
 9: end function

    /* Flush the buffer and compress the RDF items */
10: function Flush
11:     compressorCopy ← compressor /* Clone the state of the Zlib compressor */
12:
13:     /* Compress the items (serialized in Turtle) only with Zlib */
14:     turtleItems ← serializeInTurtle(items)
15:     string ← mux.multiplex(turtleItems)
16:     outZlib ← compressorCopy.compress(string)
17:
18:     /* Compress the items with differential encoding plus Zlib */
19:     encodedItems ← encoder.encode(items)
20:     string ← mux.multiplex(encodedItems)
21:     outDiffZlib ← compressor.compress(string)
22:
23:     /* Clean the buffer */
24:     items ← []
25:
26:     /* Select the best strategy (that with smaller results) */
27:     if size(outZlib) ≤ size(outDiffZlib) then
28:         compressor ← compressorCopy
29:         return outZlib
30:     else
31:         return outDiffZlib
32:     end if
33: end function
```

Of the three main processing blocks that constitute the RDSZ compressor, the multiplexer and the Zlib compressor carry out well-known tasks: data concatenation (multiplexer) and standard compression using Zlib. Thus, we will focus the rest of this section in the analysis of the differential encoder.

Algorithm 2 details the pseudocode of the differential encoder. The input received by this encoder (line 4 in Algorithm 2) consists of a sequence of items in an RDF stream. We will use an example to illustrate the process carried out by this component. For instance, let us assume that the input is composed by the items represented in Turtle in figures 3 (first item in the sequence), and 4 (second item).

The RDF items in the input are processed sequentially and separately by the encoder. The first processing carried out with an item is to decompose it into a triple pattern and a set of variable bindings. This process is represented by the call to the method *buildPattern* in Algorithm 2 (line 9).

For the sake of brevity, we will not include here the full pseudocode of the *buildPattern* method. It works in a two stage process:

Algorithm 2 Differential encoder

```
    /* Build a differential encoder object */
 1: function CONSTRUCTOR(cacheSize)
 2:     cache ← LRUCache(cacheSize)
 3: end function

    /* Encode a sequence of RDF items in the stream */
 4: function ENCODE(items)
 5:     encodedItems ← [] /* Initialize to empty list */
 6:     for item in items do
 7:
 8:         /* Decompose the item into a triple pattern and variable bindings */
 9:         (pattern, bindings) ← BUILDPATTERN(item)
10:
11:         /* Use differential encoding if possible (pattern previously processed) */
12:         if pattern in cache then
13:             (patternId, prevBindingsPattern) ← cache.get(pattern)
14:             encodedItem ← SERIALIZE(bindings, patternId, prevBindingsPattern)
15:         else
16:             patternId ← GENID(cache)
17:             encodedItem ← serializeInTurtle(item)
18:         end if
19:
20:         encodedItems.append(encodedItem) /* Append encoded item to results */
21:         cache.put(pattern, (patternId, bindings)) /* Update the cache */
22:
23:     end for
24:     return encodedItems
25: end function
```

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix wtl: <http://webtlab.it.uc3m.es/> .

wtl:_556103084 dc:date "2013-02-20T16:58:32Z";
    dc:author "Wonderboy";
    wtl:pageid 6227038;
    wtl:title "Villeroy & Boch" .
```

Fig. 3: Differential encoder example: 1st input item.

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix wtl: <http://webtlab.it.uc3m.es/> .

wtl:_556103110 dc:date "2013-02-20T16:58:40Z";
    dc:author "Wonderboy";
    wtl:pageid 31317733;
    wtl:title "2013 Women's Cricket World Cup" .
```

Fig. 4: Differential encoder examples: 2nd input item.

1. It orders the triples in the RDF graph of the item. The triples are first ordered taking into account the Turtle serialization of the subject. Those with the same subject are ordered on the basis of the Turtle serialization of the property. Finally, those sharing subject and property are ordered taking into account the Turtle serialization of the object.
2. It iterates over the ordered list of triples in the input RDF item and, for each of these triples, replaces the subject and object by variables. It returns as result a string, which represents the *pattern* obtained as an output of the replacement process, plus a table of variable *bindings*, which map each variable to its particular value in the input.

For instance, given the input RDF item shown in Figure 3, the output of the *buildPattern* method consists of: (1) a string with the pattern represented in Figure 5; and, (2) the variable bindings represented in Table 1. Note that, in

```
?x0 <http://purl.org/dc/elements/1.1/author> ?x1 .
?x0 <http://purl.org/dc/elements/1.1/date> ?x2 .
?x0 <http://webtlab.it.uc3m.es/pageid> ?x3 .
?x0 <http://webtlab.it.uc3m.es/title> ?x4 .
```

Fig. 5: Triple pattern for RDF items in figures 3 and 4.

variable	value
?x0	<http://webtlab.it.uc3m.es/_556103084>
?x1	"Wonderboy"
?x2	"2013-02-20T16:58:32Z"
?x3	6227038
?x4	"Villeroy & Boch"

Table 1: Variable bindings for RDF item in Figure 3.

variable	value
?x0	<http://webtlab.it.uc3m.es/_556103110>
?x1	"Wonderboy"
?x2	"2013-02-20T16:58:40Z"
?x3	31317733
?x4	"2013 Women's Cricket World Cup"

Table 2: Variable bindings for RDF item in Figure 4.

case of the RDF item shown in Figure 4, the pattern would be the same, whereas the bindings would be those indicated in Table 2.

Once the decomposition into pattern and bindings of the input RDF item has been obtained, the encoder needs to determine whether this item can be represented on the basis of a previously processed item in the stream or not.

To take this decision, the encoder uses information about previously processed items that is stored within a Least Recently Used (LRU) cache of size *cacheSize* (defined in line 2 of Algorithm 2). This cache stores patterns of recently processed items. For each pattern, the associated bindings and a unique pattern identifier (an integer $idx \in [0, cacheSize - 1]$) are also stored.

Using the cache information, the encoder takes one of the following options:

(I) If the pattern of the RDF item being processed is already within the cache, this means that another item with the same pattern has been recently processed. Thus, the current item is encoded on the basis of the preceding one. As both items have the same pattern, there is no need to explicitly send all the pattern information to the decompressor again. Only the pattern identifier will be included in the encoded item. Regarding the bindings, the variables may have the same value both in the current and preceding RDF item or not. If the value is the same, there is no need to send it again. Otherwise, the value is included within the encoded item. The result of the encoding process in this case is a string that contains a line for the pattern identifier plus a line for each variable in the bindings. We adopt the following conventions to serialize variable values:
   - The variables are included in the order of their number. The value for variable $?x0$ will be the first, then the value for $?x1$, etc. Due to this, there is no need to include the variable name in the encoded item.
   - The variable values are represented in Turtle format (URIs between <>, string literals between quotes, etc.). Blank nodes are represented with a single underscore (note that when several blank nodes are present, each of them will be a different, unambiguous variable).

```
1
<http://webtlab.it.uc3m.es/_556103110>

"2013-02-20T16:58:40Z"
31317733
"2013 Women's Cricket World Cup"
```

Fig. 6: Differential encoder example: results of the *serialize* method.

– When a variable has the same value both in the current and preceding items, an empty line is included in the encoded item.

The process of representing the current item on the basis of a preceding one is denoted by a call to method *serialize* in Algorithm 2 (line 14). As it can be seen, this method receives as input all the required information: the bindings of both the current item (*bindings*) and the preceding item (*prevBindingsPattern*), and the pattern identifier (*patternId*).

In our particular example, the encoder uses differential encoding when processing the second RDF item, because it has the same pattern as the first item (see Figure 5). The result of the *serialize* method in this case is shown in Figure 6, where it has been assumed, without loss of generality, that the *patternId* in the cache for the pattern of this item is $idx = 1$. It can also be seen the empty line used to represent that the value of $?x1$ is the same for both RDF items (compare Table 1 and Table 2). Note also that some redundancies are present in the results of *serialize* (for instance, two variables share the prefix *"2013*). These redundancies are later exploited by the Zlib compressor included in RDSZ.

(II) In case the pattern of the RDF item being processed is not included within the cache, the *encodedItem* variable is assigned the string serialization in Turtle of the RDF item, without any change (line 17 in Algorithm 2).

In any case, the *encodedItem* is added (line 20 of Algorithm 2) to the list of encoded items to be returned as output.

Finally, the differential encoder updates the cache, storing the information about the RDF item just processed (line 21 of Algorithm 2). In particular, the cache maps the pattern of the item to the associated bindings plus a pattern identifier. The value of this identifier depends on whether the pattern was already in the cache or not. In case the pattern was already in the cache, its previous identifier is reused. In case the pattern was not previously in the cache, a new identifier is generated (call to method *genId* on line 16 of Algorithm 2). This method returns the index of the cache where the new entry is going to be stored.

Once all the input RDF items are processed, the result of the encoding process (list *encodedItems* in Algorithm 2) is returned as output to be processed by the next element in the compressor pipeline: the multiplexer (see Figure 2).

Algorithm 3 RDSZ decompressor

```
    /* Build a RDSZ decompressor object */
 1: function CONSTRUCTOR(cacheSize)
 2:     decompressor ← ZlibCompressor()
 3:     demux ← Demultiplexer()
 4:     decoder ← DiffDecoder(cacheSize)
 5: end function

    /* Decompress a buffer with binary data */
 6: function DECOMPRESS(buffer)
 7:     string ← decompressor.decompress(buffer)
 8:     tokens ← demux.demultiplex(string)
 9:     decodedItems ← decoder.decode(tokens)
10:     return decodedItems
11: end function
```

2.3 Decompression

The pseudocode of the RDSZ decompressor is provided in Algorithm 3. It consists of a set of blocks (see Figure 2) that carry out the inverse processes of their compressor counterparts:

- **Zlib decompressor**: it takes as input binary data in a buffer and decompresses it using Zlib, obtaining a string.
- **Demultiplexer** (*Demux* in Figure 2): it splits the string generated by the decompressor into a sequence of tokens (each of them representing an RDF item, encoded or not). To do so, it uses the same delimiter defined in the compressor multiplexer.
- **Differential decoder** (*DiffDecoder* in Algorithm 3): gets the tokens from the demultiplexer and decodes the items that have been encoded at compression time, returning as result a list of RDF graphs, each of them representing an item in the stream.

The rest of this section will be focused on the RDSZ differential decoder, as the processes of the other two components are well-known. The pseudocode of this decoder is shown in Algorithm 4. It processes the tokens provided by the demultiplexer one by one. For each token it carries out the following tasks:

(I) The tokens at the input can represent an encoded item (like the one depicted in Figure 6) or an unencoded one (that is, a Turtle serialization of the RDF item as shown for instance in Figure 3). The decoder differentiates between these cases by checking the first line of the token, which can be an integer (the *patternId* of an encoded item) or not (unencoded item).
   (a) In case the token represents an encoded item, it is decoded. To do so:
      i. The decoder reads the *patternId* from the first line of the input token (line 10 in Algorithm 4).
      ii. It uses its internal state (a LRU cache with the same information as the encoder cache at the compressor) to obtain the pattern and bindings associated to the *patternId* (line 11 in Algorithm 4).

Algorithm 4 Differential decoder

```
    /* Build a differential decoder object */
 1: function CONSTRUCTOR(cacheSize)
 2:     cache ← LRUCache(cacheSize)
 3: end function

    /* Decode a sequence of tokens from the demultiplexer */
 4: function DECODE(tokens)
 5:     decodedItems ← [] /* Initialize to empty list */
 6:     for token in tokens do
 7:
 8:         /* Check if the token is really encoded or not */
 9:         if token is encoded then
10:             patternId ← token.readFirstLine()
11:             (pattern, prevBindingsPattern) ← cache.searchID(patternId)
12:             decodedItem ← DESERIALIZE(pattern, token, prevBindingsPattern)
13:         else
14:             decodedItem ← deserializeFromTurtle(token)
15:         end if
16:
17:         decodedItems.append(decodedItem) /* Append decoded item to results */
18:
19:         /* Update the cache to keep it in sync with the compressor */
20:         (pattern, bindings) ← BUILDPATTERN(decodedItem)
21:         if pattern in cache then
22:             patternId ← cache.get(pattern)
23:         else
24:             patternId ← GENID(cache)
25:         end if
26:         cache.put(pattern, (patternId, bindings))
27:
28:     end for
29:     return decodedItems
30: end function
```

iii. It reconstructs the original set of triples in the RDF item (call to *deserialize* in line 12 of Algorithm 4) and stores the results in the *decodedItem* variable. Note that using the differential encoding process does not introduce any RDF information loss, as the original item triples are reconstructed at the receiver.

In our particular example, if the token contains the encoded representation of the second input item, shown in Figure 6, the decoder:

i. Reads the pattern identifier ($idx = 1$) from the first line.
ii. Obtains from the cache the pattern for that identifier (pattern in Figure 5) as well as the associated bindings (that will be those of the preceding item with the same pattern, that is, the bindings of the first item, shown in Table 1).
iii. With the bindings of the first item and the contents of the token, the bindings of the second item (Table 2) can be obtained. Then, a variable replacement process over the pattern serves to obtain the triples of the second item and, from them, its RDF graph.

(b) If the token represents an unencoded item, the *decodedItem* variable is assigned the RDF graph obtained by deserializing the Turtle representation of the RDF item (line 14 in Algorithm 4). Obviously, as

the original item is received in this case, no information loss has been introduced by RDSZ.

(II) Once the *decodedItem* has been obtained, it is added to a list of *decodedItems* to be returned as result (line 17 in Algorithm 4).

(III) The decoder cache is updated (lines 20 to 26 in Algorithm 4), to keep it in sync with its counterpart at the encoder. In order to do so, the decoder carries out the same processing as is done in the encoder.

Once all the input tokens are processed, the result of the decoding process (*decodedItems* in Algorithm 4) is returned as output of the decompression process.

3 Evaluation

We implemented a first prototype of the RDSZ algorithm using Python 2.7.3 and RDFLib 4.0.1[5]. We used this prototype to validate empirically our approach, centering our evaluation in two aspects: compression performance, and processing time.

Next sections describe the datasets used in our experiments (Section 3.1), as well as the results of our analysis regarding both the compression performance (Section 3.2) and processing time (Section 3.3).

3.1 Datasets

RDSZ uses differential item encoding, which depends on the item structure. Hence we are interested in evaluating the algorithm using several different datasets with different item schemas. Table 3 describes the datasets used in the experimental evaluation[6], including name, size in bytes, number of RDF items it contains, size in triples, and average size of an item in triples (*Avg.*). We also include in the last column the number of different structural patterns found in the dataset when running RDSZ, which is related with the possibility (or not) of using differential encoding when compressing the dataset. Note that this should be a number between one (every item has the same pattern) and the total number of items in the dataset (every item has a unique pattern).

The datasets *AEMET1* and *AEMET2* represent, using different schemas, information taken from wheather stations in Spain [3]. They were obtained from the Spanish Meteorological Office (AEMET). The dataset *Identica* represents in RDF the messages in the public streamline of the microblogging site Identica[7] on a several day time frame. The dataset *Wikipedia* was obtained by monitoring every 30 seconds for a period of several hours the edits carried out on the English Wikipedia, and representing in RDF information about these edits (page,

[5] https://github.com/RDFLib (January 13th, 2014)

[6] Available for download at: http://www.it.uc3m.es/berto/RDSZ/

[7] http://identi.ca/ (January 13th, 2014)

Name	Size (bytes)	#Items	#Triples	Avg.	#Patterns
AEMET1	34,344,498	33,095	1,018,815	30.78	1,459
AEMET2	263,640,938	398,347	2,788,429	7	2
Identica	17,559,385	25,749	256,699	9.97	104
Wikipedia	14,994,109	2,004	359,028	179.16	2,004
Petrol	324,265,505	419,577	3,356,616	8	1
LOD	27,621,020	25,906	258,533	9.98	5
Mix	5,327,406	5,000	93,048	18.61	371

Table 3: Description of the experimental datasets.

timestamp, etc). The *Petrol* dataset was provided by the Spanish start-up Localidata[8], and provides metadata about credit card transactions in petrol stations. The *LOD* dataset represents sensor observations of different wheather parameters. It was extracted from the *Linked Observation Data*[9] dataset. Finally, the *Mix* dataset was generated by randomly combining items from the other datasets, so that each dataset has the same probability to contribute an item.

3.2 Compression performance

The compression performance of RDSZ depends on several parameters. First, the structure of the items in the stream, that is, the dataset schema, has an impact on the differential encoding process. Second, the results of RDSZ depend on the size of the pattern cache (*cacheSize*). Third, the performance depends also on how items in the stream are grouped to be processed by the compressor, that is, in how many calls to *compress* are made between two successive calls to *flush*, according to the interface shown in Section 2.1. We name this parameter as *batchSize*. The reason for this dependency is that each time the Zlib compressor processes a batch of items, it inserts specific information (in particular, related to Huffman coding) to be sent to the decompressor. Increasing the *batchSize* (that is, compressing larger groups) reduces the total number of item batches to be processed and, thus, reduces the Zlib (and hence RDSZ) overhead.

We first analyze the impact of the dataset on the compression performance. To do so, we run RDSZ in the different datasets assuming a fixed *batchSize* of 5 items and a *cacheSize* of 100 entries. For comparison purposes, we use as baseline the results achieved when the differential encoding process is not used, that is when the items are just serialized (using either RDF/XML, Turtle, NTriples or JSON-LD), multiplexed and compressed with Zlib. Table 4 shows the compressed size in bytes for each approach. The last column indicates the percentage of gain provided by RDSZ with respect to the best performing reference.

According to the results in Table 4, the best performing reference in all the datasets is that where Turtle serialization is used. RDSZ outperforms this reference in all but one of the datasets (*Wikipedia*). Note that, taking into account

[8] http://www.localidata.com/ (January 13th, 2014)
[9] http://wiki.knoesis.org/index.php/LinkedSensorData (January 13th, 2014)

Dataset	Zlib baseline (No differential encoding)				RDSZ	RDSZ
	XML size (bytes)	Turtle size (bytes)	N-Triples size (bytes)	JSON-LD size (bytes)	size (bytes)	Gain
AEMET1	4,325,202	2,299,308	5,552,972	3,051,049	**1,876,624**	18.38%
AEMET2	12,854,321	8,120,614	16,930,673	9,432,218	**5,633,763**	30.62%
Identica	3,335,047	2,604,441	3,569,338	3,078,349	**2,266,868**	12.96%
Wikipedia	3,017,381	**2,466,633**	3,450,287	2,821,515	**2,466,633**	0%
Petrol	29,370,624	23,594,420	34,368,532	25,689,604	**19,806,835**	16.05%
LOD	1,230,708	784,298	1,652,188	1,056,188	**537,646**	31.45%
Mix	889,410	665,786	1,019,060	804,721	**599,775**	9.91%

Table 4: Analysis of the compression performance of RDSZ on the different datasets.

the information shown in Table 3, this is an expected result, because in the *Wikipedia* dataset all the items have a different pattern and, thus, RDSZ does not take any advantage from using differential encoding.

To evaluate the impact of the *batchSize* and *cacheSize* we ran several experiments on the AEMET1 dataset with the following setup: (1) the *cacheSize* parameter varied from 32 to 2048 on a power of two basis; and (2) the *batchSize* was modelled using a Poisson random process, to simulate the scenario of an application that produces stream items (and calls *compress*) according to a Poisson traffic model and calls *flush* periodically. Figure 7 reports the results for the Turtle baseline and RDSZ when the average of the *batchSize* process was set to 2, 5 and 10 items. Due to the random nature of the Poisson process, the experiment was repeated for each pair {*cacheSize*, *batchSize*} to compute the average of the compressed size and its 95% confidence interval. The average values (in Kilobytes) are reported in Figure 7. The confidence intervals were found to be very small (with a maximum error of less than 3KB) and, thus, were not represented to ease visualization. Note that we have also included as reference in Figure 7 the results when the *cacheSize* is 0, where RDSZ matches the baseline.

As shown in Figure 7, increasing the *cacheSize* benefits performance. Furthermore, as expected, increasing the *batchSize* has a positive impact for both RDSZ and the baseline (as both of them use Zlib). Thus, it may seem that a possibility to improve compression performace is simply to increase the *batchSize* arbitrarily. However, this affects the delay perceived when transmiting the stream over a communication line, since larger batches require waiting for more items to be available at the compressor. Thus, the tradeoff *batchSize* versus delay needs to be considered for each particular application. For instance, applications with no real-time restrictions may wait to buffer a large number of items (large *batchSize*) before calling *flush*, whereas applications with real-time restrictions may prefer to call *flush* periodically with a small period to limit the delay, even if the *batchSize* to be processed at each period is small.

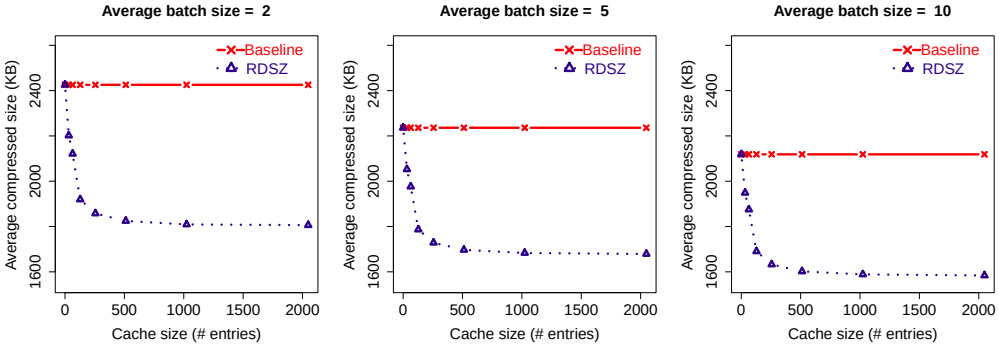

Fig. 7: Evaluating the impact of the *cacheSize* and *batchSize* parameters on the compression performance.

3.3 Processing time

We are interested in measuring the average processing time per item of our current implementation of RDSZ, as this time has an impact in the throughput that can be achieved with our stream compressor. Furthermore, as in Section 3.2, we are also interested in analyzing the influence of the dataset, the *batchSize* and the *cacheSize* into this processing time. To do so, we run our experiments in an Ubuntu 12.04 laptop with an Intel Core2 Duo, 2.53GHz CPU and 8GB RAM.

First, we measure the average compression and decompression time per item in the different datasets, assuming a constant *batchSize* of 5 items and *cacheSize* of 100 entries. The results for RDSZ and the baseline that uses Turtle serialization (the best according to results in Table 4) are reported in Table 5 (measured in milliseconds). The last column in this table shows the ratio obtained by dividing the total (compression plus decompression) average processing time per item of RDSZ by that of the baseline. As indicated in Table 5, the processing time of RDSZ is worse than that of the baseline, as expected, due to the extra processing introduced by RDSZ, and the fact that we are evaluating an unoptimized prototype.

Second, we are also interested in analyzing how the total average processing time per item of our RDSZ prototype depends on the average number of triples per item. To do so, we fitted a linear model between these two variables as measured in all the datasets. This resulted in a line with slope $\alpha = 1.08604$ and $Intercept = -1.55013$. The high $R^2 = 0.9984$ and low $p-value = 2.372e-08$ of the fitted linear model indicate that it explains adequately the relation between the variables, which suggests that the processing time per item is proportional to the number of triples per item for the datasets considered.

Finally, we followed the same experimental setup as in Section 3.2 to evaluate the impact of the *batchSize* and *cacheSize* parameters in the processing time. However, in this case we have not found any significant dependency with these parameters. In particular, running the experiments with different *batchSize* and

Dataset	No differential encoding (Turtle)		RDSZ		Ratio
	Compre. time per item (ms)	Decompre. time per item (ms)	Compre. time per item (ms)	Decompre. time per item (ms)	
AEMET1	5.58	5.30	9.81	17.08	2.47
AEMET2	1.75	1.87	2.33	5.57	2.18
Identica	2.27	2.02	3.07	6.57	2.25
Wikipedia	38.05	33.48	101.48	92.29	2.71
Petrol	2.09	2.05	2.77	6.00	2.12
LOD	2.69	2.70	3.51	7.82	2.10
Mix	3.71	3.48	6.72	10.03	2.37

Table 5: Analysis of the processing time of RDSZ.

cacheSize values, the maximum difference between the total average processing time per item measured between any two runs was less than 1 millisecond.

4 Related work

RDF compression has been only widely addressed recently. One early reference on this topic is [6], where different compression approaches are tested, including: (1) use of general purpose algorithms and (2) definition of compact RDF representations that are later compressed. The conclusions of this work suggest that RDF is highly compressible, especially with compact RDF representations.

In [7] the authors present a compact binary RDF representation, named HDT, that consists of three elements: a header, a dictionary of symbols and the triples encoded according to the dictionary. This structure can be later compressed using Huffman coding and predictive high-order compression techniques, and the result, according to the authors, outperforms universal compressors.

Another relevant work is [1], which describes a compact RDF structure (k2-triples) that allows SPARQL queries to be performed on the compressed representation and, thus, can be used to implement in-memory RDF indexes.

A logical approach to lossless RDF dataset compression is presented in [8]. It consists on automatically building a set of inference rules from the dataset to be compressed and removing all the triples that can be inferred using these rules. The remaining triples plus the inference rules constitute the compressed representation of the original dataset.

The topic of scalable compression of large RDF datasets is addressed in [12], where the authors present a parallel RDF data compression approach based on dictionary encoding techniques and MapReduce.

All of the aforementioned approaches are centered on the compression of large, static, RDF datasets. Compared to that work, the main contribution of this paper is the definition of a lossless compression algorithm for RDF streams.

The topic of RDF stream compression has been indirectly covered in CQELS Cloud [9] and Ztreamy [2]. These references stress the importance of compression for scalable transmission of RDF streams over the network. They also suggest

potential approaches to deal with this issue: dictionary encoding in [9], and Zlib in [2]. However, compression is not the central topic of these papers, and they do not provide an exhaustive analysis of these approaches.

5 Conclusions and future lines

In this paper we presented the RDSZ algorithm for lossless RDF stream compression. It allows to reduce the communication overheads when transmitting RDF streams. The algorithm is based on the combination of a differential item encoding mechanism, which takes advantage of the structural similarities between items in the stream, with the general purpose stream compressor Zlib, to take advantage of additional redundancies. The approach was implemented and evaluated using several heterogeneous RDF stream datasets. The results of this evaluation indicate that the combination in RDSZ of differential item encoding and Zlib produces gains in compression ratios with respect to using Zlib alone.

The current version of RDSZ is not designed to allow querying the compressed RDF stream without decompressing it beforehand. Addressing this issue and integrating our approach into an RDF stream processing engine could represent potential future lines of development of the work presented in this paper.

References

1. Álvarez-García, S., Brisaboa, N.R., Fernández, J.D., Martínez-Prieto, M.A.: Compressed k2-Triples for Full-In-Memory RDF Engines. In: AMCIS (2011)
2. Arias, J., Fernández, N., Sánchez, L., Fuentes-Lorenzo, D.: Ztreamy: A middleware for publishing semantic streams on the web. Web Semantics: Science, Services and Agents on the World Wide Web (in print)
3. Atemezing, G., Corcho, O., Garijo, D., Mora, J., Poveda-Villalón, M., Rozas, P., Vila-Suero, D., Villazón-Terrazas, B.: Transforming Meteorological Data into Linked Data. Semantic Web Journal (2012)
4. Barbieri, D.F., Braga, D., Ceri, S., Grossniklaus, M.: An execution environment for C-SPARQL queries. In: Proceedings of the 13th International Conference on Extending Database Technology. pp. 441–452. EDBT '10 (2010)
5. Calbimonte, J.P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I. pp. 96–111. ISWC'10 (2010)
6. Fernández, J.D., Gutierrez, C., Martínez-Prieto, M.A.: RDF compression: basic approaches. In: Proceedings of the 19th international conference on World Wide Web. pp. 1091–1092. WWW '10 (2010)
7. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semantics: Science, Services and Agents on the World Wide Web 19(0), 22 – 41 (2013)
8. Joshi, A., Hitzler, P., Dong, G.: Logical linked data compression. In: The Semantic Web: Semantics and Big Data, LNCS, vol. 7882, pp. 170–184 (2013)
9. Le-Phuoc, D., Nguyen Mau Quoc, H., Le Van, C., Hauswirth, M.: Elastic and scalable processing of linked stream data in the cloud. In: The Semantic Web ISWC 2013, LNCS, vol. 8218, pp. 280–297 (2013)

10. P. Deutsch and J-L. Gailly (ed.): ZLIB Compressed Data Format Specification version 3.3. Internet RFC 1950 (May 1996)
11. P. Deutsch (ed.): DEFLATE Compressed Data Format Specification version 1.3. Internet RFC 1951 (May 1996)
12. Urbani, J., Maassen, J., Drost, N., Seinstra, F., Bal, H.: Scalable RDF data compression with MapReduce. Concurrency and Computation: Practice and Experience 25(1), 24–39 (2013)
13. Valle, E.D., Ceri, S., Harmelen, F.v., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems 24(6), 83–89 (Nov 2009)