Serializing RDF in Compressed Space*

Antonio Hernández-Illera*, Miguel A. Martínez-Prieto*, and Javier D. Fernández†

*DataWeb Research
Department of Computer Science
Universidad de Valladolid, Spain

† Institute for Information Business
Vienna University of Economics and
Business (WU), Austria

antonio.hi@gmail.com, migumar2@infor.uva.es, jfergar@infor.uva.es

Abstract

The amount of generated RDF data has grown impressively over the last decade, promoting compression as an essential tool for storage and exchange. RDF compression techniques leverage syntactic and semantic redundancies, but structural repetitions are not always addressed effectively. This paper first shows two schema-based sources of redundancy underlying to the schema-relaxed nature of RDF. Then, we revisit the W3C HDT binary format to further compact its graph structure encoding. Our `HDT++` approach reduces the original HDT Triples requirements up to 2 times for more structured datasets, and reports significant improvements even for highly semi-structured datasets like DBpedia. In general, `HDT++` competes with the current state of the art for structural RDF compression, leading the comparison for three of the four analyzed datasets.

1 Introduction

The *Resource Description Framework* (RDF) [9] is a conceptual model which describes data in the form of *triples*. Each triple comprises the resource being described (referred to as *subject*), a property of that resource (*predicate*), and the corresponding value (*object*). Each triple can be seen as a simple graph in which the predicate labels the edge from the subject to the object node. Thus, an RDF dataset is a *labeled directed graph* linking subject descriptions in the form of triples. This flexible paradigm has seen a massive growth in interest over the past few years. RDF has been adopted in many and varied fields of knowledge and leading projects[1]: life-sciences (e.g. *Uniprot*), geography (e.g. *Geonames*), open-government (e.g. *US data.gov*), etc. Not surprisingly, *DBpedia*, an RDF conversion of *Wikipedia*, is the biggest cross-domain dataset and the most accepted reference to assess the benefits of RDF.

Despite it is being widely used, the RDF framework does not restrict how data are serialized. Recently, the RDF Working Group of the World Wide Web Consortium (W3C) collected several practical RDF serialization formats[2]. Although the original RDF/XML is still considered, Turtle-based languages are promoted over it. In any case, these formats are dominated by a document-centric and a human-readable view of RDF, adding unnecessary overheads to the final dataset representation [5]. Thus, the resulting RDF files take up much space, wasting storage and bandwidth resources.

* *Research funded by Ministerio de Economía y Competitividad, Spain: TIN2013-46238-C4-3-R, and Austrian Science Fund (FWF): M1720-G11.*

[1] Uniprot: `http://www.uniprot.org/`; Geonames: `http://www.geonames.org/`; US data-gov: `https://www.data.gov/`; DBpedia: `http://www.dbpedia.org/`

[2] See the recent new version of the RDF primer, `http://www.w3.org/TR/rdf11-primer/`

Even when considering JSON-LD, a serialization which leverages JSON features for compaction (and also makes easy data parsing), the syntax requires great amounts of bytes for effective serialization, so storage and exchange remains inefficient.

HDT [6] is another RDF syntax within the W3C scope[3] but, in contrast to the previous "plain" serializations, it proposes a binary format. HDT encodes RDF into two main data components: the *Dictionary*, providing a mapping between textual terms and numerical identifiers (IDs), and the *Triples*, which encodes the graph structure of IDs, avoiding management of nodes and edges with long strings. HDT outputs very compact RDF serializations [4], enabling meaningful savings in storage and also speeding up exchange processes. However, its graph structure encoding (the *Triples* component) is quite straightforward, and it is not able to leverage particular sources of redundancy underlying to RDF. This paper revisits HDT to improve its Triples encoding. The new approach: `HDT++`, reduces up to 2 times the original Triples space, while outperforms the most prominent RDF compressor: k^2-triples [1] by $10-13\%$.

The rest of the paper is organized as follows. Section 2 delves into the low-level details of HDT and also summarizes the current state of the art for RDF compression. Section 3 shows how some structural RDF features are potential sources of redundancy, and Section 4 explains how our current approach exploits them within HDT foundations. Section 5 compares the current approach with respect to the original HDT, and the aforementioned k^2-triples. Finally, Section 6 concludes about our current work and devises future research leveraging the reported advances.

2 Background

HDT [6] is a binary serialization format optimized for RDF storage and transmission over a network. It encodes RDF data into three components (*Header*, *Dictionary*, and *Triples*) carefully described to address some RDF peculiarities, but also considering how these data are used in the common *Publication-Exchange-Consumption* workflow.

The *Header* is a metadata component that describes relevant information for discovering, parsing and consumption purposes. It uses few kilobytes, so it is free of scalability issues. Then, the RDF graph is represented on the basis of two data components: the *Dictionary* maps all different terms in the dataset to unique identifiers (IDs), and enables the *Triples* component to encode the inner RDF structure as a compact graph of IDs. Efficient encoding of string dictionaries is a challenge beyond RDF compression [2], so the dictionary representation is orthogonal to the problem addressed in this paper. Nevertheless, note that the HDT *Dictionary* component has already been encoded using effective compressed RDF dictionaries [4, 11].

The *Triples* component encodes RDF triples as groups of three IDs: $(\texttt{id}_s\ \texttt{id}_p\ \texttt{id}_o)$, where $\texttt{id}_s$, $\texttt{id}_p$, and $\texttt{id}_o$ are respectively the IDs of the corresponding subject, predicate, and object terms in the *Dictionary*. The current *Triples* component organizes all these triples into a forest of trees, one per different subject in the dataset (see Figure 1). These trees are ordered by subject ID, *i.e.* the i^{th} tree organizes all triples rooted by the i^{th} subject in the *Dictionary*. Each tree has three levels: the root encodes the subject; the second level encodes all predicates related to the subject (predicate

[3]HDT was acknowledged as *Member Submission*, `http://www.w3.org/Submission/HDT/`

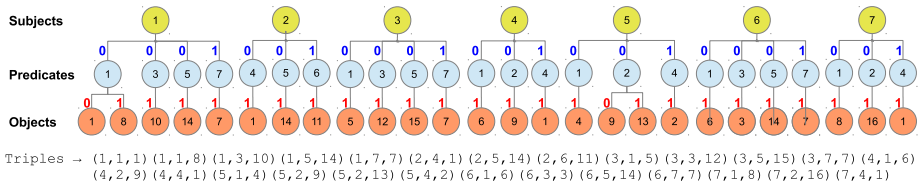

Figure 1: Forest of trees modeling ID triples in HDT.

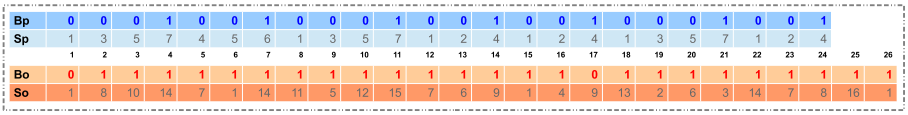

Figure 2: Configuration of binary streams used for encoding the *Triples* component.

IDs are listed in increasing order); and, the leaves encode the adjacency lists of all objects related to each (subject, predicate) pair, also listed in increasing order of object IDs. Note that if a subject is related to j different predicates, its tree encodes j different object lists. This forest organization is succinctly serialized using four binary streams. On the one hand, *two sequences*: `Sp` and `So`, which concatenate predicate and object IDs respectively, following the tree orderings. Given an RDF dataset that comprises $|P|$ different predicates and $|O|$ different objects, the encoded IDs in `Sp` and `So` take $\log |P|$ and $\log |O|$ bits per element respectively. On the other hand, *two bitsequences*: `Bp` and `Bo`, which are aligned with `Sp` and `So` respectively, in the following way. When `Sp[a]` stores the last predicate ID of an adjacency list, then `Bp[a]=1`, being `0` otherwise. In other words, the list of predicates related to the k^{th} subject ends in the k^{th} `1`-bit in the `Bp` bitsequence and starts after the $k-1^{th}$ `1`-bit. This reasoning also applies for object encoding in `Bo` and `So`.

Figure 2 shows the structures which encode the previous example. For instance, the 4^{th} predicate list is encoded from `Sp[12]` to `Sp[14]`: `{1,2,4}`, because `Bp[14]` stores the 4^{th} `1`-bit and `Bp[12]` stores the next 0-bit after the 3^{rd} `1`-bit.

State of the Art of RDF Compression

Following the categorization in [13], HDT can be considered as a *syntactic* compressor because it detects redundancy at serialization level. On the one hand, the *Dictionary* reduces symbolic redundancy from the terms used in the dataset. On the other hand, the *Triples* component leverages structural redundancy from the graph topology. This kind of redundancy is also detected in k^2-triples [1]. This approach performs a predicate-based partition of the dataset into disjoint subsets of (subject, object) pairs. These subsets are highly compressed as (sparse) binary matrices that also allow efficient data retrieval. Other approaches, like HDT-FoQ [10] or WaterFowl [3] also enable data retrieval in compressed space. Both techniques, based on HDT serialization, report competitive performance at the price of using more space than k^2-triples, which is the most effective compressor, to the best of our knowledge.

RDF compression may also leverage semantic redundancy. These *logical* compressors [8] discard triples which can be inferred from others, and they only encode these "primitive triples". Thus, these techniques save space because they reduce the number of triples to be encoded. In addition, they may also apply *syntactic* com-

pression techniques. For instance, Joshi *et al.* [8] combine their approach with HDT, but their results are similar to that obtained by simply using HDT. Recently, Wu *et al.* [13] have proposed SSP, an hybrid compressor leveraging syntactic and semantic redundancy. Its results show that SSP+bzip2 slightly improves HDT+bzip2

3 Schema-based Sources of Redundancy

RDF is described as a schema-relaxed model in which data with different degrees of structure can be integrated. That is, RDF allows structured and semi-structured data to be mixed in a single representation. This flexibility is a double-edged sword because compression techniques can no longer rely on a fixed schema, when in fact RDF datasets present inherent schema-based features that may be a source of redundancy not explicitly considered. Two main sources of redundancy are identified and then integrated into our approach.

Predicate families. The predicates used to describe a subject may vary greatly within a dataset. For instance, the list of predicates used to describe people (`name`, `age`, `e-mail`, etc.) are different to those used to categorize a song (`title`, `author`, `album`, etc.) and both can coexist in a dataset. Moreover, resources can be described with different level of detail (*semi-structured* descriptions): some people can be described using their *name* and *age*, others through their *name*, and *e-mail*, etc. However, it is nonetheless true that, given the descriptive character of RDF, i) there exist predicate repetitions when describing resources of the same nature (*e.g.* between songs and between people), and ii) although the number of predicate combinations (aka: *predicate families*) theoretically grows with the number of predicates, the number of combinations is bounded [4]. Table 1 reports some statistics for four real-world datasets (see Section 5 for more details). On the one hand, `dbpedia` and `linkedmdb` are the less-structured datasets: the *number of predicate families* is $\approx$ 22.5 times the *number of predicates* in `dbpedia`, and $\approx$ 38 times for `linkedmdb`. It denotes the use of a "light" schema. Nevertheless, the number of lists remains significantly small regarding all possible combinations of predicates, so we still found massive repetitions of subjects described with similar predicates. The number of families in `dbtune` is more bounded ($\approx$ 2.5 times) as it is a more structured dataset. On the other hand, the `us census` is a clear example of a highly-structured dataset because the number of families is even less than the number of predicates.

A more fine-grained analysis is performed when considering the presence of the `rdf:type` predicate. This property is used to set the class of the subject being described, but it is not mandatory (*e.g.* no subject in the `us census` describes it). In practice, `rdf:type` tends to be the most repeated predicate, so it is used in many triples along the dataset. As shown in Table 1 (last column), families involving `rdf:type` are a large majority of all existing ones (except for `us census` which does not use `rdf:type`). Thus, predicate families are, in general, related to typed subjects, and the type values come from a small universe of classes (column #classes).

All these features suggest that a family-based encoding may be more effective because it avoids predicate repetitions to be encoded per general or typed subject.

dataset	#triples	#predicates	#classes	#predicate families	#predicate families (class)
linkedmdb	6,147,996	222	53	8,459	8,442
dbtune	58,920,361	394	64	963	782
us census	149,182,415	429	0	106	×
dbpedia	431,440,396	57,986	351	1,309,392	1,152,617

Table 1: Statistical description of some real-world datasets (note that the **#classes** column shows the number of different values (classes) for the `rdf:type` predicate; the **#predicate families (class)** column shows the number of different families including the `rdf:type` predicate).

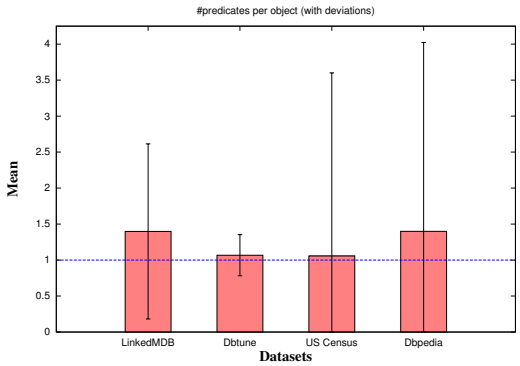

Figure 3: Number of predicates per object (mean and standard deviation).

Predicates per Object. Schema-based redundancies are often referred to the subject being described, but one can also find repetitions in the objects. In RDF, objects set the corresponding values for the descriptions, labeled by means of predicates. While any predicate could be attached to a value, it is obvious that values tend to be very tight to the predicates. For instance, `info@rdfhdt.org` is clearly attached to an *"e-mail"* predicate (it would be rare to find this value in a predicate such as `age`), yet others such as `Nevada` could be a *"family name"*, an *"album"*, etc. Despite this latter exceptional case, it is usual that object values are related to a single predicate [4]. Figure 3 illustrates this fact for the aforementioned datasets, showing that the mean number of predicates per object is very close to 1 (with a limited standard deviation).

In contrast to previous approaches, in which all objects are treated equally (using a global object-ID dictionary), all this stands that objects may be separately encoded within each predicate, thus resulting in local and smaller object IDs.

4 Our Approach

Our current approach focuses on improving the current HDT Triples component to leverage the aforementioned sources of redundancy. First, the concept of predicate families is materialized to improve the HDT predicate encoding. Then, the object encoding is lightened by introducing particular mappings which leverage the fact that most objects are related to just one predicate.

Predicate families. First, the *Triples* component is processed to identify all different predicate families in the dataset. The resulting set of families is then mapped to a range: `[1,` $|F|$`]`, so the i^{th} family is identified by the ID i. This decision enables the *predicates* level to be re-encoded. For each subject, its predicate list is replaced by its

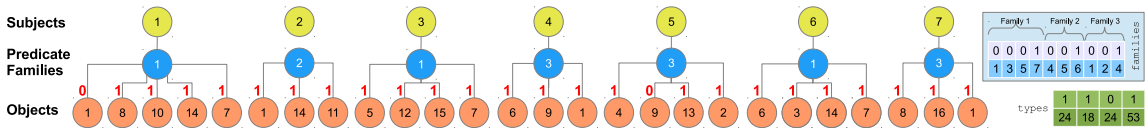

Figure 4: Forest modeling ID triples using predicate families.

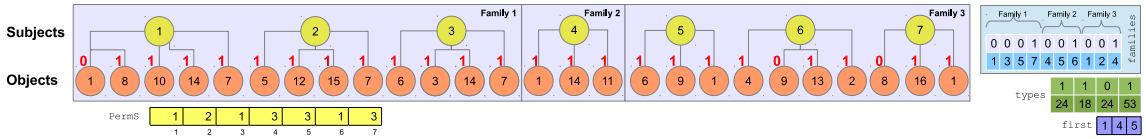

Figure 5: Forest configuration before subject reorganization.

family ID. This is illustrated in Figure 4, showing the conversion of the previous example into three different families. Each subject is now linked to a single node which encodes its corresponding family; e.g. the 4^{th} subject is linked to the 3^{rd} family, which comprises the predicates {1,2,4}. This `families` structure is encoded succinctly by means of a coordinated sequence and bitsequence (top right of the figure).

We leverage the aforementioned preponderance of families associated with typed subjects by extracting all triples involving `rdf:type`, as we represent the class values of the family in a separate `types` structure. This decision saves many object IDs to be encoded at the object level. This new structure stores the IDs which encode the correponding class values in the Dictionary, and uses the ID `0` for encoding non-typed families. Besides this, a coordinated bitsequence is required because a predicate family may involve many `rdf:type` values; e.g. the 3^{rd} family has types `24,53`.

Finally, we perform a two-step subject reorganization in order to bring together all elements related with the same family. The final result is shown in Figure 5. In a first step, we simply put together the trees of the subjects with the same family. For instance, in Figure 4, the subjects of the 1^{st} family are `1`, `3` and `6`; the subject `2` is the only related to the 2^{nd} family and the subjects `4`, `5` and `7` are related to the 3^{rd} family. To avoid the subject ID encoding, we perform a second step, in which we "re-map" the subject IDs, so that the subjects are correlative and implicitly represented, as shown in Figure 5. To do so, we add a *subject permutation* structure: `PermS`, which is aligned with the original Dictionary mapping. That is, `PermS[i]=j` if the original i^{th} subject ID is currently in the j^{th} family. To illustrate how `PermS` is used, let us suppose that we are performing a sequential HDT decoding and we will proceed to decode the 5^{th} subject in Figure 5. It is the first subject in the 3^{rd} family, so we look for the first `3` in `PermS`. It is in `PermS[4]`, so the element encodes the fourth subject term in the Dictionary component. As noted, the start of each family must be stored in a small structure: `first`, which points the first subject ID within each family. In this example, `first=[1,4,5]`, means that the 1^{st} family starts at the first subject, the 2^{nd} family at the fourth, and the 3^{rd} family at the fifth.

Note that `PermS` lists the family for each original subject ID, so it needs the same space than the previous encoding which included the level of family IDs (see the second level in Figure 4). Thus, this re-map apparently does not contribute to compression. However, it is decisive for compressing the objects, as shown below.

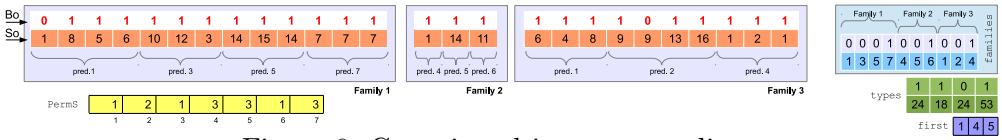

Figure 6: Grouping objects per predicate.

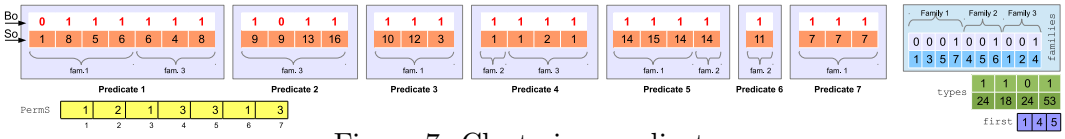

Figure 7: Clustering predicates.

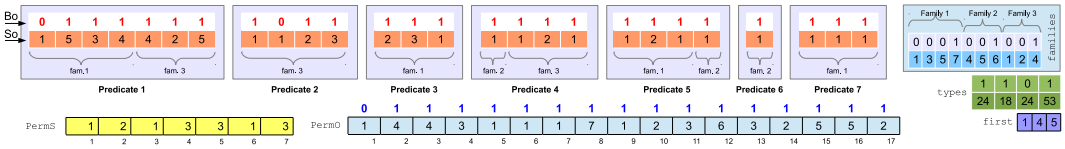

Figure 8: Replacing global by local object IDs.

Predicates per Object. The previous reorganization ensures that all subjects within a given family have the same structure of predicates, as this was the main philosophy of grouping by predicate families. For instance, the example in Figure 5 shows that the first family comprises four predicates `(1,3,5,7)`, so the three subjects in the family are related to these four predicates. This implies that each subject is related to four 1-bits in the object bitsequence[4]. Within each subject tree, objects involving the first predicate in the family (in this case, the predicate ID `1`) are encoded until the first 1-bit; then, objects involving the second predicate (the predicate ID `3`) are encoded between the first and second 1-bit of `Bo`, and so on. Thus, we can easily scan the object sequence and rearrange the objects per predicate within each family. This new organization is illustrated in Figure 6.

This encoding, though, fails to fully group all the objects of the same predicate, as we are splitting and sorting the representation by families. For instance, the objects related to the predicate `1` in the first family are `(1,8,5,6)`, but also `(6,4,8)` within the third family, and both lists are represented separately. Thus, we rearrange all the object lists by predicate (just by moving the lists together), hence achieving the predicate clustering shown in Figure 7.

In spite of the reordering, we always keep track of the objects related to a subject due to the re-mapping assigning consecutive subject IDs within predicate families. Let us suppose we are decoding subject `2`. The `first` structure, which delimits subjects per family, points that it belongs to family `1`, so its predicate list is `(1,3,5,7)`. To retrieve the related objects, and given that we are decoding the subject `2`, we just have to retrieve the second object list in the clusters of predicates `1`, `3`, `5` and `7`.

Finally, we leverage the fact that objects are mostly related to just one predicate. We replace the global ID-object assignment by a local one in which objects are identified in the scope of their predicate. For instance, the third predicate in Figure 7 is used in three triples, with objects: {`10,12,3`}. We can re-map them to a smaller ID range: `[1,2,3]`, so the resulting list is {`2,3,1`}. This is illustrated in Figure 8.

Obviously, we need to keep track of this re-mapping to be consistent with the

[4]As stated, 0-bits point that more than one object is related to the same (subject,predicate) pair.

Algorithm 1: Decoding algorithm.

```
1  for predicate ← 1 to |P| do
2      ptr ← 1;
3      F_p ← families.getFamilies(predicate);
4      for f ← 1 to |F_p| do
5          for s ← first[f] to first[f+1]-1 do
6              subject ← PermS.getSubjectID(s);
7              repeat
8                  object ← Perm0.getObjectID(So[ptr]);
9                  newtriple(subject, predicate, object);
10                 ptr ← ptr + 1;
11             until Bo[predicate][ptr] ≠ 1;
12             T_f ← types.getTypes(f);
13             if T_f[1] ≠ 0 then
14                 for t ← 1 to |T_f| do
15                     newtriple(subject, rdf : type, T_f[t]);
```

ID-object mapping performed in the Dictionary component. However, the situation is different to that explained for the subject permutation because a single object in the original Dictionary may be mapped to more than one local ID in the new Triples component. We add a second permutation: `Perm0` to deal with this issue (see Figure 8, bottom). This structure lists the predicate clusters in which each original ID object appears. For instance, the original object `1` appears within the predicates `1` and `4`, whereas object `2` appears just within the predicate `4`. Once again, a coordinated bitsequence uses `1`-bits to mark the endings of the lists. To translate the i^{th} object ID in the j^{th} dictionary, i) we *select* the i^{th} occurrence of j at the k^{th} position of the sequence; and ii) we *rank* the number of `1`'s until the k^{th} bit in the bitsequence. This rank value is the global ID. For instance, for the object `2` in the third predicate: the 2^{nd} occurrence of 3 in `Perm0` is at position `11`. There are ten `1`-bits up to this position in the bitsequence, then the global object ID is `10` (as can be checked in Figure 7).

Implementation. Our current implementation preserves the Triples component principles in the W3C HDT submission. That is, we encode *adjacency lists* with a couple of aligned structures: the ID sequence and the bitsequence delimiting each list. This ensures our current results to be directly reused by the HDT community.

Thus, predicates clusters (see Figure 8) are encoded as $|P|$ adjacency lists of objects. Encoding costs are different for each list and depends on the number of different objects related with the corresponding predicate. For instance, $\lceil \log 5 \rceil = 3$ bits are used for object IDs in the first predicate, $\lceil \log 2 \rceil = 2$ bits for the second predicate, etc. Note that the original HDT Triples for our example (Figure 2) used $\lceil \log 16 \rceil = 5$ bits to encode each object ID. In turn, predicate families can also be seen as adjacency lists comprising (in increasing order) the corresponding predicate IDs, thus using $\log |P|$ bits per ID. Regarding permutations, `PermS` is a simple array encoding one family ID per subject ($\log |F|$ bits), and `Perm0` is serialized as an adjacency list of predicate IDs ($\log |P|$ bits). Finally, `types` is an adjacency list (using $\log |O|$ bits per type value) and `first` is a sequence of $|F|$ cells ($\log |S|$ bits per cell).

Algorithm 1 describes the decoding process. It is a nested loop algorithm iterating

dataset	#triples	plain (MB)	HDT (MB)	HDT++ (MB)	k²-triples (MB)
linkedmdb	6,147,996	35.91	22.54	14.24	9.02
dbtune	58,920,361	400.36	242.05	132.10	152.27
us census	149,182,415	1,049.25	649.22	312.54	347.06
dbpedia	431,440,396	3,497.36	1,839.08	1,523.72	1,699.39

Table 2: Compression results.

over the $|P|$ different clusters of predicates (Line 1). For each one, it retrieves the families in which the predicate appears (Line 3), and iterates over them (Line 4). For each family, it gets the sequential range of subjects within the family and iterate over them (Line 5). For each subject, the algorithm uses PermS to retrieve the original subject ID (Line 6). Then, it gets the associated object/s directly accessing the object adjacency list at the current scanning position (So[ptr]), getting the original object ID with PermO (Line 8) and then obtaining the current triple (Line 9). Finally, if there are types related to this family (Line 13), then it also outputs the typed triples.

5 Experimental Evaluation

This section shows experimental results for our current approach. It is implemented on a C++ prototype: HDT++, which is built on top of the original *C++ HDT-library*[5].

We choose four different real-world RDF datasets: linkedmdb describes information about movies, actors, characters, etc.; dbtune provides music-related structured data; us census provides census data from the U.S.; and dbpedia is an RDF conversion of Wikipedia. As showed in Table 1, the us census is a well-structured dataset, in contrast to dbpedia which models many and varied types of entities. These datasets also differ in size. They comprise from ≈ 6 millions for linkedmdb and ≈ 431 millions for dbpedia. We compare the most straightforward triples encoding (referred to as plain): it uses three IDs per RDF triple and each one is encoded using $\log |S|$, $\log |P|$, and $\log |O|$ bits); the original HDT encoding [6]; our current approach: HDT++; and k²-triples [1] which is currently the most prominent RDF compressor.

HDT++ outperforms the original HDT encoding for all datasets, but the improvement is more significant for us census. In this case, HDT++ uses less than the half of the space needed by HDT. This is an expected result because it is the most structured dataset. However, the improvements for linkedmdb and dbtune are also noticeable: HDT++ needs $\approx 63\%$ and $\approx 55\%$ of the original space. For dbpedia, HDT++ saves more than 300MB regarding HDT. The comparison regarding k²-triples shows that HDT++ is better for the three largest datasets: $\approx$ 10-13% less space than k²-triples. This result is especially interesting by considering that k²-triples is a pure RDF compressor while HDT++ is a binary serialization format in which no explicit compression is performed. However, k²-triples provides efficient data retrieval in the reported space requirements.

All HDT++ structures are directly mapped to main memory for triples decoding, except the permutations. These are loaded as sparse binary matrices in which the i-th row marks those positions in which the value i is used in the permutation. Each row is compressed using a *SDArray* [12]. Besides this, the bitsequences from families and types are loaded with an overhead of 37.5% on top of their plain representation to

[5] https://code.google.com/p/hdt-it/

provide efficient `rank/select` resolution [7]. This straightforward deployment allows `HDT++` files to be loaded in roughly the same space used for disk storage (even the space is slightly reduced for `dbtune`), and triples decoding is faster than the original `HDT`. For instance, `HDT++` decodes `dbtune` in 2.8 seconds, and `HDT` needs 4.46 seconds.

6 Conclusions and Future Work

This paper revisits the W3C HDT serialization of RDF datasets, improving the compressibility of its graph structure encoding. In spite of the theoretical schema-relaxed nature of RDF, we practically show the presence of two types of schema-based redundancies underlying to RDF: predicate families are massively repeated for general and typed subjects, and objects are often related to just one predicate.

Our `HDT++` approach leverages these features, saving up to half the space used by its `HDT` predecessor and competing on equal terms with the most effective RDF compressor, `k²-triples`. Our achievements can be directly reused by the community since all decisions are aligned to the HDT foundations. Thus, solutions exchanging/consuming HDT can greatly reduce their storage requirements and network latencies. Our future work focuses on exploiting this approach to provide triple pattern resolution by reusing previous experiences on HDT-based retrieval [10].

7 References

[1] S. Álvarez-García, N. Brisaboa, J.D. Fernández, M.A. Martínez-Prieto, and G. Navarro. Compressed Vertical Partitioning for Efficient RDF Management. *Knowl. Inf. Syst.*, 2014. DOI: 10.1007/s10115-014-0770-y.

[2] N. Brisaboa, R. Cánovas, F. Claude, M.A. Martínez-Prieto, and G. Navarro. Compressed String Dictionaries. In *Proc. of SEA*, pages 136–147, 2011.

[3] O. Curé, G. Blin, D. Revuz, and D.C. Faye. WaterFowl: A Compact, Self-indexed and Inference-Enabled Immutable RDF Store. In *Proc. of ESWC*, pages 302–316, 2014.

[4] J.D. Fernández. *Binary RDF for Scalable Publishing, Exchanging and Consumption in the Web of Data*. PhD thesis, University of Valladolid, Spain, 2014.

[5] J.D. Fernández, M. Arias, M.A. Martínez-Prieto, and C. Gutiérrez. Management of Big Semantic Data. In *Big Data Computing*, chapter 4. Taylor and Francis/CRC, 2013.

[6] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange. *J. Web Semant.*, 19:22–41, 2013.

[7] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. of WEA*, pages 27–38, 2005.

[8] A. Joshi, P. Hitzler, and G. Dong. Logical Linked Data Compression. In *Proc. of ESWC*, pages 170–184, 2013.

[9] F. Manola and R. Miller. *RDF Primer*. W3C Recomm., 2004. www.w3.org/TR/rdf-primer/.

[10] M.A. Martínez-Prieto, M. Arias, and J.D. Fernández. Exchange and Consumption of Huge RDF Data. In *Proc. of ESWC*, pages 437–452, 2012.

[11] M.A. Martínez-Prieto, J.D. Fernández, and R. Cánovas. Querying RDF dictionaries in compressed space. *SIGAPP Appl. Comput. Rev.*, 12(2):64–77, 2012.

[12] D. Okanohara and K. Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. of ALENEX*, pages 60–70, 2007.

[13] J.Z. Pan, J.M. Gómez-Pérez, Y. Ren, H. Wu, and M. Zhu. SSP: Compressing RDF data by Summarisation, Serialisation and Predictive Encoding. Technical report, 2014. Available at http://www.kdrive-project.eu/wp-content/uploads/2014/06/WP3-TR2-2014_SSP.pdf.