Accepted version of the article published in: Information Sciences, Volume 508, 2020, Pages 234-259
https://doi.org/10.1016/j.ins.2019.08.081

RDF-TR: Exploiting Structural Redundancies to boost RDF Compression☆

Antonio Hernández-Illera*,[a], Miguel A. Martínez-Prieto[a], Javier D. Fernández[b,c]

[a] *Department of Computer Science, University of Valladolid, Spain.*
[b] *Vienna University of Economics and Business, Austria*
[c] *Complexity Science Hub Vienna, Vienna, Austria*

Abstract

The number and volume of semantic data have grown impressively over the last decade, promoting compression as an essential tool for RDF preservation, sharing and management. In contrast to universal compressors, RDF compression techniques are able to detect and exploit specific forms of redundancy in RDF data. Thus, state-of-the-art RDF compressors excel at exploiting syntactic and semantic redundancies, i.e., repetitions in the serialization format and information that can be inferred implicitly. However, little attention has been paid to the existence of structural patterns within the RDF dataset; i.e. structural redundancy.

In this paper, we analyze structural regularities in real-world datasets, and show three schema-based sources of redundancies that underpin the schema-relaxed nature of RDF. Then, we propose RDF-TR *(RDF Triples Reorganizer)*, a preprocessing technique that discovers and removes this kind of redundancy before the RDF dataset is effectively compressed. In particular, RDF-TR groups subjects that are described by the same predicates, and locally re-codes the objects related to these predicates. Finally, we integrate RDF-TR with two RDF compressors, `HDT` and `k`^{2}`-triples`. Our experiments show that using RDF-TR with these compressors improves by up to 2.3 times their original effectiveness, outperforming the most prominent state-of-the-art techniques.

Keywords: *RDF compression, Linked Data*

1. Introduction

The *Resource Description Framework* (RDF) [29] is a logical 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*). For instance, the triple (`<http://example.org/Dead_Man_Walking>`,

☆A preliminary version of this paper appeared in *Proc. Data Compression Conference (DCC), pages 363–372, 2015.*

*Corresponding author: Departamento de Informática, Escuela de Ingeniería Informática, Campus Miguel Delibes, Paseo de Belén 15, Valladolid, Spain.

Email addresses: `antonio.hi@gmail.com` (Antonio Hernández-Illera), `migumar2@infor.uva.es` (Miguel A. Martínez-Prieto), `jfernand@wu.ac.at` (Javier D. Fernández)

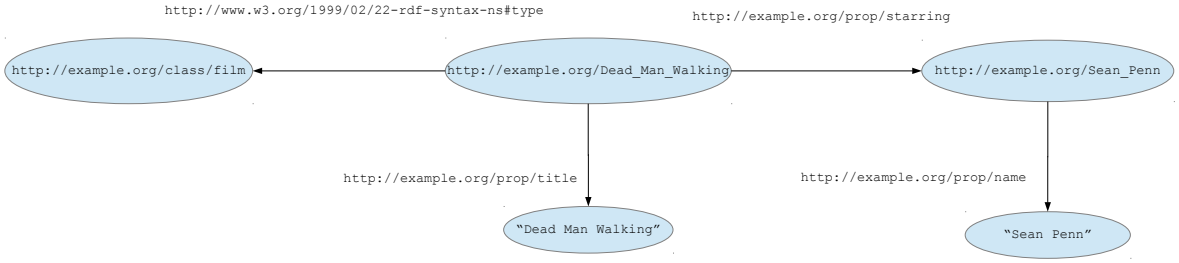

Figure 1: RDF triples modelled as a labelled directed graph.

NTriples

```
<http://example.org/Dead_Man_Walking> <http://example.org/prop/title> "Dead Man Walking".
<http://example.org/Sean_Penn> <http://example.org/prop/name> "Sean Penn".
<http://example.org/Dead_Man_Walking> <http://example.org/prop/starring> <http://example.org/Sean_Penn>.
<http://example.org/Dead_Man_Walking> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://example.org/class/film>.
```

Turtle

```
@prefix ex: <http://example.org/> .
@prefix prop: <http://example.org/prop/> .
@prefix class: <http://example.org/class/> .

ex:Dead_Man_Walking prop:title "Dead Man Walking" ;
  prop:starring ex:Sean_Penn ;
  a class:film .

ex:Sean_Penn prop:name "Sean Penn" .
```

Figure 2: RDF triples presented in NTriples and Turtle formats.

`<http://example.org/prop/title>`, `"Dead Man Walking"`) sets that the resource `<http://example.org/Dead_Man_Walking>` has a `title` property with the value `"Dead Man Walking"`.

An RDF triple can be seen as a directed graph in which the predicate labels the edge from the subject to the object node. Thus, an *RDF dataset* (a set of triples) is often represented as a *labelled directed graph* that links data descriptions in the form of triples. Figure 1 shows a simple RDF graph with four triples that provide a basic description of *Sean Penn* and one of his films, *“Dead Man Walking”*. Note that RDF restricts the types of terms that can play as subject, predicate, or object. Subject roles are always played by International Resource Identifiers (IRIs) or local identifiers (referred to as *blank nodes*) used to denote resources without explicitly naming them. Predicates are always IRIs (often described in a vocabulary or ontology), whereas the object role can be played by both IRIs, blank nodes and also literal values (such as `"Dead Man Walking"` in Figure 1).

This flexible paradigm has attracted increasingly interest over the past few years. RDF has been adopted as the mainstream data representation in diverse fields of knowledge and leading projects[1] such as life-sciences (e.g. *Bio2RDF*), geography (e.g. *Geonames*), or general knowledge (e.g. *Wikidata*), to name but a

[1]Bio2RDF: `http://bio2rdf.org/`; Geonames: `http://www.geonames.org/`; Wikidata: `https://www.wikidata.org`; DBpedia: `http://www.dbpedia.org/`

few. Not surprisingly, *DBpedia*, an RDF conversion of *Wikipedia*, is the largest cross-domain dataset[2] and the most accepted reference to assess the benefits of RDF. In fact, DBpedia is considered the nucleus for the so-called Web of Data [3], an interconnected data-to-data cloud that grows progressively encouraged by the *Linked Open Data* (LOD) initiative[3].

Despite its success, the RDF framework is a logical model, hence it does not restrict how data are (phisically) serialized. The RDF Working Group of the World Wide Web Consortium (W3C) focuses on this issue and collects several practical RDF serialization formats [45]. Serializations have evolved from the initial verbose RDF/XML specification, to more specific, simple and compact formats, such as JSON-LD, Turtle, NTriples, or NQuads. All these "plain" formats lead to document-centric, human-readable serializations of RDF, which add unnecessary overheads when storing, exchanging and consuming RDF graphs in the context of a large-scale and machine-understandable Web of Data.

Figure 2 shows the RDF representation of the previous example in two different formats, Ntriples and Turtle. These forms of representation are equivalent and they suffer from similar verbosity and redundancy problems, as they are both intended for human readability. Although Turtle mitigates redundancy by grouping prefixes (with the inclusion of `"@prefix"` terms) and using some sort of adjacency lists, arbitrary long IRIs, e.g. `ex:Mystic_River` are still present in several triples, acting as subject and object in different triples (the sources of RDF redundancies are reviewed in Section 2). Thus, RDF-specific compression has recently emerged as an effective technique to detect and leverage internal redundancies in RDF data, minimizing space requirements for storage, exchange and consumption processes [30]. In addition, RDF compression plays an increasingly important role in other application areas, such as RDF archiving and versioning [47] or distributed RDF stores [22], among others.

In this scenario, HDT [17], also within the W3C scope [16], represents one of the first and more standardized binary formats for RDF data. The HDT format results in a very compact RDF serialization, enabling significant savings in storage and speeding up data exchange (i.e., less bits over the wire). HDT minimizes the repetition of potentially large strings using the so-called HDT *Dictionary*, which assigns a numerical ID to each term in the dataset. Then, the graph structure of the dataset is managed as a graph of term IDs, in the HDT *Triples* component. While efficient encoding of string dictionaries is a challenge beyond RDF compression [32], triples encoding is an open and active research area. In particular, HDT uses a straightforward configuration and encodes the triples as a forest of trees, one per different subject, using bit and (compact) integer sequences. In turn, the k^2-triples [1] technique elaborates on the encoding of the triples and reports excellent compression ratios by representing triples as a set of (compressed) adjacency matrices, one per different predicate. These compressors, though, disregard specific sources of structural redundancies

[2]The latest DBpedia version comprises more than 13 billion triples from 128 different languages.
[3]`http://linkeddata.org/`

underlying RDF, i.e., common patterns emerging while describing a subject. Note that, although RDF is a flexible, schema-relaxed model, data represented in RDF come with different levels of *structuredness* [12], from structured data (e.g. converted from a relational database) to unstructured data (e.g. from Wikipedia).

In this paper, we analyze common patterns related to the use of predicates and objects in real-world RDF datasets, and show three structural sources of redundancy (introduced in Section 4) underlying the schema-relaxed nature of RDF. This knowledge is then used to describe and implement a new preprocessor: RDF-Tr (*RDF Triples Reorganizer*), which reorganizes triples to improve their effective encoding. Then, we practically show the application of the technique for the aforementioned HDT and k^2-triples compressors, renamed `HDT++` and `k2-triples++` respectively. Our evaluation using real-world RDF datasets shows that the improved compressors outperform their original effectiveness up to 2.3 times, and speed up decompression time up to 3.4 times in HDT and 2.4 times in the case of k^2-triples.

The rest of the paper is organized as follows. Section 2 describes the three different sources of redundancy underlying RDF datasets and summarizes the current state of the art for RDF compression. Section 3 provides background on data compression and compact data structures. Section 4 presents the concrete foundations and sources of redundancy addressed by RDF-Tr. The RDF-Tr reorganization algorithm is fully detailed in Section 5, together with the configuration of compact data structures required to implement it and how the original triples can be decoded. Sections 6 and 7 illustrate the integration of RDF-Tr with existing RDF compressors. In particular, we introduce `HDT++` and `k2-triples++`, the variants of HDT and k^2-triples that compress the "reorganized triples" . Section 8 conducts an exhaustive empirical evaluation of RDF-Tr with different real-world datasets, comparing `HDT++` and `k2-triples++` to their original counterparts. Finally, Section 9 concludes and devises future lines of research.

2. Preliminaries and State of the Art

The adoption of RDF as the main model to represent information in the Web of Data, and the development of ambitious projects such as Linked Open Data, has fostered its use in emerging areas such as *Knowledge Graphs* [7], *Smart Cities* or the *Web Of Things*, and critical sectors such as healthcare and biomedecine [25]. For instance, Bio2RDF consists of around 11 billion triples generated from 35 important biomedical data sources, such as DrugBank, PharmGKB and KEGG. Such ever-increasing dataset sizes present scalability challenges [14] and require efficient mechanisms to represent and consume RDF data.

In this context, RDF compression has emerged as an active research and development field over the past years [30]. Although universal compressors (e.g., *gzip*, *bzip2*, etc) leverage highly verbose RDF serializations, their effectiveness is far from optimal. In general, universal compressors are not able to detect and exploit all types of redundancy underlying RDF data. We first review these sources of redundancy and then analyze state-of-the-art RDF compressors.

2.1. Sources of RDF redundancies

RDF redundancies are categorized at the *semantic*, *symbolic* and *syntactic* level [40]. An RDF graph has semantic redundancy when the information it contains can be represented with fewer triples. Semantic compressors are able to detect this type of redundancy and eliminate extra triples from the original dataset [21]. Then, using inference techniques, the original dataset can be recreated, or at least, a semantically equivalent graph can be obtained. Pure semantic compressors are not so effective by themselves, hence they are often combined with symbolic and/or syntactic compressors.

Symbolic compression involves removing unnecessary repetitions of symbols in a dataset. This is achieved by encoding each element of the RDF graph (URIs, blank nodes and literals) with a corresponding integer identifier (ID), whose value is stored in a dictionary. In turn, these dictionaries provide at least two primitive operations to translate RDF terms to IDs, and vice versa. Note that RDF dictionaries reach non-negligible sizes and, in practice, they must also be compressed [33]. A survey on compressed string dictionaries [32] shows that URI dictionaries can be highly compressed (up to 5% of their original size), while literal dictionaries need more space due to their more heterogeneous composition. In both cases, translation queries can be resolved efficiently (e.g., in $1 - 2\mu s$ per operation in a standard setup [32]).

Syntactic redundancy depends on the RDF graph serialization and also on the underlying graph structure. The simplest RDF syntaxes, such as NTriples [5], write all triples to serialize this subgraph, e.g., one per line. That is, the same subject value would be repeated n times in the resulting file. This drawback can be addressed by simply grouping triples by subject, i.e., considering that the subject structure is described as an adjacency list of *(predicate,object)* pairs. RDF syntaxes, such as Turtle [6], make similar decisions to obtain more compact serializations. RDF compression at this level is traditionally achieved by serializations that firstly reorganize the structure of the graph in order to leverage such redundancies. In addition, serializations can use compact data structures (a brief background is provided in Section 3) to achieve higher levels of compression [30].

2.2. RDF Compression

The current state of the art comprises a rich and diverse set of compressors for RDF data. These are mainly lossless compressors (because they preserve the original information in the dataset), yet lossy compressors are also emerging [24]. We focus on the former and classify them into *physical* and *logical* compressors if they mainly focus on symbolic/syntactic or semantic redundancy respectively. Techniques performing at both physical and logical levels are referred to as *hybrid* compressors.

Physical compressors. These techniques adapt traditional concepts from data compression to the particular case of RDF . On the one hand, they capture and remove symbolic redundancy from RDF terms by using compressed string dictionaries [32]. As explained above, this decision enables the original RDF graph to be

processed as an ID-graph, in which IDs refer to the corresponding terms in the dictionary. On the other hand, different graph encodings have been proposed to compress the resulting ID-graph. Although this approach is widely implemented, there are some physical compressors which tune it from different perspectives.

HDT [17] pioneers this family of RDF compressors and proposes a simple but effective encoding using three main components: i) the *Header* provides descriptive metadata about the dataset; ii) the *Dictionary* maps RDF terms to IDs; and iii) the *Triples* component encodes the underlying graph. The Header is used for dataset discovery and processing, but it is not relevant for compression purposes. We focus on the other two components:

- The *Dictionary* processes RDF terms according to the role they play in the dataset (*subjects*, *predicates*, or *objects*), but organizes them into four disjoint partitions: one for each role, and a fourth one comprising terms which play both subject and object roles. This organization was originally introduced in [2] and allows subject-object terms to be encoded only once. It is a relevant improvement if one considers that, in real-world datasets, up to 60% of the terms are in fact subject-object terms [33]. Let us refer to $|SO|, |S|, |O|$, and $|P|$ as the number of different subjects-objects, total subjects, total objects, and total predicates in the dataset, respectively. Then, term-ID mappings are performed as follows: $[1, |SO|]$ for subjects-objects, $[|SO| + 1, |S|]$ for exclusive subjects, $[|SO| + 1, |O|]$ for exclusive objects, and $|P|$ for predicates. Each dictionary partition is encoded (by default) using the prefix-based Front-Coding compression [32], which ensures very efficient dictionary operations and excellent compression ratios for IRIs. In contrast, this differential encoding is not so effective for literals, hence HDT also provides a self-indexed dictionary for literals [33], which saves space storage at the price of less efficient retrieval operations. Both types of dictionaries can be parameterized to optimize space/time tradeoffs.

- The *Triples* component encodes the resulting ID-graph as a set of $|S|$ *adjacency lists*, one per different subject in the dataset. Each list is modelled as a 3-level tree where the corresponding subject is represented at the root; the middle level sorts all predicate IDs related to the subject; while the leaves organize all object IDs related to each *(subject, predicate)* pair. These trees are encoded using two integer sequences for predicates and objects (subjects are represented implicitly) and two additional bitsequences to represent the shape of the trees. More details about the HDT Triples component can be found in Section 6.1.

HDT has been widely adopted by the Semantic Web community because of its simplicity, its compression levels and its performance for data retrieval operations. It is worth noting that HDT is successfully deployed

in client-side query processors, such as Triple Pattern Fragments[4] [50] and SAGE[5] [35], indexing/reasoning systems like HDT-FoQ [31] or WaterFowl [11], or recommender systems [19] among others. However, its encoding of the graph topology is quite simple and further compression could be achieved. This is addressed by k^2-triples [1], a compressor that organizes RDF terms in the same four partitions used by HDT, but performs a more effective ID-graph encoding. In particular, k^2-triples implements a predicate-based partitioning of the ID-graph and obtains $|P|$ unlabelled graphs. Each of these predicate-graphs is independently encoded as a binary matrix $\mathcal{M}_p$, where $\mathcal{M}_p[i,j] = 1$ means that the subject i and the object j are related by the predicate p, and 0 otherwise. These adjacency matrices, which tend to be sparse, are compressed using the (universal) k^2-trees technique [9], reporting the best compression ratios in the current state of the art of RDF compressors. More details about k^2-triples are provided in Section 7.1.

Two other physical compressors have been published more recently, RDFCSA [8] and OFR [46]. Their contribution is quite different. On the one hand, RDFCSA excels in data retrieval at the cost of larger space requirements, hence it does not outperform the best RDF compressors in the state of the art. RDFCSA first performs the same dictionary transformation explained above. Then, it uses Sadakane's CSA (*Compressed Suffix Array*) [42] to encode the ID-graph. In comparison to those RDF compressors providing efficient triple retrieval, RDFCSA competes with HDT in effectiveness, but it does not reach compression ratios reported by k^2-triples. On the other hand, OFR is a two-stage compressor that mainly focuses on reducing storage requirements, disregarding triples retrieval needs. In the first stage, OFR also isolates terms and triples. Terms are organized into a structure of six sub-dictionaries, first performing partitions by subject, predicate, and object, and then building dictionaries for each different class of term inside them. These dictionaries are *run-length* and *delta* compressed [43]. Regarding triples, they are sorted by *(object,subject)* value and also run-length and delta encoding to exploit multiple object occurrences and the non-decreasing order of the consecutive subjects. Dictionary and triples outputs are then re-compressed during the second stage. The authors consider two universal compressors (*zip* and *7zip*) to remove all remaining redundancy after OFR reorganization. Compression ratios reported by OFR, combined with zip and 7zip, outperform that achieved by HDT+zip and HDT+7zip. Despite of this achievement, these numbers are not enough to compare whether a standalone OFR (with no universal compression afterwards) improves HDT, or the techniques previously explained.

Finally, gRePair [28] extends the RePair algorithm to cater for graphs, including RDF graphs. In short, gRePair builds a grammar with the relationships in the graph and replaces the original graph by another with the rules of the corresponding grammar. gRePair is effective in very specific scenarios, i.e., when the graph has very few predicates and where there is a large number of repetitions in subject-predicate or object-

[4] `http://linkeddatafragments.org/`
[5] `http://sage.univ-nantes.fr/`

predicate relationships. In addition, gRePair has not been compared with specific RDF compressors, but with the interleaved k^2-tree method, which is comparable to k^2-triples. In such scenarios, gRePair obtains the best compression, up to 10 times w.r.t the k^2-tree, in a graph with a single `rdf:type` predicate. In contrast, when the number of predicates increases, the advantage over the k^2-tree decreases, and no evaluation is provided with large and complete real-world datasets.

Logical compressors. These compressors propose different strategies to detect redundant triples (those that could be inferred) and to obtain the canonical subgraphs, which are finally encoded. Initial approaches [21, 34] consider the notion of *lean subgraph*. This concept refers to the smallest instance of the original graph which preserves the ground part of the graph (non-blank nodes and edges connecting them), and maps redundant blank nodes to labels already existing in the graph or to other blank nodes. Ianone *et al.* [21] conclude that the number of triples removed by a lean subgraph greatly depends on the graph features, but a reasonable lower limit is two triples removed] per blank node. Meier [34] states that semantic redundancy is still possible in lean graphs because some of their triples can be derived from others. The author introduces a user-specific redundancy elimination technique based on Datalog-like rules. In short, this approach understands rules in a generative way; i.e., $r(X, Y) \rightarrow t(Y, X)$ means that $t(Y, X)$ are generated from $r(X, Y)$. Thus, if $r(a, b)$ exists in the dataset, it is not necessary to store $t(b, a)$, because it can be inferred. Despite its theoretical contribution, this technique is only well-suited when user-defined rules are explicitly specified. The work of Pichler *et al.* [41] goes a step further and studies how rules, constraints, and queries influence graph minimization. Although it provides a relevant complexity analysis, it does not report any practical results. In fact, Joshi *et al.* [23] note that this approach is application dependent, hindering their adoption for compressing the ever growing RDF datasets.

The *rule-based* (RB) compression method [23] is one of the first approaches reporting effectiveness numbers. It uses mining techniques to detect two types of frequent patterns which are then used as generative rules to remove all triples that can be inferred from such patterns. On the one hand, *intra-property* patterns encompass groups of objects which are commonly used for subject description through a particular predicate. On the other hand, *inter-property* patterns group pairs of predicate-object values related to many subjects. Once the patterns are discovered, RB splits the dataset into two disjoint sets of triples: i) the *dormant* set preserves (in an uncompressed way) those triples to which no inference rule can be applied, and ii) the *active* set differentially encodes all triples to which rules are applied for inferring new triples. While *intra-property* patterns are not so effective, *inter-property* allows up to 50% of the original triples to be removed. However, it has no a significant effect on compression ratios by itself, and RB must be combined with HDT to compete with physical compressors.

The use of frequent patterns does not capture all semantic associations in the dataset [49], so effectiveness can be improved if more expressive rules are considered. The technique proposed in [49] introduces a mining

algorithm focused on Horn rules. A Horn rule can be simply expressed as $B \Rightarrow H$, where $B = B_1 \wedge B_2 \wedge \ldots B_n$ is the *body* and H is the *head*. Both B_i and H are of the form `(?s pred ?o)`, where *pred* is any predicate relating a subject and an object (which can be bounded or left as variables). An instantiation of the rule is considered invalid when a set of triples matches the body rule, but the expected heading triple does not exist in the dataset. On the contrary, a valid instantiation occurs when the corresponding heading triple is in the dataset. Once these Horn Rules are detected, all triples matching the head parts are discarded and the remaining triples are encoded by following the RB strategy. In this case, the *active* set contains all triples used in the body rules, and the *dormant* set comprises triples which do not match any rule. It is worth noting that the latter set also contains conflicting triples. That is, triples that are part of an invalid instantiation of a rule and a valid instantiation of another rule. This Horn rule-based compressor outperforms RB in compression ratio at the price of less efficient compression/decompression processes.

More recently, Guang *et al.* [18] proposed a new rule-based compressor that uses OWL2RL rules [36] to remove redundant triples. First, it analyzes subject-object entities to discover common subgraph patterns. These *entity description patterns* (EDPs) are quite similar to the *predicate families* that we previously proposed in our seminal paper [20] (further detailed in Section 5). That is, for a given entity e, the corresponding EDP comprises i) all predicates p_i such that `(e,`p_i`,`o_x`)` exists in the dataset, and (optionally) ii) the class value v if the triple `(e,rdf:type,v)` is also present. Additionally, an EDP contains all predicates p_j such that `(`s_x`,`p_j`,e)`. The original dataset can be transformed into a set of EDPs by grouping entities which are described by the same EDP. Each group is then independently processed and OWL2RL rules are matched with p_i and p_j predicates in the EDP. An EDPRule is added when the EDP satisfies a particular rule, and its inferred triples are removed. Finally, the remaining s_x and o_x values are also encoded in the context of their EDP. The authors do not provide compression ratios, but report that their approach detects up to 32.77% of redundant triples. In quantitative terms, this result does not improve the previous compressors.

Hybrid compressors. These compressors combine the best of both worlds. On the one hand, they detect and remove syntactic/symbolic redundancy at the serialization level. On the other hand, they consider different strategies to compact the graph by deleting semantic redundancy at the logical level. Although this form of compressors has barely been researched until now, interesting insights are provided in [39, 48].

The *graph-pattern based* (GPB) compressor [39] was published concurrently with our seminal paper [20], and has some common points with our current approach, as explained in the following sections. GPB converts the original dataset into a sequence of *entity description blocks* (EDBs), which group all triples that share the same subject. Each EDB is described by the set of predicates related to the subject and all types assigned to them. EDBs are then grouped into *entity description patterns* (EDPs) which comprise all EDBs with the same description. The current notion of EDP is similar to that explained above. That is, an EDP is a subgraph pattern that describes the structure of predicates and type values for a subset of subjects in

the dataset. Each EDP is encoded as a pair which comprises the corresponding pattern and all instances matching them[6]. This serialization is called *Level 0 method* (LV0). GPB introduces a merge operator that joins EDBs by their relations. This strategy is referred to as the *Level 1 method* (LV1). Finally, the *Level 2 method* (LV2) recursively joins EDBs merged in previous stages. Experimental results show that GPB-LV2 is able to detect and remove many more triples than RB, reporting better compression ratios. It is clear evidence that GPB performs better at the logical level. Regarding its effectiveness at the physical level, the paper does not compare GPB results to those achieved by other compressors. However, the authors emphasize the potential improvements of GPB due to its ability to remove syntactic redundancy.

Finally, *RDF2NormRDF* [48] is an RDF normalization approach, which cleans and eliminates redundancies from RDF datasets as a means of converging into a canonical representation. Thus, it is not a compressor by itself. At the logical level, it removes edge and node duplication by applying particular transformation rules. From a critical point of view, this problem is partially addressed by physical compressors when removing duplicate triples and assigning unique IDs to literals used in more than one triple. However, physical compressors do not deal with blank nodes particularities, preserving their inner redundancy. At the physical level, RDF2NormRDF introduces additional rules to deal with namespace issues and to provide consistent statement orders. It also normalizes how types and language tags are effectively encoded. The normalization process implemented by RDF2NormRDF does not detect more logical/physical redundancy than HDT, but it outperforms HDT for an experimental setup that only comprises small datasets. Besides its compression achievements, RDF2NormRDF outputs normalized datasets that verify all desired quality properties (completeness, minimality, compliance and consistency).

3. Data Compression and Coding

Data compression consists of reducing the number of bits required to encode data [43]. In this paper, we only focus on *lossless compression* (i.e., techniques that are able to reconstruct the original data from its compressed representation), and particularly, on the encoding of integer numbers. In the following, we first review the concept of Variable-Length codes (VLCs) [44], and we summarize state-of-the-art encodings of integer sequences. We then introduce the innovative concept of compact data structures [37] and delve into more details of functional bitsequences. Finally, we review compact data structures for graphs, which are then used in our approach.

3.1. Variable-Length Codes

Some prominent RDF compressors (such as HDT [17]) first transform the RDF dataset into a dictionary of terms and a graph of IDs, before applying additional compression techniques. This allows symbolic

[6]Instances are encoded as IDs based on their MD5 hashes.

and syntactic redundancy to be detected and removed independently, improving the overall compression effectiveness. Focusing on the ID-graph, its adjacency information is first modelled in the form of lists or matrices, and then these structures are encoded.

Variable-Length codes (VLCs) [44] are often used to encode adjacency information, represented in the form of integer IDs. Given an alphabet of integers $\mathcal{A} = \{1, 2, \ldots, \sigma\}$, a VLC maps each value into a variable-length sequence of bits. Thus, VLCs consist of short and long codewords, i.e., compression is optimized when the most frequent integers are encoded with the shortest codewords. Note that VLCs assign the shortest codewords to the initial elements of the alphabet, hence IDs often need to be rearranged to meet this premise.

Different forms of variable-length compression have been proposed in the state of the art [44]. In the following, we focus on the so-called *Elias codes* [13], which are practically used in the implementation of our approach. The *gamma code*: γ is the simplest one and encodes any positive integer n in binary, preceded by $\lfloor \log_2(n) \rfloor$ 0-bits. For instance, the binary encoding of 17 is `10001` and $\lfloor \log_2(17) \rfloor = 4$, so $\gamma(17) =$ `000010001`. γ uses $1 + 2\lfloor \log_2(n) \rfloor$ bits to encode an integer n. In contrast to γ, the Elias *delta code*: δ only uses $1 + \lfloor \log_2(n) \rfloor + 2\lfloor \log_2(1 + \lfloor \log_2(n) \rfloor) \rfloor$ bits to represent n. In this case, the delta code concatenates $\gamma(\lfloor \log_2(n) \rfloor + 1)$, followed by the binary representation of the number excluding the first 1-bit (since it is implicit); e.g., to encode 17, its *gamma* representation is first obtained: $\gamma(\lfloor \log_2(17) \rfloor + 1) = \gamma(4 + 1) = \gamma(5) =$`00101`, and then the binary encoding of 17 is added (without the first 1-bit): `001010001`.

3.2. Encoding of Integer Sequences

Although VLCs can be directly used to compress individual integer IDs from the ID-graph, they disregard potential common regularities in adjacency lists. It is worth noting that adjacency lists are often sequences of increasing IDs, which introduces an additional redundancy.

Let us suppose that we want to encode the adjacency list `L={1000, 1004, 1012, 1019, 1021}` using Elias gamma. In this case, each ID can be directly compressed as $\gamma(1000), \gamma(1004)$, etc. Thus, the length of the corresponding codewords will be proportional to the corresponding ID value. In this case, 21-bit codewords are necessary to encode each ID, so encoding the whole list takes 105 bits.

Gap-encoding is often used to compress *posting lists* in Information Retrieval systems, before using VLCs. Gap-encoding leverages that gaps between consecutive IDs in the list are short, so each ID can be rewritten as the difference to its predecessor; i.e., $L'[i] = L[i] - L[i-1]$. This also applies to the case of adjacency lists. Assuming that the first element is always encoded "as is", the previous list example can be encoded as `L={1000, 4, 8, 7, 2}`. Thus, encoding the first ID takes 21 bits, but the remaining values are encoded using 5, 7, 7, and 3 bits, respectively. Gap-encoding is effective in terms of space saving, but it introduces additional costs for decoding purposes. Note that to obtain the $i-th$ ID of the list, the $i-1$ previous values must be decoded. In practice, gap-compressed sequences are sampled and absolute values are preserved

every k positions. Thus, in the worst case, only $k-1$ values are decoded until the desired value can be obtained.

3.3. Compact Data Structures

Compact data structures are memory-efficient structures that arrange different types of data in a reduced space, and retain querying capabilities over the compressed representation [37]. All these approaches are built on top of functional bitsequences,$B[1,n]$, that provide three main operations:

- $\texttt{access}(B,i)$ returns $B[i]$, for any $1 \leq i \leq n$.
- $\texttt{rank}_v(B,i)$ counts the number of occurrences of the bit v (i.e. $v = \{0,1\}$) in $B[1,i]$, for any $1 \leq i \leq n$. Note that $\texttt{rank}_v(B,0) = 0$.
- $\texttt{select}_v(B,j)$ returns the position of the $j-th$ occurrence of the bit v (i.e. $v = \{0,1\}$) in B, for any $j \geq 0$. Note that $\texttt{select}_v(B,0) = 0$ and $\texttt{select}_v(B,j) = n+1$ if $j > \texttt{rank}_v(B,n)$.

Bitsequences must be enhanced to ensure an efficient performance for these operations. On the one hand, *plain approaches* store the bitsequence as a bit array of n elements, and add additional structures on top of it to ensure competitive time resolution. In our approach, we use the structure proposed by [10], which answers `select` in time $O(1)$ and pays a space overhead $\leq 0.2n$ bits (note that RDF-Tr algorithms do not use `rank`, and `access` can be directly performed on the bit array in constant time). On the other hand, *compressed approaches* [37] exploit different forms of bit redundancy to encode the bitsequence in compressed space while answering the previous operations efficiently. None of the approaches introduced in this paper use this class of bitsequences.

Different innovative compact data structures have been proposed on top of bitsequences and their efficient operations, implementing trees, graphs, or grids, among others [37].

3.4. Encoding of Graphs

Given the scope of this paper and the graph-based RDF model, we hereinafter focus on compact data structures for *directed graphs*. A directed graph $G = (V,E)$ is composed of a set of vertices V and a set of edges $E \subseteq V \times V$, being $n = |V|$ and $e = |E|$. Typically, these structures should provide the following operations [37]:

- $\texttt{adj}(G,v,u)$ returns if the edge $(v,u) \in E$.
- $\texttt{neigh}(G,v)$ returns the list of *direct neighbors* of v; i.e., $\{u,(v,u) \in E\}$.
- $\texttt{rneigh}(G,v)$ returns the list of *reverse neighbors* of v; i.e., $\{u,(u,v) \in E\}$.

- `outdegree`(G, v) returns the number of direct neighbors of v; i.e., $|$`neigh`$(G, v)|$.
- `indegree`(G, v) returns the number of reverse neighbors of v; i.e., $|$`rneigh`$(G, v)|$.

In the following, we distinguish between compact data structures encoding direct graphs as *adjacency lists* or *adjacency matrices*. To illustrate these approaches, we consider a directed graph composed of $n = 6$ vertices and a set of $e = 10$ edges: $E = \{(1,2),(1,3),(2,4),(3,2),(3,4),(3,5),(4,5),(4,6),(5,6),(6,1)\}$.

Adjacency Lists. The simplest compact data structure regards the graph as a sequence of *adjacency lists*, each one listing the direct neighbors of each vertex. For instance, HDT [17] uses adjacency list encoding as part of its *BitmapTriples* component.

Typically, an adjacency list structure, AL, concatenates all adjacency lists into a single sequence, S, and a bitsequence, B, which is used to mark the last element of each list. This configuration is shown in Figure 3, representing the previous example. In this case, six adjacency lists (one per vertex) are concatenated, hence six bits are activated in B (positions 2, 3, 6, 8, 9, and 10). Thus, the list for vertex 1 is encoded in $S[1,2]$, the list for vertex 2 in $S[3]$, vertex 3 in $S[4,6]$, and so on.

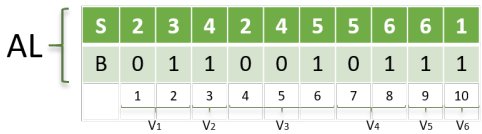

Figure 3: Example of adjacency list encoding.

Adjacency list encoding encompasses an integer sequence, S, and a functional bitsequence, B. Note that S can be *compressed* as explained in Section 3.2, or can preserve IDs in *plain* form, i.e., each ID is encoded using $\lceil \log_2(n) \rceil$ bits. In turn, *plain* or *compressed* approaches can also be used to implement B and its operations [37]. Regardless of the particular implementation, adjacency list encoding allows the aforementioned `adj`, `neigh`, and `outdegree` operations to be efficiently performed, as detailed below. In contrast, this organization results inefficient in operations on reverse neighbors unless the transposed graph is encoded, doubling the required space [37].

The resolution of `adj`, `neigh`, and `outdegree` on vertex v first requires the limits of its adjacency list to be computed. The `getListLimits` function, in Algorithm 1, shows how the left and right limits can be obtained using `select` operations. For instance, `getListLimits(AL,3)` obtains the limits of the adjacency list of vertex 3, which is encoded from $begin =$ `select`$_1(AL.B, 2) + 1 = 4$, to $end =$ `select`$_1(AL.B, 3) = 6$. Then, each operation proceeds as follows:

Algorithm 1: $\texttt{getListLimits}(AL, v)$

1 $begin \leftarrow \texttt{select}_1(AL.B, v-1)+1$;
2 $end \leftarrow \texttt{select}_1(AL.B, v)$;
3 return $(begin, end)$;

Algorithm 2: $\texttt{adj}(AL, v, u)$

1 $(begin, end) \leftarrow \texttt{getListLimits}(AL, v)$;
2 $pos \leftarrow \texttt{binarySearch}(AL.S[begin], AL.S[end], u)$;
3 return pos;

Algorithm 3: $\texttt{neigh}(G, v)$

1 $(begin, end) \leftarrow \texttt{getListLimits}(AL, v)$;
2 $neighbors \leftarrow [AL.S[begin] \dots AL.S[end]]$;
3 return $neighbors$;

Algorithm 4: $\texttt{out}(G, v)$

1 $(begin, end) \leftarrow \texttt{getListLimits}(AL, v)$;
2 return $end - begin + 1$;

- $\texttt{adj}(G, v, u)$ looks for the vertex u in $S[begin, end]$ using a binary search (see Algorithm 2). If $(v, u) \in E$, the operation returns its (local) position in the corresponding adjacency list of v, or -1 otherwise. For instance, in $\texttt{adj}(AL, 3, 4)$, i.e., checking the existence of the edge $(3, 4)$, the value 4 is binary searched in $B[4, 6]$. Thus, $\texttt{adj}(AL, 3, 4) = 2$, because 4 is found in the second element of the corresponding adjacency list of vertex 3. It is trivial to convert the result to a boolean output.

- $\texttt{neigh}(G, v)$ returns an array that includes all values in $S[begin, end]$ (see Algorithm 3). In our example, $\texttt{neigh}(AL, 3)$ returns values in $B[4, 6] = \{2, 4, 5\}$.

- $\texttt{outdegree}(G, v)$ returns $end-begin+1$ (see Algorithm 4). In the previous example, $\texttt{outdegree}(AL, 3) = 6 - 4 + 1 = 3$.

Note that this encoding also provides direct access to any element of an adjacency list. This functionality is commonly invoked as $\texttt{neigh}(G, v)[j]$. It returns the j-th direct neighbor of v, which is located at $S[begin + j - 1]$; e.g., $\texttt{neigh}(AL, 3)[2] = 4$, because $S[5] = 4$.

Finally, it is worth noting that this encoding assumes that all lists have at least one element. Otherwise, if empty lists are allowed, a slight modification must be introduced. In this case, 1-bits still mark the end of the lists, but all elements in a list are now explicitly encoded with 0-bits. For instance, the bitsequence $B' = [011001]$ encodes three adjacency lists: the first one contains one element, the second list is empty, and the third list has two elements. It also causes a slight modification in the $\texttt{getListLimits}(v)$ function, which now obtains the left limit as $pos_l = \texttt{select}_1(v-1) - (v-1)$, and the right one as: $pos_r = \texttt{select}_1(v) - v$. Note that if $pos_r = pos_l$, the corresponding list is empty. In this case, $\texttt{neigh}(G, v) = \emptyset$.

Adjacency Matrices. A naïve approach to encode adjacency matrices is using a vector of vectors. As shown in Figure 4 for our previous example, this approach uses a main vector V, of size n, where each cell stores a pointer to a secondary vector L_i $(1 \leq i \leq n)$, which encodes the neighbors of each vertex in the graph. This structure is preferable to the previous one when the average outdegrees are large, because pointers demand

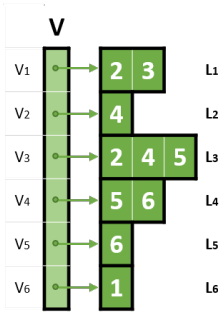

Figure 4: Example of adjacency matrix encoding using a vector of vectors.

fewer bits than the bitsequence. In addition, the independent encoding of each list L_i makes it possible to use more effective techniques to compress the IDs in each list.

Similarly to the previous structure, this approach resolves `adj`, `neigh`, and `outdegree` very efficiently, but it is not a good choice for applications that require operations on reverse neighbors. In particular:

- `adj`(G, v, u) binary searches u in the vector L_v.
- The result of `neigh`(G, v) is the vector L_v itself.
- `outdegree`(G, v) is easily obtained as the length of L_v.

More sophisticated techniques exploit the sparseness and/or clustering features of adjacency matrices to reach high compression ratios. In this respect, the k^2-tree [9] approach is one of the most-used compact data structures for compressing directed graphs.

A k^2-tree models a graph $G(V, E)$ as a binary matrix M of size $m \times m$, where m is the minimum power of k that is greater than n. Thus, $M[i, j] = 1$ *iff* the edge $(i, j) \in E$. M is recursively subdivided into k^2 submatrices, which are (conceptually) organized in a tree and encoded using a bitsequence T: 1-bit means that the corresponding submatrix has at least one non-empty cell, being 0 otherwise. The last level of the tree encodes matrix cell values using another bitsequence L, where 1-bits mean that the corresponding cells encode an existing edge in G.

Figure 5 illustrates the resulting k^2-tree for our graph example. It is modelled as an 8×8 matrix (note that the two right-most columns and the two bottom rows are filled with zeroes to reach the required matrix size, even though they do not encode any existing vertex). The conceptual tree is depicted on the right side, and the resulting bitsequences T and L are shown below. Note that only T and L are actually encoded.

A k^2-tree structure can be efficiently navigated by rows (directed neighbor operations) or by columns (reverse neighbor operations) using `rank` and `select` on the bitsequences (see [9] for more details). Besides it supports the `adj` operation, and different forms of range-based queries. Thus, the k^2-tree is a fully functional data structure for graph encoding that also ensures high compression ratio scenarios, including RDF compression [1].

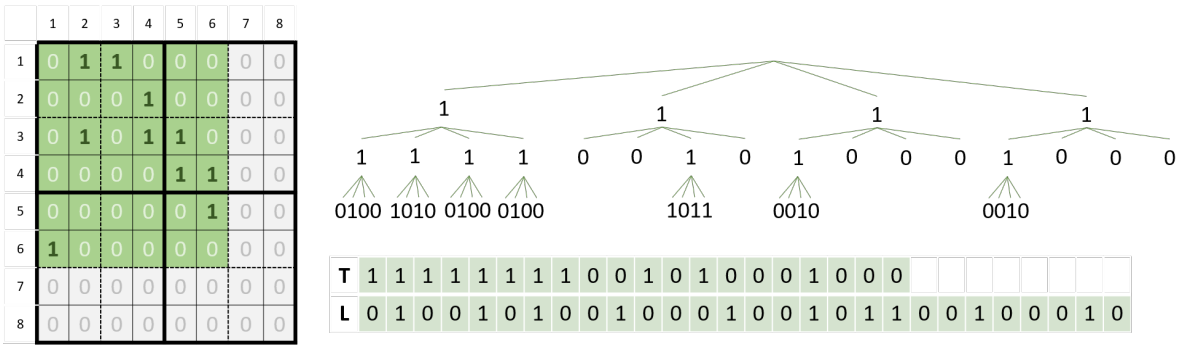

Figure 5: Example of matrix encoding using a k^2-tree.

4. RDF-Tr Foundations

RDF is described as a schema-relaxed model in which data with different degrees of structure can be integrated. However, this flexibility is a double-edged sword. At the logical level, RDF is an effective way to address data variety and allows structured and semi-structured data to be mixed in a single representation. Conversely, this lack of a fixed schema prevents RDF compressors from assuming particular subgraph structures when, in fact, RDF data present many schema-based features. As previously explained, this is a source of redundancy that introduces significant overheads in RDF serializations.

RDF-Tr foundations are drawn from structural/semantic RDF features and focus on improving compression effectiveness. These features are related to the practical use of predicates and objects in real-world RDF datasets.

4.1. Predicates

The set of predicates used to describe a subject may vary greatly within a dataset. For instance, let us suppose that an RDF dataset represents information about cinema. The set of predicates used to describe people (`name`, `age`, `nationality`, etc.) are different from those used to categorize a movie (`title`, `director`, `duration`, etc.) and both coexist in the same dataset. Moreover, resources can be modelled with different levels of detail (*semi-structured* descriptions): some people can be described using their *name* and *age*, others through their *name* and *nationality*, etc.

It is nonetheless true that, given the descriptive character of RDF, there exist predicate repetitions when describing resources of the same nature (*e.g.,* among people). Although the number of predicate combinations (referred to as *predicate families*) used for subject descriptions theoretically grows with the number of predicates, the number of such combinations is bounded, even in datasets with a light schema [15]. In the following, we formalize the concept of predicate family based on the notion of predicate lists [15].

Definition 1 (Predicate Family). *Let G be an RDF graph, and S_G, P_G, O_G be the sets of subjects, predicates and objects in G. We define the* predicate family *F_s as the set of predicates (labels) related to the*

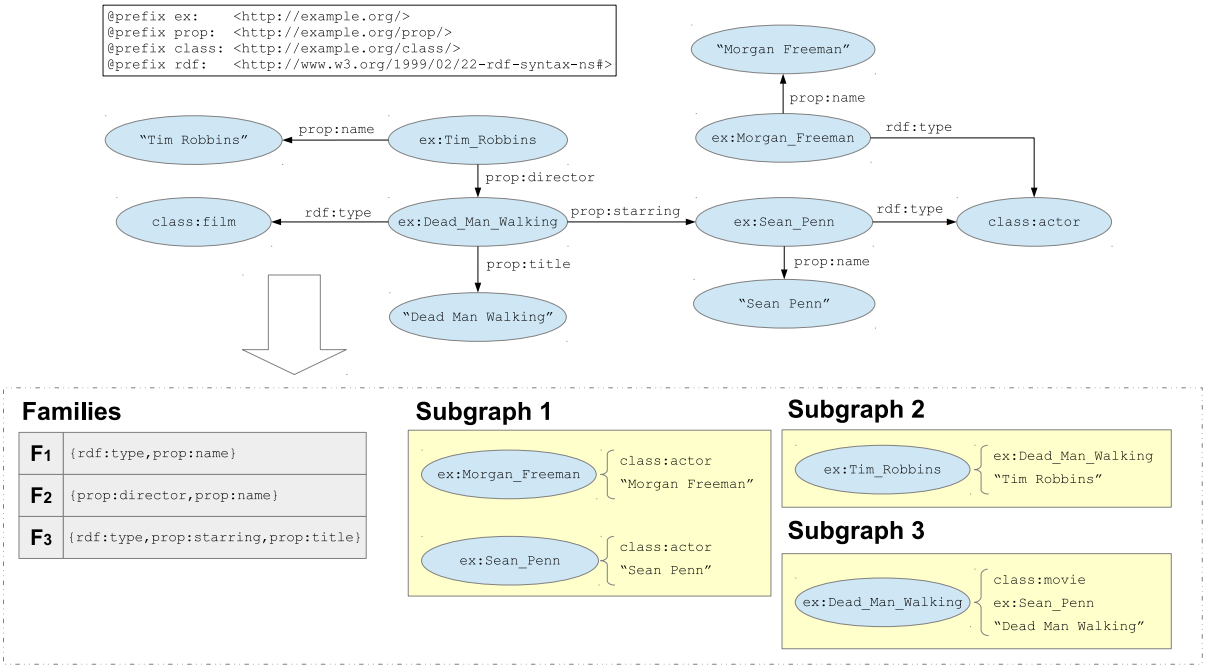

Figure 6: Example of an RDF graph and its predicate families.

subject $s \in S_G$. *That is, the set of predicates* $F_s = \{p \mid \exists z \in O_G, p \in P_G, (s,p,z) \in G\}$. *We denote as* F_G, *or just* F, *the set of different predicate families in* G. *That is,* $F_G = \{F_x, x \in S_G\}$, *hence the* number of predicate families *in the graph* G *is* $|F_G|$ *(or just* $|F|$*).*

The predicate family concept is equivalent to the *Characteristic set* definition introduced by Neumann et al. [38], and it is used to split the graph into subgraphs, each one containing all subjects described with the same set of predicates. Once the subjects are grouped, their predicate structure can be implicitly encoded attending to their corresponding predicate family.

Figure 6 illustrates the use of predicate families for a given RDF excerpt about films, which extends our previous example in Figure 1. In this example, we find three different families: F_1 ={`rdf:type, prop:name`}; F_2 ={`prop:director, prop:name`}; and F_3 ={`rdf:type, prop:starring,prop:title`}, so subject descriptions can be split into three disjoint subgraphs which implicitly encode the corresponding predicate structures. For instance, the objects {`class:actor, ''Morgan Freeman''`} describe the corresponsing subject `<http://example.org/Morgan_Freeman>` within the scope of the first subgraph. Thus, we can infer that the subject and the given objects are linked through the predicates of the first family (`rdf:type`, and `prop:name`).

All this sets the basis of our first foundation, which guides the design of our proposal:

Foundation 1. *A predicate family models a subgraph pattern that comprises all predicates to describe a set*

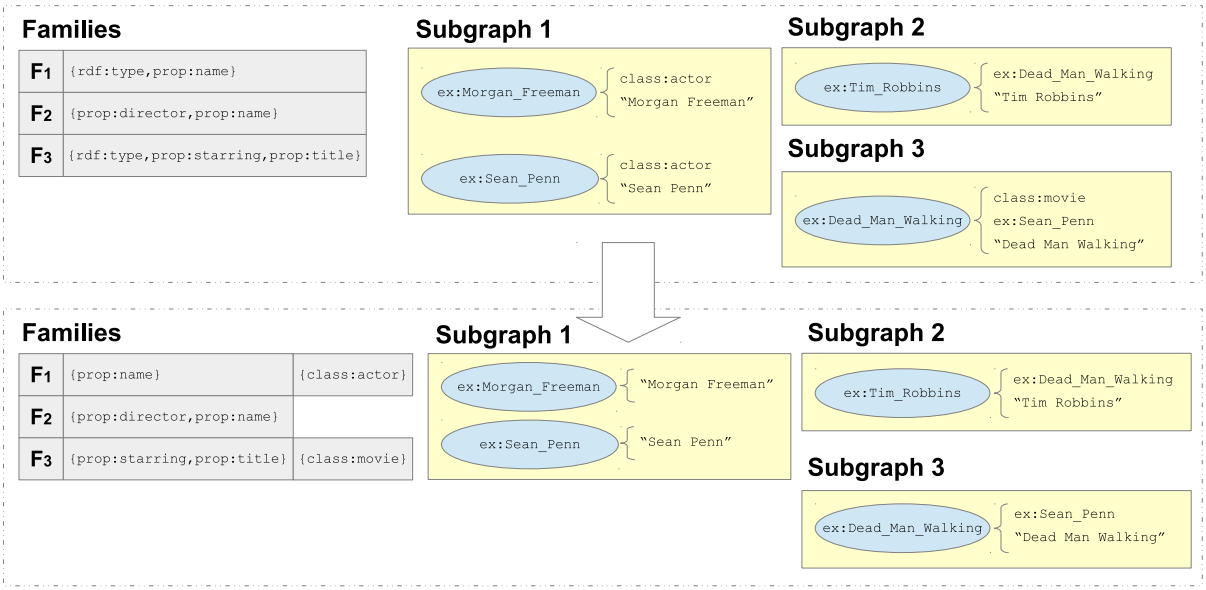

Figure 7: Integration of `rdf:type` values in predicate families.

of subjects. Then, the original RDF graph can be split into as many subgraphs as predicate families ($|F|$)*, ensuring that all subjects in a subgraph are described with the same predicates. In this way, each subject can be described as the family it belongs to and a sequence of objects (for each predicate of the family). Note that the corresponding predicates will be inferred from its predicate family. This decision will improve compression effectiveness because predicate occurrences are no longer encoded for each subject, but only as part of the corresponding families. In practice, the number of predicate families will be much lower than the number of subjects:* $|F| << |S|$.

The second RDF-Tr foundation focuses on removing redundancy in the use of `rdf:type`. This predicate provides the class of the subject being described. Although `rdf:type` is not mandatory, it is widely used in practice to categorize the information in the dataset. Consequently, the `rdf:type` predicate typically occurs in many triples.

In our previous example, both subjects in the first subgraph belongs to `class:actor`. It seems reasonable that subjects of the same class are described with the same predicates, i.e., the same predicate family. Thus, it is not necessary for the class value to be encoded for each subject in the subgraph, but to relate this value to the predicate family that describes the corresponding subgraph.

Definition 2 (Typed Predicate Family). *Let* t *be the* `rdf:type` *predicate,* F_s^t *the predicate family* F_s *where we remove* `rdf:type`*, that is* $F_s^t = \{p \mid \exists z \in O_G, p \in P_G, p \neq t, (s,p,z) \in G\}$*, and* C_s *all the class values that define a subject,* $C_s = \{o \mid (s,t,o) \in G\}$*. Formally speaking, the predicate families enriched with* `rdf:type` *values,* F'_s *can be defined as* $F'_s = \langle F_s^t, C_s \rangle$*. That is, a typed predicate family is a pair with a*

predicate family and class values, such that there exists at least one subject in that predicate family described with the given class value(s). Note that subjects can be potentially described with the same predicate family, but different class values (e.g., a director and an actor could be described with the same predicates). In this case, a predicate family can result in different typed predicate families, one for each different combination of class values related to its subjects. Note that we consider the particular case where $C_s = \emptyset$ (no class values) or $F_s^t = \emptyset$ (i.e., the original family F_s only has `rdf:type`*). Therefore, the total set of predicate families in a dataset, F', is defined as $F' = \{F'_s \mid s \in S_G\}$.*

Figure 7 shows the resulting subgraph configuration when `rdf:type` values are encoded as part of the predicate family. All class values are removed from the corresponding subject descriptions and are now linked to predicate families. For instance, the subject `<http://example.org/Morgan_Freeman>`, within the first subgraph, is explicitly described as {``Morgan Freeman''}, while the {`class:actor`} value can be inferred from the first predicate family. This establishes the principles of the second foundation of RDF-TR.

Foundation 2. *Enriching predicate families with* `rdf:type` *values allows the final serialization to discard all RDF triples involving such predicate. Thus, this decision favors compression effectiveness by considering the large number of triples using* `rdf:type` *in real-world datasets. For simplicity, we hereinafter use "predicate families" (F) to refer to predicate families enriched with* `rdf:type` *values (F'). We also consider that families are repeated among subjects, hence we hereinafter refer to the different predicate families, F_1, F_2,$\cdots$,F_z, where z is the number of different families in the dataset.*

4.2. Objects

Schema-based redundancies are often referred to the predicates used to describe subjects, but one can also find regularities in objects. RDF generally allows any predicate to be connected with any object (except for range restrictions in the definition of some predicates), but object values tend to be tightly bound to a limited number of predicates. In other words, predicate values come from a limited and well-defined range. For example, as previously explained, it would be uncommon to find ``clint@eastwood.org'' as a value for a film duration, or ``Dead Man Walking'' as the family name of a person. In fact, it is usual that object values are related to a single predicate [15].

From a structural perspective, this fact implies that *in-links* of a given object are often labelled with the same predicate, which constitutes the principle of the third foundation of RDF-TR.

Foundation 3. *The potentially large universe of object values can be divided in $|P|$ barely overlapping ranges that can be managed independently. This allows objects to be locally identified within the scope of each predicate (and not globally as is usual). Thus, local object identifiers can be encoded using fewer bits (than those used when objects are globally identified), which improves compression effectiveness.*

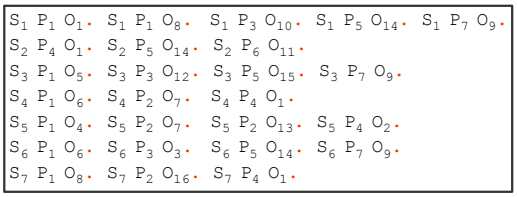

Figure 8: RDF triples used for illustrating the RDF-TR algorithm.

5. RDF-TR

RDF-TR is a preprocessing technique that reorganizes RDF triples to detect and remove redundancy at various levels. It proposes a multi-step algorithm that implements particular decisions addressing the three foundations introduced in the previous section.

In the following, we explain all these transformations (shortened to T1 to T5) on a generic example presented in Figure 8. This excerpt uses the Turtle [6] serialization, with the following remarks:

- Turtle triples are used in Figure 8, hence *(subject, predicate, object)* terms are separated by whitespaces, and triples end with a dot ('.').
- For the sake of simplicity, no concrete values are used for subject, predicate and object terms. We will refer indistinctly to S_i (similar for P_i and O_i) as the i^{th} subject, or the subject with ID i. It is worth noting that RDF-TR requires the "special" `rdf:type` predicate to be identified with the higher predicate ID (P_7, in this example).
- Finally, we consider an inner precedence relationship between subjects, predicates and objects. That is, $S_i < S_j$ if $i < j; i, j \in [1, |S|]$ (similarly for P_i and O_i in ranges $[1, |P|]$ and $[1, |O|]$, respectively).

In general, we assume that triples are sorted by *(subject, predicate, object)*, otherwise an initial transformation (referred to as T0) is needed.

T0. Subject-based reorganization. This initial step groups together all triples describing the same subject. As shown in Figure 9, this decision enables the RDF graph to be re-encoded as a forest of trees, where each subject is the root of a tree that includes all the triples in which the subject is involved. That is, triples are organized as a series of predicate-object lists (one per subject). For instance, S_1 has adjacency lists rooted by P_1, P_3, P_5, and P_7. The same four predicates are used by lists of S_3 and S_6, so the first, the third, and the sixth subjects are described using the same predicate structure. Note that red dotted lines are used, in the figure, to show triples labelled with the `rdf:type` predicate, as they will have a special treatment (see Section 5.2).

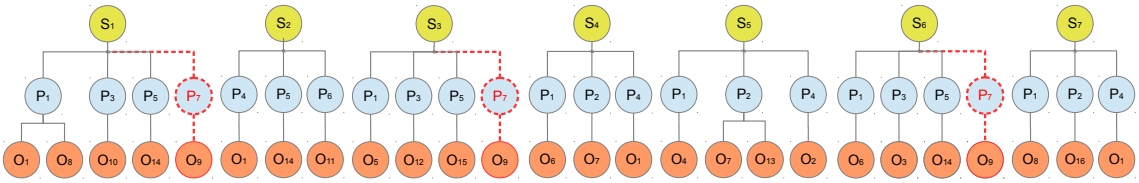

Figure 9: RDF triples organized as a forest of trees.

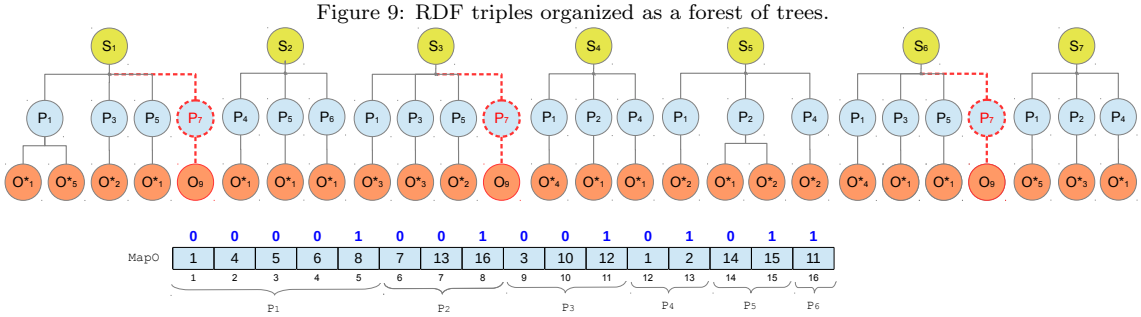

Figure 10: Object re-mapping (note that local object IDs are assigned according to the global object ID order).

5.1. Object-based transformation

Based on the results of Fernández et al [15], a particular object in an RDF dataset is often tied to a certain predicate. Under this premise, we will perform the first transformation (T1) at object level. Object identifiers will be re-coded to predicate-local IDs, as using local identifiers takes up less space than global ones (see Foundation 3).

T1. Object re-mapping. We re-map objects related to the same predicate with a new sequential identifier. These new local-IDs will be assigned in global-ID order. That is, we sort all objects of a given predicate by their (original) IDs in the dictionary, and we then assign the position of each object as its local-ID. Definition 3 formalizes this concept.

Definition 3 (Local Object ID). *Formally, we define $O_i^*|P_j$, a local object of predicate P_j, as $O_i^*|P_j = P_j[i] : P_j = \{O_k \dots O_l\}; j \in [1, |P|], \{k, l\} \in [1, |O|], k < \dots < l$. We abuse the notation to refer to a local Object ID as O_i^*, where the concrete predicate can be inferred from the context.*

For instance, in our previous excerpt, P_3 is used in 3 triples $\{$`(S1,P3,O10), (S3,P3,O12), (S6,P3,O3)`$\}$. Thus, taking into account the global order of objects, O_1^* in P_3 refers to O_3, O_2^* to O_{10}, and O_3^* to O_{13}. The same process is carried out for each predicate until all triples are rewritten with the new object identifiers. Figure 10 shows the output of this first transformation. Note that objects related to predicate `rdf:type` remain unchanged, since these triples will be treated separately (see T3 in Section 5.2).

This transformation requires the introduction of an additional *object mapping* structure (`MapO`) to obtain (during decoding, presented in Section 5.4) the original ID of a local object. As shown in Figure 10, `MapO` is implemented as an adjacency list structure that contains the original IDs of the objects related to each

predicate (except for those related to `rdf:type`). Thus, `MapO` encodes $|P-1|$ adjacency lists. As explained in Section 3.4, this structure encompasses an integer sequence, `MapO.S`, which contains the lists of object IDs, and a bitsequence, `MapO.B`, which marks with 1-bits the end of each list. This can be easily seen in Figure 10, where the predicate P_1 is related to five objects: O_1, O_4, O_5, O_6, and O_8 (note that `MapO.B[5]=1` marks the end of the list), P_2 is related to objects O_7, O_{13}, and O_{16} (`MapO.B[8]=1` marks the end of the second list), and so on.

Mapping a local object ID ($O_i^*|P_j$) to its global ID is simply implemented as `neigh(MapO,j)[i]`, i.e., the ID of the i-th direct neighbor of P_j. For instance, the global ID of $O_3^*|P_2$ can be computed as `neigh(MapO,2)[3]=16`, as the third object of predicate 2 is stored at `MapO.S[8]`$= 16$.

5.2. Predicate-based Transformations

Two predicate-based transformations are proposed to implement Foundations 1 and 2. Predicate families must first be discovered *(transformation T2)*, and then be enriched, when necessary, with `rdf:type` values *(transformation T3)*. After these transformations, general triples are re-encoded in the form of *(subject, family, object)*, and triples involving `rdf:type` are removed and represented separately.

T2. Predicate family discovering. This transformation looks for all the different combinations of predicates that are used for subject descriptions. As mentioned in Foundation 2, the different families are numbered with an autoincremental ID, hence all families are identified within the range $[1, |F|]$, where $|F|$ is the number of different families in the dataset. Thus, each subject S_i is now related to a family F_j, hence adjacency lists can be compacted by replacing (multiple) predicate occurrences with the corresponding family ID.

Figure 11 illustrates this transformation in our previous example, where three different families are discovered: $F_1 = \{P_1, P_3, P_5, P_7\}$, $F_2 = \{P_4, P_5, P_6\}$, and $F_3 = \{P_1, P_2, P_4\}$. Reconstructing the original triples from this encoding is straightforward. For instance, S_1 is related to F_1, and the last level of objects contains four lists (one per predicate):

1. The first list contains O_1^* and O_5^*, and corresponds to the first predicate in F_1, which is P_1. Thus, it encodes the triples (S_1, P_1, O_1^*) and (S_1, P_1, O_5^*).

2. The second list only includes O_2^*, and is related to the second predicate in F_1, i.e., P_3. Thus, it encodes the triple (S_1, P_3, O_2^*).

3. The third list contains O_1^*, which is related to the third predicate in F_1, i.e., P_5. Thus, it encodes the triple (S_1, P_5, O_1^*).

4. Finally, the last list is tagged with P_7, which refers to `rdf:type`. Thus, the corresponding triple will be removed from this representation (and encoded separately) in the following transformation.

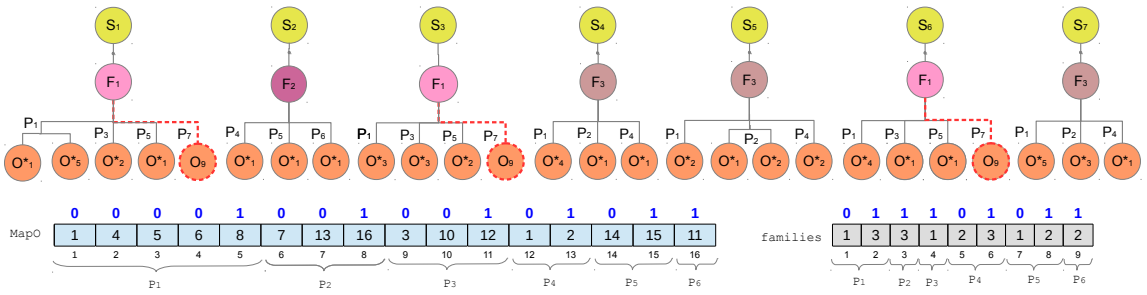

Figure 11: Predicate family discovery.

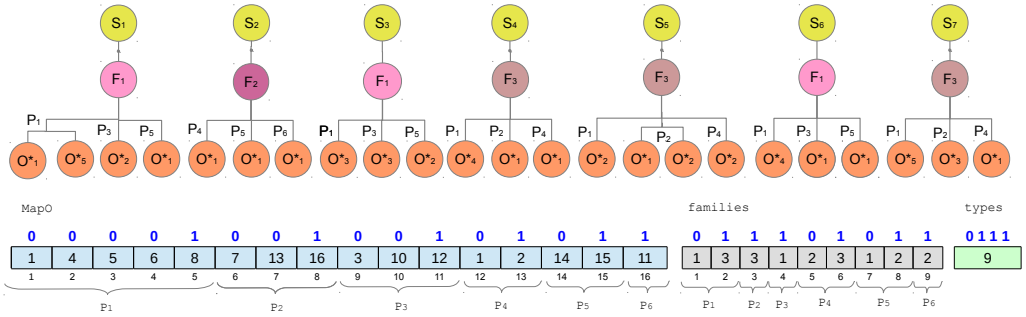

Figure 12: `rdf:type` encoding

Information about predicates and families must also be preserved as part of the encoding. A new adjacency list structure, called `families`, is used for this purpose. As shown in Figure 11, it encompasses $|P-1|$ adjacency lists, each one listing the IDs of the families in which each predicate (except for `rdf:type`) is used. For instance, the first predicate is used in two families: $\{F_1, F_3\}$, the second predicate only appears in a single family: $\{F_3\}$, and so on.

Retrieving the IDs of the families for a given predicate p is simply implemented using `neigh(families,p)`. For instance, in our example, the families in which $P4$ appears can be retrieved as `neigh(families,4)`$=\{2,3\}$, i.e., families F_2 and F_3.

T3. Encoding of `rdf:type`. This transformation processes triples with the `rdf:type` predicate, retrieving class values (i.e., the objects of these triples) and using them to *type* the corresponding predicate families. Following Definition 2, the object types are part of the predicate families, so if two subjects are related to the same initial family, but they differ in the types, two independent families will be formed. Note also that a typed family can be related to multiple types, e.g., a family with the set of predicates `prop:starring` and `prop:title` can be used to describe a subject having the general type `class:film` and the more specific type `class:Documentary`.

Figure 12 shows the resulting transformation. The typed triples (previously marked with red dotted lines) are no longer represented in the trees, as they are encoded in an additional data structure: `types`. This adjacency list preserves the IDs of the object types related to each predicate family. Note that, in this case, non-typed families are encoded as empty lists, hence `types.B` is implemented using the adjacency list

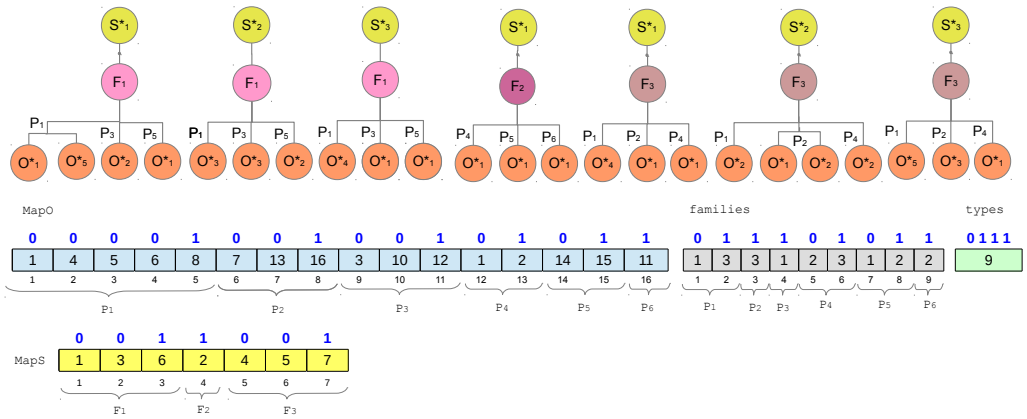

Figure 13: Subject re-mapping.

variant that allows for empty lists (see Section 3.4). For instance, in our example, `types.B=[0111]` encodes that the first family is associated with one object type, while the other families have empty lists, i.e., they are not typed.

The `types` structure is used to retrieve object types for a predicate family. For a given family f, it is easily implemented as `neigh(types,f)`, being $\emptyset$ if f is not typed. For instance, `neigh(types,1)`$= 9$, because O_9 is the class value of the first family. In contrast, `neigh(types,2)`=`neigh(types,3)`$= \emptyset$, as F_2 and F_3 are not typed.

5.3. Subject-based Transformations

As stated in Foundation 1, a subject is described by a particular family of predicates. Thus, the set of subjects described by the same family can be re-mapped as (family) local subjects. The following transformations allow local subjects to be represented and efficiently managed.

T4. Subject re-mapping. This transformation first groups subjects by the family they belong to, and then orders each group by subject ID. This rearrangement is finally used to assign a new sequential identifier for each subject within a family. Definition 4 formalizes this concept.

Definition 4 (Local Subject ID). *Formally, we define $S_i^*|F_j$, a local subject of the family F_j, as $\mathcal{S}_i^*|\mathcal{F}_j = \mathcal{F}_j[i] : \mathcal{F}_j \neq \emptyset$ and $\mathcal{F}_j = \{\mathcal{S}_k \dots \mathcal{S}_l\}; j \in [1, |\mathcal{F}|], \{k, l\} \in [1, |\mathcal{S}|], k < \dots < l$. We abuse the notation to refer to a local subject ID S_i^*, where the concrete family can be inferred from the context.*

Figure 13 shows the resulting organization on our running example, where triples are now grouped by family. As we can see, subjects have been re-encoded within the family they are related to, represented with the new local subject identifiers, S_i^*. For instance, subjects S_4, S_5 and S_7 were described by family F_3 (see Figure 12), and they are now re-mapped to S_1^*, S_2^*, S_3^*, respectively.

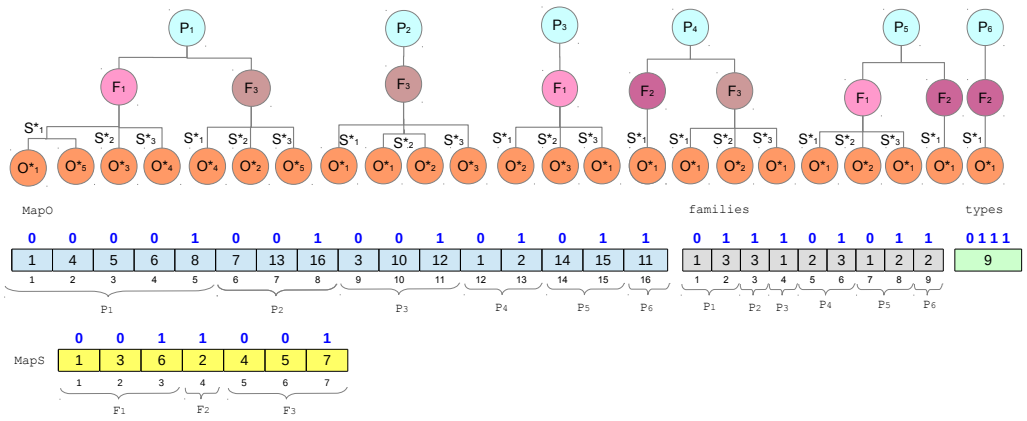

Figure 14: Triples rewritten with the RDF-TR algorithm.

A new *subject mapping* structure, (`MapS`), is required to obtain (during decoding, presented in Section 5.4) the original IDs of local subjects. `MapS` is implemented as an adjacency list structure that concatenates the original IDs of the subjects described by each predicate family. Thus, `MapS` encodes $|F|$ adjacency lists. As shown in Figure 13, the subjects S_1, S_3, and S_6 are described by the family F_1, and they are mapped to $S_1^*|F_1$, $S_2^*|F_1$, and $S_3^*|F_1$, respectively.

Mapping a local subject ID ($S_i^*|F_j$) to its global ID is simply implemented as `neigh(MapS,j)[i]`. For instance, the local subject $S_3^*|F_1$ is mapped to `neigh(MapS,1)[3]`$= 6$, i.e., the global subject S_6. Note that this structure is also used during the decoding process to retrieve all subjects described by a given family F_j. This functionality is also implemented using the `neigh` operation, accessing the whole list of directed neighbors. For instance, in our example, `neigh(MapS,1)`$= \{1, 3, 6\}$ retrieves all subjects described by F_1.

T5. Predicate Grouping. The *Subject-Family-Object* tree-shape organization from the previous transformations results in a very flat representation, as one subject is only represented by one family. Thus, the last step of our process consists of obtaining a bushy representation that can help compression and favor fast decoding. The previous representation is rearranged by predicate, obtaining *Predicate-Family-Object* trees, such as the example shown in Figure 14. Therefore, a predicate is related to several lists of objects, one per each family where the predicate is present.

This forest of trees can be represented in a more compact notation based on adjacency lists. The "abstract" representation of these adjacency data, referred to as ATR, is shown in Figure 15. ATR only needs to provide a simple operation `getObjects`, which retrieves all local object IDs given a predicate and a subject. Sections 6 and 7 describe two practical implementations of ATR on the basis of the existing `HDT` and `k`^{2}`-triples` compressors.

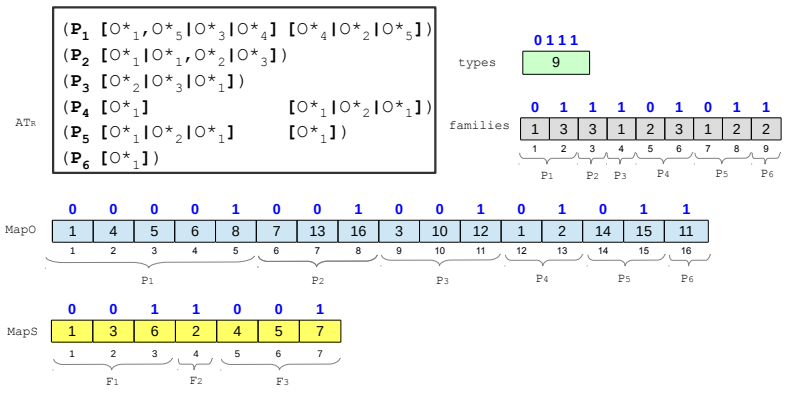

Figure 15: Predicate family based (adjacency list) encoding.

5.4. RDF-TR *Implementation and Decoding*

Figure 15 shows the final organization and structures after applying RDF-TR, which includes the compact representation of triples (ATR), and other auxiliary structures, `families`, `types`, `MapS` and `MapO`. In the following, we briefly summarize the implementation remarks shown in the previous section:

- RDF-TR focuses on the reorganization of triples in a dataset, hence it manages IDs for each subject, predicate and object term and assumes the use of a dictionary to make a bidirectional translation between terms and integer IDs (similar to most symbolic compressors).
- The mapping structures for subjects and objects, `MapS` and `MapO`, are represented as adjacency lists, which are succintly encoded using a bitsequence and an integer ID sequence (see Section 3.4). The `types` and `families` structures are similarly encoded as adjacency lists.
- The implementation of ATR (essentially, adjacency data) may vary, depending on the internal structure of the RDF syntactic compressor that uses RDF-TR (as shown in Sections 6 and 7).

Algorithm 5 illustrates the decoding process that retrieves the original triples from the RDF-TR-based encoding. It implements a multi-nested-loop algorithm that iterates through all predicates (Line `1`), except for `rdf:type`. For each predicate, we obtain the *list of families* in which the predicate is present (Line `3`), and iterate over them (Line `4`). For each family, we first obtain its ID (Line `5`), and then use it to retrieve the *list of object types* (or $\emptyset$, if it is a non-typed family), and the list of subjects related to this family (Lines `6` and `7`). These subjects are then also iterated (Line `8`). For each subject, we use ATR to retrieve the list of objects related to the current predicate and subject (Line `10`), referred to as $\mathcal{O}_s$. At this point, we have retrieved all IDs, but they must be mapped from their local encoding to their original IDs in the dictionary. In Line `9`, the local subject ID is mapped to its global one, and global object IDs are obtained in Line `13` (within a loop that iterates over all objects in $\mathcal{O}_s$). Finally, in Line `14`, the corresponding triple is emitted.

Algorithm 5: Decoding algorithm.

```
1  for predicate ← 1 to |P − 1| do
2      ptrSubject ← 1;
3      F_p ← neigh(families, predicate);
4      for f ← 1 to |F_p| do
5          family ← F_p[f];
6          T_f ← neigh(types, family);
7          S_f ← neigh(MapS, family);
8          for s ← 1 to |S_f| do
9              subject ← S_f[s];
10             O_s ← ATR.getObjects(predicate, ptrSubject);
11             ptrSubject ← ptrSubject + 1;
12             for o ← 1 to |O_s| do
13                 object ← neigh(MapO, predicate)[O_s[o]];
14                 newtriple(subject, predicate, object);
15             if T_f ≠ ∅ then
16                 for t ← 1 to |T_f| do
17                     newtriple(subject, rdf:type, T_f[t]);
```

Note that Lines 15 to 17 are only executed for typed families. In this case, object types are iterated and new typed triples are emitted for the corresponding subject and object type.

In the following sections, we show how RDF-TR can be integrated into existing compressors, which assume the responsibility of implementing ATR.

6. HDT++

The integration of HDT and RDF-TR is referred to as HDT++. We first provide an overview of HDT, with particular attention to triples encoding. Then, we show how RDF-TR can be plugged into HDT.

6.1. HDT

HDT [17] was a pioneer in RDF binary serialization, specifically focused on optimizing storage and transmission costs over a network, as well as fast retrieval on compressed space. It is specifically tailored to potentially large datasets, achieving similar compression ratios to general techniques such as *gzip*. As summarized in Section 2.2, RDF is encoded using three logical components: *Header* (i.e., metadata), *Dictionary*

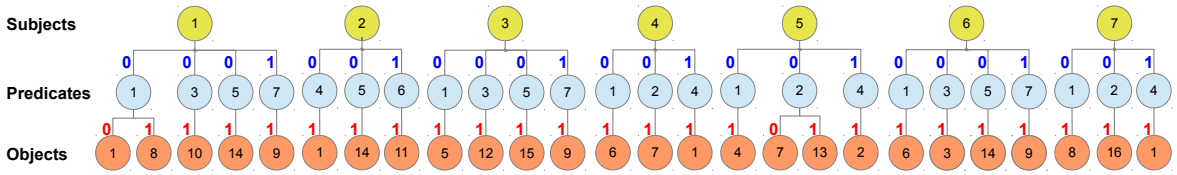

Figure 16: Forest of trees modeling ID triples in HDT.

Bp	0	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	
Sp	1	3	5	7	4	5	6	1	3	5	7	1	2	4	1	2	4	1	3	5	7	1	2	4	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Bo	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
So	1	8	10	14	9	1	14	11	5	12	15	9	6	7	1	4	7	13	2	6	3	14	9	8	16	1

Figure 17: *BitmapTriples* implementation.

(the aforementioned mapping between string terms and IDs), and *Triples* (the graph of IDs). We focus on the Triples component hereinafter, as RDF-TR is focused on triples organization.

Figure 16 shows the organization of HDT *Triples* over the original triples of our running example (see Figure 8). Triples are organized as a forest of trees, one per different subject in the dataset: the root of each tree encodes the subject, the second level encodes the predicates related to the subject and the leaves encode the adjacency lists of all objects related to each predicate within its root (subject) scope. Note that the IDs of subjects, the predicates related to each subject, and the objects related to a subject-predicate pair, are in increasing order.

HDT encodes triple IDs using the so-called *Bitmap Triples* structure. As shown in Figure 17, this approach consists of two coordinated adjacency lists that respectively encode predicate and object adjacency information for each subject. On the one hand, predicate IDs are listed in the integer sequence `Sp`, delimiting each subject list with 1-bits in `Bp`. Thus, the i-th 1-bit marks the end of the list corresponding to subject i. On the other hand, `So` contains the integer sequence of object IDs, corresponding to the leaves of the forest, where a 1-bit in the bitsequence `Bo` marks the end of the objects related to the corresponding subject-predicate pair.

6.2. Plugging RDF-TR *into HDT*

Plugging RDF-TR into HDT is straightforward. RDF-TR assumes that the HDT dictionary and the corresponding forest of trees (represented in Figure 16) has been created. Then, leaving aside the dictionary compression, which is performed as in HDT, the novel HDT++ process starts by traversing the obtained forest of trees and performing transformations T1, T2, T3, T4, and T5 to reorganize triples and build the data structures described previously. Finally, the abstract ATR structure is implemented as follows.

HDT++ encodes ATR as an array of $|P-1|$ adjacency lists (one per predicate, except for `rdf:type`), referred to as P_s. Each list is encoded as in *BitmapTriples*, i.e., using an integer sequence of IDs, S_o,

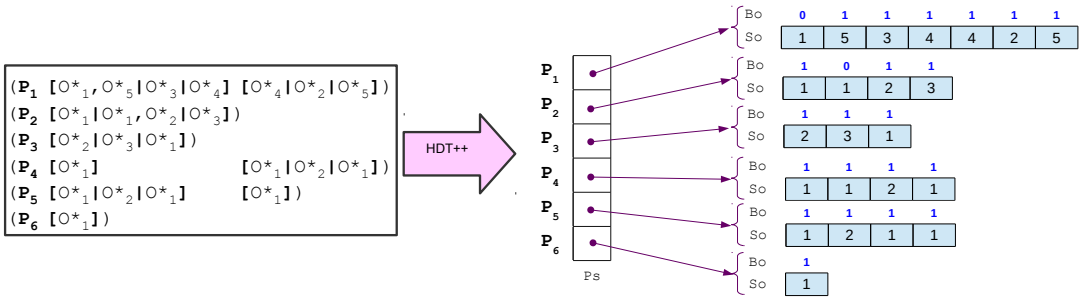

Figure 18: HDT++ implementation of ATR.

Algorithm 6: HDT++: getObjects(predicate, ptrSubject).

```
1 return neigh(P_s[predicate], ptrSubject);
```

and its aligned bitsequence, B_o. This simple but effective approach allows adjacency lists to be managed independently, hence each one can be encoded according to the features of its corresponding predicate.

Algorithm 6 shows the implementation of the `getObjects` method in HDT++. Recall that this operation is used in the decoding process (see Line `10` of Algorithm 5) to get the objects related to a given predicate and subject. Note that, in practice, the decoding algorithm does not iterate on the subject ID, but the position (i.e., '`ptrSubject`' in the code) where its object list is encoded for a given predicate, as explained in Section 5.4. This algorithm simply performs the `neigh` operation over the corresponding adjacency list structure, stored at $P_s[predicate]$, retrieving all neighbors encoded in the list of *ptrSubject*.

Finally, note that the remaining data structures (`MapO, MapS, types` and `families`) are built in HDT++ following the same aforementioned procedures[7] (see Section 5).

7. K²-triples++

We refer to k²-triples++ as the integration of k²-triples and RDF-TR. As in the previous section, we first introduce the foundations of k²-triples and then we describe the RDF-TR integration.

7.1. k²-triples

Similarly to HDT, k²-triples [1] performs dictionary compression before encoding the resulting ID-graph. It is worth noting that both approaches implement the same scheme for dictionary compression. k²-triples takes advantage of the low number of predicates used in an RDF dataset and partitions it vertically. That is, k²-triples performs a predicate-based partition of the dataset into disjoint subsets of subject-object pairs, and then these subsets are highly compressed as binary matrices (i.e., a 1-bit marks that the corresponding

[7]These structures are not represented in Figure 18 for simplicity, but their configuration is the same as in Figure 15.

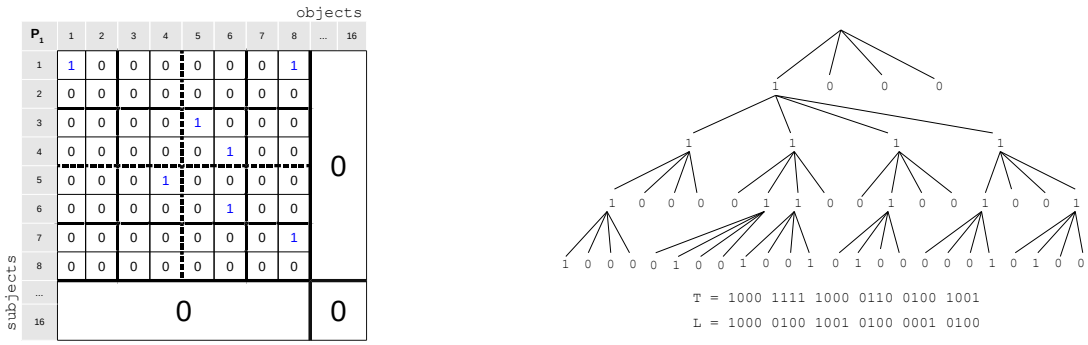

Figure 19: Vertical-Partitioning on k^{2}triples (k=2) for predicate P_1.

triple exists in the dataset) using k^2-trees [9]. The size of these matrices will be $m \times m$, where m is the minimum power of k that is greater than $max(|S|, |O|)$.

Continuing with the triples given in our running example, Figure 19 illustrates the resulting k^2-tree for the first predicate (with ID 1), i.e., it encodes all triples *(s, 1, o)*, where s and o are the IDs of the corresponding subjects and objects. The conceptual 16×16 matrix is illustrated on the left hand side (note that, in this example $|S| = 7$ and $|O| = 15$), modelling subjects by rows and objects by columns. We consider $k = 2$, hence each level is divided into k$^2 = 4$ submatrices. Recall that $(i, j) = 1$ means that there is a triple, in which the subject i (rows) is related to object j (columns) through the predicate 1.

The right hand side of Figure 19 depicts the conceptual tree and the final configuration of bitsequences T and L, which effectively encode the k^2-tree. As explained in Section 3.4, all the aforementioned graph operations (including `neigh`) are efficiently provided by the k^2-tree using `rank` and `select`.

7.2. Plugging RDF-Tr *into k^2-triples*

Transforming k^2-triples into `k2-triples++` involves a similar process to that described for `HDT++`. Thus, the dictionary and the ID-graph are first obtained, and the RDF-Tr transformations are then performed to obtain `MapO, MapS, types, families` and the ATr structure, which is represented as follows. As in k^2-triples, ATr is vertically partitioned by predicate, i.e., (subject, object) pairs are encoded in the k^2-tree corresponding to their related predicate(s). It is worth noting that, in `k2-triples++`, the size of each adjacency matrix depends exclusively on the number of subjects and objects related to the corresponding predicate (instead of the total number of subjects and objects in the dataset).

Figure 20 illustrates how k^2-triples++ implements ATr. Note that the largest matrix (of size 8×8) is modelled for predicate 1, as it is related to 6 different subjects and 5 different objects. In contrast, the matrix for predicate 6 is 1×1, as this predicate is just present in a single triple.

The implementation of ATr in `k2-triples++` provides the `getObjects` operation, required for decoding. As shown in Algorithm 7, we also make use of the `neigh` operation of the k^2-tree to process the row *ptrSubject* and retrieve the corresponding objects.

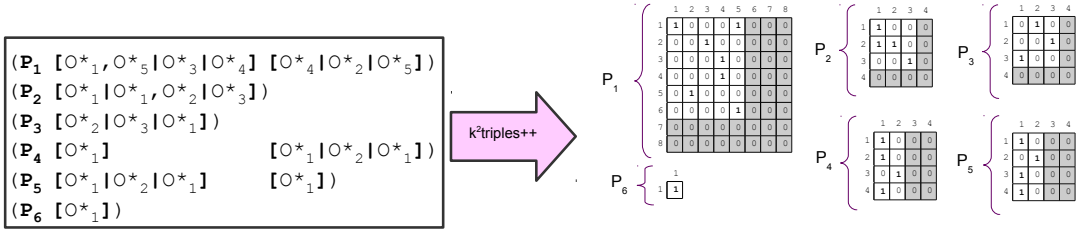

Figure 20: K^2triples++ implementation of ATR

Algorithm 7: k^2triples++: getObjects(predicate, ptrSubject).

```
1 return neigh(k^2 − tree[predicate], ptrSubject);
```

8. Experimental Evaluation

This section evaluates the performance of RDF-TR in real-world RDF datasets. We first provide concrete details of our prototype (Section 8.1) and then describe the evaluation corpus (Section 8.2). We analyze the results of the evaluation in Section 8.3 and Section 8.4 provides a final discussion of our results.

8.1. Practical RDF-TR *Implementation*

Our RDF-TR prototype[8] is built in C++11, making extensive use of the *Succinct Data Structure Library*[9] (SDSL). This library implements different compact data structures and provides rich functionality over these structures[10]. Thus, our prototype implements all the auxiliary RDF-TR structures, `types`, `families`, `MapO` and `MapS` on SDSL functionalities:

- `Types` is serialized as an adjacency list: the sequence S is implemented as an SDSL `int_vector`, which uses $\log_2(|O|)$ bits per ID, and the bitsequence B is built over a plain `bit_vector`. Note that a variant of the aforementioned Clark's structure[11] [10] is loaded to provide efficient `select` support.
- `Families` is serialized as an adjacency list, but it is loaded as a *vector of vectors* to speed up data access. Each secondary vector is implemented as an independent SDSL `int_vector`, which encodes each ID using a number of bits proportional to the greatest family ID: F' related to the given predicate; i.e. $\log_2(F') \leq \log_2(|F|)$ bits per ID.

[8] The code of the prototype is publicly available at `https://github.com/antonioillera/HDTpp-src`
[9] `https://github.com/simongog/sdsl-lite`
[10] A brief summary of the structures and operations is available at `http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf`
[11] `https://github.com/simongog/sdsl-lite/blob/master/include/sdsl/select_support_mcl.hpp`

- `MapO` is also serialized as an adjacency list, but it is loaded as a *vector of vectors* to optimize the memory footprint. Note that the list of objects related to each predicate can be very large, so bitsequences use more bits than the required pointers. Besides, object lists can be compressed, saving additional space. Thus, we implement secondary vectors using the compressed SDSL `enc_vector`. `First, we perform gap-encoding over the elements of each list and store samples each` *`t_dens`* `positions`. Then, the resulting representation is compressed using Elias-Delta. Note that *t_dens* is a user-defined value, so it is possible to tune this parameter for faster decompression, or greater compression (at the expense of speed). Thus, in the analysis section, we will evaluate how the variation of this parameter affects the decompression time and space of some datasets.

- `MapS` is loaded similarly to `MapO` in order to exploit the fact that the lists of subjects for predicate families are also large, and these are effectively compressed using gap-encoding and Elias-Delta.

Finally, we provide two concrete implementations of ATR leading to the `HDT++` and `k`^{2}`-triples++` compressors (as explained in Sections 6 and 7):

- `HDT++` serializes $|P-1|$ adjacency lists and loads them into an array for decoding purposes. Note that the `int_vector` of each adjacency list is configured to use $log_2(|O^p|)$ bits per ID, where $|O^p|$ is the number of different objects within the range of the predicate p.

- `k`^{2}`-triples++` serializes $|P-1|$ k^2-trees, each one configured according to the number of subjects and objects related to the corresponding predicate. We use $k=2$, as in the original k^2-triples approach [1].

8.2. Evaluation Corpus: Description and Statistics

Our evaluation considers five real-world RDF datasets: `dblp` provides open bibliographic information on major computer science journals and proceedings; `dbtune` includes music-related structured data; `us census` provides census data from the U.S.; `linkedgeodata` uses the information collected by the OpenStreetMap project and makes it available as an RDF knowledge base according to the Linked Data principles; and `dbpedia` is an RDF conversion of Wikipedia (mostly on the infobox information).

Table 1 reports the main statistics of these datasets, namely, the *number of triples*, and the *number of total subjects, predicates*, and *objects*, ($|S|, |P|$, and $|O|$, respectively). Furthermore, Table 2 reports relevant statistics for RDF-TR. We show, for each dataset, the *number of families* ($|F|$), the *number of different types* used in the dataset, the number of *typed-families* (recall that a typed-family is a family that is defined by at least one type), the number of *typed-triples* (i.e., triples involving `rdf:type`), as well as the maximum value of local object identifiers (i.e., the maximum number of objects in the range of a particular predicate).

A first analysis of these statistics shows that `linkedgeodata` and `dbpedia` are the less-structured datasets, inasmuch as the number of families is ≈ 24 times the number of predicates in `linkedgeodata` and ≈ 50 times

Dataset	#triples	$\|S\|$	$\|P\|$	$\|O\|$
dblp	55,586,971	3,591,091	27	25,154,979
dbtune	58,920,361	12,401,228	394	14,264,221
us census	149,182,415	23,904,658	429	23,996,813
linkedgeodata	271,180,352	51,916,995	18,272	121,749,861
dbpedia	837,257,959	113,986,155	60,264	221,623,898

Table 1: Main statistics of the evaluation corpus.

Dataset	$\|F\|$	#types	#typed-families	#typed-triples	Max local-obj
dblp	283	14	283	5,475,762	6,428,355
dbtune	1,047	64	866	12,340,116	2,254,960
us census	106	0	0	0	1,242,683
linkedgeodata	441,922	1,081	440,035	81,261,427	38,826,195
dbpedia	2,969,486	370,069	2,811,839	92,725,995	40,325,707

Table 2: Statistics related to RDF-Tr.

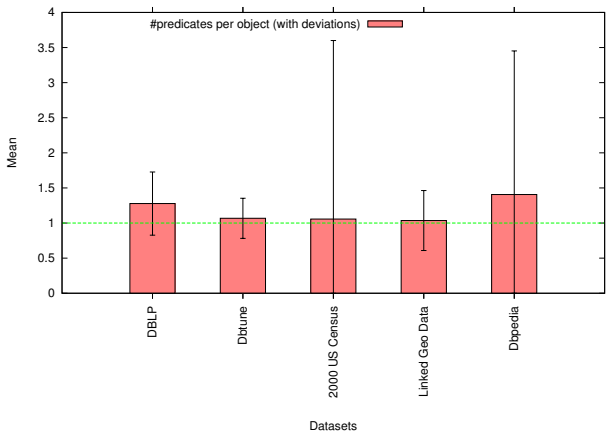

Figure 21: Number of predicates per object (mean and standard deviation).

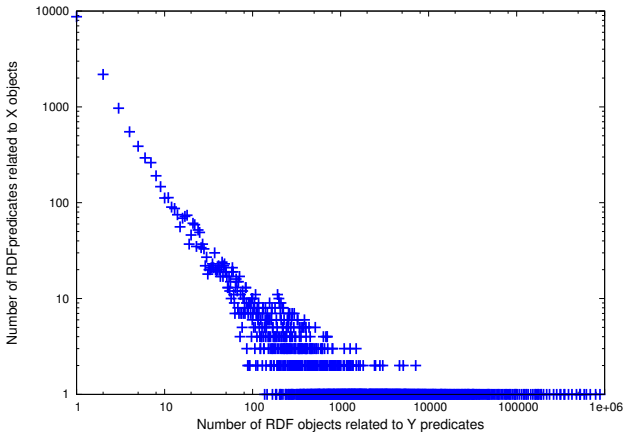

Figure 22: Distribution of RDF objects per predicate in `linkedgeodata`.

in the case of `dbpedia`. Despite their low structural level, it is important to note that the number of detected families is small compared to the possible combinations of relationships between subjects and predicates.

Conversely, `dbtune` and `dblp` are structured datasets, since the number of families is ≈ 2.5 and ≈ 10.5 times the number of their predicates, respectively. Finally, `us census` is a clear example of a highly-structured dataset because the number of families is even less than the number of predicates.

The use of types is denoted by *#types*, *#typed-families* and *#typed-triples* columns in Table 2. A comparison of *#typed-families* with the total number of families shows that most families are actually typed (except for `us census`, which does not use types). In other words, although the predicate `rdf:type` is optional in a dataset, it is actually present in most subject descriptions. In this regard, the *#typed-triples* column shows that typed datasets include a high number of triples involving `rdf:type`. For instance, `linkedgeodata has more than 81 million` typed triples, which corresponds to almost 30% of its total triples, while in the rest of the typed datasets, 10-20% of the triples are typed.

Figure 21 extends these statistics and represents the average number of predicates per object. As expected (see Section 4.2), we can observe that the number of predicates per object is very close to 1, even in the less structured datasets. In turn, Figure 22 shows the inverse relation, i.e., the number of objects per predicate, in `linkedgeodata`. In general, all datasets show a skewed distribution: most predicates are related to few objects, while there is a small number of predicates related to many objects.

All these features suggest that a typed family encoding, such as RDF-Tr, may be more effective because it groups predicates and types, thus preventing unnecessary repetitions, and it encodes objects by predicate, thus minimizing their ID lengths.

8.3. RDF-Tr *Analysis*

This section analyzes the experimental results when applying RDF-Tr to the well-known HDT and k^2-triples RDF syntactic compressors, leading to `HDT++` and `k`^{2}`-triples++` respectively. Experiments were performed in a -commodity server- Intel Xeon E5645@2.4GHz, 96GB DDR3@1066Mhz. Reported (elapsed) times are the average of five independent executions. We report in-memory spaces of the encodings (including the necessary structures to decode the serializations), disregarding the dictionary space (as all of the evaluated techniques make use of the same dictionary).

For each dataset in our evaluation setup, Table 3 shows the triples encoding size and decompression time of the original HDT and k^2-triples compressors, as well as the resulting size after applying RDF-Tr i.e., `HDT++` and `k`^{2}`-triples++` respectively. For this experiment, we fix a value of *t_dens* such that the decompression time of the original serialization is similar to the decompression time of its improved serialization with RDF-Tr. We evaluate different *t_dens* tradeoffs below.

The results in Table 3 show that, with similar decompression times, `HDT++` and `k`^{2}`-triples++` are able to significantly reduce the space requirements of their HDT and k^2-triples counterparts. The improvement of the RDF-Tr technique in `HDT++` results in 37% space savings in `dblp` (*t_dens*=128), ≈50% savings in `dbtune` (*t_dens*=32), `US census` (*t_dens*=16) and `linkedgeodata` (*t_dens*=16), and 30% in `dbpedia` (*t_dens*=16).

	HDT		HDT++		k^{2}triples		k^2-triples++	
dataset	**size (MB)**	**time (μs)**	**size (MB)**	**time (μs)**	**size (MB)**	**time (μs)**	**size (MB)**	**time (μs)**
dblp	203.19	0.0614	**127.03**	**0.0557**	99.85	0.2292	**43.71**	**0.1456**
dbtune	242.05	0.0835	**112.75**	**0.0810**	152.38	0.3309	**125.95**	**0.3302**
us census	649.22	0.0892	**323.24**	**0.0792**	347.05	0.3030	**195.46**	**0.2409**
linkedgeodata	1,446.19	0.0867	**646.17**	**0.0667**	541.28	**0.2394**	**525.39**	0.2688
dbpedia	4,152.62	**0.0639**	**2,901.12**	0.0674	2,208.40	0.2982	**1,326.74**	**0.2628**

Table 3: Compression (size) and decompression (time per triple) results.

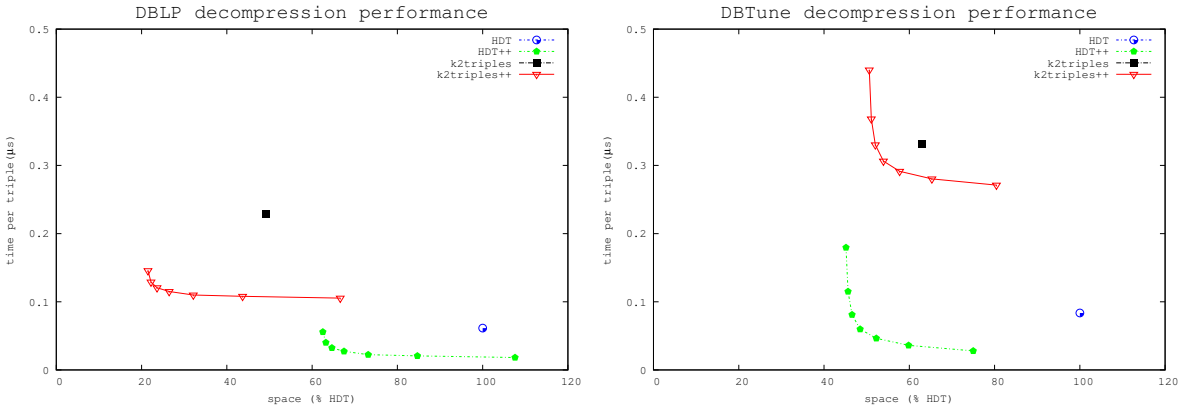

Figure 23: dblp and dbtune tradeoffs

k^2-triples++ also achieves important compression improvements over k^2-triples, with 56% space savings in dblp (t_dens=128), 17% in dbtune (t_dens=32), 45% in US Census (t_dens=32) and 30% in dbpedia (t_dens=128). In linkedgeodata, the space improvement is negligible (3%). In this case, the dataset has many different terms with respect to the number of triples (see Table 1), i.e., elements are hardly reused and less redundancies in the triples can be found. Note also that the matrices generated by k^2-triples++ are generally smaller than the k^2-triples ones because local object IDs are smaller than global object IDs. Table 1 shows the comparison between the total number of objects and the maximum local ID, which are the reference values to generate the matrices by k^2-triples and k^2-triples++ respectively.

Finally, in order to inspect potential space/time tradeoffs, Figures 23-25 evaluate different t_dens values in HDT++ and k^{2}triples++. The x-axis reflects the space given as a percentage over the size of HDT[12]. The decompression time per triple (in microseconds) is represented in the vertical axis. For simplicity, the same t_dens values have been applied to MapS and MapO. Results show how HDT++ and k^{2}triples++ can be adapted to particular scenarios. For instance, in the case of dbpedia (Figure 25), HDT++ can be tuned to take only 60% of the original (already compressed) HDT size, at the cost of additional decompression time. Note that the tradeoffs depend on the data distribution. For example, dblp (Figure 23) is a very structured

[12]We take HDT as the baseline as it is a W3C Member submission, i.e., a *de-facto* standard for RDF compression.

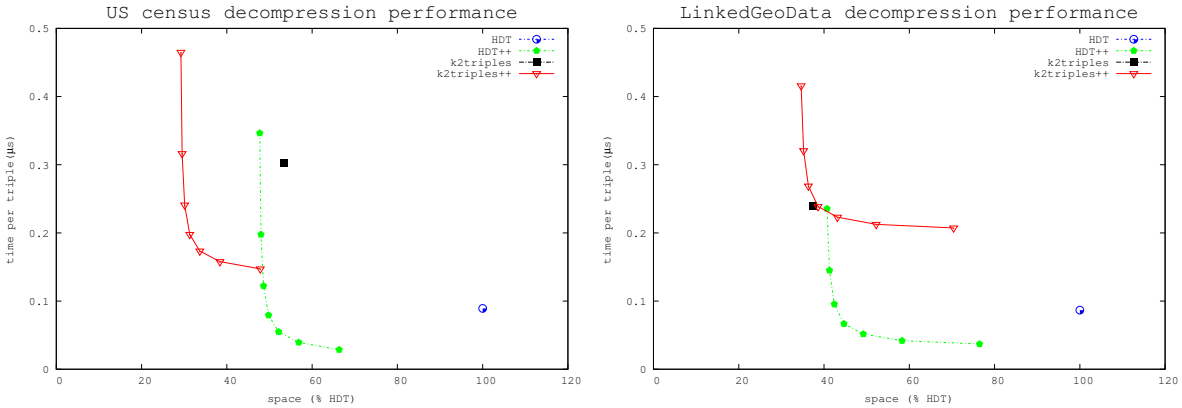

Figure 24: US census and linkedgeodata tradeoffs

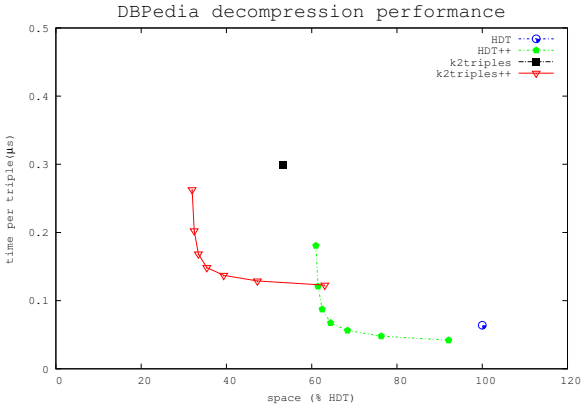

Figure 25: dbpedia tradeoffs

dataset and decompression times are not significantly degraded at more aggressive *t_dens* values.

8.4. Discussion

Our experimental results show that RDF-TR can leverage structural redundancies and achieve large space saving (approx 50% overall) as a preprocessing technique of both `HDT` and k^2`-triples` compressors.

In general, as expected, RDF-TR takes advantage of highly-structured datasets (i.e., datasets with a lower number of families). That is, `HDT++` and `k`^{2}`-triples++` achieve better compression ratios in the more structured datasets, such as `dbtune`, `dblp` and `us census`. Nonetheless, with the aforementioned exception of `linkedgeodata` in `k`^{2}`-triples++`, results also show important space savings in weakly-structured datasets such as `dbpedia`, both in `HDT++` and `k`^{2}`-triples++`.

Besides the level of structuredness of a dataset, a detailed analysis of the results and the characteristics of the datasets shows the following correlations:

- The compression ratio of RDF-TR is positively affected by the number of typed triples in the dataset (see column *#typed-triples* in Table 2). As shown in Foundation 2 and the corresponding T3 trans-

formation, RDF-Tr encodes the values of `rdf:type` within predicate families, avoiding unnecessary repetitions across subjects. For instance, as noted before, `linkedgeodata` has more than 80 million typed triples, which amounts to an impressive 30% of the total triples. This results in the reported good compression ratios of RDF-Tr, in spite of its weak structure (with more than 400K families).

- The compression ratio of RDF-Tr is negatively affected by a high proportion of RDF objects over the total number of triples (see Table 1), but positively affected by a skewed distribution of the number of objects related to each predicate (see Figure 22). Thus, excluding the auxiliary structures (i.e., `families`, `types`, `MapS` and `MapO`), the main burden of the representation lies in the encoding of ATr. Irrespective of the concrete RDF syntactic compressor that integrates RDF-Tr, given Foundation 3 and the object remapping in the transformation T1, ATr uses predicate-local IDs (i.e., sequential identifiers per predicate). Thus, the smaller the number of objects per predicate, the shorter the IDs and the smaller the space they take up. In turn, an overall large number of objects with respect to the total number of triples (e.g., in `dblp` or `linkedgeodata`) results in some large object lists, and thus large IDs (see column *Max local-obj* in Table 2), where the effect of the transformation is less remarkable.

Finally, while we show that the decompression time is not affected by RDF-Tr, it is also important to consider the time that RDF-Tr takes, in practice, to perform all the transformations described in Section 5. This time is represented on the Y-axis (logarithmic scale) of Figure 26 and is dependent on two factors, the number of triples of the dataset (X-axis) and the number of predicate families that make it up. This last dimension is depicted by the size of the bubble of each dataset (in logarithmic scale). As expected, the one-time RDF-Tr organization mainly depends on the number of triples in the dataset (as we scan all of them to discover the families), with a relative influence on its number of families (as we need to construct all RDF-Tr structures based on them). In particular, RDF-Tr takes only a few seconds for `dblp`, `dbtune`, and `uscensus`, all of them with few families and a relatively small number of triples, and ≈11 minutes for the weakly structured `linkedgeodata` dataset, with almost 300m triples and more than 400K families. As a corner case, `dbpedia` pays the price of almost 1B triples and 3M families, requiring a one-time processing of several hours. Exploring optimized construction techniques for such extremely unstructured datasets, e.g., exploiting parallelism to iterate triples and building families and RDF-Tr structures, is considered for future work.

9. Conclusions and Future Work

This paper presents RDF-Tr, a preprocessing technique that reorganizes RDF triples to leverage inherent structural redundancies. We first describe the foundations of two types of schema-based redundancies underlying RDF: predicate families are massively repeated for general and typed subjects, and objects are

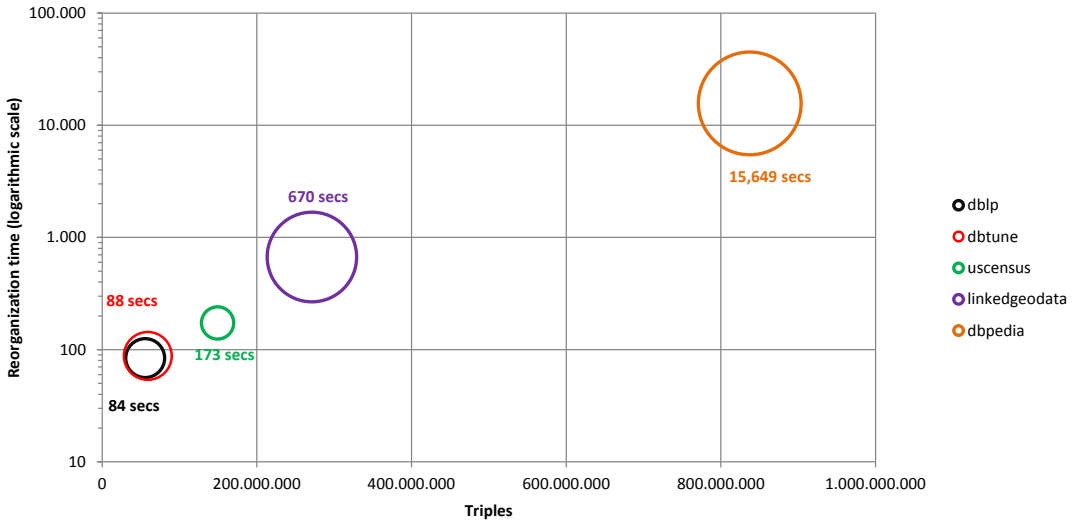

Figure 26: Datasets reorganization time.

often related to just one predicate. Then, we provide the required RDF-Tr transformations and additional structures to efficiently compress RDF data.

RDF-Tr has been applied to HDT and k^2-triples, two of the most commonly used state-of-the-art compressors. These techniques have been evaluated on real-world RDF datasets, considering different domains and structuredness. Our results show that the resultant `HDT++` and k^2`-triples++` compressors save up to half the space of their counterparts, with similar decompression times. In addition, the final configuration can be tuned to explore different space/time tradeoffs.

Our current work focuses on exploiting the RDF-Tr organization to additionally provide fast retrieval on compressed space. In particular, we work on implementing SPARQL triple pattern retrieval, partially reusing the HDT and k^2-triples functionality. In turn, both `HDT++` and k^2`-triples++` should be currently loaded entirely in main memory in order to be consumed. The adaptation of RDF data repositories for these compressed models is a challenge to face in the near future. In addition, we are also exploring how to use parallelism to practically optimize the construction of RDF-Tr structures.

Finally, the application of our foundations (i.e., heuristics) to uncover redundancies that can be further captured by RDF compression techniques sets the stage for the application of further uncovered transformations. Our future work considers both using other implicit structural similarity patterns (e.g., looking at the structure of adjacent nodes in the RDF graph [27]), as well as making use of explicitly declared constraints or regularities in the data (e.g., expressed with SHACL [26] or ShEx [4]).

Acknowledgement

This paper is funded by MINECO-AEI/FEDER-UE [Datos 4.0: TIN2016-78011-C4-1-R], the EUs Horizon 2020 research and innovation programme: grant 731601 (SPECIAL), the Austrian Research Promotion Agency's (FFG) program "ICT of the Future": grant 861213 (CitySPIN).

References

[1] S. Álvarez-García, N. Brisaboa, J.D. Fernández, M.A. Martínez-Prieto, and G. Navarro. Compressed Vertical Partitioning for Efficient RDF Management. *Knowledge and Information Systems*, 44(2):439–474, 2014.

[2] M. Atre, V. Chaoji, M.J. Zaki, and J.A. Hendler. Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 41–50, 2010.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, and Z. Ives. DBpedia: A Nucleus for a Web of Open Data. In *Procedings of the International Semantic Web Conference (ISWC)*, pages 11–15, 2007.

[4] T. Baker and E. Prudhommeaux. Shape Expressions (ShEx) Primer. *Draft Community Group Report 14 July 2017*, 2017.

[5] D. Beckett. *RDF 1.1 N-Triples*. W3C Recommendation, 2014. https://www.w3.org/TR/2014/REC-n-triples-20140225/.

[6] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. *RDF 1.1 Turtle*. W3C Recommendation, 2014. https://www.w3.org/TR/2014/REC-turtle-20140225/.

[7] P. A. Bonatti, M. Cochez, S. Decker, A. Polleres, and V. Presutti, editors. *Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web*, Schloss Dagstuhl, Germany, September 2018. To appear, http://polleres.net/bona-etal-DagstuhlReport18371.pdf.

[8] N. Brisaboa, A. Cerdeira-Pena, Fari na, and G. Navarro. A compact rdf store using suffix arrays. In *Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 103–115, 2015.

[9] N. Brisaboa, S. Ladra, and G. Navarro. Compact Representation of Web Graphs with Extended Functionality. *Information Systems*, 39(1):152–174, 2014.

[10] David Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, 1997.

[11] O. Curé, G. Blin, D. Revuz, and D.C. Faye. WaterFowl: A Compact, Self-indexed and Inference-Enabled Immutable RDF Store. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 302–316, 2014.

[12] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the ACM SIGMOD International Conference on Management of data*, pages 145–156, 2011.

[13] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, 1975.

[14] N.L. Elzein, M.A. Majid, I.B. Targio Hashem, I. Yaqoob, F.A. Alaba, and M. Imran. Managing Big RDF Data in Clouds: Challenges, Opportunities, and Solutions. *Sustainable Cities and Society*, pages 375–386, 2018.

[15] J. D. Fernández, M. A. Martínez-Prieto, P. de la Fuente Redondo, and C. Gutiérrez. Characterizing RDF Datasets. *Journal of Information Science*, 44(2):203–229, 2018.

[16] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, and A. Polleres. *Binary RDF Representation for Publication and Exchange (HDT)*. W3C Member Submission, 2011. `http://www.w3.org/Submission/HDT/`.

[17] 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.

[18] T. Guang, J. Gu, and L. Huang. Detect Redundant RDF Data by Rules. In *Proceedings of the Database Systems for Advanced Applications (DASFAA) International Workshops*, page 362368, 2016.

[19] B. Heitmann and C. Haye. SemStim at the LOD-RecSys 2014 Challenge. In *Proceedings of Semantic Web Evaluation Challenge (SemWebEval)*, pages 170–175, 2014.

[20] A. Hernández-Illera, M.A. Martínez-Prieto, and J.D. Fernández. Serializing RDF in Compressed Space. In *Proceedings of the Data Compression Conference (DCC)*, pages 363–372, 2015.

[21] L. Iannone, I. Palmisano, and D. Redavid. Optimizing RDF Storage Removing Redundancies: An Algorithm. In *Procedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE)*, pages 732–742, 2005.

[22] D. Janke, S. Staab, and M. Thimm. Impact Analysis of Data Placement Strategies on Query Efforts in Distributed RDF Stores. *Journal of Web Semantics*, 50:21–48, 2018.

[23] A. Joshi, P. Hitzler, and G. Dong. Logical Linked Data Compression. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 170–184, 2013.

[24] A. Joshi, P. Hitzler, and G. Dong. Alignment Aware Linked Data Compression. In *Proceedings of the Joint International Conference on Semantic Technology (JIST)*, pages 73–81, 2016.

[25] M. R. Kamdar, T. Tudorache, and M. A. Musen. A systematic analysis of term reuse and term overlap across biomedical ontologies. *Semantic Web*, 8(6):853–871, 2017.

[26] H. Knublauch and D. Kontokostas. Shapes constraint language (SHACL). *W3C Recommendation*, 2017.

[27] P. Maillot and C. Bobed. Measuring structural similarity between rdf graphs. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1960–1967. ACM, 2018.

[28] S. Maneth and F. Peternek. Grammar-based graph compression. *"Information Systems"*, 76:19–45, 2018.

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

[30] M. A. Martínez-Prieto, J. D. Fernández, A. Hernández-Illera, and C. Gutiérrez. Rdf compression. In Sherif Sakr and Albert Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer International Publishing, 2018.

[31] M.A. Martínez-Prieto, M. Arias, and J.D. Fernández. Exchange and Consumption of Huge RDF Data. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 437–452, 2012.

[32] M.A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical Compressed String Dictionaries. *Information Systems*, 56:73–108, 2016.

[33] M.A. Martínez-Prieto, J.D. Fernández, and R. Cánovas. Compression of rdf dictionaries. In ACM Press, editor, *ACM International Symposium on Applied Computing (SAC)*, pages 1841–1848. ACM, 2012.

[34] M. Meier. Towards Rule-Based Minimization of RDF Graphs under Constraints. In *Procedings of the International Conference on Web Reasoning and Rule Systems (RR)*, pages 89–103, 2008.

[35] T. Minier, H. Skaf-Molli, and P. Molli. SaGe: Web Preemption for Public SPARQL Query Services. In *Proc. of The Web Conference*, 2019. `https://callidon.github.io/pdf/paper_www19.pdf`.

[36] B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. *OWL 2 Web Ontology Language Profiles*. W3C Recommendation, 2012. `www.w3.org/TR/owl2-profiles/`.

[37] G. Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

[38] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *Proceedings of the International Conference on Data Engineering*, pages 984–994, 2011.

[39] J.Z. Pan, J.M Gómez-Pérez, Y. Ren, H. Wu, W. Haofen, and M. Zhu. Graph Pattern Based RDF Data Compression. In *Proceedings of the Joint International Conference om Semantic Technology (JIST)*, pages 239–256, 2015.

[40] 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.`

[41] R. Pichler, A. Polleres, S. Skritek, and S. Woltran. Towards Rule-Based Minimization of RDF Graphs under Constraints. In *Procedings of the International Conference on Web Reasoning and Rule Systems (RR)*, page 133148, 2010.

[42] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[43] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag London Limited, 2007.

[44] D. Salomon. *Variable-length Codes for Data Compression*. Springer-Verlag, 2007.

[45] G. Schreiber and Y. Raimond. *RDF 1.1 Primer*. W3C Recommendation, 2014. `https://www.w3.org/TR/rdf11-primer/.`

[46] J. Swacha and S. Grabowski. OFR: An Efficient Representation of RDF Datasets. In *Proceedings of the Symposium on Languages, Applications and Technologies (SLATE)*, pages 224–235, 2015.

[47] R. Taelman, M. Vander Sande, J. Van Herwegen, E. Mannens, and R. Verborgh. Triple storage for random-access versioned querying of RDF archives. *Journal of Web Semantics*, 54:4–28, 2019.

[48] R. Ticona-Herrera, R. Tekli, J. Chbeir, S. Laborie, I. Dongo, and R. Guzman. Toward RDF Normalization. In *Proceedings of the International Conference on Conceptual Modeling (ER)*, pages 261–275, 2015.

[49] G. Venkataraman and P. Sreenivasa Kumar. Horn-rule based compression technique for RDF data. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, pages 396–401, 2015.

[50] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. Van de Walle. Querying datasets on the web with high availability. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 180–196, 2014.