Universidad de Valladolid

Escuela de Doctorado Universidad de Valladolid

PROGRAMA DE DOCTORADO EN INFORMÁTICA

TESIS DOCTORAL:

Effective Reorganization and Self-Indexing of Big Semantic Data

Presentada por Antonio Hernández Illera para optar al grado de
Doctor/a por la Universidad de Valladolid

Dirigida por:

Miguel Ángel Martínez Prieto
Javier David Fernández García

Agradecimientos

Hace tiempo leí una dedicatoria en un trabajo de fin de grado: “A mis padres, por aguantarme”. En aquel momento la frase me pareció algo pueril, pero en realidad contiene una gran carga de emotividad, pues dependiendo del sentido que se le de al verbo, el significado de la frase se balancea entre la broma y el más profundo agradecimiento. Hoy, siendo padre de dos hijas pequeñas, solo soy capaz de intuir la difícil tarea que implicará el aguantarlas firmemente, sustentarlas para que no caigan, y si lo hacen, saber curar bien sus heridas. Que el remedio no sea peor que la enfermedad. Tiempo atrás, fueron las manos de mi madre las que nos soportaron a mi hermana y a mí. Ellá nos indicó el camino de la formación como aquel que debíamos de seguir, pese a que ella a penas lo hubo caminado. Hoy quiero aprovechar para dedicar estas líneas a todas las mujeres que me han mantenido en pie a lo largo de mi vida: mi abuela Felicitas, mi madre Genoveva, mi tía Ángeles, mi hermana Eva, mi esposa Lourdes y mis hijas Carmen y Ana. Gracias a todas ellas, en especial a mi madre (a la que nunca antes se lo había dicho), sin cuya dedicación y entereza hoy no estaría ecribiendo esto y, en cierto modo, siento que este trabajo es más suyo que mío. Gracias por el esfuerzo que te supuso el darnos una formación cuando no podías permitírtelo.

Miguel y Javi, ha sido un verdadero honor teneros a mi lado en esta aventura. Gracias por vuestro tiempo, dedicación, entrega, conocimientos, experiencia, asesoramiento y, por encima de todo, gracias por vuestro entusiasmo, pasión y fuerza vital tan contagiosa. Ni que decir tiene que nunca habría escrito esta tesis sin vuestra ayuda y apoyo. La Universidad de Valladolid debe de estar bien orgullosa de forjar profesionales como vosotros.

Abstract

The classic Web infrastructure used to publish, consume, and exchange content is also available to host *raw data* so machines can access and process such information. This so-called Web of Data has grown exponentially in recent years, weaving its own net of online, connected datasets, using RDF as a common language and a bridge between them. All this amount of generated RDF data result in huge collections, consequently opening the doors to various lines of research, including RDF data compression, which optimizes the storage and streamlines data exchange. In contrast to universal compressors, RDF compression techniques are able to detect and exploit specific forms of redundancy, leveraging syntactic and semantic redundancies in RDF data. However, to date, little attention has been paid to some structural regularities that real-world datasets follow and that constitute another source of redundancy. In this thesis we have analyzed the structural redundancy that the RDF graph inherently possesses and we have proposed a preprocessing technique called RDF-Tr *(RDF Triples Reorganizer)* which groups, reorganizes and re-codes RDF triples, alleviating two sources of structural redundancy underlying the schema-relaxed nature of RDF. We have integrated RDF-Tr into two of the main state-of-the-art RDF compressors, HDT and k^2-triples, significantly reducing in both cases the size that the original compressors achieve, thus outperforming the most prominent state-of-the-art techniques. We have denominated HDT++ and k^2-triples++ the result of applying RDF-Tr to each compressor.

RDF is supported by a whole set of semantic technologies that allows, among other things, access to data in large RDF collections thanks to SPARQL, its own SQL-like query language. In the field of RDF compression, different compact data structure configurations are used to build RDF self-indexes, providing efficient access to the data without (partial or total) decompression. The indexed HDT (called HDT-FoQ) was the pioneer in this scenario and is nowadays used by the semantic community to publish and consume large RDF data collections. In this thesis, we could not ignore this fact, and we have extended HDT++ (called iHDT++) to support full SPARQL Triple Patterns resolution, consuming less memory than its counterpart. We have proven that iHDT++ reduces by 20-45% the space that HDT-FoQ needs, while speeding up the resolution of most Triple Pattern queries, reporting space-time tradeoffs that compete and outperform, in different scenarios, the state-of-the art RDF self-indexes.

Resumen

La infraestructura de la Web clásica que utilizamos para publicar, consumir e intercambiar contenido, también está disponible para alojar el *raw data* que puede ser accedido y procesado por las máquinas. Esta Web de Datos ha experimentado un crecimiento exponencial durante los últimos años, tejiendo su propia red de *datasets* (conjuntos de datos) interconectados y disponibles en línea, utilizando RDF como lenguaje común, haciendo de puente entre ellos. Esta enorme cantidad de datos en RDF deviene en colecciones de datos de gran tamaño, y ha abierto las puertas a diversas líneas de investigación, incluyendo la compresión de datos en RDF, que optimiza el espacio de almacenamiento y, a su vez, agilizando su intercambio. A diferencia de los compresores universales, las técnicas de compresión de RDF detectan y tratan las redundancias específicas que poseen a niveles sintáctico y semántico. Sin embargo, hasta la fecha, se ha prestado poca atención a ciertos patrones estructurales que los conjuntos de datos del mundo real siguen y que constituyen otra fuente de redundancia. En esta tesis hemos analizado la redundancia estructural que los grafos RDF inherentemente poseen y hemos propuesto una técnica de preprocesamiento llamada RDF-Tr (*RDF Triples Reorganizer*) que agrupa, reorganiza y recodifica los triples, tratando dos fuentes de redundancia estructural subyacentes a la naturaleza del esquema RDF. Hemos integrado RDF-Tr en dos de los principales compresores RDF del estado del arte, HDT y k^2-triples, reduciendo significativamente en ambos casos el tamaño que obtienen los compresores originales, superando a las técnicas más prominentes del estado del arte. Hemos denominado HDT++ y k^2-triples++ al resultado de aplicar RDF-Tr en cada compresor.

RDF además, se apoya en un conjunto de tecnologías semánticas que permiten, entre otras cosas, hacer consultas a las grandes colecciones de datos en RDF gracias a SPARQL, un lenguaje de consulta propio parecido a SQL. En el ámbito de la compresión RDF se utilizan diferentes configuraciones de estructuras de datos compactas para construir auto-índices RDF, que proporcionan acceso eficiente a los datos sin necesidad de una descompresión previa de los mismos (parcial o total). HDT-FoQ, la versión indexada de HDT, fue el pionero en este ámbito y hoy en día es utilizado por la comunidad semántica para publicar y consumir grandes colecciones de datos RDF. En esta tesis no podíamos ignorar este hecho, y hemos extendido HDT++, llamándolo iHDT++, para permitir la resolución de patrones de tripletas SPARQL consumiendo menos memoria que HDT-FoQ. Hemos demostrado que iHDT++ reduce en un 20-45 % el espacio que necesita HDT-FoQ, a la

vez que acelera la resolución de la mayoría de las consultas por patrón de tripleta, mejorando la relación espacio-tiempo, en algunos escenarios, del resto de auto-índices RDF del estado del arte.

Contents

List of Figures

Chapter 1

Introduction

1.1. Motivation

The World Wide Web is a distributed set of documents in hypertext (i.e., web pages) connected through hyperlinks. The Web has been a revolution thanks to the ease of publishing and sharing information of all kinds. However, this way of publishing content is mainly aimed at human consumption, not machine processing, so the automated extraction of information from the Web is a very laborious process. To mitigate this problem, the Web of Data emerges as an extension of the World Wide Web, sponsored and defined by Tim Berners-Lee [7], one of its founders. Here, data hosted in the Web can be connected to be searched, shared and reused among applications or organizations. This information from different fields of knowledge is published in the Web as datasets, and they acquire greater value if they are interconnected, so machines can browse their content and navigate the Web of Data. The way in which data should be published in this linked model is defined [6] by four principles:

1. Use URIs to identify things.
2. Use HTTP URIs so those names can be looked up.
3. Provide useful information when a URI is looked up, using standards like RDF [32] or SPARQL [47], among others.
4. Include links to other URIs, so more things can be discovered.

Linked Data becomes *Linked Open Data* (LOD) when such content is under an open license and, therefore, free to reuse or republish. This open data model has been growing impressively over the past few years. This

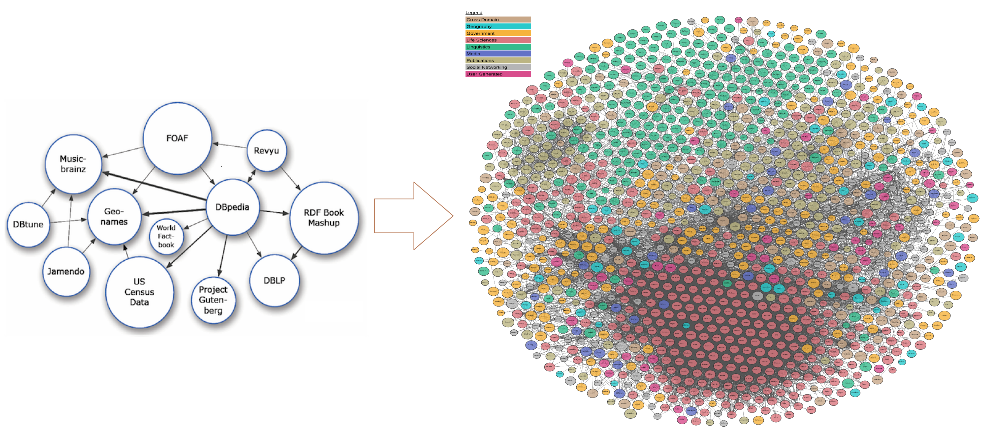


Figure 1.1: Evolution of the *Linked Open Data* cloud.

evolution can be seen in Figure 1.1, where the first twelve datasets that, in 2007, originally formed the LOD cloud [38] (i.e., open and interconnected datasets) can be seen on the left. In contrast, the right hand side of the Figure shows the growth that the LOD cloud has experienced in these twelve years, where it is virtually impossible to visualize any of the 1,239 datasets. Note that *DBpedia* [3] is really at the core of the LOD, as it is present in the center of both clouds. The reason for this is that *DBpedia* is based on the *Wikipedia* structured content (mainly infoboxes), and hence contains cross-domain information that references to (and is referenced by) specific knowledge datasets. Since the first version of the LOD cloud in 2007, many important projects of heterogenous fields of knowledge have joined the initiative, such as geography (e.g., *LinkedGeoData*[1], with more than 1.2 billion statements), life sciences (e.g., *Bio2RDF*[2], 11 billion statements) or general knowledge (e.g., *DBpedia*[3] or more recently *WikiData*[4] with 9.5 billion and 8 billion statements respectively). One of the main achievements of the *Linked Open Data* project is that the datasets that comprise the LOD cloud share their information in the same language, in order to make it easy for machines to access, browse and navigate their data model.

RDF. The W3C (*World Wide Web Consortium*) proposed a model to describe, publish and interchange data on the Web called RDF (*Resource Desciption Framework*) [32]. The information in RDF is expressed through

[1]`http://linkedgeodata.org/`
[2]`http://bio2rdf.org/`
[3]`http://www.dbpedia.org/`
[4]`https://www.wikidata.org/`

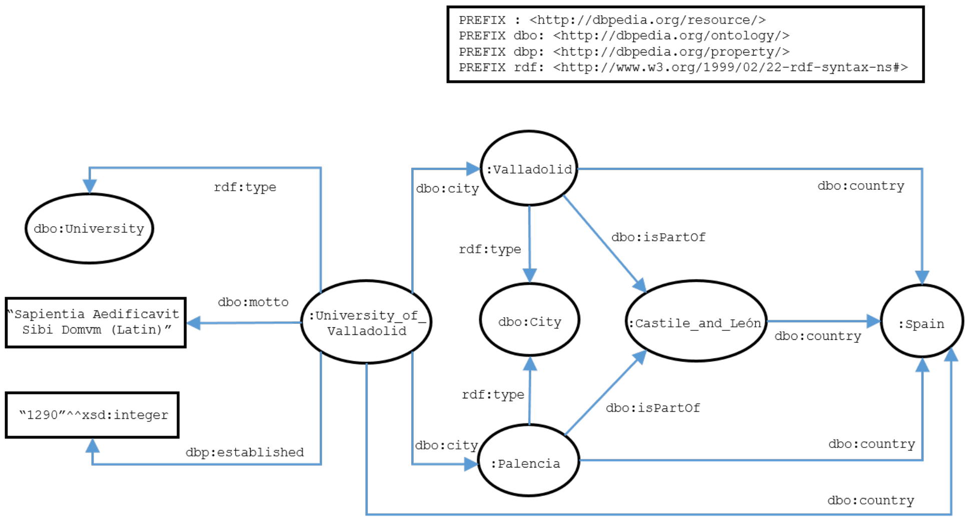

PREFIX : <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
dbo:University
rdf:type
"Sapientia Aedificavit Sibi Domvm (Latin)"
dbo:motto
"1290"^^xsd:integer
dbp:established
:University_of_Valladolid
dbo:city
:Valladolid
rdf:type
dbo:City
dbo:isPartOf
:Castile_and_León
dbo:country
:Spain
rdf:type
dbo:isPartOf
dbo:city
:Palencia
dbo:country
dbo:country
dbo:country

Figure 1.2: Example of an RDF graph.

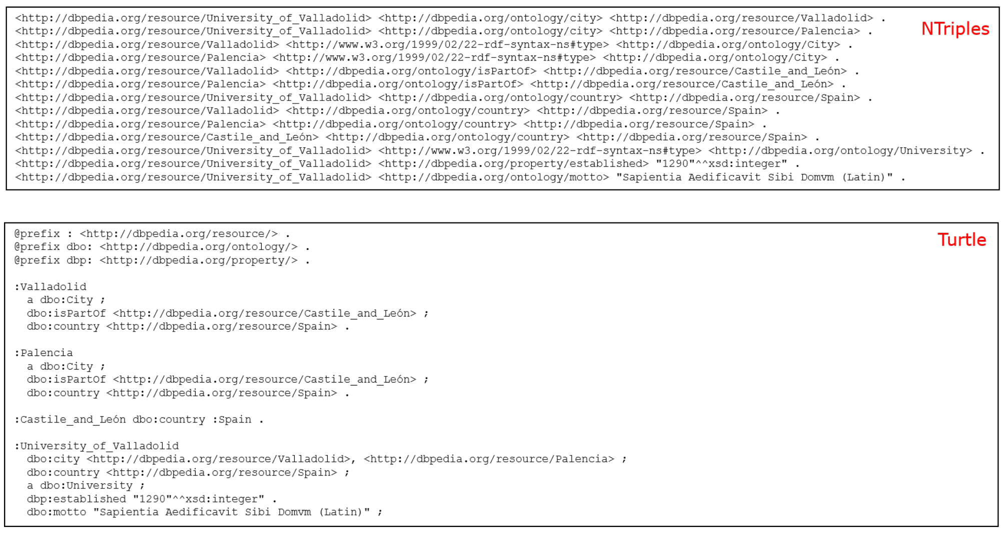
triples, each one comprising the resource being described (i.e., subject), a property of that resource (i.e., predicate), and the corresponding value (i.e., object). This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources; i.e., the graph nodes. Within a triple, a subject can be a URI or a blank node (i.e., anonymous resource), a predicate must be a URI and the object can be a URI, a blank node (or *bnode*) or a literal. A more formal definition can be found in Section 2.1.

A simple but real example of an RDF graph, extracted from *DBpedia*, is shown in Figure 1.2, where we have a few triples representing some features about the University of Valladolid[5]: It is a University established in 1290 whose motto is *"Sapientia Aedificavit Sibi Domvm (Latin)"*. Besides, it is present in two cites, Valladolid and Palencia, both are part of Castile and León, in Spain. An example of a triple is `(:University_of_Valladolid, dbo:motto, "Sapientia Aedificavit Sibi Domvm (Latin)")`, which expresses the motto of the University.

The W3C recommends syntaxes for storing and exchanging RDF such as Turtle [46], JSON-LD [50] or N-Triples [13], among others, but RDF is not tied to a fixed serialization format, hence the RDF Graph in Figure 1.2 can be written in any *standard* RDF format. For example, the Turtle and N-Triples serializations for our example are shown in Figure 1.3.

All RDF formats convey the same meaning, but they also suffer the same problems: their high level of verbosity and redundancy. Although

[5]http://dbpedia.org/page/University_of_Valladolid


NTriples
Turtle

```
<http://dbpedia.org/resource/University_of_Valladolid> <http://dbpedia.org/ontology/city> <http://dbpedia.org/resource/Valladolid> .
<http://dbpedia.org/resource/University_of_Valladolid> <http://dbpedia.org/ontology/city> <http://dbpedia.org/resource/Palencia> .
<http://dbpedia.org/resource/Valladolid> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/City> .
<http://dbpedia.org/resource/Palencia> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/City> .
<http://dbpedia.org/resource/Valladolid> <http://dbpedia.org/ontology/isPartOf> <http://dbpedia.org/resource/Castile_and_León> .
<http://dbpedia.org/resource/Palencia> <http://dbpedia.org/ontology/isPartOf> <http://dbpedia.org/resource/Castile_and_León> .
<http://dbpedia.org/resource/University_of_Valladolid> <http://dbpedia.org/ontology/country> <http://dbpedia.org/resource/Spain> .
<http://dbpedia.org/resource/Valladolid> <http://dbpedia.org/ontology/country> <http://dbpedia.org/resource/Spain> .
<http://dbpedia.org/resource/Palencia> <http://dbpedia.org/ontology/country> <http://dbpedia.org/resource/Spain> .
<http://dbpedia.org/resource/Castile_and_León> <http://dbpedia.org/ontology/country> <http://dbpedia.org/resource/Spain> .
<http://dbpedia.org/resource/University_of_Valladolid> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/University> .
<http://dbpedia.org/resource/University_of_Valladolid> <http://dbpedia.org/property/established> "1290"^^xsd:integer" .
<http://dbpedia.org/resource/University_of_Valladolid> <http://dbpedia.org/ontology/motto> "Sapientia Aedificavit Sibi Domvm (Latin)" .
```

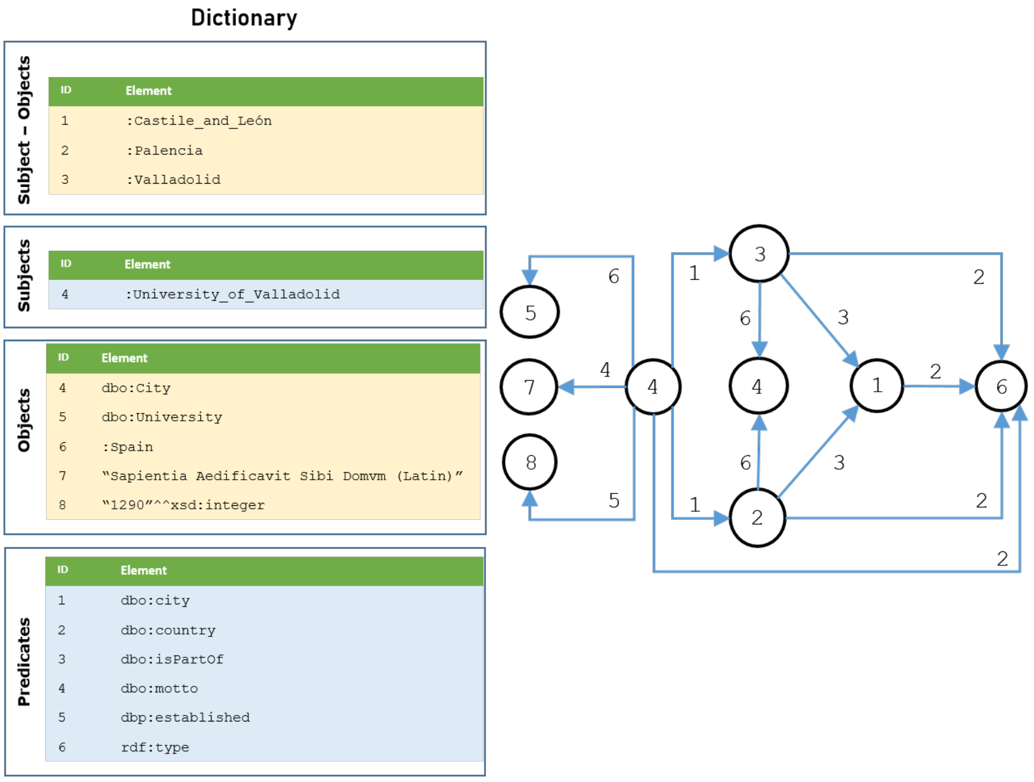
```
@prefix : <http://dbpedia.org/resource/> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix dbp: <http://dbpedia.org/property/> .

:Valladolid
  a dbo:City ;
  dbo:isPartOf <http://dbpedia.org/resource/Castile_and_León> ;
  dbo:country <http://dbpedia.org/resource/Spain> .

:Palencia
  a dbo:City ;
  dbo:isPartOf <http://dbpedia.org/resource/Castile_and_León> ;
  dbo:country <http://dbpedia.org/resource/Spain> .

:Castile_and_León dbo:country :Spain .

:University_of_Valladolid
  dbo:city <http://dbpedia.org/resource/Valladolid>, <http://dbpedia.org/resource/Palencia> ;
  dbo:country <http://dbpedia.org/resource/Spain> ;
  a dbo:University ;
  dbp:established "1290"^^xsd:integer" .
  dbo:motto "Sapientia Aedificavit Sibi Domvm (Latin)" ;
```

Figure 1.3: Serialization of the RDF graph.

Turtle mitigates redundancy by grouping prefixes (with the inclusion of `"@prefix"` terms) and using some sort of adjacency lists, arbitrary long URIs (e.g., `http://dbpedia.org/resource/Castile_and_León`) are still present in several triples, playing the role of subjects and/or objects. Verbosity and redundancy are particularly troubling in the *Linked Open Data* domain, where large datasets are increasingly consolidated. This problem is not new, but remains challenging [16] and is usually referred to as *Big Semantic Data* management [34].

In this scenario, the W3C Member Submission, HDT (*Header – Dictionary – Triples*) [19] [18], emerges as the first RDF binary serialization that proposes to transform the classical RDF graph to a graph of integer IDs. HDT minimizes the repetition of potentially large strings using a *Dictionary*, which assigns a numerical ID to each term in the dataset. It divides the RDF terms into four subsets lexicographically sorted, depending on the role that each RDF term plays within the dataset (subject, predicate, object or subject-object[6]). This partitioning [2] avoids the duplication of terms in the dictionary, since up to 60% of the RDF terms of a dataset belong to the subject-object category [37]. Figure 1.4 shows the four clusters of RDF terms in the *Dictionary* component, and the transformed ID graph corresponding to our example. To decode Triple-ID, for example `(4, 4, 7)`, we just have to look for the particular ID of each term (subject, predicate and object) in their specific Dictionary. Therefore, sub-

[6]HDT refers to ”subject-object” as the terms that play both subject and object roles in the dataset.



ID	Element
1	:Castile_and_León
2	:Palencia
3	:Valladolid

ID	Element
4	:University_of_Valladolid

ID	Element
4	dbo:City
5	dbo:University
6	:Spain
7	"Sapientia Aedificavit Sibi Domvm (Latin)"
8	"1290"^^xsd:integer

ID	Element
1	dbo:city
2	dbo:country
3	dbo:isPartOf
4	dbo:motto
5	dbp:established
6	rdf:type

Figure 1.4: Our example as a graph of integer IDs.

ject 4 corresponds to the term `:University_of_Valladolid`; the predicate 4 is `dbo:motto`; and finally, the object 7 encodes the literal *`"Sapientia Aedificavit Sibi Domvm (Latin)"`*. The original triple is retrieved replacing the identifiers with their corresponding RDF terms, `(:University_of_Valladolid, dbo:motto,` *`"Sapientia Aedificavit Sibi Domvm (Latin)"`*`)`.

SPARQL. While it is true that the main use of RDF compression is for the publication and exchange of *Big Semantic Data*, HDT finds its maximum expression when querying these large collections of linked data through a query language, SPARQL (*SPARQL Query Language for RDF*) [47]. The simplest form of querying RDF with SPARQL is a *Triple Pattern*, an RDF triple in which any of its components (*subject*, *predicate*, *object*) can be a variable (denoted by a question mark) or a constant. Therefore, eight possible combinations (i.e., Triple Patterns) are possible: {`(?,?,?)`, `(?,?,o)`, `(?,p,?)`, `(?,p,o)`, `(s,?,?)`, `(s,?,o)`, `(s,p,?)`, `(s,p,o)`}. The following example, `(:University_of_Valladolid, dbo:motto, ?o)` asks for the object of the triple; i.e., we want to know the actual motto of the University of Valladolid. Traditionally, SPARQL queries have been made possible thanks to endpoints provided by specific RDF graph storage artifacts (i.e., *triple stores*) that allow the query of their contents. However, HDT introduced a novelty in the SPARQL field, since it is the first RDF serialization technique that allows Triple Pattern query resolution on the file, without the need for any additional support (i.e., *triple stores*), but adding some indexes to accelerate their resolution.

HDT-FoQ (*HDT Focused on Querying*) [33] is the extension of the HDT model that attaches some compact data structures that make simple but efficient data retrieval possible. For these reasons, HDT is an RDF binary serialization format, but it is also used as a storage engine in well known Semantic Web projects such as *Linked Data Fragments* (LDF)[7], *LOD Laundromat*[8] [5] or *LOD-a-lot* [17]. LDF provides a query interface based on *Triple Pattern Fragments* (TPF) [53], an iterative process that converts the clients' complex SPARQL queries (over HDT datasets) into the union of simple Triple Patterns, returning paginated partial results, and therefore balancing the computational cost between servers and clients. *LOD Laundromat* is a project that cleans the LOD RDF datasets, removing blank nodes, duplicated triples and syntax errors. After being *cleaned*, data is published in many syntaxes, including HDT, and can be queried by the TPF APIs. *LOD-a-lot* exposes an HDT mashup of 28 billion *cleaned* triples (from *LOD Laundromat*), queryables by the TPF interface.

Challenges. The big boom in the exposure of large datasets on the Web of Data, and the wide acceptance of RDF as the glue that links them, has focused the research of the last few years on the compaction of the serialization of RDF graphs. The fact that HDT was the first RDF binary serializer, along with its simple and intuitive model, has made HDT a great success within the Semantic Web community. Precisely because of its simplicity, HDT (and HDT-FoQ) does not capture the peculiarities of the RDF graph and can therefore be improved by detecting and eliminating additional redundancy in the *Triples* component. Specifically, RDF collections hoard three types of redundancy [41]: ***Semantic redundancy*** occurs when the knowledge described by some triples can be inferred from others. In this case, these triples can be removed, thus reducing the size of the dataset, but preserving the knowledge. Compression in this case is merely related to how information is available within the dataset, so classic compression techniques are not valid to alleviate this type of redundancy. It should be noted that semantic

[7] `https://linkeddatafragments.org/`

[8] `http://lodlaundromat.org/`

compression can be lossy, so the decompression process will not necessarily return the original graph (but an equivalent one). ***Symbolic redundancy*** is present when there are similarities and repetitions between the URIs and Literals (i.e., symbol repetitions). Symbolic redundancy is mainly due to URIs with long prefixes. This kind of redundancy can be removed by *universal compressors* (i.e., `gzip`, `bzip2`, ...), since URI prefixes appear repeatedly throughout the dataset. However, on the contrary, they do not allow access to data without a prior decompression. This type of redundancy can also be mitigated with the use of compressed string dictionaries [35], which are used to encode the RDF terms present in a dataset as integers, making a translation possible between a term and its identifier and *vice versa*. Finally, ***syntactic redundancy*** refers to the existence of structural regularities in the RDF graph. Resources of the same class are usually described by the same predicates; for example, the predicate `dbo:motto` describes resources such as Universities, but could not be used to describe printers, for instance. Unlike N-Triples, which writes the full terms of each triple (one per line), Turtle syntax is able to minimize this redundancy by serializing the triples, grouping predicate-object pairs related to the same subject (i.e., adjacency lists). As in the case of subject-predicate connections, relationships established between predicates and objects are also restricted to a limited range. Specific graph compressors can treat this sort of redundancy by serializing the graph in compressed adjacency lists or matrix structures. The fact that graph compressors usually work with integers must be taken into account, so a previous step of generating a dictionary is necessary.

1.2. Hypothesis

Specific RDF compressors mitigate any of the three redundancies above achieving a size reduction. These compressors are classified into physiscal compressors, if they exploit the syntactic and/or symbolic redundancy; logical compressors, if they act on semantic redundancy; and hybrid compressors, which mix both types. An introduction to the main RDF compressors of the state of the art can be found in Section 2.2.

In this thesis, we have addresed a particular RDF problem: the distribution of the predicates of the RDF real-world datasets are such that they introduce overheads in their serializations. This problem brings up two challenges. On the one hand, specific syntactic redundancy should be identified and treated in order to improve the compression that the main state-of-the-art techniques currently achieve. On the other hand, new data structure configurations should be proposed to allow SPARQL Triple Pattern queries on compressed data. This feature is currently available in techniques that lead the state of the art. All of the above leads us to raise the main hypothesis of this thesis:

> *"RDF graphs are not randomly structured, on the contrary, they tend to follow organizational patterns resulting in semi-structured datasets. Based on this inherent RDF characteristic, the terms can be organized, grouped and re-encoded so that syntactic redundancy is minimized, improving the existing RDF compression techniques, such as HDT, while maintaining its ability to query in compressed space."*

Within a dataset, subjects of the same class are usually described by the same predicates. For example, a person might be described by predicates such as an ID, name, sex, date of birth, etc. However, those predicates do not fit when describing countries, for instance. In our example (see Figure 1.2), the resources `:Valladolid` and `:Palencia` are described by the same predicates {`dbo:country, dbo:isPartOf, rdf:type`}, since both are of the same class (i.e., cities). We call predicate-family (or family) the set of predicates that describe subjects of the same nature, splitting the graph into subgraphs, each one containing all subjects described by the same characteristics. Therefore, the relationships established between subjects and predicates can be replaced by those between subjects and families. A more fine-grained analysis is performed when considering the presence of predicate `rdf:type`. This predicate is used extensively to categorize the information in the dataset, providing the class of the subject it describes. Therefore, although it is not mandatory, this predicate appears many times throughout the dataset. Type objects (i.e., related to the `rdf:type` predicate) can be attached to the family to semantically categorize the subjects they are describing. Besides, although RDF allows the connection between any predicate and object, predicates are, in practice, related to a well-defined range of objects. Therefore, objects can be locally identified within the scope of each predicate, using fewer bits to be encoded.

The second part of the hypothesis proposes that our new form of serialization can be queried efficiently. In this context, RDF self-indexing provides efficient access to the data without a decompression. As seen before, HDT can generate auto-indexes on the top of its structure to allow and speed up the query of its data. HDT-FoQ (HDT and its self-indexes) manages to efficiently perform searches by subject, but requieres expensive indexes to run predicate and object-based queries. We efficiently guarantee data querying by adding new compact data structures on top of our proposal, which will alleviate the main weakness of HDT-FoQ (i.e., predicate-based queries), while preserving the efficiency of the rest.

1.3. Contribution

This thesis encompasses several contributions. First of all, a full revision of *RDF compression* [34], which has been published in the homonymous chapter of the book *Encyclopedia of Big Data Technologies* [49], where we have also described the sources of redundancy that RDF inherently possesses (see the previous section) and how the different kinds of compressors are able to mitigate them.

In this thesis we analyze common patterns related to the use of predicates and objects in RDF real-world datasets, and show how structural sources of redundancy underlying the schema-relaxed nature of RDF can be exploited to improve their effective encoding. Its main contribution is the conception of our proposed RDF graph reorganization technique, called RDF-Tr, which alleviates its structural redundancies and improves HDT (the most used RDF compressor by the community) in terms of compression space and decompression time. However, the use of RDF-Tr is not limited to HDT. In particular, it is applied to another syntactic compressor, k^2-triples [1], which performs a more effective ID-graph encoding, organizing the RDF terms like HDT (four partitions), but encoding the triples in $|P|$ binary matrices, which are subsequently compressed using k^2-trees [11] (see Section 2.2 for more details). Applying RDF-Tr to k^2-triples, as in the case of HDT, achieves improvements in compression size and decompression speed.

RDF-Tr groups triples by families of predicates and recodes object IDs within the scope of the predicate in which they act. This results in a new binary representation of the triples, called HDT++, while retaining the original HDT *Dictionary* component. HDT++, which was presented in the *Data Compression Conference* (DCC) in 2015 [22] (see Chapter 3), outperforms its original effectiveness up to 2.3 times, accelerating the decompression time up to 3.4 times.

The fact that RDF-Tr acts only on the *Triples* component of HDT, leaving the *Dictionary* intact, makes it possible to apply the same transformations to some other RDF compressors, as in the case of the aforementioned k^2-triples. RDF-Tr, which was published in the journal *Information Sciences* in 2020 [24] (see Chapter 4), formalizes the DCC proposal, optimizing the configuration of some parameters that allow the improvement of HDT++. In addition, it is also used to improve k^2-triples. Once RDF-Tr is applied over k^2-triples, a more compact version is obtained, called k^2-triples++, saving up to 2.3 times the space needed by its original version, and increasing the decompression time up to 2.4 times.

In addition to compression, one of the strengths that both HDT and k^2-triples have, as well as RDFCSA [10] (another of the leaders in RDF compression), is that they all allow for the resolution of SPARQL Triple Patterns on the serialized file. Therefore, and despite the great numbers obtained in terms of compression, another contribution is needed for this thesis: A proposal for indexing the reorganized RDF graph for the sake of providing resolution of Triple Patterns. To complete this challenge, iHDT++ (*indexed HDT++*) extends the concept of HDT++ with the inclusion of proficient self-indexes that alleviate the main weakness of HDT-FoQ (i.e., predicate-based queries). On the one hand, iHDT++ outperforms HDT-FoQ in terms of space complexity by up to $\approx 45\%$. On the other hand, regarding the resolution of Triple Patterns, the main achievement is the improvement of two orders of magnitude when solving the query {`(?,P,?)`}, while {`(?,?,?)`} is up to one order of magnitude faster in iHDT++, depending on the dataset. For the rest of the Triple Patterns accessed by predicate (i.e., {`(S,P,O), (S,P,?)`}), iHDT++ is still faster, although the difference is less significant. Accessing by object (i.e., {`(S,?,O), (?,P,O), (?,?,O)`}) reports similar times as HDT-FoQ, being the access by subject (i.e., `(S,?,?)`), the only operation that HDT-FoQ solves faster than iHDT++. A preliminary version of iHDT++ was presented at the *Jornadas de Ingeniería del Software y Bases de Datos* (JISBD) [23] in 2017. Finally, an optimized and definitive version was presented in the *International Symposium on Language & Knowledge Engineering* (LKE) [25] in 2019, and published in the *Journal of Intelligent & Fuzzy Systems* [26] in 2020 (see Chapter 5).

Finally, all these contributions have been compiled into a tool[9], capable of being integrated into the original HDT library, which allows it to perform the reorganization of the triples from an HDT file, transforming it into HDT++. This transformation can also be applied to k^2-triples. In addition, iHDT++ self-indexes can be built with this tool to enable Triple Pattern query processing.

1.4. Thesis Structure

This thesis is presented as a compendium of publications. Chapter 2 provides the background on which the compendium is based, including the processes of publishing and querying data on the Web of Data, basic compression concepts and a state of the art of RDF compressors. Chapters 3

[9]`https://github.com/antonioillera/iHDTpp-src`

to 5 gather the three publications that constitute this thesis, each of them corresponding to a particular contribution, which are the following:

– Chapter 3: *"Serializing RDF in Compressed Space".* [22] In proceedings of the *Data Compression Conference* (DCC 2015). Conference indexed in GII-GRIN-SCIE (GGS)[10] Conference Rating: GGS Class 2.

– Chapter 4: *"RDF-TR: Exploiting structural redundancies to boost RDF compression".* [24] Journal article published in *Information Sciences*, indexed in the *Journal Citation Reports* (JCR) Ranking: Impact factor 5,910. Q1: Computer Science, Information Systems (9/156).

– Chapter 5: *"iHDT++: Improving HDT for SPARQL Triple Pattern Resolution".* [26] Journal article published in *Journal of Intelligent & Fuzzy Systems*, indexed in the *Journal Citation Reports* (JCR) Ranking: Impact factor 1,851. Q3: Computer Science, Artificial Intelligence (79/136).

Finally, Chapter 6 presents the Conclusions of the thesis and the open lines of research that are left as future work.

[10]`http://gii-grin-scie-rating.scie.es/`

Chapter 2

Background

2.1. Semantic Web and Technologies

We have previously introduced the notion of Web of Data, in contrast to the document-centric traditional Web, and how Linked Open Data helps to share and complete information among datasets, navigating from generalist datasets like DBpedia to specific ones such as Linkedmdb or LinkedGeoData. The term *Semantic Web* refers to this Web of Linked Data and technologies around it. Tim Berners-Lee identified five different levels of linked data quality [6], depending on how a particular dataset is published in the Web:

1. Available on the web (irrespective of the format) but with an open license, to be considered as Open Data.

2. Available as machine-readable structured data (e.g., Excel instead of an image scan of a table).

3. Use non-proprietary format (e.g., CSV instead of Excel).

4. Use open standards from W3C (RDF and SPARQL) to identify things, so that people can point to your content.

5. Link your data to other people's data to provide context.

Levels four and five clearly identify RDF as a key semantic technology within the Linked Open Data initiative. Some techniques have been developed to facilitate the generation of RDF data. This is the case of R2RML (*RDB to RDF Mapping Language*) [15], a language to dump the content of relational databases into RDF datasets; or GRRDL (*Gleaning Resource Descriptions from Dialects of Languages*) [14], which proposes transformations over markup languages such as XML or XHTML [43] to convert the content into RDF; thus, web page contents can be extracted and integrated into the Semantic Web.

As previously stated, data in RDF are expressed by triples with the form (*subject*, *predicate*, *object*) and these triples can be seen as labelled directed graphs. Looking back at Figure 1.2, we can see that there are two types of *objects*: *literals*, which are surrounded by a square in the example; and *URIs*, which are encircled in the same example. Special nodes are the *blank nodes*, which are anonymous resources (acting as *subjects* and/or *objects*) without a URI that are normally used to group nodes. At this point, we can show formal definitions [21] of an RDF triple and an RDF graph assuming infinite, mutually disjoint sets U (RDF URI references), B (blank nodes), L (RDF literals) and a set of variables V.

Definition 1 (RDF triple): *A triple* $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ *is called an RDF triple. In such a triple, s is called the subject, p the predicate and o the object.*

Definition 2 (RDF graph): *An RDF graph is a set of RDF triples. Each triple* (s, p, o) *in an RDF graph can be graphically represented by* $s \xrightarrow{p} o$.

Definition 3 (Triple Pattern): *An RDF Triple Pattern is an RDF triple* $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$

Just as it is important that different datasets represent their data in the same language (i.e., RDF) to enable their connection, it is equally important to use the same vocabularies to interpret their content in certain contexts. For this purpose, the community has proposed RDFS and OWL to restrict RDF semantics. Figure 2.1 shows the *Semantic Web Stack* or *Semantic Web Layer Cake*[1], which illustrates the hierarchy of technologies that make up the Semantic Web, each of them sustained by the elements of the lower layers.

RDFS (*RDF Schema*) [9] is a semantic extension of RDF (see Figure 2.1), which provides a very simple way to construct vocabularies (i.e., ontologies), mainly based on the use of class hierarchies and restrictions in the association of terms. RDFS uses *rdfs:Class* to identify resources as classes and *rdfs:subClassOf* to establish relationships between classes and superclasses. The next example defines `Public_University` as a subclass of `University`

(:University, rdf:type, rdfs:Class)
(:Public_University, rdf:type, rdfs:Class)
(:Public_University, rdfs:subClassOf, :University)

[1]`https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#(24)`

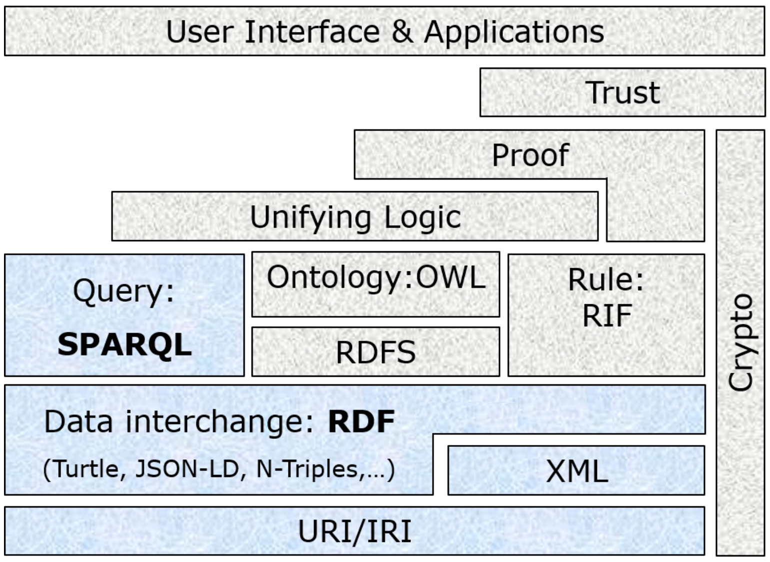


Figure 2.1: The Semantic Web Stack.

Restrictions in the relationships that can be established between resources are defined by the use of two properties: domain (*rdfs:domain*) and range (*rdfs:range*). The *rdfs:domain* predicate indicates that a property applies to instances of a particular class; while the *rdfs:range* predicate indicates that the values of a property are instances of a particular class. For instance, we could declare that a predicate `enrolled` is restricted to the relationship established between instances of the classes `Student` and `University`.

(ex:enrolled, rdf:type, rdf:Property)
(ex:enrolled, rdfs:domain, rdf:Student)
(ex:enrolled, rdfs:range, rdf:University)

In addition to RDFS, OWL (*Web Ontology Language*) [42] offers a bigger and more expressive vocabulary to model domains in a more flexible way. Properties in OWL can be characterized as *Symmetric*, *Reflexive* or *Transitive*, among others, giving greater value to the relationships established between resources. For example, if we have defined the property `isPartOf` as *Transitive* (see Figure 2.2), we could express (`Valladolid`, `isPartOf Castile_and_León`) and (`Castile_and_León`, `isPartOf`, `Spain`). However, it would not be necessary to describe the fact that (`Valladolid`, `isPartOf Spain`), since this information is inferred by the transitive property of `isPartOf`.

OWL incorporates a great variety of predicates that provide semantic content to classes and individuals (i.e., instances), allowing the use of logical operators in definitions, such as union (*owl:unionOf*) or intersection

```
:isPartOf rdf:type owl:ObjectProperty ;
          rdfs:subPropertyOf owl:topObjectProperty ;
          rdf:type owl:TransitiveProperty ;
          rdfs:domain [ rdf:type owl:Restriction ;
                        owl:onProperty :isPartOf ;
                        owl:someValuesFrom owl:Thing
                      ] ;
          rdfs:range [ rdf:type owl:Restriction ;
                       owl:onProperty :isPartOf ;
                       owl:someValuesFrom owl:Thing
                     ] .
```

Figure 2.2: A transitive property defined in OWL.

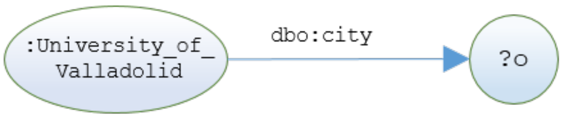
(*owl:intersectionOf*) between ranges, restricting relationships through cardinalities (*owl:cardinality*, *owl:minCardinality*, *owl:maxCardinality*) or constraining the values (*owl:allValuesFrom*, *owl:someValuesFrom*, *owl:hasValue*), to name but a few. OWL also introduces the *owl:sameAs* property, which is vital in the Linked Data project, since it allows equivalences to be established between concepts of different datasets.

RDFS and OWL not only provide semantics to RDF, but also facilitate reasoning processes to infer new knowledge (i.e., triples) that is not explicitly contained within the dataset. Even if, in our example, there were no triple (`:Valladolid`, `dbo:country`, `:Spain`), this knowledge could be inferred from (`:Valladolid`, `dbo:isPartOf`, `:Castile_and_León`) and (`:Castile_and_León`, `dbo:coutry`, `:Spain`). While RDFS is simpler and provides lighter semantics, OWL is more complex, richer, and in turn introduces greater complexity of reasoning [45].

One of the main attractions of RDF is that complex queries can be performed on datasets thanks to its SQL-like language, SPARQL (briefly introduced in Section 1.1). Queries in SPARQL are based on graph pattern matching over the RDF graph, returning a subgraph that satisfies the established conditions in the query. Recall that Triple Patterns are the basic construction queries in SPARQL, where any term can be a variable or a constant. Back to the example in Figure 1.2, let us suppose that we want to know the cities where the University of Valladolid is present, which means querying (`:University_of_Valladolid`, `:city`, `?`). Figure 2.3 shows how this Triple Pattern is expressed as a SPARQL. Solving the query in our example will return two objects (`:Valladolid` and `:Palencia`) mapped to the variable `?o`. Note that the result of applying the same query to the DBPedia endpoint[2] will return more results, as our example graph is a basic excerpt of the original.

Triple patterns are the prelude to much more complex queries that can be

[2]`https://dbpedia.org/sparql`



```
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?s ?p ?o
FROM <http://dbpedia.org/>
WHERE {
   ?s ?p ?o
   VALUES (?s ?p){(:University\_of\_Valladolid dbo:city)}
}
```

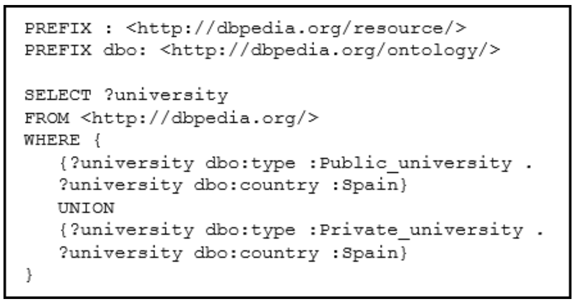
Figure 2.3: Querying by SPARQL Triple Pattern

```
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?city ?country
FROM <http://dbpedia.org/>
WHERE {
   :University\_of\_Valladolid dbo:city ?city .
   ?city dbo:country ?country .
}
```

Figure 2.4: Basic Graphic Pattern in DBPedia

performed in SPARQL. A set of Triple Patterns constitutes a Basic Graph Pattern (or BGP), really constituting a *join* of those Triple Patterns matching the RDF graph. Figure 2.4 shows an example of BGP made up of two Triple Patterns (surrounded by the WHERE clause) which looks in DBPedia for the cities and the countries of those cities where the University of Valladolid is present. The UNION operator forms a disjunction of two graph patterns; solutions to both sides of the union are included in the results. For example, the sentence shown in Figure 2.5 returns Public and Private Universities in Spain.

The well known SQL modifiers GROUP BY, HAVING, ORDER BY or LIMIT can be used in SPARQL sentences with the same purpose. SPARQL also handles other commonly used and important clauses, such as FILTER, which eliminates from the solution those results that do not satisfy a condition, or OPTIONAL, that tries to match a graph pattern, but if the optional match fails, the whole query does not. One of the most relevant updates of SPARQL has been the introduction of the so-called *property paths* that allow graph pattern matching of arbitrary length paths. In addition, different query forms, such us SELECT (returns the values selected in sentence), CONSTRUCT (returns an RDF subgraph), ASK (returns a boolean answer)



```
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?university
FROM <http://dbpedia.org/>
WHERE {
   {?university dbo:type :Public_university .
   ?university dbo:country :Spain}
   UNION
   {?university dbo:type :Private_university .
   ?university dbo:country :Spain}
}
```

Figure 2.5: SPARQL UNION in DBPedia

and DESCRIBE (returns the description of a resource) can be used. Yet, SPARQL not only allows the querying of RDF data, it also supports updating the data, by adding or deleting triples.

2.2. RDF Compression

The main objective of RDF compression is to serialize an RDF graph, or a semantic equivalent, using fewer bits than traditional representations. For this purpose, RDF redundancies introduced in Section 1.1 must be detected and treated. RDF specific compressors are classified into the following three types, depending on the redundancy they treat.

Physical compressors usually remove symbolic redundancy, transforming the RDF graph into a compressed-dictionary ID graph that replaces the original graph. The most widespread way to perform the dictionary compression, the aforementioned four-section vocabulary (subjects, predicates, objects, and those terms with the subject-object role), is present in many RDF physical compressors, such us HDT [19] or k^2-triples [1] (see Figure 1.4). After creating the dictionary, the syntactic redundancy needs to be addressed on the transformed graph of integers. The graph is succintly encoded, for example, as adjacency lists or matrices.

HDT, the pioneer of this type of compressors, conceives the graph as a forest of $|S|$ subject-rooted trees, each of them storing the relationships between that particular subject and predicates. The last layer of the tree contains the objects related to the subject-predicate pairs. Later, the forest is encoded with two sequences of integers, the first concatenates the predicate IDs related to the root-subject and the second stores the relationships between objects and the subject-predicate pairs. Two additional sequences of bits mark the ranges of subject-predicate relationships and predicate-object

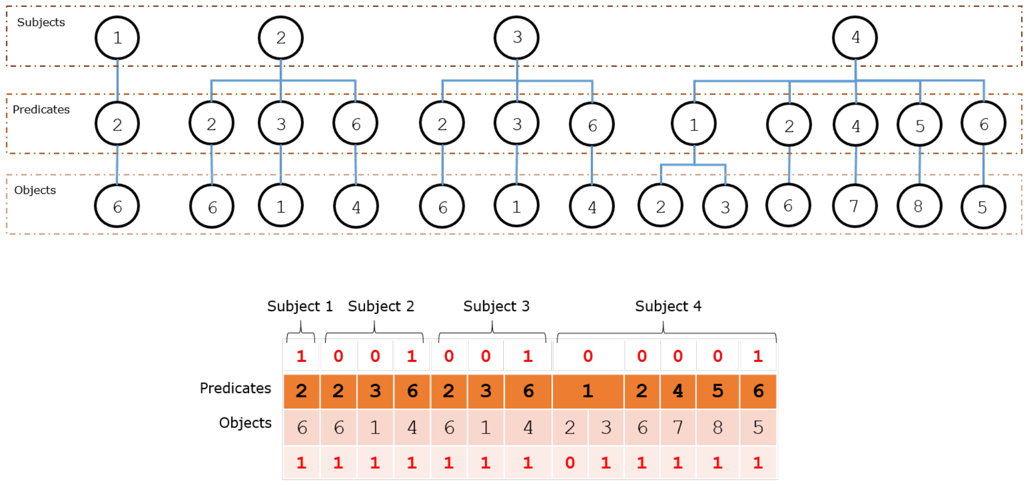


Figure 2.6: HDT *BitmapTriples*

relationships within the scope of each subject. This resulting structure is called *BitmapTriples*. Figure 2.6 shows the forest of trees representing the ID-graph we had in Figure 1.4, as well as its *BitmapTriples* encoding. HDT allows subject-based queries to simply traverse the trees starting from the subject roots, hence solving those triple patterns with a bounded subject. In contrast, this encoding needs additional indexes that HDT-FoQ [33] builds on top of it to solve the rest of the Triple Patterns.

Specifically, HDT-FoQ replaces the predicate list by a wavelet tree [20] (called W_P) to provide indexed predicate-based access: `(?, p, ?)` and `(?, p, o)`, and uses an additional adjacency list (called O-Index) to store the positions where each object is located within the *BitmapTriples* sequence of objects, allowing the resolution of `(?, ?, o)`. Both structures are shown in Figure 2.7 along with the *BitmapTriples*.

K^2-triples uses the same four-vacabulary dictionary as HDT, but proposes a different way of encoding the graph, building $|P|$ adjacency matrices. A 1-bit in the coordinate `(i,j)` of the n^{th} matrix means that the triple `(i,n,j)` is an existing triple within the dataset. The resulting matrices, which are very sparse, are subsequently compressed using k^2-trees [11], improving HDT-FoQ compression ratios. Figure 2.8 illustrates the resulting k^2-tree for the predicate 2 of our example, which encodes all triples for the second predicate. We consider $k = 2$, hence each level is divided into $k^2 = 4$ submatrices. The right hand side of the Figure depicts the conceptual tree and two sequences of bits T and L, which encode the k^2-tree.

RDFCSA [10] recodes ID-triples to avoid ID overlappings among subjects, predicates, and objects. This rearrangement ensures that all subject

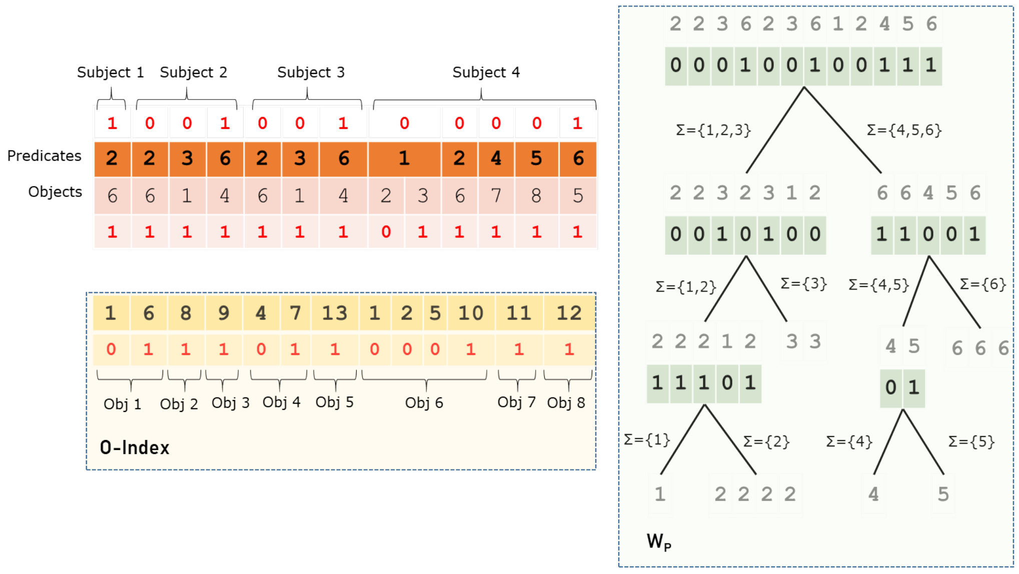


Figure 2.7: HDT-FoQ.

IDs are smaller than predicate IDs, and that these are smaller than object IDs, ensuring that ID-triples are "lexicographically" sorted, respectively, by subject, predicate, and object. RDFCSA exploits the fact that this organization can be effectively encoded using the Compressed Suffix Array (CSA) [48], which also guarantees efficient queries over the compressed representation, competing with k^2-triples, at the cost of using more space.

BMatrix [12], also based on k^2-trees, is specifically designed to work with datasets with a large number of predicates, in this case improving the state-of-the-art RDF compressors. RDF terms are encoded using the four-vacabulary dictionary, and it builds two binary matrices compressed in the end with k^2-trees; the first one stores the subjects occurrences (in rows), in triples (in columns), while the other one does the same with the objects. Triples in columns are grouped by predicate, so a last data structure is necessary to mark the positions where the triples change their predicate.

Karim et al. [29] propose a technique to detect Frequent Star Patterns across the RDF graph: pairs of predicates-objects that define entities (i.e., subjects) of the same Class. The graph is subsequently factorized, replacing the original triples with RDF molecules (graph patterns that match those Frequent Star patterns), thus decreasing the edges and compacting the graph. This method is more efficient in ontologies, where classes are well defined.

RDF compression is also applied in specific domains, as in the case of the provenance data [8], RDF metadata generated when creating or updating content in web documents, such as Wikipedia. In this specific and

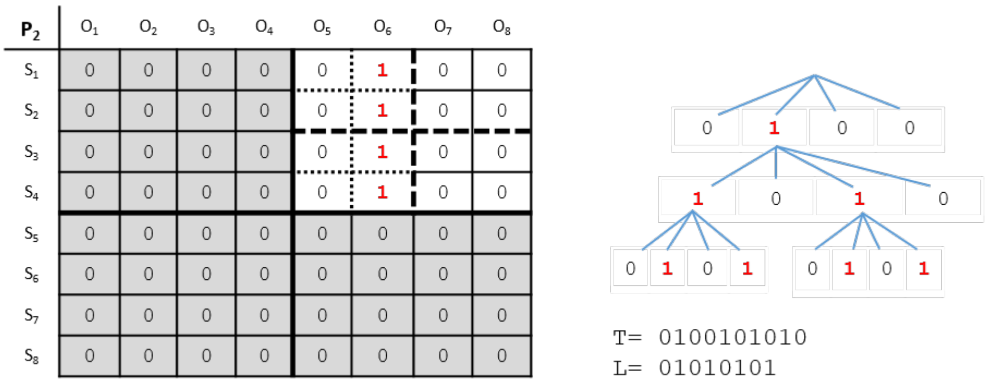


Figure 2.8: Vertical-Partitioning on k^2-triples (k=2) for P$_2$.

restricted scenario, the RDF graph follows structural patterns that are exploited, achieving an effective compression in this domain.

There are other works, not focused on the compression of the RDF graph itself, but on the optimization of RDF self-indexes used for the resolution of SPARQL Triple Patterns. Pibiri et al. [44] propose 3 three-layer indexes (i.e., permutations), one for the triples ordered in SPO, a second index stores the triples ordered in POS, and the last contains the triples ordered in SOP. Note that the SPO index is actually the HDT *BitmapTriples* configuration, replacing the bit sequences with pointers. As the set of triples in the dataset is tripled, the nodes can subsequently be recoded with the relative positions of the index where the information is already located, reducing the size of the indexes.

Logical compressors address the semantic redundancy, detecting and removing from the RDF graph redundant triples (i.e., those that can be inferred), obtaining the canonical subgraph. The first contributions in this field are based on the notion of *lean subgraph* [27] [39], the smallest instance (i.e., subgraph) of the original RDF graph. The structure of the graph clearly influences the number of triples removed by the lean subgraph, the threshold being in two triples eliminated per blank node in the graph [27]. However, the subgraph obtained does not ensure that it is the canonical graph, so there may be triples that could be inferred and therefore redundant [39].

The rule-based (RB) technique [28] mines the graph, looking for patterns of relationship between RDF terms (intra-property and inter-property patterns), which are subsequently used to create rules and eliminate inferable triples. However, the compression obtained when applying RB is not very important, and the authors ultimately compress the datasets with HDT in order to be competitive with the rest of the RDF compressors. Effectiveness in this kind of compression can be improved by using more expressive rules, as frequent patterns do not catch all the semantic associations. Horn rules can be detected by exploring the dataset [52] and then used to delete triples that match with the *Head* of a Horn rule, while the remaining triples are compressed with the RB method.

Hybrid compressors encompass logical and physical techniques, so that the three types of redundancies can be tackled. Although they combine the best of both compression methods, in practice, it is a field that has been little explored.

The graph pattern-based (GPB) compressor [40] groups the triples that share the same subject in Entitiy Description Blocks (EDB). Each EDB is described by an Entity Description Patterns (EDP), which is a concept similar to predicate families. Each EDP is encoded as a pair containing an EDP and instances that match it, constituting the simplest level of de GPB (LV0); then better patterns are acquired (LV1) by merging the EDBs. The last level in GPB (LV2) recursively joins the merged EDBs. Experiments show that, at the logical level, GPB (LV2) removes more triples (i.e., compresses more) than RB; however, its effectiveness has not been compared to physical compressors.

Finally, RDF2NormRDF [51] is not a compressor *per se*, but an attempt to normalize the RDF graph. It deals with blank node particularities and cleans duplicated RDF terms from the graph by applying several transformations rules. At the physical level, RDF2NormRDF applies another set of rules to normalize types and certain tags, such as language. Experiments show that RDF2NormRDF only outperforms HDT when dealing with small datasets.

Chapter 3

Serializing RDF in Compressed Space

Serializing RDF in Compressed Space*

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

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

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

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

Abstract

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

1 Introduction

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

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

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

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

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

1068-0314/15 $31.00 © 2015 IEEE
DOI 10.1109/DCC.2015.16

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

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

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

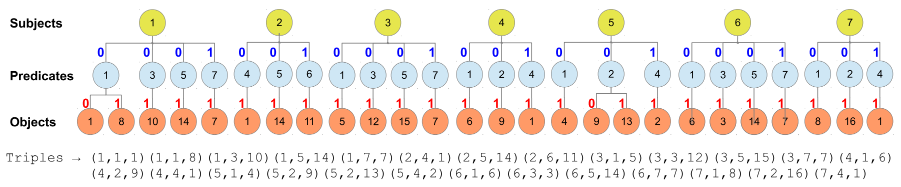
2 Background

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

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

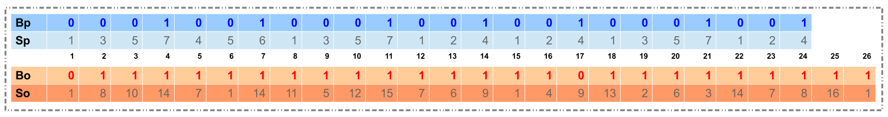
The *Triples* component encodes RDF triples as groups of three IDs: $(\mathtt{id}_s\ \mathtt{id}_p\ \mathtt{id}_o)$, where $\mathtt{id}_s$, $\mathtt{id}_p$, and $\mathtt{id}_o$ are respectively the IDs of the corresponding subject, predicate, and object terms in the *Dictionary*. The current *Triples*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

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



```
Triples → (1,1,1)(1,1,8)(1,3,10)(1,5,14)(1,7,7)(2,4,1)(2,5,14)(2,6,11)(3,1,5)(3,3,12)(3,5,15)(3,7,7)(4,1,6)
          (4,2,9)(4,4,1)(5,1,4)(5,2,9)(5,2,13)(5,4,2)(6,1,6)(6,3,3)(6,5,14)(6,7,7)(7,1,8)(7,2,16)(7,4,1)
```

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



	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
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		
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	7	1	14	11	5	12	15	7	6	9	1	4	9	13	2	6	3	14	7	8	16	1

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

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

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

State of the Art of RDF Compression

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

RDF compression may also leverage semantic redundancy. These *logical* compressors [8] discard triples which can be inferred from others, and they only encode these "primitive triples". Thus, these techniques save space because they reduce the number of triples to be encoded. In addition, they may also apply *syntactic* compression techniques. For instance, Joshi *et al.* [8] combine their approach with HDT, but their results are similar to that obtained by simply using HDT. Recently, Wu *et al.* [13] have proposed SSP, an hybrid compressor leveraging syntactic and semantic redundancy. Its results show that SSP+bzip2 slightly improves HDT+bzip2

3 Schema-based Sources of Redundancy

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

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

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

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

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

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

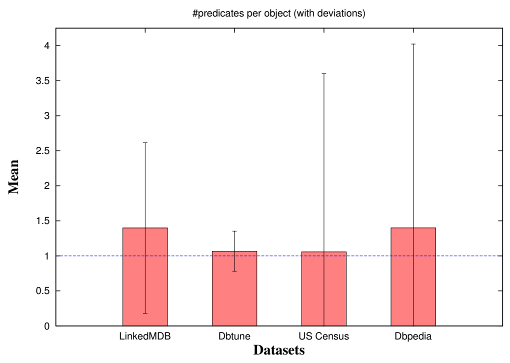


Figure 3: Number of predicates per object (mean and standard deviation).

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

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

4 Our Approach

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

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

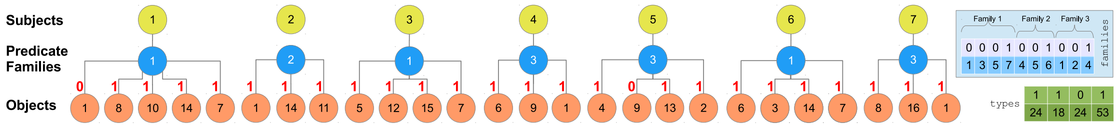


Figure 4: Forest modeling ID triples using predicate families.

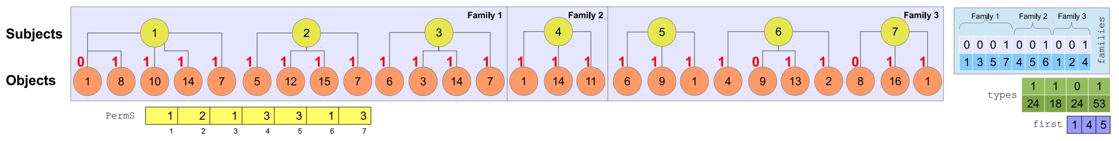


Figure 5: Forest configuration before subject reorganization.

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

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

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

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

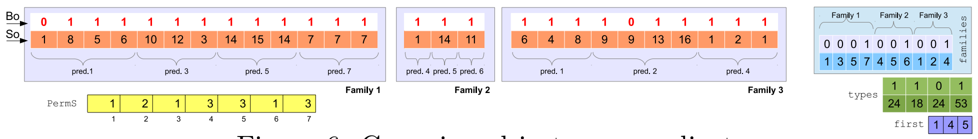


Figure 6: Grouping objects per predicate.

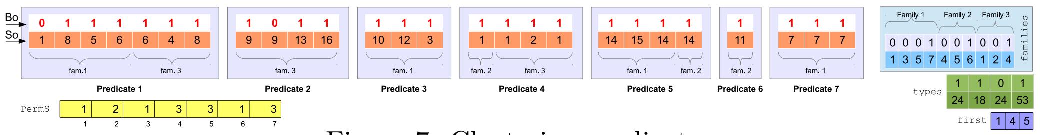


Figure 7: Clustering predicates.

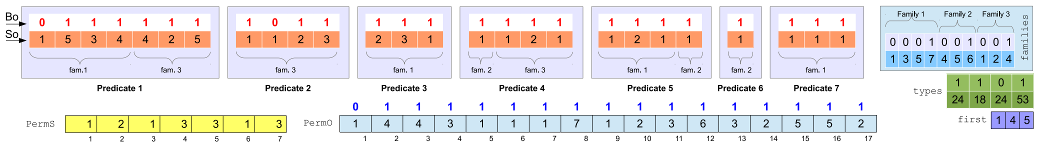


Figure 8: Replacing global by local object IDs.

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

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

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

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

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

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

Algorithm 1: Decoding algorithm.

```
for $predicate \leftarrow 1$ to $|P|$ do
    $ptr \leftarrow 1$;
    $\mathcal{F}_p \leftarrow$ families.getFamilies($predicate$);
    for $f \leftarrow 1$ to $|\mathcal{F}_p|$ do
        for $s \leftarrow$ first[f] to first[f+1]-1 do
            $subject \leftarrow$ PermS.getSubjectID($s$);
            repeat
                $object \leftarrow$ PermO.getObjectID(So[ptr]);
                $newtriple(subject, predicate, object)$;
                $ptr \leftarrow ptr + 1$;
            until Bo[$predicate$][ptr] $\neq 1$;
            $\mathcal{T}_f \leftarrow$ types.getTypes($f$);
            if $\mathcal{T}_f[1] \neq 0$ then
                for $t \leftarrow 1$ to $|\mathcal{T}_f|$ do
                    $newtriple(subject, rdf:type, \mathcal{T}_f[t])$;
```

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

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

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

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

dataset	#triples	plain (MB)	HDT (MB)	HDT++ (MB)	k^2-triples (MB)
linkedmdb	6,147,996	35.91	22.54	14.24	9.02
dbtune	58,920,361	400.36	242.05	132.10	152.27
us census	149,182,415	1,049.25	649.22	312.54	347.06
dbpedia	431,440,396	3,497.36	1,839.08	1,523.72	1,699.39

Table 2: Compression results.

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

5 Experimental Evaluation

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

We choose four different real-world RDF datasets: `linkedmdb` describes information about movies, actors, characters, etc.; `dbtune` provides music-related structured data; `us census` provides census data from the U.S.; and `dbpedia` is an RDF conversion of Wikipedia. As showed in Table 1, the `us census` is a well-structured dataset, in contrast to `dbpedia` which models many and varied types of entities. These datasets also differ in size. They comprise from $\approx$ 6 millions for `linkedmdb` and $\approx$ 431 millions for `dbpedia`. We compare the most straightforward triples encoding (referred to as `plain`): it uses three IDs per RDF triple and each one is encoded using $\log|S|$, $\log|P|$, and $\log|O|$ bits); the original `HDT` encoding [6]; our current approach: `HDT++`; and `k`^{2}`-triples` [1] which is currently the most prominent RDF compressor.

`HDT++` outperforms the original `HDT` encoding for all datasets, but the improvement is more significant for `us census`. In this case, `HDT++` uses less than the half of the space needed by `HDT`. This is an expected result because it is the most structured dataset. However, the improvements for `linkedmdb` and `dbtune` are also noticeable: `HDT++` needs $\approx$ 63% and $\approx$ 55% of the original space. For `dbpedia`, `HDT++` saves more than 300MB regarding `HDT`. The comparison regarding k^2-triples shows that `HDT++` is better for the three largest datasets: $\approx$ 10-13% less space than `k`^{2}`-triples`. This result is especially interesting by considering that `k`^{2}`-triples` is a pure RDF compressor while `HDT++` is a binary serialization format in which no explicit compression is performed. However, k^2-triples provides efficient data retrieval in the reported space requirements.

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

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

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

6 Conclusions and Future Work

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

Our `HDT++` approach leverages these features, saving up to half the space used by its `HDT` predecessor and competing on equal terms with the most effective RDF compressor, $\texttt{k}^2$`-triples`. Our achievements can be directly reused by the community since all decisions are aligned to the HDT foundations. Thus, solutions exchanging/consuming HDT can greatly reduce their storage requirements and network latencies. Our future work focuses on exploiting this approach to provide triple pattern resolution by reusing previous experiences on HDT-based retrieval [10].

7 References

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 4

RDF-TR: Exploiting structural redundancies to boost RDF compression

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>`, `<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

[☆]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)

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

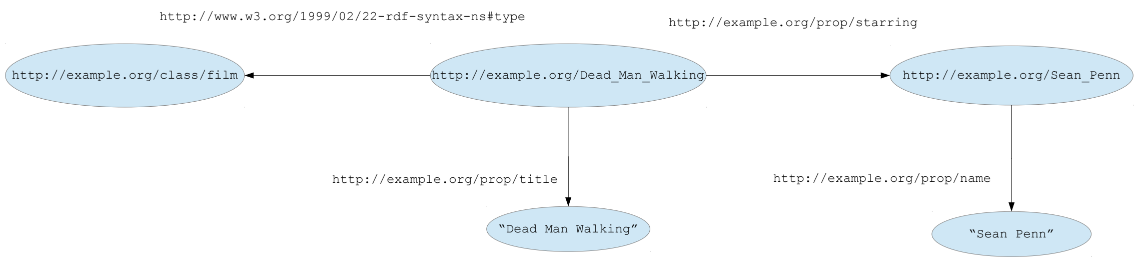

http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://example.org/prop/starring
http://example.org/class/film
http://example.org/Dead_Man_Walking
http://example.org/Sean_Penn
http://example.org/prop/title
http://example.org/prop/name
"Dead Man Walking"
"Sean Penn"

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.

as life-sciences (e.g. *Bio2RDF*), geography (e.g. *Geonames*), or general knowledge (e.g. *Wikidata*), to name but a 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

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

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 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 `k`^{2}`-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 `k`^{2}`-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 `k`^{2}`-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-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

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

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

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

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 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:

- `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]:

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

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.

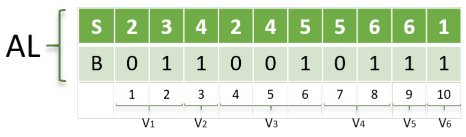


Figure 3: Example of adjacency list encoding.

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 $begin = \texttt{select}_1(AL.B, 2) + 1 = 4$, to $end = \texttt{select}_1(AL.B, 3) = 6$. Then, each operation proceeds as follows:

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

- `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 `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, `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.

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

- `outdegree`(G, v) returns $end - begin + 1$ (see Algorithm 4). In the previous example, `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 `neigh`$(G, v)[j]$. It returns the j-th direct neighbor of v, which is located at $S[begin + j - 1]$; e.g., `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

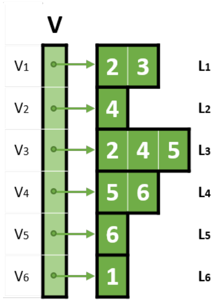


Figure 4: Example of adjacency matrix encoding using a vector of vectors.

second list is empty, and the third list has two elements. It also causes a slight modification in the `getListLimits`(v) function, which now obtains the left limit as $pos_l =$`select`$_1(v-1)-(v-1)$, and the right one as: $pos_r =$`select`$_1(v)-v$. Note that if $pos_r = pos_l$, the corresponding list is empty. In this case, `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 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.

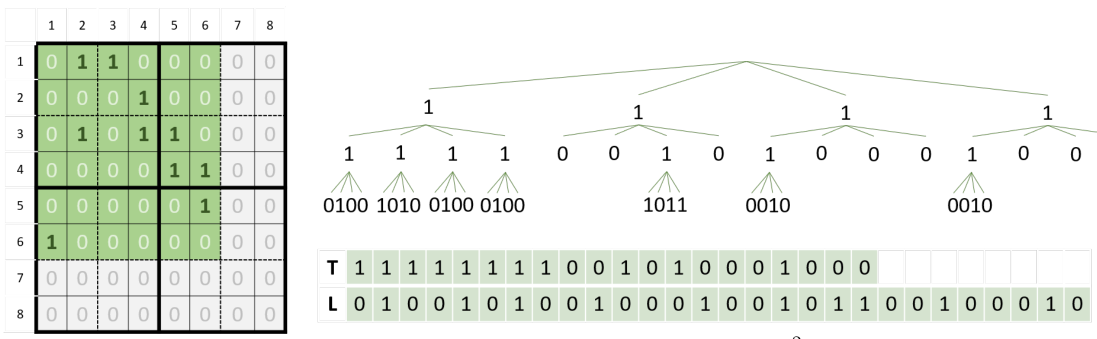


Figure 5: Example of matrix encoding using a k^2-tree.

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].

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 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 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*

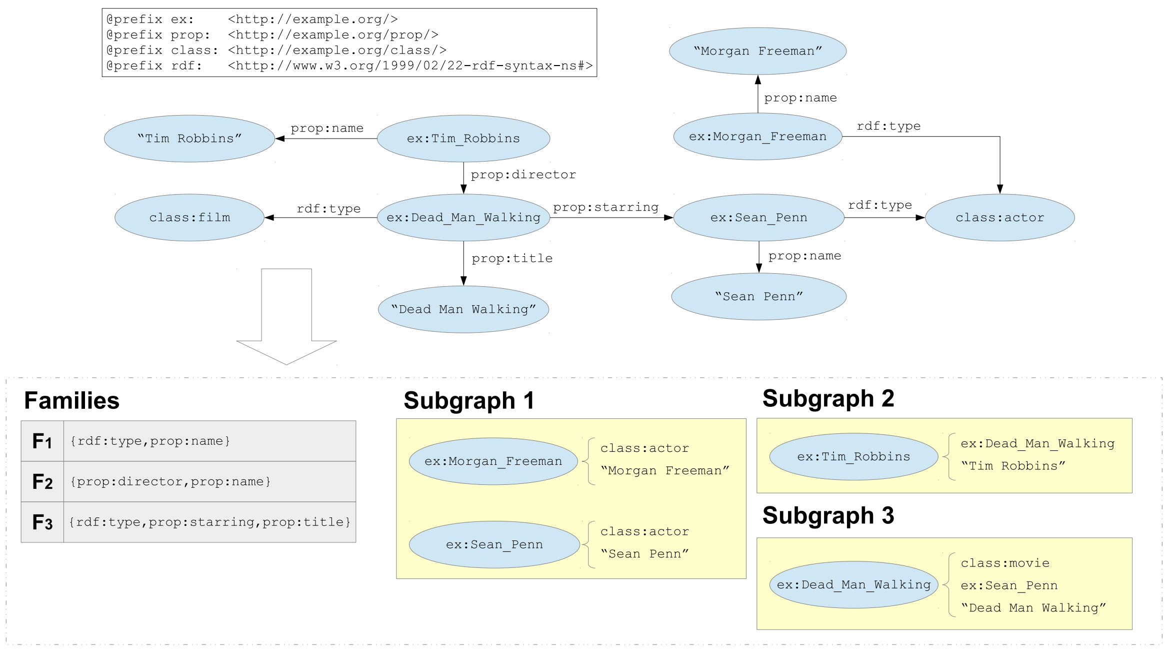


Figure 6: Example of an RDF graph and its predicate families.

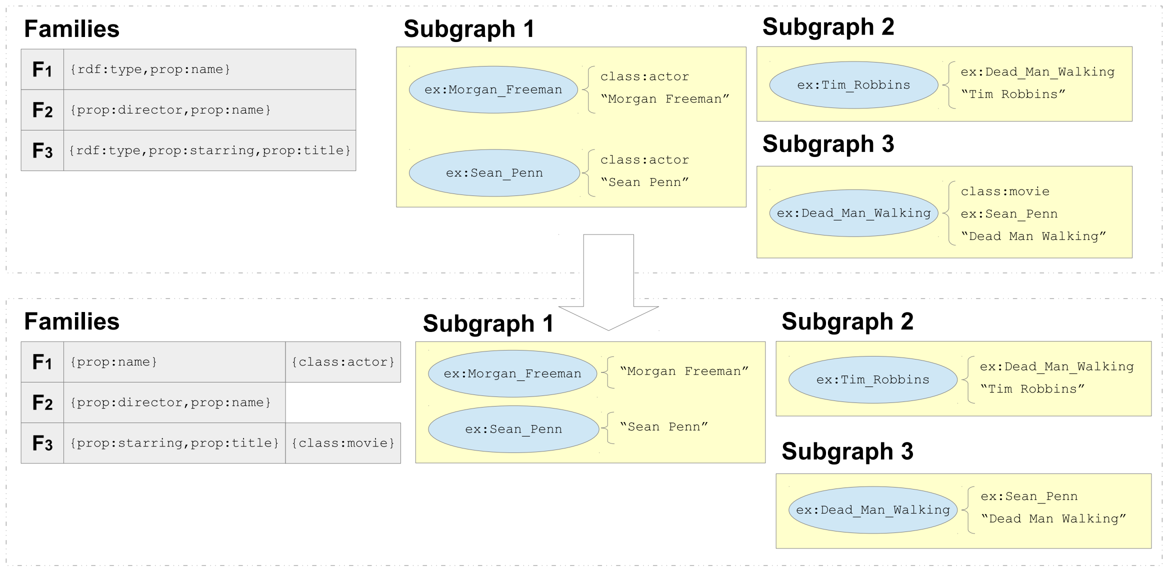


Figure 7: Integration of **`rdf:type`** values in predicate families.

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.*

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).

$\mathtt{S_1\ P_1\ O_1.\ \ S_1\ P_1\ O_8.\ \ S_1\ P_3\ O_{10}.\ \ S_1\ P_5\ O_{14}.\ \ S_1\ P_7\ O_9.}$
$\mathtt{S_2\ P_4\ O_1.\ \ S_2\ P_5\ O_{14}.\ \ S_2\ P_6\ O_{11}.}$
$\mathtt{S_3\ P_1\ O_5.\ \ S_3\ P_3\ O_{12}.\ \ S_3\ P_5\ O_{15}.\ \ S_3\ P_7\ O_9.}$
$\mathtt{S_4\ P_1\ O_6.\ \ S_4\ P_2\ O_7.\ \ S_4\ P_4\ O_1.}$
$\mathtt{S_5\ P_1\ O_4.\ \ S_5\ P_2\ O_7.\ \ S_5\ P_2\ O_{13}.\ \ S_5\ P_4\ O_2.}$
$\mathtt{S_6\ P_1\ O_6.\ \ S_6\ P_3\ O_3.\ \ S_6\ P_5\ O_{14}.\ \ S_6\ P_7\ O_9.}$
$\mathtt{S_7\ P_1\ O_8.\ \ S_7\ P_2\ O_{16}.\ \ S_7\ P_4\ O_1.}$

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

- 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).

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).

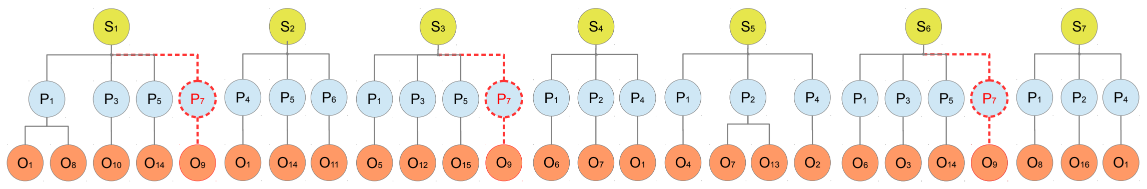
T1. Object re-mapping. We re-map objects related to the same predicate with a new sequential identifier. These new local-IDs will be assigned in global-ID order. That is, we sort all objects of a given predicate by their (original) IDs in the dictionary, and we then assign the position of each object 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 $\{$`(S`$_1$`,P`$_3$`,O`$_{10}$`), (S`$_3$`,P`$_3$`,O`$_{12}$`), (S`$_6$`,P`$_3$`,O`$_3$`)`$\}$. 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]`$= 16$.


S1 S2 S3 S4 S5 S6 S7
P1 P3 P5 P7 P4 P5 P6 P1 P3 P5 P7 P1 P2 P4 P1 P2 P4 P1 P3 P5 P7 P1 P2 P4
O1 O8 O10 O14 O9 O1 O14 O11 O5 O12 O15 O9 O6 O7 O1 O4 O7 O13 O2 O6 O3 O14 O9 O8 O16 O1
Figure 9: RDF triples organized as a forest of trees.

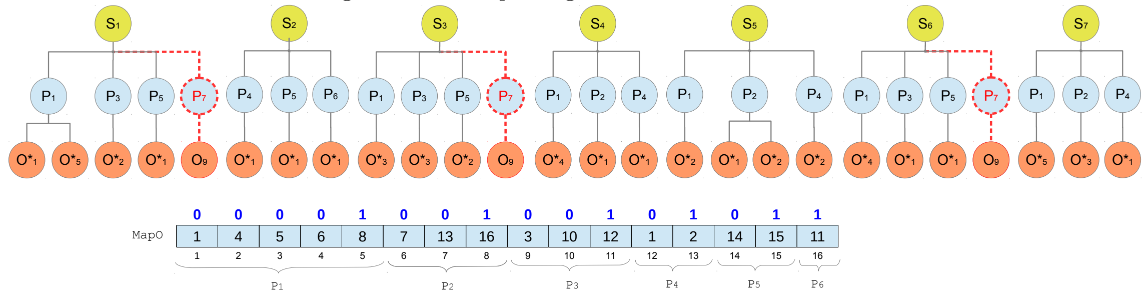


Figure 10: Object re-mapping (note that local object IDs are assigned according to the global object ID order).

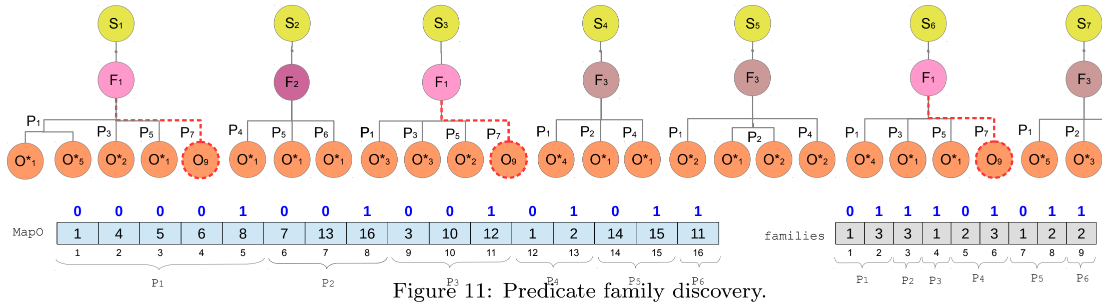


Figure 11: Predicate family discovery.

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.

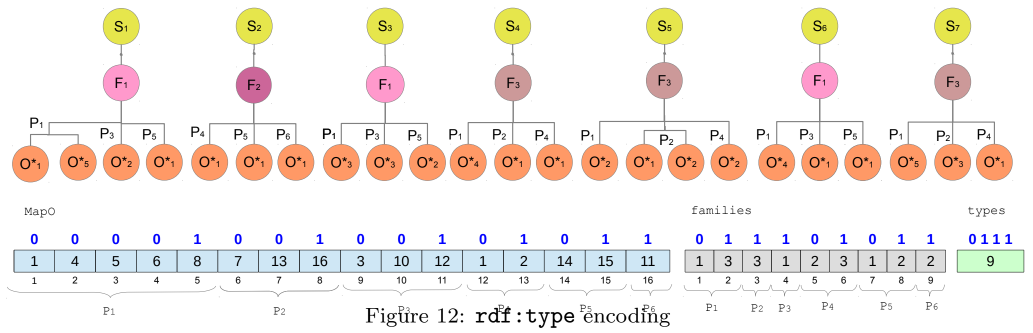


Figure 12: `rdf:type` encoding

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 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 $\mathcal{F}_j = \{\mathcal{S}_k \dots \mathcal{S}_l\}; j \in [1, |\mathcal{F}|], \{k, l\} \in [1, |\mathcal{S}|], k < \cdots < l$. 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.

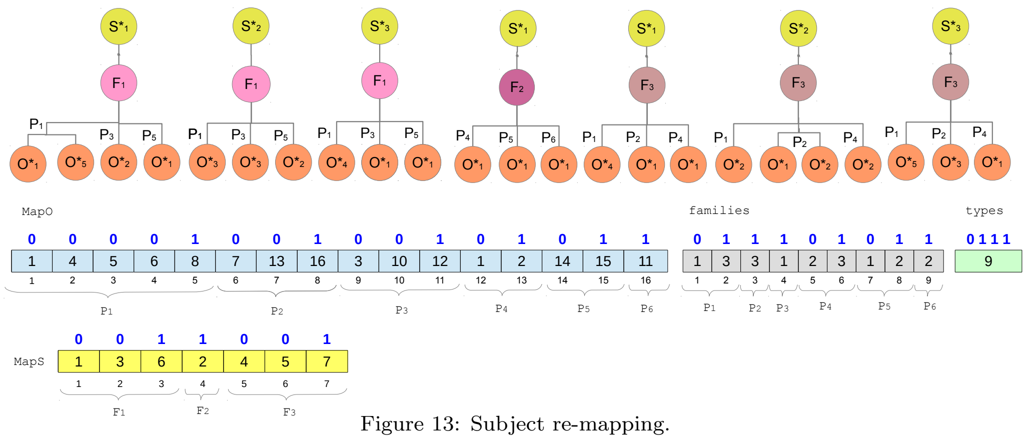


Figure 13: Subject re-mapping.

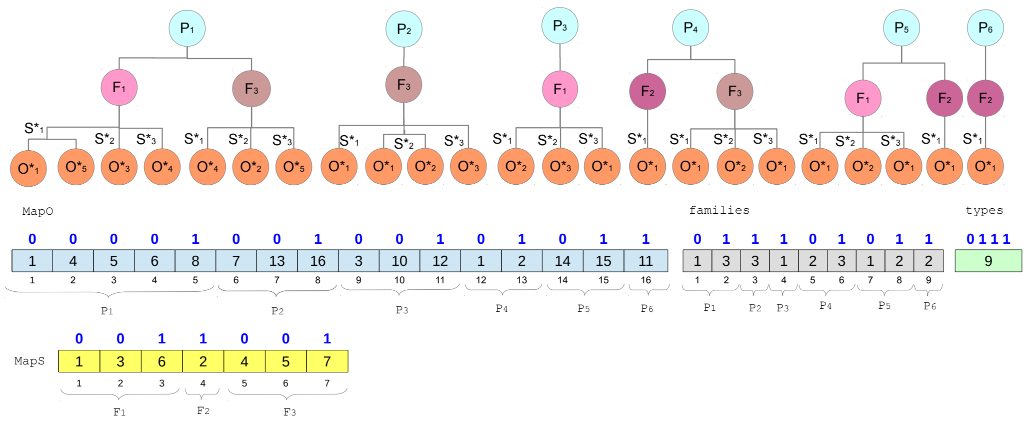


Figure 14: Triples rewritten with the RDF-Tr algorithm.

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.

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

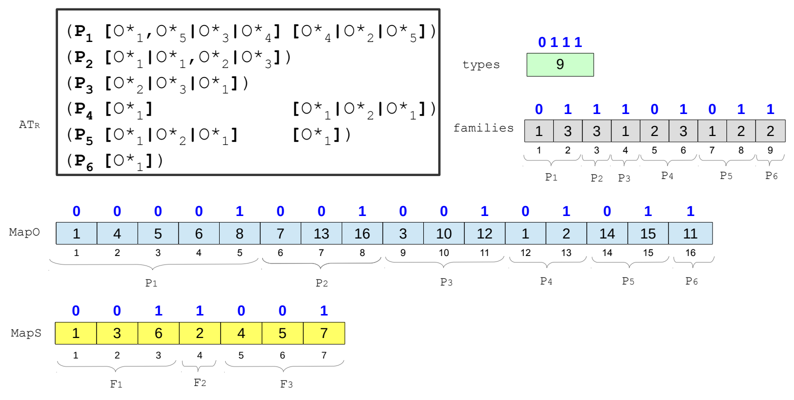


Figure 15: Predicate family based (adjacency list) encoding.

`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. 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* (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.

Algorithm 5: Decoding algorithm.

```
for $predicate \leftarrow 1$ to $|P-1|$ do
    $ptrSubject \leftarrow 1$;
    $\mathcal{F}_p \leftarrow$ neigh(families, $predicate$);
    for $f \leftarrow 1$ to $|\mathcal{F}_p|$ do
        $family \leftarrow \mathcal{F}_p[f]$;
        $\mathcal{T}_f \leftarrow$ neigh(types, $family$);
        $\mathcal{S}_f \leftarrow$ neigh(MapS, $family$);
        for $s \leftarrow 1$ to $|\mathcal{S}_f|$ do
            $subject \leftarrow \mathcal{S}_f[s]$;
            $\mathcal{O}_s \leftarrow$ ATr.getObjects($predicate, ptrSubject$);
            $ptrSubject \leftarrow ptrSubject + 1$;
            for $o \leftarrow 1$ to $|\mathcal{O}_s|$ do
                $object \leftarrow$ neigh(MapO, $predicate$)$[\mathcal{O}_s[o]]$;
                $newtriple(subject, predicate, object)$;
            if $\mathcal{T}_f \neq \emptyset$ then
                for $t \leftarrow 1$ to $|\mathcal{T}_f|$ do
                    $newtriple(subject,$ rdf:type$, \mathcal{T}_f[t])$;
```

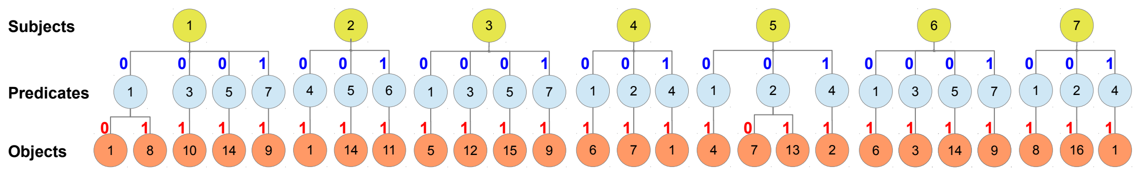
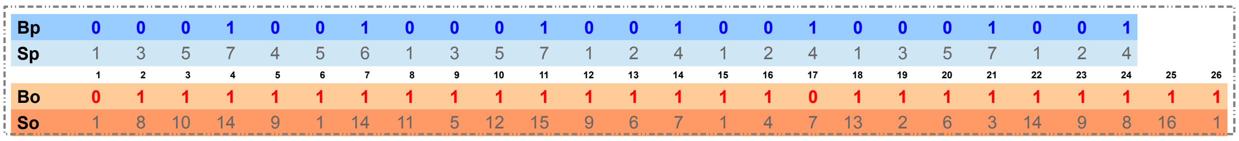


Figure 16: Forest of trees modeling ID triples in HDT.



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.

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.

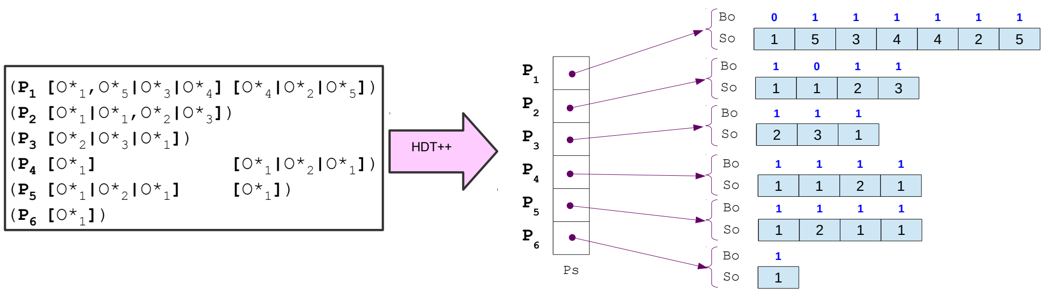


Figure 18: HDT++ implementation of ATr.

```
Algorithm 6: HDT++: getObjects(predicate, ptrSubject).
1 return neigh($P_s[predicate], ptrSubject$);
```

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, 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. K^2-triples++

We refer to k^2-triples++ as the integration of k^2-triples and RDF-Tr. As in the previous section, we first introduce the foundations of k^2-triples and then we describe the RDF-Tr integration.

7.1. *k^2-triples*

Similarly to HDT, k^2-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. k^2-triples takes advantage of the low number of predicates used in an RDF dataset and partitions it vertically. That is, k^2-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 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]These structures are not represented in Figure 18 for simplicity, but their configuration is the same as in Figure 15.

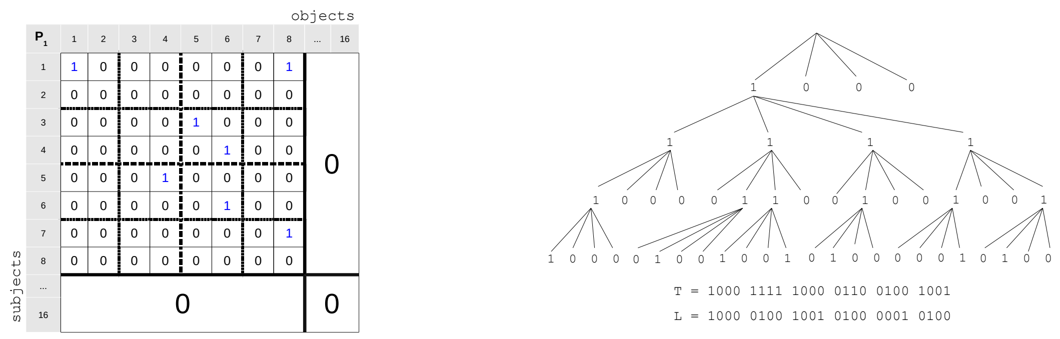

objects
subjects
P1
T = 1000 1111 1000 0110 0100 1001
L = 1000 0100 1001 0100 0001 0100

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

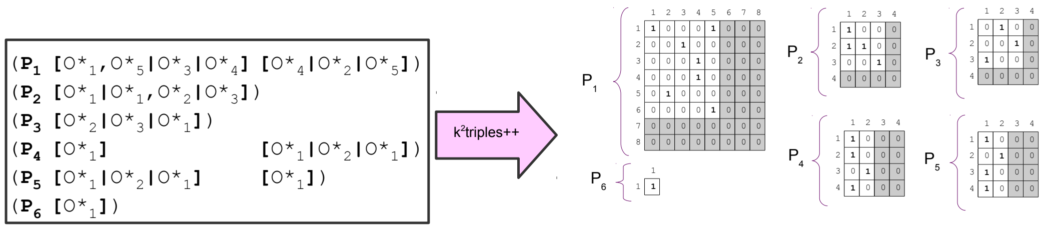


Figure 20: K^2triples++ implementation of ATr

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

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

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

Transforming k^2-triples into `k`^{2}`-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 `k`^{2}`-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 `k`^{2}`-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.

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

[8]The code of the prototype is publicly available at `https://github.com/antonioillera/HDTpp-src`
[9]`https://github.com/simongog/sdsl-lite`

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.
- `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 bit-sequences 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 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

[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`

Dataset	**#triples**	$\lvert S \rvert$	$\lvert P \rvert$	$\lvert O \rvert$
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	$\lvert F \rvert$	**#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.

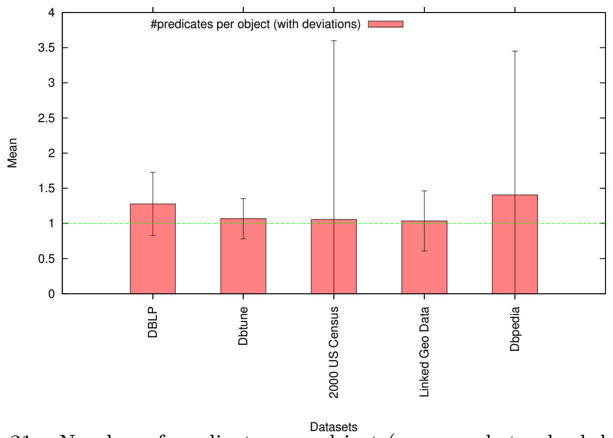


Figure 21: Number of predicates per object (mean and standard deviation).

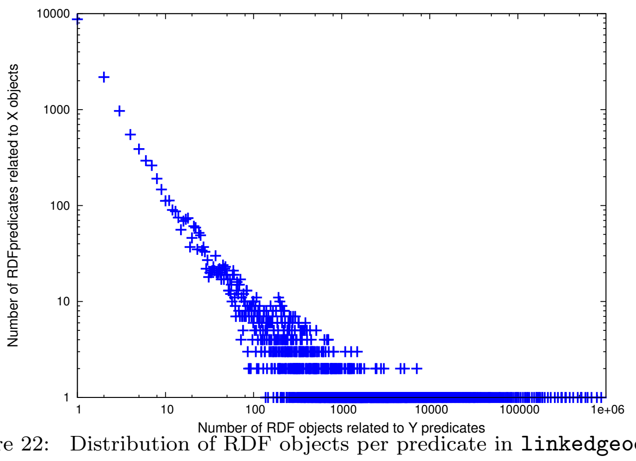


Figure 22: Distribution of RDF objects per predicate in `linkedgeodata`.

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

	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.

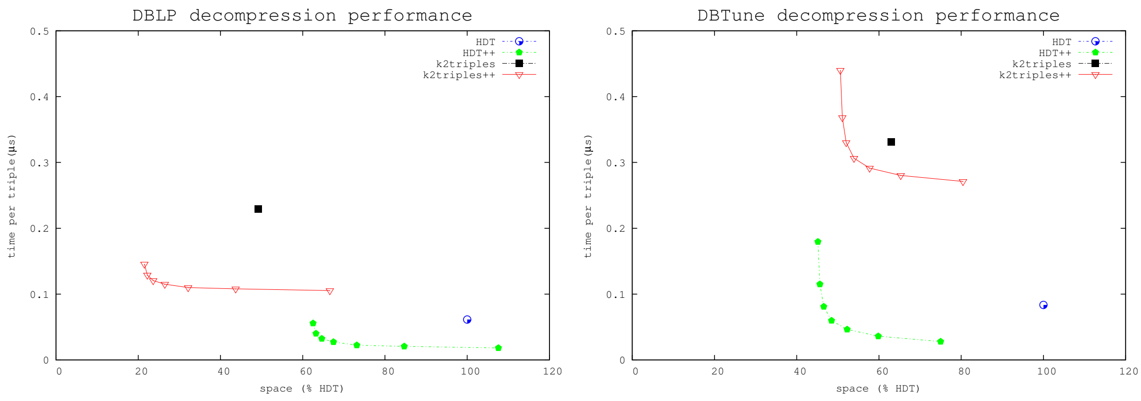


Figure 23: dblp and dbtune tradeoffs

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*

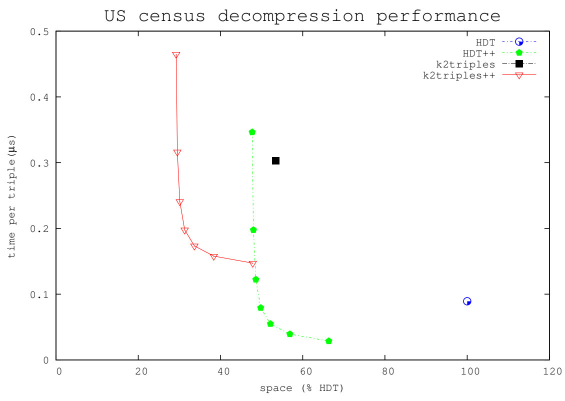
This section analyzes the experimental results when applying RDF-Tr to the well-known HDT and k^2-triples RDF syntactic compressors, leading to `HDT++` and `k`^{2}`-triples++` respectively. Experiments were performed in a -commodity server- Intel Xeon E5645@2.4GHz, 96GB DDR3@1066Mhz. Reported (elapsed) 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).

`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



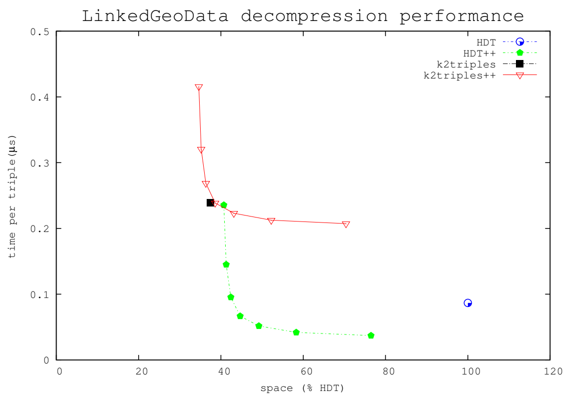


Figure 24: US census and linkedgeodata tradeoffs

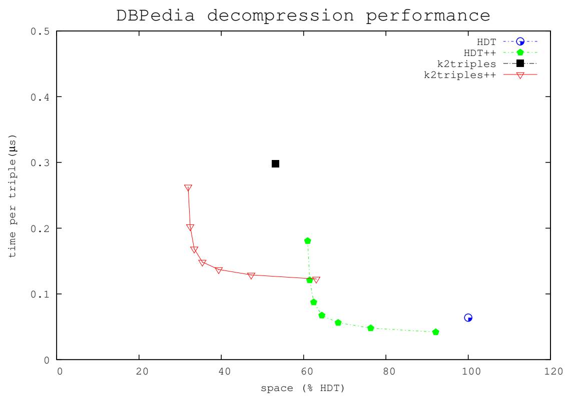


Figure 25: dbpedia tradeoffs

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 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 transformation, 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

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

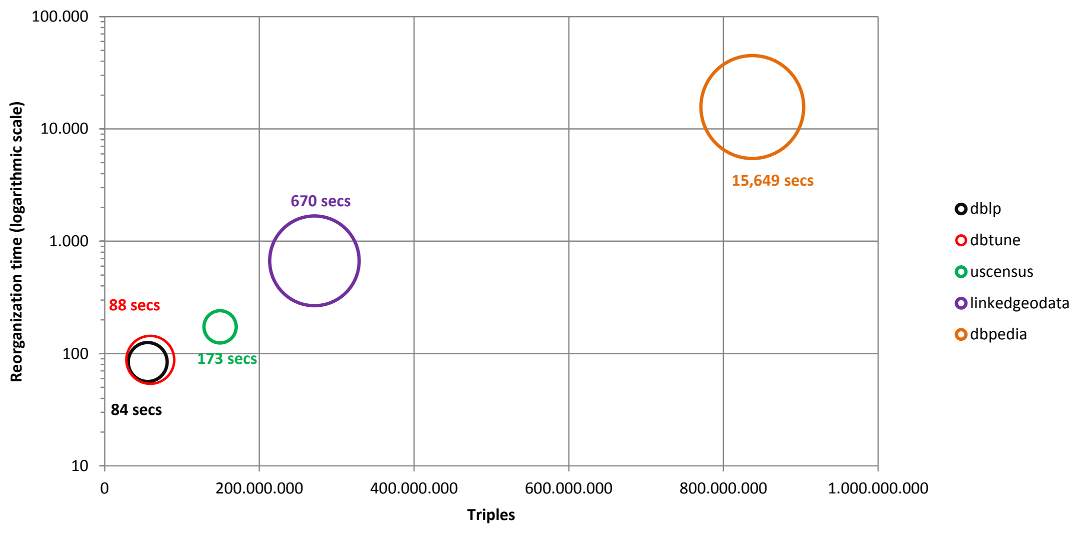


Figure 26: Datasets reorganization time.

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 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.

Chapter 5

iHDT++: Improving HDT for SPARQL Triple Pattern Resolution

iHDT++: Improving HDT for SPARQL Triple Pattern Resolution

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

[a] *Department of Computer Science, University of Valladolid, Spain*
[b] *Vienna University of Economics and Business & Complexity Science Hub Vienna, Austria*
[c] *University of A Coruña, Database Lab, CITIC, Spain*

Abstract. RDF self-indexes compress the RDF collection and provide efficient access to the data without a previous decompression (via the so-called SPARQL triple patterns). HDT is one of the reference solutions in this scenario, with several applications to lower the barrier of both publication and consumption of Big Semantic Data. However, the simple design of HDT takes a compromise position between compression effectiveness and retrieval speed. In particular, it supports scan and subject-based queries, but it requires additional indexes to resolve predicate and object-based SPARQL triple patterns. A recent variant, HDT++, improves HDT compression ratios, but it does not retain the original HDT retrieval capabilities. In this article, we extend HDT++ with additional indexes to support full SPARQL triple pattern resolution with a lower memory footprint than the original indexed HDT (called HDT-FoQ). Our evaluation shows that the resultant structure, iHDT++, requires $70-85\%$ of the original HDT-FoQ space (and up to $48-72\%$ for an HDT Community variant). In addition, iHDT++ shows significant performance improvements (up to one level of magnitude) for most triple pattern queries, being competitive with state-of-the-art RDF self-indexes.

Keywords: HDT, RDF compression, Triple pattern resolution, SPARQL, Linked Data.

1. Introduction

The *World Wide Web* is a network of documents, in which nodes (*web pages*) contain pieces of information intended for human consumption, and the edges relate this information through *links*, which facilitate navigation among pages. This document-centric information architecture does not facilitate access to raw data, hindering to automate different processes. The *Web of Data* arises as a response to this situation and offers, on the own infrastructure of the Web, mechanisms to represent and interconnect data with sufficient semantics and level of granularity to allow automatic processing [2]. RDF *(Resource Description Framework)* [24] plays a fundamental role in the Web of Data.

RDF models and interconnects data using ternary sentences (*triples*) formed by a *subject* (S), a *predicate* (P), and an object (O). These RDF triples can be interpreted as directed graphs in which subjects and objects act as nodes and predicates are the edges between them. The flexibility of RDF has facilitated its use as a standard *de facto* for the publication of raw data on the Web, and, more recently, Knowledge Graphs [3]; *DBpedia* or *Bio2RDF* publish billions of triples, being a clear example of the volume reached by RDF collections and, in turn, the scalability challenges that entail its management and consumption. One of these scalability problems is the way RDF datasets are *serialized*. Traditionally, *"flat"* formats (like XML) have been used, whose verbosity is a limiting factor when managing Big Semantic Data. The alternative is to use binary formats that encode the RDF datasets according to its structural and/or semantic properties.

HDT (Header-Dictionary-Triples) [7] is positioned in this scenario and proposes a binary format that exploits RDF redundancy [14]. HDT obtains compression ratios comparable to those reached by `gzip`, and it reports competitive performance for scan queries and subject-based retrieval [8], with no prior decompression. In addition, *HDT-FoQ* (Focused on Querying)

[*]Corresponding author. Antonio Hernández Illera, Department of Computer Science, University of Valladolid. Campus María Zambrano, 40006, Segovia, Spain. E-mail: antonio.hi@gmail.com.

[15] adds two indexes (either loaded into memory or mapped from disk) on top of HDT to allow for full SPARQL [21] triple pattern (TP) resolution.[1]

HDT has been adopted in the Web of Data because of its simplicity and a competitive space/time tradeoff, taking a key role in the development of client-side query processors such as Triple Pattern Fragments [25] and SAGE [17]. However, both HDT and HDT-FoQ are limited by a design that emphasizes simplicity of representation and disregards other sources of redundancy. HDT++ [11] modifies that design and implements a reorganization of triples that partially eliminates structural redundancies. Specifically, HDT++ takes advantage of the fact that subjects of the same type are described by similar sets of properties and that their value ranges have little overlap. HDT++ notably improves the compression ratios obtained by HDT, as well as its decoding speed. Yet, it does not provide the necessary mechanisms to solve SPARQL TPs.

In this paper, we present iHDT++ an enhanced representation that allows HDT++ files to be efficiently queried. In particular, we extend the existing HDT++ structures with additional information to resolve predicate-based and subject-based TPs (i.e. those in which the predicate or subject are provided, respectively). Then, we provide a new object-based index that completes the iHDT++ proposal and enables full SPARQL TP resolution. Our experiments show that iHDT++ uses around $70 - 85\%$ of the memory footprint of HDT-FoQ, largely outperforming most of the TPs (e.g. the challenging predicate-based retrieval, (?P?)). The space differences are even more noticeable with the HDT Community version (48-72%), a practical proposal to speed up predicate-based issues (presented in Section 2.3). iHDT++ also shows competitive space/time tradeoffs with state-of-the-art RDF self-indexes, k^2-triples and RDFCSA.

The rest of the article is organized as follows. Section 2 presents the background of iHDT++. Section 3 describes the structures added by iHDT++ on top of HDT++, and explains how these can be used to resolve SPARQL TPs. Section 4 compares the performance of iHDT++ with the existing HDT-based solutions and the most promising RDF self-indexes. Finally, our conclusion and future work are discussed in Section 5.

[1] A *TP* is an RDF triple in which any of its components can be variable (? is used to indicate components that are variables): (SPO), (SP?), (S?O), (S??), (?PO), (?P?), (??O), and (???).

2. Background

This section provides the basic background of the paper. We introduce the notion of compact data structure [18], with particular attention to those structures used by the HDT-based approaches and iHDT++. Compact data structures are also at the core of the most competitive RDF compressors, including efficient RDF self-indexes. We also review state-of-the-art RDF compression techniques, and we delve into particular details of HDT-based approaches, which set the foundations of our proposal.

2.1. Compact Data Structures

A compact data structure [18] proposes a data arrangement that uses an amount of space close to the theoretical optimal number of bits (required to preserve the data), while providing efficient functionality with no prior decompression. Thus, a compact data structure compresses the original data and allows it to be queried and manipulated in compressed form.

The main blocks of compact data structures are *functional bitsequences*, explained as follows.

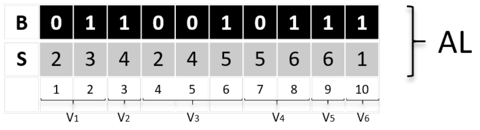
Bitsequences. A bitsequence $B[1, n]$ is an array of n bits that provides three basic operations:

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

iHDT++ uses a "plain bitsequence" that implements Clark's approach [6], which adds additional structures on top of B to efficiently resolve rank and select (access is directly performed on the bit array in constant time). Bitsequences can be compressed [18] to save space requirements, but none of the RDF compressors analyzed in this paper use them.

Sequences. A sequence $S[1, n]$ is a generalization of a bitsequence, whose elements $S[i]$ (i.e. *symbols*) come from to an *alphabet* $\Sigma = [1, \sigma]$. They support the same operations: $\texttt{access}(S, i)$ returns the symbol stored at $S[i]$, while $\texttt{rank}_s(B, i)$ and $\texttt{select}_s(B, j)$ allow any symbol $s \in \Sigma$ to be queried.

The simplest sequence implementation is an array that encodes each symbol using $\lceil log_2(\sigma) \rceil$ bits. This



B	0	1	1	0	0	1	0	1	1	1
S	2	3	4	2	4	5	5	6	6	1
	1	2	3	4	5	6	7	8	9	10

Fig. 1. Example of adjacency list encoding.

"plain sequence" answers access(S, i) in $O(1)$, by accessing $S[i]$, but it does not resolve rank and select efficiently. The *wavelet tree* [10] proposes an alternative for sequence encoding. It organizes symbols in a balanced tree of height $h = \log(\sigma)$, comprising h bitsequences of n bits each. It requires $n \log_2(\sigma) + o(n)$ bits of space, using plain bitsequences, and answers access, rank, and select in $O(h)$.

Sequences of symbols are highly compressible in many cases; e.g. posting lists in Information Retrieval or adjacency lists in (Semantic) Web Graphs are usually gap-encoded [13] to exploit that symbols are sorted, in increasing order, within the sequence. Different forms of *variable length compression* [23] can also be adopted to compress the sequence of symbols. They compress sequences at the cost of slower access, as the symbols must be previously decompressed.

Adjacency Lists. Adjacency lists are typically used to encode *graphs*. Given the RDF scope of this paper, we hereinafter focus on *directed graph* encoding. A directed graph $G = (V, E)$ is composed of a set of vertices, V, and the set of edges, $E \subseteq V \times V$. Typically, the *direct neighbors* of a vertex v refer to all vertices that can be reached from v, i.e. $\{(v, u) \in E\}$. Conversely, the set of *reverse neighbors* of a vertex v contains vertices u such that $\{(u, v) \in E\}$.

Figure 1 shows the adjacency list encoding for a graph with $n = 6$ vertices and a set of $e = 10$ edges: $E = \{(1,2),(1,3),(2,4),(3,2),(3,4),(3,5),(4,5),(4,6),(5,6),(6,1)\}$. Note that the structure AL concatenates all adjacency lists into a single sequence, S, and a bitsequence, B, in which 1-bits mark the last element of the list of each vertex. In the example, the list for the first vertex v_1 is encoded in $S[1,2]$, the list for v_2 in $S[3]$, and so on. The direct neighbors of v_1 are $\{v_2, v_3\}$, and the reverse neighbors of v_2 are $\{v_1, v_3\}$.

Adjacency lists are optimized to obtain direct neighbors for a vertex v: neigh(G, v), and to check if two vertices v and u are connected: adj(G, v, u), which returns the position of u in the list if $(v, u) \in E$, or -1 otherwise. Both operations are implemented using select on B and then access to the corresponding positions in S, but this organization is not well suited for reverse neighbors queries, unless the transposed graph is encoded, doubling the required space [18].

Self-Indexes. A self-index is a *compressed* index that provides search functionality over a data collection and contains enough information to reproduce it [19]. Thus, a self-index can replace the original data collection by a compressed representation that also enables efficient retrieval operations to be performed. Although self-indexes were originally designed for text collections, they are currently used to manage different types of data, including RDF.

In the scope of this paper, we refer the k^2-tree [5], a highly compressed binary matrix that is used for graph encoding and supports efficient direct and reverse neighbors queries, and CSA [22], a fully-functional compressed suffix array.

2.2. RDF Compression

RDF compressors detect and remove redundancy at *symbolic*, *syntactic*, and/or *semantic* levels [20], reporting impressive space savings, and enabling efficient management of big semantic data [14].

HDT [8] was originally devised as binary serialization format for RDF, but it has been used as RDF compressor due to its compactness (similar to gzip). HDT also allows for basic, but efficient retrieval functionality. This feature was further improved by HDT-FoQ [15], a compact data structure configuration that enables full SPARQL TPs resolution to be performed on top of HDT files, with no prior decompression. This functionality was rapidly adopted, making HDT a core component of state-of-the-art client-side query processors such as Triple Pattern Fragments [25] and SAGE [17]. More recently, HDT++ [11] revisited HDT to reduce its memory footprint, but the resulting approach did not retain the retrieval capabilities of HDT-FoQ. More details about HDT are provided in Section 2.3.

RDF self-indexes [14] detect and remove syntactic redundancy underlying to the graph structure of RDF. These self-indexes support full SPARQL TPs resolution, like HDT-FoQ, but their optimized configurations of compact data structures make them more competitive in terms of space. K^2-triples [1] partitions the RDF dataset by predicate and, for each predicate, it models pairs *(subject, object)* as binary matrices where $[i, j] = 1$ mean that the *i-th* subject and the *j-th* object are connected by the given predicate. The resulting matrices are very sparse and can be effectively compressed using k^2-trees [5]. RDFCSA [4] models the RDF dataset as a text, in which subjects precede lexicographically predicates and objects. This "text" is then indexed using a compressed suffix array (CSA)

[22], which ensures efficient data retrieval. Nevertheless, this organization promotes subject-based queries, which are more efficient than the remaining SPARQL TPs. Both self-indexes are included in our experimental setup and compared to iHDT++ (see Section 4).

Finally, note that other RDF compressors purely focus on space reduction and disregard search functionality [14], which is our core contribution.

2.3. HDT-based Approaches

HDT [8] is a binary serialization format that organizes the content of an RDF dataset into two components (*Dictionary* and *Triples*), which are primarily responsible for the effectiveness of HDT. On the one hand, the *Dictionary* faces the symbolic redundancy of an RDF graph providing a compressed catalog with the terms used in the nodes and edges of the RDF graph, assigning a unique identifier (ID) to each of them. These IDs are used to encode the structure of the graph in the *Triples* component. In this paper, we leave aside Dictionary compression and retrieval [16], as it is orthogonal to our current approach, and we focus on optimizing the *Triples* component.

The *Triples* (in the form of IDs) conform a forest with *subject*-rooted trees and *(predicate, object)* sorted branches. As shown in Figure 2, the content of these trees is stored in two correlated adjacency lists, that represent the *predicates* of each subject, and the *objects* of each subject-predicate pair.[2] The HDT adjacency list implementations encompass a plain sequence (i.e. an integer array) and a plain bitsequence [9], where 1-bits mark the end of each list; i.e the last descendent of a branch. This organization makes triples decompression efficient and facilitates access *per subject* (i.e. in SPO order), but prevents the rest of SPARQL TPs from being efficiently resolved.

2.3.1. HDT-FoQ (Focused on Querying)

HDT-FoQ [15] enhances HDT files with two additional indexes to provide full TPs resolution. On the one hand, it replaces the sequence S_p (in the adjacency list of predicates) by a wavelet tree [10], which provides indexed access by predicate (PSO order). It adds a little space overhead, but ensures that all predicate-based accesses are performed in logarithmic time (with the number of predicates). On the other hand, HDT-FoQ defines an object-index in the form of adjacency list (OPS-order). It keeps track of the positions of each object (in the adjacency list of objects), enabling fast object-based TPs. However, this object index requires non-negligible space, reducing the overall HDT-FoQ effectiveness.

Although HDT-FoQ reports competitive space-time tradeoffs, it is worth noting that its performance is not competitive for the TP that only binds the predicate: (?P?). In this case, predicate occurrences are performed via select operations over the wavelet tree, which suffer from scalability problems with a medium-large number of predicates. A community version of HDT-FoQ, referred to as *HDT Community* hereinafter, solve this issue pragmatically. First, it removes the wavelet tree and restores the original plain adjacency list of predicates. Then, it uses the transposed version of this latter to speed up predicate-based queries. Thus, this alternative improves predicate-based queries, but increases space requirements.

2.3.2. HDT++

HDT++ [11] proposes an alternative serialization for RDF datasets that optimizes the HDT effectiveness by applying the RDF-TR transformation [12]. RDF-TR preprocesses the HDT Triples component (see Figure 2) to detect and eliminate redundancy at various levels, using three types of transformations.

Object-based transformation. The ranges of objects related to different predicates tend to be disjoint, i.e. *an object does not usually relate to more than one different predicate* [11]. This fact enables objects to be *locally* identified within the range of each predicate, hence using lower IDs to encode each object. It reduces drastically the number of bits used to encode object occurrences, but requires a *mapping structure* (referred to as MapO) to translate the new local IDs to the original ones. MapO is an adjacency list that encompasses (in increasing order) the original IDs of the objects related to each predicate. Figure 3 illustrates the MapO configuration for the triples in Figure 2: predicate 1 is related to the original object IDs {1,4,5,6,8}, predicate 2 with the objects {7,13,16}, etc. MapO uses the neigh primitive, of the adjacency list structure, to map local IDs to their global counterparts.

Predicate-based transformations. RDF does not restrict how entities are described, but subjects are usually described using common sets of properties. For instance, in the graph in Figure 2, subjects 1, 3 and 6 are described with the same properties {1,3,5,7},

[2]In Figure 2, we highlight the triples involving the predicate rdf:type, as they will have a special treatment in HDT++.

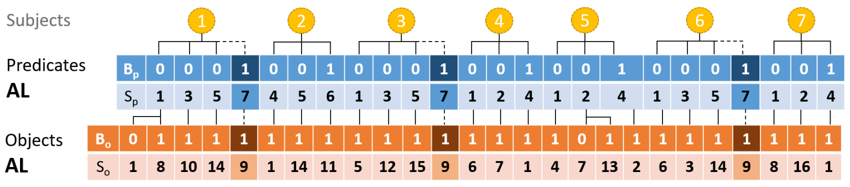


Fig. 2. Organization of *Triples* component in HDT (note that only *predicates* and *objects* adjacency lists are preserved).

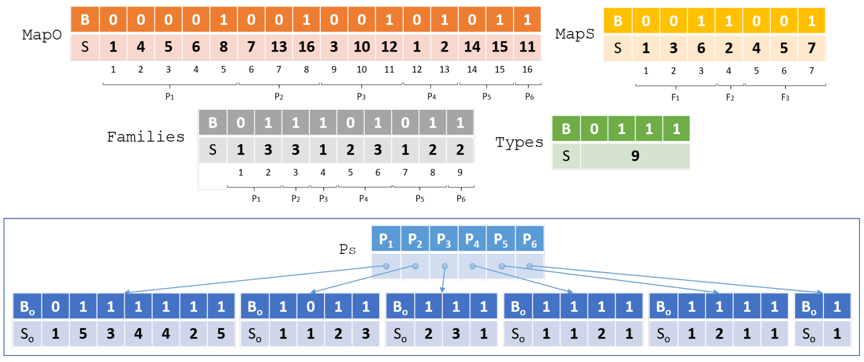


Fig. 3. HDT++ *Triples* component.

or subjects 4, 5, and 7 use the properties {1,2,4}. RDF-TR determines these *predicate families* and assigns them a unique identifier in [1,|F|]. In our example, there are three families: F_1 = {1,3,5,7}, which describes subjects 1, 3 and 6; F_2 = {4,5,6}, which describes subject 2; and F_3 = {1,2,4}, which describes subjects 4, 5, and 7. A new adjacency list, called Families, preserves the families in which each predicate is used. As shown in Figure 3, the first predicate is present in families {1,3}, predicate 2 is only in family {3}, etc. Families also uses neigh to retrieve the list of families for a given predicate.

The repetitions of the predicate families are even more explicit with the use of the predicate rdf:type. In these cases, it is quite likely that *subjects of the same type are described using the same set of predicates*. RDF-TR considers the existence of *"typed" predicate families*, i.e. families that declare some value for the predicate rdf:type, and enhances the definition of the family with the value(s) of this predicate. This decision avoids triples tagged with rdf:type to be explicitly encoded. Managing typed families requires an additional adjacency list structure: types, which preserves the type values of each family. Figure 3 illustrates this structure and encodes[3] that the first family is typed with the object 9. Finally, note that HDT++ also maps rdf:type to the last predicate ID; in our example, it is identified using the ID $|P| = 7$.

Subject-based transformation. Each subject can be now described by a predicate family, hence all subjects of the same family have the same connection structure. RDF-TR exploits this by grouping subjects of the same family, which are now *locally* re-encoded within their corresponding family. This decision requires an additional mapping structure (MapS) to translate the new local subject IDs to their corresponding counterparts. As shown in Figure 3, it is implemented as an adjacency list that arranges subject IDs per family; e.g. family 1 is related to subjects 1, 3, and 6, which correspond to local subjects 1, 2 and 3 (for such family).

The previous transformations allow triples to be serialized in the form of *Subject-Family-Object* trees, with the local ID objects (per predicate) and local ID subjects (per family). However, it is a flat representation in which each subject is connected to a single family. RDF-TR proposes a final transformation to obtain a bushy (and more compressible) encoding in the form *Predicate-Family-Object*. Each tree is now rooted by a predicate, which is connected to objects (in leaves) by the corresponding family. Subjects are implicitly encoded in this representation, thanks to the family-based grouping and the local subject IDs. The structure Ps is required to implement this encoding. As shown in Figure 3, it is a vector of $|P|$ adjacency lists (one per predicate), called P_s in which sequences S_o preserve local

[3]In this case, the bitsequence implements a slightly different encoding to allow empty lists, as some families may not be typed.

object IDs and bitsequences B_o encodes relationships between local objects and subjects, within the scope of each predicate. Ps provides the getObjects(p, pos) operation, which retrieves the list of objects starting in position *pos* for the predicate p (see [12] for additional details).

Finally, note that the inner sequences of MapS and MapO are gap-encoded (with parameterizable samples) and then compressed using Elias-Delta [23]. The remaining adjacency lists are encoded using plain sequences and bitsequences. The experiments reported in [12] showed that HDT++ is faster than HDT for triple scanning (decompression), while it uses less than half the HDT space for more-structured datasets. However, HDT++ does not retain the HDT-FoQ retrieval capabilities, so it cannot be directly used to replace the current HDT-based infrastructure in query processors.

3. iHDT++

HDT++ ensures efficient data *scan*, i.e. it resolves the (???) TP. In contrast, subject-based and predicate-based TP can be resolved in a non-efficient manner, and object-based TPs are practically discarded (they might require a full scan). iHDT++ transforms HDT++ into a query processor for SPARQL TPs. We enhance the existing structures with additional information to ensure subject and predicate-based TPs to be efficiently resolved. In addition, a new index, iObjects, is proposed to resolve object-based TPs.

3.1. Additional Data Structures

HDT++ uses *adjacency lists* to implement their components. These structures are optimized to obtain direct neighbors for a given vertex v, but are inefficient to retrieve the reverse neighbors of a v (i.e. vertices u such that $(u, v) \in E$). However, reverse neighbor operations are needed to resolve SPARQL TPs, hence MapS, MapO, and Families must be enhanced with their transposed structures.

Transposed Structures. MapO arranges object IDs by predicate, allowing local objects to be mapped to their original IDs. This operation is useful for decoding purposes, but is not enough for TPs resolution because triple patterns use global IDs instead. iHDT++ proposes to use the transposed of MapO (referred to as MapO′) to list the predicate(s) of each object (i.e. usually just one). MapO′ is implemented as an adjacency

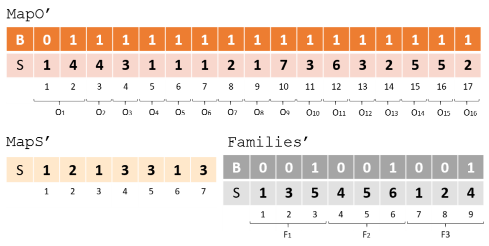


Fig. 4. Transposed structures of iHDT++.

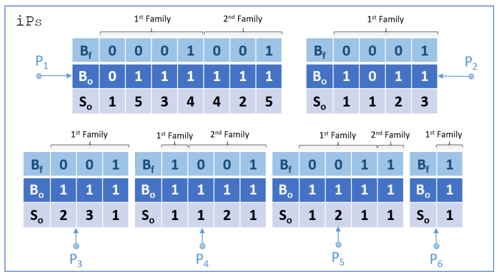


Fig. 5. Indexed Ps (iPs).

Algorithm 1: getObjSubject$(pred, fam, subj)$

```
1  $pos_f \leftarrow$ select$_1$(iPs$[pred].B_f, fam - 1$);
2  $rnk \leftarrow$ rank$_1$(iPs$[pred].B_o, pos_f$);
3  $pos_s \leftarrow 1+$ select$_1$(iPs$[pred].B_o, subj + rnk - 1$);
4  return iPs.getObjects$(pred, pos_s)$ ;
```

list, encompassing a plain bitsequence and a plain sequence that uses $\log_2(|P|)$ bits per ID.

The previous reasoning also applies for MapS and Families. The transposed of these structures, MapS′ and Families′ respectively, are needed to support subject-based retrieval: MapS′ is used to obtain the ID of the family related to a given subject (the subject is referred by its global ID) and Families′ allows the predicate set of a given family to be efficiently retrieved. MapS′ is implemented as an ID array, as each subject is only related to a single family; i.e. MapS′[i] stores the ID of the family corresponding to the $i - th$ subject. It uses $\log_2(|F|)$ bits per ID. Families′ is implemented as an adjacency list, in which each ID is encoded using $\log_2(|P|)$ bits.

Figure 4 shows the resulting configuration of MapO′, MapS′, and Families′ for the previous example.

Indexing Ps. The Ps structure encodes *Predicate-Family-Object* trees, but the limits of each family (within each predicate) are not explicitly delimited. This information is not needed for decoding purposes because the scan algorithm traverses Ps sequentially [12]. However, family limits must be explicitly encoded to allow random access. An additional bitse-

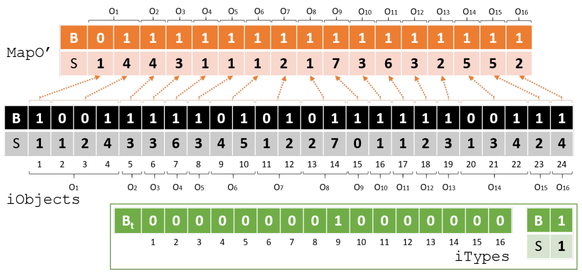

MapO'
B 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
S 1 4 4 3 1 1 1 2 1 7 3 6 3 2 5 5 2
O1 O2 O3 O4 O5 O6 O7 O8 O9 O10 O11 O12 O13 O14 O15 O16
B 1 0 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 0 1 1 1
S 1 1 2 4 3 3 6 3 4 5 1 2 2 7 0 1 1 2 3 1 3 4 2 4
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
iObjects
Bt 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
iTypes
B 1
S 1

Fig. 6. `iObjects` configuration.

quence B_f is added on top of each adjacency list to mark the end of each family within the predicate. The resulting structure is called `iPs`.

`iPs` enhances the `getObject` primitive to retrieve the objects related to a given *(subject,predicate)* pair within a given *family*. Algorithm 1 describes this operation, called `getObjSubject`, and Figure 5 illustrates the `iPs` configuration for our current example. For instance, if we are looking for the objects related to the third subject of the second family of P_1, `getObjSubject`$(1, 2, 3)$ finds that the corresponding list is encoded from $pos_s = 7$ and `getObjects`$(1, 7) = \{5\}$.

The `iObjects` *Index.* This structure enhances `HDT++` for object-based queries, storing the positions in which each object occurrence is encoded in `iPs`. The special value `0` is used to encode that a given object is only associated with predicate `rdf:type`. These objects have a special consideration, as explained below.

`iObjects` is also implemented as an adjacency list, which concatenates object positions according to their global IDs; i.e. positions of O_1 are first encoded, then positions of O_2, and so on. The positions of each object are internally organized in increasing order for each related predicate, and 1-bits mark the last object occurrence for a given predicate. The resulting `iObjects` for our example is illustrated in Figure 6 (we also show `MapO'` for explanation purposes). For instance, O_1 is related to two predicates: P_1 and P_4, as shown in `MapO'`. Thus, `iObjects` encodes two list of occurrences for O_1, one for each predicate: $L_{1,1} = \{1\}$ and $L_{1,4} = \{1, 2, 4\}$. To decode the corresponding triples, the adjacency lists of each predicate must be accessed in `iPs`, retrieving the corresponding positions; e.g. positions `1`, `2`, and `4` of `iPs`[4] encodes the (local) subject IDs of the triples that relate P_4 and O_1.

`iObjects` needs a secondary structure (`iTypes`) to manage the set of objects that are related to the predicate `rdf:type`. Note that, in Figure 6, $S[15] = 0$.

Algorithm 2: `pattern_SPO(subj,pred,obj)`

```
$family \leftarrow$ MapS$'[subj]$;
if $pred < |P|$ then // pred is a regular predicate
    if adj(Families', $family, pred) \neq -1$ then
        $local_o \leftarrow$adj(MapO, $pred, obj$);
        if $local_o \neq -1$ then
            $local_s \leftarrow$adj(MapS, $family, subj$);
            $id_f \leftarrow$adj(Families, $pred, family$);
            $\mathcal{O} \leftarrow$iPs.getObjSubject($pred, id_f, local_s$);
            if bsearch($\mathcal{O}, local_o$) $\neq -1$ then return $true$ ;
            else return $false$ ;
        end
        else return $false$ ;
    end
    else return $false$ ;
end
else // pred is rdf:type
    if adj(Types, $family, obj$) $\neq -1$ then return $true$ ;
    else return $false$ ;
end
```

It means that O_9 is related to `rdf:type`, but the related family is unknown. `iTypes` is composed of a bitsequence (B_t) that marks those objects related to `rdf:type`, and an adjacency list that contains the IDs of the families that are typed with the corresponding object. The corresponding `iObjects` configuration for our example is depicted in Figure 6 (bottom). Note that the bitsequence only sets the bits corresponding to O_9 and the adjacency list has a single element that encodes F_1, because F_1 has the type O_9.

3.2. *Triple Pattern Resolution*

In this section, we explain how `iHDT++` can resolve all SPARQL TPs, except for `(???)`, which corresponds to the scan of the dataset and it is already provided by `HDT++` [12]. Note that we assume that the bounded terms in queries are IDs (in the HDT Dictionary) that identify the corresponding subjects, predicates, or objects.

3.2.1. *Access by Predicate*

The organization of `iHDT++` promotes predicate-based operations, as it encodes *Predicate-Family-Object* trees that can be efficiently traversed. Thus, besides `(???)`, all TPs binding the predicate can exploit the `iHDT++` organization. In the following, we present the algorithms to resolve `(SPO)`, `(SP?)` and `(?P?)`. Even though `(?PO)` could be also resolved, but its performance improves notably by accessing by the value of the object (see Section 3.2.3), as there are generally fewer triples associated to a particular object than to a given predicate [7].

Algorithm 3: `pattern_SP?(subj,pred)`

```
1  $family \leftarrow$ MapS'$[subj]$;
2  if $pred < |P|$ then // pred is a regular predicate
3      if adj(Families', $family, pred) \neq -1$ then
4          $local_s \leftarrow$adj(MapS, $family, subj$);
5          $id_f \leftarrow$adj(Families, $pred, family$);
6          $\mathcal{O} \leftarrow$iPs.getObjSubject($pred, id_f, local_s$);
7          $res \leftarrow \emptyset$;
8          for $i \leftarrow 1$ to $|\mathcal{O}|$ do
9              $res \leftarrow res \cup$ neigh(MapO, $pred$)[$\mathcal{O}[i]$];
10         end
11         return $res$;
12     end
13     else return $false$ ;
14 end
15 else // pred is rdf:type
16     return neigh(Types, family);
17 end
```

`(SPO)` This TP checks the existence of the triple *(subj,pred,obj)* in the RDF dataset, as shown in Algorithm 2. First, the *family* of the subject is retrieved (line 1), and then the predicate is checked (line 2) to determine if it is a regular predicate or it is `rdf:type`. The latter case is easily resolved because the requested triple exists in the dataset only if *family* and *obj* are related in `Types` (line 17). The former case, which involves a regular predicate, requires a multiple check: we verify that *family* includes *pred* (line 3), and then obtain the local ID of *obj* within *pred*; if *pred* and *obj* are not related (i.e. ID = `-1`), the triple does not exist (line 12). The following step maps *subj* to its local ID within its *family* (line 6), and then the position of *family* in *pred* is retrieved (line 7). Line 8 gets the set of objects related to *(subj,pred)* and then *obj* is binary searched in $\mathcal{O}$ (line 9); if $local_o \in \mathcal{O}$, the triple exists in the dataset.

`(SP?)` This TP retrieves all objects associated with the pair *(subj,pred)*, as shown in Algorithm 3. It first obtains the *family* of *subj* and then evaluates *pred*, as in the previous pattern. If the TP asks for `rdf:type`, the requested objects are the direct neighbors of *family* in `Types` (line 16). Looking for the objects associated to a normal predicate also requires checking that *family* includes *pred*, obtaining the local ID of *subj*, the position of *family* in *pred*, retrieving the corresponding objects using `getObjSubject` (line 6) and finally mapping them to their original counterparts (lines 8-10).

`(?P?)` This TP returns all the pairs *(subject,object)* described by *pred*, which was poorly resolved by HDT-FoQ. Algorithm 4 illustrates the resolution with `iHDT++`. For a normal predicate (lines 2-18), the algorithm proceeds as the decompression process [12], but for a concrete predicate. First, the families including *pred* are retrieved (line 4) and iterated (lines 5-17). For each *family*, its related subjects are obtained (line 7) and also iterated (lines 8-16). The objects related to each pair *(subject, pred)* are obtained (line 10) and then mapped to their global IDs (lines 12-15), as in the previous algorithms. The process for `rdf:type` also requires a nested loop algorithm. In this case, the algorithm iterates over all families and, for each one, it retrieves its type values (line 21). If $\mathcal{O}$ is not empty (line 22), the family is typed and its related subjects are retrieved from `MapS`. Finally, we iterate over $\mathcal{S}$ and $\mathcal{O}$ to return all the pair combinations from each set.

Algorithm 4: `pattern_?P?(pred)`

```
1  $res \leftarrow \emptyset$;
2  if $pred < |P|$ then // pred is a regular predicate
3      $ptrSubj \leftarrow 1$;
4      $\mathcal{F} \leftarrow$ neigh(Families, $pred$);
5      for $i \leftarrow 1$ to $|\mathcal{F}|$ do
6          $family \leftarrow \mathcal{F}[i]$;
7          $\mathcal{S} \leftarrow$ neigh(MapS, $family$);
8          for $j \leftarrow 1$ to $|\mathcal{S}|$ do
9              $subject \leftarrow \mathcal{S}[j]$;
10             $\mathcal{O} \leftarrow$ iPs.getObjects($predicate, ptrSubject$);
11             $ptrSubj \leftarrow ptrSubj + 1$;
12             for $k \leftarrow 1$ to $|\mathcal{O}|$ do
13                 $object \leftarrow$ neigh(MapO, $pred$)[$\mathcal{O}[k]$];
14                 $res \leftarrow res \cup (subject, object)$;
15             end
16         end
17     end
18 end
19 else // pred is rdf:type
20     for $i \leftarrow 1$ to $|F|$ do
21         $\mathcal{O} \leftarrow$ neigh(Types, $i$) ;
22         if $\mathcal{O} \neq \emptyset$ then
23             $\mathcal{S} \leftarrow$ neigh(MapS, $i$) ;
24             for $i \leftarrow 1$ to $|\mathcal{S}|$ do
25                 for $j \leftarrow 1$ to $|\mathcal{O}|$ do
26                     $res \leftarrow res \cup (\mathcal{S}[i], \mathcal{O}[j])$;
27                 end
28             end
29         end
30     end
31 end
32 return res;
```

3.2.2. *Access by Subject*

As opposed to the original HDT, `iHDT++` resolves only a single TP accessing by subject: `(S??)`.

`(S??)` This TP looks for all pairs *(predicate, object)* describing a given subject (*subj*). As shown in Algorithm 5, *subj* is used to retrieve its related *family*, which is then used to obtain the corresponding predicates (lines (2-3). The set of predicates is then iterated to retrieve all objects related to *subj* and each predicate. It is easily resolved by calling `pattern_SP?` (line 5), and the returned objects are appended to the result set (lines 6-8). Finally, we check whether the *family*

Algorithm 5: `pattern_S??(subj)`

```
1  $res \leftarrow \emptyset$;
2  $family \leftarrow$ MapS'$[subj]$;
3  $\mathcal{P} \leftarrow$ neigh(Families'$, family$);
4  for $i \leftarrow 1$ to $|\mathcal{P}|$ do
5      $\mathcal{O} \leftarrow$ pattern_SP?$(subj, P[i])$;
6      for $j \leftarrow 1$ to $|\mathcal{O}|$ do
7          $res \leftarrow res \cup (P[i], O[j])$;
8      end
9  end
10 $\mathcal{O} \leftarrow$ neigh(Types$, family$);
11 if $\mathcal{O} \neq \emptyset$ then
12     for $i \leftarrow 1$ to $|\mathcal{O}|$ do
13         $res \leftarrow res \cup (|P|, O[j])$;
14     end
15 end
16 return $res$;
```

Algorithm 6: `pattern_S?O(subj,obj)`

```
1  $res \leftarrow \emptyset$;
2  $\mathcal{P} \leftarrow$ neigh(MapO'$, obj$);
3  for $i \leftarrow 1$ to $|\mathcal{P}|$ do
4      if pattern_SPO$(subj, \mathcal{P}[i], obj)$ then
5          $res \leftarrow res \cup P[i]$;
6      end
7  end
8  return $res$;
```

is typed, to add the corresponding pairs (`rdf:type`, $value$) to the result set. In line 10, the possible type values of the *family* are retrieved from `Types`; if there exist, they are added to the final result set (note that the ID $|P|$, in line 13, refers to the predicate `rdf:type`).

3.2.3. Access by Object

`iHDT++` provides efficient object-based search via `MapO'` and `iObjects`, resolving the TPs `(S?O)`, `(??O)`, and `(?PO)`.

`(S?O)` This TP retrieves all *predicates* that label the pair *(subj,obj)*, illustrated in Algorithm 6. It uses `MapO'` to get the predicates related to *obj* (line 2), and then invokes `pattern_SPO` to check the combinations $(subj, \mathcal{P}[i], obj$ (line 3), for each retrieved predicate $P[i]$. If the triple exists, $P[i]$ is added to the result set.

`(?PO)` This TP retrieves all *subjects* characterized by the pair *(pred,obj)*. It distinguishes between normal predicates and `rdf:type`. The process for normal predicates first checks if *obj* is related to $pred$ (lines 3-4), and then retrieves the position in which these occurrences are encoded in `iObjects` (lines 5-6). For each occurrence in $\mathcal{O}ccs$, we navigate the adjacency list of $pred$ in `iPs` to finally decode the corresponding subject, which is mapped to its original ID (line 11). If $pred$ is `rdf:type`, we also check if obj is related to such predicate. In this case, we retrieve the families

Algorithm 7: `pattern_?PO(pred,obj)`

```
1  $res \leftarrow \emptyset$;
2  if $pred < |P|$ then // pred is a regular predicate
3      $pos_p \leftarrow$adj(MapO'$, obj, pred)$ ;
4      if $pos_p \neq -1$ then
5          $pos \leftarrow pos_p$+select$_1$(MapO'$.B, obj - 1)$;
6          $\mathcal{O}ccs \leftarrow$ neigh(iObjects,$pos$);
7          for $i \leftarrow 1$ to $|\mathcal{O}ccs|$ do
8              $id_f \leftarrow 1$+rank$_1$(iPs$[pred].B_f, \mathcal{O}ccs[i] - 1)$;
9              $family \leftarrow$ neigh(Families$, pred)[id_f]$;
10             $local_s \leftarrow \mathcal{O}ccs[i] -$
                 select$_1$(iPs$[pred].B_f, id_f - 1)$;
11             $res \leftarrow res \cup$ neigh(MapS$, family)[local_s]$;
12         end
13     end
14 end
15 else // pred is rdf:type
16     if access$_1$(iTypes$.B_t, obj) = 1$ then
17         $object \leftarrow$ rank$_1$(iTypes$.B_t, obj)$ ;
18         $\mathcal{F} \leftarrow$ neigh(iTypes$, object)$ ;
19         for $i \leftarrow 1$ to $|\mathcal{F}|$ do
20             $\mathcal{S} \leftarrow$ neigh(MapS$, \mathcal{F}[i])$ ;
21             for $j \leftarrow 1$ to $|\mathcal{S}|$ do
22                 $res \leftarrow res \cup \mathcal{S}[j]$;
23             end
24         end
25     end
26 end
27 return res;
```

Algorithm 8: `pattern_??O(obj)`

```
1  $res \leftarrow \emptyset$;
2  $\mathcal{P} \leftarrow$ neigh(MapO'$, obj$);
3  for $i \leftarrow 1$ to $|\mathcal{P}|$ do
4      $\mathcal{S} \leftarrow$ pattern_?PO$(\mathcal{P}[i], obj)$;
5      for $j \leftarrow 1$ to $|\mathcal{S}|$ do
6          $res \leftarrow res \cup (\mathcal{S}[j], \mathcal{P}[i])$;
7      end
8  end
```

typed by obj from `iTypes` (line 18). For each family, we obtain its corresponding *subjects*, which are added to the final result set.

`(??O)` This TP retrieves all the *(subject,predicate)* pairs described with the given *obj* value. The resolution is illustrated in Algorithm 8. It uses `MapO'` to retrieve all predicates $\mathcal{P}[i]$ related to obj. Then the `pattern_?PO` is invoked for each one, and the returned *subjects*, and the corresponding $\mathcal{P}[i]$, are added to the result set.

4. Evaluation

This section presents a comprehensive evaluation that compares `iHDT++` to its predecessors, HDT-FoQ [15] and its *Community* variant. Our goal is to show that `iHDT++` can replace the existing HDT-based deployments by a more lightweight approach, without losing the current HDT performance. We also compare `iHDT++` to k^2-triples [1] and RDFCSA [4], to show that it competes with the state-of-the-art RDF self-indexes, keeping the standardized features of HDT.

4.1. Experimental Setup

The `iHDT++` prototype[4] is coded in C++ 11 and uses the SDSL library[5] to implement all compact data structures. HDT-FoQ and HDT Community prototypes are publicly available[6] and the C-based k^2-triples and RDFCSA have been kindly provided by their authors. All experiments in this study were run on an Intel Xeon CPU E5-2470 0 @ 2.30GHz, 8 cores/16 siblings, 64GB RAM, Debian GNU/Linux 9.8 (stretch).

Datasets. Table 1 shows the main features of 4 real-world datasets used in this evaluation: DBLP (scientific publications), DBTUNE (music data), USCENSUS (census data from U.S.) and LINKEDGEODATA (geographic data from *OpenStreetMap*). The selected datasets differ in size, topic and level of structure.

We only show figures for representative USCENSUS and LINKEDGEODATA due to lack of space, but all conclusions drawn from them apply to the other datasets. On the one hand, USCENSUS provides highly-structured contents, as shown by its low number of predicate families, 106, which is even less than the number of different predicates, 429. Note that USCENSUS does not use the `rdf:type` predicate. On the other hand, LINKEDGEODATA is an unstructured dataset that uses a high number of predicates, $18,272$, including `rdf:type`. In this case, $1,081$ different classes are related to `rdf:type`, which are used to type $440,035$ families. In addition, LINKEDGEODATA has almost 2,000 non-typed families.

Experiments. Our experiments evaluate the space complexity and query performance of all SPARQL TPs over the aforementioned datasets. In all cases, we have randomly chosen 1,000 different query patterns[7] that return, at least, one result. The performance time is averaged over five independent runs. In turn, we report compression ratios for each dataset and technique: we calculate these numbers as the amount of memory used by each technique with respect to the original size of the dataset (expressed in terms of triple-IDs).

[4]`https://github.com/antonioillera/iHDTpp-src`
[5]`https://github.com/simongog/sdsl-lite`
[6]`https://github.com/rdfhdt/hdt-cpp`
[7] `(?P?)` is limited by the number of predicates in the dataset.

Figures 7 and 8 show the corresponding space-time tradeoffs for each dataset and TP. Each graph reports query times, in $\mu s/pattern$, in Y-axis (logarithmic scale) and compression ratios in X-axis. Note that multiple space-time tradeoffs are reported for `iHDT++`, k^2-triples, and RDFCSA, as follows. In our case, `MapS` and `MapO` are configured with different sampling values $t_{dens} = 2^i, 1 \leq i \leq 7$, (better performance is reported for low t_{dens} values, at the cost of less compressed representations). K^2-triples has a plane configuration that can be enhanced with two additional indexes to speed up some TPs. Finally, RDFCSA is tuned with ψ sampling values $t_\psi = \{16, 32, 64, 256\}$.

4.2. Analysis of the Results

This section analyzes the space-time tradeoffs in Figures 7-8, comparing `iHDT++` with the HDT-based predecessors and the most efficient RDF self-indexes.

Compression. `iHDT++` outperforms HDT-based solutions. Its memory footprint is between $\approx 50\%$ and $\approx 60\%$ of the original size of USCENSUS and $52\% - 70\%$ of LINKEDGEODATA, while HDT-FoQ uses 81% and 91%, respectively, and HDT Community more than 100% in both cases. These numbers endorse the RDF-TR transformation [12], underlying to `iHDT++`, but also the lower cost of its additional structures compared to HDT-FoQ and HDT Community ones.

In turn, RDFCSA and `iHDT++` report roughly the same numbers for both datasets, but the comparison with k^2-triples demonstrates that more optimized self-indexes are clearly superior in space. The plain configuration of k^2-triples has a memory footprint of 20% of the original space for both datasets, while enhancing it with additional indexes just increases to 25% for USCENSUS and 30% for LINKEDGEODATA. It is an expected result as k^2-triples is highly-optimized for compression.

Query Performance. The first line of Figures 7 and 8 shows plots for TPs using predicate-based access in `iHDT++`, i.e. `(???)`, `(SPO)`, `(SP?)`, and `(?P?)`. `iHDT++` is always faster than HDT-FoQ and HDT Community. The difference is particularly significant in `(???)` and `(?P?)`, in which `iHDT++` outperforms its predecessors by an order of magnitude in USCENSUS. The difference decreases in `(???)` for LINKEDGEODATA, but for `(?P?)` `iHDT++`is almost two orders of magnitude faster than HDT-FoQ. For `(SPO)` and `(SP?)`, `iHDT++` is also faster, although the difference is less than $1\mu s$ per pattern in each case. The com-

Table 1

Dataset features.

Dataset	Triples	Subjects	Predicates	Objects	Types	Families	Typed families	Triples Size (MB)
DBLP	55,586,971	3,591,091	27	25,154,979	14	283	283	636.14
DBTune	58,920,361	12,401,228	394	14,264,221	64	1,047	866	647.29
USCensus	149,182,415	23,904,658	429	23,996,813	0	106	0	1,707.19
LinkedGeoData	271,180,352	51,916,995	18,272	121,749,861	1,081	441,922	440,035	3,103.41

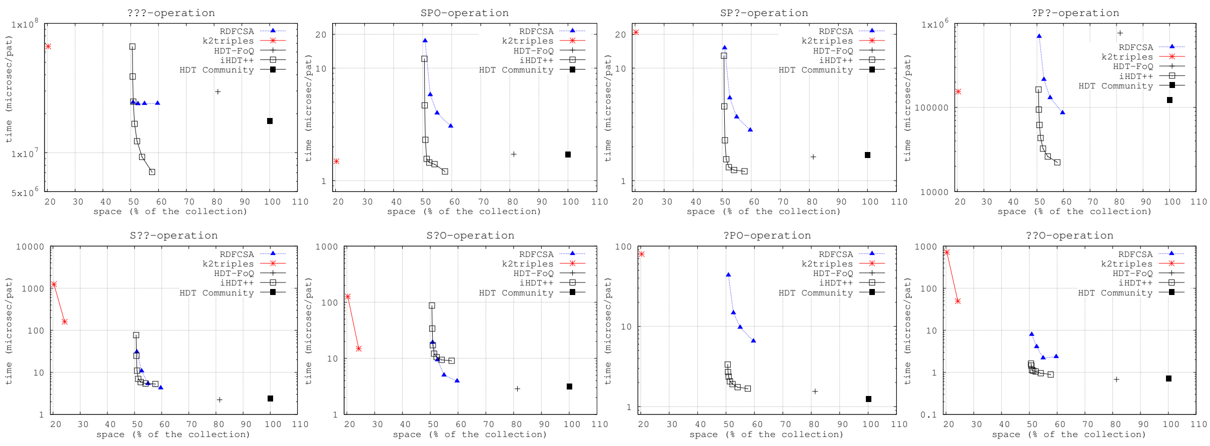


Fig. 7. TPs Resolution: USCensus

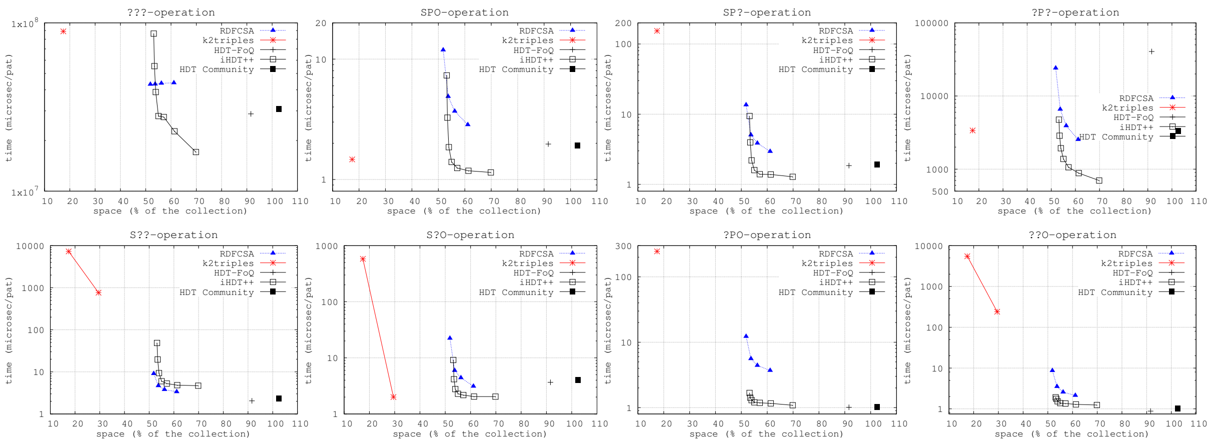


Fig. 8. TPs Resolution: LinkedGeoData

parison to self-indexes also shows that iHDT++ is the fastest choice. RDFCSA performs in the same order of magnitude than iHDT++, but it is always slower. Regarding k^2-triples, it only competes in (SPO), being one order of magnitude slower for the remaining TPs.

The left-most plots in the second line show the tradeoffs for (S??), the only TP that is resolved by subject. HDT-FoQ and HDT Community report the best time as both leverage their subject-based organizations, but the difference with iHDT++ is not significant. It needs $\approx 2-3$ more μs per pattern, reporting similar numbers than RDFCSA. K^2-triples performs 2 orders of magnitude slower than the rest.

Finally, we analyze the TPs in which iHDT++ accesses by object: (S?O), (?PO) and (??O). These are the less-favoured queries in iHDT++, but their performance remain competitive. HDT variants are slightly faster in USCensus, but the difference decreases in LinkedGeoData, where iHDT++ is the fastest choice in (S?O). On the other hand, iHDT++ outperforms RDFCSA with roughly the same memory footprint, while k^2-triples only competes in (S?O), being 2 orders of magnitude slower for other TPs.

5. Conclusion

Scalable HDT-based technologies have emerged as the de-facto standard to manage large RDF compressed data in the Web of Data. These systems exploit the compact data structures of HDT to resolve SPARQL TPs with an affordable memory footprint. Despite their success, all these systems are limited by the simplicity of the HDT encoding, which causes space overheads and lack of scalability for some predicate-based TPs. In this paper, we enhance the existing HDT++ compressor (a variant that leverages structural redundancies) with additional compact indexes to support full SPARQL TP resolution. Our experiments show that iHDT++ halves the memory footprint of HDT Community, the most extended variant of HDT, while it improves the resolution of the less efficient predicate-based TP by one order of magnitude. In addition, iHDT++ speeds up the majority of TPs. Our experiments also report better space/time tradeoffs than the most competitive RDF self-indexes in the state of the art, k^2-triples and RDFCSA.

These results show that iHDT++ can replace current HDT-backends, improving the performance of the tools relying on HDT-based technology for publication and consumption. Our current efforts focus on providing the integration toolset for this purpose.

Acknowledgements

This paper is funded by MINECO-AEI/FEDER-UE: TIN2016-78011-C4-1-R; by EU H2020: 731601 (SPECIAL) and 690941 (BIRDS); by FFG: 861213 (CitySPIN); by Xunta de Galicia/FEDER-UE [CSI: ED431G/01 and GRC: ED431C 2017/58]; by Xunta de Galicia Conecta-Peme 2018 [Gema: IN852A 2018/14]; by MCIU-AEI/ FEDER-UE [ETOME-RDFD3: TIN2015-69951-R; BIZDEVOPS: RTI2018-098309-B-C32]

References

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

[2] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[3] P. A. Bonatti, M. Cochez, S. Decker, A. Polleres, and V. Presutti. Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web (Dagstuhl Seminar 18371). *Dagstuhl Reports*, 8(9):29–111, 2019.

[4] N. Brisaboa, A. Cerdeira-Pena, A. Farina, and G. Navarro. A Compact RDF Store Using Suffix Arrays. In *Proc. of SPIRE*, pages 103–115, 2015.

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

[6] D. Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, 1997.

[7] 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/.

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

[9] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[10] R. Grossi, A. Gupta, and J.S. Vitter. High-order Entropy-compressed Text Indexes. In *Proc. of SODA*, pages 841–850, 2003.

[11] A. Hernández-Illera, M.A. Martínez-Prieto, and J.D. Fernández. Serializing RDF in Compressed Space. In *Proc. of DCC*, pages 363–372, 2015.

[12] A. Hernández-Illera, M.A. Martínez-Prieto, and J.D. Fernández. RDF-TR: Exploiting Structural Redundancies to boost RDF Compression. *Inform Sciences*, 508:234–259, 2020.

[13] C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[14] M. A. Martínez-Prieto, J. D. Fernández, A. Hernández-Illera, and C. Gutiérrez. RDF Compression. In *Encyclopedia of Big Data Technologies*, pages 1–11. Springer, 2018.

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

[16] M.A. Martínez-Prieto, J.D. Fernández, and R. Cánovas. Compression of RDF Dictionaries. In *Proc. of SAC*, pages 1841–1848, 2012.

[17] T. Minier, H. Skaf-Molli, and P. Molli. SaGe: Web Preemption for Public SPARQL Query Services. In *Proc. of The Web Conference*, 2019.

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

[19] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput Surv*, 39(1):article 2, 2007.

[20] J.Z. Pan, J.M Gómez-Pérez, Y. Ren, H. Wu, W. Haofen, and M. Zhu. Graph Pattern Based RDF Data Compression. In *Proc. of JIST*, pages 239–256, 2015.

[21] E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation, 2008.

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

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

[24] G. Schreiber and Y. Raimond. *RDF 1.1 Primer*. W3C Recommendation, 2014.

[25] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, and E. Mannens. Querying Datasets on the Web with High Availability. In *Proc. of ISWC*, pages 180–196, 2014.

Chapter 6

Conclusions and Future Work

6.1. Conclusions

The Web has changed in recent years: going from being an entity for human consumption at its dawn, to beginning to be a huge container of semi-structured data described in the same language (i.e., RDF), freely accessible (i.e., Open Data) and connected to each other (i.e., Linked Open Data) to be (re)used, exchanged, searched (i.e., SPARQL) and consumed by machines.

RDF compressors have become, in a short time, an important line of research in the field of the Web of Data thanks to their ability to reduce the size of large data collections and their performance for the resolution of SPARQL Triple Patterns. However, the redundancies that the schema-relaxed nature of RDF causes have not been taken into account, or have not been particularly relevant. In this thesis, we have demonstrated the hypothesis formulated in Section 1.2, addressing two types of structural-based redundancies: predicate families are massively repeated for general and typed subjects, and objects are often related to just one predicate. To prove the hypothesis, we reorganize the RDF graph configuration to reduce these redundancies underlying the RDF datasets, thus improving the compressibility of its graph structure encoding.

Our proposal, RDF-Tr, reorganizes the triples to leverage these features and generalizes the reorganization process in such a way that it can be plugged into different symbolic compressors (those that use integer IDs to encode RDF terms), such as HDT and k^2-triples, two of the main and most commonly used RDF compressors of the state of the art. Experiments show that it is possible to save up to 50% of the dataset size when RDF-Tr is applied to HDT (thus leading to a novel proposal named HDT++), and 56% in the case of k^2-triples (i.e., k^2-triples++), the most effective RDF compressor.

Regarding decompression, the original compressor times are similar to their counterparts. Additionally, the final configuration can be tuned to explore different space/time tradeoffs.

The RDF-Tr organization can be exploited to additionally provide fast SPARQL Triple Pattern retrieval on compressed space. In this thesis we have enhanced HDT++ with additional compact indexes (i.e., iHDT++) to support full SPARQL Triple Pattern resolution. It is experimentally shown that iHDT++ halves the memory footprint of HDT-FoQ, while also improving the resolution of the less efficient predicate-based Triple Patterns by one order of magnitude. In addition, iHDT++ speeds up the majority of Triple Patterns. Our experiments also report better space/time tradeoffs than the most competitive RDF self-indexes in the state of the art, k^2-triples and RDFCSA. These results show that iHDT++ can replace current HDT-backends, improving the performance of the tools relying on HDT-based technology for publication and consumption. On the other hand, we have experimentally verified that the effectiveness of our technique (i.e., compression ratio) is penalized when facing highly unstructured datasets, maintaining its efficiency in data access. This may open the door to new lines of research to mitigate the consequences of the lack of structure in some data collections.

6.2. Future Work

As we have demonstrated in this thesis, processes carried out in the RDF graph, such as the reorganization of triples and the recoding of identifiers are highly effective, improving the compression of some RDF graph techniques. These redundancies, tackled by RDF-Tr, were detected heuristically and new patterns could be discovered to get a better compression (e.g., looking at the structure of adjacent nodes in the RDF graph [31]). In this context, queries can take advantage of family organization and optimize more complex queries, for example, joins per subject. In addition, self-indexes built on the top of HDT++ (i.e., iHDT++) could be adapted for use in k^2-triples++ and thus, providing SPARQL Triple Pattern resolution (i.e., ik^2-triples++). Although the cost of applying RDF-Tr is acceptable nowadays, the construction of the necessary structures and self-indexes could be parallelized in order to optimize the construction of RDF-Tr, iHDT++ and ik^2-triples++. Different (compact) data structures can be explored to build the indexes, so the size and/or speed of information retrieval can be optimized. Furthermore, family grouping could help to decrease the versioning update time on datasets, dealing only with families involved in data changes. Besides, RDF compressors can be adapted to make use of explicitly declared constraints or regularities in data (e.g., expressed with SHACL [30] or ShEx [4]).

One of the main characteristics of Big Semantic Data is the speed of growth of the number of RDF datasets and the interconnections between them (as seen in Figure 1.1). Although some large semantic data collections remain static (in particular those following a one-off conversion from other sources), some large datasets (e.g., *WikiData*) are frequently updated or even generated in real time (e.g., in the area of RDF streaming). In general, the RDF compression process has a non-negligible creation/recreation cost that may hamper dynamic updates. Given this fact, a challenge to be faced is the inspection of time/space tradeoffs to boost fast and efficient compression, and the adaptation of RDF data stores to host compressed and uncompressed data that would allow for both types of data and therefore the update of datasets in real time [36].

Bibliography

[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, R. Cyganiak, and Z. Ives. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web*, pages 722–735. Springer Berlin Heidelberg, 2007.

[4] T. Baker and E. Prud'hommeaux. Shape Expressions (ShEx) Primer. *Draft Community Group Report 14 July 2017*, 2017.

[5] W. Beek, L. Rietveld, H.R. Bazoobandi, J. Wielemaker, and S. Schlobach. LOD Laundromat: A Uniform Way of Publishing Other People's Dirty Data. In *The Semantic Web – ISWC 2014*, pages 213–228. Springer International Publishing, 2014.

[6] T. Berners-Lee. Linked Data. URL `https://www.w3.org/DesignIssues/LinkedData.html`, 2006.

[7] T. Berners-Lee, M. Fischetti, and M.L. Dertouzos. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor.* 1st edition, 1999.

[8] K. Bok, J. Han, J. Lim, and J. Yoo. Provenance compression scheme based on graph patterns for large RDF documents. *The Journal of Supercomputing*, pages 1–23, 2019.

[9] D. Brickley and Guha R.V. *RDF Schema 1.1.* W3C Recommendation, 2014. `https://www.w3.org/TR/r2rml/`.

[10] N. Brisaboa, A. Cerdeira-Pena, A. Farina, and G. Navarro. A Compact RDF Store Using Suffix Arrays. In *Proc. of SPIRE*, pages 103–115, 2015.

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

[12] N.R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, and A. Fariña. Revisiting compact RDF stores based on k2-trees. In *Proceedings of the Data Compression Conference (DCC 2020)*, pages 123–132, 2020.

[13] G. Carothers and A. Seabourne. *RDF 1.1 N-Triples: A line-based syntax for an RDF graph.* W3C Recommendation, 2014. `https://www.w3.org/TR/n-triples/`.

[14] D. Connolly. *Gleaning Resource Descriptions from Dialects of Languages (GRDDL).* W3C Recommendation, 2007. `https://www.w3.org/TR/grddl/`.

[15] D. Connolly. *R2RML: RDB to RDF Mapping Language.* W3C Recommendation, 2012. `https://www.w3.org/TR/r2rml/`.

[16] 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.

[17] J.D. Fernández, W. Beek, M.A. Martínez-Prieto, and M. Arias. LOD-a-lot: A Queryable Dump of the LOD Cloud. In *The Semantic Web – ISWC 2017*, pages 75–83. Springer International Publishing, 2017.

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

[19] 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.

[20] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, page 841–850. Society for Industrial and Applied Mathematics, 2003.

[21] C. Gutiérrez, C. Hurtado, A.O. Mendelzon, and J. Pérez. Foundations of Semantic Web Databases. *Journal of Computer and System Sciences*, 77:520–541, 2011.

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

[23] A. Hernández-Illera, M.A. Martínez-Prieto, and J.D. Fernández. iHDT++: un Autoíndice Semántico para la Resolución de Patrones de Consulta SPARQL. In *Proceedings of Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2017.

[24] A. Hernández-Illera, M.A. Martínez-Prieto, and J.D. Fernández. RDF-Tr: Exploiting Structural Redundancies to boost RDF Compression. *Information Sciences*, 508:234–259, 2020.

[25] A. Hernández-Illera, M.A. Martínez-Prieto, J.D. Fernández, and A. Fariña. iHDT++: Improving HDT for SPARQL Triple Pattern Resolution. In *Proceedings of the 7th Int Symp On Language and Knowledge Engineering (LKE)*, 2019.

[26] A. Hernández-Illera, M.A. Martínez-Prieto, J.D. Fernández, and A. Fariña. iHDT++: Improving HDT for SPARQL Triple Pattern Resolution. *Journal of Intelligent & Fuzzy Systems*, 2020. http://doi.org/10.3233/JIFS-179888.

[27] 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.

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

[29] F. Karim, M.E. Vidal, and S. Auer. Compacting Frequent Star Patterns in RDF Graphs. *ArXiv*, abs/2003.05238, 2020.

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

[31] 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.

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

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

[34] M.A. Martínez-Prieto, J.D. Fernández, A. Hernández-Illera, and C. Gutiérrez. RDF Compression. In *Encyclopedia of Big Data Technologies.* Springer International Publishing, 2018.

[35] M.A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical compressed string dictionaries. *Information Systems*, 56:73 – 108, 2016.

[36] M.A. Martínez-Prieto, C.E. Cuesta, M. Arias, and J.D. Fernández. The solid architecture for real-time management of big semantic data. *Future Generation Computer Systems*, 47, 10 2014.

[37] M.A. Martínez-Prieto, J.D. Fernández, and R. Cánovas. Compression of RDF Dictionaries. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 340–347. Association for Computing Machinery, 2012.

[38] J.P. McCrae, A. Abele, P. Buitelaar, R. Cyganiak, A. Jentzsch, V. Andryushechkin, and J. Debattista. The Linked Open Data Cloud. URL `https://lod-cloud.net/`, 2019.

[39] 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.

[40] 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.

[41] 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.

[42] H. Pascal, M. Krötzsch, B. Parsia, Patel-Schneider P.F., and Rudolph S. *OWL 2 Web Ontology Language Primer*. W3C Recommendation, 2012. https://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[43] S. Pemberton. *XHTML 1.0 The Extensible HyperText Markup Language*. W3C Recommendation, 2000. https://www.w3.org/TR/xhtml1.

[44] G.E. Pibiri, R. Perego, and R. Venturini. Compressed Indexes for Fast Search of Semantic Data. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[45] A. Polleres, A. Hogan, R. Delbru, and J. Umbrich. *RDFS and OWL Reasoning for Linked Data*, pages 91–149. Springer Berlin Heidelberg, 2013.

[46] E. Prud'hommeaux and G. Carothers. *RDF 1.1 Turtle: Terse RDF Triple Language*. W3C Recommendation, 2014. https://www.w3.org/TR/turtle/.

[47] E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation, 2008. https://www.w3.org/TR/json-ld/.

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

[49] Sherif Sakr and Albert Zomaya. *Encyclopedia of big data technologies*. Springer Publishing Company, Incorporated, 2019.

[50] Kellogg G. Sporny, M. and Lanthaler M. *JSON-LD 1.0: A JSON-based Serialization for Linked Data*. W3C Recommendation, 2014. https://www.w3.org/TR/json-ld/.

[51] 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.

[52] 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.

[53] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37-38:184 – 206, 2016.

Appendix A

Using RDF-Tr and iHDT++

This appendix presents the iHDT++ library, which contains the practical implementation of the theoretical work of the research developed over the last few years and now compiled in this thesis. The library is free software under the terms of the GNU Lesser General Public License, and it is available at GitHub[1]. This has a special relevance for the scientific community, since it allows the results of the experiments contained in the papers that are part of the thesis to be reproduced.

This project makes use of two already existing libraries: HDT and SDSL. On the one hand, HDT is the first and most used binary representation of RDF, so it was taken as a starting point to apply our reorganization and self-indexing processes, preserving and reusing the dictionary that HDT implements. HDT serializes the triple IDs succinctly using the *Compressed Data Structure Library* (libcds)[2], whose development has been discontinued despite its effectiveness in compression. That is the main reason why we use the *Succinct Data Structure Library* (SDSL)[3], which must be installed to use the compact data structures (and their operations) provided and used by iHDT++. The source code is written in C++ and is available online in a public repository, along with the latest available HDT version. The main folders of the project are: `libsdsl`, which contains the installation of the SDSL library; `hdtpp` contains the core of our work, which includes the data structures and methods necessary to reorganize triples and access information in compressed space; finally the `tools` folder provides simple utilities to perform these operations. Below there is a representation of the project folder tree where the mentioned directories are located.

[1]https://github.com/antonioillera/iHDTpp-src

[2]https://github.com/fclaude/libcds2

[3]https://github.com/simongog/sdsl-lite

```
iHDTpp-src
├── libsdsl
└── libhdt
    ├── src
    │   └── hdtpp
    └── tools
```

By means of an example, we can see the necessary steps to create a representation in HDT++ from a data collection (e.g., *dblp*) serialized in HDT to later access its data. Given an HDT dataset, `hdt2hdtpp` applies RDF-Tr and recompresses the HDT file into HDT++ (see Code 1).

```
$ iHDTpp—src/libhdt/tools/hdt2hdtpp dblp.hdt dblp.hdtpp
```

Bash Code 1: Applying RDF-Tr on an HDT file.

iHDT++ indexes allow data in a compressed HDT++ file to be accessed. The `hdtppSearch` utility implements a simple interface to access the dataset by SPARQL Triple Patterns. A simple use of this tool is shown in Code 2, where we ask for all triples. Indexes are created/loaded in execution time.

```
$ iHDTpp—src/libhdt/tools/hdtppSearch dblp.hdtpp "?␣?␣?"
```

Bash Code 2: Applying RDF-Tr on an HDT file.

On the other hand, `hdtpp2rdf` decompresses the HDT++ file, obtaining the RDF version of the dataset (see Code 3).

```
$ iHDTpp—src/libhdt/tools/hdtpp2rdf dblp.hdtpp dblp.rdf
```

Bash Code 3: Decompressing an HDT++ file.