Compact Encoding of Reified Triples Using HDTr

Jose M. Gimenez-Garcia[1,2](✉), Thomas Gautrais[2], Javier D. Fernández[3], and Miguel A. Martínez-Prieto[4]

[1] Group of Intelligent and Cooperative Systems, Univ. of Valladolid, Valladolid, Spain
jm.gimenez.garcia@gsic.uva.es

[2] Univ. de Lyon, CNRS, UMR 5516, Lab. Hubert-Curien, Saint-Étienne, France
thomas.gautrais@univ-st-etienne.fr

[3] Data Science Acceleration (DSX), F. Hoffman-La Roche, Basel, Switzerland
javier_d.fernandez@roche.com

[4] Department of Computer Science, Univ. of Valladolid, Valladolid, Spain
miguelamp@uva.es

Abstract. Contextual information about a statement is usually represented in RDF knowledge graphs via *reification:* creating a fresh 'anchor' term that represents the statement and using it in the triples that describe it. Current approaches make the connection between the reified statement and its anchor by either extending the RDF syntax, resulting in non-compliant RDF, or via additional triples to connect the anchor with the terms of the statement, at the cost of size and complexity.

This work tackles this challenge and presents HDTr, a binary serialization format for reified triples that is model-agnostic, compact, and queryable. HDTr is based on, and compatible with, the counterpart HDT format, leveraging its underlying structure to connect the reified statements with the terms that represent them. Our evaluation shows that HDTr improves compression and retrieval time of reified statements *w.r.t.* several triplestores and HDT serialization of different reification approaches.

Keywords: Knowledge Graphs · RDF · Reification · Contextual Data · HDT · Compression · Compact Data Structures

1 Introduction

Knowledge Graphs (KG) are increasingly being used to build innovative data-driven applications (semantic search, question answering, product recommendation, etc.) thanks to their ability to store and communicate the semantics of real-world knowledge. KGs represent information as a graph of entities and relationships, typically modeled as a directed labelled graph due to its relative succinctness [22]. The *Resource Data Framework* (RDF) [7] has been used extensively to model such KGs in terms of `<subject>, <predicate>, <object>` *statements*, also called *triples*. Thus, each statement establishes the relationship (labeled in the predicate) between the two resources declared in the subject and object values. However, it does not natively allow information (such as validity time, provenance, certainty...) about the triple itself to be expressed [28].

The most common solution to express this information is to *reify* the statement; *i.e.*, to introduce a new element that represents the statement, which we call the *anchor*. Then, this anchor is used as subject and/or object in a set of triples that describe the statement, which we refer to as the *contextual annotation*. Existing approaches to represent reified statements fall in two categories. The first approach can be seen as applying a *reification function* to the triple and its contextual annotation, to obtain a new RDF graph where the original statement is replaced by a set of triples connecting the anchor to both the terms of the statement and the contextual annotation (*e.g.*, RDF reification [4]). However, the introduction of different sets of triples per reified statement can degrade performance by increasing the size and complexity of the graph, and introduce interoperability problems between the results of two or more contextualization functions. A second alternative is to extend the syntax (and possibly the semantics) of RDF to cover the reification needs, at the cost of obtaining non-compliant RDF graphs (*e.g.*, RDF-Star [17]).

The management of reified statements is therefore an emerging challenge with increasingly large KGs, which account billions of triples that are subject to reification; e.g. Wikidata contains more than 100 million different subjects and more than 1.2 billion statements, from which at least 900 million are reified.[1] A simple but effective approach to deal with this complexity in regular KGs (without reification) is *knowledge graph compression*, *i.e.*, *"encoding a KG using less bits than its original representation"* [25]. Managing compressed KGs leads to more efficient storage and lower transmission costs due to space savings. In addition, some compressors, such as HDT *(Header-Dictionary-Triples)* [11], allow efficient access to triples without prior decompression due to their self-indexed internal structures. Unfortunately, none of these solutions natively support reification. A practical approach is to compress the graph produced by a reification function at the cost of encoding potentially much larger datasets, with the consequent impact that this has on space requirements and, in particular, the retrieval time of a reified statement and its anchor.

This paper presents HDTr, an extension of HDT for compressing and querying reified triples. This approach enhances HDT components to link statements to their anchors, without adding new triples nor adapting to a new syntax. The resulting HDTr compressed files comply with the HDT *standard*, allowing HDTr to be used in applications that consume HDT (see Sect. 4 for some examples). Our evaluation comparing HDTr with an HDT-based deployment, and with triplestores using named graphs to encode contextual annotations, shows that HDTr outperforms both approaches in terms of space and performance.

The rest of the paper is organized as follows. Sections 2 and 3 provide a basic background on reification and review practical solutions to manage reified KGs. Section 4 explains the foundations of HDT, which is extended into HDTr

[1] https://wikidata-todo.toolforge.org/stats.php.

in Sect. 5. A comprehensive evaluation is provided in Sect. 6, and we conclude and provide future work in Sect. 7.

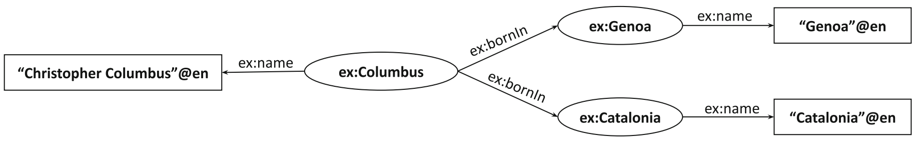


Fig. 1. Example of RDF graph

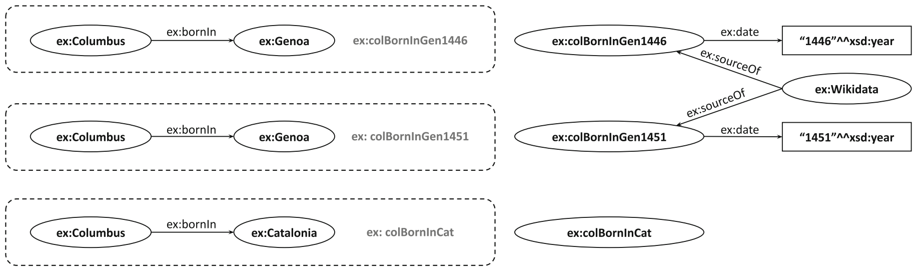


Fig. 2. Example of abstract reification

2 Reifying RDF Statements

RDF can only represent binary relations between two entities, but does not natively allow to express information about the relations themselves [28]. Figure 1 shows an RDF graph composed of five statements that describe *Christopher Columbus*. It gives his *name* and two birthplaces (using the predicate `ex:bornIn`), which correspond to those given by two information sources. So, *what is his real birthplace?* We cannot answer this question with the information we have.

In order to state something about the context of a statement itself, it is necessary to resort to reification. *Reification* stands for the mechanism by which a new term (that we call the *anchor*) is used to represent the statement [18]. The anchor is used as subject and/or object in other triples to provide the *contextual annotation* (*e.g.*, the source of the information, a temporal or spatial dimension, etc.). Figure 2 shows an abstract representation of the reification of two of the triples in Fig. 1, one of them associated with an anchor and the other with two (left), and their contextual annotations (right). As can be seen, the triple that says *Columbus* was born in *Genoa* is now qualified with two different contextual annotations, each indicating a different date, and both using *Wikidata* as their source. On the contrary, for the statement that *Columbus* was born in

Table 1. Triples introduced by each reification function

RDF Reification	N-Ary Relations	Singleton Properties	NdFluents
(a,rdf:subject,s)	(s,p_1,a)	(s,a,o)	(s_a,nd:contextualPartOf,s)
(a,rdf:predicate,p)	(a,p_2,o)	(a,rdf:singletonPropertyOf,p)	(o_a,nd:contextualPartOf,o)
(a,rdf:object,o)	(p,subjectProperty,p_s)	(a,rdf:type,rdf:SingletonProperty)	(s_a,nd:contextualExtent,a)
(a,rdf:type,rdf:Statement)	(p,objectProperty,p_o)		(o_a,nd:contextualExtent,a)
			(s_a,rdf:type,nd:ContextualPart)
			(o_a,rdf:type,nd:ContextualPart)
			(a,rdf:type,nd:ContextualExtent)

Catalonia, there is actually no information in the contextual annotation (the anchor `ex:ccBornInCat` is not connected to anything).

Current reification approaches can be categorized into two main families. On the one hand, there are approaches that apply a *reification function*, *i.e.*, a function that maps a triple and a contextual annotation to a new graph, where the original statement is replaced by a set of triples that connect the anchor with both the terms of the statement and the contextual annotation. For example, traditional RDF reification [4] replaces the statement `(s,p,o)` by the set of triples `(a,rdf:subject,s)`, `(a,rdf:predicate,p)`, and `(a,rdf:object,o)`, where `a` is the anchor term that will be used to refer to the statement in the contextual annotation. Other such approaches are n-ary relations [31], the singleton property [30], the companion properties [13], NdFluents [16], and NdProperties [14]. Each approach generates a different number of triples to implement reification, as shown in Table 1, which lists the triples added by RDF reification, n-ary relations, singleton property, and NdFluents (evaluated in Sect. 6).[2] On the other hand, other approaches extend the RDF syntax and/or semantics to link the statements and their corresponding anchors. For example, RDF-Star [17] extends the definition of term to allow for a triple itself to be used as subject or object in other statements. Other such approaches are Notation 3 [2] or using Named Graphs [6], with each named graph containing only a reified statement.

Finally, note that RDF-Star and N3 follow a *"quoted triples"* model, where the triple (or a unique identifier thereof) can be used as a term in the subject or object position of another statement. This means that a statement can have at most one contextual annotation. On the other hand, reification functions and Named Graphs follow a model of *"contextualized statements"*, allowing a triple to be represented by different anchors, and therefore to have an indefinite number of different contextual annotations for it, each one in a different context. Some approaches, such as NdFluents, go even further, creating contextual versions of individuals to allow inference for sets of statements within the same context.

3 Related Work on Efficient Reification Representation

Efficiently representing KGs containing reified statements remains an open challenge. Hernández et al. [19,20] compare the management of several reification

[2] Note that the triples that indicate the types of some individuals can be considered as optional, since they can be inferred by the semantics of the vocabularies.

approaches by different triplestores, concluding that using *named graphs* is the most efficient solution. Moreover, triplestores commonly support named graphs natively, hence this is a relatively simple and practical approach, at the cost of losing named graphs for other purposes (*e.g.*, versioning). On the other hand, there are some triplestores (such as AnzoGraph[3], Blazegraph[4], GraphDB[5], or Stardog[6]) which are able to manage the RDF-Star [17] extension, so they can be used to store and query reified statements modeled in this way.

HDTQ [12] allows multiple named graphs to be managed and queried in compressed form. Like our approach, HDTQ extends HDT, but in this case, it proposes a *Quad Information* component that indexes whether a particular triple appears in a particular graph, allowing quad pattern queries (triple patterns augmented with graph information) to be performed efficiently in-memory.

TrieDF [32] also proposes an in-memory architecture to handle RDF data augmented with any type of metadata. For this purpose, it regards tuples of arbitrary length: `<subject>, <predicate>, <object>,` <a$_1$> ... <a$_n$>, where <a$_i$> are annotations, and models them using tries [3]. The resulting representation is able to compress shared prefixes (of any length) between tuples, and supports fast prefix-based retrieval. TrieDF shows competitive performance for managing statements with 1 and 2 annotations, consolidating a first step toward managing arbitrary levels of metadata in an application-agnostic manner.

Finally note that all of the approaches mentioned in this section follow the *"contextualized statements"* model, except for triplestores that support RDF-Star, which implements the *"quoted triples"* model.

4 HDT

HDT (Header-Dictionary-Triples) is a framework originally designed to optimize the storage and transmission of RDF data [11], and then enhanced to support efficient querying of triple patterns [23]. This versatility has allowed HDT to be adopted as backend for many tools in the Semantic Web community, such as Triple Pattern Fragments for client-based triple pattern querying over the Web [34], in natural language query answering systems such as WDAqua-core1 [9], or in SPARQL endpoints in commodity hardware [36]. It has also been used for data archiving [1,35], encoding one of the largest datasets from the LOD cloud (with more than 28 billion triples) [10], and for efficiently computing the PageRank summarization of datasets [8], among other applications.

HDT transforms the RDF graph into two main elements, illustrated in Fig. 3, which are then independently encoded to remove their respective redundancies: (i) the *dictionary*, which maps each RDF term used in the dataset (*i.e.*, its vocabulary) to a unique integer ID, and (ii) the *ID-graph*, which replaces the original terms in the RDF graph by their corresponding IDs in the dictionary.

[3] https://cambridgesemantics.com/anzograph/.
[4] https://www.blazegraph.com/.
[5] http://graphdb.ontotext.com/.
[6] https://www.stardog.com/.

That is, each triple is transformed into a tuple of 3-integer IDs (*ID-triples*): $< id_s, id_p, id_o >$; where id_s, id_p, and id_o are the IDs of the subject, predicate, and object in the dictionary, respectively.

HDT Dictionary. HDT proposes a practical dictionary implementation, called *Four Section Dictionary*, where each section SO, S, O, and P is independently sorted and compressed using Front Coding [24]. In this way, each section has its own mapping to exclusively identify its terms:

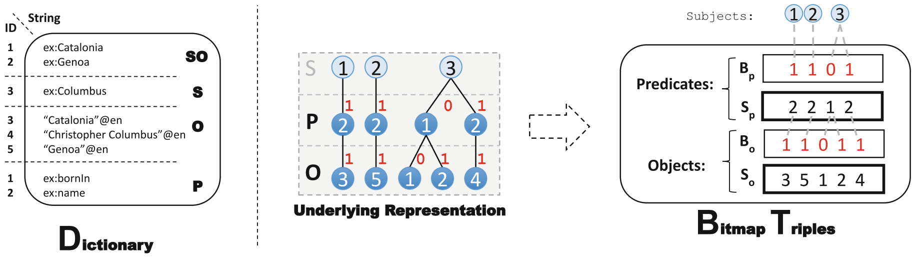


Fig. 3. HDT Dictionary and Triples components

– The section SO maps shared *subjects-objects* (*i.e.*, terms that play both subject and object roles) to $[1, |SO|]$, where $|SO|$ is the number of RDF terms that appear as subject and object in the set of triples.
– The section S maps single *subjects* (not appearing as objects) to $[|SO| + 1, |SO| + |S|]$, where $|S|$ is the number of RDF terms that act only as subjects.
– The section O maps single *objects* (not appearing as subjects) to $[|SO| + 1, |SO| + |O|]$, where $|O|$ is the number of RDF terms that act only as objects.
– The section P maps *predicate* to $[1, |P|]$, where $|P|$ is the number of RDF terms that appear as predicate in the triples.

The number of RDF terms in each section and how they are serialized are stored in the HDT *Header*, together with other HDT metadata. Figure 3 (left) shows the resulting Dictionary for the example given in Fig. 1. Note that the mapping ID-term is implicitly encoded by the position of the term in its section.

The Dictionary is encoded using compact data structures [29], which self-index each section and efficiently resolve two main operations:

– $locate(term, role)$ returns the unique ID for the given term and role (*subject*, *predicate*, or *object*), if their combination exists in the dictionary.
– $extract(id, role)$ returns the term for the given ID and role, if their combination exists in the dictionary.

HDT Triples. They provide an interface for accessing the ID-graph using SPARQL triple patterns [33]. A practical implementation, called *Bitmap Triples*, transforms the ID-graph into a forest of three-level trees, where each tree is rooted by a subject ID with its adjacency list of predicates in the second level and, for each of them, the adjacency list of related objects in the third (leaf) level. Figure 3 (middle) illustrates this organization.

In practice, the entire SPO forest is then encoded using compact data structures [29] to ensure efficient data retrieval in minimal space. On the one hand, two *integer sequences* are used to encode each level of the forest: S_p for predicate IDs and S_o for object IDs. On the other hand, two *bitsequences* are used to encode the shape of each tree: B_p encodes the number of branches hanging from each subject, and B_o encodes the number of leaves in each branch. This is easily done by simply marking the last children of the parent node (in the previous level) with 1 bits and the rest with 0 bits. This is illustrated in Fig. 3 (right). Note that this representation allows each triple to be identified by the position of its object in S_o, a fundamental feature for building HDTr, as described in the next section.

All structures used in Bitmap Triples support positional access to their values, and bitsequences provide two additional operations, which are essential for traversing the SPO forest:

- $rank_a(B, i)$ counts the number of occurrences of $a \in \{0, 1\}$ in $B[1, i]$, for any $1 \leq i \leq n$; $rank_a(B, 0) = 0$.
- $select_a(B, j)$ returns the position of the j^{th} occurrence of $a \in \{0, 1\}$ in B, for any $1 \leq j \leq n$; $select_a(B, 0) = 0$ and $select_a(B, j) = n{+}1$, if $j > rank_a(B, n)$.

Bitmap Triples is able to resolve the four SPARQL triple patterns binding the subject[7]: `(s,p,o)`, `(s,p,?o)`, `(s,?p,o)`, and `(s,?p,?o)`. First, it translates the bound terms in the triple pattern to their corresponding IDs: $< id_s$,id_p,$id_o >$, using the Dictionary `locate(term,role)` method, and then traverses the ${id_s}^{th}$ tree, as follows:

1. Predicates related to the subject id_s are located at $S_p[p_x, p_y]$, $p_x = select_1(B_p, id_s - 1) + 1$ and $p_y = select_1(B_p, id_s)$. If the predicate is bound, id_p is binary searched in $S_p[p_s, p_y]$, returning pos_p if there is any triple connecting id_s and id_p $(p_x \leq pos_p \leq p_y)$.
2. Objects related to a pair $< id_s$,$id_p >$ are located at $S_o[o_x, o_y]$, $o_x = select_1(B_o, pos_p{-}1){+}1$ and $o_y = select_1(B_o, pos_p)$. If the object is also bound, id_o is binary searched in $S_o[o_x, o_y]$, returning pos_o if the triple $< id_s$,id_p,$id_o >$ exists in the graph $(o_x \leq pos_o \leq o_y)$.

Finally, the original terms in the returned triples are retrieved using the existing $extract(id, role)$ method in the Dictionary.

Note that the pattern `(?s,?p,?o)`, which returns all existing triples, can also be resolved by traversing all trees sequentially and retrieving the original terms from the Dictionary. To resolve the remaining triple patterns with unbound subject (*i.e.*, `(?s,p,o)`, `(?s,p,?o)`, and `(?s,?p,o)`), *HDT-FoQ* (HDT Focused on Querying) [23] extends Bitmap Triples with additional structures, which ensure efficient predicate- and object-based retrieval while keeping the Triples component in compressed form.

[7] `?` is used to indicate variables in the triple pattern.

5 Extending HDT for Reified Triples: HDTr

In this section we present HDTr, a binary serialization for reified statements. HDTr takes advantage of the structure of the Triples component in HDT, where an RDF statement is implicitly identified by the position of its object in S_o, to connect reified statements with their anchors in the Dictionary. HDTr files are backward compatible with HDT, so HDTr can be easily adopted by existing HDT-based applications that need to manage reified triples.

Table 2. Five Section Dictionary configuration

Section	IDs	
SO $= SO_T \cup SO_A$	$[1, \|SO\|]$,	where $SO = \|SO_T\| + \|SO_A\|$
S $= S_T \cup S_A$	$[\|SO + 1\|, \|SO + S\|]$,	where $\|S\| = \|S_T\| + \|S_A\|$
O $= O_T \cup O_A$	$[\|SO + 1\|, \|SO + O\|]$,	where $\|O\| = \|O_T\| + \|O_A\|$
P $= P_T$,	$[1, \|P\|]$,	where $\|P\| = \|P_T\|$
A $= SO_A \cup S_A \cup O_A \cup U_A$	$[1, \|A\|]$,	where $\|A\| = \|SO_A\| + \|S_A\| + \|O_A\| + \|U_A\|]$

5.1 The HDTr Dictionary

Reified statements include terms used as anchors, which can then be used in the subject and/or object position in contextual annotation triples. The HDTr Dictionary extends the logical model of the Four Section Dictionary into a *Five Section Dictionary*, adding a new section for the anchors, referred to as A, while preserving SO, S, O and P, which now encode the terms used in regular and contextual triples, according to their respective roles.

The Five Section Dictionary is implemented in practice as two subdictionaries. On the one hand, the *Triples Dictionary* rearranges all terms that do not play as anchors in a Four Section Dictionary: SO_T, S_T, O_T, and P_T.[8]. This component is essential to ensure HDT and HDTr compatibility. On the other hand, the *Anchors Dictionary* manages only anchor terms, so they are organized in sections for shared subject-objects, subjects and objects: SO_A, S_A, and O_A, as well as U_A for "unused" anchors (*i.e.*, not used as terms in any triple).[9]. At the physical level, each section of both subdictionaries is also compressed using Front Coding [24], as in HDT, so it is lexicographically sorted before encoding it. This decision also ensures that $locate(element, role)$ and $extract(id, role)$ operations are performed efficiently in all dictionary sections.

Figure 4 (left) shows the resulting Five Section Dictionary and its corresponding implementation as two subdictionaries for our example. Note that all anchor terms (shown in Fig. 2) are organized in the section corresponding to their role in the contextual annotation triples, while the Triples subdictionary contains the same terms as the HDT Dictionary had for the non-reified statements (Fig. 3) plus the additional terms appearing in the contextual annotation (Table 2).

[8] The subscript T is added to the names of the sections to indicate that they belong to the Triples Dictionary.

[9] The subscript A refers to the Anchors Dictionary.

5.2 The HDTr Triples

The Triples component must handle the need of compactly connecting reified statements to their anchors. This can be done by identifying each triple by the position of its object in S_o; *i.e.*, the object of the i^{th} triple is encoded at $S_o[i]$. This HDT feature allows reification to be efficiently implemented in the Triples component by adding two compact data structures: (i) a bitsequence B_a, that marks with 1 bits the positions corresponding to reified triples ($B_a[i] = 1$, *iff* the i^{th} triple is reified); and (ii) a sequence P_a that encodes anchor IDs (in the Dictionary) for all reified triples. At the physical level, B_a is implemented as B_p and B_o in BitmapTriples, while P_a is implemented as a compact permutation [27] that provides two basic methods:

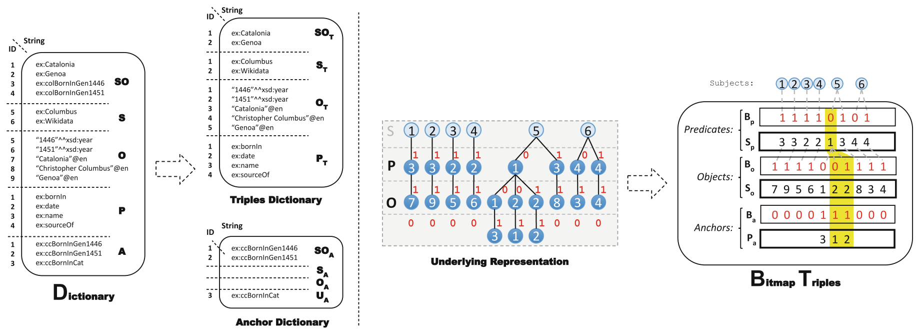


Fig. 4. HDTr Dictionary and Triples components

- $\pi(P_a, j)$: returns the value at $P_a[j]$; it allows to efficiently retrieve the anchor ID for the reified statement j^{th}.
- $\pi^{-1}(P_a, k)$: returns the position at where value k is stored in P_a; it allows to efficiently find the (reified) statement connected to the anchor with ID k.

It is worth noting the particular case where a triple $< id_s, id_p, id_o >$ is annotated with two or more different anchors. In this situation, each pair (triple, anchor) is considered as a different statement, and id_o is encoded in S_o as many times as anchors are related to the triple.

Figure 4 (right) illustrates the HDTr Triples configuration that encodes the statements from the running example. The "Underlying Representation" shows how (the 3rd, the 4th, and the 6th) statements are logically connected to their anchors, while the Anchors level (at the "Bitmap Triples" representation) illustrates its physical encoding; note that $B_a[3] = B_a[4] = B_a[6] = 1$ and $Pa[1] = 3$ because the first reified triple is connected to anchor 3, $Pa[2] = 2$ because the second reified triple is connected to anchor 2, and so on. For example, the highlighted statement states that *Christopher Columbus* (subject 5) *was born in* (predicate 1) *Genoa* (object 2), and it is identified with `ex:ccBornInGen1446` (anchor 1) and `ccBornInGen1451` (anchor 2).

5.3 Querying HDTr

HDT files can be loaded as HDTr files, so that all SPARQL triple patterns can be resolved efficiently. However, the HDT retrieval methods needs to be extended to query reified statements, *i.e.*, a query containing a triple pattern (tp) and an anchor (a), where any component of tp and a can be a variable. Algorithm 1 describes the proposed method for querying reified statements[10], with two main flows depending on whether the anchor is provided (Line 1–5) or not (Lines 6–18).

Algorithm 1. `query`(tp, a)

```
1:  if $bound(a)$ then
2:      $pos_a \leftarrow \pi^{-1}(P_a, a)$
3:      $pos_t \leftarrow select_1(B_a, pos_a)$
4:      $t \leftarrow check(pos_t, tp)$
5:      return$(t, a)$
6:  else
7:      $T \leftarrow search(tp)$
8:      for $t \in T$ do
9:          if $(B_a[pos_t] = 1)$ then
10:             $pos_a \leftarrow rank_1(B_a, pos_t)$
11:             $id_a \leftarrow \pi(P_a, pos_a)$
12:             $r.add(t, id_a)$
13:         else
14:             $r.add(t)$
15:         end if
16:     end for
17:     return $r$
18: end if
```

Let us suppose that we search a statement with the subject `ex:Columbus` and identified by the anchor `ex:ccBornInGen1451` ($tp =< 5, ?p, ?o >, a = 2$). Intuitively, we need to navigate the tree in Fig. 4 (right) from bottom (once we locate the anchor) to top (the subject). In line 2, the anchor ID is used to find its single occurrence in P_a: $pos_a \leftarrow \pi^{-1}(P_a, 2) = 3$ and then, in line 3, its reified statement is obtained from B_a: $pos_t \leftarrow select_1(B_a, 3) = 7$. The pos_t is then used to traverse up the tree encoding the triple (as in HDT [23]), checking that the variables are satisfied (line 4); all variables in tp are bound to their corresponding IDs ($?p = 1, ?o = 2$) and the result is returned (line 5), as ($< 5, 1, 2 >; 2$).

In the second case, let us suppose that we search the *birth place* of *Christopher Columbus* and all related annotations ($tp =< 5, 1, ?o >, a = ?a$). In this situation, we proceed top to bottom in Fig. 4 (right), so tp is first searched (line 7) using the original HDT methods [23]. Note that, in HDT, for each matching triple t (line 8), its position, pos_t, is immediately known, *e.g.*, <5,1,1> has position 5. Thus, the algorithm iterates over each matching triple and checks if it is reified (line 9); otherwise the triple is returned with no anchor value (line 14). In our example, three statements are annotated: $B_a[5] = B_a[6] = B_a[7] = 1$. In line 10, the anchor of the triple is identified in B_a (for the first triple, $pos_a \leftarrow rank_1(B_a, 5) = 1$) and its ID is retrieved from P_a in line 11 ($id_a \leftarrow \pi(P_a, 1) = 3$). Finally, the matching triple and anchor are added to the resultset, in our example $(< 3, 1, 1 >; 3), (< 3, 1, 2 >; 1), (< 3, 1, 2 >; 2)$.

[10] For the sake of simplicity, we assume that tp and a have previously mapped to IDs and, conversely, the returned resultset is then mapped to their corresponding terms.

6 Evaluation

HDTr extends the theoretical specification of HDT to incorporate reified triples, including a dictionary interface separate from the implementation, and retrieval algorithms. The implementation itself makes several design choices: using two subdictionaries, leveraging the structure of Bitmap Triples by using the object positions to reference triples, using a permutation as a bidirectional link between anchors and triples, and adapted serialization and query algorithms. In this section, we analyze the performance of HDTr[11] and we evaluate the impact of these decisions against a selected state-of-the-art baseline. For that, we use data from *NELL* [5,26], a system that learns categories and relations from the Web, keeping track of their provenance. Specifically, we use *NELL2RDF* [15], which extracts the beliefs of NELL and their provenance contextual information into RDF. In order to test the impact of the proportion of reified triples, we create two evaluation scenarios: (i) we keep in the dataset all triples extracted from NELL, including their contextual annotations, to ensure a low proportion of refied statements *w.r.t.* the total number or triples ($reif_{CA}$ scenario); and (ii) we remove the contextual annotations, leaving only the reified triples, to achieve a high proportion of reified statements ($reif_{\emptyset}$ scenario). All KGs are generated using the reification functions provided by NELL2RDF: RDF reification (*Reif*), n-ary relations (*N-Ary*), singleton properties (*SP*), and NdFluents (*NdF*), as well as *n-quads*. Finally, we use three randomized slices of NELL's beliefs: the *full* dataset, *half* of the statements, and a *quarter* of the statements, with the goal of testing the scalability of our design choices. The resulting KGs are described in Table 3, showing the number of statements per dataset, its size, and either

Table 3. Summary of NELL2RDF datasets (in # millions of statements, GB, and percentage)

File		Full Data			Half Data			Quarter Data		
		#	Size	Prop.	#	Size	Prop.	#	Size	Prop.
Reif	$reif_{CA}$	1,069.6	239.2	8.00%	551.7	123.1	8.16%	284.2	63.3	8.23%
	$reif_{\emptyset}$	113.7	21.5	75.26%	59.7	11.3	75.37%	31.0	5.8	75.42%
N-Ary	$reif_{CA}$	1,012.7	229.1	2.84%	521.8	117.8	2.91%	268.7	60.5	2.93
	$reif_{\emptyset}$	56.8	11.4	50.52%	29.9	6.0	50.75%	15.5	3.1	50.85
SP	$reif_{CA}$	1,041.1	234.4	5.49%	536.7	120.6	5.61%	276.4	62.0	5.65%
	$reif_{\emptyset}$	85.3	16.7	67.01%	44.8	8.7	67.16%	23.2	4.5	67.23%
NdF	$reif_{CA}$	1173.3	254.0	16.14%	600.7	130.0	15.65%	308.3	66.6	15.40%
	$reif_{\emptyset}$	217.5	36.3	87.07%	108.7	18.1	86.47%	55.1	9.2	86.17%
Quads	$reif_{CA}$	984.0	224.6	2.18%	506.6	115.4	2.90%	260.8	59.3	2.92%
	$reif_{\emptyset}$	28.1	6.9	100%	14.7	3.6	2.90%	7.6	1.8	2.92%

[11] We use a HDTr prototype implemented in C++. See the supplemental material statement at the end of the document.

the proportion of triples that are generated by the reification function or the proportion of quads in the dataset.

We first assess the performance of HDTr with respect to encoding with HDT[12] the KG resulting from the application of any reification function. This is an important comparison because HDTr files can directly replace HDT files in applications that use reified KGs.

We then measure space-time tradeoffs of HDTr against some well-known triplestores. Arguably, HDT and HDTr are not directly comparable with triplestores, since they are production systems that include additional overheads to fully conform to SPARQL (including datatype filter expressions). However, this evaluation allows us to show in a quantitative way how an index such as HDTr compares with those used in such triplestores, as well as to position HDTr in the context of similar HDT comparisons. The triplestore experiments were made loading and querying quads, using named graphs to store the anchor, since this is reportedly the most efficient approach for triplestores to manage reified triples [19,20].

HDTQ and TrieDF were also included in the planned evaluation, but HDTQ failed in all the experiments performed due to memory limitations,[13] while the published code of TrieDF does not allow to serialize files different to those provided for their experiments.

All experiments were performed in a cluster in which the same resources are always reserved: 2 cores and 320 GB of RAM. The same installation and configuration is used for HDT, HDTr, and all triplestores, which received basic optimizations to use the reserved resources according to the instructions by their publishers. To avoid as much skew as possible, we report the average of 10 independent executions.

6.1 HDTr vs. HDT

We first compare HDTr with its counterpart HDT in terms of space requirements and retrieval performance.

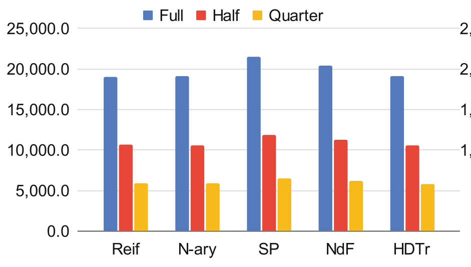
Space Requirements. Figure 5 compares the space requirements of HDTr with those of HDT for all reification functions in the two $reif_{CA}$ and $reif_{\emptyset}$ scenarios: Figs. 5 a-b compare the size of the serialized KG, whereas Figs. 5 c-d report the size of the indexes required for efficient triple pattern resolution [23].

HDTr clearly outperforms HDT in the $reif_{\emptyset}$ scenario, because it represents the same information with fewer triples and encodes them more compactly. We can observe that the size of the HDTr files is smaller than the HDT files with any

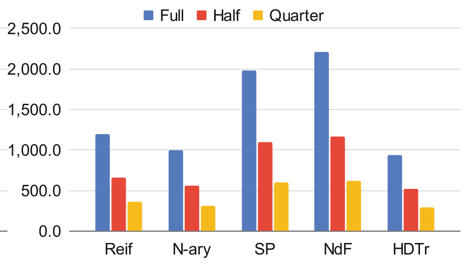
[12] We use the HDT C++ library. Please find the concrete forked version and additional details in the supplemental material statement specified at the end of this document.

[13] Our hypothesis is that HDTQ was designed to encode named graphs, making assumptions (*e.g.*, the proportion of named graphs to triples or the use of named graphs as terms in statements) that have a negative impact in its ability to encode reified statements.

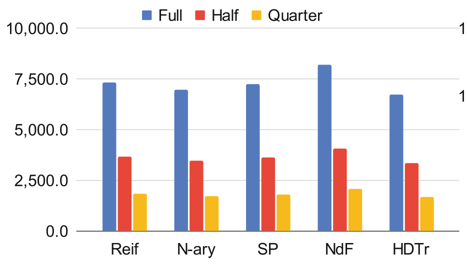
contextualization approach, ranging 6% for n-ary relations to 237% for NdFluents. The difference for the indexes is even more pronounced, going from 263% to 1461% for the same approaches. However, if we look at $reif_{CA}$ scenario, we see that the HDTr and the HDT files for RDF reification and n-ary relations have similar size, and the space savings for the singleton property and NdFluents have been reduced. Index sizes are also lower for HDTr, although to a lesser extent than in the $reif_{\emptyset}$ case. This shows that the HDTr savings, due to its additional structures to identify reified statements by its position and connect them with their anchor in the dictionary, are proportional to the number of statements that are reified and the number of triples introduced by each reification function.



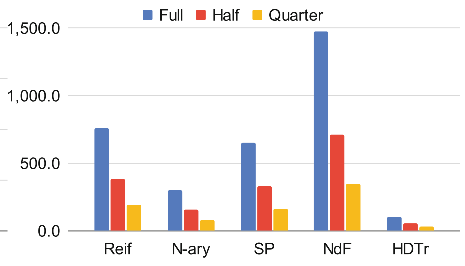
(a) HDT/HDTr sizes ($reif_{CA}$)



(b) HDT/HDTr sizes ($reif_{\emptyset}$)



(c) Index Sizes ($reif_{CA}$)



(d) Index Sizes ($reif_{\emptyset}$)

Fig. 5. Size of HDT and HDTr files and their indexes (in MB)

Retrieval Performance. HDTr is able to identify reified triples and connect them with their anchor in the dictionary thanks to the additional bitsequence and permutation. This allows to efficiently obtain a triple and its anchor with a single quad pattern. However, HDT needs to translate them into a set of triple patterns, depending on the reification approach that they encode. Table 4 lists the set of triple patterns that must be resolved to answer the equivalent quad pattern for each reification function; for example, to resolve the pattern `(s,?,?,?)`, which asks for all statements related to subject `s` and their anchors, *HDT+Reif* resolves `(?,rdf:subject,s)` to get the anchors of the statement (corresponding to the pattern type `(?,p,o)` in the table) and then `(a,rdf:predicate, ?)` and `(a,rdf:object,?)` for each anchor (corresponding to the pattern type `(s,p,?)` twice). To estimate a lower bound for HDT, we compute, for each triple, the average retrieval time for each triple pattern, and sum together the times for all of them, ignoring the cost of the joint operations. Note that this is an optimistic estimate in favor of HDT. We then average this time and compare it to the average retrieval time of a quad pattern type for the same set of statements in HDTr.

Table 4. Types of patterns needed to obtain a triple and its anchor

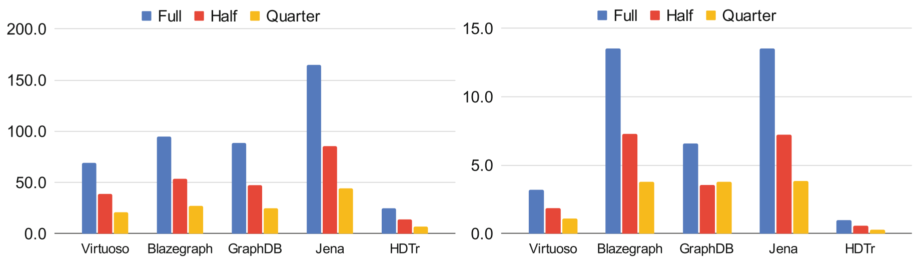
Quad	(s,?,?,?)	(?,p,?,?)	(?,?,o,?)	(s,p,?,?)	(s,?,o,?)	(?,p,o,?)	(s,p,o,?)	(?,?,?,a)
Reif	(?,p,o)	(?,p,o)	(?,p,o)	(?,p,o)	(?,p,o)	(?,p,o)	(?,p,o)	(s,p,?)
	(s,p,?)	(s,p,?)	(s,p,?)	(s,p,o)	(s,p,?)	(s,p,o)	(s,p,o)	(s,p,?)
	(s,p,?)	(s,p,?)	(s,p,?)	(s,p,?)	(s,p,?)	(s,p,?)	(s,p,o)	(s,p,?)
N-Ary	(s,?,?)	(s,p,?)	(?,?,o)	(s,p,?)	(s,?,?)	(s,p,?)	(s,p,?)	(?,?,o)
	(?,p,o)	(s,p,?)	(?,p,o)	(s,p,?)	(s,?,?)	(s,p,?)	(s,p,?)	(?,p,o)
	(s,p,?)	(?,p,?)	(s,p,?)	(s,p,?)	(s,p,?)	(?,p,o)	(s,p,?)	(s,p,?)
	(s,p,?)	(s,p,?)	(?,p,o)	(s,p,?)	(s,p,?)	(?,p,o)	(s,p,o)	(s,p,?)
SP	(s,?,?)	(?,p,o)	(?,?,o)	(s,p,?)	(s,?,o)	(?,p,o)	(?,p,o)	(?,p,?)
	(s,p,?)	(?,p,?)	(?,p,o)	(?,p,o)	(s,p,?)	(?,p,o)	(s,p,o)	(s,p,?)
NdF	(?,p,o)	(?,p,?)	(s,p,?)	(?,p,o)	(?,p,o)	(s,p,o)	(?,p,o)	(?,p,o)
	(s,?,?)	(s,p,?)	(?,?,o)	(s,p,?)	(?,p,o)	(?,p,o)	(s,p,?)	(s,?,o)
	(s,p,?)	(s,p,?)	(s,p,?)	(s,p,?)	(s,?,o)	(s,p,?)	(s,p,o)	
		(s,p,?)	(s,p,?)	(s,p,?)	(s,p,?)	(s,p,?)	(s,p,?)	

We extract 10,000 different triple patterns of each type and report the average execution per pattern type in Fig. 7 for resolving query patterns in the $reif_{\emptyset}$ scenario (similar results are obtained in the $reif_{CA}$ scenario). HDTr clearly dominates the comparison, outperforming HDT (with any reification function) in five out of eight cases and being comparable to the best alternative in the other three: `(s,?,o,?)`, `(s,p,o)`, and `(?,?,?,a)`. The comparison is similar for the three slices (*Full*, *Half*, and *Quarter*), demonstrating that HDTr retains the scalability features of HDT.

6.2 HDTr vs. Triplestores

We now compare HDTr against a representative baseline of four well-known triplestores: Virtuoso 7.2.6, Blazegraph 11.0.19, GraphDB 8.8.1, and Fuseki 3.10, using the quads datasets for all of them. Comparison is also performed in terms of space requirements and retrieval performance.

Space Requirements. Fig. 6 compares the space requirements of HDTr (including data and indexes) with respect to the evaluated triplestores. HDTr dominates the comparison in both scenarios. *HDTr* exploits its compressibility and uses more than 3 times less space than *Virtuoso* (for the *Full* slice) but up to 13 times less space than *Blazegraph* or *Fuseki*. These numbers demonstrate HDTr's ability to save disk space, which in turn enable scalable and efficient in-memory data management.



(a) Data $reif_{CA}$ (b) Data $reif_{\emptyset}$

Fig. 6. Size of HDTr files and triplestores data (in GB)

Retrieval performance. Figure 8 reports query times for HDTr and the triplestores in the $reif_{\emptyset}$ scenario, using a testbed containing 10,000 randomly generated quad patterns, as explained earlier. HDTr's performance is clearly superior for seven out of eight patterns, with reported query times between 2 and 4 orders of magnitude faster, depending on the pattern and the triplestore. HDTr shows slower performance only with the pattern `(?,p,?,?)`, which is a known weakness of HDT [21]. The same conclusions can be drawn for the $reif_{CA}$ scenario, and are consistent with the previous results comparing HDT and triplestores [23], confirming that HDTr is able to compete with the most prominent state-of-the-art solutions with guarantees.

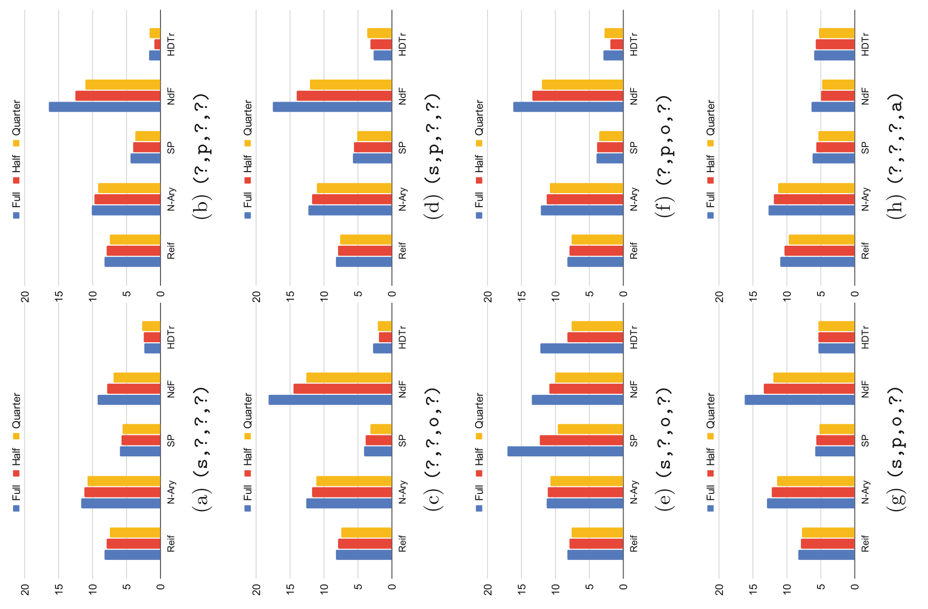


Fig. 7. HDTr vs HDT estimated retrieval time (in μs.)

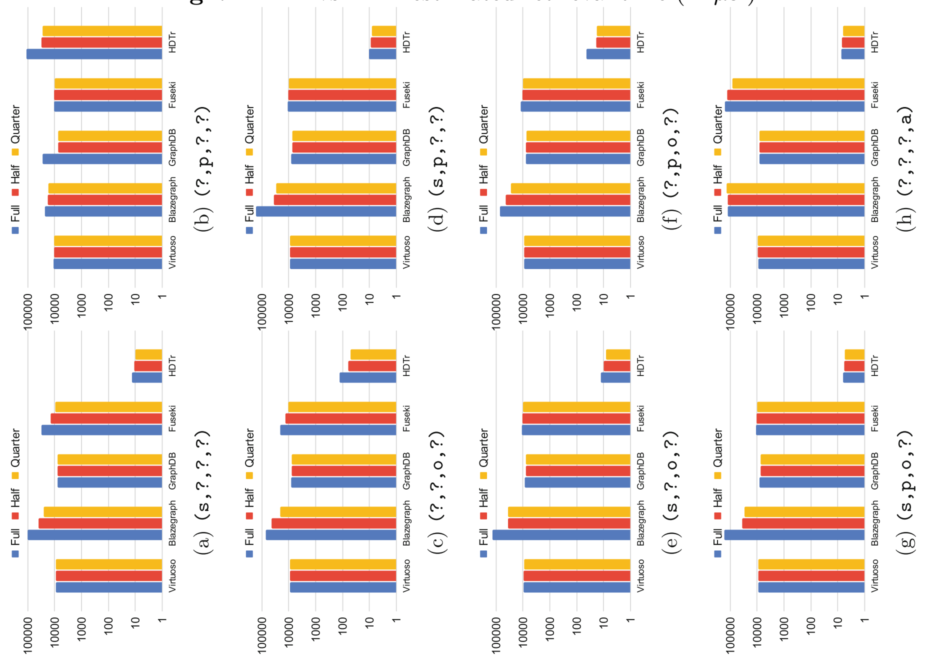


Fig. 8. HDTr vs triplestores quad retrieval time (in μs.)

7 Conclusions and Future Work

This work presents HDTr, an extension of HDT to serialize reified statements in a model-agnostic binary representation that is compact and queryable. Our proposal is HDT compatible and leverages its current compact structures to make a connection between the reified statements and the anchors to their contextual annotations. Our evaluation against four triplestores, Virtuso, Blazegraph, GraphDB, and Jena Fuseki, shows that HDTr is between two and four orders of magnitude faster than all triplestores when querying for reified triples for all patterns patterns but one, keeping the compelling compression (at least 350% gains) and loading results of HDT.

As future work, we plan to leverage, even more, the existing redundancies in the data. First, we aim at exploring the possibility of splitting the reified and non-reified statements in the HDT components. Then, we plan to extend HDTr with the option of not storing anchors terms that are merely a placeholder. This would be specially useful for reification approaches where this is given (such as RDF-star). Extensions for full SPARQL resolution and using HDTr as a support to convert between different reification approaches are also considered.

Supplemental Material Statement: Source code is available at https://github.com/jm-gimenez-garcia/hdtr-cpp. The HDT source code at fork time is available at https://github.com/jm-gimenez-garcia/hdtr-cpp/tree/base. Data and instructions to replicate the evaluation, as well as additional experimental results on loading times, are available at https://doi.org/10.6084/m9.figshare.22787495.

Acknowledgments. This work has been partially funded by the Spanish Ministry of Science and Innovation through LOD.For.Trees (TED2021-130667B-I00), EXTRA-Compact (PID2020-114635RB-I00), and PLAGEMIS-UDC (TED2021-129245B-C21) projects, and from the EU H2020 research and innovation program under the Marie Skłodowska-Curie grant No 642795.

References

1. Beek, W., Rietveld, L., Bazoobandi, H.R., Wielemaker, J., Schlobach, S.: LOD laundromat: a uniform way of publishing other people's dirty data. In: ISWC (2014)
2. Berners-Lee, T., Connolly, D.: Notation3 (N3): A readable RDF syntax. W3C (2011)
3. Briandais, R.D.L.: File searching using variable length keys. In: IRE-AIEE-ACM (1959)
4. Brickley, D., Guha, R.V.: RDF vocabulary description language 1.0: RDF schema. In: W3C (2004)
5. Carlson, A., Betteridge, J., Hruschka, E.R., Mitchell, T.M.: Coupling Semi-supervised Learning of Categories and Relations. In: SSLNLP (2009)
6. Carroll, J.J., Bizer, C., Hayes, P.J., Stickler, P.: Named graphs. JWS **3**(4), 247–267 (2005)
7. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 concepts and abstract syntax. In: W3C (2014)

8. Diefenbach, D., Both, A., Singh, K., Maret, P.: Towards a question answering system over the Semantic Web. Semantic Web **11**(3), 421–439 (2020)
9. Diefenbach, D., Thalhammer, A.: PageRank and Generic Entity Summarization for RDF Knowledge Bases. In: ESWC (2018)
10. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: LOD-a-lot: a queryable dump of the LOD cloud. In: ISWC (2017)
11. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). JWS **19**, 22–41 (2013)
12. Fernández, J.D., Martínez-Prieto, M.A., Polleres, A., Reindorf, J.: HDTQ: Managing RDF Datasets in Compressed Space. In: ESWC (2018)
13. Frey, J., Müller, K., Hellmann, S., Rahm, E., Vidal, M.-E.: Evaluation of metadata representations in RDF stores. SWJ **10**(2), 205–229 (2017)
14. Giménez-García, J.M., Duarte, M., Zimmermann, A., Gravier, C., Hruschka Jr., E.R., Maret, P.: NELL2RDF: Reading the web, tracking the provenance, and publishing it as linked data. In: CKG (2018)
15. Giménez-García, J.M., Zimmermann, A.: NdProperties: encoding contexts in RDF predicates with inference preservation. In: CKG (2018)
16. Giménez-García, J.M., Zimmermann, A., Maret, P.: NdFluents: an ontology for Annotated Statements with Inference Preservation. In: ESWC (2017)
17. Hartig, O., et al.: RDF-star and SPARQL-star. In: W3C (2021)
18. Hayes, P.J., Patel-Schneider, P.F.: RDF 1.1 semantics. In: W3C (2014)
19. Hernández, D., Hogan, A., Krötzsch, M.: Reifying RDF: What Works Well With Wikidata? SSWS (2015)
20. Hernández, D., Hogan, A., Riveros, C., Rojas, C., Zerega, E.: Querying Wikidata: Comparing SPARQL. Relational and Graph Databases, ISWC (2016)
21. Hernández-Illera, A., Martínez-Prieto, M.A., Fernández, J.D., Fariña, A.: iHDT++: improving HDT for SPARQL triple pattern resolution. JIFS **39**(2), 2249–2261 (2020)
22. Hogan, A., et al.: Knowledge Graphs. Springer, Cham (2021). https://doi.org/10.1007/978-3-031-01918-0
23. Martínez-Prieto, M., Arias, M., Fernández, J.: Exchange and consumption of huge RDF Data. In: ESWC (2012)
24. Martínez-Prieto, M.A., Brisaboa, N.R., Cánovas, R., Claude, F., Navarro, G.: Practical compressed string dictionaries. IS **56**, 73–108 (2016)
25. Martínez-Prieto, M.A., Fernández, J.D., Hernández-Illera, A., Gutierrez, C.: Knowledge graph compression for big semantic data. In: Encyclopedia of Big Data Technologies (2022)
26. Mitchell, T.M., et al.: Never-ending learning. In: AAAI (2015)
27. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations and functions. TCS **438**, 74–88 (2012)
28. Nardi, D., Brachman, R.J.: An Introduction to Description Logics. Theory, Implementation, and Applications, The Description Logic Handbook (2003)
29. Navarro, G.: Compact Data Structures - A Practical Approach. Cambridge University Press, Cambridge (2016)
30. Nguyen, V., Bodenreider, O., Sheth, A.: Don't like RDF reification?: Making statements about statements using singleton property. In: WWW (2014)
31. Noy, N., Rector, A., Hayes, P., Welty, C.: Defining N-ary relations on the semantic web. In: W3C (2006)
32. Pelgrin, O.P., Hose, K., Galárraga, L.: TrieDF: Efficient in-memory indexing for metadata-augmented RDF. In: MEPDaW (2021)

33. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (2008)
34. Verborgh, R., et al.: Triple pattern fragments: a low-cost knowledge graph interface for the web. JWS **37–38**, 184–206 (2016)
35. Verborgh, R., Vander Sande, M., Shankar, H., Balakireva, L., Van de Sompel, H.: Devising affordable and functional linked data archives. TCDL. **13**(1), 1–8 (2017)
36. Willerval, A., Diefenbach, D., Bonifati, A.: qEndpoint: A Wikidata SPARQL endpoint on commodity hardware. WWW_Demo (2023)