Undefined 1 (2019) 1–5
IOS Press

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*
E-mail: antonio.hi@gmail.com, migumar2@infor.uva.es
[b] *Vienna University of Economics and Business & Complexity Science Hub Vienna, Austria*
E-mail: jfernand@wu.ac.at
[c] *University of A Coruña, Database Lab, CITIC, Spain*
E-mail: antonio.farina@udc.es

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) [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

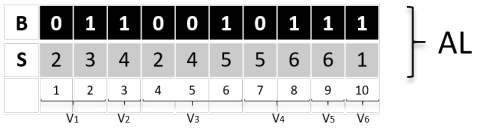

Fig. 1. Example of adjacency list encoding.

"plain sequence" answers $\texttt{access}(S, i)$ 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: $\texttt{neigh}(G, v)$, and to check if two vertices v and u are connected: $\texttt{adj}(G, v, u)$, 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++.

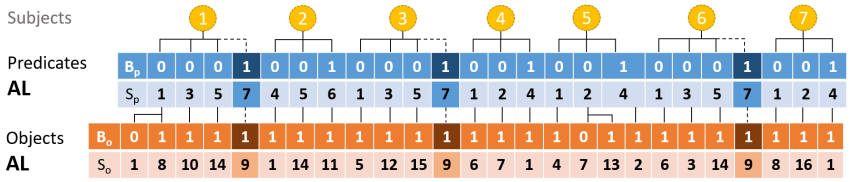

Fig. 2. Organization of *Triples* component in HDT (note that only *predicates* and *objects* adjacency lists are preserved).

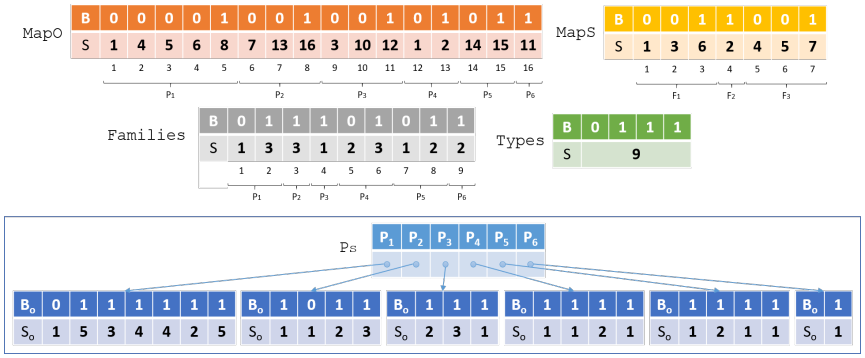

Fig. 3. HDT++ *Triples* component.

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

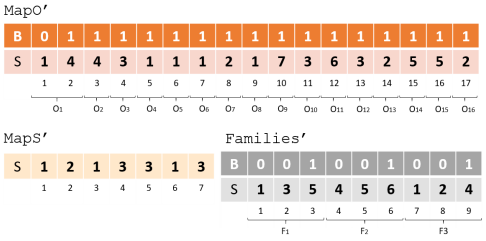

Fig. 4. Transposed structures of `iHDT++`.

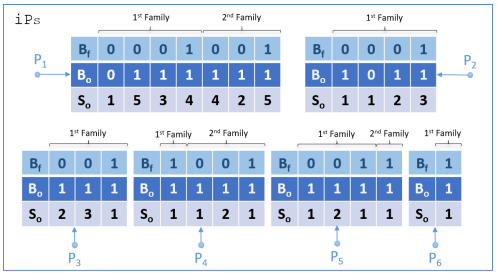

Fig. 5. Indexed `Ps` (`iPs`).

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

```
1 pos_f ← select_1(iPs[pred].B_f, fam − 1);
2 rnk ← rank_1(iPs[pred].B_o, pos_f);
3 pos_s ← 1+ select_1(iPs[pred].B_o, subj + rnk − 1);
4 return iPs.getObjects(pred, pos_s) ;
```

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-

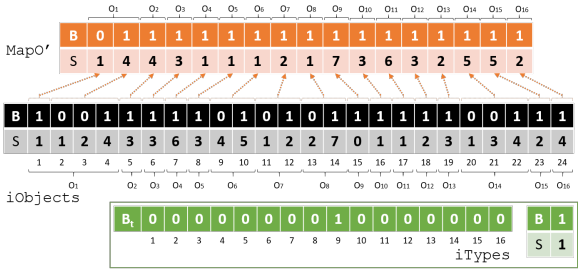

Fig. 6. iObjects configuration.

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)

```
1  family ← MapS′[subj];
2  if pred < |P| then // pred is a regular predicate
3      if adj(Families′, family, pred) ≠ −1 then
4          local_o ← adj(MapO, pred, obj);
5          if local_o ≠ −1 then
6              local_s ← adj(MapS, family, subj);
7              id_f ← adj(Families, pred, family);
8              O ← iPs.getObjSubject(pred, id_f, local_s);
9              if bsearch(O, local_o) ≠ −1 then return true;
10             else return false;
11         end
12         else return false;
13     end
14     else return false;
15 end
16 else // pred is rdf:type
17     if adj(Types, family, obj) ≠ −1 then return true;
18     else return false;
19 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 ← MapS'[subj];
2  if pred < |P| then // pred is a regular predicate
3      if adj(Families', family, pred) ≠ −1 then
4          local_s ←adj(MapS, family, subj);
5          id_f ←adj(Families, pred, family);
6          O ←iPs.getObjSubject(pred, id_f, local_s);
7          res ← ∅;
8          for i ← 1 to |O| do
9              res ← res ∪ neigh(MapO, pred)[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 ← ∅;
2  if pred < |P| then // pred is a regular predicate
3      ptrSubj ← 1;
4      F ← neigh(Families, pred);
5      for i ← 1 to |F| do
6          family ← F[i];
7          S ← neigh(MapS, family);
8          for j ← 1 to |S| do
9              subject ← S[j];
10             O ← iPs.getObjects(predicate, ptrSubject);
11             ptrSubj ← ptrSubj + 1;
12             for k ← 1 to |O| do
13                 object ← neigh(MapO, pred)[O[k]];
14                 res ← res ∪ (subject, object);
15             end
16         end
17     end
18 end
19 else // pred is rdf:type
20     for i ← 1 to |F| do
21         O ← neigh(Types, i) ;
22         if O ≠ ∅ then
23             S ← neigh(MapS, i) ;
24             for i ← 1 to |S| do
25                 for j ← 1 to |O| do
26                     res ← res ∪ (S[i], 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 ← ∅;
2  family ← MapS'[subj];
3  P ← neigh(Families', family);
4  for i ← 1 to |P| do
5  |   O ← pattern_SP?(subj, P[i]);
6  |   for j ← 1 to |O| do
7  |   |   res ← res ∪ (P[i], O[j]);
8  |   end
9  end
10 O ← neigh(Types, family);
11 if O ≠ ∅ then
12 |   for i ← 1 to |O| do
13 |   |   res ← res ∪ (|P|, O[j]);
14 |   end
15 end
16 return res;
```

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

```
1  res ← ∅;
2  P ← neigh(MapO', obj);
3  for i ← 1 to |P| do
4  |   if pattern_SPO(subj, P[i], obj) then
5  |   |   res ← res ∪ 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 ← ∅;
2  if pred < |P| then // pred is a regular predicate
3  |   pos_p ← adj(MapO', obj, pred) ;
4  |   if pos_p ≠ −1 then
5  |   |   pos ← pos_p+select_1(MapO'.B, obj − 1);
6  |   |   Occs ← neigh(iObjects,pos);
7  |   |   for i ← 1 to |Occs| do
8  |   |   |   id_f ← 1+rank_1(iPs[pred].B_f, Occs[i] − 1);
9  |   |   |   family ← neigh(Families, pred)[id_f];
10 |   |   |   local_s ← Occs[i]−
   |   |   |     select_1(iPs[pred].B_f, id_f − 1);
11 |   |   |   res ← res ∪ 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 ← rank_1(iTypes.B_t, obj) ;
18 |   |   F ← neigh(iTypes, object) ;
19 |   |   for i ← 1 to |F| do
20 |   |   |   S ← neigh(MapS, F[i]) ;
21 |   |   |   for j ← 1 to |S| do
22 |   |   |   |   res ← res ∪ S[j];
23 |   |   |   end
24 |   |   end
25 |   end
26 end
27 return res;
```

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

```
1  res ← ∅;
2  P ← neigh(MapO', obj);
3  for i ← 1 to |P| do
4  |   S ← pattern_?PO(P[i], obj);
5  |   for j ← 1 to |S| do
6  |   |   res ← res ∪ (S[j], 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

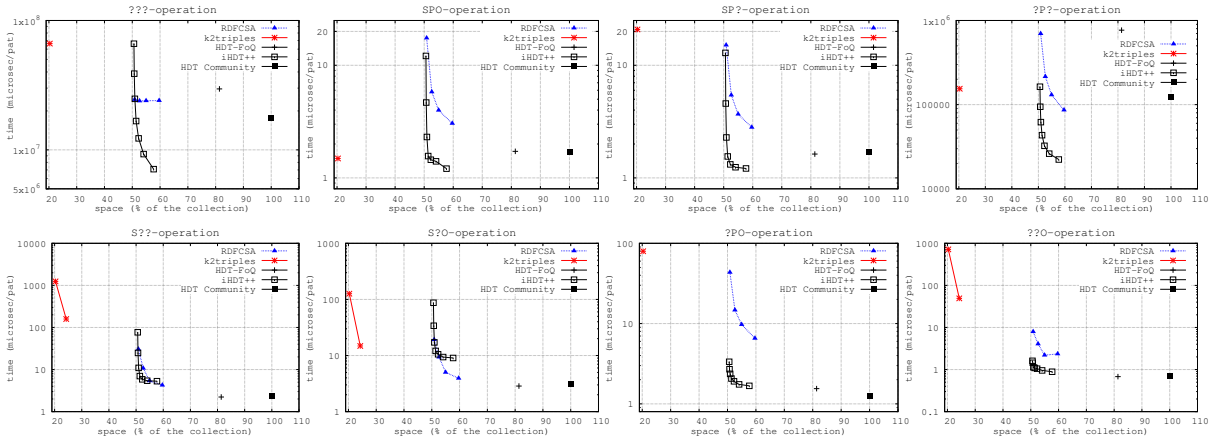

Fig. 7. TPs Resolution: USCENSUS

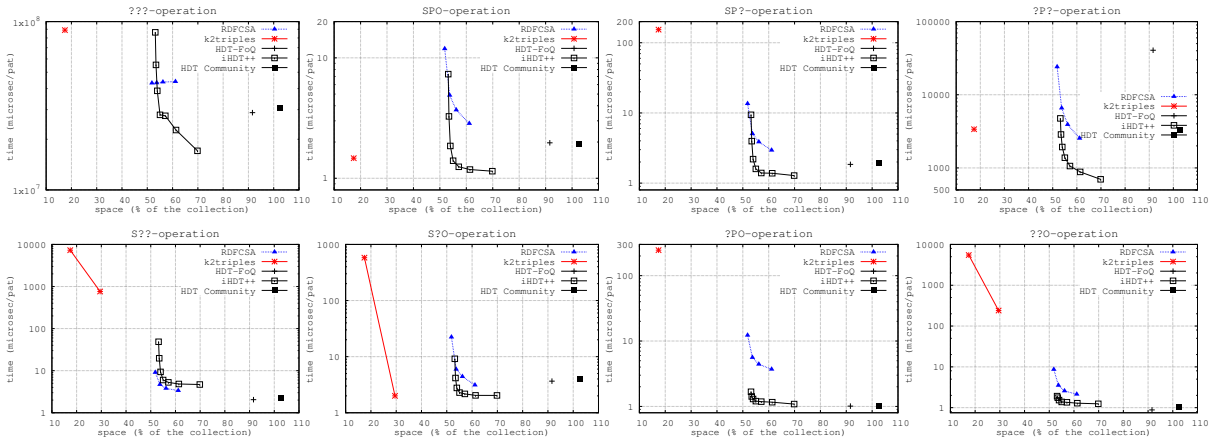

Fig. 8. TPs Resolution: LINKEDGEODATA

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.