HDTQ: Managing RDF Datasets in Compressed Space

Javier D. Fernández[1,2], Miguel A. Martínez-Prieto[3], Axel Polleres[1,2,4], and Julian Reindorf[1]

[1] Vienna University of Economics and Business, Austria
[2] Complexity Science Hub Vienna, Vienna, Austria
[3] Dept. of Computer Science, Universidad de Valladolid, Spain
[4] Stanford University, CA, USA
{javier.fernandez, axel.polleres}@wu.ac.at,
migumar2@infor.uva.es, julian.reindorf@gmail.com

Abstract. HDT (Header-Dictionary-Triples) is a compressed representation of RDF data that supports retrieval features without prior decompression. Yet, RDF datasets often contain additional graph information, such as the origin, version or validity time of a triple. Traditional HDT is not capable of handling this additional parameter(s). This work introduces HDTQ (HDT Quads), an extension of HDT that is able to represent quadruples (or quads) while still being highly compact and queryable. Two HDTQ-based approaches are introduced: Annotated Triples and Annotated Graphs, and their performance is compared to the leading open-source RDF stores on the market. Results show that HDTQ achieves the best compression rates and is a competitive alternative to well-established systems.

1 Introduction

In little more than a decade, the *Resource Description Framework (RDF)* [17] and the *Linked Open Data (LOD) Cloud* [4] have significantly influenced the way people and machines share knowledge on the Web. The steady adoption of Linked Data, together with the support of key open projects (such as schema.org, DBpedia or Wikidata), have promoted RDF as a de-facto standard to represent facts about arbitrary knowledge in the Web, organized around the emerging notion of knowledge graphs. This impressive growth in the use of RDF has irremediably led to increasingly large RDF datasets and consequently to scalability challenges in *Big Semantic Data* management.

RDF is an extremely simple model where a graph is a set of *triples*, a ternary structure (subject, predicate, object), which does not impose any physical storage solution. RDF data management is traditionally based on human-readable serializations, which add unnecessary processing overheads in the context of a large-scale and machine-understandable Web. For instance, the latest DBpedia (2016-10) consists of more than 13 billion triples. Even though transmission speeds and storage capacities grow, such graphs can quickly become cumbersome to share, index and consume.

HDT [7] tackles this issue by proposing a compact, self-indexed serialization of RDF. That is, HDT keeps big datasets compressed for RDF preservation and sharing and –at the same time– provides basic query functionality without prior decompression.

HDT has been widely adopted by the community, (i) used as the main backend of Triple Pattern Fragments (TPF) [18] interface, which alleviates the traditional burden of LOD servers by moving part of the query processing onto clients, (ii) used as a storage backend for large-scale graph data [16], or (iii) as the store behind LOD Laundromat [3], serving a crawl of a very big subset of the LOD Cloud, to name but a few.

One of the main drawbacks of HDT so far is its inability to manage RDF datasets with multiple RDF graphs. HDT considers that all triples belong to the same graph, the *default graph*. However, triples in an RDF dataset can belong to different (named) graphs, hence the extension to the so-called RDF quadruples (subject, predicate, object, graph), or *quads*. The graph (also called context) is used to capture information such as trust, provenance, temporal information and other annotations [19]. Since RDF 1.1 [17] there exist standard RDF syntaxes (such as N-Quads or Trig) for representing RDF named graphs. SPARQL, with its *GRAPH* keyword, allows for querying and managing RDF named graphs, which most common triple stores have implemented. Interestingly, while RDF compression has been an active research topic for a while now, there is neither a compact RDF serialization nor a self-indexed RDF store for quads, to the best of our knowledge.

In this paper we extend HDT to cope with quads and keep its compact and queryable features. HDTQ extends the HDT format with (i) a dictionary that keeps track of all different graph names (or contexts) present in an RDF dataset and assigns a unique integer ID to each of them, and (ii) a compressed bit matrix (named Quad Information) that marks the presence (or absence) of a triple in the graphs. We propose two implementations for this matrix, Annotated Triples (HDT-AT) and Annotated Graphs (HDT-AG), based on indexing the matrix per triple or per graph. Then, we define efficient algorithms for the resolution of quad patterns, i.e. quads where each of the components can be a variable, on top of HDTQ. Our empirical results show that HDTQ keeps compression ratios close to general compression techniques (such as gzip), excels in space w.r.t state-of-the-art stores and remains competitive in quad pattern resolution, respecting the low-cost philosophy of HDT. All in all, HDTQ opens up HDT to a wider range of applications, since GRAPH querying is a key feature in triple stores and SPARQL.

The paper is organized as follows. Section 2 describes the related work and Section 3 provides preliminaries on RDF and HDT. HDTQ, the proposed extension of HDT to handle RDF quads, is presented in Section 4, and evaluated in Section 5. We conclude and devise future work in Section 6.

2 State of the art

RDF datasets with named graphs are traditionally serialized in standard verbose formats such as N-Quads, Trig or JSON-LD [17]. Although they include some compact features (e.g. prefixes or lists), their human-readable focus still adds unnecessary overheads to store and transmit large RDF datasets. Instead, HDTQ proposes a compact, binary serialization that keeps retrieval features.

In turn, all major triple stores supporting SPARQL 1.1 also support named graphs. Regardless of the underneath model (based on a relational schema, implementing a native index or a NOSQL solution), RDF stores often speed up quad-based queries by indexing different combinations of the subject, predicate, object and graph elements in RDF [13]. Virtuoso [5] implements quads in a column-based relational store, with two

full indexes over the RDF quads, with PSOG and POSG order, and 3 projections SP, OP and GS. The well-known Apache Jena TDB[5] stores RDF datasets using 6 B+Trees indexes, namely SPOG, POSG, OSPG, GSPO, GPOS and GOSP. A recent approach, RDF-4X [1] implements a cloud-based NOSQL solution using Apache Accumulo[6]. In this case, quads are organized in a distributed table where 6 indexes, SPOG, POG, OGS, GSP, GP and OS, are built to speed up all triple patterns. Blazegraph[7] (formerly BigData) follows a similar NOSQL approach making use of OGSP, SPOG, GSPO, PGSO, POGS, and SPOG indexes.

Finally, other approaches focus on extending current triple indexes to support quads, such as RQ-RDF-3X [14] or annotating triples with versions, such as v-RDFCSA [6]. HDTQ shares a similar annotation strategy as this latter, extending this concept to general named graphs on top of HDT in order to achieve, to the best of our knowledge, the first compact and queryable serialization of RDF datasets.

3 Preliminaries

This section introduces some terminology and basic concepts of RDF and HDT.

3.1 RDF and SPARQL

An *RDF graph* G is a finite set of triples (subject, predicate, object) from $(I \cup B) \times I \times (I \cup B \cup L)$, where I, B, L denote IRIs, blank nodes and RDF literals, respectively. RDF graphs can be grouped and managed together, conforming an *RDF dataset*, that is, a collection of RDF graphs [17]. Borrowing terminology from [10], an *RDF dataset* is a set $DS = \{G, (g_1, G_1), \ldots, (g_n, G_n)\}$ consisting of a (non-named) default graph G and *named graphs* s.t. $g_i \in I$ are graph names. Figure 1 represents a dataset DS consisting of two named graphs (*aka* subgraphs), `graphWU` and `graphTU`, coming from different sources (e.g. from two universities). Note that terms[8] (i.e. subjects, predicates and objects) and triples can belong to different named graphs. For instance, the triple $(Vienna, locatedIn, Europe)$ is shared among the two subgraphs.

An *RDF quad* can be seen as an extension of a triple with the graph name (*aka* context). Formally, an RDF quad q from an RDF dataset DS, is a quadruple (subject, predicate, object, g_i) from $(I \cup B) \times I \times (I \cup B \cup L) \times I$. Note that the graph name g_i can be used in other triples or quads to provide further meta-knowledge, e.g. the subgraph provenance. We also note that quads and datasets (with named graphs) are in principle interchangeable in terms of expressiveness, i.e. one can be represented by the other.

RDF graphs and datasets are traditionally queried using the well-known SPARQL [10] query language. SPARQL is based on graph pattern matching, where the core component is the concept of a triple pattern, i.e. a triple where each subject, predicate and object are RDF terms or SPARQL variables. Formally, assuming a set V of variables, disjoint from the aforementioned I, B and L, a triple pattern tp is a tuple from $(I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$. In turn, SPARQL defines ways of specifying and querying by graph names (or the default graph), using the GRAPH keyword. To capture this, following the same convention as the triple pattern, we define a quad

[5] `http://jena.apache.org/documentation/tdb/index.html`
[6] `https://accumulo.apache.org/`
[7] `https://www.blazegraph.com/`
[8] All terms are IRIs whose prefix, `http://example.org/`, has been omitted for simplicity.

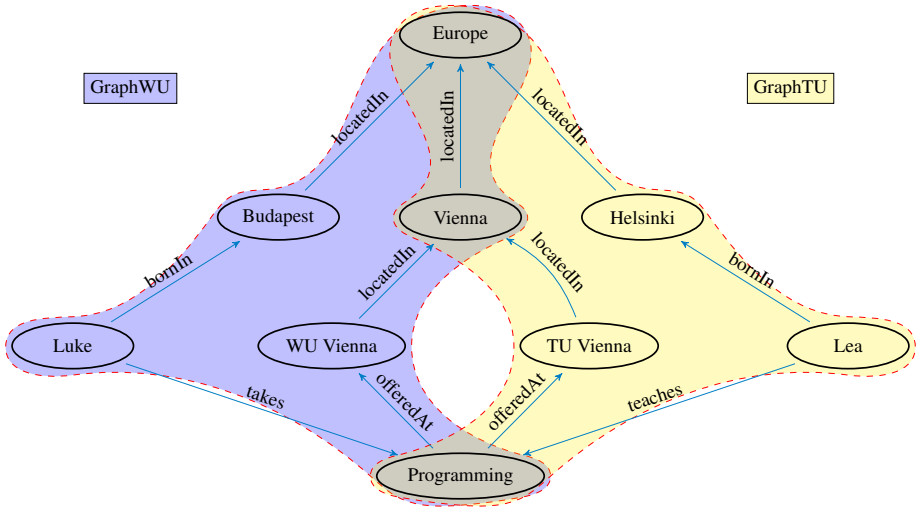


Fig. 1: An RDF dataset DS consisting of two graphs, `GraphWU` and `GraphTU`.

pattern qp as an extension of a triple pattern where also the graph name can be provided or may be a variable to be matched. That is, a quad pattern qp is a pair $tp \times (I \cup V)$ where the last component denotes the graph of the pattern (an IRI or variable).

3.2 HDT

HDT [7] is a compressed serialization format for single RDF graphs, which also allows for triple pattern retrieval over the compressed data. HDT encodes an RDF graph G into three components: the *Header* holds metadata (provenance, signatures, etc.) and relevant information for parsing; the *Dictionary* provides a catalog of all RDF terms in G and maps each of them to a unique identifier; and the *Triple* component encodes the structure of the graph after the ID replacement. Figure 2 shows the HDT dictionary and triples for all RDF triples in Figure 1, i.e. disregarding the name graphs.

HDT Dictionary. The HDT dictionary of a graph G, denoted as D_G, organizes all terms in four sections, as shown in Figure 2 (a): SO includes terms occurring both as subject and object, mapped to the ID-range `[1,|SO|]`. Sections S and O comprise terms that only appear as subjects or objects, respectively. In order to optimize the range of IDs, they are both mapped from `|SO|+1`, ranging up to `|SO|+|S|` and `|SO|+|O|`, respectively. Finally, section P stores all predicates, mapped to `[1,|P|]`. Note that (i) no ambiguity is possible once we know the role played by the term, and (ii) the HDT dictionary provides fast lookup conversions between IDs and terms.

HDT Triples. The Triples component of a graph G, denoted as T_G, encodes the *structure* of the RDF graph after ID replacement. Logically speaking, T organizes all triples into a forest of trees, one per different subject, as shown in Figure 2 (b): subjects are the roots of the trees, where the middle level comprises the ordered list of predicates associated with each subject, and the leaves list the objects related to each (subject,

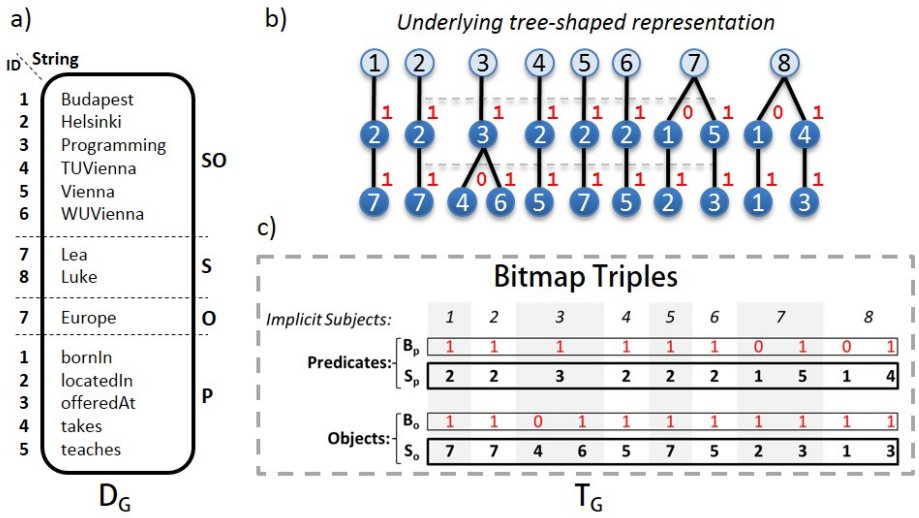


Fig. 2: HDT Dictionary and Triples for a graph G (merging all triples of Fig. 1).

predicate) pair. This *underlying representation* is practically encoded with the so-called *BitmapTriples* approach [7], shown in Figure 2 (c). It comprises *two sequences*: `Sp` and `So`, concatenating all predicate IDs in the middle level and all object IDs in the leaves, respectively; and *two bitsequences*: `Bp` and `Bo`, which are aligned with `Sp` and `So` respectively, using a 1-bit to mark the end of each list. Bitsequences are then indexed to locate the 1-bits efficiently. These enhanced bitsequences are usually called *bitmaps*. HDT uses the Bitmap375 [11] technique that takes 37.5% extra space on top of the original bitsequence size.

Triple Pattern resolution with HDT. As shown, *BitmapTriples* is organized by subject, conforming a SPO index that can be used to efficiently resolve subject-bounded triple pattern queries [10] (i.e. triples where the subject is provided and the predicate and object may be a variable) as well as listing all triples. HDT-Focused on Querying (HDT-FoQ) [16] extends HDT with two additional indexes (PSO and OPS) to speed up the resolution of all triple patterns.

4 HDTQ: Adding Graph Information to HDT

This section introduces HDTQ, an extension of HDT that involves managing RDF quads. We consider hereinafter that the original source is an RDF dataset as defined in Section 3, potentially consisting of several named graphs. For simplicity, we assume that graphs have no blank nodes in common, otherwise a re-labeling step would be possible as pre-processing.

4.1 Extending the HDT Components

HDT was originally designed as a flexible format that can be easily extended, e.g. to include different dictionary and triples components or to support domain-specific ap-

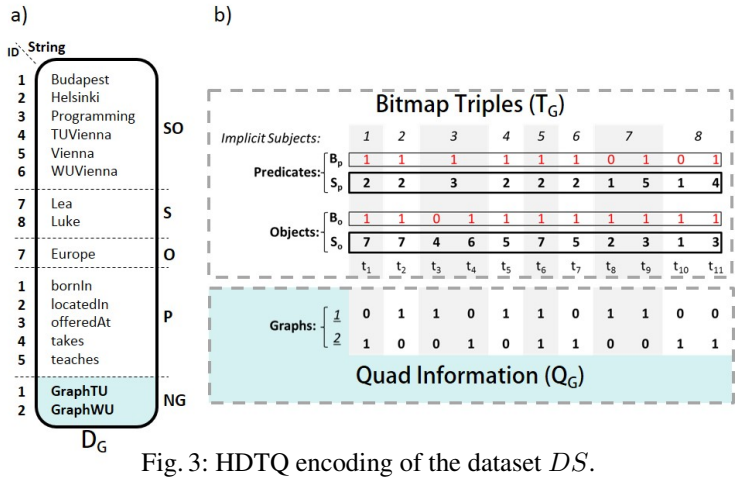


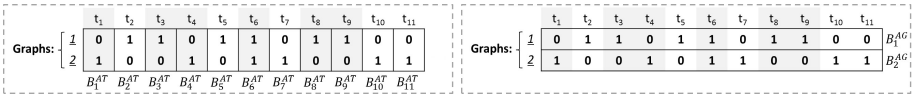
Fig. 3: HDTQ encoding of the dataset DS.

plications. In the following, we detail HDTQ and the main design decisions to extend HDT to cope with quads. Figure 3 shows the final HDTQ encoding for the dataset DS in Figure 1. We omit the header information, as the HDTQ extension only adds implementation-specific metadata to parse the components.

Dictionary. In HDTQ, the previous four-section dictionary is extended by a fifth section to store all different graph names. The IDs of the graphs are then used to annotate the presence of the triples in each graph, further explained below. Figure 3 (a) shows the new HDTQ dictionary encoding for the dataset DS. Compared to the dictionary shown in Figure 2, i.e. the HDT conversion of all triples disregarding the named graphs, two comments are in order:

- The terms of all graphs are merged together in the traditional four dictionary sections, SO, S, O, P, as explained in Section 3. This decision can potentially increase the range of IDs w.r.t an individual mapping per graph, but it keeps the philosophy of storing terms once, when possible.
- The graph names are organized in an independent graph section, NG (named graphs), mapped from 1 to n_g, being n_g the number of graphs. Note that these terms might also play a different role in the dataset, and can then appear duplicated in SO, S, O or P. However, no ambiguity is possible with the IDs once we know the role of the term we are searching for. In turn, the storage overhead of the potential duplication is limited as we assume that the number of graphs is much less than the number of unique subjects and objects. An optimization for extreme corner cases is devoted to future work.

Triples. HDTQ respects the original *BitmapTriples* encoding and extends it with an additional *Quad Information* (Q) component, shown in Figure 3 (b). Q represents a boolean matrix that includes (for every *triple* - *graph* combination) the information on whether a specific triple appears in a specific graph. Formally, having a triple-ID t_j (where $j \in \{1..m\}$, being m the total number of triples in the dataset DS), and a graph-ID k (where $k \in \{1..n_g\}$), the new Q component defines a boolean function $graph(t_j, k) = \{0, 1\}$, where 1 denotes that t_j appears in the graph k, or 0 otherwise.



(a) Annotated Triples (b) Annotated Graphs

Fig. 4: Annotated Triples and Annotated Graphs variants for the RDF dataset DS.

4.2 Quad Indexes: Graph and Triples Annotators

HDTQ proposes two approaches to realize the Q matrix, namely Annotated Triples (HDT-AT) and Annotated Graphs (HDT-AG). They both rely on bitmaps, traditionally used in HDT (see Section 3).

Annotated Triples. Using the Annotated Triples approach, a bitmap is assigned to each triple, marking the graphs in which the corresponding triple is present. A dataset containing m triples in n different graphs has $\{B_1^{AT}, \cdots, B_m^{AT}\}$ bitmaps each of size n. Thus, if $B_j^{AT}[i] = 1$, it means that the triple t_j is present in the i^{th} graph, being $B_j^{AT}[i] = 0$ otherwise. This can be seen in Figure 4 (a), where 11 bitmaps (one per triple) are created, each of them of two positions, corresponding to the two graphs. In this example, the bitmap for the first triple holds $\{0, 1\}$, meaning that the first triple, (1,2,7), only appears in the second graph, which is `graphWU`.

Intuitively, Annotated Triples favors quad patterns having the graph component as a variable, like `SPO?`, as only a single bitmap needs to be browsed. On the other hand, if the graph is given, like in the pattern `???G`, all of the bitmaps need to be browsed.

Annotated Graphs. This approach is orthogonal to Annotated Triples: a bitmap is assigned to each graph, marking the triples present in the corresponding graph. Thus, a dataset containing m triples in n different graphs has $\{B_1^{AG}, \cdots, B_n^{AG}\}$ bitmaps each of size m. Thus, if $B_j^{AG}[i] = 1$, it means that the triple t_i is present in the j^{th} graph, being $B_j^{AG}[i] = 0$ otherwise. This can be seen in Figure 4 (b), including 2 bitmaps, each of size 11. For instance, the bitmap for the first graph, `graphTU`, holds $\{0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0\}$ meaning that it consists of the triples $\{t_2, t_3, t_5, t_6, t_8, t_9\}$, which can be found in the respective positions in BitmapTriples.

Compared to Annotated Triples, Annotated Graphs favors quad patterns in which the graph is given, like `???G`, as only a single bitmap (the bitmap of the given graph G) needs to be browsed. On the other hand it penalizes patterns with graph variables, as all bitmaps need to be browsed to answer the query.

Finally note that, both in HDT-AT and HDT-AG, depending on the data distribution, the bitmaps can be long and sparse. However, in practice, HDT-AT and HDT-AG can be implemented with compressed bitmaps [15] to minimize the size of the bitsequences.

4.3 Search Operations

The resolution of quad patterns in HDTQ builds on top of two operations inherently provided by the *BitmapTriples* component (BT):

– **BT.getNextSolution(quad, startPosition)**. Given a $quad$ pattern, BT removes the last graph term and resolves the triple pattern, outputting a pair $(triple, posTriple)$ corresponding to the next triple solution and its position in BT. The search starts

Algorithm 1: SEARCHQUADS - *quad patterns with unbounded graphs*

```
   Input: BitmapTriples BT, Quad Information Q, quad pattern q
   Output: The quads matching the given pattern
1  result ← (); graph ← 0
2  (triple, posTriple) ← BT.getNextSolution(q, 0)
3  while posTriple ≠ null do
4      graph ← Q.nextGraph(posTriple, graph + 1)
5      if graph ≠ null then
6          result.append(triple, graph)
7      else
8          (triple, posTriple) ← BT.getNextSolution(q, posTriple)
9          graph ← 0
10 return result
```

at the $startPosition$ provided, in BT. For instance, in our example in Figure 3, with a pattern $quad =$ `7???`, an operation `BT.getNextSolution(quad,8)` will jump the first 8 triples in BT,$\{t_1, \cdots, t_8\}$, hence the only solution is the pair $((7, 5, 3), 9)$ or, in other words, t_9.

– **BT.getSolutionPositions(quad)**. This operation finds the set of triple positions where solution candidates appear. In subject-bounded queries, these positions are actually a consecutive range $\{t_x, .., t_y\}$ of BT. Otherwise, in queries such as `?P?G`, `??OG` and `?POG`, the positions are spread across BT. For instance, t_2 and t_5 are solutions for $quad =$`?2?1`, but t_3 and t_4 do not match the pattern.

Note that we assume that the HDT-FoQ [16] indexes (PSO and OPS) are created, hence BT can provide these operations for all patterns. In the following, we detail the resolution depending on whether the graph term is given or it remains unbounded.

Quad Pattern Queries with Unbounded Graph. Algorithm 1 shows the resolution of quad patterns in which the graph term is not given, i.e. `????`, `S???`, `?P??`, `??O?`, `SP??`, `S?O?`, `?PO?` and `SPO?`. It is mainly based on iterating through the solutions of the traditional HDT and, for each triple solution, returning all the graphs associated to it. Thus, the algorithm starts by getting the first solution in BT (Line 2), using the aforementioned operation *getNextSolution*. While the end of BT is not reached (Line 3), we get the next graph associated with the current triple (Line 4), or $null$ if it does not appear in any further graph. This is provided by the operation *nextGraph* of Q, explained below. If there is a graph associated with the triple (Line 5), both are appended to the results (Line 6). Otherwise, we look for the next triple solution (Line 8).

The auxiliary *nextGraph* operation of Q returns the next graph in which a given triple appears, or $null$ if the end is reached. Algorithm 2 shows this operation for HDT-AT. First, the bitmap corresponding to the given triple is retrieved from Q (Line 1). Then, within this bitmap, the location of the next 1 starting with the provided graph ID is retrieved (or null if the end is reached) and returned (Line 2). This latter is natively provided by the bitmap indexes.

Algorithm 3 shows the same process for HDT-AG. In this case, a bitmap is associated with each graph. Thus, we iterate on graphs and access one bitmap after the other (Line 1-7). The process ends when a 1-bit is found (Line 3), returning the graph (Line 4), or the maximum number of graphs is reached (Line 7), returning null (Line 8).

Algorithm 2: NEXTGRAPH - AT

```
Input: Quad Information Q, int posTriple, int graph
Output: The position of the next graph
1 bitmap ← Q[posTriple]
2 return bitmap.getNext1(graph)
```

Algorithm 3: NEXTGRAPH - AG

```
Input: Quad Information Q, int posTriple, int graph
Output: The position of the next graph
1 do
2     bitmap ← Q[graph]
3     if bitmap[posTriple] = 1 then
4         return graph
5     else
6         graph ← graph + 1
7 while graph ≤ Q.size()
8 return null
```

Algorithm 4: SEARCHQUADSG - *quad patterns with bounded graphs*

```
Input: BitmapTriples BT, Quad Information Q, quad pattern q
Output: The quads matching the given pattern
1  graph ← getGraph(q); result ← ()
2  sol[] ← BT.getSolutionPositions(q)
3  while !sol.isEmpty() do
4      posTripleCandidateBT ← sol.pop()
5      posTripleCandidateQT ← Q.nextTriple(posTripleCandidateBT, graph)
6      if posTripleCandidateBT = posTripleCandidateQT then
7          (triple, posTriple) ← BT.getNextSolution(q, posTripleCandidateBT − 1)
8          result.append(triple, graph)
9      else
10         sol.removeLessThan(posTripleCandidateQT)
11 return result
```

Quad Pattern Queries with Bounded Graph. Algorithm 4 resolves all quad patterns where the graph is provided. To do so, the graph ID is first retrieved from the quad pattern (Line 1). The aforementioned *getSolutionPositions* operation of BT finds the triple positions in which the solutions can appear (Line 2). Then, we iterate on this set of candidate positions until it is empty (Line 3). For each *posTripleCandidateBT* extracted from the set (Line 4), we check if this position is associated with the given graph (Line 5), using the operation *nextTriple* of the Q structure. This operation, omitted for the sake of concision as it is analogous to *nextGraph* (see Algorithms 2 and 3), starts from *posTripleCandidateBT* and returns the next triple position (*posTripleCandidateQT*) that is associated to the given graph. Thus, if this position is exactly the current candidate position (Line 6), the actual triple is obtained for that position (Line 7), and appended to the final resultset (Line 8). Otherwise, the candidate position was not a valid solution (it was not related to the graph), and we can remove, from the set of candidate solutions, all positions lesser than *posTripleCandidateQT* (Line 10), given that none of them are associated to the given graph.

5 Evaluation

We evaluate the performance of HDTQ in terms of space and efficiency on quad pattern resolution. The HDTQ prototype[9], built on top of the existing HDT-Java library[10], im-

[9] HDTQ library: `https://github.com/JulianRei/hdtq-java`

[10] HDT-Java library: `https://github.com/rdfhdt/hdt-java`

		Subjects	Predicates	Objects	Graphs	Triples	Quads
BEAR	A	74,908,887	41,209	64,215,355	58	378,476,570	2,071,287,964
	B day	100	1,725	69,650	89	82,401	3,460,896
	B hour	100	1,744	148,866	1,299	167,281	51,632,164
LUBM500	G1	10,847,183	17	8,072,358	1	66,731,200	66,731,200
	...	...	...	...	...	...	...
	G9998	10,847,183	17	8,072,358	9998	66,731,200	68,823,803
LDBC		668,711	16	2,743,645	190,961	5,000,197	5,000,197
LIDDI		392,344	23	981,928	392,340	1,952,822	2,051,959

Table 1: Statistical dataset description.

plements both HDT-AG and HDT-AT approaches using existing compressed bitmaps (called Roaring Bitmaps [15]), which are optimal for sparse bitsequences.

Datasets. Experiments are carried out on heterogeneous RDF datasets[11], described in Table 1. BEAR-A [8], a benchmark for RDF archives, includes 58 weekly crawls of a set of domains in the LOD cloud. Each of the snapshots is considered to be a graph, resulting in a dataset of 58 graphs. BEAR-A is relatively dynamic as 31% of the data change between two versions, resulting in more than 2 billion quads. BEAR-B day and BEAR-B hour [9] extend BEAR-A to consider more dynamic information. They crawl the 100 most volatile resources in DBpedia Live over the course of three months, and consider a new version by day or hour, respectively. Each of the versions is seen as a graph, summing up 89 and 1,299 graphs, respectively. Given that most of the triples remain unchanged, most triples appear in multiple graphs.

The well-known LUBM data generator [12] is also considered as a way to generate several RDF datasets with increasing number of graphs: 1, 10, 20,..., 100, 1,000, 2,000,..., 9,000 and 9,998. We first set up the generator to produce synthetic data describing 500 universities (LUBM500), which results in 66m triples. Given that the generator produces 9,998 files, $\{f_1, f_2 \cdots, f_{9998}\}$, we first consider an RDF dataset with a graph per file, $\{g_1, g_2 \cdots, g_{9998}\}$, named as LUBM500-G9998. Then, to generate a dataset with an arbitrary number of graphs n ($n <= 9998$), each file f_j was merged to a graph g_i, where $i = j \mod n$. For simplicity, Table 1 only shows LUBM500-G1 and LUBM500-G9998. In general, triples are rarely repeated across graphs.

LDBC[12] regards the Semantic Publishing Benchmark (SPB), which considers diverse media content. We use the default SPB 2.0 generator that generates more than 190k named graphs, where each triple appears only in one graph.

Finally, the Linked Drug-Drug Interactions (LIDDI) dataset [2] integrates multiple data collections, including provenance information. This results in an RDF dataset with an extremely large number of graphs, 392,340, as shown in Table 1.

Triple Stores. We compare HDTQ against two well-known triple stores in the state of the art, Apache Jena TDB 2.10 store[13] and Virtuoso 7.1 Open Source[14]. Following

[11] Datasets, queries, scripts, raw results and additional material is available at: `https://aic.ai.wu.ac.at/qadlod/hdtq/eswc2018/`

[12] LDBC: `http://ldbcouncil.org/developer/spb`

[13] Apache Jena: `http://jena.apache.org`

[14] Virtuoso: `http://virtuoso.openlinksw.com`

		Size (GB)	gzip	HDT-AG	HDT-AT	Jena	Virtuoso	Virtuoso+
BEAR	A	396.9	5.8%	**2.3%**	2.8%	96.8%	NA	NA
	B day	0.6	4.8%	**0.7%**	0.8%	97.7%	13.7%	33.7%
	B hour	9.7	4.8%	0.3%	**0.1%**	96.4%	4.3%	25.6%
LUBM500	G1	11.4	3.0%	**6.6%**	17.0%	118.8%	17.2%	21.0%
	G9998	11.6	3.0%	**6.6%**	16.7%	120.1%	17.5%	27.5%
LDBC		0.9	9.7%	**15.9%**	25.1%	126.3%	71.2%	80.8%
LIDDI		0.7	3.7%	**11.8%**	15.6%	78.1%	49.9%	53.4%

Table 2: Space requirements of different systems.

Virtuoso instructions, we also consider a variant, named as Virtuoso+, which includes an additional index (GPOS) that may speed up quad patterns where the subject is not given. Experiments were performed in a -commodity server- (Intel Xeon E5-2650v2 @ 2.6 GHz, 16 cores, RAM 180 GB, Debian 7.9). Reported (elapsed) times are the average of three independent executions. Transactions are disabled in all systems.

5.1 Space Requirements and Indexing Time

Table 2 lists the space requirements of the uncompressed RDF datasets in N-Quads notation (column "Size"), in gigabytes, the respective gzipped datasets (column "gzip") and the systems under review, as the ratio between the size for the required space and the uncompressed size. The numbers reported for HDT-AG and HDT-AT include the size of HDTQ and the additional HDT indexes (created with HDT-FoQ [16]) needed to resolve all quad patterns[15]. Note that Virtuoso was not capable of importing the BEAR-A dataset due to a persistent error when inserting large quad data. In fact, a similar bug in the current Java implementation of Roaring Bitmaps [15] made us use Bitmap375 [11] for this particular scenario, as the HDTQ prototype supports both implementations interchangeably, being transparent to users consuming/querying HDTQ.

As expected, gzip achieves large space savings that outperform the space needs of the RDF stores. However, HDTQ improves upon gzip in all BEAR datasets, where a large amount of (verbose) triples are shared across graphs. In this scenario, HDTQ is able to mitigate the repetitions thanks to the dictionary and Quad Information structures.

HDTQ outperforms Jena and Virtuoso in all datasets, being particularly noticeable in BEAR datasets (1-2 levels of magnitude smaller), with a limited number of graphs (58 to 1,299) and many shared triples across graphs. HDTQ gains are still noticeable in LDBC (3 to 5 times smaller than Virtuoso and Jena) with a very large number of graphs (190k) and no shared triples. Similar results are obtained in LIDDI, with 392k graphs.

In turn, HDT-AG reports better compression ratios than HDT-AT (except for a small difference in BEAR-B hour). The main reason lies in the compressed implementation of the bitmaps [15] that exploits consecutive runs of 0's or 1's to achieve further compression. In most of the cases, longer runs are produced when annotating the triples per graph (HDT-AG), than viceversa (HDT-AT). In fact, HDT-AT largely outperforms all systems except for LUBM, where the compression is only slightly better than Virtuoso (without additional indexes) given that triples are mostly present in one single graph, hence HDT-AT needs to pay the price of storing multiple bitmaps (one per triple), each of them representing just a single value. In LDBC, with a similar scenario, the HDT-

[15] The HDTQ website additionally includes the size of each structure of HDT and HDTQ.

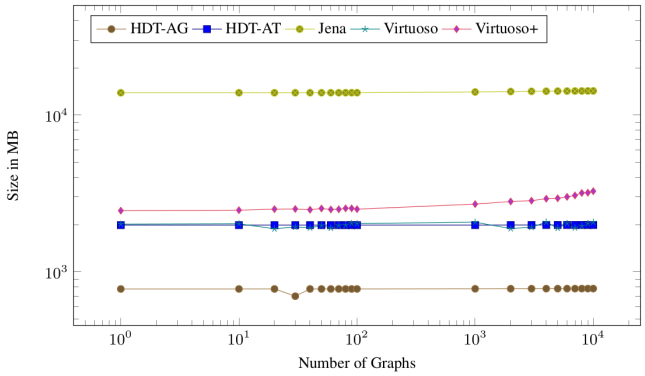


Fig. 5: Space requirements for the LUBM500 datasets with increasing number of graphs.

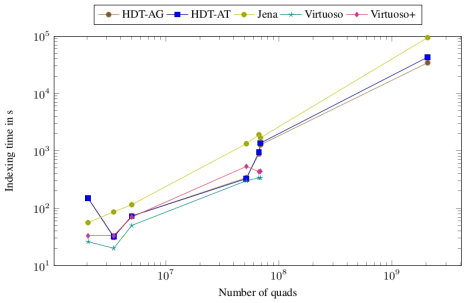


Fig. 6: Indexing times (in s) of the RDF datasets.

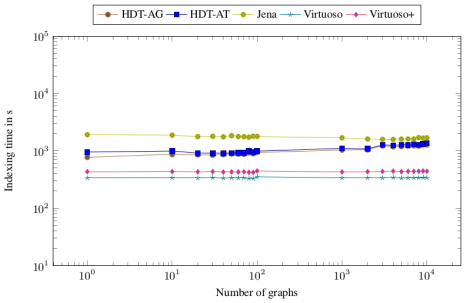


Fig. 7: Indexing times (in s) of LUBM500 datasets.

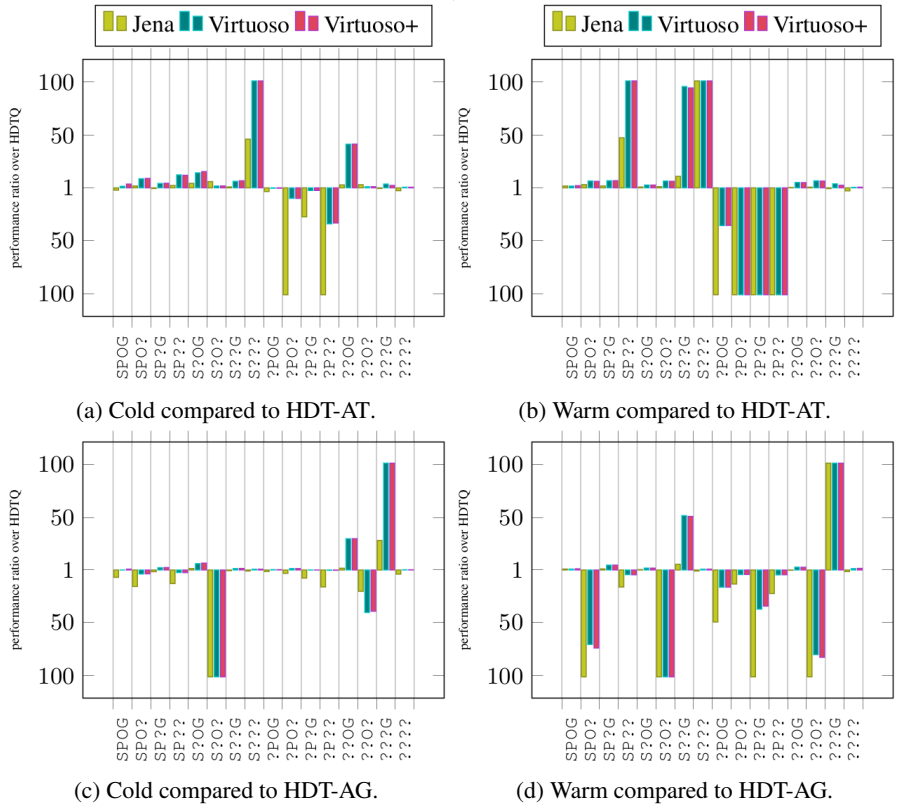
AT overhead is compensated by the overall compression of the dictionary and triples components, and HDTQ still excels in space.

Figure 5 shows the space requirements (in MB, and log log scale) of LUBM500 at increasing number of graphs (1, 10, 20,..., 100, 1,000, 2,000,..., 9,000 and 9,998), where few triples are shared across graphs. All systems, including HDTQ, demand close to constant size regardless of the number of graphs. The exception is Virtuoso+, as it pays the price of the additional index GPOS.

Figure 6 represents (in log log scale) the indexing times of the RDF datasets in Table 2, sorted by the number of quads of each dataset. For HDT-AG and HDT-AT, this includes the creation of all components (standard HDT and HDT-FoQ, and the novel graph dictionary and Quad Information structures). Virtuoso is the fastest system regarding creation time in all cases, except for the failed BEAR-A. In mostly all cases, Jena doubles the time required by HDTQ. As expected, the time in all systems shows a linear growth with an increasing number of quads. In turn, Figure 7 focuses on LUBM500 at increasing number of graphs. In general, Jena and Virtuoso perfectly scale in this scenario, whereas HDTQ pays the overhead of the creation of increasing large bitmaps in HDT-AG and HDT-AT. Nonetheless, the overhead is limited and the creation can be seen as a one-off cost by the publisher.

5.2 Performance for Quad Pattern Resolution

To test the performance of the systems, we select, for each dataset, 100 random queries for each combination of quad patterns (except for the pattern `????` and those patterns



(a) Cold compared to HDT-AT.

(b) Warm compared to HDT-AT.

(c) Cold compared to HDT-AG.

(d) Warm compared to HDT-AG.

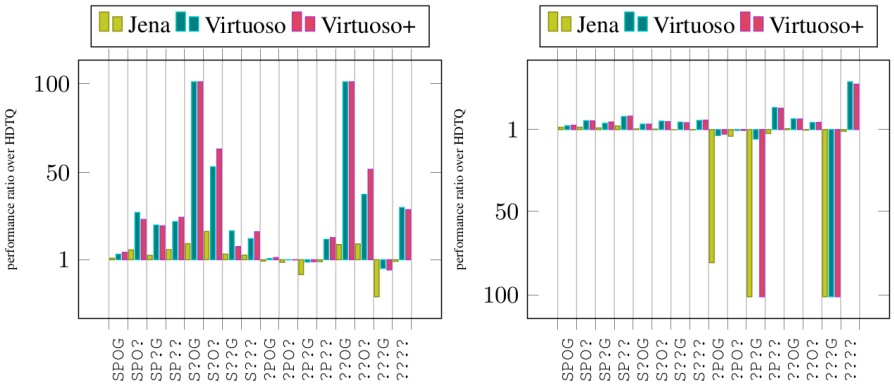
Fig. 8: BEAR-B day quad pattern resolution speed.

such as `?P??` where the data distribution prevents from having 100 different queries). Each query is then executed in two scenarios: cold, where cache was first cleared, and warm, which considers a warmup by first querying `????` and taking 100 results.

Figures 8 and 9 show the averaged resolution times of the selected queries for two exemplary datasets, BEAR-B day with limited number of named graphs and many repeated triples across graphs, and LIDDI, with opposite characteristics. In these figures, a k number above the x-axis means that HDTQ is k times faster than the compared system. A k number below shows that the system is k times faster than HDTQ.

Results for HDT-AT in BEAR-B day, Figures 8a and 8b, show that HDT-AT, while taking 1-2 order of magnitude less in space, excels in subject-based queries (in special `S???` and `S??G`), in cold and warm scenarios. In contrast, it is penalized in predicate-based queries (such as `?P??` and `?P?G`). This result is in line with previous HDT-FoQ [16] remarks, which shows that adding the quad information as a triple annotation in HDT-AT keeps the retrieval features of HDT.

HDT-AG reports less promising numbers in Figures 8c and 8d. As expected, listing the triples by graphs (`???G`) is extremely efficient. However, HDT-AG design penalizes most operations, in particular those with unbounded graphs, as results must be first



(a) Cold compared to HDT-AT

(b) Warm compared to HDT-AT

Fig. 9: LIDDI quad pattern resolution speed.

found, and then all graph bitmaps have to be accessed to check the presence of the triple. In contrast HDT-AT allows for quickly jumping to the next solution (the next 1-bit annotation) of the triple.

Figure 9 shows the performance of HDT-AT in LIDDI, with almost 400k different graphs. In such extreme case, HDT-AT still remains competitive for most queries, although it still pays the price of HDT-FoQ in some predicate-based queries and graph listing (`???G`). In this case, although HDT-AG achieves the best compression, it was unable to compete, being 1-2 orders of magnitude slower in most cases.

These results show that, in a general case, HDT-AT should be preferred over HDT-AG as it provides the best space/performance tradeoff. Nonetheless, HDT-AG remains a candidate solution to achieve greater space savings with reasonably performance if the number of graphs is limited. Given that HDT-AG excels when listing graphs, further inspection of a combined approach (AG-AT) is devoted to future work.

6 Conclusions and Future Work

This work presents HDTQ as an extension of HDT, a compact and queryable serialization of RDF, to support RDF datasets including named graphs (quads). HDTQ considers a new dictionary to uniquely store all different named graphs, and a new Quad Information component to annotate the presence of the triples in each graph of the RDF dataset. Two realizations of this component are proposed, HDT-AG and HDT-AT, and space/performance tradeoffs are evaluated against different datasets and state-of-the-art stores. Results show that HDTQ keeps the same HDT features, positioned itself as a highly compact serialization for RDF quads that remains competitive in quad pattern resolution. Our ongoing work focuses on inspecting an hybrid AT-AG strategy for the quad information and supporting full SPARQL 1.1. on top of HDTQ. To do so, we plan to use HDTQ as a backend store within existing frameworks, such as Jena.

Acknowledgements

Supported by the EU's Horizon 2020 research and innovation programme: grant 731601 (SPECIAL), the Austrian Research Promotion Agency's (FFG) program "ICT of the

Future": grant 861213 (CitySpin), and MINECO-AEI/FEDER-UE ETOME-RDFD3: TIN2015-69951-R and TIN2016-78011-C4-1-R; Axel Polleres is supported under the Distinguished Visiting Austrian Chair Professors program hosted by The Europe Center of Stanford University. Thanks to Tobias Kuhn for the pointer to the LIDDI dataset.

References

1. S. Abbassi and R. Faiz. RDF-4X: A Scalable Solution for RDF Quads Store in the Cloud. In *Proc. of MEDES*, pages 231–236, 2016.
2. J. M. Banda, T. Kuhn, N. H. Shah, and M. Dumontier. Provenance-Centered Dataset of Drug-Drug Interactions. In *Proc. of ISWC*, pages 293–300, 2015.
3. W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach. LOD Laundromat: A Uniform Way of Publishing other People's Dirty Data. In *Proc. of ISWC*, pages 213–228, 2014.
4. C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227, 2009.
5. P. Boncz, O. Erling, and M.-D. Pham. Advances in large-scale rdf data management. In *Linked Open Data–Creating Knowledge Out of Interlinked Data*, pages 21–44. 2014.
6. A. Cerdeira-Pena, A. Farina, J. D. Fernández, and M. A. Martínez-Prieto. Self-Indexing RDF Archives. In *Proc. of DCC*, pages 526–535, 2016.
7. J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange (HDT). *JWS*, 19:22–41, 2013.
8. J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth. Evaluating Query and Storage Strategies for RDF Archives. In *Proc. of SEMANTiCS*, pages 41–48, 2016.
9. J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth. Evaluating Query and Storage Strategies for RDF Archives. Semantic Web Journal. Under review. *SWJ*, 2017. Available at http://www.semantic-web-journal.net/content/evaluating-query-and-storage-strategies-rdf-archives.
10. S. H. Garlik, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language, W3C Recommendation, 2013.
11. R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical Implementation of Rank and Select Queries. In *Proc. of WEA*, pages 27–38, 2005.
12. Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *JWS*, 3(2):158 – 182, 2005.
13. A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *Proc. of LA-WEB*, pages 10–pp, 2005.
14. J. Leeka and S. Bedathur. RQ-RDF-3X: going beyond triplestores. In *Proc. of ICDEW*, pages 263–268, 2014.
15. D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai. Roaring Bitmaps: Implementation of an Optimized Software Library. *arXiv preprint arXiv:1709.07821*, 2017.
16. M. A. Martínez-Prieto, M. A. Gallego, and J. D. Fernández. Exchange and Consumption of Huge RDF Data. In *Proc. of ESWC*, pages 437–452, 2012.
17. G. Schreiber and Y. Raimond. *RDF 1.1 Primer*. W3C Working Group Note, 2014.
18. R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. *JWS*, 37–38:184–206, 2016.
19. A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia. A General Framework for Representing, Reasoning and Querying with Annotated Semantic Web Data. *JWS*, 11:72–95, 2012.