HDTCat: Let's Make HDT Generation Scale

Dennis Diefenbach[1,2](✉) and José M. Giménez-García[2]

[1] The QA Company SAS, Saint-Étienne, France
dennis.diefenbach@qanswer.eu
[2] Univ Lyon, UJM-Saint-Étienne, CNRS, Laboratoire Hubert Curien, UMR 5516, 42023 Saint-Étienne, France
jose.gimenez.garcia@univ-st-etienne.fr

Abstract. Data generation in RDF has been increasing over the last years as a means to publish heterogeneous and interconnected data. RDF is usually serialized in verbose text formats, which is problematic for publishing and managing huge datasets. HDT is a binary serialization of RDF that makes use of compact data structures, making it possible to publish and query highly compressed RDF data. This allows to reduce both the volume needed to store it and the speed at which it can be transferred or queried. However, it moves the burden of dealing with huge amounts of data from the consumer to the publisher, who needs to serialize the text data into HDT. This process consumes a lot of resources in terms of time, processing power, and especially memory. In addition, adding data to a file in HDT format is currently not possible, whether this additional data is in plain text or already serialized into HDT.

In this paper, we present HDTCat, a tool to merge the contents of two HDT files with low memory footprint. Apart from creating an HDT file with the added data of two or more datasets efficiently, this tool can be used in a divide-and-conquer strategy to generate HDT files from huge datasets with low memory consumption.

Keywords: RDF · Compression · HDT · Scalability · Merge · HDTCat

1 Introduction

RDF (Resource Description Framework)[1] is the format used to publish data in the Semantic Web. It allows to publish and integrate heterogeneous data. There exists a number of standard RDF serializations in plain text (N-triples, RDF/XML, Turtle, ...). While these serializations make RDF easy to process, the resulting files tend to be voluminous. A common solution consists of using a universal compressor (like bzip2) on the data before publication. This solution, however, requires the decompression of the data before using it by the consumer.

[1] https://www.w3.org/TR/rdf11-concepts/.

HDT (Header-Dictionary-Triples) is a binary serialization format that encodes RDF data in two main components: The Dictionary and the Triples. The Dictionary gives an ID to each term used in the data. These IDs are used in the Triples part to encode the graph structure of the data. Both components are serialized in compressed space using compact data structures that allow the data to be queried without the need to decompress it beforehand. Because of this, HDT has become the center piece of RDF data stores [2,11], public query endpoints [12], or systems for query answering in natural language [3,4]. However, the serialization process requires important amounts of memory, hampering its scalability. In addition, the current workflow to serialize RDF into HDT does not cover use cases such as adding data to an existing HDT file or merging two separate HDT files into one. This forces a user to fully decompress the HDT file.

In this paper we present HDTCat, a tool to merge two HDT files. This allows several functionalities: (1) to create an HDT file that combines the data of two HDT files without decompressing them, (2) to add data to an existing HDT file, by compressing this data first into HDT and then merging with the existing file, or (3) compressing huge datasets of RDF into HDT, by the means of splitting the data in several chunks, compressing each one separately and then merging them.

The rest of the paper is organized as follows: Sect. 2 presents background information about RDF and HDT, as well as related work on scalability of HDT serialization. Section 3 describes the algorithms of HDTCat. Section 4 shows how HDT performs against current alternatives. Finally, in Sect. 5 we give some closing remarks and present current and future lines of work for HDTCat.

2 Background

In this section, we provide basic background knowledge about RDF and how it is serialized into HDT. This is necessary to understand the approach to merge two HDT files.

2.1 RDF

RDF is the data model used in the Semantic Web. The data is organized in *triples* in the form (s, p, o), where s (the subject) is the resource being described, p (the predicate) is the property that describes it, and o (the object) is the actual value of the property. An object can be either a resource or a literal value. In a set of triples, resources can appear as subject or object in different triples, forming a directed labeled graph, which is known as *RDF graph*. Formal definitions for RDF triple and RDF graph (adapted from [9]) can be seen in Definition 1 and 2, respectively.

Definition 1 (RDF triple). *Assume an infinite set of terms* $\mathcal{N} = \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$, *where* $\mathcal{I}$, $\mathcal{B}$, *and* $\mathcal{L}$ *are mutually disjoint, and* $\mathcal{I}$ *are IRI references,* $\mathcal{B}$ *are Blank Nodes, and* $\mathcal{L}$ *are Literals. An RDF triple is a tuple* $(s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$, *where "s" is the subject, "p" is the predicate and "o" is the object.*

Definition 2 (RDF graph). *An RDF graph G is a set of RDF triples of the form (s, p, o). It can be represented as a directed labeled graph whose edges are $s \xrightarrow{p} o$.*

Example 1. The following snippet show an RDF file, that we call RDF_1, in N-Triples format:

```
<so1> <p1> <o1>.
<so1> <p1> <o2>.
<s1>  <p2> <so1>.
```

Moreover we denote as RDF_2 the following RDF file in N-Triples:

```
<so1> <p3> <o2>.
<o2> <p1> <s1>.
```

We will use these two files as running examples and show how they can be compressed and merged using HDTCat.

2.2 HDT

HDT [6] is a binary serialization format for RDF based on compact data structures. Compact data structures are data structures that compress the data as close as possible to its theoretic lower bound, but allow for efficient query operations. HDT encodes an RDF graph as a set of three components: (1) Header, that contains the metadata about the file and the data itself; (2) Dictionary, which assigns an unambiguous ID to each term appearing in the data; and (3) Triples, that replaces the terms by their ID in the dictionary and encodes them in a compressed structure. While HDT allows for different implementations of both Dictionary and Triples components, efficient default implementations are currently published. These implementations are the Four-Section Dictionary and the Bitmap Triples. We provide brief descriptions of those implementations down below.

The *Header* component stores metadata information about the RDF dataset and the HDT serialization itself. This data can be necessary to read the other sections of an HDT file. The *Dictionary* component stores the different IRIs, blank nodes, and literals, and assigns to each one an unambiguous integer ID. The *Triples* component stores the RDF graph, where all the terms are replaced by the ID assigned in the Dictionary component. From now on to represent an HDT file, we write $HDT = (H, D, T)$, where H is the header component, D is the dictionary component, and T is the triples component. In theory, each component allows different encoding. In practice, however, current compression formats are based in sorting lexicographically their elements. We describe thereafter characteristics of current HDT encoding.

In the Four-Section Dictionary an integer ID is assigned to each term (IRI, Blank Node and Literal). The set of terms is divided into four sections: (1) the *Shared* section, that stores the terms that appear at the same time as subjects and objects of triples; the *Subjects* section, which stores the terms that appear exclusively as subjects of triples; the *Objects* section, which contains the terms that appear only as object of triples; and finally the *Predicates* section, storing the terms that appear as predicates of the triples. From now on, we write the Dictionary as a tuple $D = (SO, S, O, P)$, where SO is the shared section, S is the subjects section, O is the objects section, and P is the predicates section. In each section the terms are sorted lexicographically and compressed (*e.g.*, using Plain [1] or Hu-Tucker FrontCoding [10]). The position of each term is then used as its implicit ID in each section. This way to each term an integer is assigned in a space-efficient way. The dictionary needs to provide global IDs for subjects and objects, independently of the section in which they are stored. Terms in P and SO do not change, while IDs for S and O sections are increased by the size of SO (*i.e.*, $ID_S := ID_S + \max(ID_{SO})$ and $ID_O := ID_O + \max(ID_{SO})$).

Example 2. Consider the file RDF_1 from Example 1. We call the corresponding HDT file $HDT[1] = (H_1, D_1, T_1)$ with $D_1 = (SO_1, S_1, O_1, P_1)$. The dictionary sections look as follows (note that the compression is not shown here as it is not important to understand HDTCat):

SO_1

IRI	ID
<so1>	1

S_1

IRI	ID
<s1>	2

O_1

IRI	ID
<o1>	2
<o2>	3

P_1

IRI	ID
<p1>	1
<p2>	2

Note that the ids in the S_1 and O_2 section start by 2 since there is one entry in the common section SO_1. Similarly for RDF_2 we get HDT_2 with:

SO_2

IRI	ID
<o2>	1
<so1>	2

S_2

IRI	ID

O_2

IRI	ID
<s1>	3

P_2

IRI	ID
<p1>	1
<p3>	2

$\square$

In the triples component T, each term in the triples is replaced by the ID from the dictionary and sorted in what is known as *Plain Triples*. The ordering is defined in the following.

Definition 3. *If* $T_1 = (s_1, p_1, o_1)$ *and* $T_2 = (s_2, p_2, o_2)$ *are two triples then* $T_1 \geq T_2$ *if and only if:*

1. $s_1 \geq s_2$;
2. *if* $s_1 = s_2$ *then* $p_1 \geq p_2$;
3. *if* $s_1 = s_2$ *and* $p_1 = p_2$ *then* $o_1 \geq o_2$;

Example 3. The triples from RDF_1 from Example 2 in *Plain Triples* are:

```
1 1 2
1 1 3
2 2 1
```

Note that they respect the order defined in Definition 3. The one from RDF_2 are:

```
1 1 3
2 2 1
```

Note that the triples were reordered.

□

The triples can be compressed in *Compact Triples*, which uses two coordinated sequences of IDs, Q_P and Q_O, to store the IDs of predicates and objects respectively, in the order they appear in the sorted triples. The first ID in Q_P is assumed to have the subject with $ID_s = 1$. Each following ID is assumed to have the same ID as its predecessor. If the ID 0 appears in the sequence, it means a change to the following ID (*i.e.*, the ID is incremented by one). Respectively, the first ID in Q_O is matched with the property in the first position of Q_P. Each following ID is assumed to have the property as its predecessor, and if the ID 0 appears in the sequence, it means a change to the following ID (that is, the next ID in Q_P). This can be further compressed in *BitMap Triples* by removing the 0 from the ID sequences and adding two bit sequences, B_P and B_O, that mark the position where the change of subject (for Q_P) or predicate (for Q_O) happen. Note that the data-structures described above allow fast retrieval of all triple patterns with fixed subject. Some more indexes are added to resolve fast triple patterns with fixed predicate or object. Moreover, note that due to the global ordering updates are not supported.

2.3 Works on Scalability of HDT

To the best of our knowledge, there are only two publications that deal with scalable HDT generation. The first one is HDT-MR [7], a MapReduce-based tool to serialize huge datasets in RDF into HDT. MDT-MR has proven able to compress more than 5 billion triples into HDT. However, HDT-MR needs a MapReduce cluster to compress the data, while HDTCat can run in a single computer.

A single HDT file containing over 28 billion triples has been published in LOD-a-lot [5]. The aim was to generate a snapshot of all current RDF triples in the LOD cloud. However, both the algorithm and tool used to create this HDT file are not public. HDTCat tries to fill this gap making the algorithm and tool needed to create HDT files of this size open to the public.

3 HDTCat

In this section we describe the HDTCat algorithm. Given two HDT files HDT_1 and HDT_2, HDTCat generates a new HDT file HDT_{cat} that contains the union of the triples in HDT_1 and HDT_2. Its goal is to achieve this in a scalable way, in particular in terms of memory footprint, since this is generally the limited resource on current hardware.

Let's assume two HDT files, $HDT_1 = (H_1, D_1, T_1)$ and $HDT_2 = (H_2, D_2, T_2)$ are given. The current solution to merge these two HDT files is to first decompress them into text. Then, the two text files are concatenated, and the resulting file is serialized again into HDT. Basically, two ordered lists are put one after the other and ordered again without exploiting their initial order. The problem addressed by HDTCat is how to merge the dictionaries D_1, D_2 and the triples T_1, T_2 without decompressing them, so that the resulting HDT file contains the union of the RDF triples. The result of of merging the two HDT files needs to be the same as the serialization of the contatenation of the two uncompressed files, that is `rdf2hdt(`RDF_1`+`RDF_2`)=hdtcat(rdf2hdt(`RDF_1`),rdf2hdt(`RDF_2`))`.

The algorithm can be decomposed into three phases:

1. Joining the dictionaries,
2. Joining the triples,
3. Generating the header.

For the two first phases, HDTCat uses merge-sort-based algorithms that take advantage of the initial ordering of the HDT components. The general idea of the algorithms is described in Fig. 1. Briefly, there are two iterators over the two lists. Recursively, the current entries of the two iterators are compared and the lowest entry is added to the final list.

There are two important consequences. Imagine the two components have n respectively m entries. The first consequence is that the time complexity is reduced. If two components are merged by first decompressing and then serializing their union, the time complexity is $O((n + m) \cdot log(n + m))$ because of the need to merge an unsorted set of triples. However, when sorting two already sorted lists, using the algorithm above, the time complexity is $O(n + m)$. The second, and in our eyes the more important, is the memory consumption. The existing approach to serialize RDF into HDT stores every uncompressed triple in memory so that the memory complexity is in the order of $O(n+m)$. Iterating over the sorted lists by letting them compressed, and decompressing only the current entry, reduces the memory complexity to $O(1)$ This explains the main idea behind HDTCat. We are now going to explain more in detail the merging strategy and the data-structures needed.

3.1 Joining the Dictionary

Assume two HDT dictionaries $D_1 = (SO_1, S_1, O_1, P_1)$ and $D_2 = (SO_2, S_2, O_2, P_2)$. The goal is to create a new HDT dictionary $D_{cat} = (SO_{cat}, S_{cat}, O_{cat}, P_{cat})$.

Merging the sections P_1 and P_2 is a simple process. P_1 and P_2 are two arrays of ordered compressed strings. Algorithm 1 assumes that there are two iterators over the two lists. Recursively, the current entries of the two iterators are compared and the lowest entry is added to the final list. To compare the entries they are decompressed, and the new entry is compressed directly and

```
Data: Two sorted lists a and b
Result: A sorted list c containing all entities in a and b
n= length of a; m = length of b
allocate c with length n+m
i = 1; j = 1
while $i < n \,||\, j < m$ do
    if $i = n$ then
        copy rest of b into c
        break
    end
    if $j = m$ then
        copy rest of a into c
        break
    end
    if $a[i] < b[j]$ then
        copy a[i] into c
        i=i+1
    end
    if $b[j] < a[i]$ then
        copy b[j] into c
        j=j+1
    end
    if $a[i] = b[j]$ then
        copy a[i] into c
        i=i+1
        j=j+1
    end
end
```

Algorithm 1: Algorithm to merge two sorted lists. Note that the algorithm has a time complexity of $O((n+m))$. All computation do not need to be done on RAM but can be performed on disk.

added to P_{cat}. Note that since the strings are uncompressed and compressed directly, the memory footprint remains low.

Example 4. The predicate section of HDT_{cat} is:

P_{cat}

IRI	ID
<p1>	1
<p2>	2
<p3>	3

□

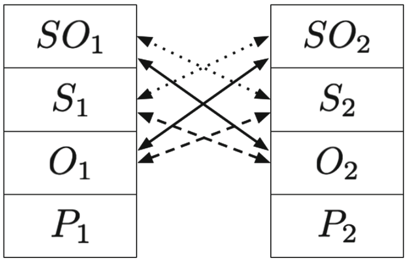

Fig. 1. This figure shows the non-trivial sections that can share an entry. Clearly SO_1 and SO_2, S_1 and S_2, O_1 and O_2, P_1 and P_2 can contain common entries. The other sections that can contain common entries are connected by a double arrow. It is important to take care of these common entries when merging the dictionaries.

Merging the other sections needs to take into account, however, that some terms can move to different sections in the HDT files to be merged. For example, if S_1 contains an IRI that appears also in O_2. Figure 1 shows the sections that can contain common elements (excluding the non-trivial cases). The following cases need to be taken into account:

- If SO_1 and S_2, or S_1 and SO_2 contain common entries, then they must be skipped when joining the S sections.
- If SO_1 and O_2, or O_1 and SO_2 contain common entries, then they must be skipped when joining the O sections.
- If S_1 and O_2, or O_1 and S_2 contain common entries then they must be skipped when joining the S and O sections, and additionally they must be added to the SO_{cat} section.

For this reason, terms can be assigned to different sections in the final HDT dictionary. For the example where S_1 contains as IRI that appears also in O_2, this IRI should be assigned to the section SO_{cat}, since the IRI will appear both in the subject and the object of some triples. Figure 2 shows to which sections the terms can be assigned depending on where they are in the initial dictionaries.

Example 5. The sections of HDT_{cat} different from P_{cat} look like this:

SO_{cat}

IRI	ID
<o2>	1
<so1>	2
<s1>	3

S_{cat}

IRI	ID

O_{cat}

IRI	ID
<o1>	4

Note that the IRI <s1> moved from section S_1 to section SO_{cat}.

□

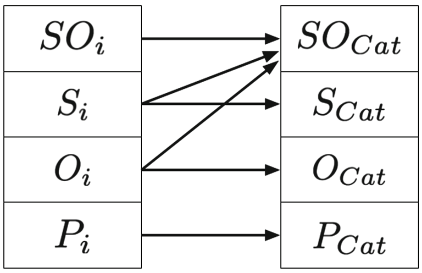

Fig. 2. This figure shows to which sections of the HDT_{cat} dictionary, the entries from the dictionary section of either HDT_1 or HDT_2 can move. The SO section and the P section ids are going to the SO_{cat} and P_{cat} section respectively. If there is an entry that appears both in the S section and the O section, then the corresponding entry will go to the SO_{cat} section. Otherwise the entry goes to the S or O section.

To store the merged sections of HDT_{cat}, since they are written sequentially, data-structures stored on disk can be used, reducing their memory complexity to $O(1)$.

When joining the triples in the next step, it will be necessary to know the correspondence between the IDs in D_1 and D_2, and the IDs in D_{cat}. To keep track of those mappings, we introduce data structures that, for each ID in the section $Sec \in \{SO_1, S_1, O_1, P_1, SO_2, S_2, O_2, P_2\}$,. assign the new ID in the corresponding section $Sec_{cat} \in \{SO_{cat}, S_{cat}, O_{cat}, P_{cat}\}$. For one section Sec the data structure contains two arrays:

1. An array indicating, for each ID of Sec, which is the corresponding section in Sec_{cat}.
2. An array mapping the IDs of Sec to the corresponding ID in the section Sec_{cat}.

We indicate every such mapping as M(Sec). Moreover, we construct also the mappings form SO_{cat},S_{cat} (note: the IDs of these two sections are consecutive) to SO_1, S_1 and SO_2, S_2 respectively. This consists of two arrays:

1. An array indicating ,for each ID of SO_{cat} or S_{cat}, the corresponding ID in SO_1, S_1 (if it exists).
2. An array indicating for each ID of SO_{cat} or S_{cat}, the corresponding ID in SO_2,S_2 (if it exists).

The arrays are directly written to disk. We indicate the two mappings as M(cat,1) and M(cat,2) .

Example 6. The mappings for HDT_{cat} are as follows:

M(SO_1)

ID	Sec_{cat}	ID_{cat}
1	SO_{cat}	2

M(S_1)

ID	Sec_{cat}	ID_{cat}
2	SO_{cat}	3

M(O_1)

ID	Sec_{cat}	ID_{cat}
2	O_{cat}	4
3	SO_{cat}	1

M(P_1)

ID	Sec_{cat}	ID_{cat}
1	P_{cat}	1
2	P_{cat}	2

M(SO_2)

ID	Sec_{cat}	ID_{cat}
1	SO_{cat}	1
2	SO_{cat}	2

M(S_2)

ID	Sec_{cat}	ID_{cat}

M(O_2)

ID	Sec_{cat}	ID_{cat}
3	SO_{cat}	1

M(P_2)

ID	Sec_{cat}	ID_{cat}
1	P_{cat}	1
2	P_{cat}	3

M(cat,1)

ID_{cat}	ID_{old}
1	-
2	1
3	2

M(cat,2)

ID_{cat}	ID_{old}
1	1
2	2
3	-

□

3.2 Joining the Triples

In this section we describe the process to merge the triples T_1, T_2 in HDTCat. This process exploits the fact that the triples are ordered only indirectly. That is, the fact that the HDT files are queriable.

Remember that by Definition 3 the triples need to be ordered first by subjects, then by predicates, and finally by objects. The order of the subjects is given by the subjects section in the merged dictionary S_{cat}. Then, for each ID in S_{cat}, we use the mappings $M(S[cat], S[1])$ and $M(S[cat], S[2])$ (constructed when joining the dictionary sections) to find the IDs ID_1 and ID_2 of the original HDT files HDT_1 and HDT_1 that mapped to ID_{cat} in HDT_{cat}. Since both HDT_1 and HDT_2 are queriable, we can retrieve all triples with subjects ID_1 and ID_2 respectively. By using again the mappings constructed when joining the dictionaries, we can now translate the IDs of these triples used in HDT_1 and HDT_2 to the corresponding IDs in HDT_{cat}. We generate the triples by iterating over the subjects and by writing the triples directly to disk.

Example 7. Let's first join the triples with $ID_{cat} = 1$. According to $M_{Cat,2}$ there are only triples in HDT_2 mapping to it. In fact there is only the triple:
1 2 3
By using the mappings of Example 6 this will become:
1 3 1
For $ID_{cat} = 2$ we search all triples associated to $ID_{cat} = 2$. These triples are:
1 1 2
1 1 3
in HDT_1 and:
2 2 1

in HDT_2. By using the mappings of Example 6 these correspond to the new IDs:
2 1 3
2 1 1
and:
2 3 1
Note that the triples of HDT_1 where initially ordered, while the mapped triples are not ((2,1,3)>(2,1,1)). The merged triples for $ID_{cat} = 2$ are then:
2 1 1
2 1 3
2 3 1

□

3.3 Creating the Header

While the dictionary and the triples must be merged from the corresponding sections of the two HDT files, the header just contain some statistical information like the number of triples and the number of distinct subjects. This means that there is nothing to do here except writing the statistics corresponding to D_{cat} and T_{cat} that have been generated.

4 Experiments

In this section we evaluate the performance of HDTCat. In particular we compare the scalability of HDTCat when generating HDT files (starting from N-Triples against (1) the regular HDT serialization, using the command line tool `rdf2hdt` that is part of the HDT repository[2], and (2) HDT-MR. We perform three different experiments to compare how HDTCat performs in different situations.

Experiment 1. We use synthetic data generated using LUBM [8]. LUBM is a benchmark to test the performance of SPARQL queries and contains both a tool to generate synthetic RDF data and a set of SPARQL queries. The generated RDF contains information about universities (like departments, students, professors and so on). We generated the following LUBM datasets: (1) from 1000 to 8000 universities in steps of 1000, and (2) from 8000 to 40000 universities in steps of 4000. We used 3 methods to compress these files to HDT:

– **rdf2hdt:** We concatenate the LUBM datasets generated to obtain the datasets of increasing size by steps of 1000 universities up to 8000 universities, then we increase the steps by 4000 universities. We then used rdf2hdt to generate the corresponding HDT files.
– **HDT-MR:** HDT-MR is used in the same way as rdf2hdt, using the same concatenated files, and then converted to HDT.

[2] https://github.com/rdfhdt/hdt-java.

- **HDTCat:** We first serialized the generated datasets into HDT, then we used HDTCat to recursively compute the merged HDT files. *I.e.*, we generated lubm.1–2.000.hdt from lubm.1–1.000.hdt and lubm.1001–2.000.hdt; then lubm.1–3.000.hdt from lubm.1–2.000.hdt and lubm.2001–3.000.hdt; and so on.

We run the experiments for rdf2hdt and HDTCat on different hardware configurations:

- **Configuration 1:** A server with 128 Gb of RAM, 8 cores of type Intel(R) Xeon(R) CPU E5–2637 v3 @ 3.50 GHz. RAID-Z3 with 12x HDD 10TB SAS 12Gb/s 7200 RPM. We run hdt2rdf and hdtCat on this configuration. For the results of HDT-MR we report the ones achieved by [7], that where executed on a cluster with a total memory of 128 Gb of RAM. While rdf2hdt and HDTCat are designed to be used on a single server, HDT-MR is designed to be used on a cluster. To make the results comparable we choose a single node and a cluster configuration with the same amount of RAM since this is the limited resource for compressing RDF serializations to HDT.
- **Configuration 2:** A server with 32 Gb of RAM, 16 cores of type Intel(R) Xeon(R) CPU E5–2680 0 @ 2.70 GHz. RAID-Z3 with 12x HDD 10TB SAS 12Gb/s 7200 RPM.
- **Configuration 3:** A desktop computer with 16 Gb of RAM, AMD A8–5600K with 4 cores. 1x HDD 500GB SCSI 6 Gb/s, 7200 RPM.

Note that while the two first configurations have a RAID deployment with 10 drives, the third one is limited to a single HDD. Since HDTCat is I/O intensive, this can affect its performance.

The results obtained by the 3 methods on the 3 hardware configurations are shown in Table 1. It summarizes the comparison between the three methods to generate HDT from other N-Triples of LUBM datasets. **T** indicates the time and **M** the maximal memory consumption of the process. In the case of HDTCat we also report T_{com} the time to compress the N-Triples into HDT and T_{cat} the time to cat the two files together. $\star$ indicates that the experiment failed with an OUT OF MEMORY error. "$-$" indicates that the experiment was not performed. This has two reasons. Either a smaller experiment failed with an OUT OF MEMORY, or the experiment with HDT-MR was not performed on the corresponding configuration. The experiments in the T_com column are very similar because we compress similar amount of data. We report the average times of these experiments and indicated that with "$*$".

The results for Configuration 1 show that while hdt2rdf fails to compress lubm-12000, by using HDTCat we are able to compress lubm-40000. This means that one can compress at least as much as the HDT-MR implementation. Note that lubm-40000 does not represent an upper bound for both methods. For lubm-8000, HDT-MR is 121% faster then HDTCat. This is expected since HDT-MR exploits parallelism while HDTCat does not. Moreover while the single node configuration has HDD disks, the cluster configuration used SSD disks. For

Table 1. Comparison between methods to serialize RDF into HDT.

Configuration 1: 128 Gb RAM

LUBM	Triples	hdt2rdf		HDT-MR	HDTCat			
		T (s)	M (Gb)	T (s)	T_com (s)	T_cat (s)	T (s)	M_cat (Gb)
1000	0.13BN	1856	53.4	936	970*	–	–	—
2000	0.27BN	4156	70.1	1706		317	2257	26.9
3000	0.40BN	6343	89.3	2498		468	3695	35.4
4000	0.53BN	8652	105.7	3113		620	5285	33.8
5000	0.67BN	11279	118.9	4065		803	7058	41.7
6000	0.80BN	23595	122.7	4656		932	8960	47.5
7000	0.93BN	78768	123.6	5338		1088	11018	52.9
8000	1.07BN	⋆	⋆	6020		1320	13308	58.7
12000	1.60BN	–	–	9499	4710*	1759	19777	54.7
16000	2.14BN	—	–	13229		2338	26825	73.4
20000	2.67BN	–	–	15720		2951	34486	90.5
24000	3.20BN	–	–	26492		3593	42789	90.6
28000	3.74BN	–	–	36818		4308	51807	84.9
32000	4.27BN	–	–	40633		4849	61366	111.1
36000	4.81BN	–	–	48322		6085	72161	109.4
40000	5.32BN	–	–	55471		7762	84633	100.1

Configuration 2: 32 Gb RAM

LUBM	Triples	HDT		-	HDTCat			
		T (s)	M (Gb)	-	T_com (s)	T_cat (s)	T (s)	M_cat (Gb)
1000	0.13BN	1670	28.3	-	1681*	–	–	–
2000	0.27BN	⋆	⋆	-		454	3816	17.3
3000	0.40BN	—	–	–		660	6366	20.1
4000	0.53BN	–	–	—		869	8916	25.5
5000	0.67BN	–	–	–		1097	11694	29.3
6000	0.80BN	–	–	–		1345	14720	28.5
7000	0.93BN	–	–	–		1584	17985	30.6
8000	1.07BN	–	–	–		1830	21496	30.4
12000	1.60BN	–	–	–	⋆	2748	-	31.0
16000	2.14BN	–	–	–	–	3736	—	31.1
20000	2.67BN	–	–	–	–	5007	–	30.5
24000	3.20BN	–	–	–	–	5514	–	30.8
28000	3.74BN	–	–	–	–	6568	–	30.8
32000	4.27BN	–	–	–	–	7358	–	30.8
36000	4.81BN	–	–	–	–	9126	–	30.6
40000	5.32BN	–	–	–	–	9711	–	30.8

Table 1. (*continued*)

Configuration 3: 16 Gb RAM

LUBM	Triples	HDT		-	HDTCat			
		T (s)	M (Gb)	–	T_com (s)	T_cat (s)	T (s)	M_cat (Gb)
1000	0.13BN	2206	14.5	–	2239*	–	–	–
2000	0.27BN	⋆	⋆	–		517	4995	10.7
3000	0.40BN	–	–	–		848	8082	11.8
4000	0.53BN	–	–	–		1301	11622	11.9
5000	0.67BN	–	–	–		1755	15616	12.7
6000	0.80BN	–	–	–		2073	19928	11.8
7000	0.93BN	–	–	–		2233	24400	12.6
8000	1.07BN	–	–	–		3596	30235	12.2
12000	1.60BN	–	–	–	⋆	4736	–	14.3
16000	2.14BN	–	–	–	–	6640	–	14.3
20000	2.67BN	–	–	–	–	9058	–	14.4
24000	3.20BN	–	–	–	–	10102	–	14.3
28000	3.74BN	–	–	–	–	13287	–	12.8
32000	4.27BN	–	–	–	–	14001	–	13.9
36000	4.81BN	–	–	–	–	17593	–	14.0
40000	5.32BN	–	–	–	–	19929	–	13.9

lubm-40000 the speed advantage reduces, HDT-MR is 52% faster then HDT-Cat. The results for Configuration 2 show that the speed of hdtCat to compress lubm-40000 in comparison to Configuration 1 is reduced, but only by 25%. The results for Configuration 3 show that it is possible to compress on a 16 Gb machine HDT files containing 5 Billion triples. In particular this means that it is possible to index on a 16Gb machine an RDF file with 5 Billion triples and construct a SPARQL endpoint on top. This is unfeasible for every other SPARQL endpoint implementation we are aware of. Moreover this also shows that for Configuration 1, lubm-4000 is far from being an upper bound so that potentially huge RDF files can be indexed, which was not imaginable before.

Experiment 2. While the above results are using the synthetic data provided by LUBM we also performed an experiment using real datasets. In particular we join the Wikidata dump of the 19-02-2018 (330G in ntriple format) and the 2016 DBpedia dump[3] (169G in ntriple format). This corresponds to 3.5 billion

[3] All files retrieved by: wget -r -nc -nH –cut-dirs=1 -np -l1 -A ‘*ttl.bz2’ -A ‘*.owl’-R ‘*unredirected*’–tries 2 http://downloads.dbpedia.org/2016-10/core-i18n/en/, i.e. all files published in the english DBpedia. We exclude the following files: nif_page_structure_en.ttl, raw_tables_en.ttl and page_links_en.ttl since they do not contain typical data used in application relying on DBpedia.

triples. We where able to join the corresponding HDT file in 143 min and 36 s using a 32 Gb RAM machine. The maximal memory consumption was 27.05 Gb.

Experiment 3. Note that Wikidata and DBpedia are not sharing many IRIs. So one valid argument is if HDTCat is also performing well when the two joined HDT files contain many common IRIs. To test this we randomly sorted the lubm.2.000.nt file and split it in two files containing the same amount of triples. We then join them using HDTCat. While joining lubm.1–1000.hdt and lubm.1001–2000.hdt took 287 s, joining the randomly sorted files took 431 s. This corresponds to a 66% increase of time which is expected. This shows that HDTCat is still performing well in such a scenario.

Code. The code for HDTCat is currently part of the HDT code repository available under https://github.com/rdfhdt/hdt-java. The code is released under the *Lesser General Public License* as the existing Java code. We also provide a command line tool, called rdf2hdtcat, that allows to compress HDT in a divide and conquer method (pull request #109) to easily serialize big RDF file to HDT.

5 Conclusion and Future Work

In this paper we have presented HDTCat, an algorithm and command line tool to merge two HDT files with improved time and memory efficiency. We have described in detailed how the algorithm works and we have compared our implementation against the other two available alternatives: regular HDT serialization and HDT-MR, a MapReduce-based designed to tackle scalability in HDT serialization. The experiments shows that it is possible to compress 5 billion triples on a 16 Gb machine which was not imaginable before.

Our future work include the creating of a tool that combines rdf2hdt and HDTCat to parallelize RDF serialization into HDT to generate HDT files faster. Moreover we are working on extending HDTCat to be able to merge an arbitrary number of HDT files simultaneously.

In the long term, we plan to work in the use of HDTCat to support updates on HDT-based tools. A strategy is to have a read-only index and to store the updates in a delta structure that is periodically merged (with HDTCat) with the read-only part.

Finally we believe that HDTCat will enable the Semantic Web Community to tackle scenarios which were non feasible before because of scalability.

Acknowledgements. We would like to thank Pedro Migliatti for executing part of the experiments as well as Javier D. Fernández for the helpful discussions with him. We also want to thank Dimitris Nikolopoulos and Wouter Beek from Triply for porting the algorithm to C++. This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 642795.

References

1. Brisaboa, N.R., Cánovas, R., Martínez-Prieto, M.A., Navarro, G.: Compressed string dictionaries. CoRR abs/1101.5506 (2011). http://arxiv.org/abs/1101.5506
2. Curé, O., Blin, G., Revuz, D., Faye, D.C.: WaterFowl: a compact, self-indexed and inference-enabled immutable RDF store. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8465, pp. 302–316. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07443-6_21
3. Diefenbach, D., Both, A., Singh, K., Maret, P.: Towards a question answering system over the semantic web. Semant. Web J. 1–19 (2020)
4. Diefenbach, D., Singh, K., Maret, P.: WDAqua-core0: a question answering component for the research community. In: Dragoni, M., Solanki, M., Blomqvist, E. (eds.) SemWebEval 2017. CCIS, vol. 769, pp. 84–89. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69146-6_8
5. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: LOD-a-lot. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 75–83. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_7
6. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). J. Web Semant. **19**(22–41), 00124 (2013)
7. Giménez-García, J.M., Fernández, J.D., Martínez-Prieto, M.A.: HDT-MR: a scalable solution for RDF compression with HDT and MapReduce. In: Gandon, F., Sabou, M., Sack, H., d'Amato, C., Cudré-Mauroux, P., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9088, pp. 253–268. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18818-8_16
8. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Semant. **3**(2), 158–182 (2005)
9. Gutierrez, C., Hurtado, C.A., Mendelzon, A.O., Pérez, J.: Foundations of semantic web databases. J. Comput. Syst. Sci. **77**(3), 520–541 (2011). https://doi.org/10.1016/j.jcss.2010.04.009
10. Hu, T.C., Tucker, A.C.: Optimal computer search trees and variable-length alphabetical codes. Siam J. Appl. Math. **21**(4), 514–532 (1971)
11. Martínez-Prieto, M., Arias, M., Fernández, J.: Exchange and consumption of huge RDF data. In: Proceeding of ESWC, pp. 437–452 (2012)
12. Verborgh, R., et al.: Querying Datasets on the Web with High Availability. In: Mika, Peter, Tudorache, Tania, Bernstein, Abraham, Welty, Chris, Knoblock, Craig, Vrandečić, Denny, Groth, Paul, Noy, Natasha, Janowicz, Krzysztof, Goble, Carole (eds.) ISWC 2014. LNCS, vol. 8796, pp. 180–196. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_12, http://linkeddatafragments.org/publications/iswc2014.pdf