Generate and Update Large HDT RDF Knowledge Graphs on Commodity Hardware

Antoine Willerval[1,2(✉)], Dennis Diefenbach[2], and Angela Bonifati[1]

[1] CNRS, LIRIS UMR 5205, University of Lyon, Lyon, France
angela.bonifati@univ-lyon1.fr
[2] The QA Company SAS, Saint-Étienne, France
{antoine.willerval,dennis.diefenbach}@the-qa-company.com

Abstract. HDT is a popular compressed file format to store, share and query large RDF Knowledge Graphs (KGs). While all these operations are possible in low hardware settings (i.e. a standard laptop), the generation and updates of HDT files come with an important hardware cost especially in terms of memory and disk usage.

In this paper, we present a new tool leveraging HDT, namely k-HDTDiffCat, that allows to a) reduce the memory and disk footprint for the creation of HDT files and to b) remove triples from an existing HDT file thus allowing updates.

We show that in a system with 8 times less memory, we can achieve HDT file generation in almost the same time as existing methods. Moreover, our system allows to remove triples from an HDT file catering for updates. This operation is possible without the need to uncompress the original data (as it was the case in the original HDT file) and by keeping low memory consumption.

While HDT was suited for storing, exchanging and querying large Knowledge Graphs in low hardware settings, we also offer the novel functionality to generate and update HDT files in these settings. As a side effect, HDT becomes an ideal indexing structure for large KGs in low hardware settings making them more accessible to the community.

In particular, we show that we can compress the whole Wikidata graph, which is the largest knowledge graph currently available, on a standard laptop with 16 GB of RAM as well as generate Wikidata indexes that are at most 24 h behind the live Wikidata endpoint.

Keywords: HDT · HDTq · HDTCat · HDTDiff · LUBM · Wikidata · Update

1 Introduction

RDF Knowledge Graphs (KGs) are growing in size, storing and querying them becomes increasingly challenging. For example, the amount of triples in the Wikidata KG increased from 18 billion to 19.2 billion in the last year. Moreover,

A. Meroño Peñuela et al. (Eds.): ESWC 2024, LNCS 14665, pp. 128–144, 2024.
https://doi.org/10.1007/978-3-031-60635-9_8

serving large graphs can easily become expensive in terms of hardware resources, making them inaccessible for consumers in low hardware settings, thus impeding the democratization of the Web.

To overcome this problem, the HDT file format was created [10]. HDT uses succinct data structures [13] to store RDF graphs. The aim of succinct data structures is to compress the underlying dataset as much as possible while preserving the query capabilities. In the case of HDT, it is possible to compress RDF graphs to similar sizes as a GZIP compressed NTriple file while allowing to search for triple patterns at a speed that is in the same order of magnitude of common triple stores. As a concrete example, it is possible to compress the whole Wikidata KG to an HDT file of 300 GB and have a triple pattern access that is equal or faster then the one of the public Wikidata query service.

This explains why HDT is used as the backbone of past projects where scalability becomes a problem. For instance in the LOD laundromat [2] in order to expose datasets in the LOD cloud, for question-answering engines [3] and as an indexing structure of Linked Data Fragments [18]. HDT was also used to store the 28 billion triples dump of the LOD cloud called LOD-a-lot [8].

HDT is used to provide large data, for example in LOD laundromat, providing an infrastructure to clean and serve all datasets in the LOD cloud and LOD-a-lot [8], an index of the whole Linked Open Data cloud published as HDT. For the question answering system WDAqua-core1 [5], a question answering system that allows to query some of the largest datasets in the LOD cloud, namely: DBpedia, Wikidata, MusicBrainz and DBlp. But HDT is mostly use for triple stores and RDF graph querying like Linked Data Fragments [18] to query the Web of data on Web-scale by moving the query load from servers to clients, qEndpoint [19] a SPARQL endpoint that uses HDT as an underlying index and OSTRICH [17], a hybrid archiving approach for RDF (Resource Description Framework) graphs. It is designed to provide efficient triple pattern queries for different versioned query types while keeping storage requirements reasonable.

The key contribution of the paper is k-HDTDiffCat, a new method to combine and delete triples from one or multiple HDT files into a new one. Unlike HDTCat [4], a method to combine two HDT files into one HDT file, our method allows to combine an arbitrary number of HDT files while keeping a small time overhead per added HDT file. It allows to create big HDT files by splitting the HDT generation into small HDT files. The small HDT files are then merged with k-HDTDiffCat into one HDT file. Moreover k-HDTDiffCat allows also to eliminate triples from already generated HDT files.

With provide experiments showing the efficiency of our method compared with previous implementation in systems with few resources.

While storing, exchanging and querying HDT files is easy in low hardware settings, this is not the case for the generation and updates. The tool presented in this paper fills exactly this gap, i.e. allowing to efficiently generate and update HDT files in low hardware settings. The paper is organized as follows.

We will first explain the HDT internals in Sect. 2, in Sect. 3 we discuss the related works, in Sect. 4 we present our tool, in Sect. 5 we present our

experimental results and within Sect. 6 a presentation of the open source repository where to find our tool with a command line to use it.

2 HDT Internals

Before describing the related works and our contribution, we describe on a high level the internals of a HDT file. HDT [10,11] is a file format to compress an RDF graph while keeping the ability to query over it by triple patterns, it is composed of 3 components. The header component, the dictionary component and the triples component.

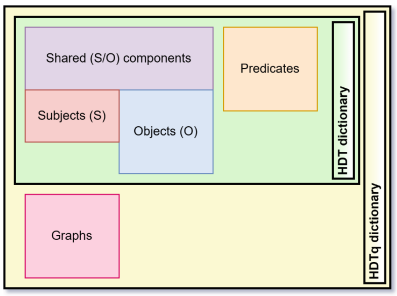


Fig. 1. The different HDT and HDTq components in the dictionary part.

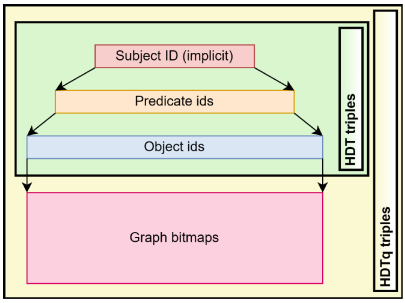


Fig. 2. The different HDT and HDTq components in the triples part.

H*eader*. It contains metadata information about the HDT file like the number of RDF terms, the raw size, the number of distinct subjects, etc. Due to the trivial aspect of this component, we won't consider it in the future sections.

D*ictionary* (Fig. 1). It contains a mapping between the terms in the RDF graph (i.e. URIs, Literals and Blank Nodes) and numerical IDs.

The default dictionary is called Four Section Dictionary (FSD), previous HDT Dictionary [10] is composed of four sections: Shared (SH), Subjects (S), Predicates (P) and Objects (O). The Shared section contains all terms that appear both as a subject and as an object inside the graph. The subject section contains terms that appear exclusively in the subject position. Similarly with the object section, only containing the term appearing at the object position. The predicate section contains all terms contained in the predicate position. In each section the terms are sorted lexicographically and compressed using Front-Coding [21] which is a technique based on differential compression over the string prefixes.

Other dictionary implementations exist called Multiple Section Dictionaries (MSD) [16]. These versions are based on the FSD above, but are splitting the literals in the object section by data-types or language. This saves space by grouping the literals with the same datatypes and allows quick retrieval of the

literals of a given datatype or language while allowing a fast lookup of a datatype or language using the literal numerical ID.

T*riples* (Fig. 2). It contains the triples encoded with the above IDs using adjacency lists. On a high level these are lists of triples sorted by subject-predicate-object (i.e. an SPO index). These are stored in a compressed format using two bitmaps and two sequences. The shared section terms are represented by adding to non-shared subject/object terms the count of shared terms. If the ID is lower than the shared terms count, it means the ID is representing the position inside the shared section. If the ID is greater than the shared terms count, it means the ID is representing a subject or an object.

Overall one can summarize HDT contains two components, the Dictionary component and the Triples component that use a global ordering to store and compress the data.

2.1 HDTq

HDTq [9] is an extension of HDT to allow the storage of named graphs or **q**uads.

In the dictionary, it adds another sorted section to store the named graphs. (HDTq dictionary part of Fig. 1).

In the triples component, this new section of n named graphs is reflected with n bitmaps that have a length corresponding to the number of triples. Each bitmaps indicates with 1 if the corresponding triple is in the named graph associated with the bitmap. These bitmaps are compressed using Roaring bitmaps [15]. (HDTq triples in Fig. 2).

Another version of HDTq exists using one bitmap per triple instead of one bitmap per named graph. We do not consider this version because it was getting worse results than the one bitmap per named graph version.

3 Related Work

We first describe the current methods to compress an RDF file into HDT and then describe general RDF graph indexing methods. There are currently 3 methods to compress an RDF file to HDT. *rdf2hdt* [11], is the original implementation used to compress an RDF file to HDT. This implementation is fully in memory. It basically performs a lexicographic sorting of the terms (URIs, Literals, blank nodes) for the dictionary and of the IDs in SPO order in the triples. *HDT-MR* [12], is an implementation using a MapReduce setup to compress large dataset into HDT. This algorithm can only be executed on a MapReduce cluster. This requirement is a hard stop for users having access only to commodity hardware. *HDTCat* [4], is an algorithm to merge 2 HDT files together without the need to uncompress the data. This method was used to compress large HDT files by creating small HDT files and combine them recursively into one big HDT file. There are no methods to delete triples from an existing HDT file without uncompressing the dataset.

Besides methods to compress RDF graphs to HDT, the most related works include triple store indexing strategies. In these cases the RDF graph is stored into some internal indexing structure that is subsequently used to query it via SPARQL. Typical indexing structures are:

- B+-trees like Blazegraph[1], Apache Jena[2] or RDF4J Native Store[3].
- Storing the graph into a SQL database, like Virtuoso [6].
- Custom binary structure, for example inside QLever [1] using a read-only compressed data structure.

Overall these indexing strategies are memory intensive making it difficult to run them on commodity hardware.

4 Contribution

In this paper, we present k-HDTDiffCat, a new tool that allows to merge and update multiple HDT files. In combination with an indexing algorithm, it can be used to compress and update large HDT files in low hardware settings. More precisely, we provide the following contributions:

1. Allow to combine multiple HDT files together at the same time.
2. Allow to subtract triples from HDT files.
3. Extend these algorithms also for named graphs.

4.1 k-HDTCat

While all functionalities are integrated in the same tool, we first describe the functionality of k-HDTCat, i.e. merging together mutliple HDT files.

Dictionary Generation. An HDT dictionary is composed of multiple sorted sections. Inside each section the RDF terms respect multiple properties,

1. An RDF term is unique in its section. (No duplicated term)
2. If an RDF term is used as a subject and as an object, it is inside the shared section and doesn't appear inside a subject or an object section.
3. An RDF term is represented by its position or index in the section.

We describe in the following how we merge multiple HDT files named $HDT_1 \dots HDT_k$ with Shared, Subject, Object and Predicate sections named respectively $SH_1 \dots SH_k$, $S_1, \dots S_k$, $O_1, \dots O_k$ and $P_1, \dots P_k$.

As all sections are sorted, we create iterators over each section type. it_{SH} is an iterator over all shared sections $(SH_1, \dots SH_k)$ that returns all terms of the shared sections sorted without duplicates using a merge join. Analogously we

[1] https://blazegraph.com.
[2] https://jena.apache.org/index.html.
[3] https://rdf4j.org/.

create it_S for subjects, it_O for objects and it_P for predicates. By comparing it_S and it_O we can compute the terms that need to be moved into the new shared section SH_{new} with the already shared terms from it_{SH}. All terms that are not moved to this section will remain in the S_{new}, O_{new} sections respectively. With that, we respect the properties 1 and 2.

Added to this method, we keep track of the original term index with the origin HDT IDs and sections, so when we compute the end section S_{new}, O_{new}, SH_{new} and P_{new}, we have both the old and new IDs of each term. Using them, we create 4 arrays $M_{Sec,HDT}$, one per HDT file and section $Sec \in \{S, O, P, SH\}, HDT \in \{HDT_1, \dots HDT_k\}$. We call them "maps" and we fill them using the origin term location i.e. ID/Section/HDT file as the index and the new location and if the term is shared for the value. $M_{Sec,HDT}[OldID] = (NewID, IsShared)$.

With this task done, we have now access to the new sections for the dictionary with a mapping between the old and the new term's indexes.

Triples Component Generation. The dictionary being created, we need to build the triples component, for this part, we extended the HDTCat [4] algorithm to handle multiple triples by using a merge join. First we read the triple ids from the HDT files into multiple stream mapped using $M_{Sec,HDT}$, the arrays created during the dictionary creation. Before merging them, we need to take into consideration that a subject or object term can become a shared term leading to our ids in our streams to be unsorted due to the shared IDs representation defined in the triples paragraph in Sect. 2.

To fix that we split our streams into two different streams. A stream with the triples with shared subjects and a stream with the triples with non-shared subjects. Once this is done, we use a merge join to create two streams from all the streams, removing the duplicated terms at the same time.

This solves the unsorted problem for the subject terms, not for the object terms. To sort again the objects, we can use the fact that our triples are sorted by subjects and predicates, so if we peek from our stream all the triples with the same subject and predicate, we can retrieve all the unsorted objects for a given (subject, predicate) couple. This number of objects is usually small compared to the triples count, so we can sort them in memory and stream the sorted results.

The result streams being sorted, we can create our triple component.

Large File Indexing Using k-HDTCat. Using this new algorithm version, we can combine multiple HDT files into one. Removing the overhead added by the generation of the HDT file, but inducing the overhead of reading multiple HDT files at once. We made the hypothesis that this added overhead isn't as big as the one removed from the original HDTCat. This can be used to index big HDT files by first chunking the data and indexing smaller HDT files and then combining them to create a bigger HDT file using k-HDTCat. A comparison with existing methods is available in Sect. 5.2 (Fig. 3).

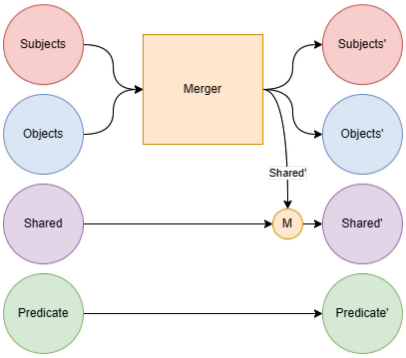


Fig. 3. k-HDTCat Merge streams

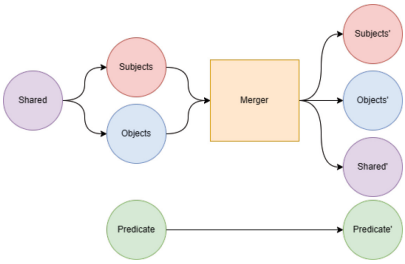


Fig. 4. k-HDTDiffCat Merge streams

4.2 k-HDTDiffCat

The main drawback of HDTCat is the fact that it is only providing the addition on datasets, but not the ability to remove triples from datasets. To do it, one is forced to reindex the dataset without the unwanted triples.

To overcome this missing part, we created k-HDTDiffCat, a modified version of k-HDTCat to remove triples from one or multiple HDT files into one HDT file. It is built on top of k-HDTCat.

Dictionary Generation. Unlike k-HDTCat, during a diff, an RDF term can be removed from the dictionary of a HDT file, leading to 3 cases,

1. The term is still used at least once in the same section, in which case nothing should be done.
2. The term was a shared term, but now it is only an object or a subject, then we need to consider its sharedness loss.
3. The term isn't used anywhere in the dictionary, then we need to completely remove it.

To mark a triple to be deleted, we are using a bitmap per HDT file. If the bit at the index i is set to 1, it means that the triple at the index i will be removed from this HDT file.

To compute the deleted triples, we first create one bitmap per section per HDT file, a bit set to 1 in a bitmap represents if an term is used inside the section. We then fill the bitmaps by reading the triples for each HDT file while ignoring the ones with the same position as a 1 in the triples bitmap for this HDT.

With these bitmaps, when we are reading a section, we only have to check first if this term should be read or ignored.

Due to the property 2, we need to check if a shared element is still shared after the diffcat. To do that, as represented in Fig. 4, we are merging the shared

Data: it_S, it_O two sorted iterators of respectively subjects and objects terms.
Result: it'_S, it'_O, it'_{SH}, three sorted iterators of the final subjects, objects and shared terms.

$it'_S \leftarrow \emptyset$ $it'_O \leftarrow \emptyset$ $it'_{SH} \leftarrow \emptyset$
SharedLoop: **while** $it_S \neq \emptyset$ *and* $it_O \neq \emptyset$ **do**
  $NewSubject \leftarrow \text{Next}(it_S)$ $NewObject \leftarrow \text{Next}(it_O)$
  while $NewSubject \neq NewObject$ **do**
    if $NewSubject < NewObject$ **then**
      $it'_S \leftarrow it'_S \cup \{NewSubject\}$
      if $it_S = \emptyset$ **then**
        break SharedLoop
      $NewSubject \leftarrow \text{Next}(it_S)$
    else
      $it'_O \leftarrow it'_O \cup \{NewObject\}$
      if $it_O = \emptyset$ **then**
        break SharedLoop
      $NewObject \leftarrow \text{Next}(it_O)$
  $it'_{SH} \leftarrow it'_{SH} \cup \{NewSubject\}$
while $it_S \neq \emptyset$ **do**
  $it'_S \leftarrow it'_S \cup \{\text{Next}(it_S)\}$
while $it_O \neq \emptyset$ **do**
  $it'_O \leftarrow it'_O \cup \{\text{Next}(it_O)\}$

Algorithm 1: Split subject/object iterators into subject/object/shared iterators

term iterator it_{SH} with both the subjects it_S and the objects it_O iterators. As described in Algorithm 1, by comparing the two iterators in a merged way, k-HDTDiffCat is able to recompute the shared terms with the subjects/objects when merging the HDT files by comparing when two elements in the iterators are equal.

We then run the k-HDTCat method to compute the dictionary without using it_{SH} at the end because it was already used at the start.

Triples Generation. Like in the dictionary generation, we are reusing the k-HDTCat method to compute the triples, we add to the triple reading and mapping part a check reading the input delete bitmap to see if the read triple is deleted or not. If it is deleted we ignore the triple, otherwise we continue the k-HDTCat method. The k-HDTCat method already taking care of the sharedness "loss" with the same strategy as the sharedness "gain".

Complexity. The k-HDTDiffCat method being close to HDTCat [4], it is reflected with a same time and space complexity. In HDTCat, for n, m the number of triples in the two input HDT files, the time complexity is $O((n + m)\log(n + m))$ with at best $O(n + m)$ and the space complexity is $O(n + m)$. Unlike HDTCat, we don't have a constant amount of input HDT files. We denote the complexities using n, the number of all triples in the input HDT files and we add to the definition m, the number of all RDF terms in the input HDT file dictionaries. We will first explain the time complexity for the triples generation, then the time complexity for the dictionary generation and the space complexity.

The input dictionaries being already sorted, computing the union of them is done using a merge fashion and thus, giving a complexity of $O(m)$.

The triples generation is done by grouping the triples with the same subject-predicate couple and then sort these groups. The worst case scenario is only one group giving a time complexity of $O(n \log n)$. In the best scenario, where all the groups are containing a constant amount of triples, the time complexity is $O(n)$. By combining both the dictionary and triples generation, the time complexity is $O(m + n \log n)$.

The space complexity is given by the space required to store the new HDT file sections, which is $O(m)$ for the dictionary and $O(n)$ for the triples. The combined space complexity is $O(m + n)$.

4.3 HDTq Integration

To integrate HDTq inside these two methods, we need to consider the new graph section and the graph bitmaps inside the triples (or quads) section.

The graph section is compressed like all the other sections and isn't merged with a shared section, so we can apply our merge join with or without the section delete bitmaps depending on if we are deleting triples or not using k-HDTDiffCat.

For the quads, in k-HDTDiffCat, we are already sorting in memory the objects ids by grouping them by (subject, predicate) because the memory impact is low. But we can also notice that the memory impact for grouping the quads ids by (subject, predicate) is also low so we can extend the triples generation of k-HDTCat to sort the graphs with the objects.

The main issue comes with the deletion bitmap, HDTq files not being based on a triple list, but on a quad list, we now consider our delete bitmap as a 2D matrix (i, j) instead of a 1D list (i), the i staying the triple id, but we add the j to denote the graph id. To construct this matrix, like in HDTq [9], we used Roaring bitmaps [15] to avoid having to have full matrix lines with only few bits because named graphs are only containing few triples.

5 Experiments

To evaluate HDTDiffCat we run four different experiments. Their codes are available on Github[4].

1. This experiment aims to determine the best value of k when compressing a dataset using k-HDTCat.
2. In this experiment we compare our compression methods with the existing ones, i.e. *rdf2hdt* [11], *HDT-MR* [12] and HDTCat [4].
3. This experiment compares the compression process we propose with other compression methods, using as an example the Wikidata dataset.
4. This experiment shows how it is possible to use k-HDTDiffCat to create an up to date HDT file of Wikidata with a max delay of 24 h.

[4] https://github.com/ate47/kHDTGenDiskBenchmark.

5.1 Experiment: Experimentally Determine a Good Value of k in k-HDTCat

This first experiment was made using the LUBM benchmark [14]. LUBM is a benchmark for SPARQL that allows to generate synthetically an RDF graph with different sizes. The size is generally indicated by the number of universities contained in the graph. We use a small LUBM dataset of 1000 universities, giving us a dataset with 133.5 million RDF triples. We then split this dataset into small datasets of 1 million triples converted into HDT files, giving us $n = 134$ HDT files. We then run k-HDTCat in a k-way merge fashion to create one HDT file from all the 134 HDT files by using different k. We run our experiments on a KVM virtual machine with 16GB of RAM with 1TB of SSD with 4 vcpu Intel(R) Xeon(R) CPU E5-2667 v4 3.20GHz in a University server. The results are reported in the Table 1.

Table 1. Time in function of the k and the number of layers

k	layers	cats	time (s)	average (s)
2	8	132	2264	2264
4	4	44	1245	1245
8	3	19	987	971
10		14	955	
12	2	12	849	854
20		7	805	
28		5	840	
36		4	890	
44		4	884	
134	1	1	1135	1135

We used 4 different metrics, the number of k-HDTCat (cats) used obtained with $\lceil \frac{n-1}{k-1} \rceil$, the number of layers in the k-ways tree, obtained with $\lceil \log_k(n) \rceil$, the time to create the HDT file for a given k value and the average time for a given layer count. We notice that the time to merge the HDT files can be divided by 2 with a good k choice. We can also see that the time is linked with the number of layers during the k-ways merge, a large number of layers is increasing the time, the results for $k = 10$ and $k = 12$ is also showing that the number k-HDTCat has less impact than the number of layers. We can also see that, after a certain point the higher the number of HDT file is, the slower the method is, the HDT files being mapped from disk, the higher the number is, the higher the number of file pointer is, reducing the speed.

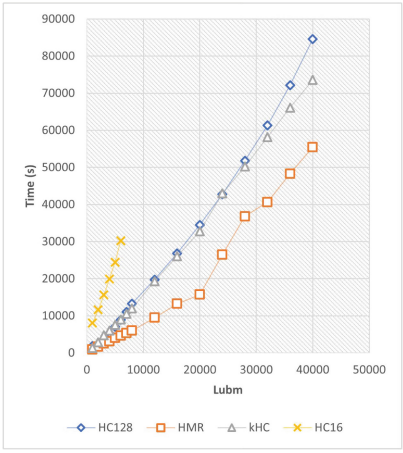
In our future experiments, we will set $k = 20$, this k being the one with the best time.

5.2 Experiment: Comparison with Existing HDT Compression Methods

To compare our approach with existing methods for compressing HDT file, we follow the same procedure as in the HDTCat and HDT-MR papers. The idea is to measure the time to compress different sizes of the data generated in the LUBM benchmark [14]. Following existing evaluations we try to generate an HDT file for the LUBM dataset with a university count of 1000, 2000, 3000, ..., 8000 (i.e. with a step of 1000 universities), then from 8.000, 12.000, 16.000, ..., 40000 (i.e. with a step of 4.000 universities). This gives us different datasets with sizes from 130 million triples to 5.3 billion triples. For the other approaches we do no replicate the results but report the results from the previous papers. We run our experiments on the same machine as the one described in the previous experiment in Sect. 5.1. Each experiment was run 3 times and reported using the average and standard deviation. The results are reported in Table 2. We see that despite using only a 16 GB RAM machine we are outperforming the original HDTCat implementation running on both a machine with 16 GB and 128 GB of RAM. The map reduce setup is still more competitive by 65% but uses a map reduce cluster with 8 times more memory. Overall we can see that we are able to compress big datasets on commodity hardware using only one machine.

Table 2. Comparison in seconds (s) between methods to serialize RDF graphs into HDT using HDTCat [4], HDT-MR [12] and k-HDTCat

LUBM		**128GB**		**16GB**		
Univ.	Triples	HC128	HMR	HC16	kHC Avg	kHC SD
1k	0.13B	1856	936	8082	1305	74
2k	0.27B	2257	1706	11622	2878	36
3k	0.40B	3695	2498	15616	4791	377
4k	0.53B	5285	3113	19928	6041	173
5k	0.67B	7058	4065	24400	7556	168
6k	0.80B	8960	4656	30235	9046	39
7k	0.93B	11018	5338	-	10501	83
8k	1.07B	13308	6020	-	11955	149
12k	1.60B	19777	9499	-	19355	1057
16k	2.14B	26825	13229	-	26078	1480
20k	2.67B	34486	15720	-	32777	1959
24k	3.20B	42789	26492	-	43006	550
28k	3.74B	51807	36812	-	50116	1362
32k	4.27B	61366	40633	-	57736	1727
36k	4.81B	72161	48322	-	65641	1706
40k	5.32B	84633	55471	-	73110	2081



HC: HDTCat, HMR : HDT-MR, kHC : k-HDTCat, Avg : average, SD : standard deviation.

5.3 Experiment: Indexing Wikidata

In this section, we want to show how we compare with respect to indexing strategies of other SPARQL endpoints.

To do this, we want to see how our approach behaves on indexing the whole Wikidata dump. This is seen as a challenging problem. As reported in [7] there is only a limited number of successful attempts to index the whole of Wikidata on existing triple stores since most do not scale to KGs of this size (even with large hardware configurations). In the year 2020 the Wikidata dump surpassed 13B triples. Since then only 6 triple stores have been reported to be capable of indexing the whole dump, namely: Virtuoso [6], Stardog[5], Apache Jena[6], QLever [1] and Blazegraph. We report in Table 3 the loading times, the number of indexed triples, the amount of needed RAM, the final index size and the documentation for indexing Wikidata.

Table 3. Wikidata Index characteristics for different endpoints.

System	Loading Time	#Triples	RAM	Index size	Doc
Apache Jena	9d 21h	13.8 B	64 GB	2TB	https://wiki.bitplan.com/index.php/WikiData_Import_2020-08-15.
Virtuoso	"several days"[a] (preprocessing) + 10 h	11.9 B	378 GB	NA	https://community.openlinksw.com/t/loading-wikidata-into-virtuoso-open-source-or-enterprise-edition/2717.
Blazegraph	∼5.5d	11.9 B	128 GB	1.1 T	https://addshore.com/2019/10/your-own-wikidata-query-service-with-no-limits/.
Stardog	9.5 h	16.7 B	256 GB	NA	https://www.stardog.com/labs/blog/wikidata-in-stardog/.
QLever	14.3 h	17 B	128 GB	823 GB	https://github.com/ad-freiburg/qlever/wiki/Using-QLever-for-Wikidata.
qEndpoint	57 h	**19.2 B**	16 GB	**340 GB**	https://github.com/the-qa-company/qEndpoint/wiki/Use-qEndpoint-to-index-a-dataset.

[a] The precise number of days for the preprocessing is not indicated in the documentation.

Using k-HDTCat we were able to index the Wikidata public dump from February 2024 with 19.2 billion triples[7]. The experiment was run on a 16 GB desktop computer with a 12 cores AMD Ryzen 5 5600 3.50 GHz CPU and 1 TB of SSD. The experiment was repeated two times, the average time and standard deviation are used as a metric. The compression time was 52 h. The compression time is the third best between the reported once. On the other hand, this compression was achieved on a commodity hardware with 16 GB of RAM which is from 8 to 23 smaller than other approaches. The final index size of 340 GB is the smallest between the reported ones.

We measured the total time for the different generation steps. The k-HDTCat generation steps are *dict* for merging and writing the dictionaries, *triples* for

[5] https://www.stardog.com/.
[6] https://jena.apache.org.
[7] https://dumps.wikimedia.org/wikidatawiki/entities/.

merging and writing the triples and *save* to write the end HDT file. This results are reported in Table 4.

Table 4. Time split per part during WD compression

Method	Section	Time (s)	Time SD	Percent.	Total (s)	Total SD	Percent.
HDT file Generation	–				104449	1124	59,68 %
k-HDTCat	dict	18964	478	26.87 %	70582	1498	40,32 %
	triples	42767	1092	60.6 %			
	save	8837	73	12.52 %			
Total	–				175032	378	100 %

SD = Standard deviation

We can see that most of the time is used to compress the small HDT files.

5.4 Experiment: k-HDTDiffCat to Keep Up to Date Datasets

Table 5. Times per part to compute HDTDiff over a Wikidata Truthy HDT file with 371M changed triples.

Step	Diff only time (s)	Cat only time (s)	DiffCat time (s)
Diff bitmap	501	–	462
Dictionary	2121	2155	2206
Triples	4446	3401	4352
Total	7068	5556	7020

A way to keep RDF graphs up to date, is by using buffer updates, i.e. adding and deleting the changes over time in a buffer and by applying all of them at the same time. In this experiment we are using the Wikidata recent changes API[8] to fetch the recent changes from the Wikidata dataset [20] and combine them with an HDT dump of the Wikidata dataset.

For our experiment, we are using a Wikidata truthy dump from 2023-10-28 00:40 (CET) and the recent changes changes from the dump date to the 2023-11-06 14:00. This accounts for $2,509,863$ Wikidata entity changes. We create a bitmap for the HDT triples, when an entity e is changed, we mark as removed by a 1 in the bitmap all the triples matching the pattern $(e, ?, ?)$. We put in a new HDT file the triples with e as a subject by getting them from Wikidata. This gives us 371.278.906 added triples inside what we call the "Delta" HDT file.

[8] https://www.mediawiki.org/wiki/API:RecentChanges.

Table 6. Times (s) per part to compute k-HDTDiffCat over a Wikidata Truthy HDT file with 371M changed triples using different counts. The added time is compared with the time to compute a k-HDTDiffCat with 2 HDT files.

Step	1	2	3	4	5	6	7
Diff bitmap	581	553	530	525	550	552	548
Dictionary	2873	3006	3219	3371	3615	3739	3959
Triples	6157	6416	6863	7149	7661	8083	8161
Total	9611	9975	10612	11045	11826	12374	12668
Added time	–	+4%	+10%	+14%	+23%	+28%	+32%

Using these datasets, we did four experiments to test the efficiency of k-HDTDiffCat, they were done on a 16GB laptop with 200GB of SSD available.

The first one was to only apply the HDTDiff part on the Truthy Wikidata HDT file with the delete bitmap. This allows to see the speed of only the diff part. A second test is made by running the HDTCat part on the HDTDiff result with the delta graph. Giving a time on how fast the HDTCat part is working, it is not made on the first Truthy Wikidata HDT file to avoid duplicated terms. The third is to run both algorithms to compare how fast is the couple k-HDTDiffCat vs HDTCat + HDTDiff. The results of these experiments are available in Table 5. In the table, the time to compute each HDT section is mentioned with the total time to run the method, the diff bitmap building is omited in the cat because we aren't running a diff method.

The results of these three experiments are available in Table 5. We can see that HDTDiff is adding an overhead to HDTCat, we believe that it is due to the bitmap reading to test if an term is in the result HDT file. Another result is the time difference between HDTDiff only + HDTCat only and k-HDTDiffCat, showing that it is pertinent to use k-HDTDiffCat instead of HDTDiff and then HDTCat when building a new HDT file.

A fourth experiment is done to understand how the increasing count of HDT files is affecting our method. To do that, we first split our delta HDT file into 7 pieces and we run the k-HDTDiffCat on the truthy HDT file and i pieces, with $i = \{1, \dots 7\}$. The results of this experiment are available in Table 6. The diff being only done on the input HDT file, the time to compute the diff bitmap is the same for all the steps. We then notice an increase in time for each added HDT file between 4 and 9 percents added to the time to compute two k-HDTDiffCat.

6 Code

k-HDTDiffCat and its HDTq support are part of the qEndpoint core repository[9]. A fork of the Java HDT library[10]. It is used in qEndpoint, a SPARQL endpoint

[9] https://github.com/the-qa-company/qEndpoint/tree/master/qendpoint-core.
[10] https://github.com/rdfhdt/hdt-java.

based on HDT and RDF4J. The code is released under the *Lesser General Public License*. The tool is available as a command line interface[11] together with existing tools like rdf2hdt, hdt2rdf, hdtInfo, hdtVerify. The new tool can be used as:

```
hdtDiffCat <input files> -diff <delta file> <output HDT file>
```

The input files are the HDT files to combine, an additional HDT file can be added to tell which triples to delete.

It can also be used as a lightweight library without the endpoint for more specific operations.

7 Conclusion

HDT is an RDF file format that allows to store, share and query large RDF Knowledge Graphs on commodity hardware. In this paper, we presented a new tool, k-HDTDiffCat, a generalization of HDTCat to combine multiple HDT files and to remove some of the existing triples.

It is available online as a command line interface or by using our Java library in the qEndpoint repository[12] under the *Lesser General Public License*. We have shown how this tool can be used to compress HDT file and how it compares with respect to existing methods. We have also compared the HDT compression with the indexing strategies of other performant SPARQL endpoints. Finally, we have seen how they can be used to maintain an up-to-date Wikidata index. To conclude, k-HDTDiffCat allows generating and update HDT files on commodity hardware, something that was not possible before. This is particularly important since now all operations in the HDT life-cycle can be carried out on commodity hardware. As a consequence HDT becomes an ideal indexing structure for large KGs in low hardware settings allowing the community to manipulate these graphs more easily.

References

1. Bast, H., Buchhold, B.: QLever: a query engine for efficient SPARQL+ text search. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, pp. 647–656 (2017)
2. Beek, W., Rietveld, L., Bazoobandi, H.R., Wielemaker, J., Schlobach, S.: LOD laundromat: a uniform way of publishing other people's dirty data. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 213–228. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_14
3. Diefenbach, D., Both, A., Singh, K., Maret, P.: Towards a question answering system over the semantic web. Semant. Web **11**(3), 421–439 (2020)

[11] https://github.com/the-qa-company/qEndpoint/wiki/qEndpoint-CLI-commands.
[12] https://github.com/the-qa-company/qEndpoint/tree/master/qendpoint-core.

4. Diefenbach, D., Giménez-García, J.M.: HDTCat: let's make HDT generation scale. In: Pan, J.Z., et al. (eds.) ISWC 2020. LNCS, vol. 12507, pp. 18–33. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62466-8_2
5. Diefenbach, D., Singh, K., Maret, P.: WDAqua-core1: a question answering service for rdf knowledge bases. In: Companion Proceedings of the The Web Conference 2018, WWW 2018, pp. 1087–1091. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE (2018)https://doi.org/10.1145/3184558.3191541
6. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: Pellegrini, T., Auer, S., Tochtermann, K., Schaffert, S. (eds.) Networked Knowledge - Networked Media. Studies in Computational Intelligence, vol. 221, pp. 7–24. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02184-8_2
7. Fahl, W., Holzheim, T., Westerinen, A., Lange, C., Decker, S.: Getting and hosting your own copy of Wikidata. In: 3rd Wikidata Workshop @ International Semantic Web Conference (2022). https://zenodo.org/record/7185889#.Y0WD1y0RoQ0
8. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: LOD-a-lot: a queryable dump of the LOD cloud. In: d'Amato, C., et al. (eds.) ISWC 2017, Part II. LNCS, vol. 10588, pp. 75–83. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_7
9. Fernández, J.D., Martínez-Prieto, M.A., Polleres, A., Reindorf, J.: HDTQ: managing RDF datasets in compressed space. In: Gangemi, A., et al. (eds.) ESWC 2018. LNCS, vol. 10843, pp. 191–208. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93417-4_13
10. 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 (2013). https://doi.org/10.1016/j.websem.2013.01.002, https://www.sciencedirect.com/science/article/pii/S1570826813000036
11. Fernández García, J.D., et al.: Binary RDF for scalable publishing, exchanging and consumption in the web of data (2013)
12. 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
13. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_28
14. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for owl knowledge base systems. J. Web Semant. **3**(2–3), 158–182 (2005)
15. Lemire, D., et al.: Roaring bitmaps: implementation of an optimized software library. Softw.: Pract. Exp. **48**(4), 867–895 (2018)
16. Martínez-Prieto, M.A., Fernández, J.D., Cánovas, R.: Compression of rdf dictionaries. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC 2012, pp. 340–347. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2245276.2245343
17. Taelman, R., Mahieu, T., Vanbrabant, M., Verborgh, R.: Optimizing storage of RDF archives using bidirectional delta chains. Semant. Web **13**(4), 705–734 (2022)
18. Verborgh, R., et al.: Triple pattern fragments: a low-cost knowledge graph interface for the web. J. Web Semant. **37–38**, 184–206 (2016). https://doi.org/10.1016/j.websem.2016.03.003, http://linkeddatafragments.org/publications/jws2016.pdf

19. Willerval, A., Bonifati, A., Diefenbach, D.: qEndpoint: a Wikidata SPARQL endpoint on commodity hardware. In: Companion Proceedings of the ACM Web Conference 2023, pp. 119–122 (2023)
20. Willerval, A., Diefenbach, D., Maret, P.: Easily setting up a local Wikidata SPARQL endpoint using the qEndpoint (2022)
21. Witten, I.H., et al.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann (1999)