HDT-MR: A Scalable Solution for RDF Compression with HDT and MapReduce

José M. Giménez-García[1], Javier D. Fernández[2], and Miguel A. Martínez-Prieto[1]

[1] DataWeb Research, Department of Computer Science, Univ. de Valladolid (Spain)
josemiguel.gimenez@alumnos.uva.es, migumar2@infor.uva.es
[2] Vienna University of Economics and Business (Austria)
jfernand@wu.ac.at

Abstract. HDT a is binary RDF serialization aiming at minimizing the space overheads of traditional RDF formats, while providing retrieval features in compressed space. Several HDT-based applications, such as the recent *Linked Data Fragments* proposal, leverage these features for diverse publication, interchange and consumption purposes. However, scalability issues emerge in HDT construction because the whole RDF dataset must be processed in a memory-consuming task. This is hindering the evolution of novel applications and techniques at Web scale. This paper introduces HDT-MR, a MapReduce-based technique to process huge RDF and build the HDT serialization. HDT-MR performs in linear time with the dataset size and has proven able to serialize datasets up to several billion triples, preserving HDT compression and retrieval features.

1 Introduction

The *Resource Description Framework* (RDF) was originally proposed as a data model for describing resources in the Web [12], and has evolved into a standard for data interchange in the emergent Web of (Linked) Data. RDF has been widely used in the last years, specially under the *Linked Open Data* initiative, where it shows its potential for integrating non-structured and semi-structured data from several sources and many varied fields of knowledge. This flexibility is obtained by structuring information as triples: (i) the *subject* is the resource being described; (ii) the *predicate* gives a property about the resource; and (iii) the *object* sets the value of the description. A set of RDF triples is a labeled directed graph, with subjects and objects as nodes, and predicates as edges.

This "graph view" is a mental model that helps to understand how information is organized in RDF, but triples must be effectively serialized in some way for storage and/or interchange. The World Wide Web Consortium (W3C) Working Group addresses this need in the last RDF Primer proposal[3]. The considered RDF serialization formats (JSON-LD, RDF/XML or Turtle-based ones) provide different ways of writing down RDF triples, yet all of them serialize an RDF graph as plain text. This is a double-edged sword. On the one hand,

[3] http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624

serialization is an easy task with no much processing overhead. On the other hand, the resulting serialized files tend to be voluminous because of the verbosity underlying to these formats. Although any kind of universal compressor (*e.g.* gzip) reduces space requirements for RDF storage and interchange purposes [6], space overheads remain a problem when triples are decompressed for consumption (parsing, searching, etc.). This situation is even more worrying because end-users have, in general, less computational resources than publishers.

HDT (*Header-Dictionary-Triples*) is an effective alternative for RDF serialization. It is a binary format which reorganizes RDF triples in two main components. The *Dictionary* organizes all terms used in triples and maps them to numerical identifiers. This decision allows the original graph to be transformed into a graph of IDs encoded by the *Triples* component. Built-in indexes, in both components, allow RDF triples to be randomly retrieved in compressed space. In other words, HDT outputs more compact files than the aforementioned formats and also enables RDF triples to be efficiently accessed without prior decompression [13]. This fact makes HDT an ideal choice to play as storage engine within semantic applications. *HDT-FoQ* [13] illustrates how HDT can be used for efficient triple pattern and SPARQL join resolution, while *WaterFowl* [4] goes a step further and provides inference on top of *HDT* foundations. This notion of HDT-based store is deployed in applications such as *Linked Data Fragments* [18], the *SemStim* recommendation system [8] or the Android app *HDTourist* [9].

Nevertheless, these achievements are at the price of moving scalability issues to the publishers, or data providers in general. Serializing RDF into HDT is not as simple as with plain formats, given that the whole dataset must be exhaustively processed to obtain the *Dictionary* and *Triples* components. Current HDT implementations demand not negligible amounts of memory, so the HDT serialization lacks of scalability for huge datasets (*e.g.* those having hundreds of millions or billions of triples). Although these datasets are currently uncommon, semantic publication efforts on emerging data-intensive areas (such as biology or astronomy) or integrating several sources into heterogeneous mashups (as RDF excels at linking data from diverse datasets) are starting to face this challenge.

This paper improves the HDT workflow by introducing MapReduce [5] as the computation model for large HDT serialization. MapReduce is a framework for the distributed processing of large amounts of data, and it can be considered as *de facto* standard for Big Data processing. Our MapReduce-based approach, HDT-MR, reduces scalability issues arising to HDT generation, enabling larger datasets to be serialized for end-user consumption. We perform evaluations scaling up to 5.32 billion triples (10 times larger than the largest dataset serialized by the original HDT), reporting linear processing times to the dataset size. This states that HDT-MR provides serialization for RDF datasets of arbitrary size while preserving both the HDT compression and retrieval features [13, 6].

The rest of the paper is organized as follows. Section 2 summarizes the background required to understand our approach, which is fully described in Section 3. Section 4 reports experimental results about HDT-MR. Finally, Section 5 concludes about HDT-MR and devises some future work around it.

2 Background

This section provides background to understand our current approach. We give basic notions about MapReduce and explain HDT foundations. Then, we compare HDT to the current state of the art of RDF compression.

2.1 MapReduce

MapReduce [5] is a framework and programming model to process large amounts of data in a distributed way. Its main purpose is to provide efficient parallelization while abstracting the complexity of distributed processing. MapReduce is not schema-dependent; unstructured and semi-structured can be processed, at the price of parsing every item [11]. A MapReduce job comprises two phases. The first phase, `map`, reads the data as pairs key-value $(k1, v1)$ and outputs another series of pairs key-value of different domain $(k2, v2)$. The second phase, `reduce`, processes the list of values $v2$, related to each key $k2$, and produces a final list of output values $v2$ pertaining to the same domain. Many tasks are launched on each phase, all of them processing a small piece of the input data. The following scheme illustrates input and output data to be processed in each phase:

$$\begin{array}{ll} \text{map:} & (k1, v1) \rightarrow list(k2, v2) \\ \text{reduce:} & (k2, list(v2)) \rightarrow list(v2) \end{array}$$

MapReduce relies on a master/slave architecture. The *master* initializes the process, distributes the workload among the cluster and manages all bookkeeping information. The *slaves* (or *workers*) run `map` and `reduce` tasks. The workers commonly store the data using a distributed filesystem based on the GFS *(Google File System)* model, where data are split in small pieces and stored in different nodes. This allows workers to leverage *data locality* as much as possible, reading data from the same machine where the task runs [5]. MapReduce performs exhaustive I/O operations. The input of every task is read from disk, and the output is also written on disk. It is also intensive in bandwidth usage. The `map` output must be transferred to `reduce` nodes and, even if most of the `map` tasks read their data locally, part of them must be gathered from other nodes.

Apache Hadoop[4] is currently the most used implementation of MapReduce. It is designed to work in heterogeneous clusters of commodity hardware. Hadoop implements HDFS *(Hadoop Distributed File System)*, as distributed filesystem providing data replication. It replicates each split of data in a number of nodes (commonly three), improving data locality and also providing fault tolerance.

2.2 HDT

HDT[5] [6] is a binary serialization format optimized for RDF storage and transmission. Besides, HDT files can be mapped to a configuration of succinct data structures which allows the inner triples to be searched and browsed efficiently.

[4] `http://hadoop.apache.org/`

[5] HDT is a *W3C Member Submission*: `http://www.w3.org/Submission/HDT/`

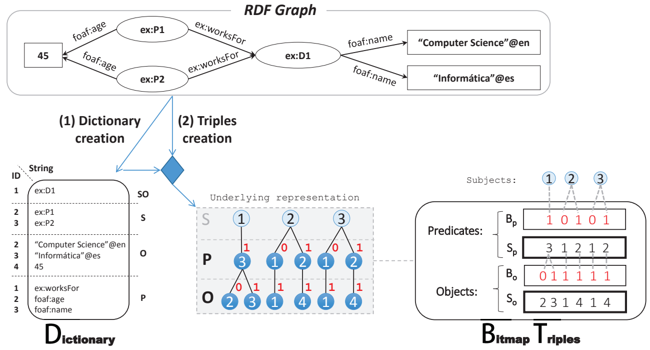

Fig. 1. HDT *Dictionary* and *Triples* configuration for an RDF graph.

HDT encodes RDF into three components carefully described to address RDF peculiarities within a *Publication-Interchange-Consumption* workflow. The *Header* (**H**) holds the dataset metadata, including relevant information for discovering and parsing, hence serving as an entry point for consumption. The *Dictionary* (**D**) is a catalogue that encodes all the different terms used in the dataset and maps each of them to a unique identifier: ID. The *Triples* (**T**) component encodes the RDF graph as a graph of IDs, *i.e.* representing tuples of three IDs. Thus, *Dictionary* and *Triples* address the main goal of RDF compactness. Figure 1 shows how the *Dictionary* and *Triples* components are configured for a simple RDF graph. Each component is detailed below.

Dictionary. This component organizes the different terms in the graph according to their role in the dataset. Thus, four sections are considered: the section **SO** manages those terms playing both as subject and object, and maps them to the range `[1, |SO|]`, being `|SO|` the number of different terms acting as subject and object. Sections **S** and **O** comprise terms that exclusively play subject and object roles respectively. Both sections are mapped from `|SO|+1`, ranging up to `|SO|+|S|` and `|SO|+|O|` respectively, where `|S|` and `|O|` are the number of exclusive subjects and objects. Finally, section **P** organizes all predicate terms, which are mapped to the range `[1, |P|]`. It is worth noting that no ambiguity is possible once we know the role played by the corresponding ID.

Each section of the *Dictionary* is independently encoded to grasp its particular features. This allows important space savings to be achieved by considering that this sort of string dictionaries are highly compressible [14]. Nonetheless, efficient encoding of string dictionaries [2] is orthogonal to the current problem, hence it is not addressed in this paper.

Triples. This component encodes the structure of the RDF graph after ID substitution. That is, RDF triples are encoded as groups of three IDs (ID-triples hereinafter): $(\mathtt{id}_s\ \mathtt{id}_p\ \mathtt{id}_o)$, where $\mathtt{id}_s$, $\mathtt{id}_p$, and $\mathtt{id}_o$ are respectively the IDs of the corresponding subject, predicate, and object terms in the *Dictionary*. The

Triples component organizes all triples into a forest of trees, one per different subject: the subject is the root; the middle level comprises the ordered list of predicates reachable from the corresponding subject; and the leaves list the object IDs related to each (subject, predicate) pair. This *underlying representation* (illustrated in Figure 1) is effectively encoded following the *BitmapTriples* approach [6]. In brief, it comprises *two sequences*: `Sp` and `So`, concatenating respectively all predicate IDs in the middle level and all object IDs in the leaves; and *two bitsequences*: `Bp` and `Bo`, which are respectively aligned with `Sp` and `So`, using a 1-bit to mark the end of each list.

Building HDT. Once *Dictionary* and *Triples* internals have been described, we proceed to summarize how HDT is currently built[6]. Remind that this process is the main scalability bottleneck addressed by our current proposal.

To date, HDT serialization can be seen as a three-stage process:

- **Classifying RDF terms.** This first stage performs a triple-by-triple parsing (from the input dataset file) to classify each RDF term into the corresponding *Dictionary* section. To do so, it keeps a temporal data structure, consisting of three hash tables storing subject-to-ID, predicate-to-ID, and object-to-ID mappings. For each parsed triple, its subject, predicate, and object are searched in the appropriate hash, obtaining the associated ID if present. Terms not found are inserted and assigned an auto-incremental ID. These IDs are used to obtain the temporal ID-triples ($\mathtt{id}_s$ $\mathtt{id}_p$ $\mathtt{id}_o$) representation of each parsed triple, storing all them in a temporary ID-triples array. At the end of the file parsing, subject and object hashes are processed to identify terms playing both roles. These are deleted from their original hash tables and inserted into a fourth hash comprising terms in the SO section.
- **Building HDT *Dictionary*.** Each dictionary section is now sorted lexicographically, because prefix-based encoding is a well-suited choice for compressing string dictionaries [2]. Finally, an auxiliary array coordinates the previous temporal ID and the definitive ID after the *Dictionary* sorting.
- **Building HDT *Triples*.** This final stage scans the temporary array storing ID-triples. For each triple, its three IDs are replaced by their definitive IDs in the newly created *Dictionary*. Once updated, ID-triples are sorted by subject, predicate and object IDs to obtain the *BitmapTriples* streams. In practice, it is a straightforward task which scans the array to sequentially extract the predicates and objects into the `Sp` and `So` sequences, and denoting list endings with 1-bits in the bitsequences.

2.3 Related work

HDT was designed as a binary serialization format, but the optimized encodings achieved by *Dictionary* and *Triples* components make HDT also excels as RDF compressor. Attending to the taxonomy from [16], HDT is a *syntactic* compressor

[6] HDT implementations are available at `http://www.rdfhdt.org/development/`

because it detects redundancy at serialization level. That is, the *Dictionary* reduces symbolic redundancy from the terms used in the dataset, while the *Triples* component leverages structural redundancy from the graph topology.

To the best of our knowledge, the best space savings are reported by syntactic compressors. Among them, k^2-triples [1] is the most effective approach. It performs a predicate-based partition of the dataset into subsets of pairs (subject, object), which are then encoded as sparse binary matrices (providing direct access to the compressed triples). k^2-triples achievements, though, are at the cost of exhaustive time-demanding compression processes that also need large amounts of main memory. On the other hand, *logical* compressors perform discarding triples that can be inferred from others. Thus, they achieve compression because only encode a "primitive" subset of the original dataset. Joshi *et al.* [10] propose a technique which prunes more than 50% of the triples, but it does not achieve competitive numbers regarding HDT, and its compression process also reports longer times. More recently, Pan, *et al.* [16] propose an hybrid compressor leveraging syntactic and semantic redundancy. Its space numbers slightly improves the less-compressed HDT configurations, but it is far from k^2-triples. It also shows non-negligible compression times for all reported datasets.

Thus, the most prominent RDF compressors experience lack of scalability when compressing large RDF datasets. This issue has already been addressed by using distributed computation. Urbani *et al.* [17] propose an algorithm based on dictionary encoding. They perform a MapReduce job to create the dictionary, where an ID is assigned to each term. The output of this job are key-value pairs, where the key is the ID and the value contains the triple identifier to which the term belongs, and its role on it. Then, another MapReduce job groups by triple and substitutes the terms by their ID. This work makes special emphasis on how RDF skewness can affect MapReduce performance, due to the fact that many terms can be grouped and sent to the same reducer. To avoid this problem, a first job is added, where the input data are sampled and the more popular terms are given their ID before the process starts. Finally, Cheng *et al.* [3] also perform distributed RDF compression on dictionary encoding. They use the parallel language X10, and report competitive results.

3 HDT-MR

This section describes HDT-MR, our MapReduce-based approach to serialize large RDF datasets in HDT. Figure 2 illustrates the HDT-MR workflow, consisting in two stages: *(1) Dictionary Encoding* (top) and *(2) Triples Encoding* (bottom), described in the following subsections. The whole process assumes the original RDF dataset is encoded in N-Triples format (one statement per line).

3.1 Process 1: Dictionary Encoding

This first process builds the HDT *Dictionary* from the original N-Triples dataset. It can be seen as a three-task process of (i) identifying the role of each term in

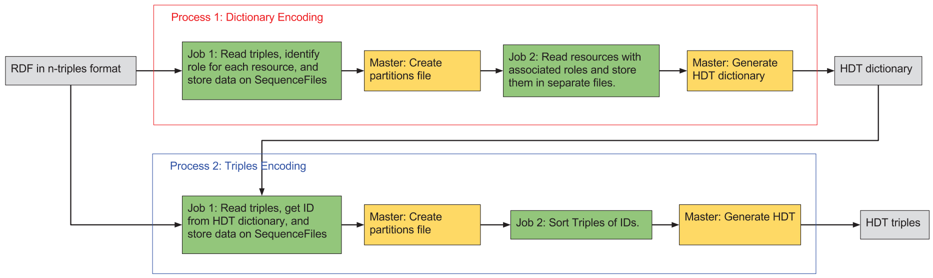

Fig. 2. HDT-MR workflow.

the dataset, (ii) obtaining the aforementioned sections (**SO**, **S**, **O**, and **P**) in lexicographic order, and (iii) effectively encoding the *Dictionary* component.

We design HDT-MR to perform these three tasks as two distributed MapReduce jobs and a subsequent local process (performed by the *master* node), as shown in Figure 2. The first job performs the role identification, while the second is needed to perform a global sort. Finally, the *master* effectively encodes the *Dictionary* component. All these sub-processes are further described below.

Job 1.1: Roles Detection. This job parses the input N-Triples file to detect all roles played by RDF terms in the dataset. First, mappers perform a triple-by-triple parsing and output (key,value) pairs of the form `(RDF term, role)`, in which role is S (subject), P (predicate) or O (object), according to the term position in the triple. It is illustrated in Figure 3, with two processing nodes performing on the RDF used in Figure 1. For instance, `(ex:P1,`S`)`, `(ex:worksFor,`P`)`, and `(ex:D1,`O`)` are the pairs obtained for the triple `(ex:P1, ex:worksFor, ex:D1)`.

These pairs are partitioned and sorted among the reducers, which group the different roles played by a term. Note that RDF terms including roles S and O, result in pairs `(RDF term, `SO`)`. Thus, this job outputs a number of lexicographically ordered lists `(RDF term, roles)`; there will be as many lists as reducers on the cluster. Algorithm 1 shows the pseudo-code of these jobs.

Finally, it is important to mention that a *combiner* function is used at the output of each `map`. This function is executed on each node node before the `map` transmits its output to the reducers. In our case, if a mapper emits more than one pair `(RDF term, role)` for a term, all those pairs are grouped into a single one comprising a list of all roles. It allows the bandwidth usage to be decreased by grouping pairs with the same key before transferring them to the reducer.

Job 1.2: RDF Terms Sectioning. The previous job outputs several lists of pairs `(RDF term, roles)`, one per `reduce` of previous phase, each of them sorted lexicographically. However, the construction of each HDT *Dictionary* section requires a unique sorted list. Note that a simple concatenation of the output lists would not fulfill this requirement, because the resulting list would not maintain

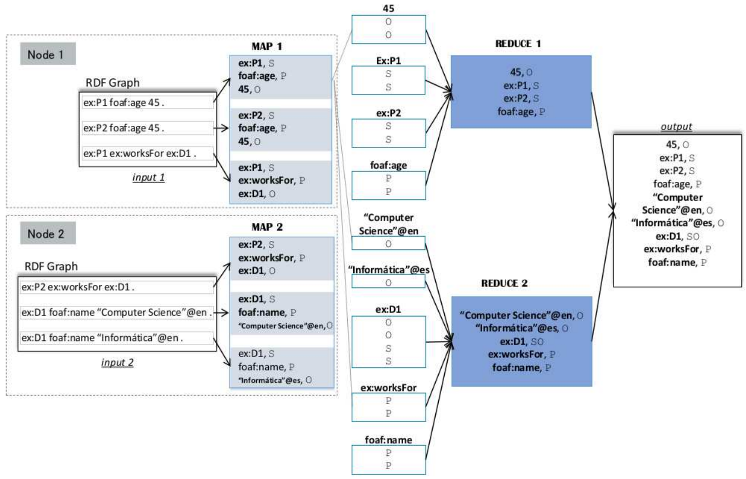

Fig. 3. Example of Dictionary Encoding: roles detection (Job 1.1).

Algorithm 1 Dictionary Encoding: roles detection (Job 1.1)

---

function MAP(key,value) ▷ key: line number (discarded) ▷ value: triple
  $emit(value.subject, "S")$
  $emit(value.predicate, "P")$
  $emit(value.object, "O")$
end function
function COMBINE/REDUCE(key,values) ▷ key: RDF term ▷ value: roles (S, P, and/or O)
  for role in values **do**
    if role contains "S" **then** $isSubject \leftarrow true$
    else if role contains "P" **then** $isPredicate \leftarrow true$
    else if role contains "O" **then** $isObject \leftarrow true$
    end if
  end for
  $roles \leftarrow ""$
  if isSubject **then** $append(roles, "S")$
  else if isPredicate **then** $append(roles, "P")$
  else if isObject **then** $append(roles, "O")$
  end if
  $emit(key, roles)$
end function

---

a global order. The reason behind this behavior is that, although the input of each reducer is sorted before processing, the particular input transmitted to each reducer is autonomously decided by the framework in a process called *partitioning*. By default, Hadoop *hashes* the key and assigns it to a given reducer, promoting to obtain partitions of similar sizes. Thus, this distribution does not respect a global order of the input. While this behavior may be changed to assign the reducers a globally sorted input, this is not straightforward.

A naïve approach would be to use a single reducer, but this would result extremely inefficient: the whole data had to be processed by a single machine, losing most of the benefits of distributed computing that MapReduce provides. Another approach is to manually create partition groups. For instance, we could send terms beginning with the letters from *a* to *c* to the first reducer, terms

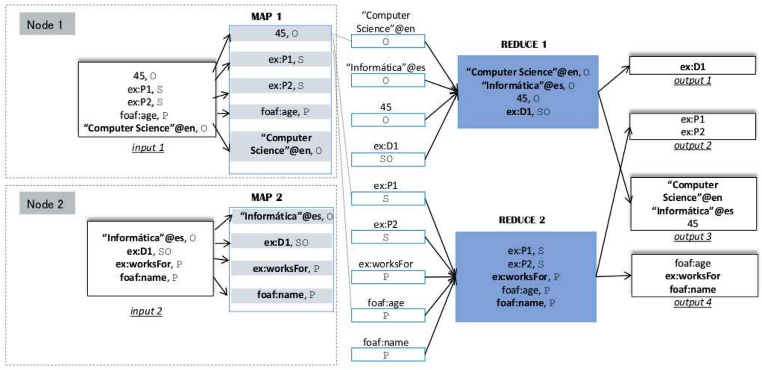

Fig. 4. Example of Dictionary Encoding: RDF terms sectioning (Job 1.2).

Algorithm 2 Dictionary Encoding: RDF terms sectioning (Job 1.2)

function REDUCE(key,value) ▷ key: RDF term ▷ value: roles (S, P, and/or O)
  for resource in values **do**
    if resource contains "S" **then** $isSubject \leftarrow true$
    else if resource contains "P" **then** $isPredicate \leftarrow true$
    else if resource contains "O" **then** $isObject \leftarrow true$
    end if
  end for
  $output \leftarrow$ ""
  if isSubject & isObject **then** $emit\_to\_SO(key, null)$
  else if $isSubject$ **then** $emit\_to\_S(key, null)$
  else if $isPredicate$ **then** $emit\_to\_P(key, null)$
  else if $isObject$ **then** $emit\_to\_O(key, null)$
  end if
end function

beginning with the letters from d to f to the second reducer, and so on. However, partitions must be chosen with care, or they could be the root of performance issues: if partitions are of very different size, the job time will be dominated by the slowest reducer (that is, the reducer that receives the largest input). This fact is specially significant for RDF processing because of its skewed features.

HDT-MR relies on the simple but efficient solution of sampling input data to obtain partitions of similar size. To do so, we make use of the *TotalOrderPartitioner* of Hadoop. It is important to note that this partitioning cannot be performed while processing a job, but needs to be completed prior of a job execution. Note also that the input domain of the reducers needs to be different from the input domain of the job to identify and group the RDF terms (that is, the job receives triples, while the reducers receive individual terms and roles).

All these reasons conforms the main motivation to include this second MapReduce job to globally sort the output of the first job. This job takes as input the lists of `(RDF term, roles)` obtained in the precedent job, and uses role values to sort each term in its corresponding list. In this case, identity mappers deliver directly their input (with no processing) to the reducers, which send RDF terms to different outputs depending on their role. Figure 4 illustrates this job. As only

Algorithm 3 Triples Encoding: ID-triples serialization (Job 2.1)

function MAP(key,value) ▷ key: line number (discarded) ▷ value: triple
    $emit(value.subject, dictionary.id(value.subject)$
    $emit(value.predicate, dictionary.id(value.predicate))$
    $emit(value.object, dictionary.id(value.object))$
end function

the term is needed, a pair `(RDF term, null)` is emitted for each RDF term (*nulls* are omitted on the outputs). We obtain as many role-based lists as reducers in the cluster, but these are finally concatenated to obtain four sorted files, one per *Dictionary* section.The pseudo-code for this job is described in Algorithm 2.

Local sub-process 1.3: HDT Dictionary Encoding This final stage performs locally in the *master* node, encoding dictionaries for the four sections obtained from the MapReduce jobs. It means that each section is read line-per-line, and each term is differentially encoded to obtain a Front-Coding dictionary [2], providing term-ID mappings. It is a simple process with no scalability issues.

3.2 Process 2: Triples Encoding

This second process parses the original N-Triples dataset to obtain, in this case, the HDT *Triples* component. The main tasks for such *Triples* encoding are (i) replacing RDF terms by their ID in the *Dictionary*, and (ii) getting the ID-triples encoding sorted by subject, predicate and object IDs. As in the previous process, HDT-MR accomplishes these tasks by two MapReduce jobs and a final local process (see the global overview in Figure 2), further described below.

Job 2.1: ID-triples serialization This first job replaces each term by its ID. To do so, HDT-MR first transmits and loads the –already compressed and functional– *Dictionary* (encoded in the previous stage) in all nodes of the cluster. Then, mappers parse N-Triples and replace each term by its ID in the *Dictionary*. Identity reducers simply sort incoming data and output a list of pairs `(ID-triple, null)`. We can see this process in action in Figure 5, where the terms of each triple are replaced by the IDs given in the previous example (note that *nulls* are omitted on the outputs). The output of this job is a set of lexicographically ordered lists of ID-Triples; there will be as many lists as reducers on the cluster. The pseudo-code of this job is illustrated in Algorithm 3 .

Job 2.2: ID-triples Sorting Similarly to the first process, Triples Encoding requires of a second job to sort the outputs. Based on the same premises, HDT-MR makes use of Hadoop *TotalOrderPartitioner* to sample the output data from the first job, creating partitions of a similar size as input for the second job. Then, this job reads the ID-triples representation generated and sorts it by subject, predicate and object ID. This is a very simple job that uses identity mappers

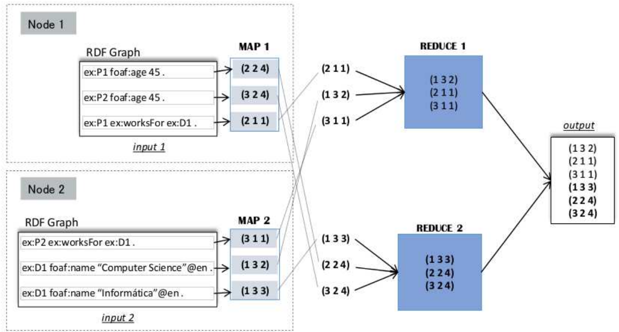

Fig. 5. Example of Triples Encoding: ID-triples Serialization (Job 2.1).

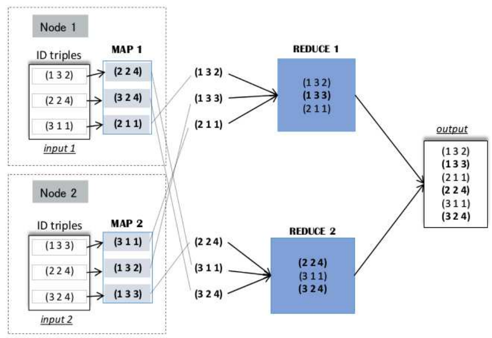

Fig. 6. Example of Triples Encoding: ID-triples Sorting (Job 2.2)

and reducers. As in the previous job, ID-triples are contained in the key and the value is set to *null*. In fact, all the logic is performed by the framework in the partitioning phase between `map` and `reduce`, generating similar size partitions of globally sorted data. Figure 6 continues with the running example and shows the actions performed by this job after receiving the output of the previous job (note again that *nulls* are omitted on the outputs).

Local sub-process 2.3: HDT Triples Encoding This final stage encodes the ID-triples list (generated by the previous job) as HDT *BitmapTriples* [6]. It is performed locally in the *master* node as in the original HDT construction. That is, it sequentially reads the sorted ID-triples to build the sequences `Sp` and `So`, and the aligned bitsequences `Bp` and `Bo`, with no scalability issues.

4 Experimental Evaluation

This section evaluates the performance of HDT-MR, the proposed MapReduce-based HDT construction, and compares it to the traditional single-node ap-

Machine	Configuration
Single Node	Intel Xeon E5-2650v2 @ 2.60GHz (32 cores), 128GB RAM. Debian 7.8
Master	Intel Xeon X5675 @ 3.07 GHz (4 cores), 48GB RAM. Ubuntu 12.04.2
Slaves	Intel Xeon X5675 @ 3.07 GHz (4 cores), 8GB RAM. Debian 7.7

Table 1. Experimental setup configuration.

						Size (GB)			
Dataset	Triples	|SO|	|S|	|O|	|P|	NT	NT+lzo	HDT	HDT+gz
LinkedGeoData	0.27BN	41.5M	10.4M	80.3M	18.3K	38.5	4.4	6.4	1.9
DBPedia	0.43BN	22.0M	2.8M	86.9M	58.3K	61.6	8.6	6.4	2.7
Ike	0.51BN	114.5M	0	145.1K	10	100.3	4.9	4.8	0.6
Mashup	1.22BN	178.0M	13.2M	167.2M	76.6K	200.3	18.0	17.1	4.6
LUBM-1000	0.13BN	5.0M	16.7M	11.2M	18	18.0	1.3	0.7	0.2
LUBM-2000	0.27BN	10.0M	33.5M	22.3M	18	36.2	2.7	1.5	0.5
LUBM-3000	0.40BN	14.9M	50.2M	33.5M	18	54.4	4.0	2.3	0.8
LUBM-4000	0.53BN	19.9M	67.0M	44.7M	18	72.7	5.3	3.1	1.0
LUBM-5000	0.67BN	24.9M	83.7M	55.8M	18	90.9	6.6	3.9	1.3
LUBM-6000	0.80BN	29.9M	100.5M	67.0M	18	109.1	8.0	4.7	1.6
LUBM-7000	0.93BN	34.9M	117.2M	78.2M	18	127.3	9.3	5.5	1.9
LUBM-8000	1.07BN	39.8M	134.0M	89.3M	18	145.5	10.6	6.3	2.2
LUBM-12000	1.60BN	59.8M	200.9M	133.9M	18	218.8	15.9	9.6	2.9
LUBM-16000	2.14BN	79.7M	267.8M	178.6M	18	292.4	21.2	12.8	3.8
LUBM-20000	2.67BN	99.6M	334.8M	223.2M	18	366.0	26.6	16.3	5.5
LUBM-24000	3.74BN	119.5M	401.7M	267.8M	18	439.6	31.9	19.6	6.6
LUBM-28000	3.74BN	139.5M	468.7M	312.4M	18	513.2	37.2	22.9	7.7
LUBM-32000	4.27BN	159.4M	535.7M	357.1M	18	586.8	42.5	26.1	8.8
LUBM-36000	4.81BN	179.3M	602.7M	401.8M	18	660.5	47.8	30.0	9.4
LUBM-40000	5.32BN	198.4M	666.7M	444.5M	18	730.9	52.9	33.2	10.4

Table 2. Statistical dataset description.

proach. We have developed a proof-of-concept HDT-MR prototype (under the Hadoop framework: version 1.2.1) which uses the existing HDT-Java library[7] (RC-2). This library is also used for the baseline HDT running on a single node.

The **experimental setup** is designed as follows (see Table 1). On the one hand, we use a powerful computational configuration to implement the role of data provider running HDT on a single node. On the other hand, we deploy HDT-MR using a potent *master* and 10 *slave* nodes running on a more memory-limited configuration. This infrastructure tries to simulate a computational cluster in which further nodes may be plugged to process huge RDF datasets. For a fair comparison, the amount of main memory in the single node is the same as the total memory available for the full cluster of Hadoop.

Regarding **datasets**, we consider a varied configuration comprising real-world and synthetic ones. All of them are statistically described in Table 2. Among the real-world ones, we choose them based on their volume and variety, but also attending to their previous uses for benchmarking. *Ike*[8] comprises weather measurements from the Ike hurricane; *LinkedGeoData*[9] is a large geo-spatial dataset derived from *Open Street Map*; and DBPedia 3.8[10] is the well-known knowledge base extracted from Wikipedia. We also join these real-world

[7] http://code.google.com/p/hdt-java/

[8] http://wiki.knoesis.org/index.php/LinkedSensorData

[9] http://linkedgeodata.org/Datasets, as for 2013-07-01

[10] http://wiki.dbpedia.org/Downloads38

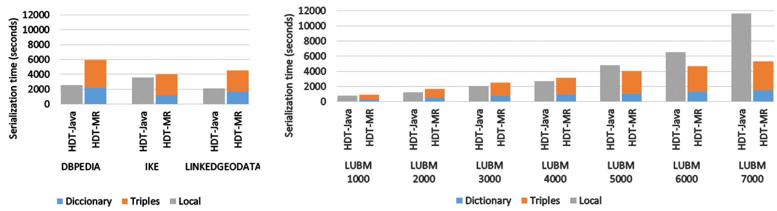

Fig. 7. Serialization times: HDT-Java vs HDT-MR.

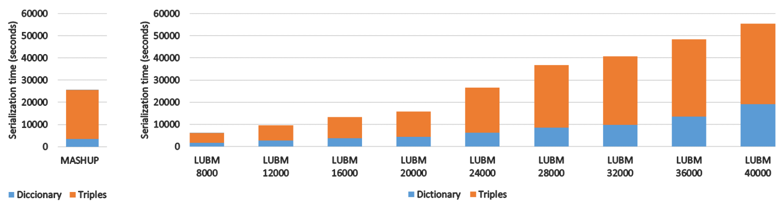

Fig. 8. Serialization times: HDT-MR.

datasets in a *mashup* which comprises all data from the three data sources. On the other hand, we use the LUBM [7] data generator to obtain synthetic datasets. We build "small datasets" from 1,000 (0.13 billion triples) to 8,000 universities (1.07 billion triples). From the latter, we build datasets of incremental size (4,000 universities: 0.55 billion triples) up to 40,000 universities (5.32 billion triples).

Table 2 also shows original dataset sizes both in plain NTriples (NT) and compressed with `lzo`. It is worth noting that `HDT-MR` uses `lzo` to compress the datasets before storing them in HDFS. This format allows for compressed data to be split among the reducers, and provides storage and reading speed improvements [15]. As can be seen, our largest dataset uses 730.9 GB in NTriples, and this spaces is reduced up to 52.9 GB with `lzo` compression.

Figure 7 compares serialization times for HDT-Java and HDT-MR, while Figure 8 shows HDT-MR serialization times for those datasets where HDT-Java is unable to obtain the serialization. These times are averaged over three independent serialization processes for each dataset. As can be seen, HDT-Java reports an excellent performance on real-world datasets, while our current approach only achieves a comparable time for *Ike*. This is an expected result because HDT-Java runs the whole process in main-memory while HDT-MR relies on I/O operations. However, HDT-Java crashes for the *mashup* because the 128 GB of available RAM are insufficient to process such scale in the single node. The situation is similar for the LUBM datasets: HDT-Java is the best choice for the smallest datasets, but the difference decreases with the dataset size and HDT-MR shows better results from *LUBM-5000* (0.67 billion triples). HDT-Java fails to process datasets from *LUBM-8000* (1.07 billion triples) because of memory requirements. This is the target scenario for HDT-MR, which scales to the

LUBM-40000 without issues. As can be seen in both figures, serialization times increase linearly with the dataset size, and triples encoding remains the most expensive stage.

RDF compression is not the main purpose of this paper, but it is worth emphasizing HDT space numbers, as previous literature does not report compression results for such large datasets. These numbers are also summarized in Table 2. HDT always reports smaller sizes than the original datasets compressed with `lzo`. For instance, HDT serializes *LUBM-40000* using 19.7 GB less than `NT+lzo`. The difference increases when compressed with `gzip`. For *LUBM-40000*, `HDT+gz` uses 42.5 GB less than `NT+lzo`. In practice, it means that `HDT+gz` uses 5 times less space than `NT+lzo`. Finally, it is worth remembering that HDT-MR obtains the same HDT serialization than a mono-node solution, hence achieving the same compression ratio and enabling the same query functionality. Source code and further details on HDT-MR are available at the HDT-MR project[11].

5 Conclusions and Future Work

HDT is gaining increasing attention, positioning itself as the *de facto* baseline for RDF compression. Latest practical applications exploit the HDT built-in indexes for RDF retrieval with no prior decompression, making HDT evolve to a self-contained RDF store. In this paper we introduce HDT-MR, a technique tackling scalability issues arising to HDT construction at very large scale. HDT-MR lightens the previous heavy memory-consumption burden by moving the construction task to the MapReduce paradigm. We present the HDT-MR distributed workflow, evaluating its performance against the mono-node solution in huge real-world and benchmarking RDF datasets, scaling up to more than 5 billion triples. Results show that HDT-MR is able to scale up to an arbitrary size in commodity clusters, while the mono-node solution fails to process datasets larger than 1 billion triples. Thus, HDT-MR greatly reduces hardware requirements for processing Big Semantic Data.

Our future work focuses on two directions. First, we plan to exploit HDT-MR achievements as these can be directly reused by the HDT community, fostering the development of novel applications working at very large scale. Finally, our research consider to combine HDT and MapReduce foundations to work together on other Big Semantic Data tasks, such as querying and reasoning.

Acknowledgments

This paper is funded by the Spanish Ministry of Economy and Competitiveness: TIN2013-46238-C4-3-R, Austrian Science Fund (FWF): M1720-G11, and ICT COST Action KEYSTONE (IC1302). We thank Javier I. Ramos by his support with the Hadoop cluster, and Jürgen Umbrich for lending us his sever.

[11] `http://dataweb.infor.uva.es/projects/hdt-mr/`

References

1. S. Álvarez-García, N. Brisaboa, J.D. Fernández, M.A. Martínez-Prieto, and G. Navarro. Compressed Vertical Partitioning for Efficient RDF Management. *Knowledge and Information Systems*, 2014. DOI: 10.1007/s10115-014-0770-y.
2. N. Brisaboa, R. Cánovas, F. Claude, M.A. Martínez-Prieto, and G. Navarro. Compressed String Dictionaries. In *Proc. of SEA*, pages 136–147, 2011.
3. L. Cheng, A. Malik, S. Kotoulas, T.E. Ward, and G. Theodoropoulos. Efficient Parallel Dictionary Encoding for RDF Data. In *Proc. of WebDB*, 2014.
4. O. Curé, G. Blin, D. Revuz, and D.C. Faye. WaterFowl: A Compact, Self-indexed and Inference-Enabled Immutable RDF Store. In *Proc. of ESWC*, pages 302–316, 2014.
5. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Largee Clusters. In *Proc. of OSDI*, pages 137–150, 2004.
6. J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange. *Journal of Web Semantics*, 19:22–41, 2013.
7. Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 3(2):158–182, 2005.
8. B. Heitmann and C. Haye. SemStim at the LOD-RecSys 2014 Challenge. In *Proc. of SemWebEval*, pages 170–175, 2014.
9. E. Hervalejo, M.A. Martínez-Prieto, J.D. Fernández, and O. Corcho. HDTourist: Exploring Urban Data on Android. In *Proc. of ISWC (Poster & Demos)*, volume CEUR-WS 1272, pages 65–68, 2014.
10. A. Joshi, P. Hitzler, and G. Dong. Logical Linked Data Compression. In *Proc. of ESWC*, pages 170–184, 2013.
11. K.-h. Lee, Y.-j. Lee, H. Choi, Y.D. Chung, and B. Moon. Parallel Data Processing with MapReduce: a Survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.
12. F. Manola and R. Miller. *RDF Primer*. W3C Recommendation, 2004. www.w3.org/TR/rdf-primer/.
13. M.A. Martínez-Prieto, M. Arias, and J.D. Fernández. Exchange and Consumption of Huge RDF Data. In *Proc. of ESWC*, pages 437–452, 2012.
14. M.A. Martínez-Prieto, J.D. Fernández, and R. Cánovas. Querying RDF dictionaries in compressed space. *SIGAPP Appl. Comput. Rev.*, 12(2):64–77, 2012.
15. N. Mirajkar, S. Bhujbal, and A. Deshmukh. Perform wordcount Map-Reduce Job in Single Node Apache Hadoop cluster and compress data using Lempel-Ziv-Oberhumer (LZO) algorithm, 2013. arXiv:1307.1517.
16. J.Z. Pan, J.M. Gómez-Pérez, Y. Ren, H. Wu, and M. Zhu. SSP: Compressing RDF Data by Summarisation, Serialisation and Predictive Encoding. Technical report, 2014. Available at http://www.kdrive-project.eu/wp-content/uploads/2014/06/WP3-TR2-2014_SSP.pdf.
17. J. Urbani, J. Maassen, H. Bal, N. Drost, F. Seintra, and H. Bal. Scalable RDF Data Compression with MapReduce. *Concurrency and Computation: Practice and Experience*, 25:24–39, 2013.
18. R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. Van de Walle. Querying datasets on the web with high availability. In *Proc. of ISWC*, pages 180–196, 2014.