# Compresión de Big Semantic Data basada en HDT y MapReduce

José M. Giménez-García[1], Javier D. Fernández[2], and Miguel A. Martínez-Prieto[3]

[1] Univ Lyon, UJM-Saint-Etienne, CNRS, Laboratoire Hubert Curien UMR 5516, F-42023 Saint Etienne (Francia)
jose.gimenez.garcia@univ-st-etienne.fr,
[2] Vienna University of Economics and Business (Austria)
jfernand@wu.ac.at
[3] DataWeb Research, Departmento de Informática, Univ. de Valladolid (España)
migumar2@infor.uva.es

**Resumen** HDT es un formato binario que nace con el objetivo de reducir los requisitos de almacenamiento que presentan las sintaxis RDF tradicionales. Su estructura interna no sólo favorece la compresión, sino que también facilita la resolución de algunas consultas de interés en el ámbito de la Web Semántica. Sin embargo, el proceso de codificación HDT requiere una cantidad importante de memoria principal, lo que limita su adopción en aplicaciones que manejan *Big Semantic Data*. Para afrontar este problema, proponemos HDT-MR, una revisión del algoritmo de construcción de HDT basada en tecnología MapReduce. Los experimentos que presentamos en este artículo demuestran que el coste de HDT-MR crece linealmente con el volumen de los datos de entrada, lo cual facilita la serialización de colecciones RDF de miles de millones de triples utilizando un pequeño *cluster Hadoop*.

**Keywords:** Web Semántica, Compresión, RDF, HDT, MapReduce

## 1. Introducción

RDF (*Resource Description Framework*) [8] se ha convertido en el formato de referencia para la publicación e intercambio de datos en la Web de Datos. Iniciativas como *Linked Open Data* ponen de manifiesto su potencial para la integración de conjuntos heterogéneos de datos, con diferentes grados de estructura y procedentes de fuentes diversas. La flexibilidad de RDF se debe a la estructura ternaria (*triple*) que utiliza para describir la información: (i) el *sujeto* identifica el recurso que está siendo descrito, (ii) el *predicado* establece una propiedad sobre dicho recurso y (iii) el objeto fija el valor de la propiedad para el recurso descrito. Una colección RDF está formada por un conjunto de triples cuyos sujetos y objetos pueden verse como nodos de un grafo dirigido y en el que los predicados juegan el rol de aristas. Aunque esta noción de grafo es una buena metáfora para entender la forma en la que RDF organiza la información, el almacenamiento

y/o intercambio de estas colecciones requiere el uso de formatos de serialización. Esta necesidad ha sido reconocida explícitamente en la última propuesta del *RDF Primer*[4], realizada por el grupo de trabajo en RDF dentro del *World Wide Web Consortium (W3C)*. Este documento sugiere diversas alternativas (JSON-LD, RDF/XML o los formatos basados en Turtle) que serializan el grafo RDF en *texto plano*. Esta decisión simplifica, notablemente, el proceso de escritura de los triples y puede implementarse sin grandes sobrecargas computacionales. En contrapartida, los ficheros resultantes tienden a ser muy voluminosos. Una solución trivial (y muy usada en la práctica) es utilizar un compresor universal, como `gzip`, para reducir el tamaño de estos ficheros. Sin embargo, esta decisión sólo oculta el problema, ya que el uso efectivo de los triples requiere un proceso previo de descompresión que devuelve la colección a su tamaño original.

HDT (*Header-Dictionary-Triples*) es un formato binario de serialización que surge como alternativa a las propuestas anteriores. HDT codifica la colección utilizando dos componentes: *Diccionario* y *Triples*. El *Diccionario* asocia cada término usado en el grafo (URIs y literales) con un identificador numérico único (ID). Esta decisión permite obtener un grafo de IDs, cuya codificación binaria se realiza en el componente *Triples*. Ambos componentes se serializan en espacio comprimido y permiten acceder a los datos sin necesidad de descomprimirlos previamente [9]. Esta propiedad ha facilitado que HDT pase a considerarse una pieza básica para la construcción de motores de almacenamiento semánticos. *HDT-FoQ* [9] muestra el potencial de la tecnología HDT para la resolución de algunas consultas SPARQL, mientras que *WaterFowl* [3] incopora HDT en su motor de inferencia. Otras tecnologías como *Linked Data Fragments* [13] o el sistema de recomendación *SemStim* demuestran la eficiencia de HDT como motor de almacenamiento.

Todos los beneficios generados por HDT, en el usuario final, están sujetos al coste de serialización que debe pagar el publicador de la colección. La construcción de los componentes *Diccionario* y *Triples* requiere un procesamiento exhaustivo de toda la colección RDF para el que necesita un equipo de cómputo con una gran cantidad de memoria RAM. Por lo tanto, estamos ante un proceso poco escalable que dificulta la serialización de colecciones RDF de gran tamaño (en el orden de los cientos o miles de millones de triples). Este tipo de colecciones no son muy comunes (aunque comienzan a aparecer en áreas como la biología o la astronomía), pero sí es más habitual encontrarnos con *mashups* que integran datos heterogéneos procedentes de fuentes diversas (por ejemplo, en ámbitos como las *smart cities*) y que materializan la noción de *Big Semantic Data*.

En este artículo proponemos HDT-MR, una nueva implementación del *workflow* de publicación de colecciones RDF (basado en HDT) desarrollada sobre el *framework* MapReduce [4]. Esta decisión supone una mejora sustancial en la escalabilidad del proceso original de construcción de HDT y nos permite serializar colecciones RDF de gran tamaño. En los experimentos actuales, HDT-MR procesa colecciones 10 veces más grandes que las serializadas con la propuesta original y obtiene tiempos de codificación que crecen linealmente con el tamaño

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

de los datos de entrada. Estos resultados avalan la integración de HDT-MR como componente principal del *workflow* de serialización HDT, garantizando la escalabilidad del mismo y preservando todas las características originales de formato de cara a su explotación en el usuario final [9,5].

El resto artículo se organiza como sigue: en la sección 2 se presentan los conocimientos necesarios para entender la propuesta actual, cuya descripción se realiza en la sección 3. La sección 4 describe los resultados experimentales obtenidos con HDT-MR y, finalmente, la sección 5 discute nuestras conclusiones actuales y las líneas de trabajo futuro que surgen entorno a ellas.

## 2. Background

Esta sección resume los fundamentos de MapReduce y, a continuación, describe la tecnología HDT y su relevancia en el ámbito de la compresión RDF.

### 2.1. MapReduce

MapReduce [4] plantea un modelo de programación y un *framework* optimizado para procesar grandes volúmenes de datos en entornos distribuidos. Su objetivo principal es abstraer la complejidad subyacente a estos entornos y ofrecer un mecanismo eficiente que permita paralelizar el procesamiento de datos de cualquier naturaleza. Una tarea (*job*) MapReduce comprende dos fases. La primera fase (`map`) lee los datos de entrada como pares clave-valor $(k1, v1)$ y genera, como resultado, series de pares clave-valor en un dominio diferente al inicial $(k2, v2)$. La segunda fase (`reduce`) procesa los valores $v2$, relacionados con cada clave $k2$, y obtiene una lista final de resultados. Cada una de las fases ejecuta múltiples procesos que actúan sobre una pequeña parte de los datos de entrada y, por tanto, generan una parte del resultado final.

MapReduce implementa una arquitectura maestro-esclavo (*master-slave*). El nodo maestro se encarga de iniciar la tarea, distribuir la carga de trabajo dentro del *cluster* y procesar la información administrativa generada durante la ejecución, mientras que los nodos esclavos ( trabajadores) ejecutan los `map` y/o `reduce` asignados. MapReduce hace un uso exhaustivo de operaciones de I/O. Los trabajadores leen y escriben datos en discos que implementan el modelo de ficheros distribuido GFS *(Google File System)*, de forma que los resultados temporales se almacenan en diferentes nodos del *cluster*. Esto favorece la *localidad de los datos* y maximiza el rendimiento individual de los trabajadores. Aún así, todavía hay información que debe transferirse entre los nodos.

Apache Hadoop[5] es la implementación de MapReduce más utilizada actualmente. Hadoop introduce un modelo de almacenamiento distribuido denominado HDFS *(Hadoop Distributed File System)* que permite gestionar el factor de replicación de los datos (comúnmente, cada fragmento de datos se replica en tres nodos diferentes), con el objetivo de mejorar su localidad y, a su vez, incrementar la tolerancia a la ocurrencia de fallos.

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

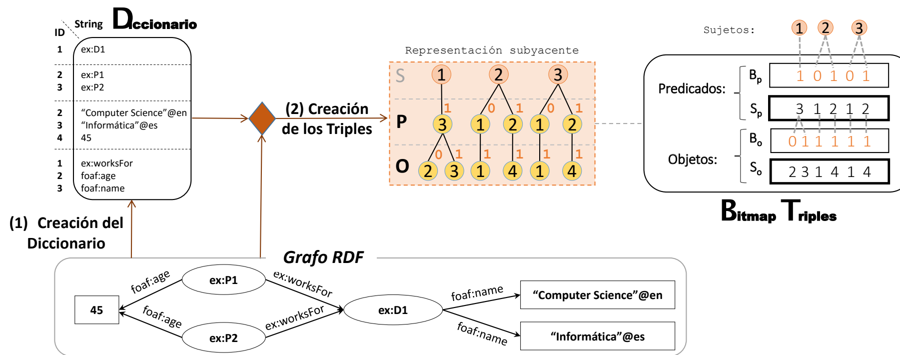


**Figura 1.** Componentes *Diccionario* y *Triples* (HDT) para codificar un grafo RDF.

## 2.2. HDT

HDT[6] [5] es un formato binario diseñado para optimizar el almacenamiento y la transmisión de ficheros RDF. HDT codifica una colección RDF utilizando tres componentes: i) la *Cabecera (Header)* contiene los metadatos necesarios para el descubrimiento y el parseo del fichero HDT; ii) el *Diccionario (Dictionary)* es un catálogo que organiza el conjunto de términos diferentes utilizados en la colección y asigna, a cada uno de ellos, un identificador único: ID; iii) finalmente, el componente *Triples* reemplaza los términos utilizados en cada triple RDF por su correspondiente ID y codifica sucintamente el grafo de IDs resultante. Los componentes *Diccionario* y *Triples* (ilustrados en la figura 1) son los responsables de los resultados de compresión que obtiene HDT.

El *Diccionario* organiza los términos utilizados en la colección de acuerdo al rol que desempeñan en la misma. Esto da lugar a cuatro particiones disjuntas: los términos que juegan el papel de sujetos y objetos (**SO**) se identifican en el rango `[1, |SO|]` (siendo `|SO|` el número de términos diferentes que desempeñan este rol); los términos que desempeñan, exclusivamente, roles de sujeto (**S**) u objeto (**O**) (ambas secciones utilizan IDs en el rango de `|SO|+1` a `|SO|+|S|` y `|SO|+|O|`, respectivamente, siendo `|S|` y `|O|` el número de sujetos y objetos exclusivos); finalmente, los predicados (**P**) se identifican en el rango `[1, |P|]`, donde `|P|` cuantifica el número total de predicados en la colección. Nótese que no existe una partición que combine los términos que aparecen como predicado y, simultáneamente, sujeto y/o objeto. Por ello, dichos términos aparecerán repetidos en la secciones que corresponda. Cada sección del *Diccionario* se codifica independientemente para aprovechar sus características particulares. La codificación de diccionarios [10] es un problema ortogonal al tratado en este trabajo, por lo tanto no profundizaremos más en el.

El componente *Triples* codifica el grafo que resulta de reemplazar los términos por sus correspondientes IDs en el *Diccionario*. Esto es, los triples RDF se codifican como tuplas de tres IDs (ID-triples a partir de aquí): ($\mathtt{id}_s$ $\mathtt{id}_p$ $\mathtt{id}_o$), donde

[6] HDT es una *Member Submission* del W3C: `http://www.w3.org/Submission/HDT/`

$\mathtt{id}_s$, $\mathtt{id}_p$, y $\mathtt{id}_o$ son, respectivamente, los IDs de los términos sujetos, predicado y objeto en el *Diccionario*. Este componente codifica los triples como un conjunto de árboles, uno por cada sujeto en la colección: la raíz de cada árbol identifica al propio sujeto, el nivel intermedio contiene la lista ordenada de los predicados que establecen las propiedades del sujeto y, finalmente, las hojas listan los IDs de los objetos que fijan los valores de cada propiedad del sujeto. Esta organización codifica los IDs de predicados y objetos (que describen el nivel intermedio y las hojas de los árboles) mediante *dos secuencias* de números enteros: `Sp` y `So`, y *dos secuencias de bits*: `Bp` y `Bo`, alineadas con las anteriores [5].

**Serialización HDT.** Una vez descritos los componentes *Diccionario* y *Triples*, estamos en disposición de explicar el proceso de serialización que, actualmente, utiliza HDT. Este proceso contempla tres etapas principales.

*1. Organización de los términos RDF.* Esta primera etapa, procesa la colección y clasifica los términos de cada triple en su sección del *Diccionario*. Para ello, construye (en RAM) tres tablas *hash*, que implementan las relaciones *sujeto-ID*, *predicado-ID* y *objeto-ID*, y busca el sujeto, el predicado y el objeto en la tabla correspondiente. Si el término existe, se recupera el ID asociado; en caso contrario, se inserta el nuevo término en la tabla y se le asigna un ID autoincremental. Estos IDs se utilizan para obtener una representación ID-triples temporal que se almacena en un *array* (también en memoria principal). Una vez leído el fichero de entrada, se procesan las tablas que contienen los *mappings* de sujetos y objetos para identificar aquellos términos que desempeñan ambos roles en la colección. Estos términos comunes se descartan en sus respectivas estructuras y se insertan en una cuarta tabla *hash* que implementa el *mapping* correspondiente a la sección SO del diccionario.

*2. Construcción del Diccionario.* Cada sección del *Diccionario* se ordena lexicográficamente con el objetivo de aprovechar la existencia de prefijos comunes entre los términos y, con ello, reducir el espacio requerido para su codificación [10]. Finalmente, se construye un *array* auxiliar que almacena la permutación de los IDs desde su orden inicial (en la tabla *hash*) a su orden final.

*3. Construcción de los Triples.* La etapa final recorre el *array* temporal de IDs-triples y reemplaza los IDs inicales, de cada triple, por los obtenidos tras la ordenación del *Diccionario*. Una vez actualizado, el *array* de ID-triples se ordena por sujeto, predicado y objeto de cara a su codificación final. Esta proceso de codificación sólo requiere un recorrido secuencial del *array* de ID-triples en el que se extraen los pares (predicado,objeto) para cada sujeto y se almacenan en las secuencias `Sp/So`, actualizando, coordinadamente, las secuencias de bits.

### 2.3. Trabajo Relacionado

Atendiendo a la taxonomía presentada en [11], HDT puede considerarse un compresor *sintáctico* dada su capacidad para detectar redundancia a nivel de serialización. Por un lado, el *Diccionario* aborda la redundancia simbólica existente en los términos de la colección, mientras que el componente *Triples* explota

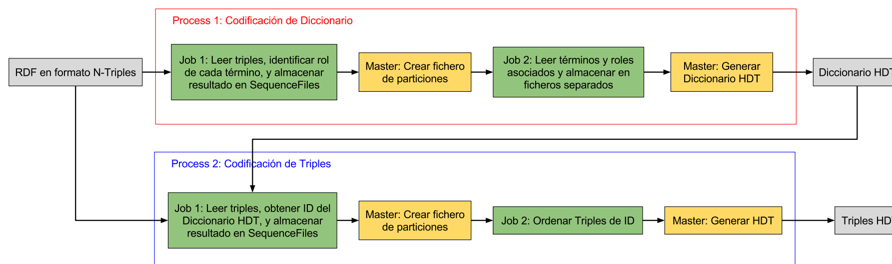


**Figura 2.** Descripción del *workflow* HDT-MR.

la redundancia estructural subyacente a la topología del grafo. Los compresores sintácticos son los que consiguen unos mejores resultados, en la práctica. Propuestas como $k^2$-triples [14] o RDFCSA [1] utilizan diferentes estrategias de codificación que reducen notablemente el tamaño de los ficheros RDF. Ambas soluciones también comparten la necesidad de llevar a cabo un proceso de compresión complejo que también requiere grandes cantidades de memoria. Por otra parte, los compresores *lógicos* se centran en detectar aquellos triples que pueden inferirse a partir de otros triples ("primitivos") y sólo codifican la parte "primitiva" de la colección original. Técnicas como la propuesta en [7] eliminan hasta más del 50 % de los triples, pero su efectividad no compite con la obtenida por los compresores sintácticos. Además, el proceso de detección de los triples redundantes también es complejo y exhaustivo en el uso de recursos. Un trabajo más reciente [11] ha presentado una propuesta que detecta y explota tanto la redundancia sintáctica como la semántica. Sus resultados mejoran ligeramente a HDT, pero quedan lejos de los obtenidos por técnicas como $k^2$-triples. En sintonía con todas las técnicas anteriores, su proceso de compresión resulta muy complejo en la práctica.

En resumen, los compresores RDF están lastrados por un problema importante de escalabilidad que ya ha sido estudiado en el ambiente MapReduce [12]. En este caso, se implementó un algoritmo que construía el *mapping* entre términos e IDs y, posteriormente, reescribía los triples de acuerdo a los IDs de sus términos. Un trabajo más reciente [2] ha abordado este problema mediante otro algoritmo distribuido desarrollado, en lenguaje X10. En ambos casos, los resultados son un paso adelante respecto al estado del arte actual.

## 3. HDT-MR

HDT-MR se implementa mediante un proceso MapReduce que comprende dos etapas centradas en la codificación de los componentes HDT (ver figura 2). Cada una de estas etapas, y sus consiguientes *jobs*, se explican a continuación.

### 3.1. Codificación del Diccionario

Esta primera etapa se centra en la construcción del *Diccionario* HDT, partiendo de la premisa de que la colección RDF original se encuentra en formato

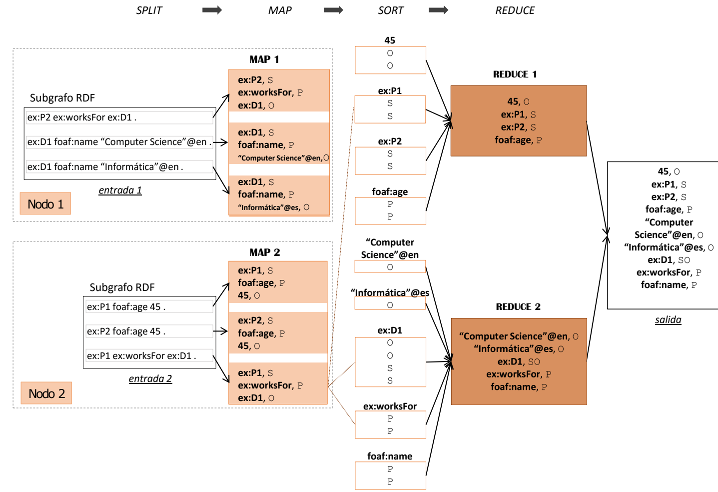


**Figura 3.** Job 1.1: identificación de roles.

**Algoritmo 1** Job 1.1: identificación de roles.

```
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 ← true
        else if role contains "P" then isPredicate ← true
        else if role contains .O" then isObject ← true
        end if
    end for
    roles ← ""
    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
```

N-Triples. Para su construcción, es necesario identificar el rol de cada término en la colección, obtener las cuatro particiones (en orden lexicográfico) que describen el Diccionario y, finalmente, codificarlas. Para ello, empleamos dos *jobs* MapReduce y dos procesos locales que se ejecutan en el nodo *maestro*.

**Job 1.1: Identificación de roles.** Este *job* se encarga de identificar los roles que juegan los términos RDF en la colección. Para ello, los *mappers* procesan (uno a uno) los triples y emiten pares clave-valor de la forma `(término RDF, rol)`, donde el valor de rol puede ser $S$ (sujeto), $P$ (predicado), u $O$ (objeto), de acuerdo a la posición del término en el triple. La figura 3 muestra un *cluster* básico de dos nodos, en el cada uno de ellos participa en el procesamiento de la colección representada en la figura 1. Por ejemplo, los pares (`ex:P1`,$S$), (`ex:worksFor`,$P$), y

`(ex:D1,`$O$`)` son los resultados de procesar el triple `(ex:P1, ex:worksFor, ex:D1)`. La salida de cada `map` pasa por un *combiner* (en cada nodo en el que se realizan las tareas `map`) antes de ser enviada a los *reducer*. Su objetivo es eliminar los pares `(término RDF, rol)` repetidos y, con ello, reducir los costes de comunicación.

Los pares clave-valor resultantes se ordenan y distribuyen entre los *reducers*, que se encargan de identificar los diferentes roles asociados a cada término. Cuando un término tiene asociados valores $S$ y $O$, el *reducer* genera un par `(término RDF, `$SO$`)`. En caso contrario, genera como salida el par original asociado al término. El resultado del *job* consiste en varias listas de pares `(término RDF, rol)` (tantas como número de *reducers*). El algoritmo 1 muestra el pseudocódigo que implementa este *job*.

**Job 1.2: Ordenación de los términos.** Para construir las diferentes secciones de un *Diccionario* HDT, es necesario que todos los términos están ordenados lexicográficamente en un único fichero de entrada. Sin embargo, el *job* anterior genera múltiples ficheros, cuyos contenidos están ordenados por el valor de la clave. Además, se plantea una cuestión adicional acerca de qué términos están en cada fichero. Por defecto, Hadoop utiliza una función *hash* sobre la clave antes de asignar el par a un determinado *reducer* con el objetivo de distribuir uniformemente el universo de claves entre el conjunto de *reducers* disponibles.

No obstante, el *framework* permite configurar diferentes políticas de distribución. La alternativa más sencilla y, a la vez, menos eficiente pasa por utilizar un único *reducer* que procese todos los pares. Esta opción sería equivalente a renunciar al procesamiento paralelo que ofrece Hadoop. La segunda alternativa pasaría por crear manualmente los grupos de claves que se asignarán a cada nodo. Sin embargo, estas particiones deberían elegirse cuidadosamente ya que cualquier desequilibrio provocaría que alguno de los *reducers* necesitase más tiempo que los demás para acabar su tarea y, por consiguiente, se convertiría en el cuello del botella del proceso. Descartadas las opciones anteriores, HDT-MR usa el *TotalOrderPartitioner* para muestrear los pares de entrada de una manera más eficiente. Sin embargo, este particionamiento no puede llevarse a cabo durante la ejecución de *job*, sino que tiene que completarse antes de su inicio.

Esta situación motiva la inclusión del segundo *job*, cuyo objetivo es obtener una ordenación global de la salida en el *job* anterior. Así, se toma como entrada las listas de pares `(término RDF, rol)`, obtenidas en el *job* anterior, y se agrupan de acuerdo a su valor de rol. Para ello, empleamos *identity mappers*, que envían directamente su entrada a los *reducers* sin realizar procesamiento alguno. Por su parte, los *reducers* simplemente generan como salida una lista ordenada de los términos correspondientes a cada rol. La figura 4 ilustra este *job*. Nótese que la construcción del Diccionario HDT sólo requiere el valor del término RDF, por lo que los *reducers* emiten pares de la forma `(término RDF, `*`null`*`)`. El resultado final de cada *reducer* comprende cuatro listas, una por cada sección del *Diccionario*, que finalmente se concatenan para obtener el orden global de cada una de estas secciones. El pseudocódigo de este *job* se describe en el algoritmo 2.

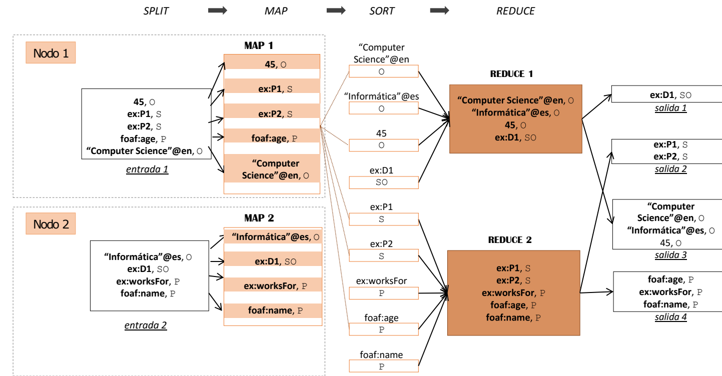


**Figura 4.** Job 1.2: ordenación de los términos.

---

**Algoritmo 2** Job 1.2: ordenación de los términos.

```
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 ← true
        else if resource contains 'P' then isPredicate ← true
        else if resource contains 'O' then isObject ← true
        end if
    end for
    output ← ""
    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
```

---

**Proceso local 1.3: Codificación del Diccionario HDT** La tarea final, de la etapa de codificación del Diccionario, se realiza en el nodo *maestro* y se centra, exclusivamente, en la compresión de las cuatro secciones obtenidas en el *job* anterior. Para cada una de ellas, el proceso lee (línea a línea) los términos y los codifica utilizando la técnica *Plain Front-Coding* [10]. Este es un proceso sencillo que no presenta problemas de escalabilidad.

## 3.2. Codificación de los Triples

Esta segunda etapa realiza una nueva lectura de la colección original con el objetivo de construir el componente *Triples* de HDT. En este caso, es necesario reemplazar los términos RDF por su correspondiente ID en el *Diccionario* y codificar sucintamente esta representación ID-triples de acuerdo a los principios considerados en HDT. HDT-MR afronta estas necesidades mediante dos *jobs* MapReduce y dos procesos locales (ver figura 2).

**Job 2.1: Construcción de ID-triples** Para ejecutar este primer *job* es necesario que cada nodo disponga del Diccionario comprimido (en la etapa anterior). Una vez recibido y cargado en memoria, los *mappers* simplemente leen los términos de cada triple y los reemplazan por sus IDs (obtenidos mediante operaciones

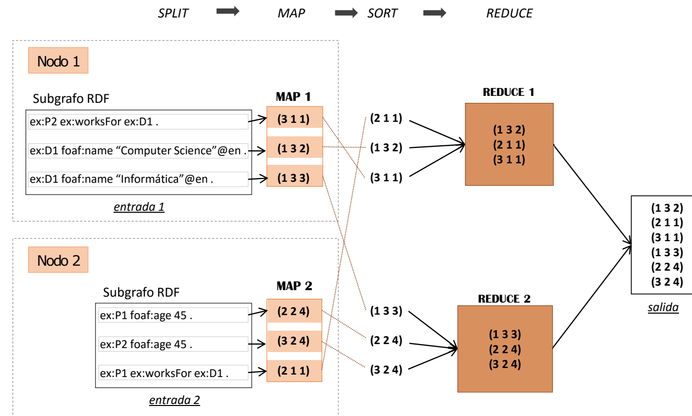


**Figura 5.** Job 2.1: construcción de ID-triples

**Algoritmo 3** Job 2.1: construcción de ID-triples

**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**

de búsqueda en el Diccionario). Estos resultados se transmiten a *identity reducers* que sencillamente ordenan su entrada y emiten una lista de pares `(ID-triple, null)`, como puede verse en la figura 5 (omitimos *null* por simplicidad). El resultado final de este *job* (ver Algoritmo 3) comprende tantas listas ordenadas de ID-Triples como *reducers* hayan participado en el job.

**Job 2.2: Ordenación de ID-Triples** Al igual que en la primera etapa, este *job* asume la responsabilidad de ordenar la salida obtenida por el *job* previo. Antes de comenzar el *job*, se utiliza el *TotalOrderPartitioner* para muestrear la entrada y, posteriormente, ordenar los ID-triples por sujeto, preciado y objeto. Es, por tanto, un proceso sencillo que combina *identity mappers* e *identity reducers*. El resultado contiene una lista ordenada de pares (ID-triples, *null*), La figura 6 ilustra este *job* (nótese de nuevo que los valores *null* se omiten en la salida).

**Proceso Local 2.3: Codificación de los Triples HDT** Este proceso final se ejecuta (en el maestro) mediante un bucle que itera sobre el fichero de ID-triples, obtenido en el *job* anterior, y construye las secuencias de bits (`Bp`, `Bo`) y enteros (`Sp`, `So`), explicadas previamente.

## 4. Evaluación Experimental

Esta sección evalúa el rendimiento de HDT-MR y lo compara con la propuesta original (mono-nodo). Para ello, hemos desarrollado un prototipo de `HDT-MR`[7] (en

[7] Disponible en `http://goo.gl/Z5vl72`.

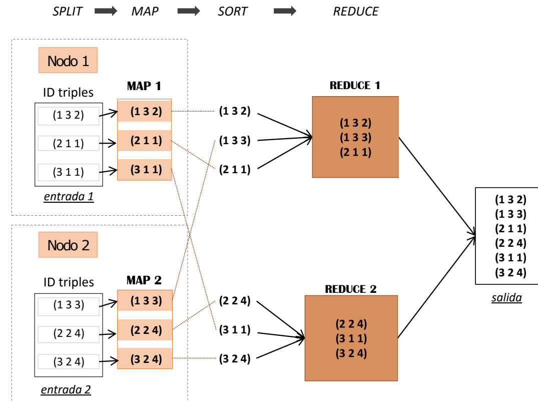


**Figura 6.** Job 2.2: Ordenación de ID-Triples

| Máquina | Configuración |
|---|---|
| **Nodo Único** | Intel Xeon E5-2650v2 @ 2.60GHz (32 núcleos), 128GB RAM. Debian 7.8 |
| **Maestro** | Intel Xeon X5675 @ 3.07 GHz (4 núcleos), 48GB RAM. Ubuntu 12.04.2 |
| **Esclavos** | Intel Xeon X5675 @ 3.07 GHz (4 núcleos), 8GB RAM. Debian 7.7 |

**Cuadro 1.** Entorno experimental.

| | | | | | | Tamaño (GB) | | | |
|---|---|---|---|---|---|---|---|---|---|
| Colección | 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-68000 | 9,05BN | 337,9M | 1202,4M | 757,0M | 18 | 1244,4 | 90,2 | 57,6 | 18,9 |
| LUBM-72000 | 9,59BN | 357,8M | 1269,4M | 801,7M | 18 | 1318,0 | 95,5 | 61,3 | 20,0 |

**Cuadro 2.** Descripción de los conjuntos de datos de prueba.

Hadoop, versión 1.2.1) que usa la librería HDT-Java existente[8] (RC-2). Consideramos esta misma librería HDT-Java como referencia original de la implementación mono-nodo. Nuestro entorno de experimentación (ver Cuadro 1) considera dos configuraciones computacionales. Por un lado, utilizamos un servidor potente que ejecuta la implementación HDT mono-nodo. Por otro lado, desplegamos HDT-MR sobre un cluster con *maestro* potente y 10 *esclavos* con recursos de memoria más limitados. Para una comparación más justa, la memoria disponible en el servidor (HDT mono-nodo) es la misma que en la suma de la memoria de todos de los nodos cluster Hadoop (HDT-MR).

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

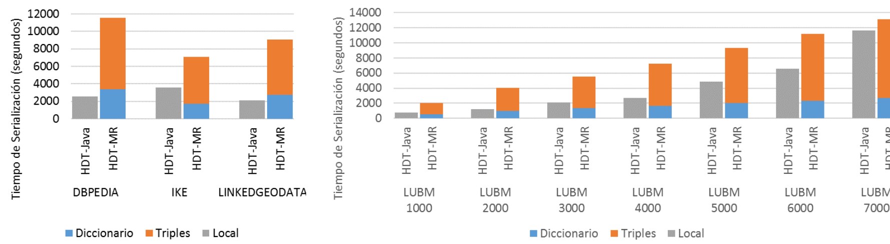


**Figura 7.** Tiempos de serialización: HDT-Java vs. HDT-MR.

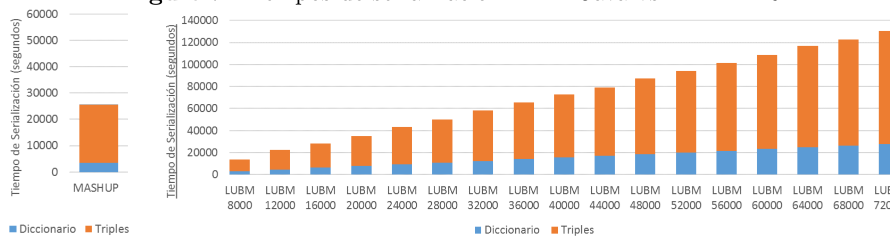


**Figura 8.** Tiempos de serialización: HDT-MR.

En lo referente a las colecciones de datos de prueba (ver Cuadro 2), consideramos un corpus variado, compuesto de colecciones reales y sintéticas. La elección de las colecciones reales considera criterios de volumen y variedad, así como su uso previo para evaluación en el estado del arte. *Ike*[9] incluye mediciones meteorológicas del huracán Ike; *LinkedGeoData*[10] es un gran conjunto de datos espaciales derivado de *Open Street Map*; y DBPedia 3.8[11] es una base de conocimiento extraída de Wikipedia. De igual modo, combinamos todos ellos en un *mashup* que incluye los datos de las tres fuentes anteriores. Por otra parte, empleamos la herramienta LUBM [6] como generador de datos sintéticos (basado en modelar un sistema académico con un número parametrizable de universidades). Construimos "colecciones pequeñas" desde 1000 ($0,13$ billones[12] de triples) a 8000 universidades ($1,07$ billones de triples) y "colecciones grandes" (añadiendo 4000 universidades en cada una de ellas: $0,55$ billones de triples), hasta un máximo de 72000 universidades ($9,59$ billones de triples).

El Cuadro 2 muestra los tamaños originales en formato NTriples (NT) y los comprimidos con `lzo`. Cabe destacar que `HDT-MR` usa `lzo` para comprimir las colecciones antes de almacenarlas en HDFS. Como se muestra en el cuadro, la colección más grande ocupa 1318 GB en NTriples, mientras que comprimida con `lzo` apenas requiere $95,5$ GB.

[9] `http://wiki.knoesis.org/index.php/LinkedSensorData`
[10] `http://linkedgeodata.org/Datasets`, versión 01-07-2013
[11] `http://wiki.dbpedia.org/Downloads38`
[12] Por motivo de consistencia con el estado del arte, empleamos la escala anglosajona, esto es, 1 billón = 1 000 000 000 (mil millones).

La figura 7 compara los tiempos de serialización de HDT-Java y HDT-MR en las colecciones de menor tamaño, mientras que la Figura 8 sólo muestra los tiempos de HDT-MR, dado que HDT-Java no es capaz de serializar las colecciones más grandes. HDT-Java obtiene un buen rendimiento en los conjuntos de datos reales, y HDT-MR sólo puede competir en el conjunto *Ike*. No obstante, no se trata de un resultado inesperado, puesto que HDT-Java se ejecuta en memoria principal mientras que HDT-MR paga el sobrecoste de las operaciones de I/O que realiza Hadoop. En cambio, HDT-Java no es capaz de serializar el *mashup*, ya que los 128 GB de RAM disponibles son insuficientes para procesar tal volumen de información en un único nodo. El resultado es similar para las colecciones sintéticas LUBM: HDT-Java es la mejor opción para los conjuntos de datos pequeños, pero la diferencia respecto a HDT-MR se reduce a medida que el tamaño del conjunto de datos se incrementa. HDT-Java, además, no escala para colecciones mayores que *LUBM-8000* (1,07 billones de triples). Este es el escenario indicado para HDT-MR, que consigue procesar sin problemas todas las colecciones hasta *LUBM-72000*. Como muestran ambas figuras, los tiempos de serialización se incrementan de manera lineal con el tamaño de la *colección*, siendo la codificación de los Triples la etapa más costosa.

Aunque la compresión no es el objetivo principal de este artículo, cabe mencionar los resultados obtenidos por HDT, ya que ninguno de los experimentos previos reportaba número para colecciones tan grandes. El cuadro 2 resume dichos resultados y puede observarse que HDT mejora los espacios obtenidos por `lzo` en todos los casos. La diferencia es aún mayor cuando HDT se combina con gzip. Por ejemplo, HDT utiliza 19,7GB menos que `NT+lzo` para comprimir *LUBM-40000* y la diferencia se incrementa hasta 42,5GB frente a `HDT+gz`.

## 5. Conclusiones

HDT está adquiriendo un reconocimiento cada vez mayor como una referencia *de facto* en el área de compresión RDF. Además, los índices que incluye HDT permiten recuperar RDF sin necesidad de descompresión previa, por lo que han surgido aplicaciones semánticas que emplean HDT como motor de almacenamiento RDF. En este artículo presentamos HDT-MR, una técnica para resolver los problemas de escalabilidad que surgen al construir HDT a gran escala. HDT-MR reduce el consumo de memoria empleado originalmente en su construcción, trasladando esta tarea al paradigma MapReduce. En este trabajo, presentamos el *workflow* distribuido de HDT-MR, evaluamos su rendimiento frente a una solución en un único nodo, en colecciones de hasta 9000 millones de triples RDF. Los resultados muestran como HDT-MR escala a cualquier tamaño en *clusters* con hardware de uso común, mientras que la solución original en un único nodo no es capaz de procesar tamaños mayores que 1000 millones de triples. HDT-MR, por tanto, reduce los requisitos hardware para procesar Big Semantic Data.

Nuestro trabajo futuro considera explotar los resultados de HDT-MR ya que la comunidad HDT puede disponer de colecciones HDT más grandes, promoviendo con ello el desarrollo de nuevas aplicaciones a gran escala. De igual modo,

consideramos combinar HDT y MapReduce para otras tareas como la consulta y razonamiento en Big Semantic Data.

## Agradecimientos

Este proyecto ha sido financiado por el programa de innovación e investigación Horizon 2020 de la Unión Europea gracias a la ayuda Marie Skłodowska-Curie No 642795, Austrian Science Fund (FWF): M1720-G11 y el Ministerio de Economía y Competitividad: TIN2013-46238-C4-3-R.

## Referencias

1. N.R. Brisaboa, A. Cerdeira-Pena, A. Fariña, and G. Navarro. A Compact RDF Store using Suffix Arrays. In *String Processing and Information Retrieval*, pages 103–115, 2015.
2. L. Cheng, A. Malik, S. Kotoulas, T.E. Ward, and G. Theodoropoulos. Efficient Parallel Dictionary Encoding for RDF Data. In *Proc. of WebDB*, 2014.
3. 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.
4. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Largee Clusters. In *Proc. of OSDI*, pages 137–150, 2004.
5. 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.
6. Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 3(2):158–182, 2005.
7. A. Joshi, P. Hitzler, and G. Dong. Logical Linked Data Compression. In *Proc. of ESWC*, pages 170–184, 2013.
8. F. Manola and R. Miller. *RDF Primer*. W3C Recommendation, 2004.
9. 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.
10. M.A. Martínez-Prieto, N.R. Brisaboa, F. Claude, R. Cánovas, and G. Navarro. Practical Compressed String Dictionaries. *Information Systems*, 56:73–108, 2016.
11. 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.
12. 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.
13. 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.
14. 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.