A MapReduce-based Approach to Scale Big Semantic Data Compression with HDT

J. M. Giménez, J. D. Fernández and M. A. Martínez

***Abstract*— Data generation and publication on the Web has increased over the last years. This phenomenon, usually known as "Big Data", poses new challenges related with Volume, Velocity, and Variety (*"The three V's"*) of data. The Semantic Web offers the means to deal with variety, where RDF (*Resource Description Framework*) is used to model data in the form of triples *subject-predicate-object*. In this way, it is possible to represent and interconnect RDF triples to build a true Web of Data. Nonetheless, a problem arises when big RDF collections must be stored, exchanges, and/or queried because the existing serialization formats are highly verbose, hence the remaining Big Semantic Data challenges (volume and variety) are aggravated when storing, exchanging, or querying big RDG collections. HDT addresses this issue by proposing a binary serialization format based on compact data structures that allows RDF to be compressed, but also to be queried without prior decompression. Thus, HDT reduces data volume and increases retrieval velocity. However, this achievement comes at the cost of and expensive RDF-to-HDT serialization in terms of computational resources and time. Therefore, HDT alleviates velocity and volume challenges for the end user, but moves Big Data challenges to the data publisher. In this work we show HDT-MR, a MapReduce-based algorithm that allows RDF datasets to be serialized to HDT in a distributed way, reducing processing resources and time, but also enabling larger datasets to be compressed.**

***Keywords*— Compression, HDT, MapReduce, RDF, Semantic Web, Web of Data**

I. Introducción

RDF (*Resource Description Framework*) [1] es 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 colecciones de datos heterogéneas, con diferentes grados de estructura, y procedentes de diferentes fuentes. La flexibilidad de RDF se debe a su estructura en forma de triple, en la que (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, que se puede visualizar como un grafo dirigido y etiquetado, donde los sujetos y objetos son los nodos del grafo y los predicados actúan como aristas. La serialización estándar de RDF comprende diferentes alternativas en texto plano, tales como *JSON-LD*, *RDF/XML* o los formatos basados en *Turtle*. Aunque esta serialización simplifica el procesamiento de triples y puede implementarse sin grandes sobrecargas computacionales, los ficheros resultantes tienden a ser muy voluminosos. Una solución usada en la práctica es utilizar un compresor universal, como *gzip*, para reducir el tamaño de estos ficheros. Sin embargo, esta solución solo es efectiva para el intercambio de datos, ya que su procesamiento 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 codifica la colección utilizando dos componentes principales: *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 descomprimirlos previamente [2]. Esta propiedad ha facilitado que HDT pase a considerarse una pieza básica en la construcción de motores de almacenamiento semánticos. *HDT-FoQ* [2] muestra el potencial de HDT para la resolución de algunas consultas SPARQL, mientras que *WaterFowl* [3] incorpora HDT en su motor de inferencia. Otras tecnologías como *Linked Data Fragments* [4] demuestran la eficiencia de HDT como motor de almacenamiento. Sin embargo, la construcción de los componentes *Diccionario* y *Triples* requiere un procesamiento exhaustivo de toda la colección RDF en memoria. Es por tanto un proceso poco escalable que dificulta la serialización de colecciones RDF de gran tamaño, cada vez más frecuentes en la Web de Datos.

En este artículo abordamos los problemas de escalabilidad de HDT y proponemos HDT-MR, una nueva implementación del proceso de serialización de RDF en HDT desarrollada sobre *MapReduce* [5]. Esta nueva solución mejora la escalabilidad del proceso original de construcción de HDT y nos permite serializar colecciones RDF de gran tamaño. En los experimentos realizados, HDT-MR es capaz de serializar colecciones 10 veces más grandes que las procesadas con la propuesta original y obtiene tiempos de codificación que crecen linealmente con el tamaño 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

J. M. Giménez, Univ Lyon, UJM-Saint-Etienne, CNRS, Laboratoire Hubert Curien UMR 5516, Saint Etienne (France), jose.gimenez.garcia@univ-st-etienne.fr

J. D. Fernández, Vienna University of Economics and Business, Vienna (Austria), jfernand@wu.ac.at

M. A. Martínez, DataWeb Research, Departmento de Informática, Univ. de Valladolid, Segovia (Spain), migumar2@infor.uva.es

Corresponding author: J. M. Giménez

características originales de formato, de cara a su explotación en el usuario final [2,6].

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

II. ANTECEDENTES

Esta sección comienza con un resumen general de los fundamentos del modelo *MapReduce* y del *framework de descripción de recursos* (RDF). A continuación, describimos la tecnología HDT, destacando su relevancia actual en el ámbito de la compresión de grandes conjuntos de datos RDF.

A. MapReduce

MapReduce [5] es un modelo de programación y *framework* optimizado para procesar grandes volúmenes de datos en entornos distribuidos. Su objetivo principal es ofrecer un mecanismo eficiente para la paralelización del procesamiento en el que se abstrae la complejidad al usuario.

El procesamiento MapReduce se construye sobre el concepto de *job*. Un *job* MapReduce comprende dos fases. La primera fase (*map*) lee los datos de entrada como pares clave-valor *(k1,v1)* y genera 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.

MapReduce implementa una arquitectura maestro-esclavo, en la que el nodo maestro se encarga de iniciar la tarea, distribuir la carga de trabajo dentro del *cluster* y procesar la información administrativa, mientras que los esclavos se responsabilizan de la ejecución de las tareas *map* y *reduce*.

MapReduce hace un uso exhaustivo de operaciones de entrada/salida *(I/O)*. Los esclavos leen y escriben datos en discos de manera local, de forma que los resultados temporales se almacenan en diferentes nodos del *cluster*. Esto favorece la *localidad de los datos*, minimizando la transferencia de datos en el cluster.

Apache Hadoop (http://hadoop.apache.org) es la implementación de MapReduce más utilizada actualmente. Hadoop introduce un modelo de almacenamiento distribuido denominado HDFS (*Hadoop Distributed File System*) que mejora la disponibilidad y permite incrementar la tolerancia a la ocurrencia de fallos.

B. RDF

El *framework* RDF [1] plantea un modelo de datos muy sencillo que permite describir recursos mediante sentencias ternarias, más conocidas como *triples RDF*. Por ejemplo, el triple *(ex:P1, foaf:age, 45*) indica que el recurso identificado como *"ex:P1"* tiene una propiedad (identificada como "*foaf:age"*) cuyo valor es *"45"*. Una interpretación más semántica de este triple vendría a decir que *"existe un determinado individuo, exP1, cuya edad es de 45 años"*. Los componentes de un triple RDF suelen referirse como *sujeto*, *predicado* y *objeto*, respectivamente.

Cada triple puede verse como un pequeño grafo etiquetado y dirigido en el que el nodo sujeto y el nodo objeto se relacionan (en sentido sujeto-objeto) mediante una arista etiquetada con el valor del predicado. Por lo tanto, el conjunto de todos los triples que forman parte de una colección RDF pueden interpretarse, a su vez, como un gran grafo dirigido en el que las etiquetas de las aristas establecen la semántica de las relaciones existentes entre los recursos y/o los valores de la colección. Se recomienda utilizar vocabularios estandarizados para la elección de los predicados utilizados en los triples, con el objetivo de facilitar su reutilización e integración en otras colecciones RDF (por ejemplo, en el ámbito de la iniciativa *Linked Open Data* antes mencionada).

RDF restringe los valores que pueden tomar los diferentes componentes de un triple:

1) Sujetos. Los sujetos siempre describen recursos, por lo que su valor debe ser el identificador de dicho recurso en el ámbito de la colección. RDF utiliza *URIs (Uniform Resource Identifiers)* para identificar, globalmente, los recursos, aunque también permite utilizar identificadores locales (referidos como *nodos anónimos* o *blank nodes*). En este artículo nos sólo consideraremos URIs para simplificar las explicaciones.
2) Predicados. El valor de los predicados siempre es un *URI* cuya semántica, preferiblemente, habrá sido caracterizada en un algún vocabulario estandarizado.
3) Objetos. El rol de objeto puede ser desempeñado tanto por un recurso como por un valor determinado (por ejemplo el *"45"* utilizado en el ejemplo anterior). En este caso, el objeto puede ser un *URI* o un valor *"literal"* de un tipo de datos determinado.

RDF define la estructura ternaria que deben tener las sentencias y restringe los diferentes valores que pueden tener cada uno de sus componentes. Sin embargo, no limita la forma en la que los triples se serializan (para más información sobre los formatos de serialización RDF, se recomienda consultar la sección 5 del RDF Primer anteriormente citado), tanto para propósitos de almacenamiento como de transmisión. Originalmente, las colecciones RDF se serializaron en ficheros XML. Este formato, conocido como *RDF/XML*, transforma el grafo RDF en una estructura arborescente restringida por las propiedades de XML. Propuestas como *NTriples* o *Turtle* fueron reemplazando, progresivamente, a *RDF/XML*. Ambos formatos consideran la estructura de grafo subyacente a la colección RDF y codifican los triples de acuerdo a su información de adyacencia. A pesar de que siguen siendo utilizados en la actualidad, el crecimiento en el tamaño de las colecciones RDF ha dejado patentes las debilidades de estos formatos. Por un lado, los ficheros obtenidos ocupan mucho espacio debido, principalmente, al uso de *URIs* de gran longitud que tienden a repetirse, en mayor o menor medida, a lo largo de la colección. Por otro lado, el coste de *parsear* y procesar estos ficheros también se ve penalizado por el hecho de tener que operar con grandes ficheros en los que cada triple RDF ocupa una número variable de bytes.

Como respuesta a esta situación, la comunidad de Web Semántica ha comenzado a adoptar soluciones basadas en

JSON. El formato *JSON-LD* es el mejor de ejemplo de ello. Sin embargo, esta solución aún sigue pagando (aunque de forma menos pronunciada) el coste de gestionar grandes volúmenes de datos. En este caso, la solución pasa por introducir la *compresión de datos*. Es bastante habitual encontrar proveedores de datos que publican sus colecciones RDF (serializadas en *NTriples*, *Turtle* o cualquiera de los otros formatos previamente explicados) comprimidas con técnicas bien conocidas como *gzip* o *bzip2*. Es una obviedad que el tamaño de estos ficheros es mucho menor que el de sus equivalentes sin compresión, reduciendo notablemente los costes de almacenamiento y/o transmisión. Sin embargo, para poder consumir estas colecciones RDF es necesario descomprimirlas previamente, retornando al problema original. En este contexto, es donde formatos de serialización como HDT ofrecen un valor añadido respecto a los que referimos como *"formatos tradicionales"*.

C. HDT

HDT [6,7] 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 consumo 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 Fig. 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 se utilizan como 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. Cada sección del *Diccionario* se codifica independientemente para aprovechar sus características particulares. La compresión de diccionarios [8] es un problema ortogonal al tratado en este trabajo, por lo tanto, no profundizaremos más sobre él.

Por su parte, 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): $(id_s\ id_p\ id_o)$, donde id_s, id_p e id_o son, respectivamente, los IDs de los términos sujeto, 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 sujeto, el nivel intermedio contiene la lista ordenada de los predicados que establecen las propiedades de dicho 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 [6].

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.

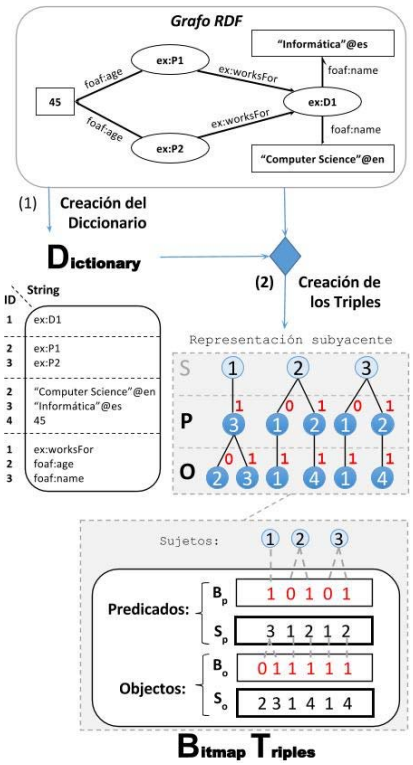

Figure 1. Diccionario y Triples (HDT) que codifican un grafo RDF.

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 los valores de sujeto, predicado y 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 finalizada la lectura del 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 [8]. 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 *ID-triples* y reemplaza los IDs iniciales (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. Este 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.

D. Trabajo Relacionado

Atendiendo a la taxonomía presentada en [9], HDT debe 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 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* [10] o *RDFCSA* [11] 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 complejo de compresión que 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 [12] 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 [9] 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 entorno MapReduce [13]. 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 [14] ha abordado este problema mediante otro algoritmo distribuido desarrollado en lenguaje X10. Ambos resultados constatan la necesidad de introducir tecnologías de procesamiento distribuido para comprimir grandes colecciones de datos RDF.

III. HDT-MR

HDT-MR se implementa mediante un proceso *MapReduce* que comprende dos etapas, una para la codificación de cada uno de los componentes de HDT (ver Fig. 2). Estas etapas se explican a continuación.

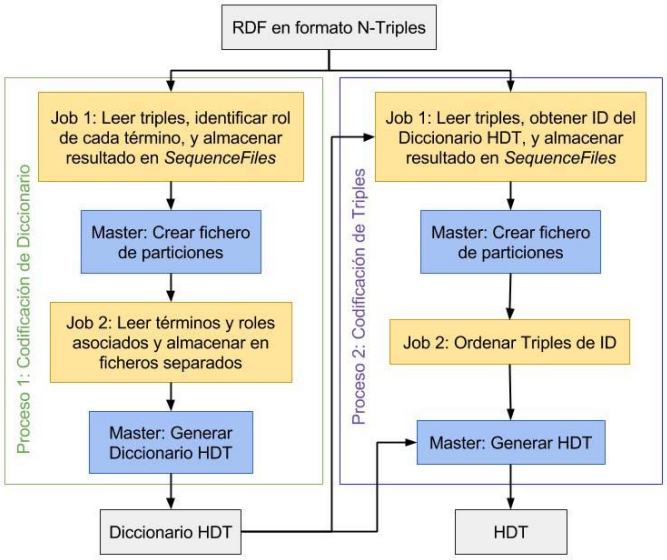

Figure 2. Descripción del workflow de HDT-MR.

A. Codificación del Diccionario

La primera etapa se centra en la construcción del *Diccionario* HDT, partiendo de la premisa de que la colección RDF original se encuentra en formato N-Triples. Para ello, es necesario identificar el rol que juega cada término en la colección, obtener las cuatro particiones (en orden lexicográfico) que describen el Diccionario, y codificarlas. Estas tareas se implementan mediante 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 los triples secuencialmente y emiten pares clave-valor en 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 Fig. 3 ilustra la ejecución de este *job* sobre el ejemplo de la Fig. 1, utilizando un cluster con dos esclavos.

Estos pares clave-valor, se ordenan y distribuyen entre los *reducers*, que se encargan de identificar los diferentes roles asociados a cada término. El *reducer* elimina pares duplicados y, cuando un término tiene asociados valores "S" y "O" simultáneamente, descarta estos pares y genera uno nuevo: *(término RDF, SO)*. El resultado del *job* consiste en tantas listas de pares *(término RDF, rol)* como número de *reducers* en el cluster. El Alg. 1 muestra el pseudocódigo que implementa este *job*.

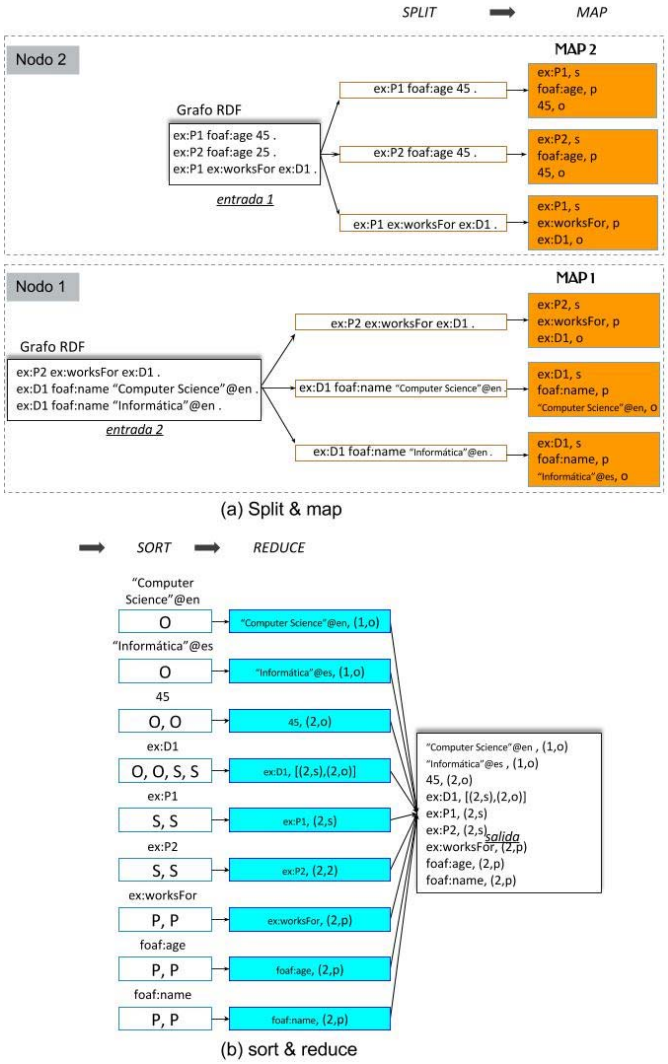

Figure 2. Job 1.1: Identificación de Roles

Algoritmo 1. Job 1.1: identificación de roles.

```
function MAP(key,value) ▷ key: número de línea ▷ value: triple
    emit(value:subject; "S")
    emit(value:predicate; "P")
    emit(value:object; "O")
end function
function combine/reduce(key,values) ▷ key: término RDF ▷ value: roles (S, P,
O)
    for role in values do
        if role contains "S" then isSubject   true
        else if role contains "P" then isP redicate   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
```

Job 1.2: Ordenación de los términos. Para construir las diferentes secciones de un *Diccionario* HDT es necesario que todos los términos estén ordenados lexicográficamente de manera global. El *job* anterior genera múltiples ficheros, cuyo contenido se encuentra localmente ordenado- Sin embargo, el resultado de concatenar todos estos ficheros no da lugar a un resultado globalmente ordenado. Esto se debe a que, por defecto, Hadoop usa una función *hash* sobre la clave con el objetivo de distribuir de la manera más uniforme posible el universo de claves entre el conjunto de *reducers*.

No obstante, el *framework* permite configurar diferentes políticas de distribución. La alternativa más sencilla, pero 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. Una segunda posibilidad consiste en crear, de manera manual, 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 de botella del proceso.

HDT-MR descarta las opciones anteriores y utiliza el *TotalOrderPartitioner* suministrado por Hadoop. Con él se realiza un muestre de los datos de entrada y se seleccionan las particiones que asignan de manera equitativa la muestra entre los *reducers* Sin embargo, este particionamiento no puede llevarse a cabo durante la ejecución del *job*, sino que tiene que realizarse antes de su inicio. Esta situación motiva la inclusión de este segundo *job*, cuyo objetivo es obtener una ordenación global de la salida del anterior. Así, se toma como entrada las listas de pares *(término RDF, rol)* y se agrupan de acuerdo a su valor. Para ello, se emplean *identity mappers*, que envían directamente su entrada a los *reducers* sin realizar procesamiento alguno. Por su parte, los *reducers* generan como salida una lista ordenada de los términos correspondientes a cada rol. La Fig. 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 Alg 2.

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 ocupa de la compresión de las cuatro secciones obtenidas en el *job* anterior. El proceso lee secuencialmente los términos de cada partición y los codifica utilizando la técnica *Plain Front-Coding* [8]. Este es un proceso sencillo que no presenta problemas de escalabilidad.

B. Codificación de los Triples

Esta segunda etapa realiza una nueva lectura del fichero original para construir el componente *Triples* de HDT. En este caso, es necesario reemplazar los términos RDF por su correspondiente ID en el *Diccionario* y codificar esta representación *ID-triples* de acuerdo al formato HDT.

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* leen los términos de cada triple y los reemplazan por sus IDs (obtenidos mediante operaciones de búsqueda en el Diccionario). Estos resultados se transmiten a

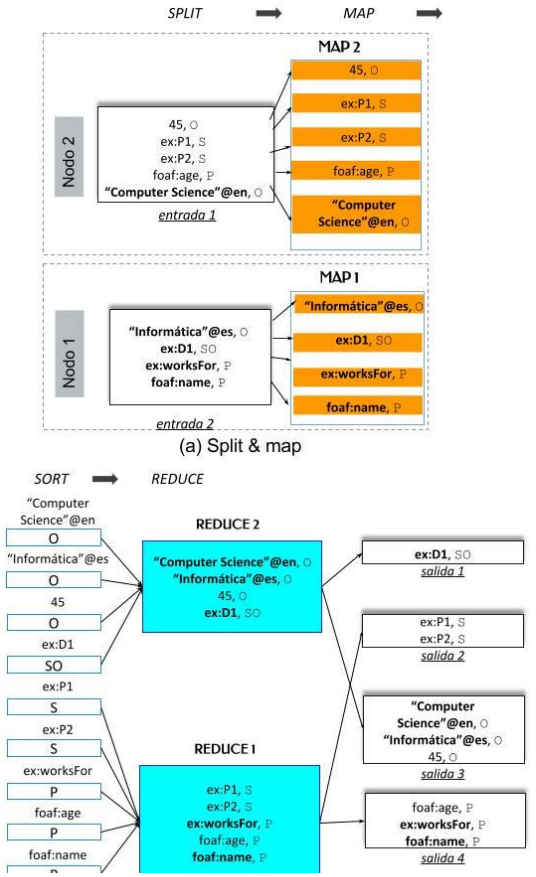

Figure 3. Job 1.2: ordenación de los términos.

identity reducers que, ordenan su entrada y emiten una lista de pares *(ID-triple, null)*, como puede verse en la Fig. 5 (omitimos *null* por simplicidad). El resultado final de este *job* (ver Alg. 3) comprende tantas listas ordenadas de *ID-Triples* como *reducers* hayan participado en la resolución del *job*.

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 isP redicate ← 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 isP redicate then emit to P(key; null)
    else if isObject then emit to O(key; null)
    end if
end function
```

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* anterior. 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 Fig. 6 ilustra este *job* (de nuevo los valores *null* se omiten en la salida).

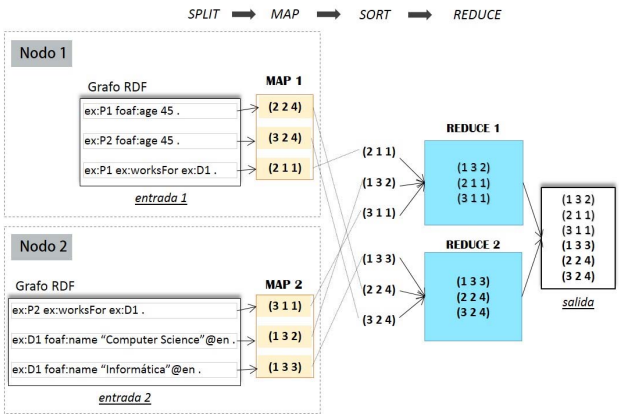

Figure 5. Job 2.1: construcción de ID-triples.

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
```

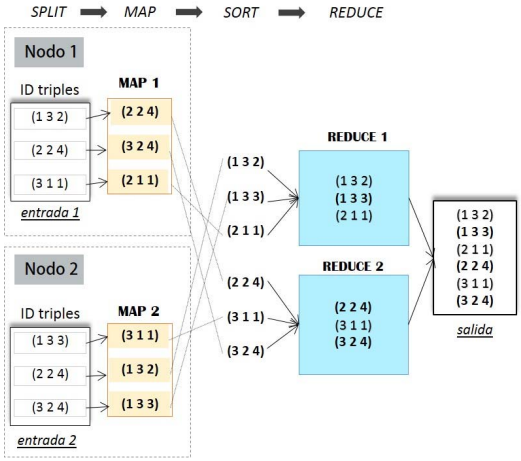

Figure 6. Job 2.2: Ordenación de ID-Triples.

Proceso Local 2.3: Codificación de los Triples HDT. Este proceso final se ejecuta en el nodo 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 en la Sec. II.C.

IV. Evaluación Experimental

Esta sección describe el estudio realizado para analizar el rendimiento de HDT-MR y lo compara con el obtenido por la propuesta HDT original (*mono-nodo*). Para ello, hemos desarrollado un prototipo de HDT-MR, (en *Hadoop*, versión 1.2.1), disponible en *http://goo.gl/Z5vl72* que usa la librería HDT-Java disponible en *http://code.google.com/p/hdt-java* (RC-2).

Consideramos esta misma librería HDT-Java como referencia original para la implementación *mono-nodo*. Nuestro entorno de experimentación (ver Tabla I) considera dos configuraciones computacionales. Por un lado, utilizamos un servidor de gran potencia en el que se despliega la implementación *HDT mono-nodo*. Por otro lado, HDT-MR se

ejecuta sobre un cluster con un nodo *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 en el cluster Hadoop *(HDT-MR)*.

En lo referente a las colecciones de datos (ver Tabla II), 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. *LinkedGeoData* (*http://linkedgeodata.org/Datasets*, versión 01-07-2013) es una colección de datos espaciales derivado de *Open Street Maps*, *DBPedia 3.8* (*http://wiki.dbpedia.org/Downloads38)* es una base de conocimiento extraída de Wikipedia, e *Ike* (*http://wiki.knoesis.org/index.php/LinkedSensorData*) incluye mediciones meteorológicas del huracán *Ike*. Además, realizamos combinaciones de colecciones dos a dos, y una combinación final que incluye las tres fuentes. Por otra parte, empleamos la herramienta LUBM [15] como generador de datos sintéticos (basado en modelar un sistema académico con un número parametrizable de universidades). Construimos "colecciones pequeñas" desde 1000 (134 millones de triples) a 8000 universidades (1068 millones de triples) y "colecciones grandes"' (añadiendo 4000 universidades en cada paso: aproximadamente 534 millones de triples), hasta un máximo de 72000 universidades (9586 millones de triples).

La Tabla II 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 esta tabla, la colección más grande ocupa 1318 GB en *NTriples*, mientras que comprimida con *lzo* requiere 95,5 GB.

TABLE I
ENTORNO EXPERIMENTAL

Máquina	Configuración
Nodo único	Intel Xeon E5-2650v2, 32x2.60GHz, 128GB RAM. Debian 7.8
Maestro	Intel Xeon X5675, 4x 3.07 GHz, 48GB RAM. Ubuntu 12.04.2
Esclavos	Intel Xeon X5675, 4x 3.07 GHz (4 núcleos), 8GB RAM. Debian 7.7

TABLE II
DESCRIPCIÓN DE LAS COLECCIONES DE DATOS DE PRUEBA.

Colección	Triples (mill.)	Tamaño (GB)			
		NT	NT+LZO	HDT	HDT+gz
LinkedGeoData	271	38.5	4.4	6.4	1.9
DBPedia	431	61.6	8.6	6.4	2.7
IKE	515	100.3	4.9	4.8	0.6
LDG+DBP	703	100.1	13.0	12.6	3.7
LDG+IKE	786	138.8	9.3	10.37	1.7
DBP+IKE	946	161.8	13.5	10.45	3.0
LDG+DBP+IKE	122	200.3	18.0	17.1	4.6
LUBM-1000	134	18.0	1.3	0.7	0.2
LUBM-2000	267	36.2	2.7	1.5	0.5
…	…	…	…	…	…
LUBM-7000	935	127.3	9.3	5.5	1.9
LUBM-8000	1068	145.5	10.6	6.3	2.2
LUBM-12000	1602	218.8	15.9	9.6	2.9
LUBM-16000	2136	292.4	21.2	12.8	3.8
…	…	…	…	…	…
LUBM-68000	9052	1244.4	90.2	57.6	18.9
LUBM-72000	9586	1318.0	95.5	61.3	20.0

La Fig. 7 muestra los tiempos de serialización de *HDT mono-nodo* y HDT-MR para las colecciones reales. Las Fig. 8 y 9 muestran los tiempos para las colecciones sintéticas. HDT mono-nodo obtiene un buen rendimiento en las colecciones de datos reales, y HDT-MR sólo puede competir en la colección *Ike*. No obstante, no se trata de un resultado inesperado, puesto que HDT mono-nodo se ejecuta en memoria principal mientras que HDT-MR paga el sobrecoste de las operaciones de I/O que realiza Hadoop. En cambio, HDT mono-nodo no es capaz de serializar las combinaciones de colecciones, 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 mono-nodo es mejor opción para las colecciones de datos pequeñas, pero la diferencia respecto a HDT-MR se reduce a medida que el tamaño de la colección se incrementa. HDT mono-nodo, 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 las 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.

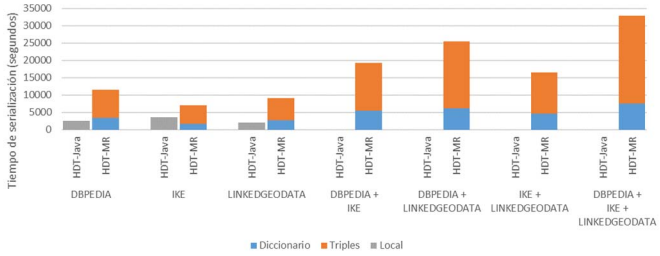

Figure 7. Tiempos de serialización de datasets reales y combinaciones.

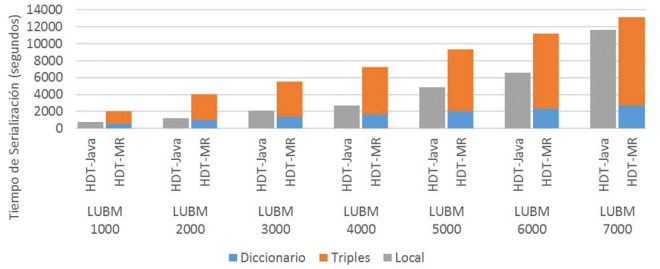

Figure 8. Tiempos de serialización de LUBM en conf. mono-noto y HDT-MR.

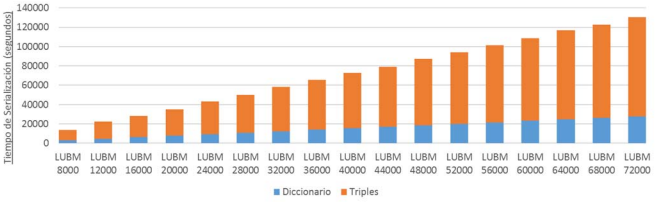

Figure 9. Tiempos de serialización de LUBM en HDT-MR.

Aunque el ratio de compresión no es el objetivo principal de este artículo, cabe mencionar los resultados obtenidos por HDT-MR, ya que ninguna de las técnicas propuestas en los artículos previos ha sido capaz de comprimir colecciones de

tamaños tan grandes. La Tabla II 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*.

V. Conclusiones

El uso de HDT para la compresión de RDF ha crecido progresivamente en los últimos años hasta convertirse en una referencia *de facto* en la comunidad de Web Semántica. Además, los índices que incluye HDT permiten recuperar RDF sin necesidad de descompresión previa, por lo que han surgido diferentes aplicaciones semánticas que emplean HDT como motor de almacenamiento RDF. Sin embargo, la adopción de HDT en entornos Big Data está fuertemente limitada por su falta de escalabilidad durante el proceso de compresión. En este artículo nos hemos enfrentado a este problema, introduciendo HDT-MR, una técnica que permite resolver los problemas de escalabilidad que surgen al construir HDT a gran escala. HDT-MR reduce el consumo de memoria empleado originalmente durante el proceso de compresión, trasladando esta tarea al paradigma MapReduce. Hemos presentado el *workflow* distribuido de HDT-MR, y evaluado su rendimiento frente a una solución mono-nodo, utilizando colecciones de hasta más de 9000 millones de triples RDF. Los resultados muestran como HDT-MR es capaz de comprimir de forma escalable grandes colecciones de RDF, utilizando un *cluster* construido sobre configuraciones y dispositivos computacionales de uso común. En cambio, la solución original, desplegada sobre una configuración hardware dedicada y de prestaciones notables, no es capaz de procesar colecciones cuyo tamaño supere los 1000 millones de triples. Este hecho no sólo es una demostración de la escalabilidad del algoritmo implementado por HDT-MR, sino que también es una prueba clara del ahorro que supondría utilizar este tipo de tecnologías para la procesar y explotar *Big Semantic Data*.

Agradecimientos

Este proyecto ha sido financiado parcialmente por el programa de innovación e investigación Horizon 2020 de la Unión Europea gracias a la ayuda Marie Sklodowska-Curie No 642795, Austrian Science Fund (FWF): M1720-G11 y el Ministerio de Economía y Competitividad (España): TIN2013-46238-C4-3-R y el programa de investigación e innovación Horizon 2020 de la Unión Europea: 731601.

Referencias

[1] F. Manola and R. Miller. RDF Primer. W3C Recommendation, 2004.
[2] 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.
[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] 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.
[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large 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] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres. Binary RDF representation for publication and exchange (HDT). W3C Member Submission, 2011. https://www.w3.org/Submission/HDT/.
[8] M.A. Martínez-Prieto, N.R. Brisaboa, F. Claude, R. Cánovas, and G. Navarro. Practical Compressed String Dictionaries. Information Systems, 56:73-108, 2016.
[9] 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.
[10] 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.
[11] 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.
[12] A. Joshi, P. Hitzler, and G. Dong. Logical Linked Data Compression. In Proc. of ESWC, pages 170-184, 2013.
[13] 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.
[14] L. Cheng, A. Malik, S. Kotoulas, T.E. Ward, and G. Theodoropoulos. Efficient Parallel Dictionary Encoding for RDF Data. Proc. of WebDB, 2014.
[15]. Y. Guo, Z. Pan, and J. Hein. LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics, 3(2):158-182, 2005.

José M. Giménez es actualmente estudiante de doctorado en la Universidad Jean Monnet de Saint-Étienne. Sus principales intereses son el tratamiento distribuido de datos semánticos, manejo eficiente de meta-datos semánticos, y estimación de la veracidad y procedencia de datos abiertos.

Javier D. Fernández es Doctor en Informática por la Universidad de Valladolid y la Universidad de Chile y actialmente es investigador en la Universidad de Economía de Viena. Su trabajo se enmarca en el ámbito del manejo eficiente de grandes volúmenes de datos semánticos, con especial atención a la seguridad (privacidad, acceso, encriptación).

Miguel A. Martínez es Doctor en Informática por la Universidad de Valladolid (2010) y, actualmente, es profesor del Departamento de Informática de la misma Universidad. Su trabajo de investigación actual se encuadra en el área de la compresión de datos, con especial interés en el diseño y utilización de estructuras de datos compactas en problemas relacionados con la gestión de Big Data. Ha publicado más de 60 trabajos de investigación en esta área, destacando su contribución al desarrollo del formato HDT (*W3C Member Submission*).