v-RDFCSA: Compresión e Indexación de Colecciones de Versiones RDF

Ana Cerdeira-Pena[1], Antonio Fariña[1], Javier D. Fernández[2], and Miguel A. Martínez-Prieto[3]

[1] Laboratorio de Bases de Datos, Universidade da Coruña
acerdeira@udc.es, fari@udc.es
[2] Vienna University of Economics and Business (WU)
jfernand@wu.ac.at
[3] DataWeb Research, Universidad de Valladolid
migumar2@infor.uva.es

Resumen La gestión de grandes colecciones RDF es un tema de gran interés en la Web de Datos, pero cuyas técnicas más relevantes no van más allá de una simple visión estática de la colección, dejando al margen la problemática relacionada con su evolución temporal. Sin embargo, las colecciones evolucionan para actualizar la descripción de su dominio y, con ello, generan múltiples versiones que precisan un almacenamiento efectivo de cara a su explotación por diferentes tipos de aplicaciones semánticas. En este artículo proponemos una nueva técnica para la compresión de colecciones de versiones RDF. Nuestra propuesta (v-RDFCSA) extiende el autoíndice RDFCSA con estructuras de bits que codifican la información de versionado. De esta manera, preservamos los triples RDF en espacio comprimido y, sobre esta representación, resolvemos operaciones de consulta temporales basadas en patrones SPARQL. Nuestra evaluación muestra que v-RDFCSA reduce los requisitos de almacenamiento entre 35 y 60 veces respecto al estado del arte actual y consigue más de un orden magnitud de ventaja en la resolución de consultas.

1. Introducción

El uso de RDF (*Resource Description Framework*) [14] se ha generalizado durante la última década. Proyectos de datos abiertos, como *Linked Open Data*, o iniciativas colaborativas, como *schema.org*, han promovido el uso de RDF como estándar *de facto* para la descripción, en la Web, de entidades procedentes de diferentes campos de conocimiento. Colecciones RDF de datos biológicos, publicaciones científicas, multimedia o datos gubernamentales son sólo algunos ejemplos de la variedad que podemos encontrar en la *Web de Datos*.

La Web de Datos plantea un espacio de conocimiento que está creciendo de forma progresiva [12] y en el que, además, los contenidos publicados en cada colección evolucionan con el objetivo de describir los nuevos hechos que se producen a lo largo del tiempo. Por ejemplo, *DBpedia live*[4] actualiza diariamente sus contenidos para incorporar los cambios producidos durante ese día en Wikipedia. Estos cambios dan lugar a nuevas versiones de la colección que, en la

[4] http://live.dbpedia.org/

práctica, tienden a ser similares a sus predecesoras. La última versión es la que está vigente en un momento dado, pero también es necesario preservar las versiones previas con el objetivo de satisfacer necesidades como, por ejemplo, el análisis evolutivo los datos o la posibilidad de revertir los cambios realizados.

El presente problema es bastante reciente en la Web de Datos, pero tiene un precedente claro en la *World Wide Web*. En este caso, los archivos contienen múltiples versiones de páginas web y su gestión presenta problemas de escalabilidad importantes en su almacenamiento y explotación [9]. Aunque el nivel de escala que se presenta en la Web de Datos es, por el momento, inferior al de la WWW, el reto es comparable, y abre con ello una nueva línea de investigación relacionada con la preservación de colecciones históricas de RDF (referidas como *archivos RDF*) y la necesidad de poder consultar estos datos de una manera eficiente [6]. El estado del arte presenta algunas estrategias para afrontar este nuevo reto (ver Sección 2.1), pero ninguna de ellas puede considerarse efectiva atendiendo a las necesidades de almacenamiento que demandan. Por ejemplo, dichas estrategias utilizan hasta 15 veces más espacio que el que requiere el compresor *gzip* para almacenar un archivo RDF de referencia [8]. Estos números están muy alejados de los que se obtienen al utilizar técnicas específicas para la compresión de archivos web [4]. Dichas técnicas no sólo consiguen reducir drásticamente las necesidades de almacenamiento (en algunos casos, utilizan menos de un 2 % del espacio originalmente ocupado por el archivo), sino que también permiten realizar búsquedas eficientes sobre los datos comprimidos.

La brecha existente ente ambos escenarios fundamenta la investigación formulada en este trabajo, considerando que ya existen técnicas específicas para la compresión de RDF que, a su vez, facilitan la resolución eficiente de patrones básicos de consulta [7, 1, 2]. Sin embargo, todas estas técnicas se ciñen a la visión estática de la colección RDF y no han considerado todavía ningún tipo de solución para afrontar las necesidades subyacentes a la gestión de sus versiones.

En este artículo presentamos `v-RDFCSA`, la primera técnica específica para la compresión de archivos RDF que, a su vez, proporciona algoritmos eficientes para la resolución de operaciones de consulta sobre ellos. `v-RDFCSA` identifica el conjunto de todos los triples diferentes usados en el archivo y los comprime con RDFCSA [2], un *autoíndice* capaz de resolver consultas SPARQL sobre la representación comprimida de la colección. Por otro lado, `v-RDFCSA` utiliza secuencias de bits para codificar, de forma sucinta, la información de versionado. Nuestros experimentos, utilizando el *benchmark* BEAR [8], muestran que `v-RDFCSA` usa apenas $5{,}7 - 7{,}3$GB de espacio para almacenar un archivo RDF de 325GB y resuelve las operaciones de consulta estudiadas un orden de magnitud más rápido que el *baseline* considerado.

El resto del artículo se organiza de la siguiente manera. En la Sección 2 planteamos una revisión general de los conceptos necesarios para entender nuestra propuesta, cuya explicación abordamos en la Sección 3. La Sección 4 muestra una evaluación experimental de `v-RDFCSA` y compara su rendimiento respecto al obtenido por un *baseline* de referencia. Finalmente, la sección 5 resume las conclusiones obtenidas con este trabajo y las líneas de trabajo futuro.

2. Trabajo Relacionado

Los archivos RDF utilizan dos tecnologías principales: el modelos de datos RDF y el lenguaje de consulta SPARQL. RDF [14] describe hechos en forma de estructuras ternarias, llamadas *triples* (o tripletas). Cada triple `(sujeto, predicado, objeto)` describe una propiedad (*predicado*) de un *sujeto*, otorgándole un valor (*objeto*) concreto. Por ejemplo, los triples `(JohnDoe, age, 45)` y `(JohnDoe, email, john@example.org)` establecen la edad y el correo electrónico del sujeto *JohnDoe*. En la práctica, cada uno de los triples puede verse como un grafo que conecta los nodos sujeto y objeto mediante una arista etiquetada con el predicado correspondiente. Por tanto, una colección de triples conforma un grafo etiquetado y dirigido que representa una base de conocimiento sobre un conjunto de entidades. Por su parte, SPARQL [11] es un lenguaje de consulta para grafos RDF, con una expresividad (y sintaxis) similar a SQL, pero basado en la correspondencia entre patrones de grafo. Una consulta SPARQL está formada por patrones de triples, esto es, triples en los que cada uno de los componentes puede ser una variable que se debe confrontar con el grafo RDF que se consulta. Operadores más complejos, como *joins*, uniones, patrones opcionales, filtros y otros modificadores de consulta, completan la funcionalidad de este lenguaje.

Archivo RDF. Un archivo RDF organiza las versiones de una colección RDF, anotando los triples con las versiones a las que pertenecen. Un *triple anotado con versión* [8] es, por tanto, un triple RDF `(sujeto, predicado, objeto)` con una etiqueta $i \in [1, \mathcal{N}]$ que representa la versión en la que es válido dicho triple. Por lo tanto, un *archivo RDF*, $\mathcal{A}$, es un conjunto de triples anotados con versión[5].

La Figura 1 ilustra un archivo RDF con 3 versiones en las que se describe información sobre jugadores y entrenadores del club de fútbol *"Atlético de Madrid"*. En la primera versión, V_1, se representan dos jugadores, `ex:Torres` y `ex:Simeone`, y que el entrenador del equipo es `ex:Manzano`. Ambos jugadores abandonan el equipo en la versión V_2, al tiempo que se contrata al jugador `ex:Falcao`. Para finalizar, en la última versión, V_3, `ex:Simeone` vuelve al equipo, pero en calidad de entrenador, reemplazando a `ex:Manzano`, mientras que `ex:Falcao` cede su puesto de jugador a `ex:Torres`, quien también regresa al equipo.

Funcionalidad de Consulta. Los archivos RDF proporcionan funcionalidad de consulta SPARQL sobre una o más versiones, o sobre las diferencias (*deltas*) existentes entre dos o más versiones dadas. Dichas funcionalidades se pueden soportar sobre la base de tres primitivas básicas [6]:

- *Materialización de versiones*: $Mat(Q, V_i)$, devuelve los resultados que satisfacen la consulta SPARQL Q en la versión V_i. Por ejemplo, en el caso anterior, la consulta $Mat($`(ex:Atletico,ex:hasCoach,?x)`$, V_2)$ obtendría que `ex:Manzano` era el entrenador del Atlético de Madrid en la versión V_2.

[5] Nótese que es posible anotar un mismo triple con diferentes etiquetas (versiones).

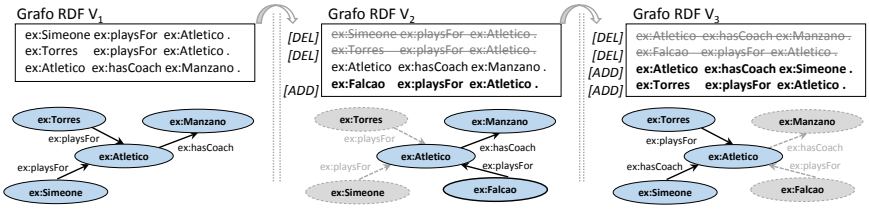

Figura 1. Ejemplo de versiones de un grafo RDF.

- *Materialización de diferencias (Delta)*: $Diff(Q, V_i, V_j)$, devuelve como resultado los valores que satisfacen Q en V_i, pero que no aparecen en V_j (*resultados borrados*), y viceversa (*resultados añadidos*). Por ejemplo, la consulta $Diff($`(?x,playsFor,ex:Atletico)`$, V_1, V_2)$ devolvería `ex:Simeone` y `ex:Torres` como resultados borrados en V_2, y `ex:Falcao` como un resultado añadido, en la misma versión.
- *Consulta de versiones*: $Ver(Q)$, resuelve Q sobre el archivo completo y anota los resultados con las versiones en las aparecen los valores recuperados. Por ejemplo $Ver($`(ex:Atletico,hasCoach,?x)`$)$ obtendría que `ex:Manzano` es un resultado válido en V_1 y V_2, mientras que `ex:Simeone` lo sería en V_3.

Estas primitivas pueden combinarse (con la semántica definida en SPARQL) para elaborar consultas más complejas. Por ejemplo, conocer qué jugadores de un equipo han sido también entrenadores se podría resolver mediante i) $Mat($`(?x,ex:playsFor,ex:Atletico)`$, V_i) \bowtie Mat($`(ex:Atletico,ex:hasCoach,?x)`$, V_j)$, $\forall i, j \in N$; o ii) realizando un join de dos consultas de versiones: $Ver($`(?x, ex:playsFor,ex:Atletico)`$) \bowtie Ver($`(ex:Atletico,ex:hasCoach,?x)`$)$.

2.1. Estado del Arte

En la actualidad, existen tres estrategias principales para representar archivos RDF [6]. Una primera aproximación consiste en mantener *copias independientes* (Ic, *independent copies*) de manera que cada versión se trata como un colección RDF independiente [13]. Esta representación conlleva un importante sobrecoste en espacio, ya que los triples que aparecen en dos o más versiones se almacenan múltiples veces. A cambio, las consultas que sólo afectan a una versión se resuelven eficientemente, mientras que el resto de primitivas (como la diferencia entre versiones) resultan altamente penalizadas.

Una aproximación totalmente contrapuesta, *basada en cambios* (Cb, *Change-based*), representa las diferencias entre dos versiones consecutivas. Es decir, la versión V_i se representa de acuerdo a sus diferencias (triples añadidos y borrados) respecto a la versión anterior $V_{(i-1)}$. Esta reorganización permite reducir notablemente el espacio requerido y favorece a aquellas consultas que operan sobre las diferencias entre versiones. Sin embargo, las consultas que requieren materializar una versión se ven afectadas negativamente de forma significativa ya que es necesario propagar todos los cambios producidos hasta la versión actualmente consultada. Para minimizar este problema, se suele optar por almacenar una

versión completa cada k deltas [5], lo que se traduce en un aumento moderado de los requisitos de almacenamiento.

Por último, las aproximaciones *basadas en marcas de tiempo* (TB, *timestamp-based*) [18] consideran una única colección RDF que contiene el conjunto de todos los triples diferentes que aparecen en alguna versión. Para cada uno de ellos, se almacena una marca de versión (habitualmente, cuándo fueron añadidos o borrados). Esta representación favorece la resolución de consultas de versiones pero, en la práctica, requiere de índices adicionales, sobre la información de las marcas, que permitan acceder a los contenidos de una versión de manera eficiente. Obviamente, estos índices adicionales también tienen un impacto no despreciable en los requisitos del almacenamiento.

Recientemente se ha publicado un trabajo sobre evaluación de archivos RDF en el que se estudian cada una de las estrategias anteriores [8]. Este estudio muestra cómo cada una de las opciones destaca en alguna de las funcionalidades requeridas, pero todas ellas tienen un importante sobrecoste en espacio. En términos cuantitativos, las soluciones estudiadas requieren entre 200 y 350GB de almacenamiento para representar un archivo RDF cuyo tamaño en gzip es de, apenas, 23GB. Atendiendo a estos resultados, es indudable que la gestión de archivos RDF requiere utilizar alguna forma de compresión.

2.2. Compresión RDF

HDT [7] fue el precursor de los compresores RDF basados en estructuras de datos compactas. Su aproximación consiste en transformar el grafo RDF en un conjunto de árboles que organizan, para cada sujeto, todos los pares (predicado, objeto) con los que se relaciona. Estos árboles se codifican utilizando secuencias de bits que proporcionan operaciones `rank/select` [10], mediante las cuales podemos navegar las ramas del árbol y resolver patrones de triples SPARQL en un espacio comprimido. Por otra parte, k^2-triples [1] se centra en aprovechar las redundancias estructurales existentes en RDF, para mejorar la efectividad de HDT. Para ello, transforma el grafo RDF en un conjunto de matrices de adyacencia (sujeto, objeto), una por cada predicado en la colección, y las comprime utilizando k^2-trees [3] (aprovechando que dichas matrices son muy poco densas). Por último, RDFCSA [2] emplea *arrays de sufijos* comprimidos e indexa los triples como *strings* cíclicos. Aunque RDFCSA no genera las representaciones más comprimidas, su rendimiento es muy competitivo y, además, sus resultados son muy estables y predecibles para todos los tipos de consultas[6]. Por este motivo, en este trabajo elegimos RDFCSA y extendemos su funcionalidad para representar y consultar archivos RDF.

3. Autoindexación de Archivos RDF (`v-RDFCSA`)

El diseño de `v-RDFCSA` parte de la experiencia adquirida por el uso de técnicas ya existentes en el estado del arte. Más concretamente, el diseño de `v-RDFCSA` se

[6] El rendimiento de HDT y k^2-triples depende del número de variables en la consulta y de su localización en el patrón, así como de la estructura de la colección de datos.

materializa como una aproximación TB *ligera* que codifica independientemente i) el conjunto de triples diferentes del archivo RDF, y ii) la información relativa a las versiones en las que se utiliza cada uno de ellos. Los triples se comprimen utilizando un autoíndice y las versiones se codifican usando representaciones sucintas de secuencias de bits. Los algoritmos de consulta diseñados en `v-RDFCSA` explotan las capacidades de autoindexación de RDFCSA [2] para recuperar los triples comprimidos y después realizan operaciones, orientadas a bit, sobre la información de versionado. Tanto los mecanismos de compresión como los algoritmos de búsqueda se explican a continuación.

3.1. Codificación de Triples RDF

`v-RDFCSA` maneja sólo el conjunto de triples diferentes utilizados en el archivo RDF. Estos *triples (sin considerar versiones)* [8] son una pequeña parte de los triples totales; por ejemplo, el archivo BEAR [8] contiene $\approx 2 \times 10^9$ triples, pero sólo ≈ 376 millones sin considerar versiones. Por ello, el problema de la codificación de triples RDF se puede reducir a la compresión tradicionalmenre realizada sobre una colección RDF habitual. En este caso, hemos elegido RDFCSA [2], un autoíndice basado en el *array de sufijos* comprimido de Sadakane (CSA) [17], que indexa un conjunto de triples en lugar de texto, y mantiene toda la funcionalidad a la hora de buscar patrones del CSA.

En primer lugar, RDFCSA transforma los triples originales usando un diccionario. Esta estructura permite reemplazar los *términos RDF* por identificadores enteros (IDs) en los rangos $[1, n_s]$ para sujetos, $[1, n_p]$ para predicados y $[1, n_o]$ para objetos. La Figura 2 (izquierda) muestra el conjunto de triples, sin considerar versiones, usados en el archivo descrito previamente. Dicho conjunto contiene $n = 5$ triples diferentes que, en la parte central, son trasformados a una representación basada en IDs[7]. El mapeo entre términos RDF e IDs se indexa independientemente usando diccionarios de texto comprimidos [15].

En la parte derecha de la figura se muestran los cuatro pasos que se realizan para convertir los triples basados en IDs en un autoíndice RDFCSA. En el primer paso, la secuencia S_{id} es una lista de ID-triples ordenados. El primer triple se almacena en $S_{id}[1, 3]$, el segundo en $S_{id}[4, 6]$, y así sucesivamente. Estos triples basados en IDs se reescriben en el paso 2 para evitar solapamientos de IDs entre sujetos, predicados y objetos. Los IDs de los sujetos no cambian y se mantienen entre $[1, n_s]$. Los IDs de predicados pasan al rango $[n_s + 1, n_s + n_p]$, y los valores para objetos a $[n_s + n_p, n_s + n_p + n_o]$. Así, el triple $(1, 1, 2)$ se transforma en $(1, 5, 8)$. Esta reasignación de IDs asegura que cada ID de sujeto es menor que cualquier ID de predicado y, a su vez, que éstos son más pequeños que cualquier ID de objeto. Esta decisión da lugar a una configuración de array de sufijos particular (paso 3) en la que los sujetos aparecen en el rango $SA[1, n]$, los predicados en $SA[n + 1, 2n]$, y los objetos en $SA[2n + 1, 3n]$. Finalmente, este array de sufijos se comprime (paso 4) usando las estructuras D y ψ de un

[7] Nótese que $n_s = 4$, $n_p = 2$, $n_o = 3$, y que los IDs 1 y 2 se usan tanto para sujetos como para objetos.

CSA[17]. $D[1,n]$ es una secuencia de bits donde los $1s$ indican el primer sufijo en SA en el que comienza cada símbolo diferente del alfabeto. A su vez, el array $\psi[1,n]$ permite recorrer el array de sufijos aprovechando que $SA[\psi[i]] = SA[i]+1$. Esto es, si $SA[i] = j$ apunta al sufijo $S[j,n]$, entonces $SA[\psi[i]] = j+1$ apunta al siguiente sufijo del texto $S[j+1,n]$. RDFCSA modifica la región $\psi[2n+1,3n]$, en la que se codifica la información de saltos desde los objetos. Originalmente, ψ permite saltar desde el objeto del k-ésimo triple al sujeto del triple $(k+1)$. Esto no es útil para resolver consultas SPARQL, ya que relaciona dos triples independientes. Nuestra decisión pasa por modificar ψ para que los valores $\psi[2n+1,3n]$ apunten al sujeto del mismo triple. Esto es, $\psi[i] \leftarrow \psi[i] - 1, \forall i \in [2n+1,3n]$ (o $\psi[i] \leftarrow n$ if $\psi[i] = 1$). De esta forma, conseguimos una codificación cíclica que comprende a los tres elementos del triple.

D y ψ permiten resolver *triple patterns* SPARQL mediante una búsqueda binaria inicial, seguida por un recorrido que recupera los triples que satisfacen el patrón buscado. En [2] se describen dichos algoritmos en profundidad.

3.2. Codificación de la Información de Versionado

La organización y codificación de los triples en RDFCSA permite que cada uno de ellos pueda identificarse fácilmente de acuerdo a la posición de su sujeto en SA. Si (s_a,p_b,o_c) es el k-ésimo triple ($1 \leq k \leq n$), podemos recuperarlo sin más que obtener su sujeto como[8] $s_a \leftarrow S[SA[k]]$, su predicado como $p_b \leftarrow S[SA[\psi[k]]]$, y su objeto mediante $o_c \leftarrow S[SA[\psi[\psi[k]]]]$.

Esta propiedad es básica para implementar las dos estrategias de codificación de la información de versionado que proponemos. Supongamos un archivo $\mathcal{A}$, que contiene N versiones diferentes y un conjunto de n *triples sin considerar versiones*. La primera estrategia de codificación (llamada `tpv`: *triples por versión*) usa N secuencias de bits $\mathcal{B}_i^v[1,n]$ para codificar los triples que aparecen en la correspondiente versión i. Esto es, si $\mathcal{B}_i^v[k] = 1$, el k-ésimo triple aparece en la versión i; en caso contrario, $\mathcal{B}_i^v[k] \leftarrow 0$. La Figura 3 (izquierda) ilustra la codificación `tpv` para el archivo de la Figura 1 (por ejemplo, la segunda versión contiene los triples 2 y 4). Nuestra segunda estrategia (llamada `vpt`: *versiones por triple*) utiliza n secuencias de bits $\mathcal{B}_i^t[1,N]$ para codificar las versiones donde aparece el k-ésimo triple. Si $\mathcal{B}_k^t$ describe el k-ésimo triple, entonces $\mathcal{B}_k^t[i] = 1$ significa que el k-ésimo triple aparece en la i-ésima versión. En caso contrario, $\mathcal{B}_k^t[i] \leftarrow 0$. La Figura 3 (derecha) muestra la codificación `vpt` para el archivo de ejemplo. Véase como el segundo triple se usa en las versiones 1 y 2.

Ambas estrategias utilizan $N * n$ bits (`tpv` incluye N secuencias de n bits cada una y `vpt` utiliza n secuencias de N bits), pero su rendimiento de búsqueda es diferente. Los algoritmos correspondientes se describen a continuación.

3.3. Algoritmos de Consulta

La explicación de los algoritmos de consulta deja de lado toda la operativa relacionada con el manejo del diccionario de *strings* y asume que tanto las

[8] $S[SA[i]] = rank_1(D,i)$, donde $rank_1$ indica el número de unos en $D[1,i]$.

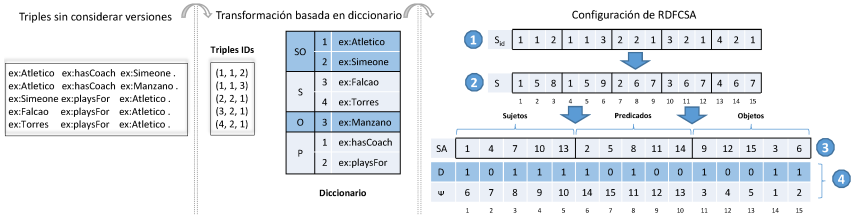

Figura 2. Construcción paso a paso del RDFCSA para el archivo RDF.

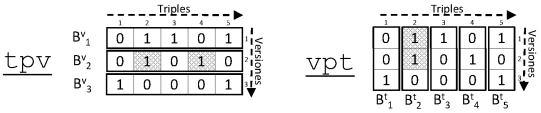

Figura 3. Codificación de la información relativa a versiones del archivo RDF.

consultas, y sus consiguientes resultados, se expresan utilizando los IDs que aparecen en S_{id} (sirva como ejemplo la Figura 2). Estas operaciones de traducción entre términos RDF e IDs se implementan utilizando primitivas de localización y extracción de *strings* en el diccionario [15].

Todos nuestros algoritmos acceden, inicialmente, a RDFCSA para recuperar el conjunto de triples candidatos que se obtiene como resultado de resolver un patrón de consulta Q' dado. Para cada triple candidato se registra la posición $SA[k]$ en la que su sujeto aparece dentro del array de sufijos. A partir de ahí, es necesario recorrer los triples candidatos y acceder a su información de versionado, con el fin de realizar las verificaciones necesarias para resolver cada consulta: si el k-ésimo triple aparece en la versión i ($Mat(Q', i)$); si un triple cambia de la versión i a la j ($Diff(Q', i, j)$); o para recuperar todas las versiones en las que aparece un triple dado ($Ver(Q')$). Nótese que los sujetos de todos los triples candidatos para las operaciones `(s??)`, `(sp?)` y `(spo)` dan lugar a un rango continuo $SA[l, r]$ en el array de sufijos[2]. Esto nos permite optimizar el acceso a la información de versiones durante el recorrido de los triples candidatos.

Consultas de materialización de versiones: $Mat(Q', i)$, recupera los triples que satisfacen Q' para una versión dada i. Una vez que las posiciones de los sujetos se han recuperado de RDFCSA, se usa la información de versionado para descartar los triples que no aparencen en la versión i. En `vpt`, cada triple candidato k se verifica mediante un acceso a la secuencia de bits $\mathcal{B}^t_k$. Si $\mathcal{B}^t_k[i] = 1$ se devuelve dicho triple, descartándolo en caso contrario. En `tpv` el proceso es similar: si $\mathcal{B}^v_i[k] = 1$, recuperamos el triple apuntado desde $SA[k]$; en caso contrario también se descarta. Los patrones `(s??)`, `(sp?)` y `(spo)` pueden optimizarse en `tpv`, ya que los sujetos de los triples candidatos son contiguos en $SA[l, r]$. Dado $c_l \leftarrow rank_1(\mathcal{B}^v_i, l)$ y $c_r \leftarrow rank_1(\mathcal{B}^v_i, r)$, el número de triples activos en la versión i dentro del rango $[l, r]$ es $c = c_r - c_l + 1$. Por lo tanto, para resolver $Mat(Q', i)$ simplemente recuperamos los triples en las posiciones $k \leftarrow select_1(\mathcal{B}^v_i, j), \forall j \in [c_l, c_r]$[9].

[9] $p \leftarrow select_1(B, j)$, indica la posición p del j-ésimo uno una secuencia de bits B.

Consultas de materialización de diferencias: $Diff(Q', i, j)$, busca los triples que satisfacen Q' y que hayan sido añadidos o borrados entre las versiones i y j. Al igual que en el caso anterior, el algoritmo recupera de RDFCSA todos las triples candidatos para Q' y, después, procesa la información de versiones para cada uno de ellos (cuyo sujeto es apuntado desde $SA[k]$). En `vpt`, esto implica dos accesos a $\mathcal{B}_k^t$: se recuperan los valores de los bits: $x \leftarrow \mathcal{B}_k^t[i]$ e $y \leftarrow \mathcal{B}_k^t[j]$, que indican si el triple k aparece en las versiones i y j, respectivamente. Análogamente, en `tpv`, se recuperan los valores $x \leftarrow \mathcal{B}_i^v[k]$ e $y \leftarrow \mathcal{B}_j^v[k]$.

A continuación, se usan los operadores $\overline{\texttt{xor}}$ y $\overline{\texttt{and}}$ para verificar cada triple candidato: *i)* si $(0 \neq ((x \ \overline{\texttt{xor}} \ y) \ \overline{\texttt{and}} \ x))$ el triple estaba activo en la versión i, pero se eliminó en la versión j. Por ello, se devuelve como un triple *eliminado*. *ii)* si $(0 \neq ((x \ \overline{\texttt{xor}} \ y) \ \overline{\texttt{and}} \ y))$ el triple no estaba en la versión i pero sí aparece en la versión j (se devuelve como un triple *añadido*). *iii)* En otro caso, el triple se descarta.

En lugar de recorrer todas los triples candidatos, en `tpv` podemos optimizar este proceso cuando dichos triples son adyacentes en $SA[l, r]$. Dada la secuencia de bits B, sea $getNext_1(B, pos)$ una función que devuelve pos si $B[pos] = 1$; y en caso contrario, devuelve la posición del siguiente 1 después de pos en B mediante $select_1(B, 1 + rank_1(B, pos))$. Dado el rango $[l, r]$, y usando $getNext_1$, podemos resolver $Diff(Q', i, j)$ en `tpv` como sigue. Calculamos $p_1 \leftarrow getNext_1(\mathcal{B}_i^v)$ y $p_2 \leftarrow getNext_1(\mathcal{B}_j^v)$. A continuación, recorremos en paralelo $\mathcal{B}_i^v$ y $\mathcal{B}_j^v$ en las posiciones que almacenan *unos*: p_1 y p_2, mientras se cumpla que $((p1 \leq r)$ *or* $(p2 \leq r))$. En cada iteración chequeamos: *i)* si $(p_1 < p_2)$, entonces sabemos que el triple en la posición p_1 fue eliminado en la versión j, y avanzamos $p_1 \leftarrow getNext_1(\mathcal{B}_i^v, p_1)$; *ii)* si $(p_2 < p_1)$ entonces el triple en la posición p_2 fue añadido en la versión j, y avanzamos $p_2 \leftarrow getNext_1(\mathcal{B}_j^v, p_2)$; *iii)* en caso contrario, calculamos $p_1 \leftarrow getNext_1(\mathcal{B}_i^v, p_1)$, $p_2 \leftarrow getNext_1(\mathcal{B}_j^v, p_2)$, y continuamos con la siguiente iteración.

Consultas de versiones: $Ver(Q')$, recupera todos los triples que satisfacen Q' y, para todos ellos, devuelve la lista de versiones en las que estaban activos. Por tanto, para cada triple candidato k, se verifica si aparece en la i-ésima versión. Esto es, $\forall i \in [1, N]$, comprobamos si el bit $\mathcal{B}_k^t[i]$ es 1, en la estrategia `vpt`, o si el bit $\mathcal{B}_i^v[k]$ es 1 en `tpv`. De nuevo es posible realizar optimizaciones en los patrones `(s??)`, `(sp?)` y `(spo)` en `tpv`. Para ello, implementamos un algoritmo de bucle anidado que recorre las N secuencias de bits $\mathcal{B}_i^v, i \in [1, N]$ y, para cada una de ellas, ejecuta un bucle interno que parte con $p \leftarrow l$ y se detiene cuando $p > r$. En cada paso del bucle devuelve como resultado el triple $p \leftarrow getNext_1(\mathcal{B}_i^v, p)$ que estaba activo en la versión i.

4. Experimentación

En esta sección presentamos los resultados de la experimentación realizada con el *benchmark* BEAR [8]. BEAR contiene un archivo RDF de evaluación que consta de 58 versiones cuyos contenidos proceden de diversos dominios. En total,

este archivo contiene $|\mathcal{A}| = 2{,}073$ millones de triples, de los cuales 376 millones son triples únicos y $3,5$ millones son triples que aparecen en todas las versiones. En promedio, el 31 % de los triples cambian entre dos versiones consecutivas, mientras que el tamaño de las versiones se incrementa desde ≈ 33 millones de triples en $|V_0|$ a ≈ 66 millones en $|V_{57}|$. El tamaño del archivo *en plano* (en formato *NTriples*) es de `325GB`, mientras que comprimido con *gzip*, ocupa `23GB`.

Además del archivo de referencia, BEAR también proporciona un conjunto variado de consultas Mat, $Diff$, y Ver. Para una comparación justa, las consultas consideradas devuelven un número similar de resultados en cada versión, y se clasifican en Q_L (bajo número de resultados), y Q_H (alto número de resultados). Para cada una de ellas, BEAR provee búsquedas por sujeto: `(s??)`, predicado: `(?p?)`, y objeto: `(??o)`. Es decir, el conjunto total de consultas comprende seis escenarios diferentes de evaluación: Q_L^S, Q_L^P, Q_L^O (consultas con baja cardinalidad) y Q_H^S, Q_H^P, Q_H^O (consultas con alta cardinalidad) para búsquedas por sujeto, predicado y objeto, respectivamente. BEAR facilita 50 consultas diferentes para cada escenario, excepto para predicados(6 y 10 consultas en Q_L^P y Q_H^P, respectivamente).

Por último, BEAR implementa un sistema de referencia para manejar archivos RDF basado en Jena TDB[10]. Este sistema considera tres variantes de indexación para reflejar las tres principales estrategias de representación descritas previamente: i) Jena-Ic indexa cada versión en un repositorio independiente; ii) Jena-Cb crea dos índices por cada versión, para los triples añadidos y eliminados; y iii) Jena-Tb usa el nombre del grafo para anotar cuando se añade o borra cada triple, empleando un único repositorio TDB.

Nuestra implementación de `v-RDFCSA` está realizada en lenguaje C y considera dos variantes, correspondientes a las estrategias `vpt` y `tpv`. En ambos casos, evaluamos distintos valores de muestreo para la estructura ψ en RDFCSA ($t_\psi = \{16, 64, 256\}$). Además, consideramos estructuras de bits planas y comprimidas con RRR [16]. En este segundo caso, concatenamos todas las secuencias de bits `vpt`, y aplicamos RRR sobre la secuencia de bits resultante, con el objetivo de mejorar los requisitos de almacenamiento de la solución. Finalmente, destacar también que nuestra variante `tpv-RRR-OPT` explota las particularidades de RRR a la hora de resolver consultas por sujeto.

El estudio experimental ha sido realizado en una máquina Intel Xeon E5-2650v2 @ 2.60GHz (32 núcleos), 256GB RAM,Debian 7.8, y nuestros prototipos se han compilado con `gcc 4.7.2` (opción `-O9`).

Tradeoffs espacio/tiempo. En primer lugar, estudiamos el rendimiento de `v-RDFCSA` en todas sus variantes. Por motivos de espacio, mostramos únicamente los resultados más representativos. En este caso, optamos por aquellos escenarios en los que se obtienen un gran número de resultados (alta cardinalidad), ya que las conclusiones son similares para aquellas consultas con menor número de resultados. Además, descartamos las consultas por predicado ya que

[10] https://jena.apache.org/documentation/tdb/

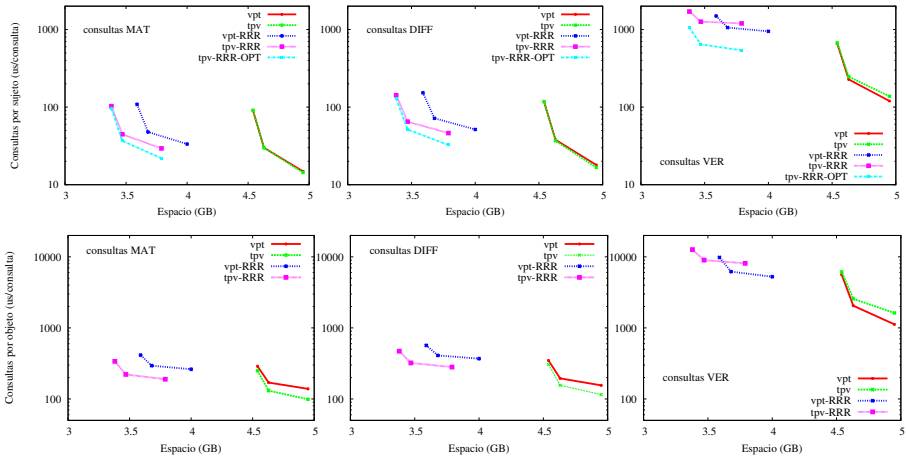

Figura 4. Comparación espacio/tiempo. Búsquedas por sujeto (arriba) y por objeto (abajo): consultas Mat, $Diff$, y Ver.

su rendimiento se enmarca entre el obtenido por las búsquedas por sujetos y por objetos.

La Figura 4 muestra los *tradeoffs* espacio/tiempo obtenidos por `v-RDFCSA`. El eje X representa los requisitos de espacio (en GB), mientras que el eje Y presenta tiempos de consulta (en μs por consulta). Centrándonos en los requisitos de espacio, las variantes en plano de `vpt` y `tpv` necesitan $4,54 - 4,95$GB (el espacio se incrementa de $t_\psi = 256$ a $t_\psi = 16$), pero el uso de RRR lo reduce a $3,38 - 3,79$GB (`tpv`) y 3,59 − 4,00GB (`vpt`). Este hecho demuestra que la información de versionado es muy compresible, al igual que el propio RDF. En este caso, utilizar secuencias de bits comprimidas puede ahorrar más de 1GB. Sin embargo, RRR es más lento que la variante que emplea secuencias de bits en plano. No obstante, aunque la diferencia es importante en las consultas Ver, ambas alternativas compiten en las consultas Mat y $Diff$. La comparación entre `vpt` y `tpv` muestra que ambas se comportan de manera muy similar en las consultas por sujeto, debido a que ambas configuraciones pueden aprovechar la localidad en el acceso a las secuencias de bits. La diferencia se incrementa en las búsquedas por objeto, debido a que los posibles resultados se encuentran dispersos en el array de sufijos y, por tanto, se producen más fallos de caché al acceder a las secuencias de bits. Este hecho revela qué configuración (de secuencia de bits) es mejor para cada tipo de consulta. Por una parte, `tpv` es la opción más rápida para las consultas Mat y $Diff$, ya que para resolverlas sólo revisa una o dos versiones, respectivamente. Por otra parte, `vpt` es la opción preferida para Ver, considerando que para resolver este tipo de consultas es necesario revisar todas las versiones de los triples recuperados de RDFCSA.

Para las consultas por sujeto, `tpv` tarda, en promedio, 14-91μs y 17-118μs, para cada operación, Mat y $Diff$, respectivamente; mientras que `tpv-RRR-OPT` necesita 22-95μs y 32-128μs, en los mismos casos. En las consultas Ver, `vpt` pre-

cisa, de media, 120-660μs, frente a 539-1055μs, en `tpv-RRR-OPT`. El rendimiento obtenido en las consultas por objeto es menos competitivo debido a que su resolución es más compleja en RDFCSA. En este caso, la variante `tpv` consigue una media de 99-250μs y 115-307μs (para las operaciones Mat y $Diff$); mientras que `tpv-RRR` reporta 189-338μs y 281-470μs, respectivamente. En las consultas Ver, `vpt` necesita $1,1$-$5,6ms$ por consulta, frente a $5,2$-$9,8ms$ en `vpt-RRR`.

Comparación con el Sistema de Referencia. A continuación, comparamos las variantes más competitivas de `v-RDFCSA` (con $t_\psi = 64$) respecto al sistema de referencia propuesto en BEAR. Para una comparación más justa, integramos un diccionario *Front-Coding* [15] que permite transformar en *términos RDF* los resultados recuperados directamente desde nuestra propuesta. De este modo, nuestro conjunto de resultados es el mismo que el obtenido por la implementación del sistema de referencia usando Jena. Este diccionario añade 2,3GB extra al espacio requerido por `v-RDFCSA`. Además, consideramos el tiempo de transformación de los resultados a términos RDF, lo que supone una pequeña sobrecarga respecto a los resultados presentados en la sección anterior.

`v-RDFCSA` mejora notablemente el espacio obtenido por las diferentes estrategias implementadas en el sistema de referencia. Nuestras variantes emplean $\approx$ 5,7-7,3 GB, mientras que Jena-Ic, Jena-Cb, y Jena-Tb necesitan 225GB, 196GB, y 353GB, respectivamente. Esta mejora en espacio se complementa con una resolución de consultas notablemente más eficiente. La Figura 5 compara el tiempo de resolución para consultas por sujeto y objeto, con alta cardinalidad. La gráfica Mat (izquierda) muestra el tiempo medio de resolución (eje Y) para cada versión en el archivo RDF (eje X), mientras que $Diff$ (centro) muestra el tiempo medio para consultas que computan las diferencias entre la versión inicial e intervalos crecientes de 5 versiones (como se define en BEAR). Para una mayor claridad, sólo incluimos los resultados de `v-RDFCSA` para `tpv` y `tpv-RRR(-OPT)`, ya que el rendimiento del resto de alternativas está en el mismo orden de magnitud. Por último, la gráfica derecha presenta el tiempo medio de resolución para las consultas Ver, considerando las variantes `vpt` en `v-RDFCSA`.

Dentro del sistema de referencia, Jena-Ic se postula como la mejor alternativa, aunque su rendimiento es similar a Jena-Tb para las consultas Ver. Sin embargo, nuestras variantes mejoran todas las estrategias consideradas en el sistema de referencia para consultas Mat y $Diff$ en búsquedas por sujeto y objeto. La diferencia a nuestro favor es de, aproximadamente, un orden de magnitud. Jena-Ic obtiene tiempos en el orden de 1-4ms/consulta, mientras que `v-RDFCSA` necesita $0,1$-$0,4ms$/consulta. En el caso de las consultas Ver, el resultado es similar, si bien nuestra ventaja se ve reducida ligeramente en las búsquedas por objeto. Nuestra mejor alternativa (`vpt`) obtiene $0,5ms$/consulta y $3,5ms$/consulta para búsquedas por sujeto y objeto, respectivamente, frente a $61,5ms$/consulta and $73,5ms$/consulta requerido por Jena-Tb.

En conclusión, `v-RDFCSA` demuestra cómo la gestión de archivos RDF comprimidos no sólo reduce el espacio requerido (mejorando la escalabilidad de las representaciones), sino que también mejora los tiempos de resolución de las consultas más relevantes en este escenario.

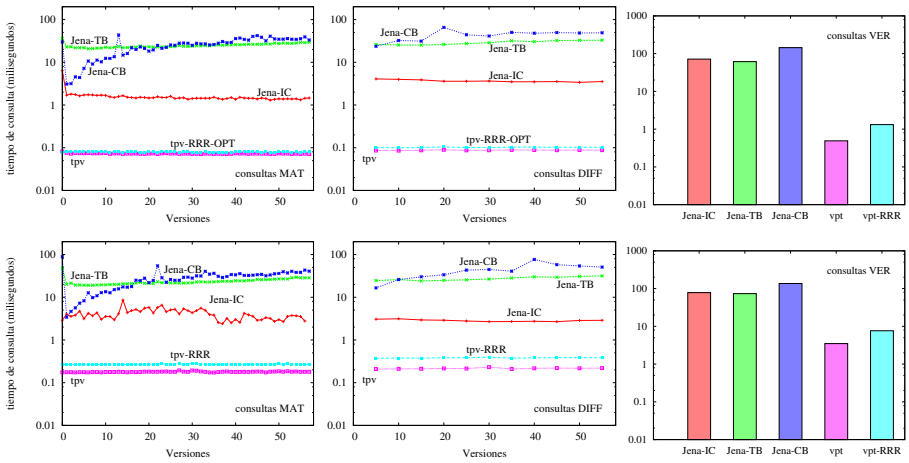

Figura 5. Comparación con el marco de referencia. Búsquedas por sujeto (arriba) y por objeto (abajo): consultas Mat, $Diff$, y Ver.

5. Conclusiones y Trabajo Futuro

En este artículo hemos presentado `v-RDFCSA`, la primera técnica diseñada específicamente para la gestión y consulta de archivos RDF comprimidos. Nuestra propuesta se ha construido como una extensión del autoíndice RDFCSA y considera dos estrategias para la codificación de la información de versionado asociada a cada triple en el archivo. Ambas estrategias están basadas en estructuras de bits con soporte para operaciones `rank` y `select`. Esta decisión no sólo garantiza un espacio de almacenamiento reducido, sino también la resolución eficiente de consultas en memoria principal.

Nuestra propuesta ha sido validada experimentalmente utilizando el *benchmark* BEAR, obteniendo unos resultados muy competitivos tanto en espacio de almacenamiento como en rendimiento de consulta. Las variantes de `v-RDFCSA` necesitan hasta 60 veces menos espacio de almacenamiento que las técnicas consideradas en el sistema de referencia y mejoran su rendimiento de consulta hasta en un orden de magnitud.

Estos resultados asientan nuestras expectativas en esta nueva línea de investigación y abren nuevos retos futuros. Actualmente, estamos trabajando en la implementación de algoritmos adicionales para la resolución de consultas SPARQL avanzadas con las que aumentar la funcionalidad provista por `v-RDFCSA`. Complementariamente, estamos extendiendo HDT para evaluar su rendimiento en este escenario. De acuerdo a nuestros experimentos iniciales, se espera que la solución basada en HDT sea algo menos competitiva en espacio, pero con un rendimiento de consulta altamente competitivo.

Agradecimientos. La investigación presentada en este artículo ha sido financiada por los proyectos TIN2013-46238-C4-3-R, TIN2013-47090-C3-3-P y

TIN2015-69951-R del MINECO (PGE y FEDER); proyecto ITC-20151247 CDTI,MINECO; ICT COST Action IC1302; proyecto GRC2013/053 de la Xunta de Galicia (FEDER) y Austrian Science Fund (FWF): M1720-G11.

Referencias

[1] S. Álvarez-García, N. Brisaboa, J.D. Fernández, M.A. Martínez-Prieto, and G. Navarro. Compressed Vertical Partitioning for Efficient RDF Management. *Knowl. Inf. Syst.*, 44(2):439–474, 2015.

[2] N. Brisaboa, A. Cerdeira, A. Fariña, and G. Navarro. A compact RDF store using suffix arrays. In *Proc. of SPIRE*, pages 103–115, 2015.

[3] N. Brisaboa, S. Ladra, and G. Navarro. Compact Representation of Web Graphs with Extended Functionality. *Infor. Syst.*, 39(1):152–174, 2014.

[4] F. Claude, A. Fariña, M.A. Martínez-Prieto, and G. Navarro. Universal Indexes for Highly Repetitive Document Collections. *Information Systems*, 2016. Disponible en `http://www.dcc.uchile.cl/ gnavarro/ps/is16.3.pdf`.

[5] I. Dong-Hyuk, L. Sang-Won, and K. Hyoung-Joo. A Version Management Framework for RDF Triple Stores. *Int. J. Softw. Eng. Know.*, 22(1):85–106, 2012.

[6] J. D. Fernández, A. Polleres, and J. Umbrich. Towards Efficient Archiving of Dynamic Linked Open Data. In *Proc. of DIACHRON*, pages 34–49, 2015.

[7] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange. *J. Web Semant.*, 19:22–41, 2013.

[8] J.D. Fernández, J. Umbrich, and A. Polleres. BEAR: Benchmarking the Efficiency of RDF Archiving. Technical report, 2015. Disponible en `http://epub.wu.ac.at/4615/`.

[9] D. Gomes, M. Costa, D. Cruz, J. Miranda, and S. Fontes. Creating a Billion-scale Searchable Web Archive. In *Proc. of WWW Companion*, pages 1059–1066, 2013.

[10] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. of WEA*, pages 27–38, 2005.

[11] S. Harris and A. Seaborne. *SPARQL 1.1 Query Language*. W3C Recomm., 2013. `http://www.w3.org/TR/sparql11-query/`.

[12] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan. Observing Linked Data Dynamics. In *Proc. of ISWC*, pages 213–227, 2013.

[13] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. Ontology Versioning and Change Detection on the Web. In *Proc. of EKAW*, pages 197–212, 2002.

[14] F. Manola and E. Miller. *RDF Primer*. W3C Recomm., 2004. `www.w3.org/TR/rdf-primer/`.

[15] M.A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical Compressed String Dictionaries. *Infor. Syst.*, 56:73–108, 2016.

[16] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of SODA*, pages 233–242, 2002.

[17] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *J. Algorithm*, 48(2):294–313, 2003.

[18] M. Vander Sander, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. Van de Walle. R&Wbase: Git for Triples. In *Proc. of LDOW*, 2013. CEUR-WS 996, paper 1.