

# Compresión y Consulta de Diccionarios de Texto en Grandes Colecciones de Datos

Nieves R. Brisaboa<sup>\*1</sup>, Rodrigo Cánovas<sup>†2</sup>, Francisco Claude<sup>‡3</sup>,  
Miguel A. Martínez-Prieto<sup>§†2,4</sup>, and Gonzalo Navarro<sup>†2</sup>

<sup>1</sup> Laboratorio de Bases de Datos, Universidade da Coruña, Spain

<sup>2</sup> Departamento de Ciencias de la Computación, Universidad de Chile, Chile

<sup>3</sup> School of Computer Science, University of Waterloo, Canada

<sup>4</sup> Departamento de Informática, Universidad de Valladolid, Spain

**Resumen** La representación compacta de diccionarios de texto es un problema transversal a numerosas aplicaciones que manejan grandes colecciones de datos. Aún así, su resolución no ha sido tratada tradicionalmente ya que el tamaño de estos diccionarios apenas suponía una pequeña fracción del tamaño total de las colecciones utilizadas. El asentamiento de aplicaciones relacionadas con la Bioinformática, la búsqueda y minería en la Web o la consulta de grafos semánticos realza la necesidad de disponer de soluciones para la compresión de los grandes diccionarios que utilizan. Este trabajo presenta diferentes técnicas para la compresión de diccionarios de texto. Los resultados muestran que el espacio se puede reducir hasta el 20 % del original, soportando la consulta en pocos microsegundos, mientras que tasas de compresión mejores (hasta el 10 %) elevan los tiempos hasta órdenes de cientos de microsegundos.

## 1. Introducción

Un *diccionario de texto* contiene el conjunto de todos los símbolos diferentes utilizados en una colección de datos. A pesar de que estos diccionarios surgen de forma natural en diferentes escenarios, su uso tiende a asociarse con aplicaciones relacionadas con el procesamiento de lenguaje natural donde “localizar el *lexicón* es el primer paso a desarrollar en el análisis de un texto” [21]. Estos diccionarios, al igual que los utilizados en la indexación de texto, están formados por todas las palabras diferentes utilizadas en sus colecciones. La representación compacta de estos diccionarios no ha sido considerada, tradicionalmente, asumiendo que su crecimiento es sublineal respecto al tamaño del texto procesado. La Ley de Heaps [15] aproxima el tamaño de estos diccionarios como  $O(n^\beta)$ , para un texto formado por  $n$  palabras y caracterizado por un parámetro  $0 < \beta < 1$  que depende del

---

\*Financiado por el Ministerio de Ciencia e Innovación (PGE and FEDER): TIN2009-14560-C03-02 y la Xunta de Galicia ref. 09TIC060E.

†Financiado por el Instituto de Dinámica Celular y Biotecnología (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

‡Financiado por el programa David R. Cheriton.

§Financiado por el Ministerio de Ciencia e Innovación: TIN2009-14009-C02-02.

tipo de texto y cuyo valor se distribuye, habitualmente, en el intervalo  $0,4 - 0,6$  [2]. Esta caracterización permite concluir que el diccionario resultante de una colección de texto de varios terabytes apenas ocupe unos pocos megabytes por lo que su representación y consulta puede llevarse a cabo, fácilmente, en la memoria principal de un ordenador personal de uso común.

Sin embargo, la Ley de Heaps no obtiene una buena representación de las propiedades subyacentes a las colecciones de texto extraídas de la Web. Estas colecciones contienen documentos en múltiples idiomas (con las consiguientes palabras utilizadas en cada uno de ellos), así como identificadores y palabras con errores ortográficos que se representan como símbolos regulares del diccionario. La colección ClueWeb09 (<http://boston.lti.cs.cmu.edu/Data/clueweb09>, gentileza de Leonid Boystov) está formada por más de 1 billón de páginas web publicadas en 10 idiomas diferentes. Su diccionario contiene más de 200 millones de palabras y su tamaño aproximado es de 1GB.

La compresión de grafos Web es un área con un intenso nivel de investigación motivado por las necesidades de representar y manejar eficientemente grafos de gran tamaño en aplicaciones de minería Web, detección de spam, búsqueda de comunidades de interés, etc. [17]. Los diccionarios de estas colecciones identifican las URLs de las páginas que forman el grafo. Su representación no ha sido directamente estudiada y, sin embargo, la problemática asociada con la misma es cada vez mayor debido a los grandes avances obtenidos en la compresión de la topología del grafo. A su vez, la utilización de estos diccionarios de URLs permite mejorar la calidad de la minería [27]. Los grafos Web almacenan las páginas en los nodos y los hipervínculos existentes entre ellas en las aristas. Algunos estudios realizados una década atrás [5,24] muestran que el espacio utilizado para representar el diccionario es un 10–25 % del necesario para los hipervínculos. Sin embargo, los avances alcanzados en la compresión de las aristas de un grafo web [3,1] (cuya representación puede llevarse a cabo con 1 o 2 bits/arista) implican que el coste actual de representar las URLs almacenadas en los nodos suponga una fracción considerable del tamaño total de la representación.

La Bioinformática es otra de las áreas donde se plantea la necesidad de operar con diccionarios comprimidos. Un ejemplo lo representa el conocido software BLAST [14] que indexa todas las subcadenas de longitud  $q$  identificadas en una colección de secuencias biológicas. Los valores comúnmente utilizados son  $q = 11, 12$  para secuencias de ADN y  $q = 3, 4$  para proteínas. En ambos casos, los diccionarios originados superan los 200 millones de caracteres (sobre alfabetos de 4 nucleótidos y 20 aminoácidos respectivamente). Un aumento en el valor de  $q$  permitiría mejorar la calidad de los resultados obtenidos en la localización de regiones conservadas. Sin embargo, la utilización de valores mayores de  $q$  aumenta, aún más, el tamaño del diccionario.

El proyecto Linked Data (<http://linkeddata.org>) enfoca la publicación de datos RDF (<http://www.w3.org/TR/rdf-syntax-grammar>) y la conexión de sus diferentes fuentes de datos en la conocida como “Web de Datos”. La popularidad de este movimiento viene motivada por la publicación de grandes colecciones de RDF procedentes de áreas diversas. El diccionario es un componente esencial en

la división lógica de una colección de datos RDF pero su representación no ha sido directamente estudiada [8].

Esta pequeña revisión muestra como la representación compacta de diccionarios de texto es un problema representativo en un conjunto heterogéneo de áreas de en el que se podrían incluir otras aplicaciones como las relacionadas con el enrutado de Internet (y la representación de diccionarios de nombres de dominio y direcciones IP llevada a cabo en los DNS). Sin embargo, no existe mucho trabajo de investigación al respecto. Esto se puede explicar porque el tamaño de estos diccionarios no representaba, en el pasado, un problema significativo y la utilización de técnicas como Front-Coding resultaban suficientes.

Este artículo enfrenta la realidad actual mediante la definición de un conjunto de técnicas diseñadas específicamente para la compresión de diccionarios de texto de gran tamaño. Estas técnicas soportan dos operaciones básicas: (1) *locate*( $p$ ) devuelve la posición (identificador) de la cadena  $p$  en el diccionario (o indica que ésta no existe); (2) *extract*( $i$ ) devuelve la cadena almacenada en la  $i$ -ésima posición. Adicionalmente, especializamos las funciones anteriores para operar con prefijos; esto permite localizar y/o extraer todas las cadenas prefijadas por un patrón dado. El rendimiento de estas técnicas se estudia sobre diccionarios reales procedentes de diferentes escenarios y muestran como las nuevas técnicas dominan diferentes nichos de aplicación. Las opciones más efectivas obtienen ratios de compresión entre el 9% y el 22% (dependiendo del tipo de diccionario), respondiendo de manera eficiente todas las operaciones anteriores.

## 2. Conceptos Básicos y Trabajo Relacionado

*Operaciones rank y select en bitmaps.* Supongamos una secuencia de bits (*bitmap*)  $B[1, n]$  de longitud  $n$  y con  $m$  bits en 1. Se define  $rank_b(B, i)$  como el número de ocurrencias del bit  $b$  en  $B[1, i]$  y  $select_b(B, i)$  como la posición de la  $i$ -ésima ocurrencia de  $b$  en  $B$ . En este trabajo utilizamos dos estructuras de datos compactas (<http://libcds.recoded.cl>) que implementan estas operaciones.

La primera: *RG* [12], usa  $(1 + x)n$  bits para representar  $B$  y soporta *rank* con dos accesos aleatorios a memoria y  $4/x$  accesos contiguos (cacheados). La operación *select* requiere una búsqueda binaria adicional. La segunda técnica: *RRR* [22], utiliza en la práctica  $\log \binom{n}{m} + (\frac{4}{15} + x)n$  bits (todos los logaritmos son en base 2) para la compresión del *bitmap*. Responde *rank* con dos accesos aleatorios más  $3 + 8/x$  accesos adicionales a memoria contigua y también implementa *select* con una búsqueda binaria extra. *RRR* obtiene compresión si  $m < 0,2n$ .

*Códigos Huffman y Hu-Tucker.* Los métodos de compresión estadísticos utilizan códigos más cortos para la representación de los símbolos más frecuentes. La codificación de Huffman [16] es óptima para un conjunto de frecuencias dado. En este trabajo utilizamos códigos de Huffman canónicos.

La codificación Hu-Tucker [18] es óptima entre todos aquellos códigos que mantienen el orden lexicográfico de los símbolos. El interés principal de esta

técnica radica en que dos secuencias codificadas con ella pueden ser comparadas lexicográficamente, a nivel de byte, en forma comprimida. Ambas técnicas utilizan bits 0 adicionales para el alineamiento de códigos a nivel de byte.

*Hashing.* Las técnicas de *hashing* han sido utilizadas, de forma tradicional, en la representación de diccionarios. Estas técnicas están basadas en el uso de *funciones de hash* que transforman los elementos en índices de una *tabla hash* en la que se insertan o buscan los valores correspondientes. Se dice que se produce una *colisión* cuando la función hash de dos elementos diferentes devuelve el mismo índice en la tabla. En este trabajo consideramos *hashing cerrado* como mecanismo para la resolución de colisiones: si la celda determinada para un elemento se encuentra ocupada, se prueba sucesivamente hasta encontrar una celda vacía (inserciones y búsquedas infructuosas) o hasta encontrar el elemento solicitado (búsquedas satisfactorias). Consideramos dos políticas de prueba tras una colisión en la celda  $x$ . *Double hashing* utiliza una segunda función de hash  $y$  (que depende de la clave) y prueba las celdas  $x + y$ ,  $x + 2y$ , etc. modulo el tamaño de la tabla. *Linear probing* es una política sencilla que prueba, sucesivamente, las celdas  $x + 1$ ,  $x + 2$ , etc. modulo el tamaño de la tabla.

El *factor de carga* es la fracción de celdas ocupadas en la tabla y su valor influye tanto en el espacio que ésta ocupa como en el rendimiento de las consultas. La utilización de funciones de hash adecuadas determina que las inserciones y las búsquedas infructuosas requieren una media de  $1/(1 - \alpha)$  pruebas con *double hashing* mientras que las búsquedas satisfactorias necesitan  $\ln(1/(1 - \alpha))/\alpha$  pruebas. El rendimiento con *linear probing* requiere, en promedio, un mayor número de pruebas:  $(1 + 1/(1 - \alpha)^2)/2$  para inserciones y búsquedas infructuosas y  $(1 + 1/(1 - \alpha))/2$  para búsquedas satisfactorias. A pesar de su menor competitividad, esta segunda política presenta propiedades interesantes para su combinación con algunas de las representaciones comprimidas consideradas.

*Front-Coding.* Es la técnica utilizada, tradicionalmente, para la compresión de diccionarios lexicográficamente ordenados [26]. Esta técnica aprovecha que las cadenas consecutivas tienden a compartir un prefijo común. Cada cadena se codifica de forma diferencial respecto a la anterior. Esta codificación distingue dos partes: un valor entero que codifica la longitud del prefijo común y la subcadena que representa el sufijo restante en la entrada.

Front-Coding divide el diccionario en bloques con el fin de soportar operaciones eficientes de búsqueda: la primera cadena, en cada bloque, se almacena explícitamente y el resto se codifica diferencialmente de acuerdo al mecanismo anterior. El algoritmo ejecuta, inicialmente, una búsqueda binaria que compara el valor consultado con la primera entrada de cada bloque. Esto permite localizar el bloque candidato que, posteriormente, se recorre de forma secuencial reconstruyendo “al vuelo” cada cadena de tal forma que se pueda comparar con la solicitada. El proceso se detiene al localizar la cadena o al finalizar el recorrido del bloque si la cadena solicitada no existe en el diccionario. Front-Coding se ha utilizado satisfactoriamente en escenarios como *WebGraph* (<http://webgraph.dsi.unimi.it>) donde se usa con diccionarios de URLs.

*Autoíndices comprimidos.* Este tipo de estructuras modelan un texto  $T[1, N]$  en espacio proporcional al utilizado por su representación comprimida y soportan acceso aleatorio y búsqueda indexada en  $T$ . De forma más precisa, un autoíndice implementa, al menos, las operaciones:  $extract(i, j)$ , que devuelve  $T[i, j]$ , y  $locate(p)$ , que recupera todas las ocurrencias del patrón  $p$  en  $T$ .

En este trabajo nos centramos en los *FM-index* [10,11]. Estos autoíndices están basados en la *Transformada de Burrows-Wheeler (BWT)* [6] y obtienen las mejores ratios de compresión, entre los autoíndices, al tiempo que responden muy rápidamente acerca de la existencia de  $p$  en  $T$ . Las implementaciones de algunos autoíndices están disponibles en *PizzaChili* (<http://pizzachili.dcc.uchile.cl>).

La BWT de  $T[1, N]$  ( $T^{bwt}[1, N]$ ) es una permutación de los símbolos  $T$ .  $T^{bwt}[j]$  es el carácter que precede al  $j$ -ésimo *sufijo* de  $T$  si todos sus sufijos están lexicográficamente ordenados. Estos índices implementan dos operaciones sobre  $T^{bwt}$ : 1) *LF-step* (LF) se desplaza desde  $T^{bwt}[j]$  (correspondiente al sufijo  $T[i, N]$ ) a  $T^{bwt}[j']$  (sufijo  $T[i - 1, N]$ ) o  $T[N, N]$  (si  $i = 1$ ). Por lo tanto,  $j' = LF(j)$ . 2) *Backward step* (BWS) se mueve desde el intervalo que contiene a todos los sufijos de  $T$  precedidos por la cadena  $x$  ( $T^{bwt}[sp, ep]$ ) hasta  $T^{bwt}[sp', ep']$ , dónde se localizan todos los sufijos precedidos por  $cx$  para un carácter  $c$ .

*Compresión basada en gramáticas.* La compresión basada en gramáticas se basa en la derivación de una gramática  $G$  de pequeño tamaño que permita generar el texto original  $T$  [7]. Su funcionamiento considera el uso de heurísticas a la hora de determinar qué reglas añadir a  $G$  ya que obtener la gramática más pequeña para  $T$  es un problema NP-difícil [7]. Estos compresores obtienen una mejor efectividad al ser aplicados a textos con muchas repeticiones.

En este trabajo utilizamos Re-Pair [19], considerando que su ejecución se desarrolla en tiempo lineal y que obtiene resultados muy competitivos en la práctica. El algoritmo que utiliza para la obtención de  $G$  busca, recursivamente, el par  $xy$  más frecuente en  $T$  y reemplaza todas sus ocurrencias por un nuevo símbolo  $R$  que inserta  $R \rightarrow xy$  como una nueva regla en  $G$ . El proceso se repite hasta que todos los pares sean únicos. El resultado de la ejecución de Re-Pair consiste en el conjunto de  $r$  reglas que forman  $G$  y la representación comprimida del texto original:  $\mathcal{C}$ . Nuestra propuesta (ver Sección 3.4) almacena tanto los componentes de las reglas como los símbolos de  $\mathcal{C}$  utilizando  $\log(\sigma + r)$  bits.

*Códigos de longitud variable y acceso directo.* La técnica *Directly Addressable variable-length Codes (DACs)* [4] reordena los símbolos de una secuencia codificada con longitudes variables de tal forma que todos sus códigos se pueden acceder directamente. Los primeros símbolos, de cada código, se concatenan en un *array*  $A_1$  y se construye un *bitmap*  $B_1$  que utiliza un bit para indicar si cada elemento (en  $A_1$ ) tiene o no más componentes. Los segundos símbolos de todos aquellos códigos con longitud mayor que 1 se concatenan en un segundo *array*  $A_2$  que también precisa de un *bitmap*  $B_2$  que marca, al igual que el anterior, con bits 1 todos aquellos elementos que están formados por más símbolos.

La extracción del  $i$ -ésimo código requiere localizar su primer símbolo en  $A_1[i]$ . Si  $B_1[i] = 0$ , entonces el código solicitado está formado exclusivamente por este

símbolo. Si no, el proceso continúa en  $A_2[\text{rank}_1(B_1, i)]$  y así, sucesivamente, hasta extraer completamente el código.

En este trabajo utilizamos Vbyte [25] como código de longitud variable orientado a byte. Esta técnica reserva el primer bit de cada byte para propósitos de marcado (indicando si el byte actual es o no el último del código) y utiliza los 7 bits restantes para la propia codificación.

*Tries y la XBW.* Un *trie* es un árbol etiquetado en cada arista con un determinado carácter. De esta manera, el recorrido de las ramas (de la raíz a las hojas) permite recuperar las cadenas de texto representadas en el *trie*. Las cadenas con prefijos comunes comparten subramas del árbol.

La estructura de *trie* es una solución natural para la representación compacta de un diccionario. La búsqueda de una cadena navega, desde la raíz, por los caracteres con los que se etiquetan las aristas, de tal forma que la localización del identificador de una cadena (si existe) se resuelve sin más que recuperar el valor asociado a la hoja que finaliza su rama en el *trie*.

Los *tries* tienden a ocupar mucho espacio en términos prácticos. Ferragina et al. [9] proponen una representación comprimida para *tries* que implementa la navegación y la búsqueda utilizando operaciones de *rank* y *select* sobre secuencias. Esta representación, denominada *XBW*, corresponde con una extensión de la BWT a árboles.

### 3. Representaciones Comprimidas de Diccionarios

Esta sección plantea diferentes propuestas para la representación comprimida de diccionarios de texto. Todas ellas se ejecutan sobre el texto  $T_{dicc}$  obtenido al concatenar, en orden lexicográfico, las cadenas que forman el diccionario. Todas las cadenas se finalizan con un carácter \$ lexicográficamente anterior al resto de los utilizados (en la práctica, se usa el código 0, de fin de cadena, en ASCII).

Todas las técnicas implementan dos operaciones básicas: 1) *locate*( $p$ ) obtiene el identificador de la cadena  $p$  si existe en el diccionario o  $-1$  en otro caso; 2) *extract*( $i$ ) devuelve la  $i$ -ésima cadena o *NULL* si no existe.

Adicionalmente, todas las técnicas (excepto *hashing*) soportan búsquedas eficientes basadas en prefijos. Estas operaciones tienen interés práctico en aplicaciones de grafos web (que necesitan detectar páginas en una ruta de un determinado dominio), de gestión y consulta de URIs (que ejecutan accesos jerárquicos: esquema, autoridad y ruta) o en las aplicaciones de enrutado en la Internet. Actualmente implementamos las operaciones que permiten 1) localizar los identificadores de todas las cadenas que comienzan por un patrón  $p$ : *locatePrefixes*( $p$ ); 2) recuperar todas las cadenas prefijadas por  $p$ : *extractPrefixes*( $p$ ).

#### 3.1. Hashing + Compresión

En primer lugar planteamos cómo las técnicas de *hashing* se pueden utilizar en la representación de los diccionarios de texto. Inicialmente comprimimos  $T_{dicc}$

con Huffman y alineamos, a nivel de byte, la codificación resultante para cada cadena. Los *offsets* (en bytes) de las cadenas codificadas se almacenan en la tabla hash. Esto supone que la función de *hashing* opera sobre las cadenas comprimidas reduciendo, con ello, tanto el tiempo invertido en el cálculo de la propia función como el utilizado en la comparación de las claves. Las operaciones de búsqueda primero codifican con Huffman la cadena solicitada y luego añaden los bits 0 necesarios para alinear, a nivel de byte, el código resultante.

Utilizamos como función de hash principal una modificación de la propuesta de Bernstein (<http://www.burtleburtle.net/bob/hash/doobs.html>):  $h$  se inicializa con un número primo elevado y  $2^{15} + 1$  se sustituye por 33, utilizando el módulo del tamaño de la tabla en cada iteración. La segunda función, utilizada para *double hashing*, se corresponde con la propuesta “*rotating hash*” de Knuth [18, Sec. 6.4] (<http://burtleburtle.net/bob/hash/examhash.html>) en la que  $h$  también se inicializa con un valor elevado.

Las cadenas se almacenan en la tabla hash de acuerdo a su orden en  $T_{dicc}$ . Esto mejora la localidad de referencia en la consulta para *linear probing*. Consideramos tres variantes de representación para la tabla hash y combinamos cada una de ellas con *linear probing* ( $lp$ ) y *double hashing* ( $dh$ ).

La variante **Hash** se basa en el planteamiento tradicional que usa un *array*  $H[1, m]$  en el que cada una de las celdas apunta a la cadena correspondiente. En este caso, cada celda almacena el byte en el que comienza la cadena codificada con Huffman. *locate*( $p$ ) devuelve el byte asociado en  $H$  con la cadena solicitada o  $-1$  si  $p$  no existe en el diccionario. Por su parte, la respuesta de *extract*( $i$ ) se reduce a descomprimir la cadena apuntada desde  $H[i]$ . Dado un factor de carga  $\alpha = n/m$  ( $n$  es el número de cadenas en el diccionario), la estructura requiere  $m$  valores enteros a mayores de las cadenas comprimidas con Huffman.

La segunda variante (**HashB**), almacena  $H[1, m]$  en forma compacta mediante un *array*  $M[1, n]$  que no representa las celdas vacías. Esta decisión requiere de la utilización de un *bitmap* (RG) adicional:  $B[1, m]$ , que marca con un bit 1 todas las celdas no vacías en  $H$ . Por lo tanto,  $B[i] = 0$  implica que  $H[i]$  está vacía y si no lo está, su valor es  $H[i] = M[\text{rank}_1(B, i)]$ . En este caso, *locate*( $p$ ) devuelve posiciones en  $M$  cuyos identificadores están comprendidos en el rango  $[1, n]$ . *extract*( $i$ ) se resuelve descomprimiendo la cadena apuntada en  $M[i]$ . Esta representación utiliza  $n$  valores enteros más  $(1 + \alpha)m$  bits adicionales, donde  $\alpha$  es un parámetro de construcción del *bitmap*. Los  $n$  enteros se representan con  $n \log N$  bits ( $N$  es el número de bytes ocupados al comprimir  $T_{dicc}$  con Huffman). Esta propuesta reduce el espacio utilizado a cambio de aumentar ligeramente el tiempo con la ejecución de un *rank* adicional en  $B$ . Esta operación se hace una única vez en *linear probing* dado que las celdas sucesivas en  $H$  son contiguas en  $M$ , por lo que  $B$  sólo se accede para determinar la siguiente celda vacía.

La variante **HashBB** también utiliza  $M$  y  $B$ , pero reemplaza  $M$  por un segundo *bitmap*. Esta decisión se basa en el hecho de que los valores son siempre crecientes en  $M$  (también en  $H$ ), de tal forma que se pueden representar con un *bitmap*  $Y[1, N]$ , donde cada 1 marca el inicio de un código:  $M[i] = \text{select}_1(Y, i)$ .  $Y$  se almacena de forma comprimida (*RRR*). Esta decisión reduce el tamaño

ocupado por  $M$  a  $\log \binom{N}{n} + (\frac{4}{15} + x)N$  bits y aumenta el tiempo de acceso de acuerdo al coste de una operación *select* adicional por cada acceso a  $M$ . En este caso, *linear probing* no ahorra operaciones sucesivas dado que no conoce, a priori, dónde termina cada código y dónde comienza el siguiente.

### 3.2. Front-Coding + Compresión

**Plain Front-Coding** (PFC) implementa la técnica original utilizando VByte para la codificación de la longitud del prefijo común mientras que el sufijo restante se añade utilizando el símbolo \$ como terminador. Esta decisión garantiza la realización, a nivel de byte, de todas las operaciones requeridas. El tamaño de cada bloque se establece como el número de cadenas que almacena, de manera que  $extract(i)$  primero determina el bloque adecuado (con una simple división) y luego lo recorre secuencialmente hasta localizar la cadena solicitada.

**Hu-Tucker Front-Coding** (HTFC) comprime con Hu-Tucker los VBytes de los prefijos y las subcadenas de los sufijos. Cada bloque comienza con un código VByte que indica la longitud, en bytes, de la primera cadena codificada con Hu-Tucker y alineada a nivel de byte. Este preludeo garantiza que la búsqueda binaria, sobre los bloques, se lleve a cabo sin la descompresión de la primera cadena de cada uno de ellos. El resto del bloque sigue una codificación de Hu-Tucker orientada a bit cuya descompresión se lleva a cabo de forma secuencial tanto en la localización como en la extracción de una cadena. Utilizamos una implementación del árbol de Hu-Tucker basada en punteros.

PFC y HTFC implementan las operaciones de prefijos mediante una búsqueda binaria, acorde a las propiedades de cada una de ellas, que determina los bloques en los que se hallan las cadenas prefijadas por el patrón  $p$  solicitado. Dado el orden lexicográfico de  $T_{dicc}$ , la localización se resuelve sin más que localizar la primera y la última cadena que contienen el prefijo. La operación de extracción se implementa, de forma eficiente, mediante un único recorrido secuencial que extrae todas las cadenas comprendidas entre los límites anteriores.

### 3.3. Representación basada en FM-index

La propuesta actual considera en el uso de las variantes de **FM-index** disponibles en *PizzaChili*. Ambas implementan la BWT con *wavelet trees* [13] basados en *RG* (SSA\_v3.1) y *RRR* (SSA\_RRR). La primera es el “*succinct suffix array*” [11], que obtiene una compresión de orden cero del texto, y la segunda representa la idea de “*implicit compression boosting*” [20], que consigue una compresión de orden superior. Ambas soportan *LF* y *BWS*, así como  $T^{bwt}[j]$  (dado un  $j$ ), en tiempo  $O(\log \sigma)$ , donde  $\sigma$  es el tamaño del alfabeto usado en  $T$ .

Esta propuesta requiere un carácter \$ al inicio de  $T_{dicc}$  con el fin de referir, indistintamente, la  $i$ -ésima cadena en orden lexicográfico y posicional. La ordenación lexicográfica de todos los sufijos en  $T_{dicc}$  hace que el primero sea el símbolo \$ final y los  $n$  siguientes sean cada uno de los \$ que preceden a las cadenas en el diccionario. Por lo tanto  $T^{bwt}[1]$  es el carácter final de la  $n$ -ésima cadena y  $T^{bwt}[i + 2]$  es el carácter final de la  $i$ -ésima cadena, para  $1 \leq i < n$ .

La operación  $extract(i)$  se inicia en la posición correspondiente de  $T^{bwt}$  y ejecuta los LF necesarios para alcanzar el carácter \$. Los caracteres  $T^{bwt}[j]$  recorridos deletrean, en orden inverso, la cadena solicitada. Por su parte,  $locate(p)$  necesita determinar si la cadena  $\$p\$$  ocurre en  $T$ . Definimos el intervalo  $(sp, ep) = (1, n + 1)$  y realizamos  $|p| + 1$  pasos hacia atrás hasta localizar el intervalo lexicográfico  $(sp', ep')$  que contiene todos los sufijos que comienzan con  $\$p\$$ . Si  $p$  existe en el diccionario, y corresponde con la  $i$ -ésima cadena, entonces  $sp' = ep' = i + 1$  y el valor de respuesta es  $i$ . Si en el proceso de búsqueda se obtiene que  $sp' > ep'$ , se devuelve  $-1$  ya que la cadena no existe.

$locatePrefix(p)$  se resuelve de forma similar a  $locate(p)$  con la diferencia de que el intervalo  $(sp, ep)$  determinado corresponde con las ocurrencias de todas las cadenas prefijadas por  $\$p$  en  $T$ . Por su parte,  $extractPrefix(p)$  ejecuta  $locatePrefix(p)$  para la identificación del intervalo y extrae las cadenas que forman el resultado mediante utilizando operaciones  $extract(i) \forall i \in (sp, ep)$ .

### 3.4. Representación basada en Re-Pair

La propuesta actual ejecuta una modificación de Re-Pair que impide que el carácter \$ intervenga en la formación de reglas. Esto asegura la codificación de cada regla con un número entero de símbolos en  $\mathcal{C}$ .

La localización de una cadena ejecuta una búsqueda binaria sobre el diccionario de reglas. Esta operación requiere descomprimir la cadena hasta el punto en el que la comparación lexicográfica se pueda resolver. La extracción, simplemente, descomprime la cadena solicitada. Ambas operaciones necesitan acceder, de forma directa, al primer símbolo en  $\mathcal{C}$  que representa la cadena solicitada.

Todas las cadenas se pueden observar como secuencias de longitud variable de símbolos en  $\mathcal{C}$ . Esto permite utilizar DAC para su representación, obteniendo acceso directo eficiente a la  $i$ -ésima cadena con un coste extra de 1,25 bits por símbolo en  $\mathcal{C}$  dado que utilizamos *bitmaps RG* para la construcción de los DACs.

La localización basada en prefijos requiere, igualmente, una búsqueda binaria sobre el diccionario de reglas que determina la primera y la última cadena prefijadas por el patrón dado. La extracción de estas cadenas parte de la propia localización y se completa mediante la extracción individual de cada una de ellas utilizando la operación primitiva  $extract(i)$ .

### 3.5. Representación basada en XBW

La propuesta actual representa un *trie* de  $N$  nodos utilizando una secuencia de  $S_\alpha[1, N+n]$  etiquetas (cada hoja se identifica con la etiqueta \$ que la precede) y un *bitmap*  $S_{last}[1, N+n]$  con  $n$  bits a 1.  $S_\alpha$  se representa con *wavelet trees* y los *bitmaps* (y  $S_{last}$ ) con *RG* o *RRR*. La función  $locate(p)$  localiza la hoja  $x$  correspondiente mediante la operación `GetChildren` [9] y después obtiene su identificador (en el rango  $[1, n]$ ) mediante  $rank_\$(S, x)$ .  $extract(i)$  parte de la hoja y recupera los caracteres que etiquetan cada arista en la rama recorrida utilizando la función `GetParent` [9].

La extracción de prefijos descende por el *trie* utilizando la ruta que representa el patrón solicitado y devuelve todas las cadenas almacenadas en el subárbol que cuelga del nodo alcanzado. La localización de las cadenas prefijadas por un patrón se resuelve, de forma sencilla, utilizando una *backward-search* sobre el patrón solicitado precedido por el valor de la raíz del *trie*.

## 4. Experimentación

La experimentación se lleva a cabo sobre una máquina de procesador Intel Core2 Duo a 3.16 GHz, con 8 GB de memoria principal y 6 MB de caché y sistema operativo Linux (kernel 2.6:24-28). Eliminamos, aleatoriamente, 1000 cadenas del diccionario original y construimos la representación sobre este resultado. Las pruebas de *locate* se basan en la búsqueda de 10000 cadenas, aleatoriamente elegidas en el diccionario. Adicionalmente buscamos las 1000 cadenas eliminadas con el fin de obtener el tiempo de respuesta para búsquedas infructuosas. Para *extract* consultamos 10000 valores aleatorios en el intervalo  $[1, n]$ . Utilizamos cuatro diccionarios representativos de las aplicaciones revisadas:

**Palabras:** contiene todas las palabras, con 3 o más ocurrencias, en la colección ClueWeb09. Está formado por 25.609.784 palabras y ocupa 256,36 MB.

**ADN** (<http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp>): está formada por todas las subsecuencias de 12 nucleótidos encontradas en la colección S. Paradoxus (*para*). Contiene 9.202.863 subsecuencias y ocupa 114.09 MB.

**URLs** (<http://law.dsi.unimi.it/webdata/uk-2002/>): es la colección uk-2002 publicada en *WebGraph*. Recoge 18.520.486 URLs y ocupa 1,34 GB.

**URIs** (<http://downloads.dbpedia.org/3.5.1/en>): contiene todas las URIs en la colección *DBpedia-en*. Está formado por 30.176.012 URIs y ocupa 1,52 GB.

Los resultados mostrados en la Figura 1 se obtienen al promediar 10 réplicas independientes de cada experimento. Algunos métodos se caracterizan por la línea obtenida al unir los puntos resultantes de su ejecución con distintos parámetros. En la columna izquierda se presentan los resultados de la búsqueda de cadenas existentes en el diccionario (los obtenidos para las no existentes son similares en todos los casos). En la columna derecha se muestran los tiempos necesarios para la extracción de cadenas. En ambos casos, se utiliza una escala de microsegundos para el tiempo mientras que el espacio representa la fracción del tamaño de  $T_{dicc}$  utilizada para el almacenamiento del diccionario en cada técnica. Para las variantes basadas en *hashing*, representamos únicamente los resultados de *double hashing* ya que siempre mejoran a *linear probing*.

*HTFC* muestra un comportamiento excelente en todos los casos, consiguiendo tiempos de respuesta muy competitivos en un espacio de almacenamiento muy pequeño (sólo mejorado por *XBW* y por *Re-Pair* en URLs). Esto supone que *PFC* obtiene un peor desempeño que su variante comprimida. La técnica más efectiva en espacio es *XBW+RRR*, aunque sus tiempos de son significativamente mayores que para el resto de propuestas. La siguiente técnica, en URLs, es *Re-Pair*, cuyos tiempos de respuesta son mucho mejores que para la *XBW* pero es más

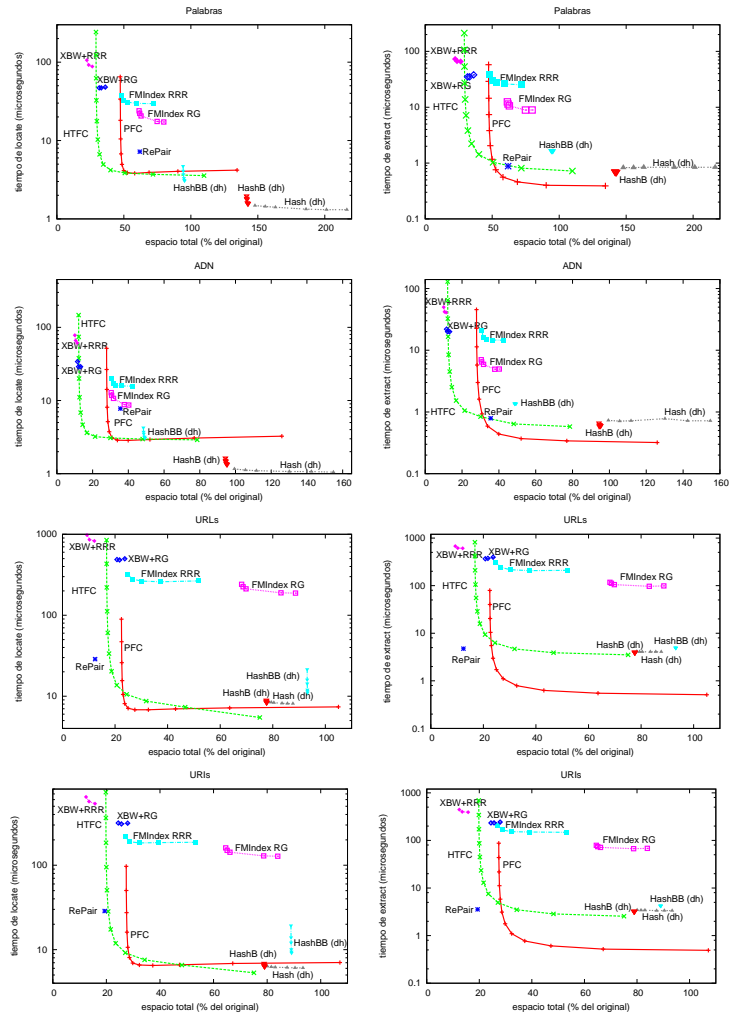
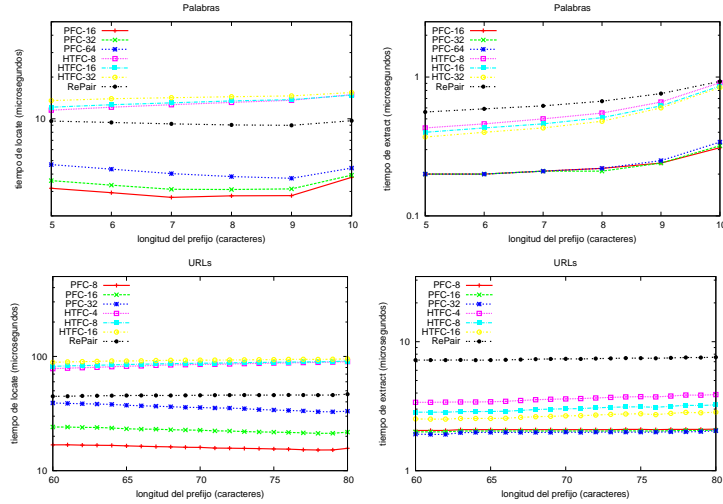


Figura 1. Tiempos de *locate* (izquierda) y *extract* (derecha).

lenta que *HTFC*. En los diccionarios formados por cadenas de menor longitud (palabras y ADN), *Re-Pair* no obtiene unos buenos resultados de compresión y *HTFC* consigue el segundo mejor espacio, obteniendo mejores tiempos que las variantes de *XBW*. Por su parte, *HashBB* requiere un espacio menor que *HashB*, para cadenas cortas, mientras que para el resto de casos el *bitmap* necesario resulta demasiado largo. Aún así, estas técnicas nunca son la mejor alternativa. *HashB* y *Hash* obtienen los mejores tiempos para la localización de cadenas cortas a costa de obtener unas tasas de compresión pobres, aunque *HashB* nunca es significativamente mejor que *Hash*. Para extracción, *PFC* y *HTFC* plantean la mejor alternativa. *Re-Pair* obtiene un espacio menor en URLs y la *XBW* es la opción más compacta a costa de unos tiempos de extracción mucho mayores.



**Figura 2.** Tiempos de *locatePrefix* (izquierda) y *extractPrefix* (derecha).

La Figura 2 resume los resultados obtenidos por las operaciones basadas en prefijos. Dicha figura presenta las gráficas resultantes para las colecciones de palabras y URLs, cuya elección se basa en la longitud media de las cadenas que las forman: más cortas en palabras (9,50 caracteres/palabra) y más largas en URLs (76,68 caracteres/URL). El eje X representa la longitud, en caracteres, del prefijo utilizado en la consulta. El eje Y muestra el tiempo promedio necesario para localizar el intervalo que contiene las palabras prefjadas por el patrón (*locate*) y para extraer cada una de las cadenas en ese intervalo (*extract*). Los resultados se obtienen al promediar 10 réplicas de cada experimento en el que se consideran 1000 patrones de igual longitud. Los resultados obtenidos por las técnicas basadas en el FM-Index y la XBW se excluyen de la representación ya que no mejoran, en ningún caso, a los presentados para PFC, HTFC y Re-Pair. Para los dos primeros se consideran diferentes tamaños de bloque elegidos de forma que el espacio ocupado por sus representaciones sea próximo.

PFC es la opción más eficiente en todas las operaciones por prefijo. Nótese que el uso de tamaños menores de bloque mejora el tiempo a costa de requerir mayor espacio (Figura 1). Por su parte, HTFC obtiene unos tiempos ligeramente peores que PFC. Re-Pair es la única técnica no basada en Front-Coding que resulta competitiva en este experimento y puede observarse que incluso supera a HTFC en *locate*. Esto hace que Re-Pair sea una opción a considerar en este contexto, sobre todo para el caso de URLs en el que obtiene mejoras tasas de compresión que las otras dos técnicas.

## 5. Discusión

En este trabajo presentamos un conjunto de propuestas para la compresión de diccionarios de texto. Todas ellas resultan interesantes en determinados ni-

chos de aplicación, obteniendo un compromiso espacio-tiempo competitivo en cada uno de los escenarios estudiados. Todas las técnicas soportan operaciones de localización de IDs y extracción de cadenas y, a su vez, algunas de ellas implementan de forma satisfactoria operaciones específicas basadas en prefijos.

El impacto de los resultados presentados en este artículo se puede apreciar con un análisis sencillo de la colección de URLs. Atendiendo a su descripción, cada página contiene un promedio de 16,09 hipervínculos y la representación de cada una de ellos, con WebGraph, requiere 1,805 bits en la matriz de adyacencia y 1,55 bits adicionales en la matriz transpuesta. De esta manera, el modelado de la estructura asociada a cada página utiliza un promedio de 5,63 bytes/nodo mientras que la representación no comprimida de cada una de las URLs supone 77,68 bytes/nodo. Este hecho degrada notablemente la efectividad alcanzada en la compresión de grafos web ya que la representación de las URLs requiere 13,80 veces más espacio que el utilizado para la estructura. La utilización de nuestra técnica basada en *Re-Pair* muestra la gran mejora obtenida gracias a la compresión del diccionario: la representación de cada URL supone un coste promedio de 9,64 bytes/nodo, lo que apenas significa 1,71 veces el espacio utilizado en la estructura de hipervínculos. Esta reducción supone una mejora importante en la escalabilidad de las aplicaciones que usan estas colecciones dada la menor cantidad de recursos que precisan para su procesamiento y consulta, la cual se completa en un tiempo eficiente de unas pocas decenas de microsegundos.

De cara al futuro, consideramos una mejora progresiva de nuestras técnicas, planteando nuevas operaciones de búsqueda interesantes. Por ejemplo, las variantes basadas en *FM-index* y *XBW* pueden localizar todas las cadenas que contienen una determinada subcadena o todas aquellas que poseen un determinado prefijo o sufijo y además soportan búsquedas aproximadas [23]. Otro aspecto a considerar tiene que ver con el orden utilizado en la construcción de  $T_{dicc}$  y, por consiguiente, el orden usado en la asignación de IDs. Las variantes basadas en *hashing* se pueden adaptar fácilmente a cualquier orden (excepto la variante *HashBB*), pero todas las demás precisan almacenar, explícitamente, la permutación que relaciona el orden lexicográfico con el orden requerido por la aplicación. El coste asociado a esta permutación es menor para el *FM-index* y la *XBW* que requieren, exclusivamente, un muestreo de la misma.

## Referencias

1. A. Apostolico and G. Drovandi. Graph compression by BFS. *Algorithms*, 2:1031–1044, 2009.
2. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.
3. P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *Proc. WWW*, pages 595–2562, 2004.
4. N. Brisaboa, S. Ladra, and G. Navarro. Directly Addressable Variable-Length Codes. In *Proc. SPIRE*, LNCS 5721, pages 122–130, 2009.
5. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web. *Comput. Netw.*, 33:309–320, 2000.

6. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.
7. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The Smallest Grammar Problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.
8. J.D. Fernández, M.A. Martínez-Prieto, and C. Gutierrez. Compact Representation of Large RDF Data Sets for Publishing and Exchange. In *Proc. ISWC*, LNCS 6496, pages 193–208 (part I), 2010.
9. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. FOCS*, pages 184–196, 2005.
10. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.
11. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed Representations of Sequences and Full-Text Indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.
12. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical Implementation of Rank and Select Queries. In *Posters WEA*, pages 27–38, 2005.
13. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.
14. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge Univ. Press, 2007.
15. H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, 1978.
16. D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proc. of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.
17. J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The Web as a Graph: Measurements, Models, and Methods. In *Proc. COCOON*, LNCS 1627, pages 1–17, 1999.
18. D.E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison Wesley, 2007.
19. N.J. Larsson and J. A. Moffat. Offline Dictionary-Based Compression. *Proc. of the IEEE*, 88:1722–1732, 2000.
20. V. Mäkinen and G. Navarro. Implicit Compression Boosting with Applications to Self-Indexing. In *Proc. SPIRE*, LNCS 4726, pages 214–226, 2007.
21. C.D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
22. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding  $k$ -ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.
23. L. Russo, G. Navarro, A. Oliveira, and P. Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.
24. T. Suel and J. Yuan. Compressing the Graph Structure of the Web. In *Proc. DCC*, pages 213–222, 2001.
25. H. Williams and J. Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42:193–201, 1999.
26. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.
27. M. Yin, D. Goh, E.-P. Lim, and A. Sun. Discovery of Concept Entities from Web Sites using Web Unit Mining. *Intl. J. of Web Inf. Sys.*, 1(3):123–135, 2005.