Estrategias de Optimización de Consultas XPath Flexibles sobre XML Wavelet Trees*

Nieves R. Brisaboa[1], Ana Cerdeira-Pena[1], Gonzalo Navarro[2], Gabriella Pasi[3]

[1] Database Lab., Univ. da Coruña, Spain. {brisaboa,acerdeira}@udc.es
[2] Dept. of Computer Science, Univ. of Chile, Chile. gnavarro@dcc.uchile.cl
[3] DISCO, Univ. degli Studi di Milano Bicocca, Italy. pasi@disco.unimib.it

Resumen El almacenamiento autoindexado y comprimido de documentos es una área de investigación muy activa y prometedora debido a que las estructuras de datos que usan permiten no sólo ahorrar espacio de almacenamiento en bases de datos documentales y bibliotecas digitales, sino que además proporcionan ahorros significativos de tiempo de procesamiento. En [2] se presentó una nueva estructura de datos denominada XML Wavelet Tree (XWT), un autoíndice comprimido para documentos XML creado mediante la adaptación de un autoíndice para textos planos [3] basado en el uso de wavelet trees. Además, se mostraba cómo el XWT podía ser usado para responder eficientemente consultas típicas XPath sobre estructura y contenido de los documentos XML. En este trabajo mostramos cómo el XWT puede usarse también para resolver consultas *flexibles* [5,8,9]. Estas consultas constituyen una extensión de XPath que trata de introducir cierta flexibilidad en las búsquedas, adecuándolas más al ámbito de la recuperacion de información; ya que permiten no sólo combinar restricciones sobre estructura y contenido, sino también puntuar y hacer *ránking* de las respuestas, es decir, de los distintos documentos y/o fragmentos de documentos recuperados. Por último, presentamos diversas estrategias heurísticas de optimización del plan de resolución de las consultas y las evaluamos experimentalmente.

1. Introducción

En los últimos años, las bases de datos documentales han cobrado una gran importancia debido al gran crecimiento experimentado por biblitecas digitales y la propia expansión de la Web. En este contexto, el lenguaje de marcado XML [1] se ha convertido en el estándar *de facto* para la representación de información semi-estructurada, y se han definido lenguajes de consulta como XPath y XQuery [1] que permiten la realización de consultas tanto por estructura como por contenido. Sin embargo, estos lenguajes asumen que el usuario tiene pleno conocimiento de la estructura de los documentos, de manera que solamente permiten formular consultas *exactas*: bien se obtiene un resultado que cumple exactamente las restricciones expresadas en la consulta, bien el resultado de la consulta es vacío. Esta suposición resulta poco realista. Es por ello que, recientemente, tanto desde el ámbito de las BD como desde el área de la RI se

* Financiado parcialmente por el MEC ref. TIN2009-14560-C03-02; por el MICINN ref. AP2007-02484 (FPU), para Ana-Cerdeira Pena; y por Fondecyt ref. 1-080019 (Chile), para el tercer autor.

han venido desarrollando distintos trabajos que tratan de introducir cierto grado de *flexibilidad* en las búsquedas que, por contenido y estructura, se realizan sobre la información almacenada en documentos XML [6,11,14].

Uno de estos trabajos es el que se recoge en [5,8,9], donde los autores presentan una extensión al lenguaje de consulta XPath. A través de este lenguaje, el usuario puede utilizar palabras clave para realizar búsquedas por contenido (como en RI), y restricciones *flexibles* sobre la estructura del documento, permitiendo así la recuperación de documentos y/o fragmentos de documentos XML con una estructura *aproximada* a la definida en la consulta. A diferencia del lenguaje XPath, donde la evaluación de una consulta produce un conjunto de resultados, la evaluación de una consulta *flexible* devuelve un *ránking* de fragmentos/documentos XML, al igual que ocurre en el ámbito de la RI.

La definición de estos lenguajes ha dado lugar al estudio de nuevas estructuras de datos para la representación e indexación de documentos XML que permitan una evaluación eficiente de las consultas, también *flexibles*. En este sentido, el uso de representaciones comprimidas y autoindexadas, permite no sólo ahorrar espacio de almacenamiento, sino que además proporciona ahorros significativos de tiempo de procesamiento. En [2] se presentó una nueva estructura de datos denominada XML Wavelet Tree (XWT), un autoíndice comprimido para documentos XML creado mediante la adaptación de un autoíndice para textos planos [3] basado en el uso de wavelet trees. Además, se mostraba cómo el XWT podía ser usado para responder eficientemente consultas típicas XPath sobre estructura y contenido de los documentos XML. En este trabajo, basándonos en las consultas *flexibles* definidas en [5,8,9], mostramos y analizamos con experimentos cómo el XWT puede usarse también para resolver de manera eficiente estas nuevas extensiones de XPath empleando distintas estrategias heurísticas de evaluación de las consultas, en función de la frecuencia e incluso número de descendientes de los elementos XML implicados en la consulta.

2. Trabajo relacionado

En [3] se presentó una nueva forma de organizar los bytes que forman los códigos de un texto comprimido utilizando una técnica de compresión estadística semi-estática, basada en palabras y orientada a bytes [7,4,12]. Básicamente la idea consiste en ir sustituyendo cada palabra del texto por un código formado por bytes, pero en vez de almacenarlos de manera consecutiva se reorganizan en diferentes nodos de un árbol, denominado Wavelet Tree (WT). El XML Wavelet Tree (XWT) [2], por su parte, sigue la misma estrategia, pero adaptada para trabajar con documentos XML. Así, se distinguen dos vocabularios diferentes, uno para los *tags* o etiquetas, y otro para el resto de palabras del documento XML (*no-tags*), siendo el texto comprimido mediante la técnica de compresión *(s,c)-DC* [4]. Este compresor permite distinguir entre bytes que sólo pueden aparecer como último byte de un código marcando el final de éste (*stoppers*), y bytes que nunca finalizan un código (*continuers*). El número de *stoppers* y *continuers* de cada vocabulario depende de sus respectivos tamaños y distribución de frecuencias de sus palabras, y se eligen de manera que minimicen la ratio de compresión [4]. El hecho de usar este compresor permite, además, reservar cier-

tos bytes para usar como primeros bytes de los códigos asignados a *tags*; de tal forma que simplemente viendo el primer byte de un código ya podamos saber si está codificando una etiqueta o una palabra que no es una etiqueta. Asimismo, esta propiedad permite, implícitamente, mantener localizados todos los *tags* en determinados nodos del árbol XWT (ver la rama *B4* de la Fig. 1), y por ende, la estructura de nuestro documento XML, proporcionando así una forma eficiente de resolver consultas estructurales (para más detalles, consultar [2]).

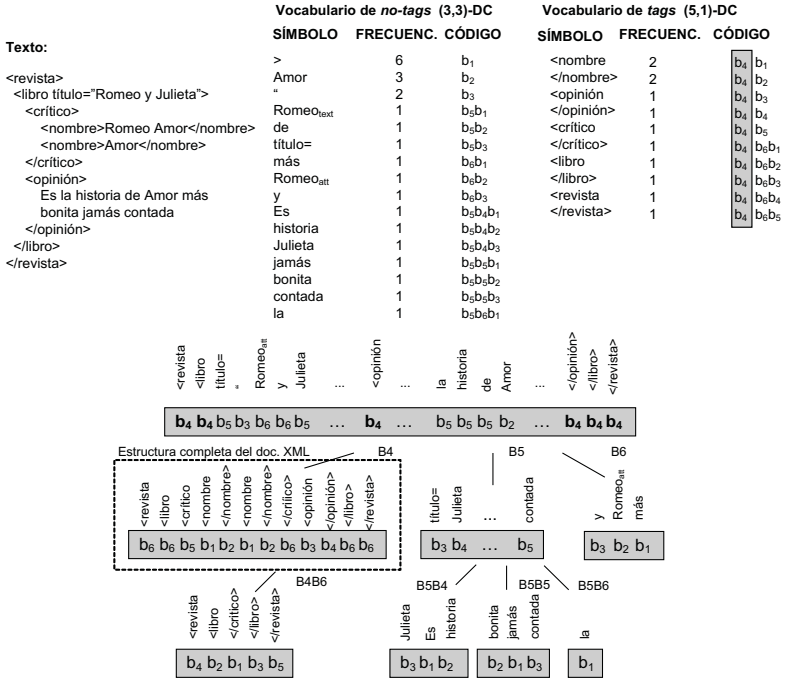


Fig. 1. Ejemplo de XWT.

La construcción del XWT se realiza en dos fases. En la primera de ellas, se lleva a cabo el *parsing* del documento XML de entrada y se crean los dos vocabularios previamente descritos. A continuación, las palabras de cada uno de los vocabularios son codificadas siguiendo un esquema de codificación *(s,c)-DC*. Tal y como se puede apreciar en la parte superior de la Fig. 1, b_1, b_2 y b_3 son los bytes usados como *stoppers* para los códigos de palabras que no son etiquetas, mientras que para las etiquetas, los *stoppers* son el conjunto de bytes formado por b_1, b_2, b_3, b_4 y b_5. Es importante fijarse también cómo el byte b_4 aparece como primer byte de todos los códigos correspondientes a etiquetas, pero no en ninguno de los códigos del vocabulario de palabras que no lo son (ver bytes sombreados en la columna `CÓDIGO` del vocabulario de *tags*). La segunda fase consiste en realizar una segunda pasada sobre el texto, reemplazando cada palabra por su código correspondiente, y organizar sus bytes en los distintos nodos del XWT. La raíz del árbol contiene el primer byte de cada código, siguiendo el mismo orden de las palabras que dichos códigos codifican en el texto original. Cada

nodo X del segundo nivel, contiene los segundos bytes de aquéllos códigos cuyo primer byte es x, siguiendo nuevamente el orden del texto. Es decir, el segundo byte correspondiente a la i-ésima ocurrencia del byte x en la raíz, estará situado en la posición i en el nodo X. Esta misma idea se utiliza para crear los niveles inferiores del árbol, que tendrá tantos como número de bytes tenga el código más largo. En la Fig. 1, se muestra un ejemplo de construcción del XWT. En la parte inferior de la figura se puede observar que el byte b_5 es el quinto byte de la raíz porque es el primer byte del código asignado a *Julieta*, la quinta palabra del texto. Por su parte, su segundo byte, b_4, está situado en la segunda posición del nodo $B5$, porque *Julieta* es la segunda palabra en la raíz que empieza por b_5. Del mismo modo, su tercer byte, b_3, está en el tercer nivel en la primera posición del nodo $B5B4$, porque su segundo byte es el primer b_4 en el nodo $B5$.

En general, el XWT permite representar un documento ocupando aproximadamente el 35 % de su tamaño original, y construirlo consume prácticamente el mismo tiempo que comprimir el documento; es decir, la reorganización de los códigos se realiza sin apenas coste a mayores sobre el tiempo de compresión.

2.1. Operaciones sobre el XWT

Dado que el XWT se basa en la misma estrategia de creación de un WT, las operaciones de búsqueda y recuperación de texto se resuelven usando dos operaciones básicas, *rank* y *select*. Sea $B{=}b_1,\ b_2,\ldots,b_n$ una secuencia de bytes:

- $rank_b(B,p) = i$ si el byte b aparece i veces en B hasta la posición p.
- $select_b(B,j) = p$ si la j-ésima ocurrencia de b en B está en la posición p.

A través de operaciones *rank* es posible recuperar la palabra de una posición cualquiera del texto, mientras que usando operaciones *select* se puede localizar eficientemente una determinada ocurrencia de una palabra. Por ejemplo, supongamos que queremos localizar la primera ocurrencia de la palabra *Julieta* en el ejemplo de la Fig. 1. La búsqueda comenzará en el nodo hoja $B5B4$, ya que el código de *Julieta* es $b_5b_4b_3$. En este nodo buscamos la posición de la primera ocurrencia del byte b_3 (el último byte de *Julieta*), calculando $select_{b_3}(B5B4, 1) = 1$. De esta forma obtenemos que esa posición es la 1. Todas las palabras cuyo último byte se almacena en el nodo $B5B4$ están representadas en el nodo $B5$ con el byte b_4, y siguen el mismo orden que en el texto original. Así, el valor 1 que hemos obtenido con la operación *select* anterior nos indica que el primer byte b_4 del nodo $B5$ es el correspondiente a la primera ocurrencia de *Julieta* en el documento. Nuevamente, calculando $select_{b_4}(B5, 1) = 2$, obtenemos que nuestro código es el segundo empezando por b_5 en el nodo raíz. Finalmente, calculamos $select_{b_5}(raiz, 2) = 5$, que nos dice que la primera ocurrencia de *Julieta* aparece en la posición 5 del texto. De forma análoga se puede recuperar una palabra del texto, empezando por la raíz del XWT y realizando operaciones *rank* sucesivas hasta alcanzar el nodo hoja que corresponda.

Puede observarse que el rendimiento del XWT depende directamente de la implementación de las operaciones *rank* y *select*[1]. Dado que su descripción queda fuera del ámbito de este trabajo, no se explicarán aquí; pero es posible encontrar

[1] Se basa en una estructura de contadores parciales que evita tener que contabilizar el número de ocurrencias de un byte desde el principio de un nodo del XWT. Existe un *tradeoff* espacio-tiempo.

más detalles en [3]. Allí también se muestra cómo los WT compiten con los II ganándoles en todo tipo de operaciones, cuando se dispone de poco espacio.

En el contexto de los lenguajes de consulta XML, también se ha demostrado el buen comportamiento del XWT, a la hora de resolver consultas típicas XPath sobre estructura y contenido de los documentos XML. En [2] se muestran varios ejemplos de procedimientos que permiten la búsqueda eficiente de atributos con un determinado valor, o la búsqueda de *elementos* XML que contengan una determinada palabra, para resolver consultas XPath del estilo *//resumen* [*contains*(., *Madrid*)]. Puesto que el XWT es una representación exacta de un documento XML, cualquier operación que se haga sobre el documento original, puede también realizarse sobre esta representación. Por lo tanto, todas las consultas XPath y cualquier extensión a ellas, se pueden resolver usando el XWT, pero de manera mucho más eficiente, haciendo uso de las propiedades de autoindexación que ofrece. En este artículo nos centramos en mostrar cómo el XWT puede resolver también eficientemente consultas con restricciones *flexibles* como las propuestas en [5,8], centrándonos en dos de las restricciones estructurales más importantes allí definidas, *below* y *near*.

3. Consultas XPath flexibles

El lenguaje de consulta XPath asume un modelo *booleano* de selección de elementos XML; es decir, particiona el conjunto de elementos de un documento XML en aquéllos que cumplen exactamente la condición que expresa una consulta, y aquéllos que no la cumplen. Sin embargo, en algunos escenarios asumir este modelo no resulta apropiado. Aún cuando un documento XML tiene asociado su correspondiente esquema, éste puede no estar disponible al usuario. Incluso distintos documentos XML compartiendo el mismo esquema pueden ser muy diferentes. Además, un mismo árbol XML puede ser descrito en ocasiones utilizando diferentes esquemas. En consecuencia, los usuarios deben realizar lo que se conoce como consultas *ciegas* [10], consultas sin un conocimiento preciso del esquema o estructura del documento XML con el que se está trabajando. En estos casos, la posibilidad de utilizar un lenguaje de consultas *flexible* que permita la realización de consultas *aproximadas* resulta de gran ayuda. Con este objetivo, en [5,8,9] los autores presentan un nuevo lenguaje que permite al usuario especificar restricciones *flexibles* tanto sobre la estructura como sobre el contenido de documentos XML. Dichas restricciones se definen como extensiones de XPath, de tal forma que las consultas del lenguaje propuesto se pueden expresar utilizando la sintaxis estándar XPath y las nuevas restricciones. Estas restricciones *flexibles* expresan una condición *aproximada* sobre la estructura y/o contenido de los documentos y su evaluación produce una puntuación o valor que expresa el grado de compatibilidad de la estructura y/o contenido de los fragmentos recuperados con respecto a la condición *exacta*. A continuación se describen las dos restricciones estructurales flexibles en las que nos centramos en este trabajo.

3.1. Añadiendo flexibilidad a la estructura: below y near

La restricción *below*, insertada como eje dentro de un *path* de búsqueda, selecciona todos aquellos elementos XML, atributos o texto de elementos que

son descendientes del elemento sobre el que se expresa la condición. Por ejemplo, la consulta *//noticia BELOW //imagen*, recuperaría todos los fragmentos correspondientes a elementos *imagen* que tuviesen como ancestro un elemento etiquetado como *noticia*. En la Fig. 2 a), 2 b) y 2 c) se pueden ver ejemplos de posibles fragmentos recuperados por la consulta anterior. Por su parte, *near* es una generalización de *below*, ya que extiende la selección a elementos que son descendientes, ancestros o que "rodean" al elemento de la condición. La siguiente consulta, *//noticia NEAR //imagen*, recuperaría todos los fragmentos correspondientes a elementos *imagen* cercanos a algún elemento etiquetado como *noticia*, pero no necesariamente en su mismo *path* desde la raíz (ver Fig. 2 d)).

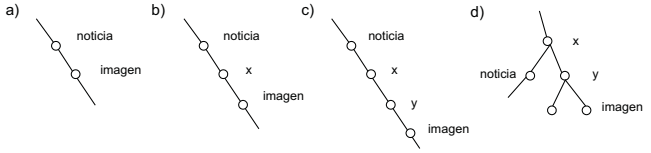


Fig. 2. Ejemplos de fragmentos recuperados por *below* (a), b), c)) y por *near* (d)).

A la hora de evaluar ambas restricciones se utiliza una función de evaluación que define el grado de *matching* entre el *path* especificado en la consulta flexible y el *path* del fragmento/s recuperado/s. En el caso de *below*, si tenemos una consulta del estilo *//A BELOW //B*, el *path* que mejor se ajustaría a la consulta (i.e. el fragmento *ideal*) sería aquél donde B es hijo directo de A; es decir, A/B. Cuanto mayor es la distancia entre A y B, menor debe ser la relevancia del *path*; esto es, la función de evaluación debe ser inversamente proporcional a la distancia entre A y B. Definimos así dicha función como $Match(c_p,\ f_p) = 1/d(A,B)$, donde c_p es el *path* de consulta, f_p es el *path* del fragmento recuperado, y $d(A,B)$ es la distancia entre los elementos A y B en f_p, dada por el número de aristas de separación entre ellos. El valor $Match(c_p,\ f_p)$ es lo que se denomina *Retrieval Status Value* (RSV) del fragmento recuperado con respecto a la consulta. En el siguiente ejemplo, se pueden ver los distintos fragmentos y su respectivo RSV en respuesta a la consulta *//libro BELOW //autor*, considerando 3 la longitud máxima de un *path* en el documento de búsqueda y donde $*$ representa un nodo cualquiera en el *path* que va de *libro* a *autor*: *i*) *libro/autor*, RSV=1; *ii*) *libro/*$*$*/autor*, RSV=1/2=0.5; *iii*) *libro/*$*$*/*$*$*/autor*, RSV=1/3=0.33.

En el caso de *near* los *paths* que mejor se ajustan a esta restricción son A/B y B/A. A mayor distancia entre A y B, menos relevante será un fragmento, con lo que es posible seguir definiendo la misma función de evaluación que para *below*, es decir, $Match(c_p,\ f_p) = 1/d(A,B)$. En teoría, se pueden identificar infinitos *paths* cuando se utilizan las restricciones *below* y *near*; en la práctica, se considera un límite máximo definido por el usuario. Por ejemplo, *//libro* $NEAR_3$ *//autor*, devolvería todos aquellos fragmentos correspondientes a elementos *autor*, que estuviesen "cerca" de algún *libro*, a una distancia no superior a 3.

4. Consultas XPath flexibles sobre XWT

Tal y como se explicó en la Sección 2, la estructura de un documento XML se representa de modo compacto y localizado en nodos específicos del XWT, ya

que los *tags* se representan en ellos siguiendo el mismo orden que en el documento. Por lo tanto, es posible resolver consultas *flexibles* sobre la estructura de un documento haciendo uso únicamente de esos nodos y omitiendo el resto del texto comprimido. Con ello se consigue un gran ahorro del tiempo de procesamiento. Asimismo, el XWT hace uso también de una estructura de bits [13], que permite la navegación entre los distintos elementos del documento XML, proporcionando de manera eficiente el padre o incluso el i-ésimo hijo de un elemento dado.

Below: a la hora de resolver consultas con la restricción *below*, como por ejemplo, *//revista* $BELOW_2$ *//volumen*, que trataría de recuperar todos los elementos *volumen* descendientes de algún elemento *revista*, a distancia no superior a 2, podemos proceder de dos formas diferentes: `1)` localizar los elementos *volumen* y buscar una ocurrencia de *revista* entre sus ancestros, a distancia menor o igual que 2; `2)` recorrer los elementos *revista* y chequear sus descendientes en búsqueda de elementos *volumen* hasta alcanzar la máxima distancia permitida. Para saber en qué dirección se debe operar y optimizar los tiempos de procesamiento, hemos desarrollado distintas estrategias heurísticas. Sea *//A* $BELOW_d$ *//B* nuestra consulta; f_A y f_B, las frecuencias respectivas de A y B; y h_{A_1} y h_{A_2}, el número medio de hijos a distancia 1 y 2 de A, respectivamente; se proponen las siguientes estrategias de evaluación que son analizadas en la Sección 5: *a*) Si $f_A < f_B \rightarrow$ `2)`; si no $\rightarrow$ `1)`; *b*) Utilizar siempre `1)`; *c*) Si $f_A * (h_{A_1} + h_{A_2}) < f_B * 2 \rightarrow$ `2)`; si no $\rightarrow$ `1)`.

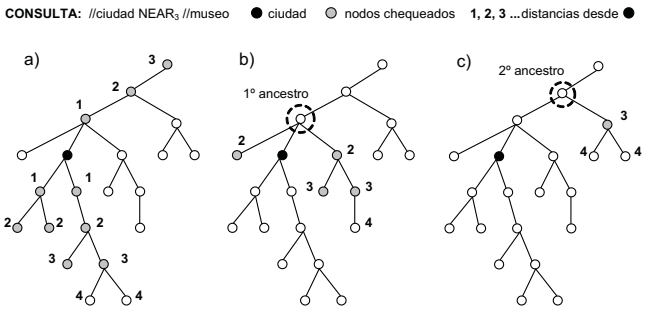


Fig. 3. Restricción *near*: a) operaciones *below*, b) descendientes del 1^{er} ancestro, c) descendientes del 2^o ancestro

Near: en el caso de la restricción *near* se sigue un procedimiento diferente. Supongamos la siguiente consulta: *//ciudad* $NEAR_3$ *//museo*, con la que pretendemos recuperar todos los elementos *museo* que tengan como ancestro, descendiente o a su alrededor, y siempre a distancia no mayor que 3, un elemento *ciudad*. Dado que se trata de una generalización de *below*, primero se realizan dos operaciones *below*, *//ciudad* $BELOW_3$ *//museo* y *//museo* $BELOW_3$ *//ciudad*. Con ellas cubrimos todos los casos de relaciones ancestro-descendiente (o descendiente-ancestro) que se pudiesen dar entre *ciudad* y *museo* en el documento XML (ver Fig. 3 a)). A continuación, se procede tomando las ocurrencias del elemento menos frecuente y chequeando los descendientes de ancestros consecutivos, hasta alcanzar la máxima distancia permitida. Es decir, siguiendo el ejemplo expuesto, supongamos que *ciudad* es el elemento menos frecuente. Una vez

localizada una ocurrencia de *ciudad*, vamos a su primer ancestro y chequeamos sus descendientes, hasta de segundo nivel, en búsqueda de un elemento *museo* (puesto que la máxima distancia permitida es 3 y el primer ancestro se encuentra a distancia 1. Ver Fig. 3 b)). Después se repite el mismo proceso, pero con el segundo ancestro, y chequeando sus descendientes de primer nivel (ya que el segundo ancestro está a distancia 2 y la máxima permitida es 3. Ver Fig. 3 c)). En general, si la distancia máxima es d, el proceso continúa hasta alcanzar el i-ésimo ancestro, donde $i = d - 1$, reduciendo el nivel de los descendientes a comprobar en cada paso del algoritmo.

Consultas por contenido y estructura: para resolver consultas con restricciones sobre la estructura y el contenido de un documento XML usando el XWT, se sigue la siguiente estrategia. Supongamos que tenemos la siguiente consulta (ver Fig. 4): *//libro* [$contains(., viaje)$] $BELOW_2$ *//imagen*. Se empieza localizando la primera ocurrencia de *viaje* en la raíz del XWT, mediante operaciones *select* consecutivas desde el correspondiente nodo hoja. Luego, se contabiliza el número de ocurrencias de las etiquetas de apertura y cierre de *libro* que hay hasta esa posición. Con ello podemos saber cuántas ocurrencias de *libro* contienen a esa ocurrencia de la palabra. Sin embargo, sólo nos interesan aquellos libros que además satisfagan la restricción estructural *libro* $BELOW_2$ *imagen*, la cual se chequeará para cada una de las ocurrencias halladas. En el ejemplo, se ve cómo la primera ocurrencia de *viaje* está contenida dentro de la primera ocurrencia de *libro*; pero al no cumplir ésta la restricción *below*, no se añade al resultado.

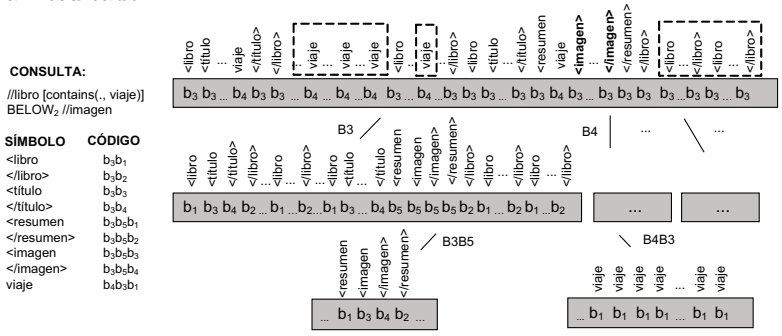


Fig. 4. Ejemplo de consulta sobre contenido con una restricción estructural flexible.

A continuación, en vez de repetir el mismo proceso con la siguiente ocurrencia de la palabra (en este caso, la 2ª ocurrencia de *viaje*), podemos saltar parte del texto si localizamos la siguiente ocurrencia de *libro* que satisfaga la restricción estructural, y que sea posterior a la ocurrencia de *viaje* que habíamos localizado en el paso previo. Siguiendo el ejemplo de la Fig. 4, la ocurrencia de *libro* que nos interesa es la tercera, no la segunda, ya que ésta no cumple la restricción impuesta por *below*. Una vez localizada, basta con mirar si contiene al menos una ocurrencia de *viaje*. Dado que contiene la 6ª ocurrencia de *viaje*, los elementos *imagen* relacionados con la tercera ocurrencia de *libro* (ver ocurrencias destacadas en negrita en la Fig. 4) formarán parte del resultado. Es importante observar que haciendo esto conseguimos ahorrar tiempo, al no tener que procesar

todas aquellas ocurrencias de la palabra que se encuentran antes de la tercera ocurrencia de ⟨*libro* y que no aportan nada a la búsqueda (ver ocurrencias de *viaje* encerradas en rectángulos en la Fig. 4). Tras esto, se repite el proceso completo tomando la primera ocurrencia de *viaje* posterior a la etiqueta de apertura de la ocurrencia de *libro* con la que hemos operado (si se permite el anidamiento de elementos iguales), o bien posterior a su etiqueta de cierre (si no se permite el anidamiento de elementos iguales). Con ello conseguimos, nuevamente, saltar aquellas ocurrencias de *libro* que no contienen ninguna ocurrencia de *viaje*, ahorrando así su procesamiento (ver ocurrencias de *libro* encerradas en un rectángulo en la Fig. 4). Nótese que este mismo procedimiento puede ser utilizado con cualquier otra restricción estructural, y cuando la condición sobre el contenido se aplica sobre una frase o incluso el valor de un cierto atributo.

5. Resultados experimentales

Para evaluar la eficiencia del XWT resolviendo consultas *flexibles* se han realizado una serie de experimentos[2]. Con este propósito se han utilizado tres documentos XML diferentes[3], cuyas propiedades (tamaño (en MBytes), número de elementos XML (EN)(x10^3), máximo nivel de anidamiento (MP), ratio de compresión (R, en %), y tiempo de compresión (TC) y descompresión (TD) (en seg.)) se muestran en el lado izquierdo de la Fig. 5. Se puede observar cómo el XWT es capaz de representar cada documento ocupando únicamente el 30 %-35 % de su tamaño original. Para correr los experimentos hemos usado una implementación del XWT que gasta un 3 % de espacio a mayores en las estructuras de contadores parciales que permiten agilizar las operaciones de *rank* y *select*.

Fig. 5. Características y propiedades de compresión de los documentos utilizados y tiempos medios de ejecución para *below*

doc.	tam.	EN	MP	R	TC	TD
nasa	23.89	476	8	31.64	2.90	0.28
0.5d	55.32	832	12	32.18	6.65	0.66
1d	111.12	1,666	12	31.91	12.46	1.32

	BELOW	nasa FR (ms)	nasa HP (ms)	nasa HJ (ms)	0.5d FR (ms)	0.5d HP (ms)	0.5d HJ (ms)	1d FR (ms)	1d HP (ms)	1d HJ (ms)
d_2	f_{baja}	7.30	10.47	**6.66**	37.02	44.84	**17.70**	74.34	89.46	**33.65**
	f_{alta}	33.26	38.29	**29.00**	93.16	115.41	**55.36**	194.73	237.77	**112.00**
d_3	f_{baja}	7.73	10.54	**6.68**	37.81	45.88	**17.78**	75.14	93.92	**35.74**
	f_{alta}	33.46	38.66	**29.46**	97.64	124.75	**56.13**	200.52	241.81	**113.22**

Distinguimos dos escenarios de prueba diferentes. Por un lado, evaluamos el comportamiento del XWT a la hora de resolver consultas estructurales con las restricciones *below* y *near* (p.ej. *//libro* $BELOW_3$ *//ref*). Por otro lado, combinamos dichas restricciones con una búsqueda por contenido dentro de la misma consulta (p.ej. *//libro* [$contains(., CERI)$] $BELOW_3$ *//ref*). En ambos casos, los conjuntos de consultas de prueba (formados cada uno por 20 consultas) se han dividido según la frecuencia f (alta o baja) de los elementos de la consulta, de acuerdo con las características de cada documento, y se han considerado dos distancias máximas, d, entre elementos, 2 (d_2) y 3 (d_3).

En la parte derecha de la Fig. 5, se presentan los tiempos medios de ejecución para *below* utilizando las estrategias heurísticas de optimización *a*) (FR), *b*)

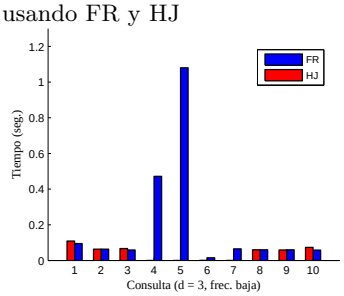
[2] En una máquina Intel®Pentium®Core 2 Duo 2.13 GHz, 4 GB dual-channel DDR-667Mhz RAM, con Ubuntu 8.04 GNU/Linux (kernel version 2.6.24.23). El compilador usado fue *gcc* version 4.2.4.

[3] `nasa` (http://xml.nasa.gov/); `0.5d` y `1d` (*XMark Project* - http://monetdb.cwi.nl/xml/)

(HP) y *c*) (HJ) descritas en la Sección 4. Como se puede observar, XWT invierte tiempos del orden de unos pocos milisegundos para resolver las consultas, independientemente de la estrategia. Sin embargo, es posible darse cuenta también que la estrategia que mejor resultados ofrece es HJ (ver valores en negrita). Esto es debido a que tomar el elemento menos frecuente (FR) no siempre es una buena opción, ya que puede suceder que éste tenga un número arbitrariamente grande de descendientes, los cuales habría que chequear si dicho elemento se encuentra en el lado izquierdo de la consulta. En ese caso, optar por el más frecuente, que estaría en el lado derecho, puede suponer un ahorro de tiempo; ya que implicaría seguir una dirección *hijo-padre*, y chequear, a lo sumo, tantos ancestros como indicase la distancia. Por su parte, seguir siempre esta dirección, como se hace en HP, basándose en el hecho de que cada elemento tiene un sólo padre y por tanto teniendo que chequear sólo un *path* hasta la raíz, resulta demasiado estricto.

Fig. 6. Tabla de tiempos medios de ejecución para *near* y gráfica de tiempos para ejemplos de consultas *near* sobre el documento `0.5d` usando FR y HJ

NEAR		nasa		0.5d		1d	
		FR (ms)	HJ (ms)	FR (ms)	HJ (ms)	FR (ms)	HJ (ms)
d_2	f_{baja_1}	57.16	**11.72**	122.44	**69.07**	212.45	**139.47**
	f_{baja_2}	10.82	11.72	68.41	69.07	145.57	139.47
	f_{alta}	**56.32**	67.92	**129.91**	174.02	**272.08**	334.84
d_3	f_{baja_1}	58.91	**12.86**	179.24	**76.49**	583.55	**162.94**
	f_{baja_2}	12.46	12.86	78.82	76.49	167.57	162.94
	f_{alta}	**86.60**	100.77	**171.99**	207.05	**352.89**	424.86



Teniendo en cuenta los resultados obtenidos en *below*, para el caso de estudio de *near* se optó por utilizar las estrategias heurísticas FR y HJ, a la hora de resolver las operaciones *below* en las que se basa. En la tabla de la Fig. 6 se muestran los resultados (en negrita aparecen marcados los que mejores valores ofrecen, para cada caso). En ella se puede observar cómo efectivamente, si realizamos un análisis comparativo entre *below* y *near*, la diferencia de tiempos cuando incrementamos la distancia es mayor en *near* que en *below*. Cuando trabajamos con *below*, cada incremento en la distancia puede implicar chequear los descendientes de un nivel más abajo (en el caso de elementos que tienen suficiente nivel de anidamiento) o un ancestro más arriba (en el mejor de los casos). Sin embargo, en *near* incrementar la distancia implica visitar los ancestros de uno o más niveles hacia arriba y luego chequear todos sus descendientes hasta un determinado nivel. A medida que se toman ancestros más alejados, mayor es la posibilidad de estar cerca de la raíz, y por tanto generalmente, mayor el número de descendientes a chequear.

Si ahora nos centramos en comparar el efecto de usar las estrategias FR y HJ dentro de las operaciones *below* que realiza *near*, se obtienen conclusiones realmente interesantes. En el caso de emplear FR, las operaciones *below* se hacen, al igual que en la última parte de *near* (correspondiente al chequeo de descendientes de ancestros sucesivos), sobre el elemento menos frecuente. Esto implica que sólo es necesario localizar las ocurrencias del elemento menos frecuente para resolver la consulta *near*. Esta característica, sin embargo, no siempre puede ser

aprovechada cuando se utiliza HJ. De acuerdo con esta heurística, si en al menos una de las dos operaciones *below* incluídas dentro de *near* se toma como elemento sobre el cual operar aquél que no es el menos frecuente, se tienen que localizar las ocurrencias de ambos elementos implicados en la consulta, al ser necesarios los dos para la resolución global de *near*. Si nos fijamos en los valores obtenidos para el caso de consultas que trabajan sobre elementos de frecuencia alta (ver filas f_{alta} en la Fig. 6) vemos cómo resulta mejor emplear la estrategia FR, ya que cuando se trabaja con estas frecuencias, utilizar HJ y tener que localizar los elementos de ambos lados de la consulta, consume más tiempo que el que se pueda perder empleando FR, yendo siempre por el menos frecuente. Si ahora observamos los valores de consultas que trabajan sobre elementos de frecuencia baja (ver filas f_{baja_1} en la Fig. 6) se da una situación diferente; es la estrategia HJ la que obtiene mejores resultados. Sin embargo esto se debe, en parte, a la desviación sobre la media que provocan determinadas consultas. Es decir, a la hora de trabajar con elementos de frecuencia baja, si se utiliza FR, hay determinadas consultas en que operar con el elemento menos frecuente puede aumentar mucho los tiempos si éste tiene un número de descendientes elevado (tal y como se vio cuando se analizaron los resultados para *below*). En f_{baja_1} se presentan los tiempos medios de ejecución teniendo en cuenta estas consultas, mientras que en f_{baja_2} se muestran esos mismos tiempos sin considerar esos casos especiales. En esta última se puede apreciar cómo, sin tener en cuenta esas consultas, las estrategias FR y HJ darían resultados muy similares. Asimismo, en la gráfica de la Fig. 6 se muestran los tiempos de ejecución de algunas consultas para el documento `0.5d` empleando las estrategias FR y HJ, donde los casos 4, 5, 6 y 7 constituyen ejemplos de estas consultas especiales. En ellas se observa cómo el seguir la estrategia HJ evita tiempos elevados.

Tabla 1. Tiempo medio de ejecución de consultas por contenido combinadas con *below* y *near*

		nasa		0.5d	
		BELOW (ms)	NEAR (ms)	BELOW (ms)	NEAR (ms)
d_3	f_{baja}	<u>2.81</u>	<u>4.21</u>	202.13	3790.61
	f_{alta}	<u>21.79</u>	472.84	<u>49.32</u>	2140.43

En el segundo escenario hemos usado los mismos conjuntos de prueba que en el contexto previo (donde sólo se trabajaba con restricciones estructurales) y hemos añadido una restricción sobre el contenido de los elementos, empleando palabras con frecuencia entre 50 y 100, elegidas aleatoriamente de los respectivos vocabularios *no-tags* de palabras. En este caso, se ha tomado 3 como distancia máxima. En la Tabla 1, se muestran los tiempos medios de ejecución. Se puede apreciar que hay situaciones en las que el hecho de usar la estrategia de *salto* explicada en la Sección 4 produce un importante ahorro (comparar valores subrayados en Tabla 1 y en las tablas de las Fig. 5 y 6); aunque no siempre es posible beneficiarse de ella (p.ej. cuando los elementos de la consulta no están relacionados estructuralmente, o cuando el elemento sobre el que se establece la condición de *contenido* es el más frecuente). En este tipo de consultas es necesario desarrollar nuevas estrategias que permitan aprovechar toda la potencia del *salto*, también en esas situaciones.

6. Conclusiones y trabajo futuro

El XWT es una estructura para la representación comprimida ($\approx 35\,\%$) y autoindexada de documentos XML, cuyas propiedades hacen que cualquier operación que se pueda realizar sobre el documento original, se pueda resolver de forma mucho más eficiente sobre el XWT que si se hace sobre el propio documento XML plano o incluso comprimido. En [2], se mostró cómo usando esta estructura era posible resolver eficientemente consultas típicas de XPath por estructura y contenido. En este trabajo se muestra, a través de experimentos, cómo el buen comportamiento del XWT se mantiene a la hora de resolver también extensiones de XPath, que tratan de introducir cierta *flexibilidad* en las búsquedas, acercándolas más al ámbito de la RI. Se han desarrollado distintas estrategias heurísticas de optimización para la resolución de las consultas, y se ha estudiado su influencia en la eficiencia, comprobándose cuáles ofrecían mejores resultados. Pero es necesario continuar este estudio, trabajando en el desarrollo de nuevas y mejores estrategias. Especialmente, es nuestro interés desarrollar nuevos algoritmos que extiendan los beneficios de la estrategia de *salto* en consultas que combinan restricciones por contenido y estructura, a todos los casos.

Referencias

1. XML 1.0, XPath 2.0 and XQuery 1.0. `http://www.w3.org/TR`.
2. N. R. Brisaboa, A. Cerdeira-Pena, G. Navarro. A compressed self-indexed representation of XML documents. *ECDL'09*, pp. 273–284, 2009.
3. N. R. Brisaboa, A. Fariña, S. Ladra, G. Navarro. Reorganizing compressed text. *SIGIR'08*, pp. 139–146, 2008.
4. N. R. Brisaboa, A. Fariña, G. Navarro, J. R. Paramá. (s, c)-dense coding: An optimized compression code for natural language text databases. *SPIRE'03*, pp. 122–136, 2003.
5. A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi, P. Spoletini. A fuzzy extension of the XPath query language. *J. Intell. Inf. Syst.*, 33(3):285–305, 2009.
6. T. T. Chinenyanga, N. Kushmerick. An expressive and efficient language for XML information retrieval. *J. Am. Soc. Inf. Sci. Technol.*, 53(6):438–453, 2002.
7. J. S. Culpepper, A. Moffat. Enhanced byte codes with restricted prefix properties. *SPIRE'05*, pp. 1–12, 2005.
8. E. Damiani, S. Marrara, G. Pasi. A flexible extension of XPath to improve XML querying. *SIGIR'08*, pp. 849–850, 2008.
9. E. Damiani, S. Marrara, G. Pasi. Fuzzyxpath: Using fuzzy logic and IR features to approximately query XML documents. *IFSA'07*, pp. 199–208, 2007.
10. E. Damiani, L. Tanca. Blind queries to XML data. *DEXA'00*, pp. 345–356, 2000.
11. N. Fuhr, K. Grobjohann. Xirql: a query language for information retrieval in XML documents. *SIGIR'01*, pp. 172–180, 2001.
12. E. Moura, G. Navarro, N. Ziviani, R. Baeza-Yates. Fast and flexible word searching on compressed text. *TOIS*, 18(2):113–139, 2000.
13. K. Sadakane, G. Navarro. Fully-functional static and dynamic succinct trees. *CoRR*, abs/0905.0768, 2009.
14. A. Theobald, G. Weikum. Adding relevance to XML. *WebDB'00*, pp. 105–124, 2001.