Un Prototipo para la Consulta sobre Documentos Transformados con LZCS

Joaquín Adiego[1], Gonzalo Navarro[2], y Pablo de la Fuente[1]

[1] Depto. Informática. Universidad de Valladolid, Valladolid, España.
{jadiego,pfuente}@infor.uva.es
[2] Depto. de Ciencias de Computación, Universidad de Chile, Santiago, Chile.
gnavarro@dcc.uchile.cl

Resumen. En este artículo presentamos nuevos algoritmos de consulta sobre documentos comprimidos mediante la transformación LZCS, que permite realizar algunas de las operaciones más habituales de XPath. La transformación LZCS permite comprimir sin pérdida colecciones de documentos XML. LZCS está inspirado en las técnicas Lempel-Ziv cuya idea subyacente es la de reemplazar subárboles por apuntadores a ocurrencias previas de los mismos y, por consiguiente, nuestros algoritmos intentan reutilizar el trabajo realizado sobre estos subárboles redundantes. Los algoritmos se han implementado en un prototipo denominado *lzcs-grep*. La principal ventaja de *lzcs-grep* es que permite procesar los documentos comprimidos obteniendo unos tiempos de respuesta muy pequeños en combinación con unas necesidades memoria muy bajas. Nuestros resultados experimentales muestran que *lzcs-grep* es competitivo respecto a otros procesadores de XPath incluso sobre documentos sin comprimir y presentando el mejor comportamiento cuando se aplica sobre los respectivos documentos comprimidos.

Palabras clave: Proceso de Consulta, Compresión, XML, XPath.

1 Introducción

El almacenamiento, intercambio y manipulación de texto estructurado como forma de representar información semiestructurada se está extendiendo a todo tipo de aplicaciones en todos los ámbitos, desde bases de datos textuales y bibliotecas digitales hasta servicios web y comercio electrónico. El texto estructurado, y en particular el formato XML, se está convirtiendo en un estándar para codificar información simple o compleja, de estructura fija o variable. Aunque en algún momento XML se ha considerado como una manera de describir datos estructurados, no ha sido hasta la reciente explosión de las aplicaciones *business-to-business* en donde se ha puesto de manifiesto su potencial para describir todos los documentos que intercambian y almacenan las organizaciones. Ejemplos de estos documentos son facturas, recibos, pedidos, pagos y otros tipos de formularios.

Aunque habitualmente la información que una organización almacena se mantiene en bases de datos relacionales y/o almacenes de datos, es importante guardar los respaldos digitales, en formato XML, de todos los documentos que se han intercambiado y/o producido a lo largo del tiempo. Un motor de recuperación de texto estructurado debe proporcionar un acceso aleatorio a dichos documentos estructurados de la misma manera que debe proporcionar servicios de búsqueda, visualización y navegación. En concreto, una propiedad extremadamente importante es la de permitir búsquedas de partes de la colección que posean una determinada propiedad. Dicha propiedad puede estar relacionada tanto con el texto como con la estructura del documento. XPath [12] es una de las maneras más populares para especificar partes de documentos XML.

En este artículo nos vamos a centrar en datos muy estructurados, tales como los formularios donde hay muy poco texto en cada campo. Las colecciones de este tipo de documentos son muy habituales en las apliacaciones *business-to-business* mencionadas con anterioridad. Además contienen mucha redundancia que los métodos de compresión tradicionales no son capaces de manejar adecuadamente. Hace unos años se ha propuesto una técnica, LZCS [1, 2], que aprovecha esta redundancia y además permite acceder con facilidad, visualizar y navegar sobre la colección comprimida. Las razones de compresión (tamaño comprimido dividido entre el tamaño sin comprimir) que obtiene LZCS son excelentes, inferiores al 10% para datos de tipo formulario XML.

Es habitual argumentar que el espacio en disco es barato y, por lo tanto, la compresión no es interesante. No obstante, la compresión no sólo ahorra espacio sino que también dismimuye los tiempos de transferencia de disco y red, los cuales son recursos muy valiosos. Este es el motivo por el que la compresión es interesante. Además, la compresión tiene la ventaja de disminuir considerablemente el tiempo de respuesta de las búsquedas, pues hay que analizar menos datos [8, 3].

En esta línea, el presente artículo propone un conjunto de algoritmos que permiten resolver las principales operaciones de XPath sobre el texto comprimido con LZCS. Estos son algoritmos de *streaming*, pues la colección no se indexa previamente sino que se inspecciona secuencialmente cuando se resuelve la consulta. La transformación LZCS comprime el texto reemplazando subárboles XML por apuntadores a ocurrencias previas del mismo. La principal idea subyacente en los algoritmos es reutilizar el trabajo que ya ha sido realizado sobre los subárboles repetidos.

Hemos implementado nuestros algoritmos en un prototipo denominado *lzcs-grep*. Los resultados experimentales muestran que *lzcs-grep* es competitivo respecto a los procesadores *streaming* de XPath que se han considerado en el estado del arte, incluso cuando el texto no está comprimido. Sobre la colección de formularios XML comprimida con LZCS, *lzcs-grep* proporciona las mejores resultados tanto en tiempo como en requisitos de memoria.

Este artículo está organizado de la siguiente manera. En la Sección 2 se recuerda brevemente qué es XML y XPath. La Sección 3 describe la transformación LZCS y comenta algunas de sus propiedades. La Sección 4 presenta nuestros algoritmos para resolver expresiones XPath sobre documentos transformados con LZCS. La Sección 5 contiene resultados empíricos comparando tanto los tiempos de respuesta como los requisitos de memoria de nuestra propuesta con otros sistemas de consulta XPath. Por último, en la Sección 6 se presentan las conclusiones y se indican algunas líneas de trabajo futuro.

2 XML y XPath

XML, eXtensible Markup Language [15], es la especificación de un lenguaje de marcado flexible para texto estructurado. El marcado se representa mediante marcas especiales denominadas *etiquetas* o *tags*, las cuales se insertan en el texto con el objeto de describir la estructura. Una etiqueta XML es una secuencia de caracteres delimitada entre dos caracteres especiales, "<" y ">". El texto dentro de las etiquetas es parte de la estructura de los documentos, el resto es el contenido. Las etiquetas son legibles por las personas, pero habitualmente se ocultan cuando el documento se visualiza, pues no son contenido pero indican cómo éste se debe entender.

Las etiquetas XML no se pueden superponer, es decir, una etiqueta no se puede cerrar hasta que se hayan cerrado todas las etiquetas que contiene en su interior. Esto induce una estructura jerárquica del documento, en la que cada nodo representa una etiqueta y cuyos hijos son sus atributos y las etiquetas que contienen. Cada etiqueta delimita una parte del texto del documento, que a su vez puede contener más etiquetas. Ya que las etiquetas tienen una longitud estrictamente positiva, ninguna pareja de etiquetas diferentes puede comenzar

en la misma posición. El orden en el que inicialmente se encuentran las etiquetas en el texto se corresponde con un recorrido en preorden del árbol representado. Por otro lado, las etiquetas de apertura de XML pueden contener atributos. Un atributo es es un par nombre-valor dentro de la etiqueta de apertura de un elemento y su finalidad es caracterizar la estructura de los documentos XML.

Además de las etiquetas descriptoras de la estructura, existen otros elementos en la especificación de XML como comentarios, comandos dirigidos a procesadores específicos, definiciones de tipos de documentos, etc. Habitualmente los lenguajes de consulta ignoran estos elementos y nosotros también lo haremos en este trabajo.

XPath, XML Path Language [12], es uno de los lenguajes más utilizados para seleccionar partes de una colección XML. Es una pieza esencial en lenguajes más completos como XQuery [14] y XSLT [13].

Un tipo de expresión importante en XPath es el *location path*. Un *location path* selecciona un conjunto de nodos de acuerdo a su relación con un contexto de nodos dados. Un *location path* es una secuencia de *location steps*, esto es, los nodos resultado de cada paso son el contexto de nodos para el siguiente. Los *location path* están formados por (i) un eje, que especifica la relación en el árbol del contexto y los nodos seleccionados; (ii) un nodo tipo, que especifica el tipo y/o nombre de los nodos seleccionados en el *location step*; y (iii) cero o más predicados que permiten refinar el conjunto de nodos seleccionados.

Aunque no existe una implementación completa y satisfactoria de XPath, hay muchas implementaciones funcionales (generalmente parciales) y propuestas algorítmicas. Algunos ejemplos son XSQ [11], TurboXPath [7], ViteX [5], Qexo, XMLCliTools, XMLStarlet, XML-Twig, XGrep, etc. (Véase la Sección 5.1).

3 La Transformación LZCS

La transformación LZCS [1, 2] está inspirada en el esquema de compresión Lempel-Ziv [17, 18], en el que se reemplazan subestructuras repetidas. LZCS reemplaza cada subestructura maximal repetida por una referencia a la *primera* ocurrencia de la misma en el texto ya transformado. El resto de los elementos permanecen inalterados. Las referencias se representan en la salida mediante unas etiquetas especiales. Dichas etiquetas especiales se forman mediante los delimitadores "<@" y ">" y contienen un entero estrictamente positivo que indica la posición absoluta, en el *texto transformado*, en la que comienza el elemento referenciado.

El resultado de esta transformación es un texto que goza de varias propiedades deseables: es legible por las personas, es "ASCII compliant", está bien estructurado, su descompresión es rápida, se puede navegar y visualizar con facilidad y se puede acceder aleatoriamente a él. Para más detalles consultar [2].

Otra ventaja que se comenta en [2] es que es posible realizar una búsqueda directa, limitada a palabras y frases, sobre el texto comprimido (al menos saber si aparecen o no en la colección). No obstante, es mucho más interesante la posibilidad de resolver búsquedas que exploten las posibilidades de XPath, en donde se pueden combinar predicados sobre contenido y estructura. Este es el objetivo de este artículo. De la misma manera que se puede buscar más rápido sobre el texto comprimido (de la forma simplificada comentada con anterioridad) o buscar más rápìdo sobre texto no estructurado [8, 3], es previsible que las consultas XPath también se resuelvan más rápido sobre el texto comprimido. Esto supone varios desafíos, tal y como se comenta en la siguiente sección.

4 Consultando Documentos Transformados con LZCS

En esta sección describiremos nuestros algoritmos de *streaming* que implementan las cinco operaciones de XPath que son las más habituales hoy en día en los *location steps*. Hemos implementado *location steps* individuales con los siguientes ejes o predicados:

Child: `A/child::B` o `A/B` selecciona nodos de tipo B que descienden directamente de un nodo de tipo A.

Descendant: `A/descendant::B` o `A//B` selecccciona nodos de tipo B que directa o transitivamente descienden de al menos un nodo de tipo A.

Parent: `B/parent::A` o `A/B/..` selecccciona nodos de tipo A que contienen directamente un nodo de tipo B.

Ancestor: `B/ancestor::A` selecccciona nodos de tipo A que directa o transitivamente contiene al menos un nodo de tipo B.

Text matching: `A[text()`$= \tau$`]` selecccciona todos los nodos de tipo A cuyo texto es τ.

Los *location paths* se forman concatenando *location steps*, por ejemplo `A//B/parent::C` selecciona nodos de tipo C que son padres de al menos un nodo de tipo B los cuales son descendientes de un nodo de tipo A. En este artículo nos centraremos en *location steps* individuales, no obstante los resultados se pueden extender a *location paths* de una manera sencilla: se resuelven de derecha a izquierda los *location steps* concatenados de forma individual, tomando los nodos respuesta de la parte izquierda del *location step* como posibles soluciones de la parte derecha del *location step*.

La principal idea para resolver los citados *location steps* es la de almacenar algunos resultados temporales en cada subárbol visitado en el texto, de esta forma se puede reutilizar el trabajo si un subárbol se referencia en el futuro. Los resultados temporales deben ser suficientes para reconstruir los resultados reales, y lo que se almacena dependerá del tipo de *location step* que se esté resolviendo.

El mecanismo general, representado en el Algoritmo 1, es el siguiente: (i) el documento se recorre secuencialmente; (ii) se introducen en una pila las etiquetas de apertura a medida que se van encontrando; y (iii) se sacan de la misma cuando se localiza su correspondiente etiqueta de cierre. Para cada nodo almacenado en la pila se mantienen dos conjuntos de nodos: el conjunto de respuestas (AS, *answer set*) y el conjunto de posibles respuestas (ACS, *answer candidate set*). El conjunto de respuestas contiene los nodos encontrados hasta el momento que responden a la consulta. El conjunto de posibles respuestas contiene nodos que pueden ser útiles para detectar otras respuestas (el significado exacto depende de la operación que se esté resolviendo). Una propiedad importante de estos conjuntos es que es posible operar con ellos desde el subárbol de su nodo, independientemente del resto del árbol.

Siempre que se encuentra una etiqueta de apertura, el nodo se introduce en la pila, se almacena su posición en el texto y se inicializan sus conjuntos de tal manera que $AS = ACS = \emptyset$. Cuando se encuentra una etiqueta de cierre se sacan los datos de la pila y se actualizan los conjuntos AS y ACS de su nodo padre (que estará en la cima de la pila) de acuerdo con la operación que se esté resolviendo. En cualquier caso se debe añadir el conjunto AS del nodo actual al conjunto AS del nodo padre pero, dependiendo de la operación, se pueden añadir otros nodos. También varía la forma de manejar los conjuntos ACS. Al finalizar el proceso de consulta, el conjunto AS de un nodo raíz imaginario, que se introdujo en la pila en primer lugar, contiene la respuesta a la consulta.

Para cada nuevo nodo que se introduce en la pila se debe almacenar una nueva "posible referencia" en el diccionario $Refs$. La clave de esta entrada es la posición en el texto transformado y los datos son los conjuntos AS y ACS. Cuando el algoritmo encuentra una referencia recupera estos conjuntos del diccionario y procede tal y como si los hubiera obtenido otra

vez del texto. Este planteamiento es correcto porque los conjuntos AS y ACS sólo deben depender del subárbol (el cual es idéntico al referenciado) y no de su contexto en el árbol.

Algoritmo 1: Proceso de consulta general

procedure *SolveQuery*()
$Refs.new()$
$Stack.new()$
$Root.AS \leftarrow \emptyset, Root.ACS \leftarrow \emptyset$
$Stack.push(Root)$
while *there are more tags* **do**
  $tag \leftarrow get\_current\_tag()$
  if (*tag is a start-tag*)
    $Node.AS \leftarrow \emptyset, Node.ACS \leftarrow \emptyset$
    $Node.tag \leftarrow tag$
    $Node.pos \leftarrow$ *current text position*
    $Stack.push(Node)$
  fi
  if (*tag is an end-tag*)
    $current \leftarrow Stack.pop()$
    $parent \leftarrow Stack.top()$
    Call corresponding Update_∗ (current, parent) procedure
    $Refs.insert(current.pos, current)$
    $parent.AS \leftarrow parent.AS \cup current.AS$
  fi
  if (*tag =* $<$ @*pos* $>$ *backward reference*)
    $current \leftarrow Refs.search(pos)$
    $parent \leftarrow Stack.top()$
    Call corresponding Update_∗ (current, parent) procedure
    $parent.AS \leftarrow parent.AS \cup current.AS$
  fi
od
Return Root.AS

El Algoritmo 2 muestra los procedimientos $Update\_*$ que actualizan la información en el nodo actual y en el padre de acuerdo a los datos contenido en el nodo actual. Estos procedimientos toman como argumentos los nodos actual (*current*) y padre (*parent*) así como los parámetros A, B y/o τ correspondientes a la consulta.

Child: El conjunto ACS contiene los nodos de tipo B que descienden directamente de la raíz de la subestructura, por lo que son sus hijos. Entonces, si la raíz de *current* es de tipo A, todos los elementos del conjunto ACS forman parte de la solución y se añaden al conjunto AS. Por otro lado, si *current* es de tipo B, se almacena en el conjunto ACS de *parent*.

Descendant: Es similar al de **Child**, excepto que en ACS almacenamos todos los nodos etiquetados como B que aparecen en el subárbol. El conjunto ACS de *current* se vacía sólo si para a formar parte de las respuestas certeras a la consulta, pues sino se podría utilizar por un nodo ascendente del padre. Por el mismo motivo, el conjunto ACS del nodo actual se añade al del padre.

Parent: El conjunto ACS es idéntico al de **Child**. En esta ocasión, cuando *current* es de tipo A y tiene hijos de tipo B, insertamos *current* en su propio conjunto AS, en vez de en el de sus hijos.

Ancestor: El conjunto ACS es similar al de **Descendant**, excepto que en este caso no se vacía porque sus elementos no se convierten en miembros de AS. En esta ocasión, cuando el nodo actual es de tipo A y tiene nodos de tipo B descendientes, la respuesta que se almacena en el padre es *current* en lugar de los descendiente de tipo B.

Text matching: Si el nodo actual es de tipo A y contiene el texto τ, se añade a su conjunto AS. No se utiliza el conjunto ACS.

Hay que destacar que la actualización de *current* se lleva a cabo sólo cuando el nodo es explícito. Cuando el nodo está referenciado, no es necesario llevarlo a cabo. Este hecho no está incluido en el pseudocódigo para mantenerlo lo más simple posible. También por simplicidad, hemos explicado los algoritmos de manera general. No obstante, se pueden llevar a cabo algunas optimizaciones. Por ejemplo, los conjuntos ACS en las operaciones **Parent** y **Ancestor** no son necesarios, pues sólo se necesita saber si están vacíos o no.

Algoritmo 2: Procedimientos de actualización de los nodos de datos actual y padre. Las actualizaciones de *current* se llevan a cabo sólo si no es una referencia.

procedure *Update_A/B*(*current*, *parent*)
if (*current.tag* = *A*)
  current.AS ← *current.AS* ∪ *current.ACS*
fi
current.ACS ← ∅
if (*current.tag* = *B*)
  parent.ACS ← *parent.ACS* ∪ {*current*}
fi

procedure *Update_A//B*(*current*, *parent*)
if (*current.tag* = *A*)
  current.AS ← *current.AS* ∪ *current.ACS*
  current.ACS ← ∅
fi
if (*current.tag* = *B*)
  parent.ACS ← *parent.ACS* ∪ {*current*}
fi
parent.ACS ← *parent.ACS* ∪ *current.ACS*

procedure *Update_A[text()=τ]*(*current*, *parent*)
if (*current* = *A* ∧ *current.text* = *τ*)
  current.AS ← *current.AS* ∪ {*current*}
fi

procedure *Update_B/parent::A*(*current*, *parent*)
if (*current.tag* = *A* ∧ *current.ACS* ≠ ∅)
  current.AS ← *current.AS* ∪ {*current*}
fi
current.ACS ← ∅
if (*current.tag* = *B*)
  parent.ACS ← *parent.ACS* ∪ {*current*}
fi

procedure *Update_B/ancestor::A*(*current*, *parent*)
if (*current.tag* = *A* ∧ *current.ACS* ≠ ∅)
  current.AS ← *current.AS* ∪ {*current*}
fi
if (*current.tag* = *B*)
  parent.ACS ← *parent.ACS* ∪ {*current*}
fi
parent.ACS ← *parent.ACS* ∪ *current.ACS*

5 Evaluación

Hemos desarrollado un prototipo en C++, denominado *lzcs-grep*, que implementa un sistema *streaming* de consulta que permite seleccionar nodos de una colección de documentos semiestructurados. Nuestro prototipo soporta un lenguaje de consulta que implementa los algoritmos descritos en la sección 4, que resuelven los *location steps* más populares.

En esta sección compararemos *lzcs-grep* con otros procesadores *streaming* de XPath. Es preciso comentar que *lzcs-grep* puede, como caso particular, procesar documentos XML tal cual, sin realizar la transformación, y así se puede comparar sobre un mismo fichero con los otros procesadores. Mostramos que *lzcs-grep* es competitivo en este experimento, lo que indica la conveniencia de utilizarlo incluso cuando la transformación LZCS no obtiene mucha compresión.

No obstante, el principal interés de *lzcs-grep* es la ventaja que obtiene cuando los documentos son muy compresibles (que es el caso de los formularios tipo XML). Nuestros experimentos sobre colecciones comprimidas muestran que *lzcs-grep* es imbatible en este escenario.

Como ejemplo de datos muy compresibles hemos elegido una colección de formularios tipo XML basada en XForms [16]. XForm es un dialecto de XML que representa la nueva generación de formularios en la Web. XForms separa la representación del contenido dividiendo los formularios XHTML tradicionales en tres partes (modelo, datos de instancia e interfaz de usuario) y, por consiguiente, permite la reutilización, proporciona un tipado robusto, reduce el número de *round-trips* del servidor, ofrece una independencia de dispositivo, y reduce la necesidad de *scripting*. XForms se está haciendo cada vez más popular para representar e intercambiar información y transacciones entre empresas.

5.1 Marco Experimental

Nuestros experimentos se han realizado en SuSE Linux 9.3, que se ejecuta en un ordenador con un procesador Pentium IV a 1.2 GHz y con 384 megabytes de memoria principal. Hemos elegido una colección de XForms de 48 megabytes que representan facturas de venta, las cuales son un documento legal chileno. Cada formulario tiene varios campos, por cuestiones de privacidad, los campos de los formularios se han generado aleatoriamente a partir del vocabulario real controlado (este hecho va en detrimento de la compresión). La Figura 1 muestra un extracto relevante de la estructura de la colección en su representación arbórea.

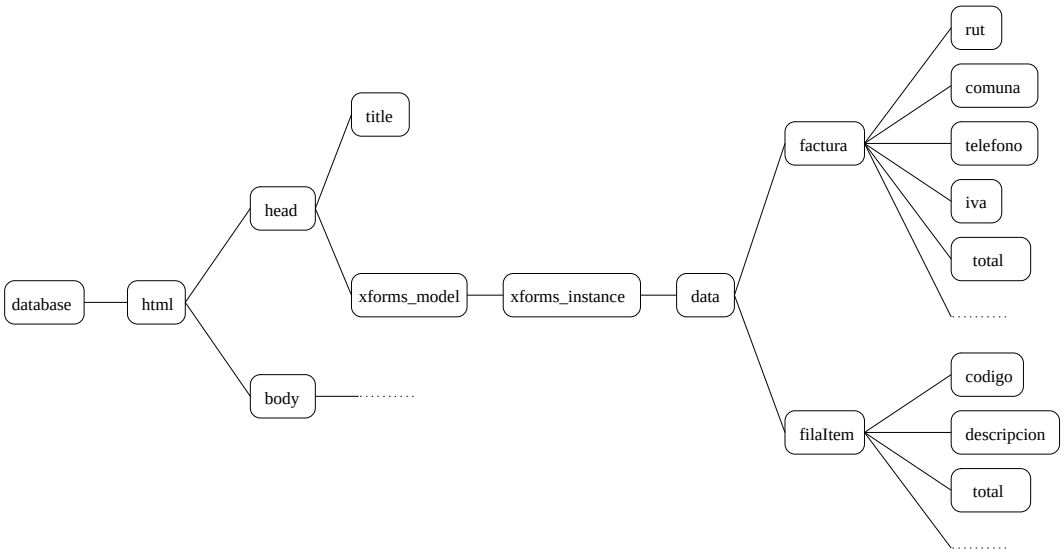

Fig. 1. Extracto de la estructura de la base de datos.

Para probar exhaustivamente nuestros algoritmos, hemos elegido varias consultas de cada uno de los cinco tipos de operaciones soportadas por *lzcs-grep*, intentando variar el número de nodos relevantes y la profundidad en la jerarquía en la que se encuentran los dichos nodos. Las consultas, junto con el número de nodos relevantes, se muestran en la Tabla 1.

Tabla 1. Consultas y número de nodos del resultado.

#	Consulta Child	Nodos
C1	`//head/title`	941
C2	`//filaItem/total`	232649
C3	`//factura/total`	941

#	Consulta Descendant	Nodos
D1	`//head//title`	941
D2	`//data//rut`	941
D3	`//head//descripcion`	941

#	Consulta Parent	Nodos
P1	`//head/title/..`	941
P2	`//filaItem/codigo/..`	232649
P3	`//factura/total/..`	941

#	Consulta Ancestor	Nodos
A1	`//title/ancestor::head`	941
A2	`//iva/ancestor::data`	941
A3	`//descripcion/ancestor::head`	941

#	Consulta Text Matching	Nodos
T1	`//comuna[text()="SANTIAGO"]`	825
T2	`//telefono[text()="6322000"]`	4
T3	`//descripcion[text()="P5"]`	16703

Hemos comparado *lzcs-grep* con los siguientes sistemas de consulta de código abierto:

- *XSQ 1.0*[3] resuelve consultas XPath sobre *streams* de datos XML. XSQ está escrito en Java, utiliza un parser SAX y, dada una consulta XPath, genera su correspondiente autómata [11].
- *XMLClitools 1.53-1*[4] suministra cuatro herramientas en línea de comandos que permiten buscar, modificar y formatear datos XML. La herramienta *xmlgrep* selecciona subconjuntos de ficheros XML mediante una notación propietaria y restringida.
- *XMLStarlet 1.0.1*[5] es un conjunto de utilidades en línea de comandos, escritas en C, que se pueden utilizar para transformar, consultar, validar y editar documentos y ficheros XML. Es similar a la manera de procesar ficheros de texto plano utilizando los diferentes comandos de UNIX presentes en cualquier distribución. Una de las herramientas busca ocurrencias en ficheros XML que respondan a las expresiones XPath suministradas.
- *XML-Twig 3.25*[6] es un módulo de Perl que busca determinados elementos en ficheros XML. No da soporte XPath completo, pero implementa un subgrupo muy reducido de las expresiones de XPath.
- *XGrep 0.3*[7] es una utilidad de tipo "grep" para documentos XML. XGrep devuelve todos los nodos de un fichero XML que responden a la expresión XPath suministrada.

Se han excluido otras implementaciones por diferentes motivos: TurboXPath [7] pertenece a IBM y aparentemente su código fuente no está disponible; Vitex [5] aún no está implementado; Qexo[8] actualmente es un compilador que genera código Java que evalúa la expresión, pero no es un sistema que procese las consultas *online*.

[3] http://www.cs.umd.edu/projects/xsq/
[4] `http://directory.fsf.org/all/xmlclitools.html`
[5] `http://xmlstar.sourceforge.net/`
[6] `http://search.cpan.org/~mirod/`
[7] `http://software.decisionsoft.com/pathanXgrepDocumentation.html`
[8] `http://www.gnu.org/software/qexo`

La Tabla 2 muestra los tiempos (en segundos) que cada sistema necesitó para resolver cada consulta. Para *lzcs-grep* se muestran los resultados tanto para la colección sin comprimir (filas "lzcs-grep (XML)") como para la colección transformada con LZCS (filas "lzcs-grep (LZCS)"). La colección de XForms que hemos utilizado es muy compresible: la transformación LZCS obtiene una razón de compresión del 6% [1, 2].

Tabla 2. Tiempo de respuesta en segundos para cada consulta y sistema. Se han omitido los casos en los que la consulta no se ha podido expresar en un sistema. Para *XMLClitools*, la consulta D1 no se puede expresar porque, en la notación que usa dicho sistema, coincide con la consulta C1. Para *XGrep*, las consultas "descendant" se abortaron cuando llevaban dos días ejecutándose.

	Consulta Child			Consulta Descendant			Consulta Text Matching		
Sistema	C1	C2	C3	D1	D2	D3	T1	T2	T3
XSQ	16.88	17.24	17.08	16.99	17.06	16.87	17.18	17.57	17.05
XMLClitools	8.59	75.44	40.26	—	39.92	86.32	—	—	—
XMLStarlet	204.00	21243.61	207.07	204.63	194.80	17271.24	306.08	325.39	486.56
XML-Twig	312.27	1203.97	365.51	—	—	—	294.99	291.89	932.21
XGrep	22.24	239.43	15.01	∞	∞	∞	12.24	11.75	22.04
lzcs-grep (XML)	53.33	55.90	53.74	53.02	52.65	55.49	55.68	55.43	56.48
lzcs-grep (LZCS)	2.69	4.48	2.73	2.68	2.71	4.69	2.90	2.75	3.11

	Consulta Parent			Consulta Ancestor		
Sistema	P1	P2	P3	A1	A2	A3
XSQ	16.81	16.85	17.03	17.02	17.02	16.70
XMLStarlet	889.40	25514.44	352.92	869.55	684.23	1054.92
XGrep	22.60	7719.04	12.69	18.51	16.63	26.74
lzcs-grep (XML)	52.56	56.04	52.74	52.05	52.84	55.19
lzcs-grep (LZCS)	2.73	4.43	2.64	2.66	2.64	2.79

Como se puede observar, *XSQ* es claramente el sistema más rápido sobre XML plano. Además es bastante estable pues sus tiempos de respuesta siempre están en torno a 16.5–17.5 segundos. No obstante, se debe mencionar que hemos representado los tiempos de usuario. En otros sistemas, los tiempos de usuario son muy similares a los tiempos de respuesta pero en el caso de *XSQ* los tiempos de respuesta son aproximadamente el doble que los tiempos de usuario, probablemente debido a la sobrecarga de la Máquina Virtual de Java.

Sobre XML plano, sólo *XGrep* y *XMLCliTools* son mejores que *XSQ* de forma ocasional. No obstante, ambos sistemas tienen unos tiempos muy inestables que pueden llegar a ser muy malos, concretamente cuando la respuesta está formada por muchos nodos. Por otro lado, *XMLStarlet* y *XML-Twig* son mucho más lentos que los demás.

En comparación, *lzcs-grep* es tan estable como *XSQ*, procesa el texto a una velocidad de 0.85–0.92 MB/seg independientemente de la consulta que esté resolviendo. Es unas tres veces más lento que *XSQ*, pero si consideramos los tiempos de respuesta este valor disminuye a 1.5 veces. En comparación con *XGrep*, *lzcs-grep* puede llegar a ser unas 5 veces más lento pero, por otro lado, también puede llegar a ser unas 100 veces más rápido. Una situación similar se presenta al compararlo con *XMLCliTools*. Los otros dos sistemas no son competitivos.

No obstante, la característica más importante de *lzcs-grep* es su rendimiento sobre el texto transformado con LZCS. Nuestro prototipo procesa el texto transformado a 0.64–1.06 MB/seg. Es importante comentar que este tiempo es muy dependiente del número de resultados que forman la respuesta a la consulta. Este valor, si la medición se traslada con respecto al documento sin comprimir, se traduce en 10–18 MB/seg, 5–6 veces mejor que

XSQ y 2.7–4.3 veces mejor que cualquier otro tiempo obtenido por cualquier otro sistema en cualquier consulta.

En la Tabla 3 hemos calculado la media de los tiempos (finitos) que ha necesitado cada sistema con el fin de obtener una visión general (muy) aproximada. *XSQ* lidera el rendimiento en la colección de XML plano. Los pocos casos en los que *XGrep* tiene un paupérrimo rendimiento penalizan severamente su rendimiento medio, y *XMLClitools* se muestra como una alternativa estable (aunque restringida). No obstante, el tiempo promedio de *lzcs-grep* está muy cercano a los tiempos reales y además ofrece una mayor estabilidad y cobertura de las operaciones de XPath. Sobre el texto transformado con LZCS, *lzcs-grep* es un orden de magnitud más rápido que cualquier otro. Su incremento de velocidad respecto al XML plano es consistente con la razón de compresión que se ha obtenido.

Tabla 3. Tiempo de respuesta medio y uso de memoria de cada sistema.

		Media por Consulta	
Sistema		Tiempo (seg)	Memoria (MB)
XSQ		16.82	256.00
XMLClitools		50.10	311.20
XMLStarlet		4650.32	473.80
XML-Twig		565.31	11.00
XGrep		688.57	324.25
lzcs-grep (XML)		54.21	250.20
lzcs-grep (LZCS)		3.12	5.46
Todos	$\bar{x}$	861.21	233.13
	σ	1694.76	170.45
Sin XMLStarlet	$\bar{x}$	229.69	193.02
	σ	310.77	146.11

En la Tabla 3 también hemos medido los requisitos de memoria de cada sistema. No reflejamos las medidas detalladas para cada experimento porque, para cada sistema, las diferencias entre ellas son mínimas. Sobre texto plano, el sistema que necesita menos memoria es *XML-Twig*, aunque se debe recordar que sus tiempos de respuesta son muy malos (generalmente los peores). El resto de los sistemas necesitan un orden de magnitud más de memoria, pero *lzcs-grep* y *XSQ* son los que tiene unos requisitos menores. Por su parte, sobre el texto transformado con LZCS, *lzcs-grep* utiliza la mitad de memoria que la requerida por *XML-Twig*. El ahorro de memoria sobre el texto comprimido supera a la razón de compresión: *lzcs-grep* necesita el 2% de memoria cuando se ejecuta sobre el texto transformado.

En la parte inferior de esa misma Tabla también se muestran la media y la desviación estándar de los resultados de todos los sistemas con el fin de proporcionar una visión mejor sobre los tiempos de respuesta y la memoria requerida por cada sistema. No obstante, los resultados obtenidos por *XMLStarlet* no son competitivos y dificultan la comparación con respecto a la media y a la desviación estándar, por eso también se han incluido estos parámetros excluyendo a *XMLStarlet*.

Aunque hemos obtenido el rendimiento de *lzcs-grep* sobre datos XML sin comprimir con el fin de hacernos una idea de lo que pudiera suceder cuando se realicen las búsquedas sobre textos que no sean muy compresibles, es interesante considerar la posibilidad de utilizar *lzcs-grep* exclusivamente sobre textos sin comprimir. En este caso no necesitamos almacenar ningún tipo de información sobre las referencias para su posible uso futuro. Hemos rehecho los experimentos implementando esta optimización, y todos los tiempos de respuesta estaban en torno a los 38–41 segundos, significativamente mejores que los 52–57 segundos que se

obtuvieron cuando se manejaban las referencias, pero todavía son peores que los obtenidos por *XGrep* en sus mejores casos. Nos gustaría destacar que estos tiempos mejorados no quedan lejos de los tiempos de respuesta de *XSQ*, por lo tanto en una situación práctica *lzcs-grep* es una alternativa bastante competitiva. Por otro lado, es interesante comentar que los requisitos de memoria de esta alternativa mejorada para datos sin comprimir son extremadamente bajos, en torno a 4 MB.

6 Conclusiones y Trabajo Futuro

Hemos presentado algoritmos *streaming* de consulta rápidos y eficientes que implementan los *location steps* de XPath más comunes sobre colecciones XML comprimidas. Estos algoritmos están en consonancia con otras propuestas para el proceso *streaming* de consultas [4, 6, 10]. Concretamente, nuestros algoritmos se pueden ejecutar sobre colecciones transformadas con LZCS, en las que los subárboles repetidos están factorizados. Nuestros algoritmos aprovechan está repetición reutilizando el trabajo ya realizado en las subestructuras repetidas. En datos XML muy redundantes (como por ejemplo los documentos tipo formulario con una estructura fija y con poco texto en cada campo), LZCS obtiene unas razones de compresión excelentes (habitualmente por debajo del 10% [1, 2]), y el tiempo de respuesta de nuestros algoritmos disminuye en la misma proporción, siendo imbatible tanto en tiempo como en requisitos de espacio. En datos que no son muy compresibles, la velocidad de nuestros algoritmos no llega a ser tan alta, pero nuestros experimentos muestran que también son competitivos en datos XML sin comprimir. Además, estos tiempos son estables y predecibles. Hemos implementado estos algoritmos en un prototipo denominado *lzcs-grep*, que acepta la notación XPath estándar para la operaciones que implementa (ejes "parent", "child", "ancestor", "descendant"; y el predicado "text matching").

El trabajo futuro pasa por la optimización del código, por la extensión del conjunto de operaciones de XPath soportadas y por la implementación de los *location paths* (actualmente sólo se resuelven *location steps*).

En este artículo sólo hemos considerado algoritmos de *streaming*. Los algoritmos de indexación (por ejemplo [9]) son una alternativa importante cuando hay muchos datos y las consultas son mucho mas frecuentes que las actualizaciones. Las características de LZCS pueden ser aprovechadas por una estructura de tipo índice invertido que almacene las localizaciones de todas las palabras y etiquetas: cuando aparecen subestructuras repetidas, estas repeticiones se pueden representar mediante una codificación diferencial en el índice invertido para ahorrar espacio. Además, también es posible disminuir los tiempos de proceso de las listas de ocurrencias si se reutiliza el trabajo que se realizó en la primera ocurrencia de un segmento repetido de la lista.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Ministerio de Educación y Ciencia como parte del proyecto TIN2006-15071-C03-02 (primer y tercer autores) y por el proyecto Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile (segundo autor).

Referencias

1. J. Adiego, P. de la Fuente, and G. Navarro. Una técnica de compresión para documentos de texto considerando su estructura. In *Proc. IX Jornadas de Ingeniería de Software y Bases de Datos (JISBD 2004)*, pages 399–410, 2004.

2. J. Adiego, G. Navarro, and P. de la Fuente. Lempel-Ziv compression of highly structured documents. *Journal of the American Society for Information Science and Technology (JASIST)*, 58(4):461–478, 2007.
3. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.
4. P. Buneman, M. Grohe, and C. Koch. Path queries on compressed xml. In *VLDB'2003: Proc. of the 29th International Conference on Very Large Data Bases*, pages 141–152. VLDB Endowment, 2003.
5. Y. Chen, S. Davidson, and Y. Zheng. ViteX: A streaming XPath processing system. In *Proc. 21st International Conference on Data Engineering (ICDE)*, pages 1118–1119. IEEE Computer Society, 2005.
6. A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *SIGMOD'03: Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419–430, New York, NY, USA, 2003. ACM.
7. V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *The VLDB Journal*, 14(2):197–210, 2005.
8. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.
9. G. Navarro and R. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *ACM Transactions on Information Systems*, 15(4):400–435, 1997.
10. D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against xml streams. In *ICDE'2003: Proc. of the 19th International Conference on Data Engineering*, page 702, 2003.
11. F. Peng and S. Chawathe. XPath queries on streaming data. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 431–442, 2003.
12. W3 Consortium. XPath 1.0: XML path language. Technical report, World Wide Web Consortium, 1999. `http://www.w3.org/TR/xpath`.
13. W3 Consortium. XSL transformations (XSLT). Technical report, World Wide Web Consortium, 1999. `http://www.w3.org/TR/xslt`.
14. W3 Consortium. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, 2001. `http://www.w3.org/TR/xquery`.
15. W3 Consortium. Extensible Markup Language (XML) 1.0, third edition. Technical report, World Wide Web Consortium, 2004. `http://www.w3.org/TR/xml`.
16. W3 Consortium. XForms 1.0, second edition. Technical report, World Wide Web Consortium, 2006. `http://www.w3.org/TR/xforms`.
17. J. Ziv and A. Lempel. An universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–343, 1977.
18. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, 1978.