UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

COMBINANDO INDEXACIÓN Y COMPRESIÓN EN TEXTO SEMI-ESTRUCTURADO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FELIPE LEOPOLDO SOLOGUREN GUTIÉRREZ

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
RODRIGO PAREDES MORALEDA

SANTIAGO DE CHILE
JUNIO 2009

ESTE TRABAJO HA SIDO FINANCIADO POR EL NÚCLEO MILENIO CENTRO DE INVESTIGACIÓN DE LA WEB, PROYECTO P04-067-F, MIDEPLAN, CHILE, Y POR LOS PROYECTOS FONDECYT 1-050493 Y 1-080019.

Resumen

El almacenamiento digital de la información debe abordar tanto el problema de la incorporación de datos al sistema como su recuperación, y debe hacer un catálogo acorde con las consultas que sobre ellos quiera hacerse. El espacio ocupado para el almacenamiento y el tiempo necesario para ingresar la información, y para recuperarla, dependen directamente de la estructura utilizada en el repositorio. De este modo, cuando nos referimos a información que ya cuenta con un grado de estructuración, la indexación debe ser coherente con la estructura formal de la misma, para favorecer así su consulta.

En el presente estudio se aborda el problema de la compresión estática de información semi-estructurada combinada con una indexación tendiente a soportar un conjunto de consultas sobre los datos, con un fuerte énfasis en el almacenamiento en memoria secundaria.

El ámbito de desarrollo de la aplicación propuesta se enmarca dentro de la documentación XML y su lenguaje de consulta XQuery. El modelo utilizado en la implementación está basado en las propuestas desarrolladas por Baeza-Yates y Navarro en Proximal Nodes. La implementación corresponde a un desarrollo posterior de un procesador destinado a soportar consultas en el lenguaje XPath desarrollado por Manuel Ortega como memoria de Ingeniería.

El desarrollo de la capa de almacenamiento del prototipo actual se enfoca en la resolución de problemas en tres áreas: recuperación del archivo fuente, consulta eficiente sobre la estructura del documento, y búsqueda de texto en lenguaje natural. Las estructuras diseñadas adhieren a técnicas recientes en el área de la compresión y de recuperación de la información en XML.

El resultado de esta memoria es un autoíndice XML con gran desempeño en colecciones de tamaño pequeño y mediano, con capacidad de abordar colecciones de gran tamaño con recursos limitados de memoria principal, y con un gran potencial de adaptación para colecciones en un contexto dinámico. El prototipo presenta un desempeño altamente competitivo con las alternativas existentes en el estado del arte.

Índice general

Índice de figuras

Índice de cuadros

Índice de algoritmos

Capítulo 1

Introducción

El almacenamiento digital de la información debe abordar tanto el problema de la incorporación de datos al sistema como su recuperación, y debe hacer un catálogo acorde con las consultas que sobre ellos quiera hacerse. El espacio ocupado para el almacenamiento y el tiempo necesario para ingresar la información, y para recuperarla, dependen directamente de la estructura utilizada en el repositorio. De este modo, cuando nos referimos a información que ya cuenta con un grado de estructuración, la indexación debe ser coherente con la estructura formal de la misma, para favorecer así su consulta.

En la actualidad existen diversas aplicaciones y estudios tendientes a resolver tanto el problema del espacio, que vendremos a definir como el problema de la compresión, como el problema del tiempo o de la eficiencia en la consulta. Ambos se complementan y apuntan al objetivo principal de una base de datos: almacenar y proveer consultas sobre la información contenida en los datos.

En el caso de la información semi-estructurada, donde se combinan consultas sobre la estructura misma de la documentación con aquellas sobre el contenido, se han desarrollado pocos estudios que se aboquen al almacenamiento sin recurrir al modelo relacional.

Por otro lado, tampoco hay muchos estudios orientados a la consulta de información semi-estructurada comprimida cuando ésta se maneja en memoria secundaria, es decir, cuando el documento o su representación sucinta o compacta no cabe en RAM.

En este estudio se pretende abordar el problema concreto del almacenamiento en memoria

secundaria de información semi-estructurada, en base a una aplicación existente llamada XPN que almacena información en formato estándar XML. La idea es proveer un conjunto de operaciones de consulta sobre los datos que permitan implementar el lenguaje XML Query (XQuery), definido como estándar para la consulta sobre este tipo de documentación. El modelo utilizado para desarrollar dicha implementación es Proximal Nodes [NBY97], y la consulta sobre el contenido está restringida a Lenguaje Natural.

1.1. Motivación

Debido al incremento de información almacenada, intercambiada y presentada haciendo uso de XML, la capacidad de realizar consultas inteligentes sobre fuentes de datos de ese tipo ha ido cobrando importancia. Una de las fortalezas de XML es su flexibilidad en representar diferentes tipos de información proveniente de diversas fuentes. Para explotar esta flexibilidad, un lenguaje de consulta XML debe proveer características para recuperar e interpretar información desde aquélla diversidad de fuentes. Considerando esto, el desarrollo de una base de datos XML debe sortear una serie de dificultades tendientes a resolver problemas de variado tipo.

La integración de compresión e indexación es una excelente técnica para implementar bases de datos porque, aparte de la reducción del espacio de almacenamiento, permite reducir los tiempos de transferencia en disco y en redes de transmisión de datos. Además, si la compresión permite acceso local a los datos, se puede trabajar eficientemente con colecciones de gran tamaño aún cuando no quepan en RAM, y es más fácil sacar partido de la información agregada. Por otro lado, si la compresión es adecuada, se puede hacer búsqueda sobre el texto comprimido, comprimiendo también el patrón de búsqueda, creando así un proceso más eficiente. La indexación, por su parte, orientada al tipo de consultas que se requiera, es una poderosa herramienta de cálculo previo, pues dispone la información de un modo favorable.

Sin embargo, la combinación de compresión e indexación sigue siendo un problema difícil de resolver en la práctica, puesto que los mejores algoritmos de compresión obtienen su

poder ya sea mediante una permutación de la información del fichero original (por ejemplo la transformada de Burrows/Wheeler [BW94]), o ya sea sacando provecho del acceso secuencial completo del fichero (p.e. ZIP [ZL77, ZL78]). Por tanto, obtener altos niveles de compresión y a la vez facilitar el acceso a partes específicas del contenido comprimido no es una tarea fácil, y tiene una alta relevancia en lo que respecta a las bases de datos, sobre todo cuando se trata de ficheros de gran tamaño.

En este contexto, el Centro de Investigación de la Web [1] cuenta entre sus proyectos con el desarrollo de una base de datos de documentos XML nativa (ver sección 2.1.2) con consultas en XQuery llamada XPN, que está basada en una implementación previa (IXPN, ver sección 2.3) que sólo admitía consultas XPath [NO03] y que fue desarrollada como memoria de ingeniería.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo de este trabajo es el desarrollo de un índice eficiente que permita almacenar documentación semi-estructurada en un repositorio, de modo de facilitar su consulta localmente y a la vez reducir el espacio de almacenamiento. Esta combinación de requerimientos en la construcción debe ser eficiente tanto en lo temporal como en lo espacial, es decir, debe considerar el tiempo de creación y el espacio utilizado, pero también proveer la implementación de un conjunto de funcionalidades que permita responder a las consultas de un lenguaje específico.

En términos prácticos, el trabajo se desarrollará sobre una implementación existente llamada XPN, cuyo objetivo es almacenar documentos XML de modo eficiente, y permitir consultas XQuery sobre su repositorio. Para resolver la parte XPath del lenguaje XQuery, es decir para la consulta sobre la estructura de los datos, el modelo subyacente será Proximal

[1] http://www.cwr.cl, Universidad de Chile, Departamento de Ciencias de la Computación. El Centro de Investigación de la Web (CIW) es posible gracias al Programa Iniciativa Científica Milenio, Ministerio de Planificación y Cooperación - Gobierno de Chile http://www.mideplan.cl/milenio.

Nodes, y el dominio de las consultas de contenido será el Lenguaje Natural.

1.2.2. Objetivos Específicos

Con el propósito de lograr una implementación funcional del índice propuesto, es necesario hacer un desarrollo tanto teórico como práctico. Por tanto, los objetivos específicos se dividen en dos áreas:

Objetivos de Diseño

- Obtener un método para almacenar de forma comprimida un documento semi-estructurado que permita hacer consulta local de modo eficiente.
- Obtener un método para consultar eficientemente un documento semi-estructurado de acuerdo al análisis de sus características estructurales.
- Obtener un método para consultar de modo eficiente un documento semi-estructurado que permita recuperar contenido de acuerdo a una búsqueda por palabras.

Objetivos de Implementación

- Implementar un algoritmo de compresión local para lenguaje XML.
- Implementar un índice eficiente para las etiquetas XML.
- Implementar un índice invertido de palabras sobre el contenido del documento XML, capaz de adaptarse a un contexto de Lenguaje Natural.

1.3. Alcances

El alcance de esta memoria está reducido al diseño de la capa de almacenamiento y recuperación de la información para XPN. Esto incluye el diseño e implementación eficiente del almacenamiento del documento incorporado a la base de datos, del índice invertido estructural y del índice invertido en lenguaje natural. Debido al nivel de complejidad observado en

la práctica, el almacenamiento será de tipo estático, no dinámico, es decir, no se considera la modificación del fichero directamente. Si se desea una modificación en un documento de la base de datos, esto implica una reindexación del mismo.

Por otra parte, debido a la extensión del trabajo desarrollado, no se profundizará en los conceptos relativos a los lenguajes de consulta sobre XML, presentándolos como una base previa para la comprensión de esta propuesta.

Una buena descripción del lenguaje XPath 1.0 se encuentra en la lectura de [NO02], que es un antecedente directo de este trabajo. Para una descripción detallada del lenguaje XQuery y de XPath 2.0, existe abundante literatura en Internet y particularmente en el sitio del World Wide Web Consortium[2].

[2]Ver http://www.w3.org/XML/Query/ y http://www.w3.org/TR/xpath respectivamente.

Capítulo 2

Antecedentes

Para comprender el desarrollo del trabajo realizado es necesario profundizar en una serie de conceptos previos. En la sección 2.1 se revisan los conceptos básicos utilizados y en la sección 2.3 se describe el trabajo realizado previamente en la línea de desarrollo de la aplicación XPN. La sección 2 cubre el trabajo existente en torno a compresión de texto, algoritmos, técnicas y estrategias para el almacenamiento y consulta de documentos en memoria principal y secundaria, que serán la base para abordar el problema que se quiere resolver.

Por último, la sección 2.4 describe las alternativas existentes.

2.1. Conceptos Básicos

A continuación se detalla una serie de definiciones necesarias para comprender los términos utilizados en esta memoria. Se abordará el concepto de información estructurada y el concepto de lenguaje XML, abordando la estructura y los lenguajes de consulta asociados, porque es el dominio de la aplicación práctica de esta memoria.

Posteriormente, se explica el concepto de offset para comprender el manejo de ficheros, la estructura de árbol necesaria para entender las características estructurales de un documento XML, y por último, el concepto de lenguaje natural que determina el manejo del contenido almacenado en las colecciones utilizadas.

2.1.1. Información Estructurada

Es un tipo de información que incluye un contenido (palabras, imágenes, etc.) y también la indicación de qué rol juega este último (por ejemplo, el contenido en la cabecera de una sección tiene un significado diferente del de un pie de página, que a su vez difiere de la leyenda de una figura o del contenido de una tabla de datos). Casi todos los documentos tienen alguna estructura.

La información estructurada es aquella que define una estructura para su contenido.

Lenguaje de marcado

Es un mecanismo para identificar estructura en un documento mediante la inclusión de marcas o etiquetas al contenido para delimitarlo.

2.1.2. XML

XML corresponde a la abreviación de **eXtensible Markup Language**. Es un lenguaje de marcado versátil, capaz de etiquetar contenidos de diversas fuentes de información incluyendo documentos estructurados y semi-estructurados, bases de datos relacionales, y repositorios de objetos. La especificación XML define una forma estándar de agregar marcado a los documentos.

XML ha jugado un importantísimo rol en el intercambio de información de variado tipo en la Web y en otros medios electrónicos. Se caracteriza básicamente por un marcado jerárquico, desarrollado a través de etiquetas de apertura y cierre para delimitar segmentos de contenido semi-estructurado. Además, cuenta con la posibilidad de agregar atributos a los elementos. La idea del etiquetado es que sea lo más sencillo posible, es decir, que las etiquetas faciliten la comprensión de la estructura que el marcado determina. A esta característica se la denomina *human readable*.

La única regla básica que debe cumplir un documento XML es la condición de `bien formado`, vale decir, que todo segmento debe ser delimitado por un par de etiquetas de apertura y cierre, y que la jerarquía debe respetarse. Esto último quiere decir que los segmentos

definidos por las etiquetas deben ser anidados. Si un segmento empieza después que otro, debe terminar antes que el primero.

Existe una excepción a la primera regla que corresponde a un elemento estructural cuyo contenido es vacío. En ese caso, la etiqueta puede denotarse de modo abreviado en una sola que sintetiza a la de apertura y cierre.

En el cuadro 2.1 se puede observar un ejemplo de documento XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is a comment. -->
<a>
  <b id="a">1</b>
  <b>2</b>
  <b id="a"></b>
</a>
```

Cuadro 2.1: Ejemplo de Documento XML.

Una descripción más detallada del estándar se puede consultar en http://www.w3.org/TR/xml/.

XQuery

Es un lenguaje de consultas que usa la estructura XML inteligentemente para expresar consultas a través de muchos tipos de datos almacenados físicamente en XML o visibles vía un *middleware*. XQuery opera en la estructura lógica abstracta de un documento XML, más allá de su sintaxis superficial. Esta estructura lógica se conoce como *data model* [1].

En el cuadro 2.2 se puede observar un ejemplo de consulta para el documento del cuadro 2.1 denominado "data.xml".

XQuery 1.0 es una Recomendación del W3C desde el 23 de enero de 2007. Su especificación se encuentra disponible en http://www.w3.org/TR/xquery/.

[1] XDM: XQuery/XPath Data Model, W3C, http://www.w3.org/TR/xquery/#datamodel.

Código XQuery:

```
for $i in fn:doc('data.xml')//b
where $i/@id = "a"
return <a>{$i/text()}</a>
```

Resultado:

```
<a>1</a>,
<a/>
```

Cuadro 2.2: Ejemplo de Consulta XQuery.

XPath

Es la definición de un lenguaje básico para hacer referencia a trozos de documentos XML. Se utiliza en lenguajes de más alto nivel como XSL, XQuery, XPointer y otros. Existen numerosas implementaciones prácticas.

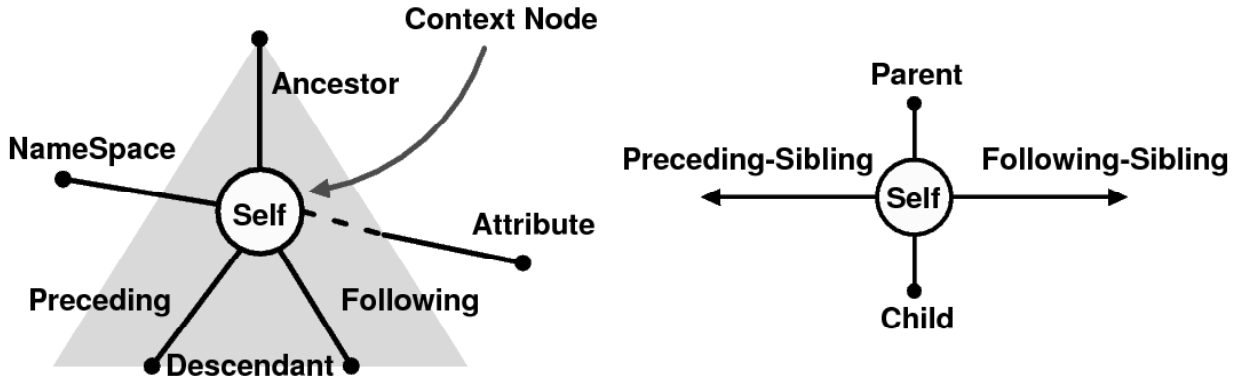

Figura 2.1: Ejes XPath.

La siguiente sentencia XPath tiene por objetivo mostrar un ejemplo de cómo recuperar todos los nodos cuya etiqueta es b en cualquier nivel de un documento XML llamado "data.xml":

$$fn : doc('data.xml')//b$$

En la figura 2.1 se muestra la descomposición de los elementos de un documento XML en relación con los ejes XPath. Cada eje establece una relación estructural del nodo consigo mismo y con los demás.

Para una descripción más detallada se puede consultar [NO02] o la especificación recomendada por el Consorcio en http://www.w3.org/TR/xpath20.

Es importante destacar que existen dos enfoques para implementar XPath: *Navigational* (aquellos donde se recorre el árbol completo buscando satisfacer las restricciones), y *Join-based* (aquellos donde se seleccionan conjuntos de nodos en base a sus etiquetas que luego son intersectados de acuerdo a la estructura) [ZKÖ04].

Base de Datos XML nativa (native XML database, NXD)

Las bases de datos nativas definen un modelo lógico para el documento XML, pero la principal característica que brindan es la capacidad de obtener los resultados de las consultas en formato XML; es por ello que dichas bases de datos pertenecen a la categoría de *XML-enabled databases* [2].

2.1.3. Offset

En informática, un offset dentro de un arreglo u otra estructura de datos (p.e. un fichero) es un entero que indica la distancia (desplazamiento) desde el inicio del objeto hasta un punto o elemento dado, presumiblemente dentro del mismo objeto. El concepto de distancia es solamente válido si todos los elementos del objeto son del mismo tamaño (típicamente dados en bytes o palabras) [3].

2.1.4. Árboles

Los árboles son una estructura fundamental en computación. Son usados en casi todos los aspectos de modelamiento y representación para computación explícita, como son búsqueda de llaves, mantenimiento de directorios, representación de *parsing* o trazas de ejecución, entre otros. Uno de los usos intensivos más recientes de árboles se da en XML.

El almacenamiento explícito de árboles, con un puntero por hijo más información auxiliar (p.e. la etiqueta), puede llegar a ser muy costo si se considera acceso a la vecindad del nodo para permitir navegación; se necesitan al menos 16 bytes para almacenar un puntero al padre, otro al primer hijo, uno al siguiente hermano y uno hacia la información auxiliar. La Teoría

[2] http://es.wikipedia.org/wiki/Bases_de_datos_nativas_xml.
[3] http://es.wikipedia.org/wiki/Offset_%28inform%C3%A1tica%29.

de la Información, provee una cota inferior al espacio necesario para almacenar un árbol cualquiera con forma y grado arbitrarios, a partir de un argumento simple de conteo: se necesitan al menos $\log |U|$ bits para distinguir dos objetos en un universo U.

Existen tres grandes clases de árboles; los **árboles ordinales**, que son árboles no etiquetados en los cuales los nodos hijos son **ordenados** de izquierda a derecha, los **árboles cardinales k-arios**, que son árboles etiquetados con sus nodos en base a símbolos de un alfabeto dado Σ, donde $k = |\Sigma|$ y el grado es a lo más k (porque sus enlaces están etiquetados con una etiqueta distinta), y por último, los **árboles etiquetados** que son el dominio de los árboles XML.

Los **árboles etiquetados** son árboles ordinales en los cuales cada nodo debe tener asignada una etiqueta con símbolos pertenecientes a un alfabeto Σ, y pueden tener más de un hijo con igual etiqueta. Esto significa que el grado de los nodos no está acotado, y el mismo camino etiquetado puede repetirse muchas veces en el árbol en distintos niveles.

Los árboles multi-etiquetados permiten más de una etiqueta por nodo.

Su costo de almacenamiento de acuerdo a la teoría de la información corresponde al análisis obtenido de separar la estructura del árbol del etiquetado. El número de árboles ordinales es

$$C_t = \binom{2t}{t}/(t+1)$$

que induce una cota inferior de $\log(C_t) = 2t - \Theta(\log(t))$ bits. Luego, para el caso etiquetado, la cota agrega el costo en bits de almacenar las etiquetas:

$$\log C_t + t \log |\Sigma| = t(\log |\Sigma| + 2) - \Theta(\log(t))$$

Por otro lado, las operaciones a soportar incluyen operaciones básicas como consultar por el padre de un nodo u, el i-ésimo hijo, el i-ésimo hijo cuya etiqueta es c, o el grado del nodo, entre otras. También pueden hacerse operaciones más elaboradas como el i-ésimo ancestro, la profundidad o nivel de u, el ancestro común más cercano de un par de nodos (Lowest

Common Ancestor - LCA), el i-ésimo nodo de acuerdo a determinado orden en el árbol, etc.

También es interesante definir consultas por caminos, es decir, por patrones de símbolos asociados a las etiquetas que se expresan de acuerdo a las secuencias que se forman, continuas o discontinuas y en distintos niveles, recorriendo nodos del árbol de acuerdo al orden definido en el documento [FR08].

Pre Orden y Post Orden

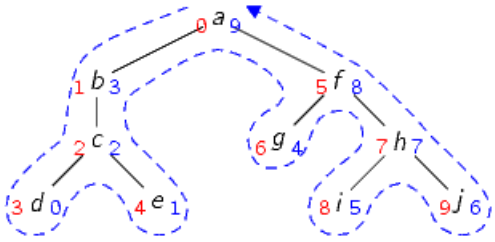

Figura 2.2: Recorrido de un árbol.

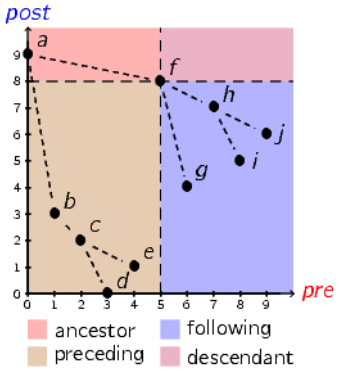

Figura 2.3: El Plano Pre-Post.

En la figura 2.2 se observa el recorrido sobre un árbol dado y la numeración de sus nodos de acuerdo a su posición de **pre** orden (color rojo) y **post** orden (color azul). El par de valores asignado a cada nodo XML define un plano que determina las cuatro regiones para los ejes XPath (en la figura 2.3 se observa el plano para el nodo de contexto f del árbol ejemplo[4]).

El pre orden es también llamado **orden de documento**.

Notación Pre-Size-Level

Esta notación está basada en el esquema planteado por Li y Moon [LM01]. La idea es representar cada nodo del árbol XML mediante una tripleta que contiene los valores de pre orden (Pre), tamaño ($Size$, número de descendientes más el propio nodo), y nivel ($Level$, distancia desde el nodo raíz). Con esta representación se pueden resolver todas las relaciones estructurales entre nodos relevantes para XPath:

[4]Imagen extraida desde http://www.pathfinder-xquery.org/.

Un nodo p es ascendiente de otro nodo q si y sólo si:

$$Pre_p \leq Pre_q \leq Pre_p + Size_p \tag{2.1}$$

lo cual es equivalente a expresar en términos de segmentos que p incluye a q, con $To_q = Pre_q + Size_q$:

$$Pre_p \leq Pre_q \wedge To_q \leq To_p \tag{2.2}$$

La determinación de la relación Padre/Hijo se verifica con:

$$Pre_p \leq Pre_q \wedge To_q \leq To_p \wedge Level_q = Level_p + 1 \tag{2.3}$$

si p es el nodo padre de q.

Un nodo p está antes de otro nodo q (relación preceding/following) si y sólo si:

$$To_p \leq Pre_q \tag{2.4}$$

Un nodo p es un hermano precedente de otro nodo q si y sólo si:

$$Pre_r < To_p \leq Pre_q < To_r \wedge Level_p = Level_q = Level_r + 1 \tag{2.5}$$

donde r es el nodo padre de p y q.

2.1.5. Lenguaje Natural

Es el lenguaje hablado y/o escrito por humanos para propósitos generales de comunicación, para distinguirlo de otros como puede ser una lengua construida, los lenguajes de programación o los lenguajes usados en el estudio de la lógica formal, especialmente la lógica matemática [5].

El lenguaje natural se trabaja en el contexto de un idioma, es decir, se define a partir de un vocabulario específico que determina el conjunto de palabras con el cual se trabaja. De este

[5] http://es.wikipedia.org/wiki/Lenguaje_natural.

modo, las palabras vacías (stopwords, ver sección 2.1.5) que se definen en ese conjunto dicen relación no sólo con el significado sintáctico y semántico que aportan, sino que determina también las cadenas de caracteres con las cuales son creadas. Cada idioma tiene sus propias reglas sintácticas y semánticas, y estas características influyen en la forma cómo se distribuyen las palabras en los textos hablados o escritos del lenguaje natural.

Definiciones

Stopwords En castellano *palabras vacías*, es el nombre que reciben las palabras sin significado, como artículos, pronombres, preposiciones, etc., y que son filtradas antes o después del procesamiento de datos en lenguaje natural (texto). La categorización de stopwords está controlada por introducción humana y no automática [6].

Separadores Es una secuencia de caracteres cuya función es separar palabras dentro de un contexto de lenguaje natural. Lo constituyen los espacios, símbolos de puntuación, y eventualmente las stopwords.

Stemming Es un método para reducir una palabra a su raíz denominada *stem* o lema. Hay algunos algoritmos de stemming que ayudan en sistemas de recuperación de información. El stemming aumenta el *recall*, que es una medida del porcentaje de documentos relevantes que se pueden encontrar con una consulta. Por ejemplo una consulta sobre `bibliotecas` también encuentra documentos en los que sólo aparezca `bibliotecario` porque el stem de las dos palabras es el mismo: `bibliotec` [7].

Lenguaje Natural

[6] http://en.wikipedia.org/wiki/Stop_word.
[7] http://es.wikipedia.org/wiki/Stemming.

2.2. Revisión Bibliográfica

Como se ha señalado previamente, un índice para XPN necesita responder a una serie de consultas tendientes a proveer navegación en el documento almacenado que permita satisfacer las necesidades que el lenguaje XQuery requiere. En este sentido, los problemas abordados se ubican en áreas de investigación de gran actividad y relevancia, como lo son Compresión, Recuperación de la Información, Lenguaje Natural, Estructuras Compactas y XML, entre otras.

2.2.1. Compresión de Texto

La compresión de texto consiste en representar una secuencia de caracteres usando menos bits que su representación original. El texto es visto como una secuencia de *símbolos fuente* (caracteres, palabras, etc.), y el objetivo principal es la reducción del espacio utilizado. Esto no es sólo importante en el almacenamiento de la información, sino también para mejorar la transferencia a memoria secundaria o para bajar los tiempos de transferencia en una red computacional [BWC89].

La compresión de texto se divide usualmente en dos tipos. La compresión **estadística** está basada en la estimación de las probabilidades de los símbolos fuentes y les asigna códigos de acuerdo a esas probabilidades. Los métodos de **diccionario** en cambio, consisten en reemplazar subsecuencias del texto por identificadores, de modo de explotar las repeticiones en el texto.

La compresión estadística se divide conceptualmente en dos tareas. El modelamiento interpreta el texto como una secuencia de símbolos y le asigna probabilidades, a veces de acuerdo a los símbolos contiguos. El modelamiento de orden cero asigna probabilidades a los símbolos interpretándolos de modo aislado, mientras que el modelamiento de orden k asigna sus probabilidades como una función de los k símbolos que los preceden.

La codificación asigna a cada símbolo fuente una secuencia de símbolos objetivo (su código), basada en las probabilidades dadas por el modelo. La salida del compresor es la secuencia de símbolos objetivo dados por el codificador.

La compresión es **semi-estática** cuando se obtiene un sólo modelo para el texto completo antes de que la codificación comience, por lo tanto todas las ocurrencias del mismo símbolo fuente (en el mismo contexto) tendrán asignadas el mismo código. La compresión **adaptativa** alterna las tareas de modelamiento y codificación, por ejemplo, cada símbolo se codifica usando el modelo actual, que luego es actualizado para considerar esta aparición, lo cual a su vez puede modificar los códigos de compresión. Por lo tanto diferentes ocurrencias del mismo símbolo fuente pueden tener asignados distintos códigos.

La compresión semi-estática requiere dos pasadas sobre el texto; una para el análisis estadístico necesario para definir los códigos, y otra para escribirlos. El almacenamiento del modelo se hace junto con el fichero comprimido, el cual puede descomprimirse a partir de cualquier posición. Por otro lado, la compresión adaptativa, no puede iniciar la descompresión en posiciones arbitrarias del fichero, porque todo el texto previo debe ser procesado para aprender el modelo que permite descomprimir el texto que sigue. Sin embargo, ésto puede ser parcialmente resuelto con un índice que almacene muestras de información a intervalos arbitrarios, de modo de poder reconstruir el modelo en el punto que se desee a partir de la muestra previa [BFNP07, ANd07b].

Codificación y Compresores

Re-Pair Es un compresor basado en diccionario que permite descompresión local rápida. Consiste en buscar repetidamente el par de símbolos más frecuente en una secuencia de enteros, e ir reemplazándolos por un nuevo símbolo, hasta que no sea conveniente efectuar más reemplazos [LM00].

Debido a que el algoritmo de tiempo lineal original utiliza demasiada memoria, Claude y Navarro [CN07] han desarrollado una aproximación que ocupa tanta memoria como se desee por sobre la del texto. Permite un compromiso entre memoria RAM usada y calidad de la compresión, y es capaz de trabajar sin dificultades en memoria secundaria debido a su patrón de acceso secuencial.

Re-Pair es un codificador de orden k [NR08].

Huffman La codificación de Huffman está diseñada para compresión estadística. Asigna un código de largo variable a cada símbolo fuente, tratando de dar códigos más cortos a símbolos más probables. El algoritmo de Huffman garantiza que la asignación de código minimiza el largo del fichero comprimido bajo las probabilidades dadas por el modelo [Huf52]. Además, el código de Huffman es libre de prefijos, es decir, la secuencia de bits que representa a un símbolo en particular nunca es prefijo de la secuencia de bits de un símbolo distinto.

Huffman es un compresor de orden cero.

La codificación de Huffman no tiene buen desempeño en la compresión de lenguaje natural (65 %), por lo cual se ha desarrollado un enfoque basado en palabras (word-based), que permite lograr resultados muchísimo mejores (25 % [BFNP07]). En este enfoque se interpreta el texto como una secuencia de palabras y separadores.

Plain Huffman Code (PHC) Es un código de Huffman basado en palabras y orientado al byte. Su objetivo es mejorar el desempeño en la descompresión y eliminar las dificultades de manipulación de bits del Huffman clásico orientado a palabras [BFNP07], usando un árbol de Huffman de aridad 256 (representación de un byte), de modo que las palabras de código sean secuencias de bytes. Esta representación mejora los tiempos de acceso y descompresión en un 30 % a cambio de empeorar el grado de compresión a un 30 % [BFNP07].

Tagged Huffman Code (THC) Es un código de Huffman que sigue la misma línea del anterior, pero permite fácil detección del alineamiento de la palabra codificada debido a que funciona con 1 bit para marcar el primer byte de la palabra. De este modo se puede acceder aleatoriamente al texto comprimido facilitando considerablemente la identificación del comienzo de la codificación de cada palabra de código, y por otro lado facilita también la búsqueda de patrones. Esto último se traduce en mayor eficiencia en la detección de los códigos, pues sólo se requiere el bit de marcado para determinar códigos de distinta longitud de palabra. Al comprimir el patrón de búsqueda, no pueden ocurrir falsos positivos como en el PHC.

El deterioro en la compresión producido por la disponibilidad de sólo 7 bits en cada byte (el

grado de compresión llega a 35 %), se compensa por la velocidad alcanzada en descompresión local y búsqueda [BFNP07].

Dense Codes Son códigos que al ser ordenados no contienen códigos consecutivos no utilizados. La idea es optimizar la representación de los códigos permitiendo utilizar un rango de valores completo. El Código de Huffman Etiquetado (THC) por ejemplo, por la inclusión de un bit de marcado, y por sus características de largo variable sujeto a frecuencia del código fuente (propias de Huffman), no utiliza todas las codificaciones posibles en el rango que definen sus códigos desde el más corto hasta el más largo. Siguiendo la filosofía de priorizar los códigos asignando los de menor valor a las ocurrencias más frecuentes, se puede ocupar mejor el rango definido por la secuencia de bytes, manteniendo la idea de comprometer un bit para representar rangos arbitrarios [BFNP07].

End-Tagged Dense Code Es un código denso surgido de un pequeño cambio al Código de Huffman Etiquetado que consiste en intercambiar la marca desde el primer byte al último de la palabra de código, logrando con ello densificar el código manteniendo la propiedad de ser libre de prefijos. Además mantiene las características de velocidad en compresión, descompresión y búsqueda, pero mejorando la calidad de la compresión (superando en más de 8 % al THC, y sólo alrededor de 2.5 % bajo PHC) [BFNP07].

(s,c)-Dense Code Es un código denso que representa el caso general de códigos basados en secuencias de bytes donde se marca el último código de la palabra con un bit. La idea es definir dos conjuntos de bytes: *stoppers* y *continuers*. Los primeros están destinados al último byte y los segundos deben ser siempre seguidos de otro (de cualquier tipo) para conformar códigos largos. Los stoppers que no están precedidos por continuers están destinados a representar ocurrencias más frecuentes. De este modo, se elige el número de stoppers y de continuers cuya proporción sea óptima; a mayor número de stoppers, mayor cantidad de códigos frecuentes de tamaño 1 byte, pero mayor probabilidad de que el resto necesite más de dos bytes para ser representado. A la inversa ocurre lo mismo, pero hasta un límite donde la mayoría puede

quedar representado por códigos de tamaño dos bytes. En ambos casos, el óptimo depende de la cantidad de códigos que se necesite representar. En general, existe una combinación que corresponde a un mínimo global, cuyo cálculo se realiza en función de la distribución de frecuencias obtenida. Además los documentos existentes en la práctica contienen vocabularios bastante menores que lo que se puede representar óptimamente con palabras de hasta tres bytes [BFNP07].

Restricted Prefix Byte Codes Es un código orientado a bytes. Para el caso típico de 8 bits, la idea es definir en el primer byte la distribución de palabras de 1, 2, 3 y 4 bytes. Es decir se divide el rango total (256 valores posibles) en 4 grupos, cada uno de los cuales corresponde a palabras de distinto tamaño. Este enfoque permite distribuir mejor los códigos asociados a cada ocurrencia del diccionario a comprimir porque se ajusta mejor a la distribución de probabilidad de los símbolos presentes en la secuencia a comprimir. Sin embargo, no permite descompresión local porque se producen problemas de sincronización que impiden identificar el primer byte de cada palabra de código. La búsqueda secuencial también requiere de esta sincronización y no es tan rápida como la desarrollada sobre Dense Codes. Sin embargo, supera a la búsqueda sobre PHC [CM05, BFNP08].

Compresión Local

Por compresión local se entenderá aquella que permite descompresión parcial con acceso aleatorio y preferentemente en memoria contigua, de modo de reducir tanto la cantidad de accesos a disco necesarios para recuperar un trozo dado como el tamaño mínimo necesario para reconstruirlo. Esto puede significar usar enfoques adaptativos de compresión siempre y cuando sean acotados a bloques de memoria de tamaño fijo calculados convenientemente, de modo que el extraer trozos de código más pequeños que un bloque implique descomprimirlo completo (limitando así la localidad), y extraer trozos más grandes que un bloque implique descomprimir varios y concatenarlos si es necesario (agregando un costo extra en la reconstrucción).

Otra característica muy relevante y deseable de los algoritmos de compresión con acceso

local, es que permitan hacer búsqueda directa sobre el texto comprimido, facilitando con ello la tarea de búsqueda sobre el texto, mediante comprimir el patrón de búsqueda [BFNP08, dNZBY98].

El algoritmo de compresión denominado Re-Pair ([LM00]) posee buenas características para descomprimir rápida y localmente. Puede ser utilizado para indexar tanto el texto simple del documento (el contenido categorizado por las marcas o etiquetas), como aquél que corresponde a la estructura (las marcas, etiquetas y atributos), permitiendo con ello su navegación. Además, debido a su naturaleza, podría capturar la redundancia presente en mucha documentación XML que está sujeta a esquemas definidos. Sin embargo, debido a que el algoritmo Re-Pair original hace uso intensivo de la memoria, puede ser necesario recurrir a una versión modificada [CN07] para la indexación.

Otro enfoque conveniente para comprimir secuencias es el de los Dense Codes, que también permite descompresión local aún cuando no obtiene los niveles de compresión de Re-Pair aplicado sobre la secuencia de palabras. Es de construcción bastante más rápida que este último y su diccionario es más pequeño, por lo cual la carga del modelo en RAM, visualizado como costo fijo, se traduce en mayor eficiencia para recuperar trozos arbitrarios de código comprimido de tamaño pequeño. Las diferencias de desempeño observadas entre ETDC y SCDC permiten desarrollar un *trade-off* entre velocidad de codificación/decodificación y calidad de la compresión [BFNP07].

Autoíndice

Autoíndice (self-index) es un índice construido de tal forma que permite obtener el contenido indexado sin recurrir al texto original, permitiendo así reemplazarlo.

Dado que en muchos casos logra una reducción del espacio utilizado, es una potente herramienta en la combinación de compresión e indexación.

2.2.2. Intersección de Secuencias Ordenadas

La intersección de secuencias ordenadas se ha utilizado ampliamente para resolver consultas combinadas sobre índices invertidos.

El ejemplo más común es el tipo de búsqueda que realizan los buscadores Web, en los cuales una lista de palabras debe reportar los documentos más relevantes que las contienen.

Dejando de lado el complejo procedimiento de ranking que aplican los buscadores hoy en día, el problema consiste en detectar en qué documentos se encuentran las palabras presentes en la lista que constituye la consulta del usuario. Considerando que para cada palabra se posee un índice invertido que detalla la lista de documentos que las contienen, la solución se obtiene intersectando todas las listas. La representación de cada ocurrencia en las listas es una secuencia de identificadores de documento.

En este sentido, existen variados enfoques para resolver eficientemente la intersección de secuencias ordenadas desde un punto de vista adaptativo [DLOM00, BYS05, BLOL06], es decir, ajustándose a las distribuciones de la entrada, para responder del modo más apropiado.

Por las características que impone el modelo Proximal Nodes, cuyas operaciones consisten precisamente en la intersección de secuencias de nodos de resultados intermedios, la aplicación de estos algoritmos tiene un impacto directo en la recuperación de los nodos estructurales del documento XML.

Sin embargo, el algoritmo de doble búsqueda binaria `Intersect` de [BYS05] no considera que el resultado deba quedar también ordenado pues la secuencia de resultados no mantiene el orden, lo cual agrega una complejidad extra en la implementación práctica del caso que nos compete, porque se debe ordenar la salida y su complejidad dependerá del tamaño. Por otro lado, las mejoras obtenidas por abordar la intersección mediante búsqueda binaria, interpolación o búsqueda exponencial planteados por Demaine et al. [DLOM01], sólo se pueden desarrollar si existe acceso aleatorio en el almacenamiento de las listas.

En este sentido, en una reciente publicación de Culpepper y Moffat [CM07], se ha observado que el acceso secuencial, con una estructura jerárquica para recorrer eficientemente las listas invertidas representadas como arreglos, tiene mejor desempeño si en ello está in-

volucrado el acceso a memoria secundaria, como es el caso de una base de datos. El mejor resultado obtenido en el estudio corresponde a una representación mediante bitmap de la lista invertida, indicando con un bit las posiciones de la palabra a lo largo de la secuencia. Sin embargo, esto supone que el vocabulario de las consultas es reducido para evitar usar un bitmap de largo n por cada palabra indexada. Esta condición no se cumple en una base de datos general, pero sí puede serlo en nuestro caso donde el vocabulario de etiquetas es asintóticamente muchísimo menor que la cantidad de nodos y el vocabulario de palabras del documento.

Peter Sanders y Frederik Transier [ST07] han propuesto resolver eficientemente la intersección de listas de enteros mediante una búsqueda secuencial combinada con una estructura jerárquica. La idea es buscar secuencialmente en el nivel más alto hasta asegurarse que el elemento buscado estará en un segmento determinado cuyos elementos contienen todos un prefijo binario común. De este modo, distribuciones no homogéneas de elementos tenderán a agruparse en torno a uno u otro segmento en la medida que compartan el prefijo.

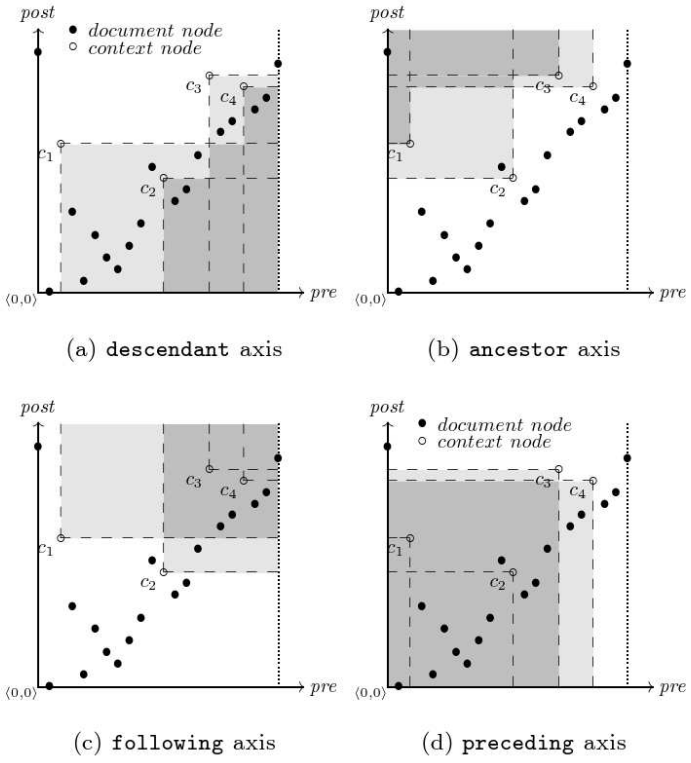

Figura 2.4: Regiones intersectables.

2.2.3. Staircase Join

Staircase Join [GvT03] es una técnica que fue desarrollada para mejorar el desempeño de la evaluación XPath sobre una representación del documento XML en el modelo relacional.

Sin embargo, su lógica puede aplicarse a cualquier representación que pueda sacar provecho de los resultados intermedios para adaptarse en la obtención de resultados posteriores.

La estrategia se desarrolla en base a operaciones sobre enteros ($+, \leq$ p.e.), y tiene como objetivo reducir las comparaciones estructurales inútiles entre nodos intermedios en la evaluación de una expresión XPath. Esta técnica es una adaptación de las técnicas de intersección al problema específico que plantea XPath, donde las relaciones entre los conjuntos obedecen a una estructura diferente de dos dimensiones.

A partir de la identificación del plano de ubicación de los nodos en base a sus valores de pre y post (i.e. posiciones de comienzo de nodo y de término respectivamente), se puede hacer *skipping* de nodos intermedios de acuerdo al tipo de consulta XPath que se quiera realizar (ver los nodos de contexto c_i en la figura 2.4[8]).

Las consultas XPath traducidas al modelo relacional obtienen un desempeño considerablemente superior gracias al Staircase Join introducido al núcleo del motor de la base de datos relacional. Esto se obtiene con una estrategia adaptativa muy similar a la doble búsqueda binaria de Baeza-Yates ([BYS05]), reduciendo significativamente las secuencias de resultados intermedios y el número de comparaciones.

2.2.4. Árboles Etiquetados y Secuencias de Paréntesis Balanceados

Los modelos existentes para árboles en general soportan las consultas más importantes en tiempo constante: búsqueda del nodo padre de un nodo, búsqueda del i-ésimo hijo, obtención del grado del nodo, incluso con restricciones sobre las etiquetas en el caso de árboles etiquetados [FR08]. Sin embargo, no consideran el contenido de los nodos pues sacan provecho de la distribución contigua de los bits para lograr la navegación eficiente por la estructura. Por esta razón, el almacenamiento de los nodos de texto de los documentos XML se debe hacer mediante una referencia implícita a dicho contenido, sea éste comprimido o no.

Mediante un enfoque diferente, Ferragina et al. han propuesto la XBWT [FLMM07] que, basada en la transformada de Burrows/Wheeler, resuelve muy bien las consultas XPath que

[8] Imagen extraída de [GvT03].

se puedan realizar sobre un documento XML cuando se combinan los contextos estructurales y de texto. Sin embargo, la XBWT tiene un serio problema cuando se trata de acceder al disco para recuperar los resultados. Cuando éstos sólo son hojas del árbol XML no hay problemas, pero recuperar un subárbol completo ya no es tan sencillo, pues el acceso a disco empieza a ser aleatorio debido a la permutación que produce la BWT sobre el documento, y por tanto la serialización se hace muy ineficiente. Si la estructura debe encontrarse en RAM para funcionar, el costo fijo que debe enfrentar cualquier consulta por muy pequeña que sea es proporcional a la carga del documento comprimido completo. Si, en cambio, el contenido correspondiente a los nodos de texto se encuentra en disco, la reconstrucción de los subárboles durante la serialización desarrolla acceso aleatorio no contiguo, ya que la distribución de bloques que plantea la solución los agrupa por contexto y no por orden de documento. Además, hay un particular conjunto de relaciones estructurales que interesa de manera especial y que corresponde a la relación de inclusión de un segmento (u nodo) en otro (correspondiente al tipo de consultas XPath de forma $a//b$). Este tipo de relaciones no se resuelve de modo eficiente en dicha estructura.

Las soluciones que abordan el problema de la estructura del árbol resuelven la navegación sobre la estructura de modo eficiente, pero sin considerar el etiquetado ni el contenido. La XBWT, en cambio, resuelve el problema de los sub-caminos al abordarlos como sufijos. Sin embargo, no todos los caminos XPath son de esa forma, en particular, los que expresan inclusión. En resumen, no existe una estructura que apoye eficientemente la consulta XPath de modo completo.

Estructuras Compactas para Secuencias Arbitrarias

Las secuencias de símbolos pertenecientes a un alfabeto dado son la forma más sencilla, efectiva y eficiente para representar un texto. En la implementación de estructuras compactas interesa poder representar dichas secuencias del modo más reducido posible, dotando a la representación de la navegabilidad básica para trabajar con ella. Es por esta razón que se ha definido un par de operaciones básicas sobre secuencias arbitrarias de símbolos, que sirven

como excelente base para un sinnúmero de aplicaciones prácticas en el dominio de éstas estructuras.

Rank es una función básica sobre una secuencia arbitraria, que permite obtener el número de ocurrencias de un símbolo determinado que existen desde el principio de la secuencia hasta una posición dada como parámetro. Se escribe de la siguiente forma:

$$rank_S(c, i)$$

donde c es el símbolo, i la posición dada y S la secuencia.

Select es una función básica cuyo objetivo es obtener la posición de la i-ésima ocurrencia de un símbolo determinado desde el principio de una secuencia dada. Se escribe de la siguiente forma:

$$select_S(c, i)$$

donde c es el símbolo, i el índice de la ocurrencia buscada y S la secuencia.

Existen numerosos estudios tendientes a proveer implementaciones de Rank y Select sobre secuencias arbitrarias de símbolos, eficientes en tiempo y espacio [Mun96, GGV03, GMR06, CN08]. Varias de ellas se enfocan al caso particular del alfabeto binario, las cuales se relacionan muy fuertemente con representaciones de árboles mediante paréntesis balanceados [Jac89, MR01, MRS01, BDM+05, Sad08]. Otras se relacionan con los árboles k-arios [MRR98, RRR02, Arr08], y varias de ellas también, al caso de los árboles rotulados, que combinan la representación de la forma estructural de los árboles, con la distribución de las etiquetas en ella [FLMM07, BHMR07, FR08] .

Sin embargo, Rank y Select sirven también para un sinnúmero de aplicaciones sobre secuencias de símbolos, como por ejemplo los índices invertidos [BFLN08].

2.2.5. Suma de Prefijos

Dada una secuencia de números enteros positivos $x = (x_1, x_2, ...x_n)$, tal que $\sum_{i=1}^{n} x_i = m$, se desea soportar la operación $sum(x, j)$ que retorne $\sum_{i=1}^{j} x_i$.

La idea es que la implementación no ocupe demasiado espacio (se sabe que el tamaño depende de la distribución de los x_i y del valor total m [DRR07]) y que tenga un tiempo de respuesta constante o muy bajo.

Una cota teórica para el espacio ocupado está dada por la fórmula:

$$B(m, n) = \lceil \log_2 \binom{m-1}{n-1} \rceil$$

proveniente de modelar la codificación necesaria para diferenciar cualquier secuencia x cuyos n elementos suman m, es decir, el número de bits necesarios para diferenciar todas las combinaciones posibles de dividir m en n valores enteros positivos.

O'Neil Delpratt et al. [DRR07] han demostrado que la secuencia puede ser representada en $n \log_2(m/n) + O(n)$ bits (i.e., alcanzando la cota) de modo que $sum(x, i)$ sea computada en tiempo $O(1)$.

La suma parcial también se puede lograr almacenando la secuencia de valores parciales en unario, y se consulta mediante composición de llamadas a Rank y Select. La idea es visualizar la suma como una secuencia de ceros. Cada número de la suma se representa con tantos ceros como sea su valor. Al final de su secuencia se escribe un uno. Esto permite representar cualquier valor no negativo. En total la secuencia tiene m ceros y n unos. Entonces, la suma parcial hasta el elemento r es:

$$s = select_B(1, r) - r \tag{2.6}$$

Y el menor elemento r cuya suma parcial acumulada es al menos s es:

$$r = 1 + rank_B(1, select_B(0, s)) \tag{2.7}$$

Donde B es la secuencia de bits.

Observe que el predecesor de r en la fórmula anterior es el mayor elemento cuya suma parcial acumulada es menor que s.

La secuencia puede ser comprimida a orden cero con el enfoque propuesto por [RRR02].

Cabe destacar que estos resultados son válidos para secuencias estáticas. Para el caso dinámico se debe consultar [HSS03, MN08, GN08].

2.2.6. Arreglo de Sufijos

La mayoría de las soluciones teóricas que existen para implementar el arreglo de sufijos de modo comprimido tienen problemas para trabajar en memoria secundaria [GNP+06, Sad03], porque del mismo modo que con la XBWT de Ferragina et al. (la cual es una adaptación de éstas técnicas al caso semi-estructurado), el acceso a disco no es aleatorio o la estructura completa en RAM es demasiado grande. La mayoría de ellos sacan provecho de una permutación del texto original para comprimir, ocupando un espacio comparable al del texto original [NM07].

González y Navarro han desarrollado una solución para indexar texto que sí considera el uso de memoria secundaria como un problema asociado, y promete responder a las consultas de modo más eficiente que en el caso descomprimido [GN07b, GN07a].

Un enfoque interesante en esta línea y que se puede utilizar en un contexto de lenguaje natural, es el presentado en [FNP08, BFLN08, BFN+08] para desarrollar *self-indexing*, lo cual permitiría su uso en vez del índice invertido.

2.2.7. Dataguide

El dataguide es un esquema descriptivo derivado de los datos del documento, que fue definido por Goldman et al en 1997 [GW97]. Su estudio abarcó a los repositorios de documentos cuyas relaciones estaban constituidas en forma de grafo, y cuya estructura esquemática no estaba definida a priori (también abordaron el caso particular de los árboles).

La novedad y ventaja del enfoque es que se inventó para apoyar al usuario en la consulta sobre dichos repositorios de modo que, a través del dataguide actualizado permanentemente,

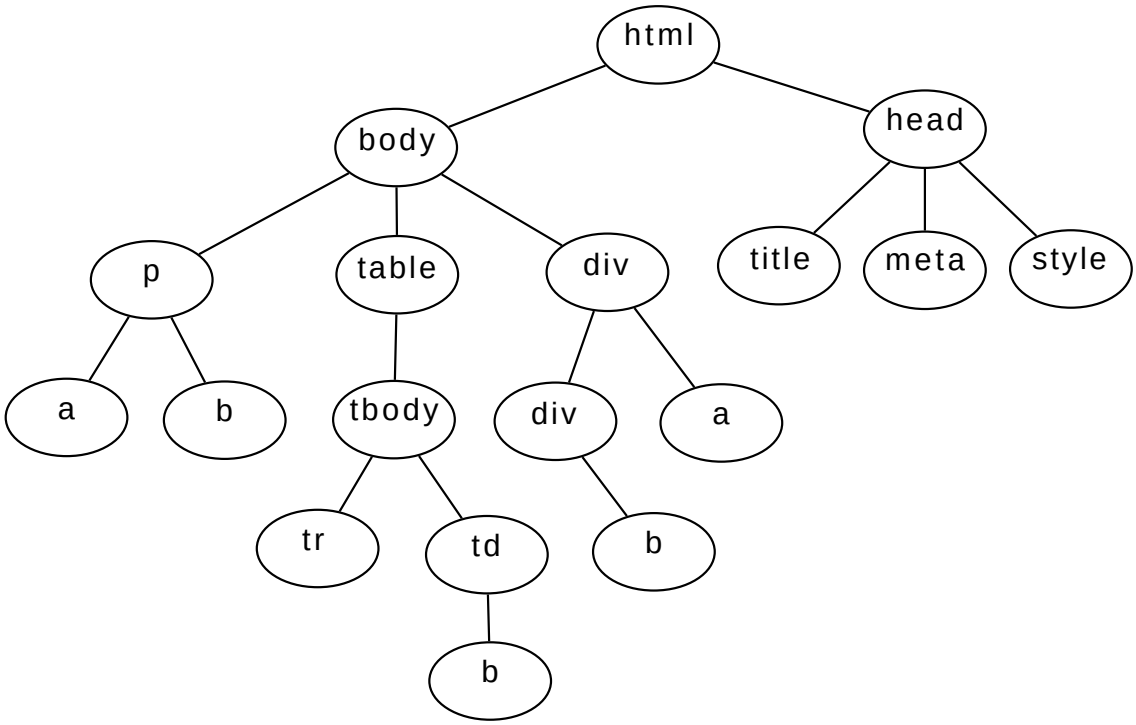

Figura 2.5: Ejemplo de Dataguide

se podía realizar consulta teniendo acceso al esquema descriptivo de la base de datos. Es una estrategia muy útil para enfrentar colecciones de documentos donde no hay control de la estructura, ni restricciones suficientemente fuertes como generalmente ocurre en el modelo relacional.

El estudio incluye resultados teóricos relevantes en torno a características que definen un dataguide; unicidad de los nodos, unicidad de los caminos, existencia, y otros.

Básicamente, un dataguide para un árbol consiste en una estructura reducida (también con estructura de árbol) que representa a todas las rutas de etiquetas desde la raíz a todas las hojas. Un dataguide resume las características estructurales de un árbol llamado fuente, y puede almacenar conjuntamente todo tipo de información estadística del mismo. Cada camino existente en el árbol fuente aparecerá sólo una vez en el dataguide, por lo tanto la existencia de una ruta en él, determina la existencia también en el árbol fuente. En la figura 2.5 se observa un ejemplo para una colección de documentos XHTML.

El poder de un dataguide está en la capacidad que provee de evaluar, en una estructura reducida, la potencialidad de una consulta estructural sobre el árbol fuente.

Un enfoque basado en esta idea fue desarrollado por Fuhr y Gövert [FG02] para apoyar índices invertidos estructurales sobre XML, y se utiliza en varias de las propuestas para nuestro problema como se verá más adelante.

2.2.8. Almacenamiento para XML y Soporte para el Lenguaje XQuery

Existe una serie de estudios teóricos destinados a resolver el problema de la compresión de documentos almacenados en formato XML. Algunos de ellos ofrecen además capacidad de hacer consultas sobre el fichero comprimido, ya sea mediante XPath o a través de XQuery.

Los compresores XML dedicados, *XML-aware compressors*, son aquellos que sacan provecho de las particularidades del formato XML para producir mayor compresión que con un enfoque estándar. Los *XML-queryable compressors* son aquellos que, además de tomar en cuenta el formato para comprimir, proveen mecanismos para consultar sobre la representación comprimida.

A continuación se describen características de cada tipo mediante un conjunto de enfoques reconocidos en el estado del arte. Algunos de ellos cuentan con implementaciones funcionales disponibles, que serán analizados más adelante (ver sección 2.4).

Compresores XML dedicados

XMill No es exactamente un compresor. Según sus propios autores ([LS00]), es una herramienta extensible para aplicar compresores existentes al almacenamiento en XML; es decir, un marco de trabajo para potenciar la compresión sobre este tipo de información estructurada.

Sus desarrolladores proponen un conjunto de estrategias para mejorar el grado de compresión alcanzado con la aplicación, pero sin duda la estrategia principal es la separación de la compresión por contextos. La idea básica es identificar el tipo de información que contiene cada nodo XML, categorizarlos y agrupar los contenidos consecutivamente para luego aplicarles un algoritmo ad-hoc. El enfoque por defecto para cada contexto es aplicar gzip (que combina Lempel-Ziv con una variante de Huffman).

La aplicación permite también la inclusión de guías por parte del usuario para favorecer

la identificación de la redundancia y el tipo de datos almacenados, y la incorporación de nuevos compresores. Sus resultados son buenos, de hecho permite mejorar considerablemente la compresión de gzip aplicado directamente sobre el fichero, y es el primer enfoque que aborda la compresión utilizando la categorización de la estructura XML para sacar provecho de los contextos.

XMLPPM Es un compresor basado en la familia de compresores PPM (Prediction by Partial Match [CW84]) y en un enfoque particular para modelar la estructura del árbol llamada Multiplexed Hierarchical Modeling (MHM) [Che01]. La idea es producir una codificación basada en bytes para los eventos producidos por el parser XML, y luego aplicarles una multiplexación de acuerdo a los contextos estructurales de modo de sacarle provecho con la compresión PPM. El contexto es dado por la caracterización del camino desde la raíz hasta el nodo de texto.

La idea es una evolución de XMill en el sentido que se usan diferentes compresores en cada contenedor, y la información de la jerarquía se usa para mejorar la compresión.

Como toda compresión basada en la familia de compresores PPM[9], XMLPPM no permite acceso local.

SCM SCM (Structural Contexts Model) es un modelo de compresión propuesto por Joaquín Adiego et al. [AdN05] para documentación semi-estructurada. En el estudio se demuestra la conveniencia de utilizar los contextos dados por la estructura XML para categorizar los contenidos y por ende sacar provecho de las similitudes al comprimir. El enfoque ya había sido abordado en XMill y XMLPPM, pero éste mejora la tasa de compresión para documentos mayores a 5Mb con la variante **SCMPPM** que combina el modelo con PPM. SCM agrupa los contextos por etiqueta, no por camino desde la raíz como XMLPPM.

Una de las variantes interesantes del modelo es desarrollada con un código de Huffman orientado a palabras llamado **SCMHuff**. Con este enfoque, la compresión sí permite acceso aleatorio, descompresión parcial y búsqueda directa sobre la colección comprimida, mejorando

[9] http://en.wikipedia.org/wiki/Prediction_by_Partial_Matching.

la tasa de compresión del enfoque no-contextual, y de aquellos compresores que ofrecen este tipo de acceso.

La idea es separar los diccionarios de cada contexto, pero además se presenta una heurística para mezclarlos si la tasa de compresión lograda al abordarlos separadamente no es más conveniente [ANd07b].

XCQ Es un compresor XML que separa la estructura del contenido. Aplica compresión gramatical para la estructura de acuerdo a la información de la DTD (Document Type Definition), y aplica compresión estándar Lempel-Ziv `gzip` al texto [NLWL06].

XWRT XML Word Replacing Transform es un compresor basado en diccionario que reemplaza las palabras más recurrentes por referencias al diccionario creado en una pasada previa. El código resultante puede ser comprimido con `gzip`[10], `LZMA`[11] o `PPM`. Tiene muy buenos resultados comparado con las alternativas existentes más destacadas [SGS07, Sak08].

Compresores XML navegables

XBZipLib Es una biblioteca open source con licencia GPL[12], desarrollada por Ferragina et al. y justificada en varias publicaciones ([FLMM06, FLMM07]). Es una propuesta que integra la estructura XML con el contenido contextual mediante una adaptación al formato de la Transformada de Burrows/Wheeler [BW94], comprimiendo dos secuencias resultantes en formato ZIP. Logra buen resultado en la compresión permitiendo navegación sobre la estructura.

Succinct DOM SDOM es una implementación en C++ de las funciones estáticas de DOM (Document Object Model) basada en estructuras sucintas. Debido a esto, permite representaciones eficientes en memoria principal de documentos de gran tamaño.

Una variante (SDOM-SC) aplica compresión `bzip` (basado en BWT [BW94]) sobre el contenido de los nodos de texto y de atributos, logrando grados de compresión comparables

[10] http://www.gzip.org/.
[11] http://en.wikipedia.org/wiki/LZMA.
[12] http://www.di.unipmn.it/Tecnical-R/Technical-3/TR-INF-2005/index.htm.

con otros compresores XML con soporte para consultas (*queryables*). Si bien SDOM-SC no es consultable directamente pues sólo provee una interfaz DOM, en la práctica aporta un importante componente para implementaciones navegables sobre XML [DRR08].

XGrind Es un compresor XML que soporta consultas sobre el fichero comprimido. Comprime de modo que se mantiene la estructura en el fichero resultante (compresión homomórfica), permitiendo el reuso de las técnicas para XML estándar.

XGrind representa las etiquetas en forma numérica, mientras que el texto es comprimido con Huffman orientado a caracteres. Las operaciones de consulta son un dialecto de XQuery/XPath [TH02].

XQzip Es un compresor XML que soporta consultas sobre el documento comprimido mediante la incorporación de un índice llamado *Structure Index Tree* (SIT). XQzip soporta un espectro amplio de consultas XPath así como predicados múltiples, anidamiento profundo y agregación [CN04].

XQueC Es un compresor XML que también desarrolla separación entre la estructura y los datos. Utiliza compresión basada en diccionario pero preservando el orden. Asocia contenedores para la información identificando los contextos por pares de tipos y rutas desde la raíz a la hoja, y utiliza un sumario (Dataguide) para la estructura que permite acceder a los contenedores. La información en los contenedores es ordenada lexicográficamente, almacenada en árboles B+ (Berkeley-DB) y se realiza búsqueda binaria en ellos para las consultas de contenido [ABMP07].

XPress [MPC03] Es un compresor homomórfico que ofrece soporte para consultas un poco más avanzadas que XGrind.

XPress es un compresor XML que soporta consulta directa al documento comprimido. Similar a XGrind, adopta una estrategia de transformación homomórfica para llevar el documento XML a una forma comprimida, preservando la sintaxis y la información semántica del original.

XPress está construido sobre un esquema de codificación novedoso llamado *Reverse Arithmetic Encoding*. Esta técnica está diseñada para codificar las rutas del árbol XML hacia los elementos del documento usando intervalos de números reales. Cada uno de estos intervalos cae en el rango $[0, 1)$, el cual representa su correspondiente camino desde la raíz hacia el elemento. Estos intervalos poseen una importante característica ya que adhieren a la propiedad de *suffix containment*. Esta propiedad asegura que si un camino de elemento P es sufijo de un camino de elemento Q, entonces el intervalo que representa a P, denotado como `Ip` , debe contener al intervalo `Iq`, que representa a Q.

La PCDATA[13] y los valores de los atributos del documento son comprimidos individualmente usando diferentes métodos de compresión libres de contexto (orden cero). Dependiendo de sus tipos de datos, numéricos, enumerados y secuencias de caracteres (strings), son codificados usando representación binaria junto a codificación diferencial, codificación basada en diccionario, y codificación de Huffman, respectivamente.

Este esquema mejora el resultado logrado por XGrind porque permite responder a las consultas sobre inclusión de nodos, y también responder a las consultas sobre valores numéricos sin descomprimir el documento, gracias a la codificación de los intervalos, la preservación del orden y la codificación para los números. Sin embargo, el costo se paga en pérdida de compresión y en aumento del tiempo de codificación [NLC06].

LZCS Es un compresor basado en un enfoque original de Lempel-Ziv, adaptado para documentos muy estructurados. LZCS aprovecha las subestructuras repetidas que puedan aparecer en los documentos, reemplazándolas por una referencia hacia la primera ocurrencia. El resultado sigue siendo una estructura fácilmente legible por un humano que además puede ser transmitida por canales ASCII. Más aún, los documentos transformados permiten búsqueda fácilmente, son accesibles aleatoriamente (i.e., hacen uso de descompresión local aunque no contigua) y son navegables.

Además, se pueden comprimir en una segunda etapa con un enfoque basado en palabras como ETDC, manteniendo las características de acceso y mejorando la compresión, o con

[13] *parsed character data*, corresponde a los datos explícitos agregados a un nodo de texto.

otro enfoque adaptativo que la mejore bajo pena de perder ventajas en el acceso.

LZCS muestra resultados favorables en colecciones altamente estructuradas (formularios, facturas, intercambio de datos para web-services, etc.), ya que saca provecho de las repeticiones literales que ocurren en ellas. Por tanto, no se perfila como un compresor XML de propósito general, aún cuando posee muy buenas características para incorporarle consulta al formato [ANd07a].

En un estudio reciente [ANd08] se muestra cómo implementar consultas sobre documentos comprimidos con esta técnica.

Traductores de XQuery al Modelo Relacional

Existen muchos estudios e implementaciones de traductores del lenguaje XQuery al modelo relacional. Sin embargo, las más destacables son las siguientes:

BLAS BLAS establece un método para incorporar documentación XML en el modelo relacional desarrollando dos tipos de *labels*. La idea es identificar a cada nodo de modo de procesar consultas XPath complejas. El primero de ellos, denominado D-Label, corresponde a la representación vía segmentos de acuerdo al orden pre del documento, más el nivel del nodo en la jerarquía del árbol XML (Level). Se define para resolver eficientemente las consultas del eje *Descendant*, puesto que mediante la comparación de inclusión de los segmentos se puede determinar inclusión de los nodos.

P-Label en cambio, es una notación que sirve para procesar consultas XPath compuestas de secuencias largas de rutas de tipo `Parent/Child` llamadas *suffix path query*. La idea es evitar los joins anidados que se forman al resolver este tipo de consultas sólo con D-Label (y que son tantos como etiquetas tenga la ruta), asociando a cada nodo un segmento de acuerdo a la ruta XPath completa desde la raíz. Funciona en forma muy semejante a un Dataguide en el sentido de definir relaciones entre nodos de acuerdo a su posición estructural. Con P-Label se pueden resolver rápidamente las consultas sobre una ruta completa.

BLAS muestra un excelente desempeño en las consultas que hacen uso de P-Label comparado con aquellas que sólo incluyen D-Label, sin embargo, no considera la creación de la

consulta relacional como parte del proceso, lo cual hace la comparación un tanto injusta, pues aquellas con P-Label son más difíciles de producir a partir de la misma consulta XPath [CDZ04].

Pathfinder Pathfinder[14] es una implementación open source de un procesador XQuery puramente relacional. El acceso eficiente a los datos mediante XPath es habilitado en términos de Staircase Join (ver sección 2.2.3). El compilador *loop-lifting* traduce las expresiones XQuery en planes algebraicos puros, de acuerdo a primitivas orientadas a conjuntos para la representación de las iteraciones de XQuery (i.e., cláusulas `for`, `let`, `where`, `order by` y `return`) [Teu07].

2.3. Trabajo Previo

A continuación se describe el modelo Proximal Nodes (sección 2.3.1) sobre el cual se basa la implementación de la aplicación descrita en este trabajo. También se detalla la primera versión de la aplicación IXPN (sección 2.3.2), cuyo alcance estaba limitado a la implementación del lenguaje XPath para las consultas.

Finalmente, se describe la versión previa al desarrollo de este trabajo, denominada XPN (sección 2.3.3), cuyo lenguaje de consulta alcanza un subconjunto relevante de XQuery.

2.3.1. Proximal Nodes (PN)

Proximal Nodes es un modelo desarrollado por Navarro y Baeza-Yates [NBY95, NBY97] que describe estructuras para indexar y consultar información estructurada. Presenta un buen compromiso entre expresividad y eficiencia.

Proximal Nodes no define un lenguaje específico, sino un modelo al cual se pueden agregar operadores útiles mientras se mantenga la eficiencia. Varias estructuras independientes pueden ser definidas en el mismo texto, cada una como una categoría estricta, pero permitiendo

[14] http://pathfinder-xquery.org/technology/xpath-accel.

traslape entre áreas delimitadas por diferentes jerarquías (p.e. `chapters/sections/paragraphs` y `pages/lines`).

Una consulta puede relacionar diferentes jerarquías, pero retorna un subconjunto de nodos de sólo una de ellas (i.e. se permiten elementos anidados en las respuestas, pero no traslapes).

Cada nodo tiene asociado un segmento, que corresponde al área del texto que representa. El segmento de un nodo incluye al de sus descendientes en las jerarquías a las que pertenece. Las consultas de calce de texto (text matching) son modeladas como nodos de una "categoría de texto" especial.

El modelo especifica un lenguaje completamente funcional con tres tipos de operadores:

1. Búsqueda en texto mediante patrones (text pattern matching)
2. Recuperación de elementos estructurales por nombre (p.e. todos los nodos `chapter`)
3. Combinación de otros resultados

La idea principal detrás de la evaluación eficiente de estos operadores es un enfoque *bottom-up*; primero resolviendo las consultas en los contenidos y luego subiendo hacia la parte estructural. Se usan dos índices, para texto y para estructura, pensados para resolver eficientemente consultas del tipo 1 y 2 sin recorrer toda la base de datos. Para realizar consultas del tipo 3 eficientemente, sólo se permiten operaciones que relacionan nodos cercanos. Nodos cercanos son aquellos cuyos segmentos son relativamente próximos. De este modo, la respuesta es construida a través de la sincronización de sus operandos, conduciendo en la mayoría de los casos a un costo constante amortizado por elemento procesado. Para el calce de texto, hay un sub-lenguaje separado que es independiente del modelo.

La expresividad de este modelo lo hace competitivo o superior a muchos otros, pues muchos operadores útiles encajan en él, y puede ser implementado eficientemente, necesitando tiempo lineal para la mayoría de sus operaciones y en todos los casos prácticos [BYN02].

Cabe destacar que la estructura XML funciona sobre una sola categoría dentro del modelo Proximal Nodes, porque este lenguaje de marcado no permite traslapes sino sólo como inclusión.

2.3.2. IXPN

Es un procesador de documentos XML con consultas a través del lenguaje XPath. IXPN es la sigla correspondiente a *Implementación de XPath a través del modelo Proximal Nodes*. Fue desarrollado por Manuel Ortega como memoria de título [NO03] y su trabajo consistió en analizar las diferencias entre los lenguajes, adaptar los modelos y desarrollar un software en base a lo anterior. Para ello el prototipo del modelo Proximal Nodes de 1997 [NBY97] fue reimplementado completamente, agregando nuevas operaciones necesarias para soportar XPath sin perturbar la filosofía básica de Proximal Nodes.

En la figura 2.6 se observa el conjunto de operadores necesarios para la aplicación del modelo a la consulta XPath sobre XML.

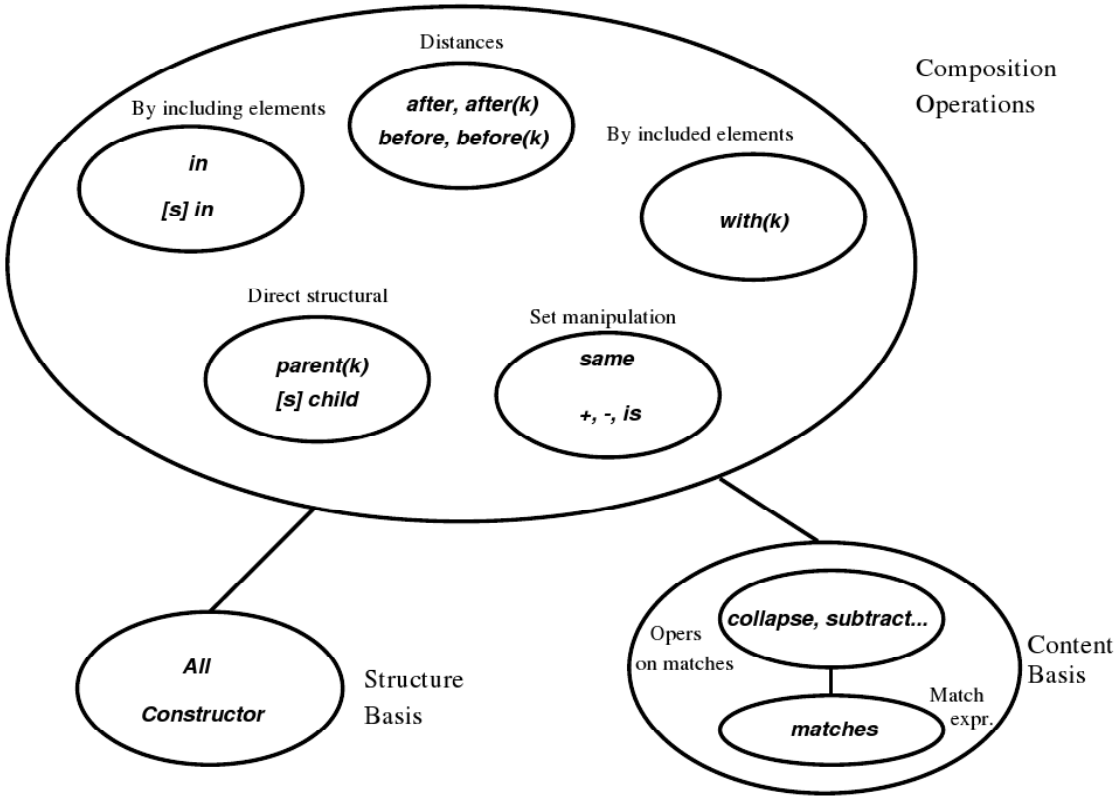

Figura 2.6: Operaciones relevantes para XML en el modelo PN.

La adaptación al modelo Proximal Nodes resultó ser exitosa y demostró resultados favorables en comparación con las alternativas gratuitas disponibles en ese momento [NO03].

2.3.3. XPN

Es un desarrollo posterior de IXPN que extiende la capacidad de consulta al lenguaje XQuery. Su desarrollo en el Centro de Investigación de la Web (CIW) ha estado a cargo del Profesor Benjamin Piwowarski, quien ha incorporado la biblioteca SOUL al código. La descripción que se hace a continuación corresponde a la implementación posterior a IXPN pero previa al desarrollo de este trabajo, a modo de diagnóstico. Dicha implementación conserva muchas de las características heredadas de IXPN, pero se han modificado otras.

XPN es una aplicación que tiene muy buen comportamiento en el análisis y creación de los planes de consultas, y resuelve por construcción problemas que poseen otras bases de datos XML en el *parsing*, que las llevan a bajar su desempeño en ordenamiento de los datos durante las consultas.

Tiene mejor respuesta en colecciones de tamaño pequeño/mediano que por ejemplo MonetDB/XQuery [BGv+06], y se ha desarrollado una solución tendiente a mejorar las consultas que implican *joins* en resultados intermedios, que son un problema cuando se realizan sobre secuencias de nodos numerosas.

XPN aún no es una base de datos plenamente funcional pues le falta implementar algunos operadores para cumplir con el estándar XQuery XQTS[15], pero se perfila como una buena solución que utiliza almacenamiento en fichero sin recurrir al modelo relacional como lo hace MonetDB/XQuery.

Aplicación del modelo Proximal Nodes

Los principales operadores definidos por el modelo Proximal Nodes fueron adaptados en la implementación de IXPN [NO03]. A continuación se presentan los operadores actualmente existentes, relevantes para XML y clasificados por tipo de acuerdo a [BYN02]:

[15] XQTS es la sigla para XML Query Test Suite. La XML Query Test Suite provee un conjunto de métricas para determinar si el lenguaje XQuery puede ser implementado interoperablemente de acuerdo a como se publica. XQTS ayudará a los implementadores a identificar posibles problemas tanto en la especificación XQuery como con su software. http://www.w3.org/XML/Query/test-suite/.

Base Estructural Este conjunto se representa por dos operadores denominados `Struct` y `Segment` destinados a la recuperación de secuencias obtenidas por tipo de elemento o atributo XML arbitrarios, y de nodos de texto XML, respectivamente. Las secuencias se obtienen ordenadas por orden de documento.

Base Contenido Este conjunto se representa por un operador denominado `Match` destinado a la recuperación de palabras en el texto. Las secuencias de posiciones se obtienen ordenadas por orden de documento. Es importante destacar que la recuperación se realiza en un contexto de lenguaje natural.

Los operadores Collapse y Subtract, destinados a la manipulación de secuencias creadas por más de un operador de tipo `Match`, no están implementados en el prototipo.

Operaciones Composicionales Este conjunto de operadores binarios está destinado a la obtención de relaciones de secuencias obtenidas ya sea por consulta directa al índice (base contenido o estructural), por construcción de secuencias como resultados intermedios, o mediante operaciones de agregación y comparación.

Relación de Distancia Agrupa operadores que permiten obtener secuencias de nodos que satisfacen condiciones en la relación **preceding/following** del eje XPath.

After: `P` *`After`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` que existen después de al menos un nodo en la secuencia `Q` en el orden dado por el documento.

Before: `P` *`Before`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` que existen antes de al menos un nodo en la secuencia `Q` en el orden dado por el documento.

Relación de Inclusión Agrupa los operadores de PN destinados a las operaciones *including* e *included* (incluyentes e incluidos). Corresponde a operaciones en el eje XPath Ancestor/Descendant.

In: `P` *`In`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` que están incluidos en al menos un nodo en la secuencia `Q`.

With: `P` *`With`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` que incluyen al menos un nodo en la secuencia `Q`.

Relación Estructural Directa Contiene operadores destinados a resolver consultas en la relación Parent/Child del eje XPath.

Child: `P` *`Child`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` que son descendientes directos de un nodo en la secuencia `Q`.

Parent: `P` *`Parent`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` que son ascendientes directos de al menos un nodo en la secuencia `Q`.

Manipulación de Conjuntos Contiene operadores destinados a resolver consultas de agregación de secuencias o de comparación estructural o de contenido. Sirven para representar consultas sobre conjuntos de nodos, para la construcción de frases, o para la comparación de valores de contenido, entre otras.

Union: `P` *`Union`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` y todos los nodos de la secuencia `Q`, manteniendo el orden de documento en dicha unión.

Is: `P` *`Is`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` que son también nodos de la secuencia `Q`.

Same: `P` *`Same`* `Q` produce una secuencia que contiene todos los nodos de la secuencia `P` cuyo segmento es igual al segmento de al menos un nodo de la secuencia `Q`. Es idéntico al operador *`Is`* pero se aplica a contextos de contenido.

Para almacenar los documentos de forma conveniente a la consulta, XPN utiliza la notación (*Pre*, *Size*, *Level*) (ver sección 2.1.4) para identificar a cada nodo en la estructura XML y permitir las operaciones entre ellos.

Archivo

La aplicación utiliza el documento original (sin comprimir) para generar las respuestas, y por ende ocupa mucho espacio. Debido a esto, XPN tiene buen rendimiento en la serialización de los resultados porque se accede al documento original, pero por lo mismo presenta mucha fragilidad en el almacenamiento: mover el repositorio deja al índice inservible, ocupa más espacio que el necesario, y hace una serie de supuestos insostenibles en una aplicación robusta: la separación entre la declaración de dos atributos está constituida por un único carácter de espacio en blanco ("␣"), no puede haber salto de línea dentro de un texto, no puede haber espacios antes de cerrar una etiqueta, etc. Estos detalles impiden la manipulación de mucha documentación XML válida existente, lo cual hace de XPN un procesador XQuery de soporte muy deficiente.

Índice Invertido Estructural

XPN tiene un índice invertido estructural que está basado en los algoritmos de intersección de listas invertidas propuestos en [ST07] para resolver eficientemente la intersección de listas de enteros. La diferencia está en que, en XPN, cada lista corresponde a una secuencia ordenada de segmentos marcados con una misma etiqueta, a la cual se accede mediante un hash. En la figura 2.7 se observa un ejemplo para las etiquetas de una colección de documentos XHTML.

Básicamente, lo que hacen los índices invertidos estructurales es almacenar secuencias de pares constituidos por un campo Offset y un campo Size que define únicamente a un segmento de caracteres, y que están ordenadas por offset de inicio. Es decir, entregan un flujo de segmentos de fichero en orden de documento.

Cada lista está acompañada de una estructura jerárquica que permite recorrerla secuencialmente haciendo saltos convenientes en base a un parámetro entregado en la llamada que corresponde a un valor de offset. Este parámetro de la llamada es denominado **Requerimiento**.

La lista se divide en bloques y a cada uno de ellos se le asigna un segmento dedicado (no

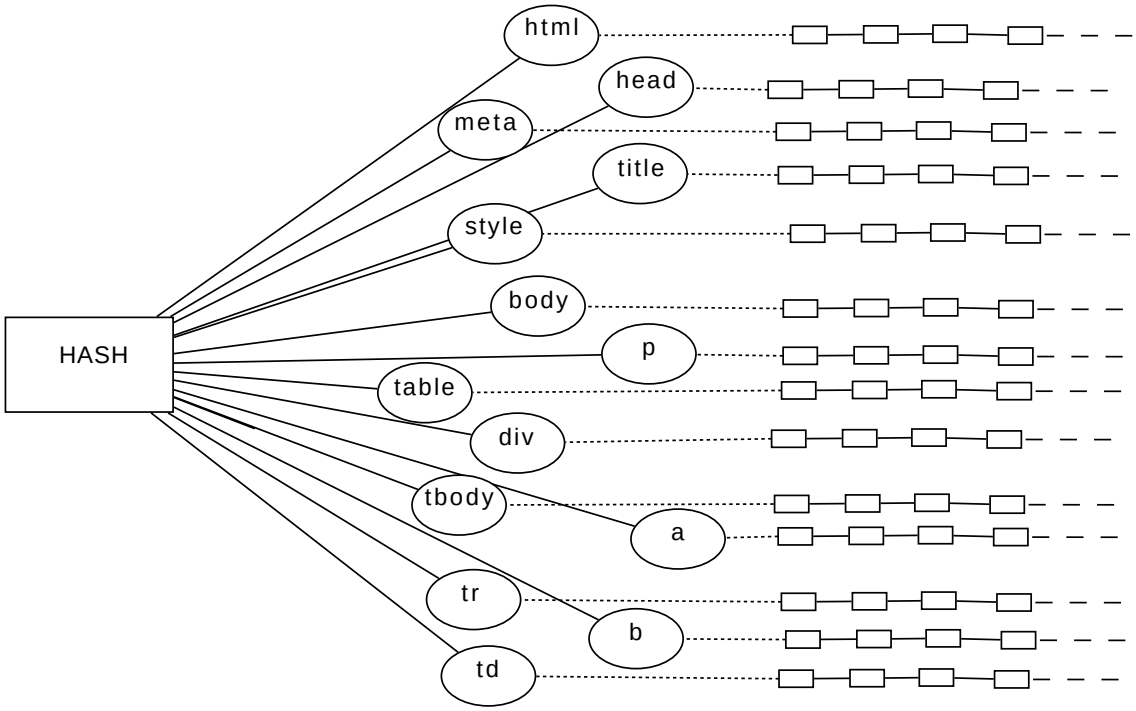

Figura 2.7: Índice Invertido Estructural

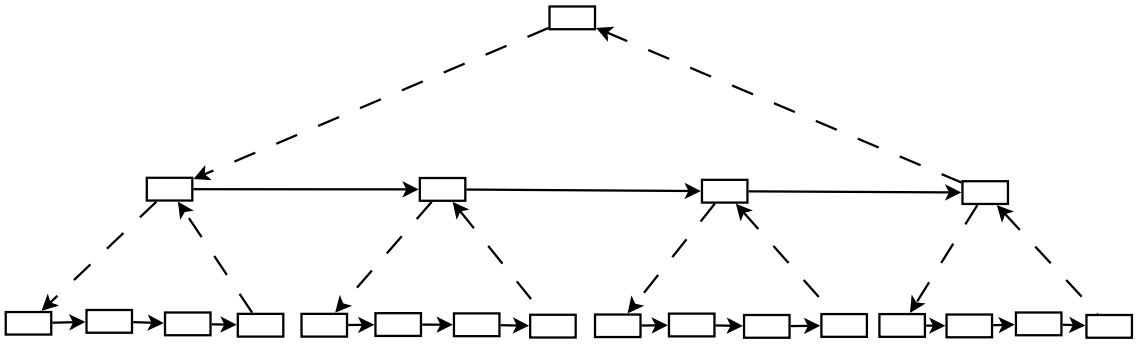

Figura 2.8: Jerarquía de Bloques

presente en el documento XML) que sintetiza la información que contiene. Es decir, define un meta-segmento que empieza en el offset del primer segmento, y termina en el offset del último. Si un bloque contiene segmentos que van desde la posición de offset 3 a la posición de offset 17, el segmento del bloque indicará que ese segmento comienza en 3 y tiene tamaño 14.

Los segmentos de bloque pertenecientes a un mismo nivel (el nivel de la lista es el nivel cero), son a su vez almacenados en un bloque superior, con su correspondiente meta-segmento. La estructura se reproduce recursivamente hasta un nivel superior en el cual sólo hay un bloque. Los bloques son de tamaño fijo en bytes (parametrizable durante la creación) por

tanto el número de niveles depende de la cantidad de elementos de la lista (ver figura 2.8).

Al iniciar la recuperación desde una lista determinada, siempre se debe recuperar el bloque de mayor jerarquía desde disco antes de comenzar la consulta.

El índice está pensado para retornar segmentos en orden secuencial a medida que se realizan llamadas sobre él, para seguir el modelo Proximal Nodes (ver sección 2.3.1) donde los operadores básicamente trabajan sobre flujos de segmentos. Esto significa que una vez que el índice ha encontrado un segmento útil, lo retorna y queda en esa posición esperando por la siguiente llamada, manteniendo así el orden de documento del flujo producido.

Los requerimientos pasados en la llamada al índice consisten en un valor mínimo de offset que debe ser satisfecho por el segmento recuperado. Así, el índice recorre el primer nivel de la jerarquía hasta que encuentra un segmento de bloque que satisface el requerimiento, recupera el bloque correspondiente y lo recorre secuencialmente hasta satisfacer nuevamente el requerimiento. El proceso continúa hasta el nivel más bajo de la jerarquía donde los segmentos almacenados sí corresponden a nodos del documento XML, retornando el primer segmento que satisface el requerimiento.

En los niveles superiores, el requerimiento se satisface cuando el segmento de bloque **contiene** al offset requerido, es decir, cuando el offset de principio de bloque es menor o igual que el offset requerido, y el offset de término de bloque es mayor o igual que éste. Esta condición significa que existe algún bloque hacia abajo de la jerarquía que también satisface la condición.

Al llegar a un segmento de documento ubicado en el nivel más bajo, el requerimiento se satisface con el primer segmento que supera o iguala el offset requerido.

La necesidad de recorrer los bloques en orden secuencial se explica porque se codifican las diferencias de los offsets de los segmentos almacenados. Ésto logra gran reducción de tamaño y permite almacenar más segmentos por bloque a cambio de tener que recorrer un bloque entero en el peor caso para encontrar el segmento buscado. Note que sólo se recorre un bloque de nivel intermedio o inferior cuando se sabe que el segmento buscado está contenido en él gracias a la búsqueda previa en el nivel inmediatamente superior. La estructura es muy

similar a un B-Tree salvo por el acceso secuencial en cada nodo.

La inclusión de los requerimientos en el funcionamiento del índice tiene como objetivo la reducción de las comparaciones inútiles al intersectar listas distintas. La implementación corresponde a una adaptación del *skipping* en la base de datos (ver *staircase join* en la sección 2.2.3 y en [GvT03]), que se produce al resolver las consultas de tipo 1 y 2 del modelo Proximal Nodes (ver sección 2.3.1). Gracias a las modificaciones hechas se redujo el tamaño de los resultados intermedios que se producían en IXPN (y en la propuesta original del modelo PN [NBY97, BYN02, NO03]).

En la implementación previa, los flujos de nodos provenientes de los índices invertidos se recorrían secuencialmente sin tomar en cuenta las características del resultado parcial obtenido, teniendo que recorrer invariantemente la lista completa hasta el resultado. Con la introducción en cada operador de un requerimiento al procedimiento que solicita el siguiente nodo desde una fuente, más la estructura necesaria en los índices invertidos, se puede saltar conjuntos considerables de nodos intermedios que, de acuerdo a cada operador, no pueden ser resultado útil. Con este enfoque adaptativo, se reduce el tamaño de los resultados intermedios y también los accesos a memoria secundaria.

La intersección se produce cuando un operador binario como por ejemplo *In* o *With* (ver sección 2.3.3) desea determinar si dos nodos de algún tipo cumplen una relación específica en la estructura del árbol XML. Esta relación se verifica de acuerdo a la posición que ocupa cada segmento en el documento. Por ejemplo, si un segmento de texto comienza antes que otro y además termina después, se puede decir que el segundo está contenido en el primero, lo cual significa que el primer nodo correspondiente es ascendiente del segundo, o al revés, que el segundo es descendiente del primero.

Ahora bien, cuando se ha recuperado desde ambos flujos un operando potencial de resultado para el operador, de modo que no se satisface la condición buscada, un operando puede servir de requerimiento para el otro, y por tanto, servir para la obtención del siguiente resultado. De este modo, a través de una propagación por los operadores en el árbol de consulta, los índices invertidos reciben como parámetro un valor mínimo de Offset que debe

ser satisfecho al retornar un nuevo segmento desde su flujo.

Volviendo al ejemplo, suponga que el primer segmento tiene etiqueta P, y comienza y termina antes que el segundo segmento cuya etiqueta es Q. Se quiere obtener nodos etiquetados P y Q que satisfagan la relación `P In Q`, o sea *todos los nodos P que son descendientes de algún nodo Q*. Entonces, se va a descartar al primer segmento como resultado, pues todos los segmentos que existen en Q son el actual o empiezan después que él, y por tanto éste P no puede ser ya descendiente de ninguno más. Luego se pide al índice invertido de P, que entregue el siguiente segmento cuyo offset no sea menor al offset del segundo segmento (el actual Q), o dicho de otro modo, que no comience antes.

A la inversa, si Q está antes que P pero no es ascendiente (es decir, termina antes que empiece P), y por tanto no puede generar resultados, entonces es deseable pedir al índice invertido estructural el siguiente segmento etiquetado Q que termine después que el offset donde termina P (observe que éste offset es igual a *Offset + Size*). De otro modo, no podrá generarse un nuevo resultado. Este algoritmo adaptativo tiene muy buenos resultados, y es utilizado en MonetDB [BGv+06] con la introducción del Staircase Join en base al Pre y Post orden (ver sección 2.2.3).

En el caso de XPN, no es trivial la implementación del segundo requerimiento sobre el índice invertido estructural porque éste ordena la secuencia por Pre, no por Post. Sin embargo, el índice redujo considerablemente el tamaño utilizado en disco de la implementación original, y permitió acelerar las consultas incorporando características propias del documento XML.

Índice Invertido de Lenguaje Natural

El índice invertido de lenguaje natural de XPN se crea para palabras filtradas del conjunto completo mediante una lista arbitraria de stopwords (palabras vacías). En un principio, el listado de palabras vacías es agregado a un árbol ternario de búsqueda (*Trie*) de modo que al encontrar cada palabra en el análisis del documento, primero se busca en la lista de stopwords, y luego, en caso de que no ser encontrada, se agrega a otro trie para implementar la búsqueda de palabras.

La indexación funciona de la siguiente forma: Primero se procesa la palabra para llevarla a una forma canónica (esto significa representarla sólo con minúsculas). Luego se busca la palabra en el trie de stopwords. Si no es encontrada, se busca en el trie de lenguaje natural. Éste almacena, para cada palabra, una lista de ocurrencias marcando su posición en la secuencia completa de palabras, es decir, almacenando en una lista secuencial de palabras un **Offset de Palabra** para cada ocurrencia. Esto significa que en la secuencia completa la primera palabra tendrá offset 1, la segunda offset 2, y así sucesivamente. La lista de cada nodo del trie sólo contendrá la posición de las ocurrencias en la secuencia. Cuando la palabra es nueva, la lista se crea con la palabra en el nodo correspondiente, si no, se agrega al final de la lista.

Una vez que el trie alcanza un tamaño definido arbitrariamente, su contenido es volcado a memoria secundaria en tres ficheros interrelacionados: el fichero del vocabulario (VOC), el fichero de las posiciones (POS) y el fichero de los offsets (OFF). El primero contiene una lista de palabras ordenadas lexicográficamente. El segundo contiene las listas de ocurrencias concatenadas de cada palabra una detrás de la otra, y también en orden lexicográfico. Cada lista incluye en primer lugar el número de ocurrencias y luego la lista de ocurrencias codificadas con diferencias. Finalmente, para cada palabra indexada, el tercer fichero almacena el offset de la palabra en el fichero del vocabulario, y luego el offset en el fichero de las posiciones indicando dónde comienza la descripción de su lista de ocurrencias. Estos dos campos del fichero de offsets deben ser de un tamaño fijo para permitir acceso aleatorio. Después incluye el largo de la codificación de la lista de ocurrencias en bytes y el valor de la última posición de la última ocurrencia de la lista (que al estar codificada con diferencias sólo puede ser obtenida al recorrer la lista entera secuencialmente). Los dos últimos campos sólo se almacenan en índices parciales, es decir, cuando el trie es volcado en la mitad del proceso de indexación con el objetivo de liberar memoria principal, y tienen el objetivo de facilitar un proceso de mezcla que se realiza al final de la indexación (*Merging*), y que junta todos los índices parciales (tríos de VOC, OFF y POS) en un trío definitivo, dejando sólo los dos primeros offsets disponibles para acceso aleatorio en OFF.

En este fichero además se almacena el número de palabras distintas del trie.

El índice invertido de lenguaje natural se utiliza del siguiente modo: En primer lugar, la palabra es llevada a su forma canónica, luego la lista de palabras vacías se carga en un trie (si es que no ha sido cargada en una búsqueda previa), y la palabra se busca en él para descartar su búsqueda en caso que pertenezca al conjunto. Si no, se abren los tres ficheros antes definidos y se procede a hacer búsqueda binaria de la palabra en el índice. Se recorre el fichero OFF partiendo por la mitad (antes se ha recuperado el número de palabras indexadas que permite obtener este valor), se recupera el offset en VOC, se extrae la palabra de dicho fichero y se compara con la buscada. Si la palabra coincide, se extrae el offset del fichero POS y se procede a recuperar la lista de ocurrencias, retornando tantos offsets de palabra como ocurrencias indica la descripción. Si la palabra no coincide, se determina si está antes o después lexicográficamente de la palabra con la cual se comparó. En uno u otro caso, la búsqueda binaria se desarrolla recursivamente hasta encontrar la palabra o hasta determinar que no existe en el índice y por ende tampoco en el documento original.

Estas operaciones las realiza el operador **Match** (ver sección 2.3.3), cuyo parámetro de entrada es la palabra de búsqueda.

Cabe destacar que la forma de creación y consulta hace que el índice no sea cargado completo en memoria RAM durante su creación ni para su consulta, lo cual permite trabajar con documentos de tamaño superior a la memoria disponible.

Combinando las Estructuras

De las descripciones anteriores, se desprende que XPN utiliza dos offsets simultáneamente para el proceso de indexación: uno para la secuencia de caracteres de acuerdo con el índice invertido estructural, y otro para la secuencia de palabras válidas (i.e., no vacías) de acuerdo al índice invertido de lenguaje natural.

Esto significa en la práctica que el índice invertido estructural almacena toda la información correspondiente a cada nodo en el nivel inferior, lo cual corresponde a 5 campos:

from Offset de Texto

size Longitud de Segmento de Texto

first_word Offset de la primera palabra contenida en el segmento

word_length Número de palabras contenidas en el segmento

level Nivel en la jerarquía del árbol XML

En base a los dos primeros valores (offset de segmento de fichero) se puede recuperar el contenido de un nodo XML completo desde el documento original. Este proceso, llamado **serialización**, consiste básicamente en imprimir el nodo correspondiente. Cabe destacar que no es necesario proveer información adicional sobre el anidamiento de las etiquetas, pues el segmento contiene un subárbol bien formado. Además, esta codificación permite desarrollar las tareas de intersección de las relaciones estructurales en conjunto con el índice invertido estructural (requerimientos).

En cambio, el offset de palabra (definido por el segundo par de valores) permite determinar si una palabra buscada en el índice invertido de lenguaje natural está contenida o no en un nodo dado, y en qué relación se encuentra con otras palabras. Esto se fundamenta en la necesidad de hacer búsqueda de frases sobre la colección, es decir, de relacionar un conjunto de palabras en posiciones consecutivas, condición que no podría ser determinada sólo mediante el offset de fichero. Dicha función la realiza el operador **Same**, cuyo objetivo es producir la intersección de resultados del operador XQuery $fn : contains$ y otros, haciendo consultas sobre el contenido para combinarlas con características estructurales.

2.4. Trabajo Relacionado: Procesadores XPath/XQuery

Como procesadores XPath/XQuery se entenderá a todos aquellos que soporten consultas sobre dichos lenguajes, ya sean las bases de datos XML o los procesadores XML estáticos.

2.4.1. Bases de Datos XML

Hay un amplio espectro de aplicaciones dedicadas a proveer soporte para el lenguaje XQuery. Un tipo de ellas abarca a los **procesadores XQuery**, que son aquellos que no funcionan en base a un repositorio propio, sino que analizan el documento fuente al momento de realizar la consulta. Otro tipo abarca a las bases de datos propiamente tales. Algunas de ellas traducen hacia el modelo relacional, otras almacenan documentos XML, y otras desarrollan formatos binarios propios para el almacenamiento.

NoK

Es una base de datos XML cuyo enfoque para resolver las consultas XPath es la búsqueda de patrones de árbol NoK (next-of-kin pattern matching). La idea es separar las relaciones estructurales en dos conjuntos: el primero, definido como el conjunto de las relaciones locales (NoK), incluye a las relaciones Parent/Child y Following-/Preceding-Sibling, y el segundo, aquél de las relaciones entre nodos más lejanos, incluye a Ancestors, Descendants, Followings, etc. De este modo, la búsqueda sobre relaciones locales, al ser aplicada con primera prioridad, permite reducir el tamaño de los conjuntos a ser operados en las relaciones estructurales menos restrictivas.

Basados en la categorización que sus autores definen de los enfoques existentes para XPath (ver sección 2.1.2), se justifica el modelo como un enfoque híbrido que toma lo mejor de cada área.

Además, desarrollan un formato de almacenamiento sucinto basado en secuencias de bytes y utilizando *hashing* para los nodos de contenido [ZKÖ04].

El software no está disponible.

ISX

Es una base de datos XML funcional con almacenamiento basado en estructuras sucintas para paréntesis balanceados. Reportan muy buenos resultados en consultas XPath, pero indexan el contenido de texto de los nodos mediante hashing y de ese modo lo consultan,

como si fuese parte de la estructura. Por esta razón, no soporta búsquedas en texto ni a nivel de lenguaje natural.

El enfoque de la consulta es recorrer la estructura del árbol haciendo matching secuencialmente [WLS07].

El software no está disponible.

MonetDB/XQuery

Es un sistema de base de datos open source para aplicaciones de alto desempeño en data mining, OLAP, GIS, XML Query, texto y recuperación de información multimedial basado en el modelo relacional[16].

Provee funcionalidad de base de datos XML abarcando soporte casi completo del lenguaje XQuery, incluyendo módulos, funcionalidad XUpdate con transacción segura, funciones definidas por el usuario, y un caché de consultas (usando módulos XQuery). La implementación se basa principalmente en la incorporación del StairCase Join al motor de una base de datos relacional y a la incorporación de Pathfinder para la traducción de XQuery al modelo relacional (ambos explicados previamente).

El sistema tiene muy buen desempeño y es escalable a colecciones XML de gran tamaño.

Qizx/db

También denominado **XQuest**, es un motor de base de datos con almacenamiento eficiente de documentos y consultas indexadas.

Se encuentra disponible una versión de evaluación limitada[17].

eXist

Es una base de datos XML nativa y open source caracterizada por ser eficiente, con procesamiento XQuery basado en índices, indexación automática, extensiones para búsqueda en

[16] http://monetdb.cwi.nl/.
[17] http://www.xmlmind.com/qizx/.

texto (full-text), soporte XUpdate, extensiones de actualización XQuery y buena integración con herramientas existentes de desarrollo XML.

La base de datos implementa el borrador actual XQuery 1.0, con la excepción de las características de importación de esquemas y validación de esquemas definidas como opcionales en dicha especificación.

Es una de las bases de datos XQuery más populares en el mercado y está programada en Java[18].

Apache Xindice

Apache Xindice[19] es una base de datos XML nativa muy utilizada en la práctica. Sin embargo, no fue posible encontrar información más allá de su API para conocer la metodología utilizada en el almacenamiento del documento XML, ni en la metodología para implementar búsqueda por etiquetas, documentos, etc.

Sedna

Sedna[20] es una base de datos XML con soporte para XQuery validado por la W3C XQuery Test Suite. Es distribuida como software open source bajo una Licencia Apache 2.0 y programada en C/C++.

Según sus especificaciones utiliza árboles B-Tree para el almacenamiento, y se apoya en el software **dtSearch**[21] para la búsqueda full-text. Implementa la incorporación de índices para acelerar las consultas

2.4.2. Procesadores XPath/XQuery Estáticos

Los procesadores XPath/XQuery estáticos son un conjunto de aplicaciones cuyo objetivo es trabajar directamente sobre ficheros XML. No modifican el fichero y lo recorren con cada consulta.

[18] http://exist.sourceforge.net/.
[19] http://xml.apache.org/xindice/.
[20] http://modis.ispras.ru/sedna/.
[21] http://www.dtsearch.com.

Galax

Galax es una implementación open source de XQuery. El objetivo según sus autores es lograr que Galax llegue a ser una implementación completa de la familia de borradores de trabajo XQuery. Actualmente, Galax implementa muchas de las especificaciones del XML Query Working Group[22].

Galax está escrito sobre Objective Caml (un lenguaje de programación, estáticamente tipado, desarrollado en INRIA desde 1985). Sus tipos algebraicos y las funciones de alto nivel simplifican la manipulación simbólica que es central a la transformación de la consulta, análisis, y optimización que se requiere. Galax es razonablemente liviano (su imagen en Linux es de alrededor de 1.2 MB) y es muy portable (OCaml incluye Win32, Macintosh, y virtualmente todas las plataformas Unix-like) [FSC+03].

Saxon-B

Es un procesador *non-schema-aware*[23], y está disponible como producto open source[24]. Está diseñado para cumplir con el nivel de conformidad básico de XSLT 2.0, y el nivel equivalente de funcionalidad en XQuery 1.0.

Está desarrollado en Java, por lo que se ejecuta mediante un intérprete.

Qizx/open

Es una implementación Java open source de la especificación XML Query. Qizx/open sólo puede manipular documentos XML almacenados en memoria.

El proyecto original se extendió a **Qizx/db** bajo una modalidad de licencia cerrada (descrito previamente en la sección Bases de datos XML), y el desarrollo del original se abandonó, pero la última versión aún está disponible[25].

[22] http://www.galaxquery.org/.
[23] No considera el XML Schema como fuente de información útil para el análisis de las consultas.
[24] http://saxon.sourceforge.net/.
[25] http://www.xmlmind.com/qizx/qizxopen.shtml.

Qexo

Qexo 1.9.1[26] implementa en Java la mayoría de la especificación núcleo XQuery, y pasa más del 98 % de las pruebas de la suite para XQuery (XQuery Test Suite).

Implementa características para todos los ejes, módulos y serialización. Sin embargo, sólo algunos parámetros de la serialización son ajustables. No implementa la característica de importación de esquema, ni tampoco la validación de esquemas.

Qexo implementa algunos tipos estáticos ad-hoc, pero no implementa la característica Static Typing.

2.4.3. Comentarios sobre el Trabajo Relacionado

De acuerdo con el análisis hecho en [NLC06], se puede agregar lo siguiente al análisis del trabajo relacionado.

Con respecto al desempeño de la compresión en promedio, XMill muestra el mejor comportamiento, ya que logra un buen grado de compresión a una velocidad comparable con gzip y con bajo consumo constante de memoria. Si bien XMLPPM también alcanza un muy buen grado de compresión, su tiempo de compresión es prohibitivamente más largo que gzip, haciéndolo inviable para muchas aplicaciones prácticas.

XGrind y Xpress logran peor compresión que gzip, aunque con la ventaja de que pueden evaluar consultas sobre el contenido comprimido. Ambos compresores soportan calce exacto y calce de prefijos sobre la información comprimida, calce parcial y calce por rango sobre la información descomprimida, y los ejes XPath child y attribute; mientras que Xpress soporta también calce de rango sobre la información numérica comprimida, y el eje XPath descendant.

[26] http://www.gnu.org/software/qexo/.

Capítulo 3

Procedimiento

Cabe recordar que el objetivo de este trabajo es el desarrollo de un índice eficiente que permita almacenar documentación semi-estructurada en un repositorio, facilitando su consulta localmente y a la vez reduciendo el espacio de almacenamiento.

En este sentido, se describe a continuación la metodología utilizada para el logro de los objetivos específicos.

Por otro lado, para comprender el alcance práctico del trabajo realizado, en las secciones 3.2 y 3.3 se detallan el software y los equipos utilizados.

3.1. Metodología

De acuerdo a los requerimientos expresados en la sección 1.2.2, se definió la forma y estructura del almacenamiento, la implementación del índice invertido estructural, su adaptación al documento comprimido almacenado, la estructura del índice invertido de lenguaje natural, también adaptado, y finalmente, las estructuras auxiliares y sus características necesarias para soportar las consultas y la serialización de los resultados.

Cabe destacar que, en este nivel de programación de la solución, fue imperioso definir un método adecuado de comunicación entre las partes, de modo de poder desarrollar la aplicación sin perder funcionalidad debido a fallas en su interrelación. Este delicado proceso de diseño de interfaces es parte de la solución, puesto que no existía en XPN.

Una vez definido el conjunto de clases y módulos encargados de las funcionalidades del almacenamiento y las interfaces mediante las cuales éstos se comunican, se procedió a la implementación de la solución. El trabajo fue hecho en etapas sucesivas de modo que la aplicación fuese siempre funcional, pero agregando complejidad al índice y mejorando su desempeño en construcción, utilización del espacio, respuesta a las consultas y serialización de los resultados.

3.1.1. Requerimientos de Software

Para cumplir con los objetivos del proyecto fue necesario identificar los requerimientos en términos funcionales que permitiesen abordar de manera lógica los problemas que se plantearon. En este sentido, se debió especificar los objetivos en términos más detallados para diferenciar las componentes de software que sería necesario implementar.

Para almacenar un documento XML para XPN se debe implementar un autoíndice que soporte los siguientes requerimientos:

1. Disposición de la siguiente información para cada nodo destinada a apoyar un conjunto de operaciones de navegación eficiente sobre la estructura del documento:

   - Offset
   - Size
   - Label
   - Word Offset
   - Word Size
   - Dataguide Id
   - Document Id

   Cabe destacar que estos valores en la práctica no necesariamente se obtienen mediante almacenamiento explícito, sino vía un método a partir de información convenientemente almacenada.

2. Categorización por tipo de nodo: de elemento (etiqueta simple o atributo) o de texto, como lo requiere el modelo Proximal Nodes.

3. Compresión en Lenguaje Natural del texto original, esto es, en base a una secuencia de identificadores de palabra junto a un diccionario que las defina. Los identificadores son elegidos por frecuencia de modo que palabras más frecuentes ocupan menos espacio individualmente.

4. Descompresión local para poder construir resultados sin recurrir a la descompresión completa del documento.

5. Para el índice invertido estructural: navegación por elemento de modo eficiente. Esto significa la realización de saltos (*skipping*) de secciones completas de nodos que, de acuerdo a la estructura de árbol, no pueden formar parte de un resultado.

6. Para el índice de lenguaje natural: adaptación a las nuevas características del almacenamiento. Esto significa que, dada la existencia del diccionario de palabras, sólo es necesario almacenar ID's para identificar cada lista de ocurrencias.

3.2. Software Empleado

En el desarrollo de la aplicación en cuestión se ha utilizado una batería de software necesaria, tanto para apoyar las actividades mismas del desarrollo como para ser incluida en la misma.

3.2.1. Herramientas de Software

Subversion

Es un sistema de control de versiones que permite administrar la evolución del software. Como tal, provee herramientas para la solución de conflictos de versiones, de bitácora, y como respaldo del trabajo local. Por tanto se ha tenido acceso permanente a la generación de versiones a medida que se ha ido programando soluciones.

Uno de los objetivos de Subversion es el reemplazo de CVS en la comunidad open source[1].

XMark

Es una benchmark suite cuyo objetivo es ayudar a los usuarios y desarrolladores a estudiar las características de sus bases de datos XML[2]. Permite generar repositorios con tamaño parametrizado y contiene un conjunto de consultas orientadas a la medición de características específicas de los motores medidos (joins, serialización, etc.).

Han desarrollado una aplicación llamada **xmlgen** para generar los documentos de prueba mediante un parámetro para el tamaño. Los documentos modelan un sitio web de remates con contenido significativo. Puede recibir otros parámetros, como el factor para controlar la distribución de probabilidad, por ejemplo. Los documentos generados mantienen las características durante el escalamiento.

También han desarrollado un conjunto de veinte consultas XQuery destinadas a plantear los problemas más comunes a las bases de datos (joins, serialización, ordenamiento de la salida, reconstrucción, fragmentación, etc) [SWK+02, SWK+01]. A este conjunto de consultas se le denomina **XMark**. Una explicación detallada del conjunto de consultas se encuentra en [SWK+02].

XCheck

Es una plataforma de software open source[3] para ejecución automática de mediciones sobre procesadores XML (tanto XPath como XQuery) [AFMZ06].

La ejecución de una medición sobre diferentes aplicaciones toma mucho tiempo y usualmente genera gran cantidad de información en bruto. Más aún, la interpretación de los resultados de la evaluación es una tarea delicada y crucial. El objetivo principal de XCheck es automatizar algunas de las tareas en torno a la práctica del benchmarking sobre procesadores XML query (o motores de consulta) y, más importante, ayudar al usuario a mostrar

[1] http://subversion.tigris.org/.
[2] http://www.xml-benchmark.org/.
[3] http://ilps.science.uva.nl/Resources/XCheck/.

las debilidades de un procesador dado y comparar el desempeño de los diferentes motores sujetos a la evaluación.

XCheck se enfoca en la medición del desempeño, en oposición a los benchmarks de correctitud. Por lo tanto, las mediciones no necesitan especificar las salidas correctas. Sin embargo, XCheck ayuda a la detección de errores en las respuestas comparando el tamaño de los resultados obtenidos de diferentes motores.

Cada aplicación debiera ser capaz de entregar mediciones de tiempo de procesos internos (como tiempo de indexación, de compilación de la consulta y el tiempo de la consulta misma, etc), de modo de obtener resultados detallados del proceso.

Para realizar un experimento se especifica la interfaz de control de cada aplicación en un fichero XML que incluye su ruta (path). Se definen los parámetros para la medición (número de iteraciones, consultas implicadas, parámetros a medir, etc.), especificando qué motores participarán de la medición comparativa, y también con qué consultas.

Luego XCheck trabaja en dos fases: ejecución y análisis de los datos. En la fase de ejecución, se lleva a cabo la medición sobre los motores XML disponibles y se almacenan tanto los tiempos de ejecución medidos como los tiempos de ejecución que los propios motores reportan. Opcionalmente, se almacena los resultados de la consulta.

En la fase de análisis, XCheck elabora algunas estadísticas sobre los tiempos de ejecución obtenidos en la fase previa. Los resultados estadísticos son almacenados en formato XML y pueden ser llevados a un reporte en HTML con gráficos generados por el software libre **Gnuplot** en formato Postscript y PNG.

El conjunto de documentos y consultas que utiliza XCheck puede ser provisto por la suite XMark (ver sección 3.2.1) [AFMZ06].

Kdevelop

El Proyecto KDevelop surgió en 1998 con el fin de desarrollar un IDE (entorno de desarrollo integrado) fácil de usar para KDE[4]. Desde entonces, el IDE KDevelop está públicamente disponible bajo la GPL y soporta muchos lenguajes de programación.

[4] http://www.kdevelop.org/.

Proporciona diagramas de herencia de clases, indentación y formato de código, administración de plantillas, marcadores, inclusión de puntos de quiebre para depuración, herramientas de edición, configuraciones de compilación, vista de variables, de prototipos, etc, todo agrupado bajo el concepto de proyectos que se almacenan en carpetas integrando en ellas el control de versiones.

A través de él se puede administrar y ejecutar automake, ctags, valgrind, subversion, diff, find, doxygen, gdb, entre otros.

GraphViz

Graphviz es un software para visualización de grafos[5]. Funciona en base a una sintaxis muy sencilla que permite un variado conjunto de figuras, nodos, enlaces, textos, aristas, etc. Provee distintos programas para distintas disposiciones de los diagramas.

3.2.2. Bibliotecas

En una segunda categoría de software se encuentran las bibliotecas que apoyan el desarrollo y proveen soluciones a problemas de conocimiento común:

Boost C++ Libraries

Esta biblioteca provee una serie diversa de soluciones de software útiles para el desarrollo en C++[6]. En este caso se utilizaron las siguientes: herramienta de análisis de la entrada desde la consola para obtener los parámetros de manera fácil y definir así las opciones de ejecución del mismo (program-options); implementación de diversas clases para uso eficiente de *smart-pointers* (usada tanto para punteros constantes como variables); y herramientas para generación de gráficos y diagramas (para la depuración y análisis del árbol de consulta).

[5] http://www.graphviz.org/.
[6] http://www.boost.org/.

Soul

Es una biblioteca[7] open source desarrollada por Benjamin Piwowarski que provee las herramientas necesarias para el análisis de XQuery (parsing) de modo de construir el árbol de consulta. Está desarrollada en C++ y es muy eficiente.

Cumple con las recomendaciones del W3C para el lenguaje.

Tree Container of Kasper Peeters

Es una plantilla C++ que implementa un contenedor genérico al estilo STL para árboles n-arios. Es creada y mantenida por Kasper Peeters[8].

3.3. Equipos

Para las mediciones se utilizó una máquina con las siguientes características:

- CPU AMD Athlon(tm) XP, 1150 Mhz, cache size 512 KB.
- RAM 1GB DDR
- 80GB Maxtor 6Y080L0, ATA DISK drive, 7200 rpm, 2MB buffer size, UDMA/133 mode selected (HDD1)
- 40GB Western Digital WD400EB-00CPF0, ATA DISK drive, 5400 rpm, 2MB buffer size, UDMA/100 mode selected (HDD2)

El repositorio está ubicado en HDD1.

[7] http://developer.berlios.de/projects/soul/.
[8] http://www.aei.mpg.de/~peekas/tree/.

Capítulo 4

Resultados

De acuerdo a los objetivos planteados, se ha desarrollado un índice eficiente que permite almacenar documentación semi-estructurada en un repositorio, que facilita su consulta localmente y que a la vez reduce el espacio de almacenamiento.

En la sección 4.1 a continuación se detalla una descripción de los componentes desarrollados. Posteriormente, en la sección 4.2, se describe el funcionamiento de los componentes y finalmente, en la sección 4.3, se describen los resultados obtenidos.

4.1. Descripción de los Componentes

Dados los requerimientos enunciados en los objetivos (sección 1.2.2), se ha planteado la creación de cuatro componentes definidos encargados de funciones específicas:

1. Archivo comprimido con descompresión local del documento XML agregado.
2. Índice Invertido Estructural con funcionalidades de skipping.
3. Índice Invertido de Lenguaje Natural descartando palabras vacías.
4. Dataguide para funcionalidades de etiquetado, relaciones en el eje XPath Ascending / Descending y análisis de la estructura.

4.1.1. Archivo

La compresión del texto se realizó mediante una variante del algoritmo de Huffman orientado a palabras (generando bytes en vez de bits) [BFNP07], concretamente el End Tagged Dense Code visto en la Sección 2.1. De este modo se puede obtener descompresión local y buenos índices de compresión.

La interfaz de codificación se ha diseñado de modo de soportar la implementación de varios de los códigos densos. Así, se encapsula la codificación y se permite hacer comparaciones prácticas. Sin embargo, sabemos por [BFNP07] cuáles son los desempeños de cada tipo, así es que nos enfocamos en implementar ETDC.

Para la codificación de las palabras y los separadores se ha hecho uso del modelo de palabras sin espacio (spaceless word model [dNZBY98]) y se han extraído algunas de las técnicas descritas en [BFN+08]. La diferencia con ésta última propuesta es que en nuestro caso no se mantendrá el invariante de intercalar un separador por cada palabra, sino que se marcará en una secuencia de bits el tipo de token a codificar (palabra o separador). Además, los stopwords se incorporaron como separadores, permitiendo así hacer búsqueda de frases haciendo caso omiso de ellos.

Para el almacenamiento se necesita mantener: dos diccionarios, uno para las palabras ($Dict_{WRD}$) y otro para los separadores ($Dict_{SEP}$); una secuencia de bits para marcar el tipo de token (BM_{TOKEN}), palabra o separador; y la secuencia de palabras codificada con ETDC almacenada en un fichero ($file_{ETDC}$).

Cabe destacar que al codificar una colección, el sistema la tratará como si fuese un sólo documento XML equivalente a la concatenación de la colección. Por esta razón se agrega un nodo ficticio al conjunto que representa un documento en el nivel más alto de la jerarquía. Así, el segmento correspondiente a ese nodo es el documento comprimido correspondiente. La justificación de este enfoque es la capacidad de manejar colecciones de gran volumen de documentos manteniendo la eficiencia en el acceso a disco.

4.1.2. Índice Invertido Estructural

El índice invertido estructural debe soportar las siguientes funcionalidades:

1. Recorrer una lista de nodos de una misma etiqueta ordenados en orden Pre.
2. Recorrer una lista de nodos de una misma etiqueta ordenados en orden Post.
3. Obtener el siguiente nodo con una etiqueta determinada a partir de una posición dada, desde cualquiera de las dos listas anteriores.
4. Obtener el segmento de palabra de cualquier nodo.
5. Obtener el segmento del fichero comprimido correspondiente al nodo.

Acceso Secuencial Filtrado por Etiqueta

La estructura apropiada para navegar por etiqueta es la secuencia de símbolos provista de estructuras auxiliares para Rank y Select. De este modo, con una sola estructura compacta podremos recorrer todos los nodos de acuerdo a una etiqueta usando como caja negra la implementación de Wavelet Tree presentada en [CN08]. Esta estructura es implementada sobre secuencias binarias en base a la propuesta de Raman Raman y Rao [RRR02]. Esta última estructura también será utilizada en los casos que se requieran secuencias de bits provistas de Rank y Select.

La ventaja más notable de los Wavelet trees es que accedemos en tiempo $O(\log \sigma)$, con σ el tamaño del alfabeto de etiquetas, al siguiente nodo etiquetado con una simple llamada compuesta de Rank y Select:

$$select_{PRE}(A, rank_{PRE}(A, i) + 1) \tag{4.1}$$

donde A es la etiqueta, e i es el offset (orden pre) del nodo de contexto.

Se eligió esta implementación porque es eficiente incluso cuando el tamaño del alfabeto de etiquetas es comparable al número de nodos etiquetados, y porque comprime la secuencia a orden 0.

De este modo, mediante la secuencia de etiquetas accedemos a posiciones que corresponden al orden Pre del nodo en el árbol. Sin embargo, para producir intersección completa como vimos en la sección 2.3.3, también es necesario proveer recorrido en orden Post. Por esta razón, se propone incluir otra secuencia de etiquetas en Post orden, también dotada de Rank y Select.

Llamaremos a estas dos estructuras **Wavelet Tree para orden Pre** (wt_{PRE}) y **Wavelet Tree para orden Post** (wt_{POST}). Con ellas se puede satisfacer los tres primeros requerimientos del índice invertido estructural.

Obtención de Segmentos

Para la realización de las consultas de matching de texto, es necesario conocer el segmento de palabra de cada nodo. Esto es, en qué número de palabra del documento empieza el el nodo, y en qué número de palabra termina. El índice invertido de palabras indexa las posiciones en el orden de documento de cada ocurrencia, por tanto un nodo contiene o no a una determinada palabra, si el offset de la palabra pertenece o no a su segmento correspondiente, y contiene o no a una frase, si contiene o no al segmento que la representa.

El cálculo de los segmentos también es necesario para implementar las operaciones de relación de Proximal Nodes.

Mediante una estructura para representar Suma de Prefijos (ver sección 2.2.5) se puede representar la función de transformación desde orden de documento de nodo a orden de documento de palabra (offset de palabra) y vice versa.

La estructura almacena las diferencias incrementales de offset de palabra que se producen entre la posición pre de un nodo y la del siguiente. Por tanto, la suma agregada de tantas diferencias de offset como determine un valor de Pre, será el offset de ese nodo en la secuencia de palabras, y la posición de término será la suma de tantas diferencias de offset como el valor dado por $Pre + Size$ del nodo, donde $Size$ representa al número de nodos descendientes del nodo dado. Por ende la aplicación deberá recuperar ése segmento para determinar el segmento de palabra que representa.

Para recuperar el contenido de un nodo dado es necesario conocer los offsets del fichero comprimido correspondientes al segmento que representa el nodo, esto es la posición donde empieza el primer código ETDC correspondiente al nodo, y la posición del último código correspondiente. Sin embargo, para transformar a offset de fichero comprimido no se necesitó la suma de prefijos porque los offsets son estrictamente crecientes (no hay códigos de tamaño cero) y no es necesario calcular el inverso (offset de palabra a partir de offset de fichero). Por ende, se marcó un bit por cada palabra en la posición donde comienza su código ETDC. Esta secuencia tiene tantos bits como bytes tiene el fichero comprimido.

La consulta sobre esta secuencia es muy similar a la anterior, pero como sólo se usa para imprimir segmentos, lo único que se hace es recuperar el offset desde donde comienza la decodificación, es decir, sólo es necesario hacer Select. Desde ahí en adelante se cuenta el número de palabras decodificadas hasta completar el tamaño en palabras del nodo a imprimir, lo cual es un dato conocido.

Estas estructuras son implementadas mediante secuencias de bits provistas de Rank y Select y comprimidas a orden 0. La implementación fue hecha por Francisco Claude [CN08] y corresponde a la estructura propuesta por Raman, Raman y Rao en [RRR02]. Aquí la utilizamos como caja negra para nuestro propósito.

A la implementación de sumas parciales para obtener el offset de palabra la llamaremos **Word Offset Partial Sum** (sum_{WRD}) y a aquella para obtener el offset en el fichero comprimido la llamaremos **File Offset Bitmap** (BM_{OFF}). Con estas dos estructuras se satisfacen los dos últimos requerimientos del índice invertido estructural.

Las estructuras necesarias para determinar el tamaño de un nodo las veremos más adelante, en el análisis hecho en torno a la ecuación 4.3.

4.1.3. Dataguide

El dataguide debe soportar las siguientes funcionalidades:

1. Definir un identificador (`StructId`) para cada etiqueta XML que aparezca alguna vez en la colección. El valor debe ser único para cada posición estructural, es decir, secuencia

de etiquetas hacia la raíz.

2. El nodo raíz del dataguide será siempre el nodo `-DOCUMENT`[1].

3. Determinar si dos nodos del dataguide satisfacen la relación ascendiente/descendiente.

4. Determinar si dos nodos del dataguide satisfacen la relación padre/hijo en el árbol.

Determinación del `StructId`

Para identificar a cada nodo del dataguide se implementó un método que analiza la relación padre/hijo entre nodos, a medida que avanza el parsing XML. De este modo se puede asignar un `StructId` único a cada uno.

La relación se determina por el par que forman la etiqueta XML del nodo y el `StructId` del nodo padre (obtenido previamente). La unicidad se demuestra por inducción si se observa que el nodo raíz (`-DOCUMENT`) siempre es el primero que ingresa al dataguide.

La estructura para lograr esto es un mapa asociativo entre el par formado por el par que constituye la etiqueta XML del nodo y el *StructId* del padre, y el *StructId* del nodo. A esta estructura la llamaremos **StructIdMap** y tiene como objetivo determinar si la posición estructural fue detectada previamente.

Además, es necesario un diccionario que almacene las etiquetas XML presentes en la colección. Cada nodo del dataguide posee un puntero a su etiqueta correspondiente en el diccionario. A esta estructura la llamaremos DG_{DICT}.

El algoritmo de construcción del dataguide es muy similar al descrito en [ANd07a] para detectar repeticiones en la estructura, pero más simple. En nuestro caso no es necesario recurrir a una firma digital porque aún en casos extremos la composición `StructId` y etiqueta se puede escribir en dos palabras de máquina.

[1] En XML están prohibidas las etiquetas que comiencen con el símbolo '-', por tanto es un buen mecanismo para agregar etiquetas ficticias al diccionario del dataguide, lo mismo se hace con los nodos de texto, en cuyo caso la etiqueta usada es `-TEXT`.

Relaciones entre Nodos del Dataguide

Para hacer consultas entre nodos del dataguide se almacenará un arreglo ($array_{SID}$) cuyos objetos contienen tres campos:

1. `LabelKey`
2. `Level`
3. `To`

El campo LabelKey corresponde a la llave correspondiente a la etiqueta XML del nodo, obtenida del diccionario del dataguide. `Level` corresponde al nivel del nodo en el árbol del dataguide que a su vez corresponde al nivel de todos los nodos XML cuya etiqueta es representada en este objeto del dataguide, y el campo `To` corresponde a la posición en pre orden del primer siguiente nodo no descendiente del actual (*following* en la nomenclatura XPath).

A cada objeto del arreglo se accede usando el *StructId* que es a su vez el valor de Pre del árbol que constituye el dataguide. Es decir el tamaño del arreglo es igual al tamaño del alfabeto del dataguide, y cada nodo del árbol se identifica con el *StructId* que representa.

La determinación de inclusión se deduce de comparar los valores de `Pre` y `To` de ambos nodos como se vio en la sección 2.1.4

Cabe destacar que en el dominio del dataguide, el orden pre entre nodos descendientes de un mismo nodo padre es arbitrario pues no interesa más que las relaciones en el eje XPath ascendiente/descendiente. Por ende sólo interesa que exista coherencia entre la definición de los valores de `StructId` y la posición efectiva de los nodos en el árbol. Sin embargo, en la práctica el dataguide contiene todos los caminos XPath posibles en la colección por orden de aparición en ella, y esa va a ser la forma del árbol que se almacenará.

Extensión del Conjunto de Etiquetas

Si se asignan nuevas etiquetas a aquellas etiquetas XML que se repitan en distintas posiciones del dataguide, se pueden optimizar las consultas XPath sobre la colección mediante el descarte de un conjunto grande de nodos.

XPN provee un mecanismo sencillo de acuerdo a Proximal Nodes para producir unión de flujos mediante el operador `Union`, por tanto éste es el mecanismo utilizado para la integración de los flujos provenientes de una misma etiqueta XML con diferente `StructId`.

Una secuencia de nodos de determinada etiqueta ordenada por pre, en el caso general, no satisface el post orden en sus elementos. Basta con que exista un par de nodos anidados con igual etiqueta para que el ascendiente, cuyo valor de pre es menor, aparezca posteriormente en la lista ordenada por post porque su segmento termina después que todos sus descendientes. Gracias a la extensión del conjunto de etiquetas se elimina la existencia de etiquetas de un mismo tipo anidadas, lo cual tiene trascendencia en la satisfacción de los requerimientos del índice invertido estructural descritos en la sección 2.3.3.

Dichas consultas se expresan como la obtención del primer nodo con etiqueta A cuyo valor de pre sea superior a i y que al mismo tiempo tenga un valor de post al menos igual a j.

Gracias a la inexistencia de anidamientos en el conjunto de nodos determinado por cada etiqueta, se puede obtener para cada nodo de la secuencia en orden pre, directamente la posición que ocupa en la secuencia ordenada por post, y vice versa, porque se verifica la siguiente propiedad:

$$rank_{PRE}(A, pre_x) = rank_{POST}(A, post_x) \quad \forall x \tag{4.2}$$

Es decir, la secuencia de nodos de etiqueta A está ordenada por pre y al mismo tiempo por post, obviando la necesidad de una secuencia de paréntesis balanceados para sincronizar ambas secuencias en torno a un nodo dado x.

Luego, sólo se necesita ejecutar $select_{PRE}(A, max(rank_{PRE}(A, i) + 1, rank_{POST}(A, j)))$ para satisfacer un requerimiento como el descrito.

Una característica importante derivada de la ecuación 4.2, es que se puede calcular el tamaño de un nodo ($Size$) a partir de su pre orden utilizando la siguiente relación:

$$Size = Post + Level - Pre + 1 \tag{4.3}$$

El valor de *Post* se calcula a partir de *Pre* haciendo Rank sobre el **Wavelet Tree para orden Pre** y luego haciendo Select sobre el **Wavelet Tree para orden Post**, y el valor de *Level* se obtiene del dataguide usando el *StructId*. Esto provee la pieza faltante al índice invertido estructural (ver sección 4.1.2).

Por último, es necesario destacar que los nodos de texto también son agregados al dataguide, lo cual optimiza las consultas sobre contenido porque, en general, están asociadas al contexto de la etiqueta que los contiene. La idea es reducir el conjunto de `StructId` potenciales restringiendo el conjunto de segmentos a intersectar con las ocurrencias reportadas por el índice invertido de palabras. Cuando la consulta de contenido no esté contextualizada estructuralmente, no se logra ninguna optimización, pero por cada contexto que se agregue como expresión XPath a la consulta, se logra una importante reducción del conjunto potencial.

Observemos que agregar los nodos de texto, en el peor caso, duplica el vocabulario de etiquetas, lo cual sigue siendo asintóticamente favorable para la manipulación del dataguide y el funcionamiento de los **Wavelet Tree para orden Pre y Post**, pues en los casos que nos interesa abordar, el vocabulario de etiquetas es muchísimo menor que el número de nodos de la colección.

4.1.4. Índice Invertido de Lenguaje Natural

El índice invertido de lenguaje natural hacía referencia al documento original. Con la incorporación del documento comprimido al índice, se hicieron varias optimizaciones a esta componente.

La primera de ellas es que el vocabulario de palabras indexadas es un subconjunto del vocabulario de la compresión, por ende lo que se hizo es almacenar los ID's del vocabulario de la compresión, en vez del vocabulario de palabras indexadas. Sin embargo, dado que las palabras indexadas han sido llevadas a una forma canónica mediante stemming, el ID en realidad representa a una de las palabras de la clase definida por su stem, y por ende el ID pasará a representar también a dicha clase. Esto significa que para buscar una palabra en el índice invertido, primero es necesario llevarla a su forma canónica. De este modo, al realizar

la búsqueda binaria descrita en la sección 2.3.3, en las comparaciones es necesario recuperar cada palabra correspondiente a cada `WordID`, y llevarla también a su forma canónica antes de compararla con la de búsqueda. Los resultados serán todas las ocurrencias de palabras indexadas con igual stem a la buscada.

El impacto directo de ésta medida es la eliminación del fichero VOC y un cambio en el fichero OFF, que ya no almacena el offset del fichero VOC donde se almacenaba la palabra, sino que directamente el `WordID` del stem. Esto se traduce en una mayor compresión global, con disminución de acceso a disco, más un castigo en desempeño en memoria principal debido a la necesidad de cálculo del stem de cada comparación.

La segunda optimización aplica sobre los índices parciales. Al final del proceso de indexación, los ficheros OFF de cada índice parcial tienen dos campos extras que sirven para realizar el proceso de mezcla del índice en uno sólo. Una vez que la mezcla era realizada, se volvía a recorrer el fichero OFF completo reescribiéndolo sin los dos campos auxiliares. Por tanto, estos dos campos, de tamaño fijo, serán almacenados en un fichero aparte llamado AUX, de modo que al finalizar la mezcla baste con borrar el fichero AUX resultante junto con los ficheros parciales, para no volver a reescribir el fichero OFF.

La tercera optimización consiste en comprimir el fichero OFF. La búsqueda binaria realiza un acceso aleatorio al arreglo ordenado que representa este fichero, pero mantener su contenido en disco se traduce en sucesivas llamadas al sistema para que las recupere. Aún cuando el sistema operativo realiza *caching* del fichero, el poder comprimirlo renunciando a su distribución homogénea y subirlo de una sola vez entero a memoria antes de comenzar las comparaciones, mejora el rendimiento global cuando el índice es consultado recurrentemente.

Una última optimización dice relación con la recuperación del vocabulario de palabras. Cuando la aplicación requiere recuperar una palabra, ya sea para serialización o para comparar su stem en las operaciones de matching, la aplicación carga el vocabulario completo en memoria principal. Ésto se traduce en un considerable costo temporal cuando se requiere un conjunto pequeño de ellas, y más aún cuando se busca una palabra que no existe. La solución propuesta es segmentar el vocabulario (que está ordenado lexicográficamente) en bloques,

y tomar la primera palabra de cada uno para mantenerla en un conjunto aparte (también ordenado lexicográficamente) que hace la función de muestra. De este modo, al hacer búsqueda binaria sobre la muestra se puede determinar qué bloque cargar a memoria principal para continuar la búsqueda binaria. Esto permite cargar un conjunto más restringido del vocabulario (la muestra que debe tener información para recuperar los bloques) y el bloque que finalmente resulte ser potencial.

4.2. Construcción y Consulta

A continuación se detalla la construcción de cada uno de los componentes propuestos previamente.

4.2.1. Archivo

Para crear un fichero comprimido con ETDC es necesario hacer dos pasadas sobre el texto: una para obtener el vocabulario y crear el modelo, y otra para almacenar en el fichero comprimido los códigos que este determina.

El modelo se crea a partir del número de ocurrencias de cada palabra haciendo un ranking de mayor a menor. Palabras con muchas ocurrencias tendrán asociados los códigos más cortos del ETDC y vice versa, al estilo del código de Huffman en el cual se inspira.

En nuestro caso, para evitar pasar dos veces por el parsing XML, se almacenará un fichero semi-comprimido consistente en el almacenamiento de los códigos ETDC temporales obtenidos por orden de asignación. La idea es que cada vez que se agrega una palabra nueva al vocabulario, se le asigna el código ETDC disponible siguiente, y este se almacena en disco. De este modo, se reduce el tamaño de la colección XML hasta alrededor de un 40 % en un peor caso, lo cual compensa con creces el tener que decodificar el primer ETDC para la segunda pasada, pues la transferencia total desde disco se reduce considerablemente. Además, se evita el sobrecosto de usar por segunda vez el parser XML.

Por otro lado, el sistema está diseñado para detectar stopwords en una función compartida

con el índice invertido de palabras.

La lista de stopwords se obtiene a partir de un fichero de referencia que consiste en un fichero con palabras separadas por salto de línea. Si un fichero con stopwords es pasado como parámetro a la aplicación mediante la línea de comandos, la lista de palabras se carga en un trie, llamado **trie de stopwords**.

Cada vez que se detecta una palabra en la colección, una copia de la palabra es canonizada para obtener su stemming (en nuestra implementación esto consiste sólo en convertir a Case Insensitive), y luego buscada en el trie de stopwords (que también contiene palabras canonizadas). Si se encuentra, entonces la original se busca en el diccionario de separadores. Si es encontrada ahí, se almacena en el fichero temporal con su código ETDC asignado. Si el separador es nuevo, se agrega en el diccionario, y se le asigna el siguiente código ETDC disponible para escribirlo en el fichero temporal.

Si la palabra canonizada no se encuentra en el trie de stopwords, entonces la original es buscada en el diccionario de palabras. Si se encuentra ahí, se almacena en el fichero temporal con su código ETDC asignado. Si la palabra es nueva, se agrega en el diccionario, y se le asigna el siguiente código ETDC disponible, para escribirlo en el fichero temporal.

Para cada uno de los dos diccionarios de tokens se calcula un modelo independiente que permite obtener dos rankings de frecuencia. Por esta razón, para cada token codificado (palabra o separador), se marca un bit en una secuencia de bits cuya función es diferenciar su tipo. Si el token es una palabra, se marca un bit activo, si en cambio es un separador, se marca un bit apagado.

Cabe destacar que ambos modelos (de separadores y de palabras) hacen uso de los códigos ETDC temporales. La función de la secuencia de bits es permitir diferenciar a qué modelo pertenece el código ETDC correspondiente que se ha escrito en el fichero.

Para asegurar que el espacio asignado a la secuencia de bits es suficiente, se chequea su tamaño en cada marcado. En caso de ser insuficiente, se reasigna a un área de memoria del doble de tamaño para garantizar complejidad espacial y temporal líneal.

Una vez que se ha terminado la primera pasada, se crea el ranking de palabras, luego se

asignan los códigos según cada modelo, y finalmente se calcula una permutación que traduce el código asignado en la primera pasada al definitivo.

Con las permutaciones se hace la (segunda) pasada sobre el fichero temporal, y se escriben los códigos definitivos en un fichero también definitivo.

Por último se almacenan los vocabularios de palabras y separadores. Cada token se almacena como secuencia de caracteres precedido de su longitud codificada con ETDC, sólo para reducir el número de bytes escritos a un tamaño cercano a los significativos para cada entero. El almacenamiento de la longitud facilita la tarea de asignación de memoria durante la recuperación y permite almacenar palabras de cualquier longitud.

4.2.2. Dataguide

Durante la construcción del dataguide, se utiliza una estructura para manipular el árbol dinámicamente. Dicha estructura es un contenedor construido al estilo de la biblioteca estándar de C++ (ver sección 3.2.2) que permite recorrer la estructura en pre orden e insertar nodos en posiciones arbitrarias.

Cada vez que se inserta un nuevo nodo al dataguide, se incrementa el alfabeto de `StructId`'s de modo que siempre el último nodo agregado al dataguide tiene el valor más alto del alfabeto. En el algoritmo 3 se describe el proceso de obtención de `StructId`.

Cuando el parsing reporta un evento que corresponde a un comienzo de nodo, es necesario obtener el `StructId` del nodo, y luego agregarlo a una pila del parsing, de modo que cuando toque procesar el evento de término del nodo, se extraiga el *StructId* de la pila. El objetivo de la pila es proveer el *StructId* del nodo padre. Los nodos de contenido no se agregan a la pila porque no tienen descendientes.

No siempre se crea un StructID cuando un nodo XML lo solicita, porque la ubicación estructural puede ser redundante. En ese caso, se retorna el `StructId` ya asignado a esa posición.

Este modo de inserción no mantiene la coherencia entre el orden pre de cada nodo del dataguide, con el StructId correspondiente, como se pretende según el diseño. Por esta razón,

una vez que se ha terminado de crear el dataguide, se recorre el árbol del dataguide en pre orden, reasignando el StructId de cada nodo en una permutación de valores que crea de nuevo el arreglo de objetos descriptores de cada nodo, y el mapa asociativo de relaciones. De este modo, se evita almacenar el StructId de cada nodo del dataguide, pues el valor estará dado por el orden pre y sólo será necesario almacenar la forma del árbol en una secuencia de paréntesis balanceados.

4.2.3. Índice Invertido Estructural

Creación del Índice

Primera pasada Debido a que los códigos ETDC resultantes no son los definitivos después de la primera pasada sobre la colección, no es posible calcular inmediatamente la lista de offsets que delimita a cada código. Por ende tampoco se puede calcular el offset de fichero de cada nodo hasta que se realice la segunda pasada sobre el temporal. Sin embargo, sí es posible identificar el offset de palabra en relación con el pre orden de los nodos.

Por otro lado, tampoco podemos crear las listas de pre orden y post orden en la primera pasada porque los StructId tampoco son los definitivos. Por esta razón, para poder emular el proceso de parsing de XML en la segunda pasada, primero es necesario almacenar la forma del árbol XML de la colección.

Por esta razón, se crea una secuencia de bits auxiliar que almacena la secuencia de paréntesis balanceados correspondiente al árbol. Cada vez que mediante el análisis de un documento XML (parsing) se gatille el evento de un nuevo nodo, se marca un bit prendido en ella, y se avanza a la siguiente posición. En el evento de final de nodo en cambio, se marca un bit apagado.

Junto con esto, es necesario almacenar las etiquetas correspondientes, razón por la cual se almacena la secuencia de llaves del diccionario de etiquetas del dataguide, que se obtienen en cada evento de comienzo de nodo. Un caso especial es el del nodo de documento, que es agregado cada vez que la aplicación procesa un nuevo fichero con el parser.

Para asegurar que el espacio asignado para las secuencias es suficiente, se mantiene un

contador que se incrementa cada vez que se agrega un valor en ella. Así, se chequea su capacidad con cada ingreso, y si no alcanza se reasigna un espacio de memoria del doble del tamaño previo. Esto garantiza que la complejidad temporal y espacial total de la asignación de memoria es líneal con el tamaño de la secuencia correspondiente. Esto mismo se hace con las secuencias de orden pre y post en la segunda pasada y con todas las secuencias de bits utilizadas.

Junto con agregar un StructId a la secuencia, es necesario marcar el offset de palabra del nodo correspondiente para luego crear las sumas de prefijos. La suma de prefijos para el offset de fichero se crea en la segunda pasada.

Para las sumas se mantiene una secuencia de bits que va reasignando su memoria a medida que lo requieran, como ya se ha dicho.

En ellas se almacenan únicamente bits apagados salvo las posiciones correspondientes a $nodePre + nodeOffset$ de cada nodo que se agrega. Es decir, almacena secuencias de nodos apagados de longitud de la diferencia de valores de offset, terminadas por un bit prendido.

Con esta codificación, se puede obtener el offset de cada nodo a partir del valor de pre, y vice-versa, según se describe en la sección 2.2.5 en las ecuaciones 2.6 y 2.7.

Segunda pasada Una vez concluida la primera pasada, tenemos la secuencia de bits con offset de palabras, la secuencia de bits de paréntesis balanceados, y el dataguide con los ID's definitivos.

En la segunda pasada, se va leyendo sincrónica y secuencialmente la secuencia de bits de paréntesis balanceados. El primer nodo de la colección siempre es un nodo de documento y corresponde a un paréntesis que abre (`(`). Su etiqueta correspondiente es `-DOCUMENT`, y por ser el primer nodo de la colección, el primer offset representado por la secuencia de bits para las palabras es cero, por tanto tiene el primer bit prendido. Esto indica que se debe marcar el primer bit de la secuencia de bits de offset de fichero, pues el nodo comienza junto con la primera palabra.

Con la etiqueta `-DOCUMENT` y un StructId del nodo padre indefinido, obtenemos el primer StructId como ya hemos visto anteriormente. Lo agregamos a la secuencia de pre orden, y

luego a una pila para proveer el StructId del nodo padre.

Una vez hecho esto, leemos secuencialmente la secuencia de bits de offset de palabra hasta el siguiente bit prendido. Por cada bit apagado, se lee una palabra del fichero temporal de ETDC, se calcula el código definitivo con la permutación que indique la secuencia de bits para el tipo de token, y se escribe al fichero final actualizando un offset global. Este offset global es la referencia para crear la secuencia de bits de offset de fichero que se marca cada vez que se esté frente a un comienzo de nodo. Cada bit prendido en la secuencia de offset corresponde a un comienzo de nodo.

Después se avanza en la secuencia de paréntesis balanceados. Por cada paréntesis que cierra, se saca un StructId de la pila y se pone en la secuencia de post orden. Cuando se ha llegado a un paréntesis que abre, se toma el valor actual de la secuencia de llaves de diccionario del dataguide, se recupera la etiqueta correspondiente, y se calcula el StructId del nodo actual. Éste se agrega a la secuencia de pre orden, y luego se inserta a la pila para proveer el StructId del siguiente nodo padre.

Por último, se marca la secuencia de bits de offset de fichero con la suma del pre orden y el offset global de acuerdo al procedimiento que se ha indicado para producir sumas parciales.

Y así sucesivamente, el proceso se detalla en el algoritmo 8.

Durante la creación del índice, las secuencias son creadas en un arreglo de memoria contigua que al final son pasados como parámetro al constructor del Wavelet Tree, junto con el tamaño del alfabeto del dataguide, y luego guardadas a disco.

Las secuencias de bits a su vez son pasadas como parámetro al constructor de la secuencia de bits provista de rank y select, y luego guardadas también en disco.

Cabe destacar que los constructores de las secuencias de bits con rank y select de la implementación de F. Claude (durante la creación de la secuencia y durante la recuperación desde fichero), necesitan la instancia de una estructura específica que puede ser compartida por todas las secuencias. Por tanto, la aplicación se encarga de instanciar y eliminar esta estructura, siguiendo el patrón de diseño Singleton, para aprovechar la memoria disponible. Los wavelet trees también necesitan esta estructura porque internamente utilizan la misma

implementación de secuencias de bits.

Consulta sobre el Índice

Para consultar el índice, se recuperan desde disco las cuatro estructuras correspondientes (wt_{PRE}, wt_{POST}, sum_{WRD} y sum_{OFF}) mediante constructores que reciben ficheros para lectura. En el algoritmo 1 se especifica cómo se recuperan los valores de offset en el caso de la suma de prefijos para los segmentos de palabra.

En el algoritmo 7 se especifica el método para imprimir un nodo XML a partir de su *StructId* y su segmento de palabra, que es la forma como los identifica XPN. El algoritmo utiliza otros métodos descritos en los algoritmos 4, 5 y 6.

4.2.4. Índice Invertido de Lenguaje Natural

El método para determinar si una palabra debe ser indexada o no depende del resultado de la búsqueda en un trie de stopwords según se ha descrito previamente.

El método para construir el índice es el mismo que en la sección 2.3.3 salvo las modificaciones descritas en la sección 4.2.1.

Para obtener la muestra que permite consultar el vocabulario de modo segmentado, primero se obtiene el vocabulario completo ordenado lexicográficamente. Luego se determina la frecuencia de muestreo, y al momento de almacenarlo en disco, se va guardando secuencialmente y se va extrayendo palabras a intervalos equivalentes, observando la posición en el archivo del vocabulario como posición de offset de bloque.

Algoritmo 1 Obtención del segmento de palabras a partir de un nodo.

Require: wt_{PRE} // pre order wavelet tree
Require: wt_{POST} // post order wavelet tree
Require: dg // dataguide
Require: sum_{WRD} // sumas parciales de offset de palabra
Require: $nodePre$ // valor pre orden del nodo

1: // obtiene el comienzo del segmento a partir del pre orden
2: $nodeFrom \leftarrow sum_{WRD}.select(nodePre + 1) - nodePre$
3: // obtiene la posición del nodo en la secuencia de su etiqueta
4: $LabelNodePosition \leftarrow wt_{PRE}.rank(nodeStructId, nodePre)$
5: // obtiene el orden post del nodo
6: $nodePost \leftarrow wt_{POST}.select(nodeStructId, LabelNodePosition)$
7: // obtiene el nivel del nodo
8: $nodeLevel \leftarrow dg.getLevel(nodeStructId)$
9: // obtiene el valor de pre del primer siguiente nodo no descendiente (following)
10: $nodeFollowing \leftarrow nodePost + nodeLevel + 1$
11: // obtiene el final del segmento a partir del nodo following
12: $nodeTo = sum_{WRD}.select(nodeFollowing + 1) - nodeFollowing$
13: **return** $[nodeFrom, nodeTo[$

Algoritmo 2 addDataguideNode($nodeLabelID$,$parentStructId$).

Require: $nodeLabelID$ // identificador de la etiqueta XML del nodo
Require: $parentStructId$ // `StructId` del nodo padre XML
Require: dg // dataguide
Require: $array_{SID}$ // arreglo de nodos del dataguide referenciados por `StructId`
Require: dg_{MAP} // mapa asociativo de pares ((StructId, XML_TAG), StructId)
Require: dg_{TREE} // árbol dinámico del dataguide
Ensure: $dg_{MAP}.size() = array_{SID}.size() = dg_{TREE}.size()$

1: $nodeStructId \leftarrow dg.getNextID()$
2: $dg_{MAP}.insert(((nodeLabelID, parentStructId), nodeStructId))$
3: $parentDGNode \leftarrow array_{SID}.at(parentStructId)$
4: $newDGNode \leftarrow (nodeLabelID, nodeStructId, parentDGNode.level + 1)$
5: $dg_{TREE}.append\_child(parentNode, newDGNode))$
6: $array_{SID}.push\_back(newDGNode)$
7: **return** $\langle nodeStructId \rangle$;

Algoritmo 3 obtainStructId($XMLnodeLabel$,$parentStructId$).

Require: $XMLnodeLabel$ // etiqueta XML del nodo
Require: $parentStructId$ // `StructId` del nodo padre XML
Require: dg // dataguide
Require: $array_{SID}$ // arreglo de nodos del dataguide referenciados por `StructId`
Require: dg_{MAP} // mapa asociativo de pares ((StructId, XML_TAG), StructId)
Require: dg_{TREE} // árbol dinámico del dataguide
Ensure: $dg_{MAP}.size() = array_{SID}.size() = dg_{TREE}.size()$

1: $dg_{DICT}.search(XMLnodeLabel)$
2: **if** $dg_{DICT}.\_FOUND$ **then**
3: $nodeLabelID \leftarrow dg_{DICT}.getKey(XMLnodeLabel)$
4: $dg_{MAP}.find((nodeLabelID, parentStructId))$
5: **if** $dg_{MAP}.\_FOUND$ **then**
6: $nodeStructId \leftarrow dg_{MAP}.\_FOUND$
7: **else**
8: $nodeStructId \leftarrow addDataguideNode(nodeLabelID, parentStructId)$
9: **end if**
10: **else**
11: $nodeLabelID \leftarrow dg_{DICT}.addKey(XMLnodeLabel)$
12: $nodeStructId \leftarrow addDataguideNode(nodeLabelID, parentStructId)$
13: **end if**
14: **return** $\langle nodeStructId \rangle$;

Algoritmo 4 printOpenLabel($nodePre$).

Require: $stack_{PRINT}$ // pila con los nodos abiertos
Require: dg_{DICT} // diccionario de etiquetas XML
Require: wt_{PRE} // pre order wavelet tree
Require: wt_{POST} // post order wavelet tree
Require: dg // dataguide

1: $nodeStructId \leftarrow wt_{PRE}.access(nodePre + 1)$
2: **if** $openedHeader \wedge \neg dg_{DICT}.isAttributeLabel(nodeStructId)$ **then**
3: **print** ">"
4: $openedHeader \leftarrow$ **false**
5: **end if**
6: **if** $dg_{DICT}.isAttributeLabel(nodeStructId)$ **then**
7: // la etiqueta corresponde a un XMLAttribute
8: **if** $openedHeader$ **then**
9: **print** "␣" + $dg_{DICT}.getLabelName(nodeStructId)$ + "'"
10: **end if**
11: **else**
12: // la etiqueta corresponde a un XMLElement
13: **print** "<" + $dg_{DICT}.getLabelName(nodeStructId)$
14: $openedHeader \leftarrow$ **true**
15: // obtiene la posición del nodo en la secuencia de su etiqueta
16: $LabelNodePosition \leftarrow wt_{PRE}.rank(nodeStructId, nodePre)$
17: // obtiene el orden post del nodo
18: $nodePost \leftarrow wt_{POST}.select(nodeStructId, LabelNodePosition)$
19: // obtiene el nivel del nodo
20: $nodeLevel \leftarrow dg.getLevel(nodeStructId)$
21: // obtiene el valor de pre del primer siguiente nodo no descendiente (following)
22: $nodeFollowing \leftarrow nodePost + nodeLevel + 1$
23: $stack_{PRINT}.push((nodeStructId, nodeFollowing))$
24: **end if**

Algoritmo 5 printCloseLabel(*nodeFollowing*).

Require: $stack_{PRINT}$ // pila con los nodos abiertos
Require: dg_{DICT} // diccionario de etiquetas XML
1: **while** $stack_{PRINT}.isNotEmpty() \wedge Stack.top.second = nodeFollowing$ **do**
2: **if** $dg_{DICT}.isAttributeLabel(stack_{PRINT}.top.first)$ **then**
3: // corresponde cerrar un XMLAttribute
4: **print** "'"
5: **else**
6: // corresponde cerrar un XMLElement
7: **if** *openedHeader* **then**
8: **print** "/>"
9: *openedHeader* ← **false**
10: **else**
11: **print** "</" + $dg_{DICT}.getLabelName(stack_{PRINT}.top.first)$ + ">"
12: **end if**
13: **end if**
14: $stack_{PRINT}.pop()$
15: **end while**

Algoritmo 6 printToken(*tokenPos*, *tokenID*).

Require: $BM_{TOKENTYPE}$ // bitmap para marcar el tipo de token (palabra o separador)
Require: $Dict_{WRD}$ // diccionario de palabras
Require: $Dict_{SEP}$ // diccionario de separadores
1: **if** $BM_{TOKENTYPE}.isBitSet(tokenPos)$ **then**
2: $token \leftarrow Dict_{WRD}.at(tokid)$
3: **if** $lastTokenWasWordType$ **then**
4: **print** "␣"
5: **end if**
6: $lastTokenWasWordType \leftarrow$ **true**
7: **else**
8: $token \leftarrow Dict_{SEP}.at(tokid)$
9: **if** $lastTokenWasWordType \wedge isStopWord(token)$ **then**
10: **print** "␣"
11: **end if**
12: $lastTokenWasWordType \leftarrow isStopWord(token)$
13: **end if**
14: **print** *token*

Algoritmo 7 Impresión de un nodo XML.

Require: wt_{PRE} // pre order wavelet tree
Require: wt_{POST} // post order wavelet tree
Require: dg // dataguide
Require: sum_{WRD} // sumas parciales de offset de palabra
Require: bm_{OFF} // bitmap para offset de fichero
Require: $file_{ETDC}$ // fichero comprimido con ETDC
Require: $nodeFrom$ // posición de palabra de comienzo del segmento
Require: $nodeTo$ // posición de palabra de término del segmento (exclusive)
Require: $nodeStructId$ // StructId del nodo

1: // obtiene el pre orden del nodo
2: $highestContainerPre \leftarrow sum_{WRD}.rank(1, sum_{WRD}.select(0, nodeFrom))$
3: $nodeRank \leftarrow wt_{PRE}.rank(nodeStructId, highestContainerPre)$
4: $nodePre \leftarrow wt_{PRE}.select(nodeStructId, nodeRank)$
5: // obtiene el offset de fichero
6: $fileFrom \leftarrow sum_{OFF}.select(nodeFrom + 1)$
7: $file_{ETDC}.setOffset(fileFrom)$
8: $nextToken \leftarrow nodeFrom$
9: $lastTokenWasWordType \leftarrow$ **false**
10: **while** $nextToken + nodePre < nodeTo + nodeFollowing$ **do**
11: **if** $sum_{WRD}.IsBitSet(nextToken + nodePre + 1)$ **then**
12: $printCloseLabel(nodePre)$
13: $printOpenLabel(nodePre)$
14: $nodePre++$
15: **else**
16: $tokid \leftarrow decode(file_{ETDC})$
17: $printToken(nextToken, tokid)$
18: $nextToken++$
19: **end if**
20: **end while**

Algoritmo 8 Segunda pasada sobre el fichero temporal.

Require: $bm_{BALANCED}$ // secuencia auxiliar de paréntesis balanceados
Require: $array_{LabelID}$ // secuencia auxiliar con las llaves de etiquetas
Require: aux_file_{ETDC} // fichero temporal comprimido con ETDC
Require: dg // dataguide
Require: dg_{DICT} // diccionario de etiquetas del dataguide
Require: bm_{WRD} // secuencia de bits para sumas parciales de offset de palabra
Require: Π_{WRD} // permutación para obtener el modelo ETDC para palabras
Require: Π_{SEP} // permutación para obtener el modelo ETDC para separadores

1: $p \leftarrow 0, t \leftarrow 0, w \leftarrow 1, total\_offset \leftarrow 0$
2: **while** $p < bm_{BALANCED}.length()$ **do**
3: **if** $bm_{BALANCED}.isBitSet(p)$ **then**
4: // marca el comienzo de nodo en la secuencia de bits para offset de fichero
5: $bm_{OFF}.set(total\_offset)$
6: $labelKey \leftarrow array_{LabelID}.at(t)$
7: $label \leftarrow dg_{DICT}.at(labelKey)$
8: **if** $stack.empty()$ **then**
9: $parentStructId \leftarrow NULL$
10: **else**
11: $parentStructId \leftarrow stack.top()$
12: **end if**
13: $StructId \leftarrow dg.obtainSID(label, parentStructId)$
14: // agrega un elemento nuevo en la secuencia de pre orden
15: $array_{PRE}.at(t) \leftarrow StructId$
16: $stack.push(StructId)$
17: $t++$
18: **while** $\neg sum_{WRD}.isBitSet(w)$ **do**
19: $tokid \leftarrow aux\_file_{ETDC}.decode()$
20: **if** $BM_{TOKENTYPE}.isBitSet(w)$ **then**
21: $finaltokid \leftarrow \Pi_{WRD}.at(tokid)$
22: **else**
23: $finaltokid \leftarrow \Pi_{SEP}.at(tokid)$
24: **end if**
25: // escribe en el fichero definitivo comprimiendo con ETDC
26: $size \leftarrow file_{ETDC}.encode(finaltokid)$
27: $total\_offset \leftarrow total\_offset + size$
28: $w++$
29: **end while**
30: **else**
31: $StructId \leftarrow stack.pop()$
32: // agrega un elemento nuevo en la secuencia de post orden
33: $array_{POST}.at(p - t) \leftarrow StructId$
34: **end if**
35: $p++$
36: **end while**
37: **return** $\langle array_{PRE}, array_{POST}, bm_{OFF}, file_{ETDC} \rangle$

4.3. Mediciones

4.3.1. Datos de Prueba

Los ficheros utilizados para medir el desempeño de la compresión han sido obtenidos desde diferentes repositorios conocidos. A continuación se detallan sus características.

Colección	Documentos	Nodos	Palabras	Tokens	Tamaño Total
allelements	1	4,619	898	12,647	92KB
mondial	1	77,315	18,432	152,803	1,8MB
religion	4	94,624	16,350	1,468,443	6,8MB
shakespeare	37	327,205	26,449	1,284,866	7,9MB
doxygen	526	967,086	19,548	1,571,551	18MB
treebank	1	3,829,513	1,938,290	6,956,344	83MB
dblp	1	14,553,325	1,301,824	44,070,942	283MB
psd7003	1	38,313,698	2,947,971	76,542,147	685MB
xcdna	1	22,302,047	1,212,997	28,381,836	703MB
10d	1	32,298,990	815,779	159,086,333	1.1GB
eswiki	1	13,987,196	4,396,510	405,874,710	1.9GB

Cuadro 4.1: Características de las Colecciones.

ALLELEMENTS Corresponde a una representación en XML de la tabla periódica de los elementos [2].

MONDIAL Base de datos geográfica mundial integrada desde la CIA World Factbook, el International Atlas, y la base de datos TERRA entre otras fuentes [May99].

RELIGION (Religion.200) Es un grupo de cuatro trabajos de religión etiquetados para publicación electrónica desde fuentes disponibles públicamente[3]. Los textos fueron inicialmente etiquetados (1992) como un ejercicio en SGML DTD y diseño de plantilla de estilos, y en 1996 fueron liberados en conjunto con SHAKESPEARE como los primeros ejemplos de documentos reales etiquetados en XML. La distribución actual está en conformidad con la Recomendación XML 1.0 liberada el 8 de Febrero de 1998.

[2] Pertenece al *Test DataOver100K, XML Binary Characterization Working Group*, http://www.w3.org/XML/Binary/2005/03/test-data/Over100K/.

[3] http://www.ibiblio.org/bosak/.

SHAKESPEARE (shaksper.200) Es un conjunto de guiones de William Shakespeare etiquetados para publicación electrónica[4]. El conjunto comenzó siendo un fichero en ASCII puesto en el dominio público por Moby Lexical Tools en 1992, y fue inicialmente etiquetado con el mismo propósito que RELIGION.

DOXYGEN Es el resultado de aplicar `doxygen` sobre el código fuente de XPN para generar su documentación en formato XML. Contiene información sobre las clases, código de cabeceras y comentarios añadidos.

TREEBANK Corresponde a una base de datos de sentencias en inglés, etiquetadas de acuerdo a su discurso, cuyos nodos de texto han sido codificados criptográficamente porque son textos con derechos de copia restringidos pertenecientes al Wall Street Journal. Sin embargo, la estructura recursiva de las etiquetas es muy profunda y por ende un interesante caso para los experimentos.

DBLP Es una base de datos que provee información bibliográfica de las conferencias y publicaciones más importantes en ciencias de la computación. DBLP indexa más de un millón de artículos y contiene más de 10000 links hacia páginas personales de científicos de la computación. El fichero corresponde a la revisión del 16 de Abril de 2008.[5]

PSD7003 Sigla de Protein Sequence Database, pertenece a la base de datos pública de secuencias de proteínas de la *Integrated Protein Informatics Resource for Genomic and Proteomic Research* ubicada en la *Georgetown University Medical Center*[6]. Contiene una colección integrada de secuencias de proteínas anotadas funcionalmente.

XCDNA Corresponde a una representación en XML de la proteína *Glutathione peroxidase 4 isoform B precursor* (H-Inv cluster ID = HIX0027508).

10D Corresponde al fichero generado con `xmlgen` (ver 3.2.1) con el parámetro para el tamaño ($-f$) igual a 10.

[4] idem.
[5] http://www.informatik.uni-trier.de/ ley/db/.
[6] http://pir.georgetown.edu/.

ESWIKI Corresponde al fichero que contiene todos los artículos activos de la Wikipedia en español. No contiene las discusiones ni las páginas de los usuarios. Corresponde a la revisión del 7 de Mayo de 2008.

En el cuadro 4.1 se muestran las características de dichos documentos[7].

4.3.2. Distribución de la Compresión

A continuación se presentan los resultados sobre la compresión obtenido con el esquema descrito en las secciones 4.1 y 4.2.

La compresión está medida como tasa entre el tamaño resultante y el tamaño original.

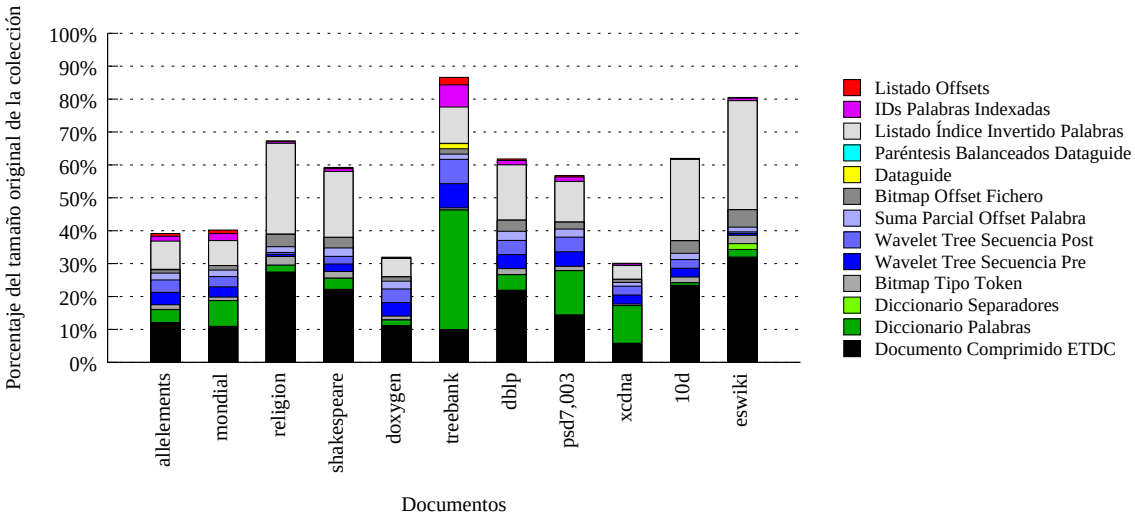

Figura 4.1: Distribución de la compresión resultante por componentes.

La figura 4.1 y el cuadro 4.2 muestran la compresión lograda con la implementación de segmentos basados en offset de palabra. Los cuatro primeros ficheros (de abajo hacia arriba o de izquierda a derecha respectivamente) están destinados a la reconstrucción del documento original, los cuatro siguientes al índice invertido de etiqueta (con la extensión del alfabeto de etiquetas descrita en la sección 4.1.3) y los tres últimos al índice invertido de palabras sin descontar stopwords. La figura 4.2 y el cuadro 4.3 muestran del mismo modo la compresión lograda descontando stopwords.

[7]El tamaño total corresponde al uso efectivo en disco. Esto significa que, con una página de 4KB, los tamaños se redondean al siguiente múltiplo de ese tamaño.

Documento	Códigos ETDC	$DICT_{WRD}$	$DICT_{SEP}$	BM_{TOKEN}	WT_{PRE}	WT_{POST}	SUM_{WRD}	BM_{OFF}	Dataguide	$BPDataguide$	Listado IIP	ID Palabras	Listado OFF
allelements	.121	.039	.001	.014	.038	.038	.021	.010	.001	.000	.086	.015	.008
mondial	.109	.078	.000	.011	.031	.031	.020	.014	.000	.000	.076	.022	.010
religion	.274	.021	.000	.026	.006	.006	.018	.038	.000	.000	.276	.005	.003
shakespeare	.222	.034	.000	.020	.023	.023	.026	.032	.000	.000	.200	.008	.004
doxygen	.112	.017	.000	.012	.041	.041	.023	.013	.000	.000	.056	.003	.001
treebank	.100	.363	.000	.007	.074	.074	.016	.017	.015	.001	.110	.067	.023
dblp	.219	.047	.000	.019	.043	.043	.028	.034	.000	.000	.168	.013	.004
psd7003	.144	.134	.000	.013	.044	.044	.025	.021	.000	.000	.124	.014	.004
xcdna	.058	.115	.000	.005	.027	.027	.011	.009	.000	.000	.042	.005	.002
10d	.234	.008	.000	.017	.027	.027	.019	.038	.000	.000	.247	.002	.001
eswiki	.320	.023	.018	.025	.005	.005	.014	.053	.000	.000	.331	.008	.002

Cuadro 4.2: Compresión de cada componente como tasa entre el tamaño resultante y el tamaño original.

Documento	Códigos ETDC	$DICT_{WRD}$	$DICT_{SEP}$	BM_{TOKEN}	WT_{PRE}	WT_{POST}	SUM_{WRD}	BM_{OFF}	Dataguide	$BPDataguide$	Listado IIP	ID Palabras	Listado OFF
allelements	.121	.039	.001	.014	.038	.038	.021	.010	.001	.000	.084	.015	.008
mondial	.109	.078	.000	.011	.031	.031	.020	.014	.000	.000	.075	.022	.010
religion	.262	.020	.000	.026	.006	.006	.018	.034	.000	.000	.123	.005	.002
shakespeare	.214	.033	.000	.020	.023	.023	.026	.030	.000	.000	.113	.008	.003
doxygen	.111	.017	.000	.012	.041	.041	.023	.013	.000	.000	.052	.002	.001
treebank	.100	.363	.000	.007	.074	.074	.016	.017	.015	.001	.110	.067	.023
dblp	.218	.047	.000	.019	.043	.043	.028	.033	.000	.000	.159	.013	.004
psd7003	.143	.134	.000	.013	.044	.044	.025	.021	.000	.000	.119	.014	.004
xcdna	.058	.115	.000	.005	.027	.027	.011	.009	.000	.000	.042	.005	.002
10d	.231	.008	.000	.017	.027	.027	.019	.038	.000	.000	.233	.002	.001
eswiki	.319	.023	.018	.025	.005	.005	.014	.052	.000	.000	.233	.008	.002

Cuadro 4.3: Compresión de cada componente como tasa entre el tamaño resultante y el tamaño original, lograda descontando stopwords.

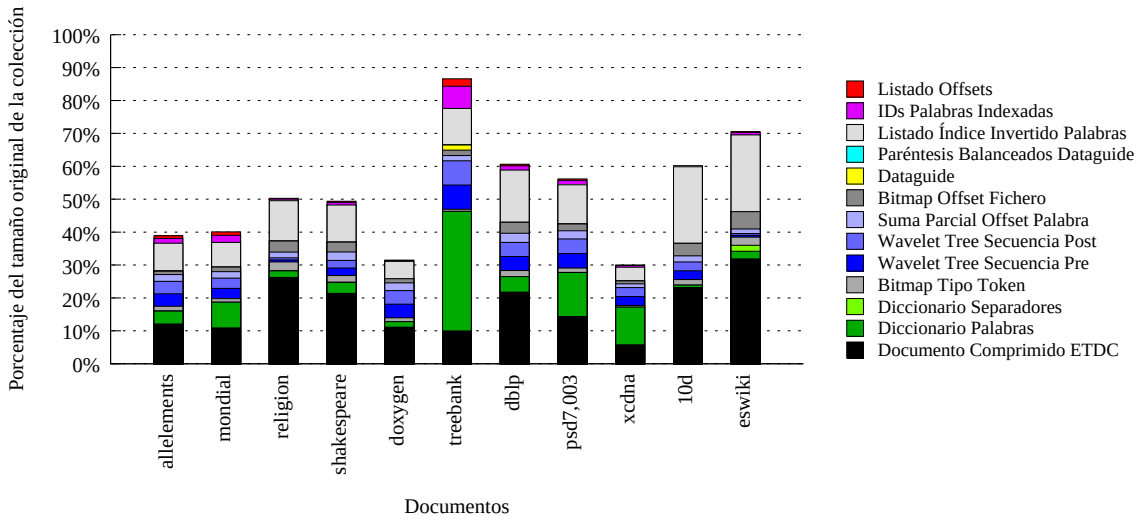

Figura 4.2: Distribución de la compresión resultante por componentes lograda excluyendo stopwords.

Es interesante observar que Wikipedia en español (`eswiki`) contiene gran cantidad de separadores, no así `dblp` porque las palabras que contiene son principalmente sustantivos y por ende no hay variedad de frases, y tampoco `d10` porque su contenido es generado automáticamente en base a concatenación de palabras seguidas de un espacio.

Los ficheros PSD7003 y XCDNA tienen diccionarios de palabras muy grandes porque contienen largas secuencias de código explícito de proteínas. DBLP y MONDIAL muestran un diccionario grande de palabras que se explica por contener principalmente sustantivos; la etiqueta `<author>` es el token más ocurrente en DBLP, en MONDIAL lo es el atributo `@country`, que referencia a una entidad país.

El `Bitmap Tipo de Token` es una secuencia de bits de largo igual al número de tokens en el texto, que sirve para diferenciar palabras significativas (palabras corrientes + tags) de los separadores (separadores + stopwords). Ocupa un espacio considerable (aprox. 5 %) a pesar de estar comprimida a orden cero. Lo mismo ocurre con el `Bitmap de Offset de Fichero`.

`Wavelet Tree de Secuencia Pre` y `Wavelet Tree de Secuencia Post`, son las secuencias provistas de Rank y Select para recorrer la estructura de nodos XML de la colección en pre y post orden respectivamente. Se puede observar que existe coherencia entre el tamaño relativo de estas estructuras, y la relación entre cantidad de nodos y tokens de cada colección

(ver cuadro 4.1).

Por otro lado, el dataguide (`Dataguide` + `Paréntesis Balanceados Dataguide`) ocupa un espacio despreciable en el total. Lo mismo ocurre con el `Listado de Offsets`, que es la lista de posiciones necesarias para acceder al `Listado del Índice Invertido de Palabras`; existe un offset por cada palabra indexada (`ID's de Palabras Indexadas`), que se identifica con uno de los ID's equivalentes por stemming. La extracción de stopwords del índice invertido de palabras llega a reducir en un 10 % el tamaño de las listas de ocurrencias más numerosas (ESWIKI y RELIGION).

Finalmente, cabe destacar el pequeño espacio relativo que ocupan las piezas ligadas a la estructura (dataguide + índice invertido estructural).

Documento	XPN(WORD)	XPN	PPMDI
allelements	0.175	0.392	0.087
mondial	0.198	0.402	0.065
religion	0.321	0.673	0.176
shakespeare	0.276	0.592	0.165
doxygen	0.141	0.319	0.041
treebank	0.470	0.866	0.325
dblp	0.285	0.618	0.102
psd7003	0.291	0.567	0.102
xcdna	0.178	0.301	0.080
10d	0.259	0.62	0.205
eswiki	0.386	0.804	0.232

Cuadro 4.4: Comparación de la compresión resultante.

La compresión de la colección, sin contar las estructuras para indexación (XPN(WORD) que incluye al Documento Comprimido con ETDC, el Diccionario de Palabras, el Diccionario de Separadores y el Bitmap Tipo de Token) alcanza gran desempeño; bajo 15 % en el mejor caso (DOXYGEN) y casi un 47 % en el peor (TREEBANK), aproximadamente un 25 % en promedio. Una comparación con el compresor PPM PPMDI[8] se puede observar en el cuadro 4.4 y en la figura 4.3, que también incluyen el tamaño total logrado con XPN. En la comparación se puede apreciar el desempeño de XPN de acuerdo a la compresibilidad de las colecciones.

[8]hrefhttp://pizzachili.dcc.uchile.clhttp://pizzachili.dcc.uchile.cl.

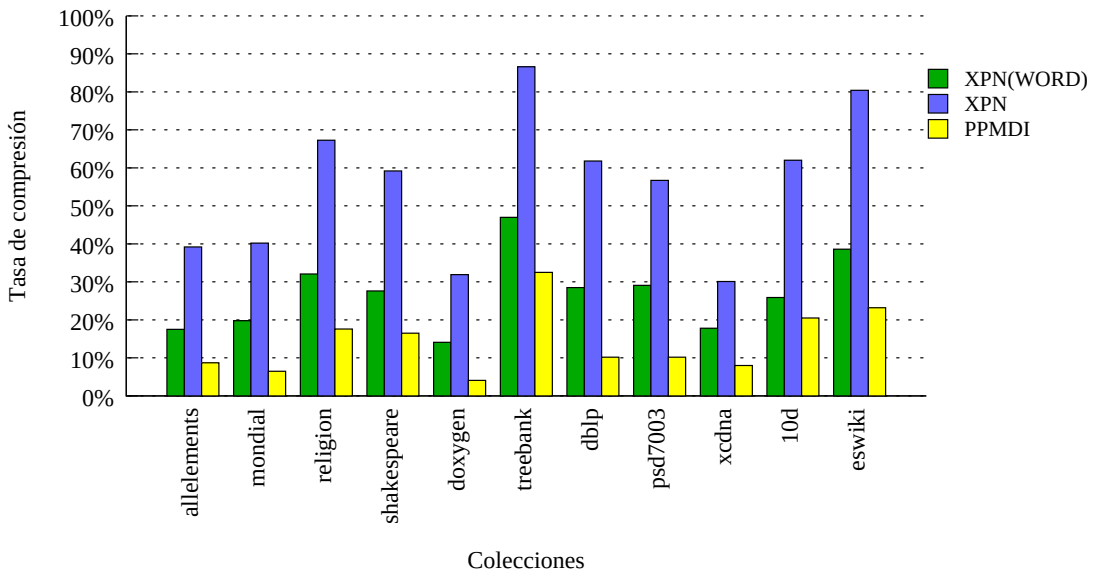

Figura 4.3: Grado de compresión contra PPMDI

4.3.3. Comparación con el Software Disponible

Como ya hemos visto, muchas implementaciones de compresión XML (como XMill o XMLPPM) no están diseñadas para hacer consultas XQuery, y no existen muchas aplicaciones prácticas con desempeño competitivo.

Para poder evaluar el desempeño obtenido con el prototipo desarrollado, es necesario establecer el desempeño de XPN previo a este estudio.

En las figuras 4.4 a 4.23 se puede observar una comparación de desempeño de XPN con algunas bases de datos y procesadores disponibles, realizada con XCheck.

XPN se presenta en tres versiones: XPN-HEAD que corresponde al prototipo descrito en esta memoria, XPN-OLD que corresponde a la última versión que sólo construía el índice (no almacenaba la colección), y XPN-INV, que implementa el dataguide y los índices invertidos con skipping (no con las secuencias de etiquetas provistas de Rank y Select).

XPN-HEAD no posee límite para almacenar el índice invertido de palabras, por lo cual no almacena índices parciales ni necesita hacer mezcla posterior. El objetivo es competir fielmente con las alternativas existentes que no tienen restricciones sobre el uso de memoria principal. Ninguna de las variantes de XPN funcionará con detección de stopwords en esta medición.

El software externo elegido para las mediciones corresponde a: MonetDB/XQuery v0.10 cliente/servidor (ver sección 2.4.1) (MonetDB), Qizx/db v2.1 (ver sección 2.4.1) (Qizx_db), Sedna v3.0.117 (ver sección 2.4.1) y SaxonB versión 9.0.0.1j (ver sección 2.4.2). SaxonB es el único procesador estático.

La medición corresponde al conjunto de consultas propuesto por Xmark (ver sección 3.2.1) y fue aplicado sobre cuatro documentos generados con `xmlgen`: `D0.001` (113.37 Kb), `D0.01` (1.11 MB), `D0,1 (11.13 MB)` y `D1` (111.12 MB).

XCheck (ver sección 3.2.1) fue configurado para detener la medición si se supera un tiempo de 20 minutos (incluyendo la indexación).

Los resultados se presentan en cinco grupos (cada uno con los cuatro ficheros) correspondientes a los tiempos de: indexación (procesamiento del documento), compilación de la consulta, ejecución de la consulta, serialización y tiempo total de ejecución.

Es importante observar que no todos los procesadores reportan los tiempos detallados. Algunos no reportan los tiempos de indexación, otros no reportan los tiempos de compilación de la consulta, y otros no dan cuenta del tiempo de serialización de las resultados.

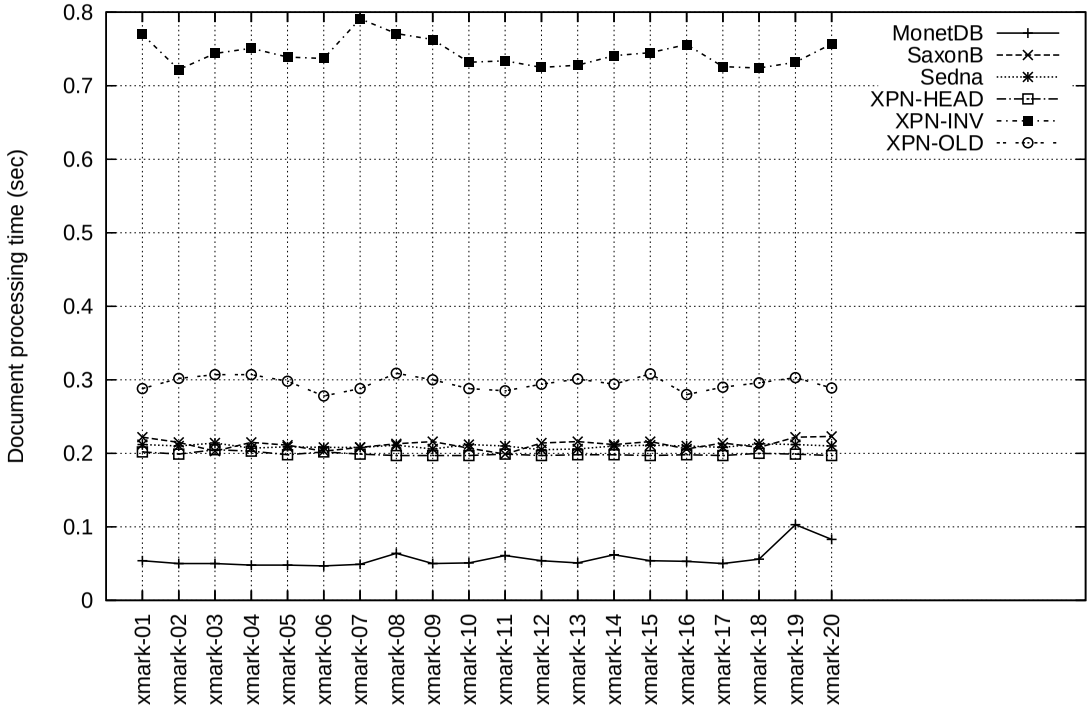

Figura 4.4: Tiempo de Procesamiento del Documento D0.001 (113.37 Kb).

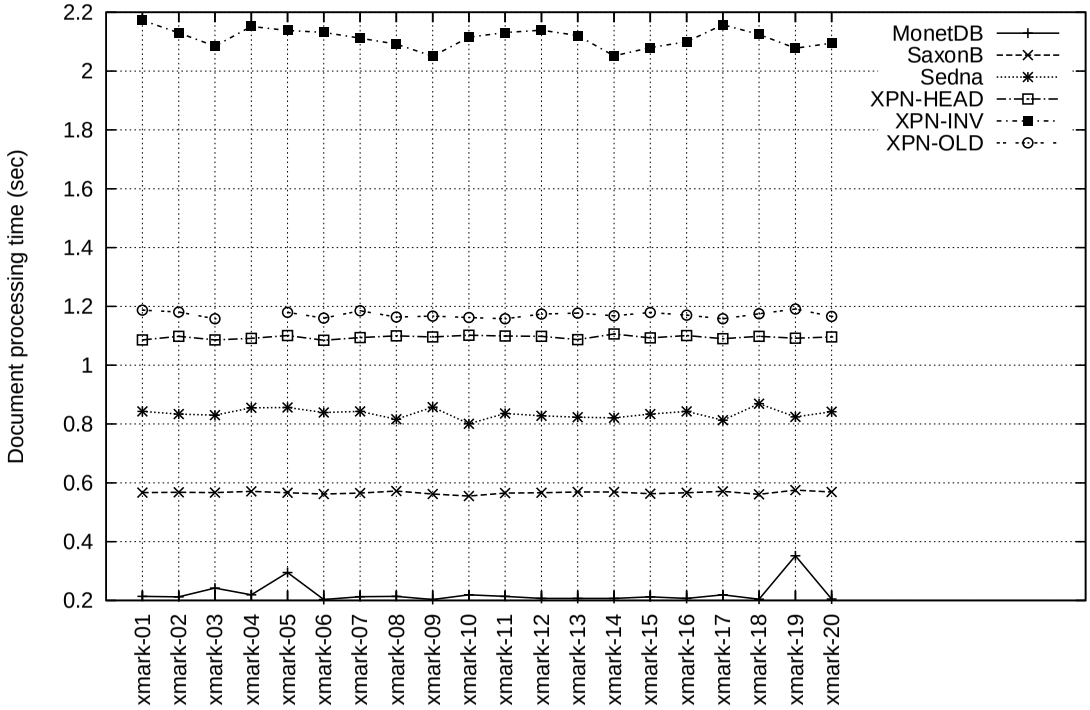

Figura 4.5: Tiempo de Procesamiento del Documento D0.01 (1.11 MB).

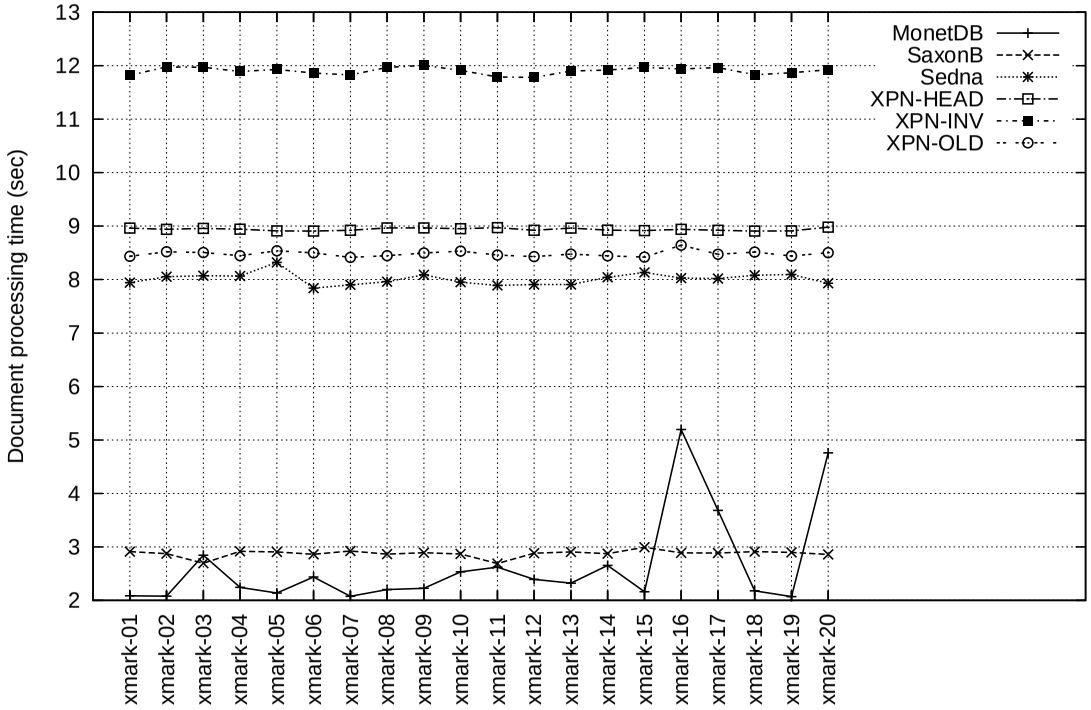

Figura 4.6: Tiempo de Procesamiento del Documento D0.1 (11.13 MB).

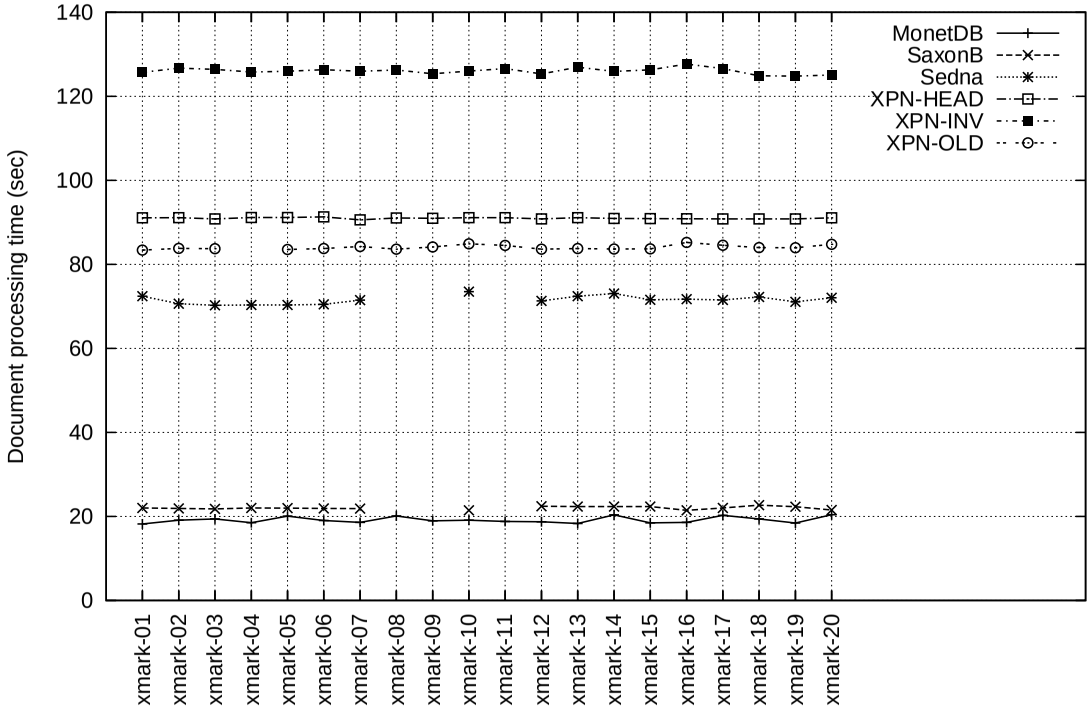

Figura 4.7: Tiempo de Procesamiento del Documento D1 (111.12 MB).

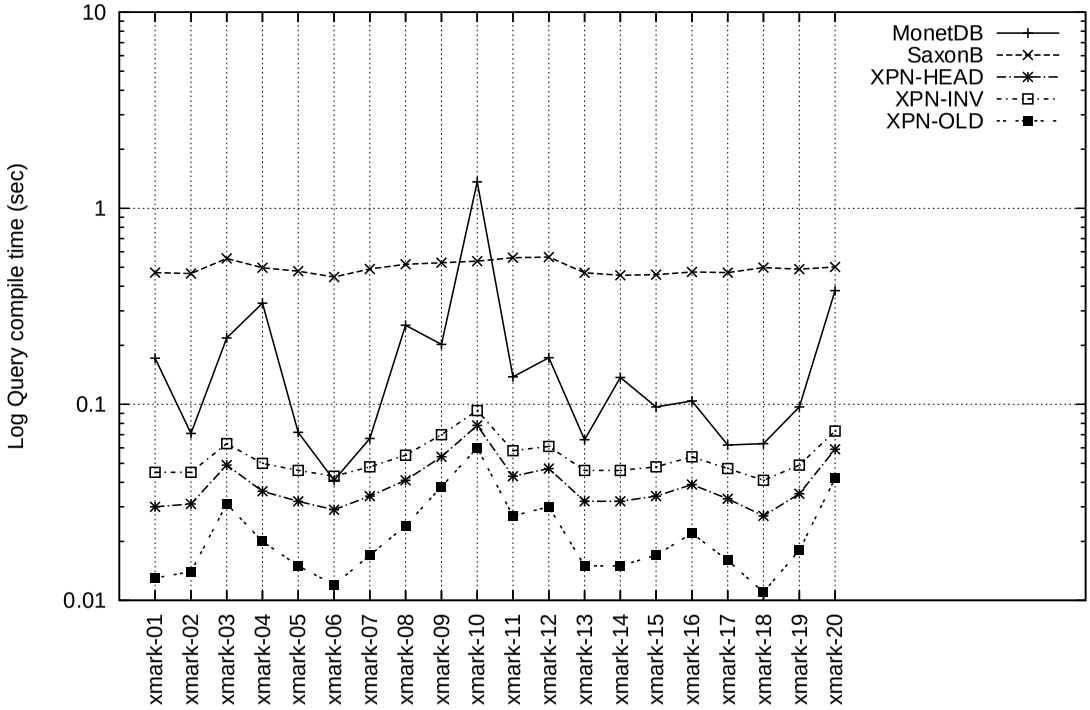

Figura 4.8: Tiempo de Compilación de las Consultas, D0.001 (113.37 Kb).

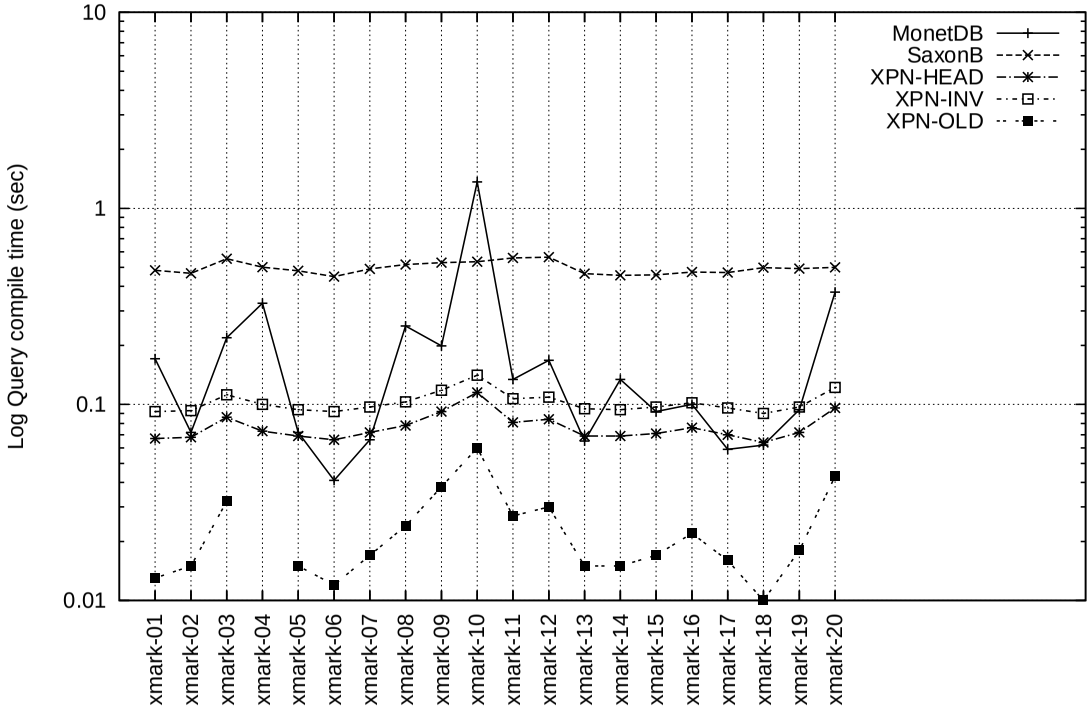

Figura 4.9: Tiempo de Compilación de las Consultas, D0.01 (1.11 MB).

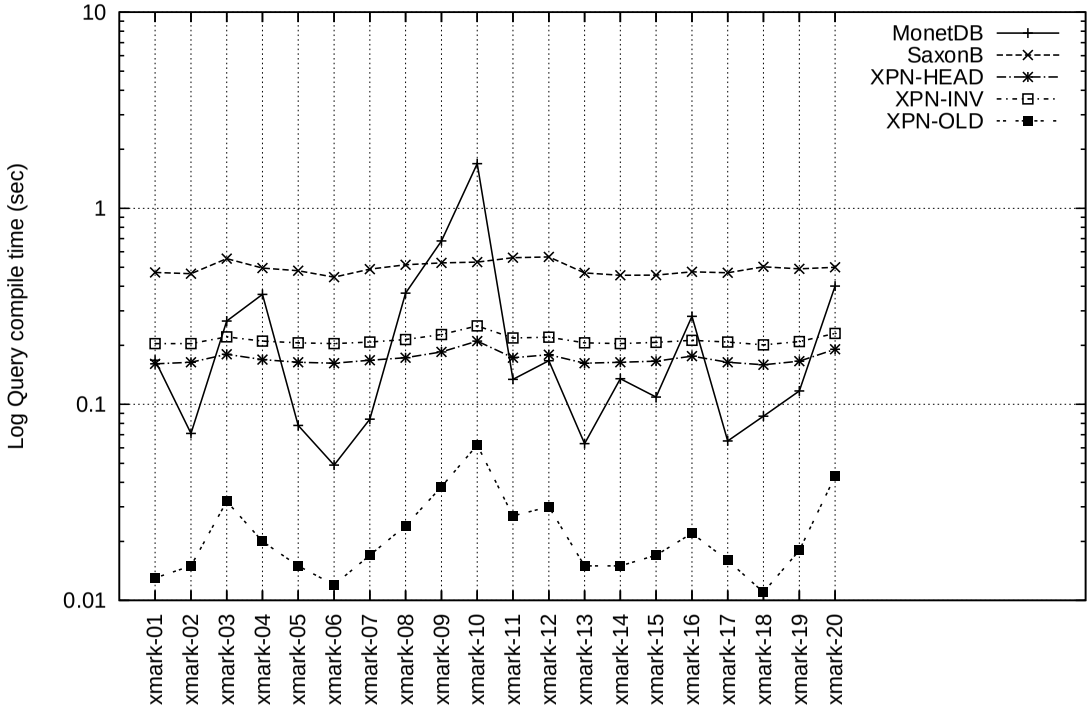

Figura 4.10: Tiempo de Compilación de las Consultas, D0.1 (11.13 MB).

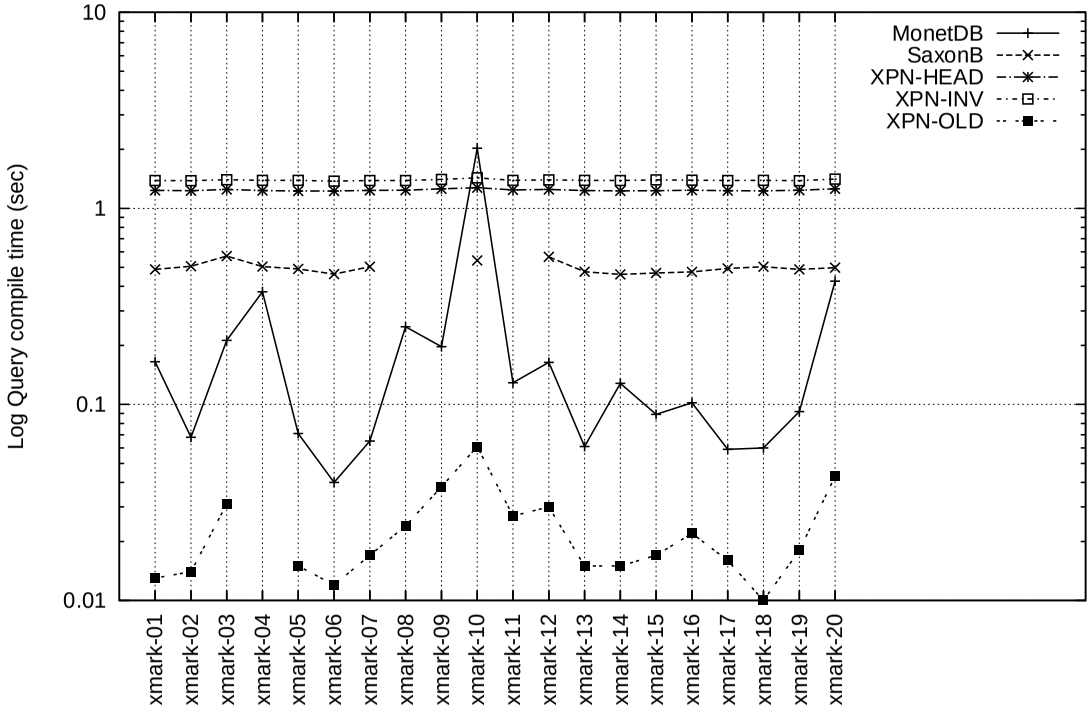

Figura 4.11: Tiempo de Compilación de las Consultas, D1 (111.12 MB).

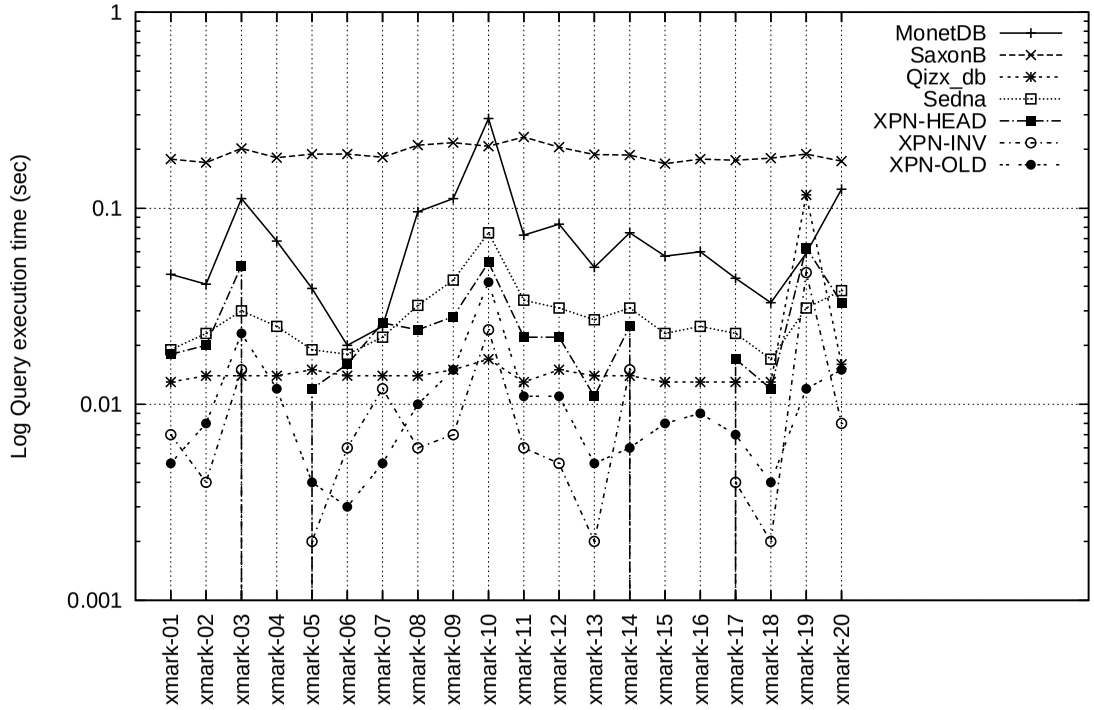

Figura 4.12: Tiempo de Ejecución de las Consultas, D0.001 (113.37 Kb).

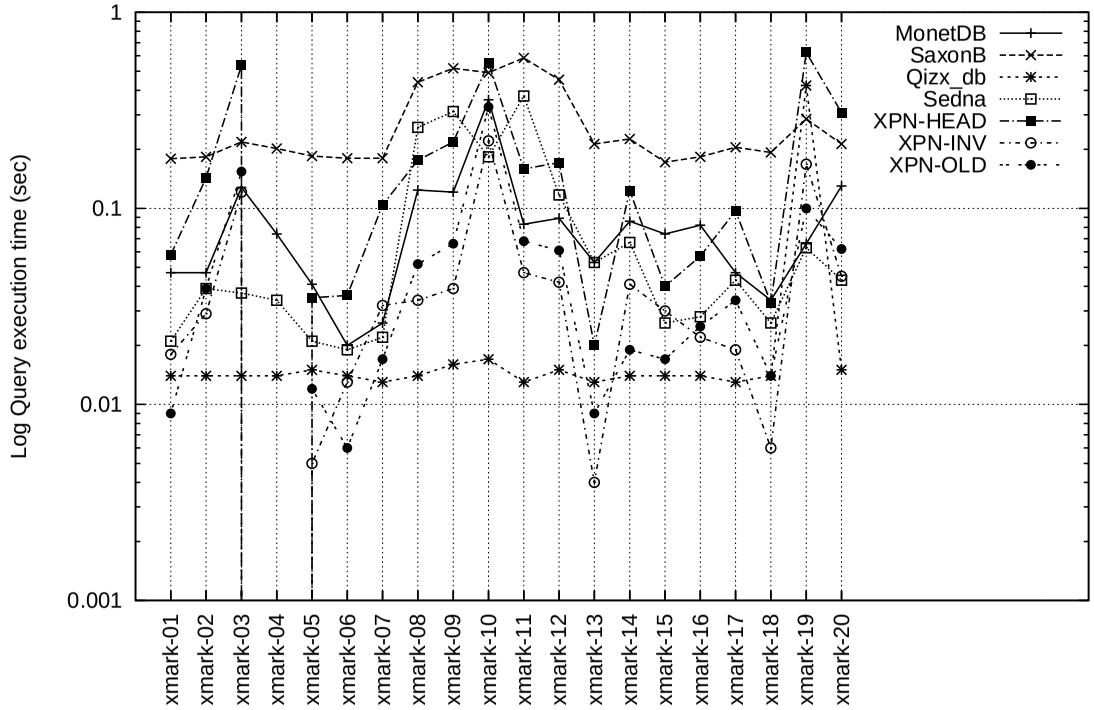

Figura 4.13: Tiempo de Ejecución de las Consultas, D0.01 (1.11 MB).

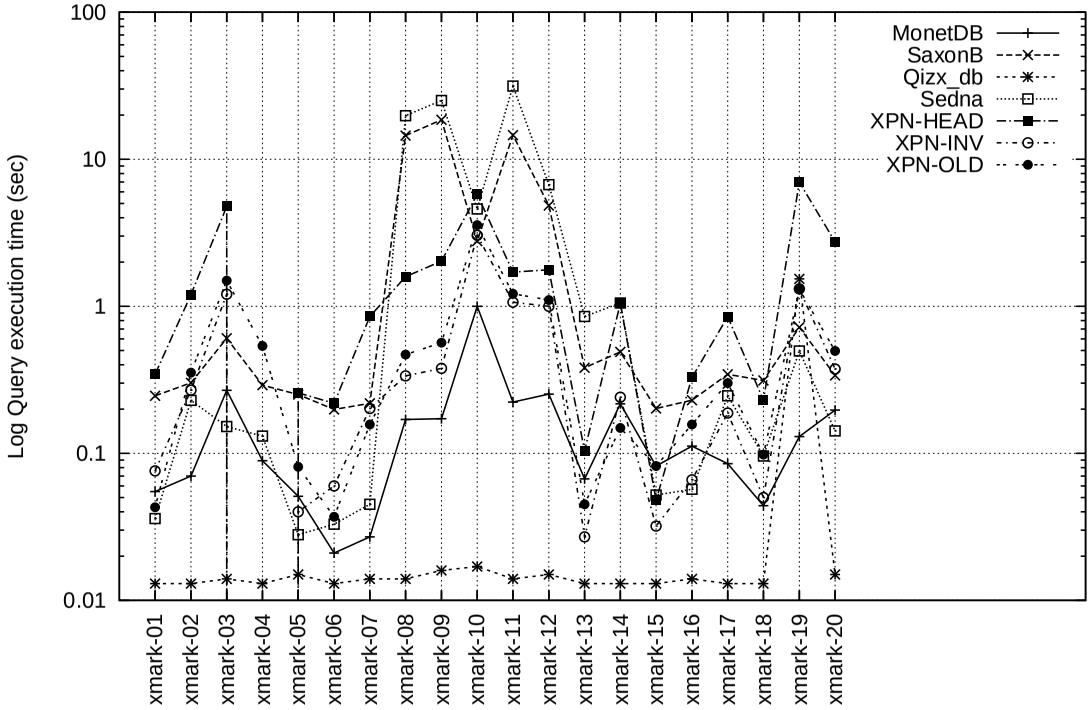

Figura 4.14: Tiempo de Ejecución de las Consultas, D0.1 (11.13 MB).

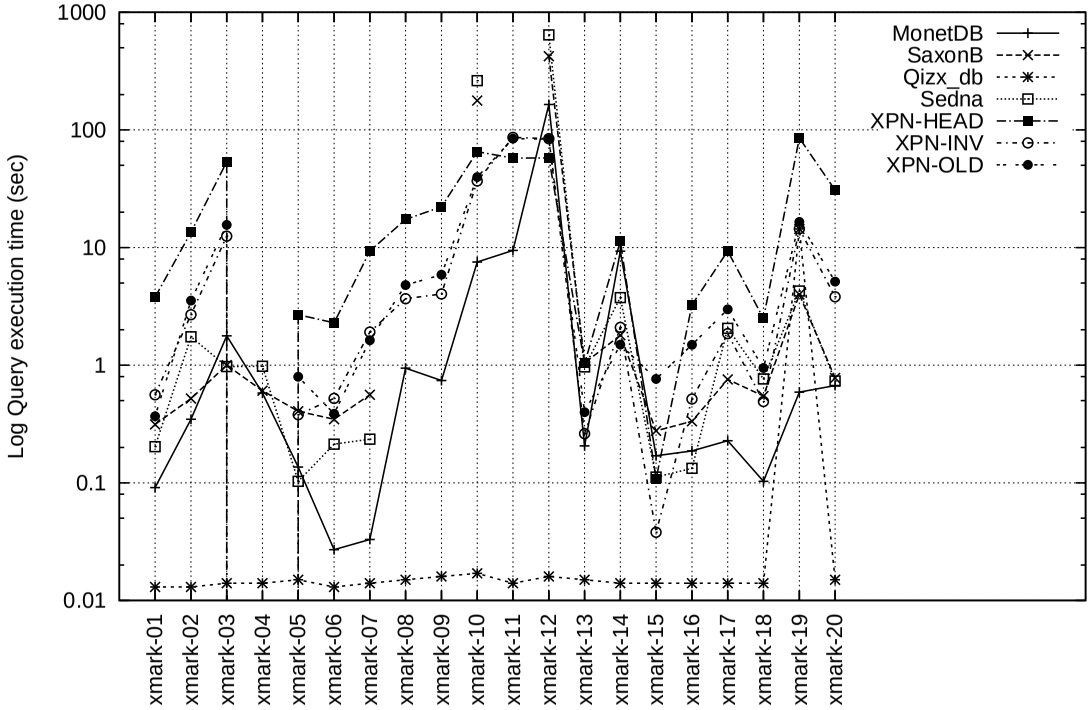

Figura 4.15: Tiempo de Ejecución de las Consultas, D1 (111.12 MB).

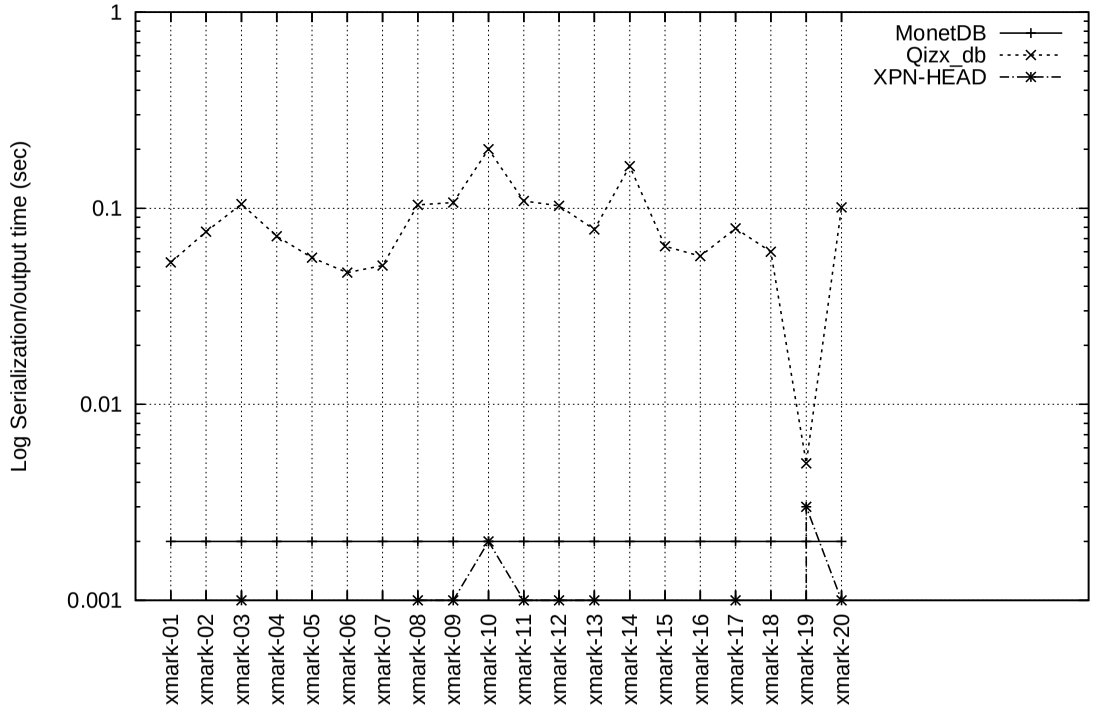

Figura 4.16: Tiempo de Serialización de las Consultas, D0.001 (113.37 Kb).

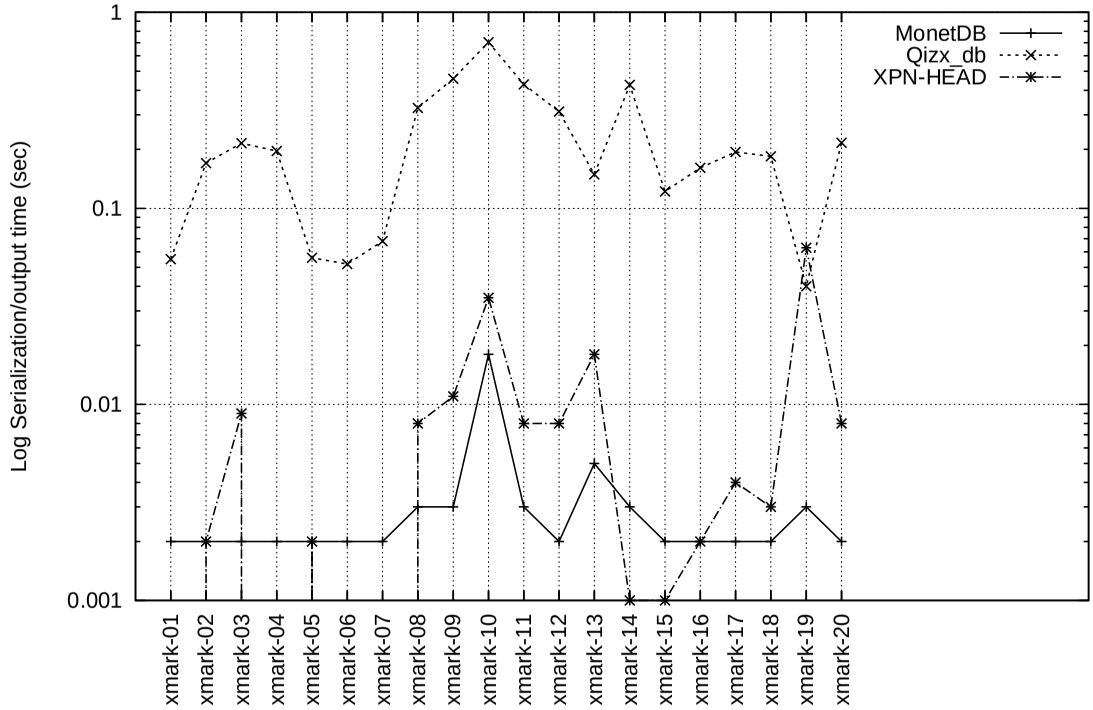

Figura 4.17: Tiempo de Serialización de las Consultas, D0.01 (1.11 MB).

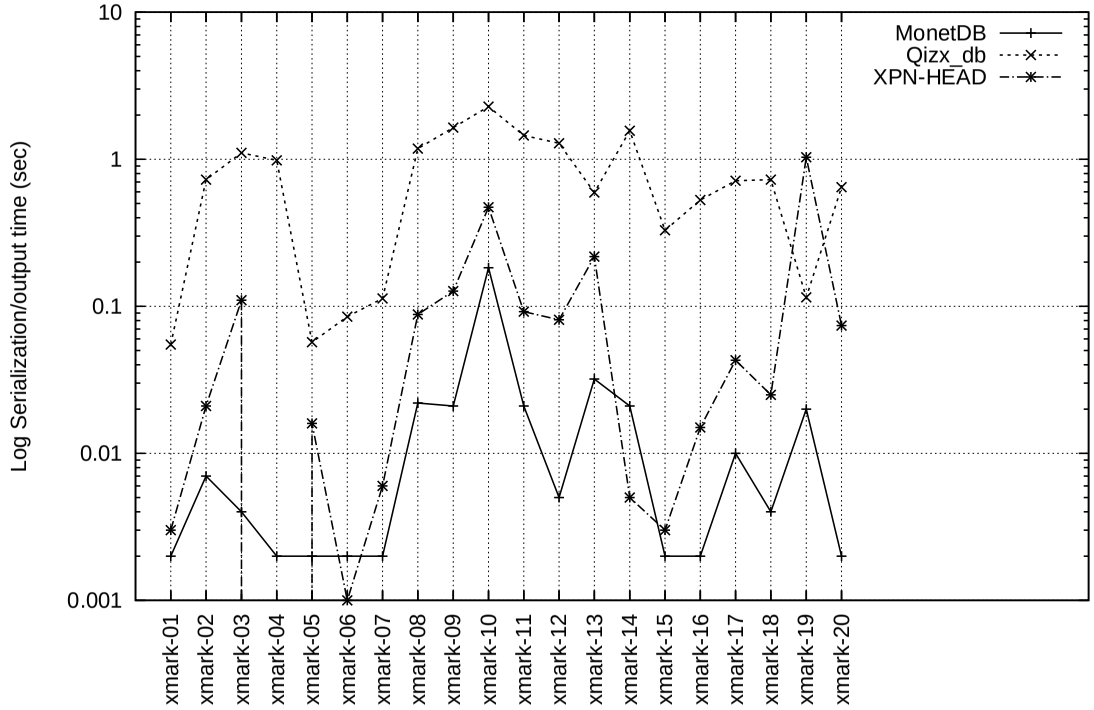

Figura 4.18: Tiempo de Serialización de las Consultas, D0.1 (11.13 MB).

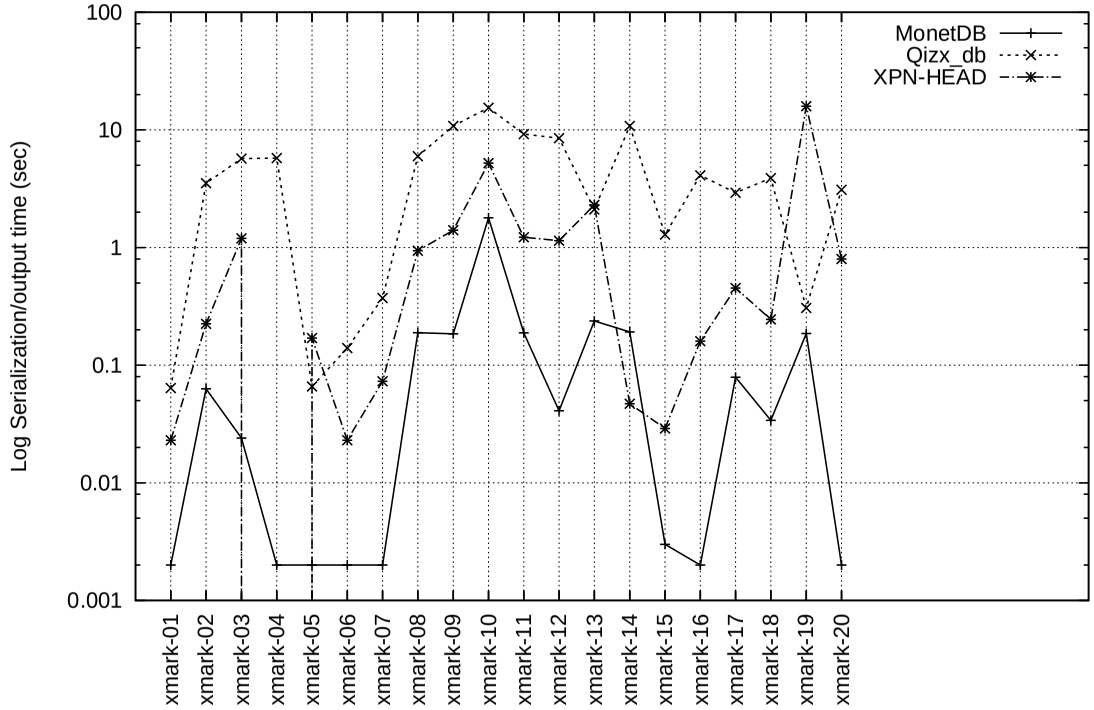

Figura 4.19: Tiempo de Serialización de las Consultas, D1 (111.12 MB).

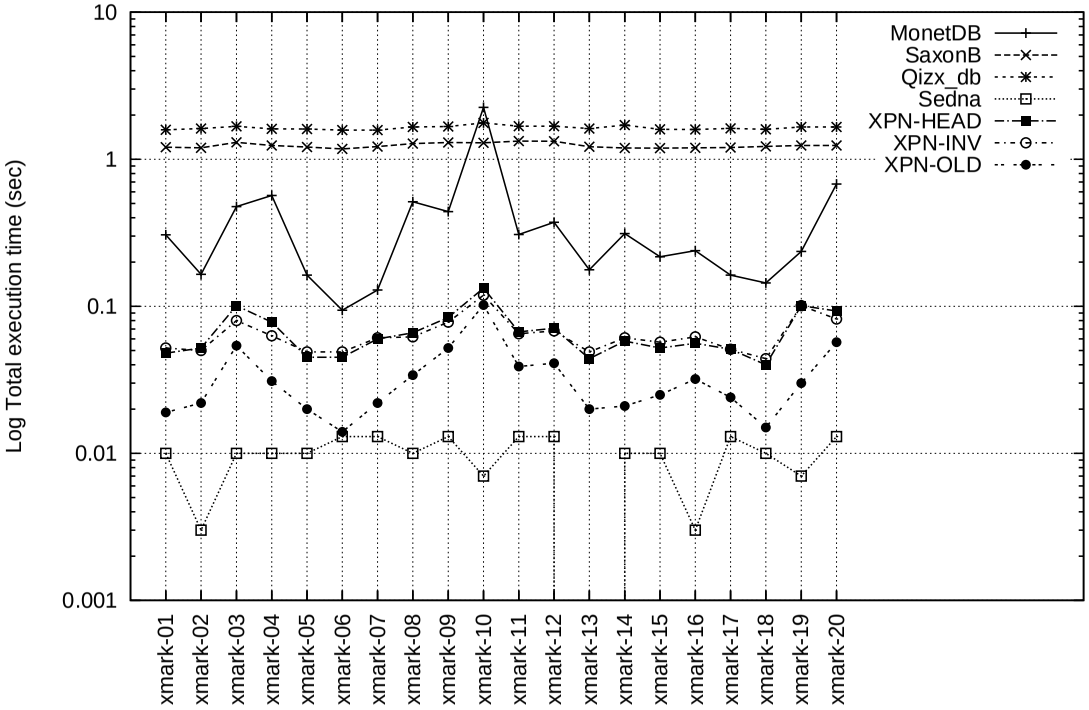

Figura 4.20: Tiempo Total de las Consultas, D0.001 (113.37 Kb).

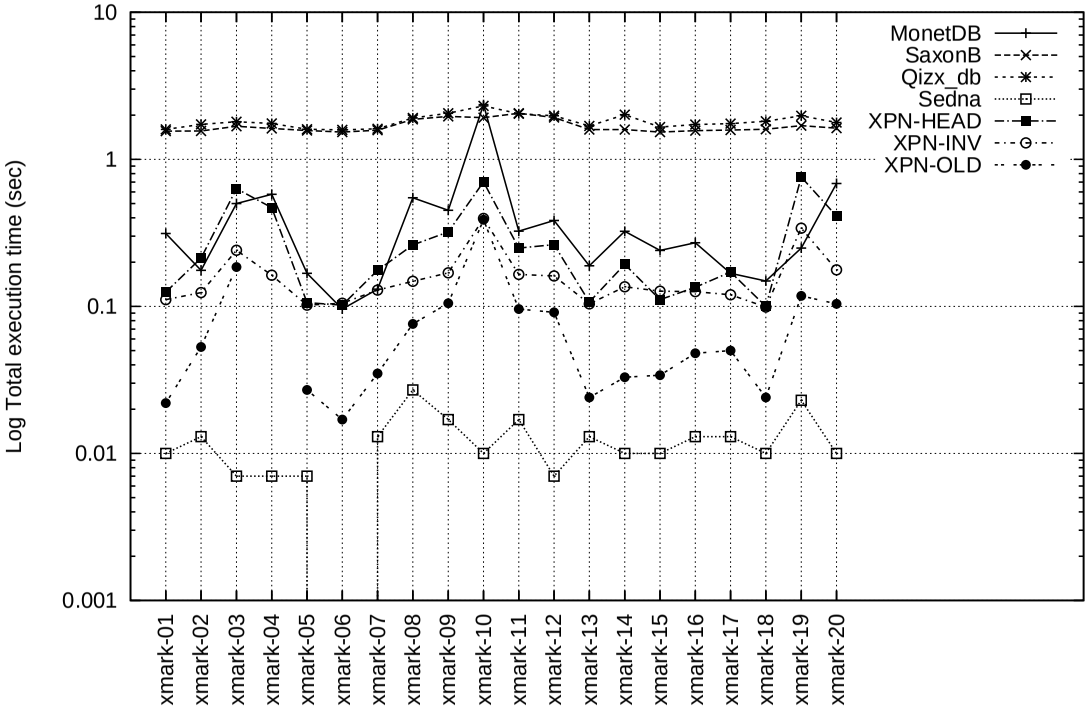

Figura 4.21: Tiempo Total de las Consultas, D0.01 (1.11 MB).

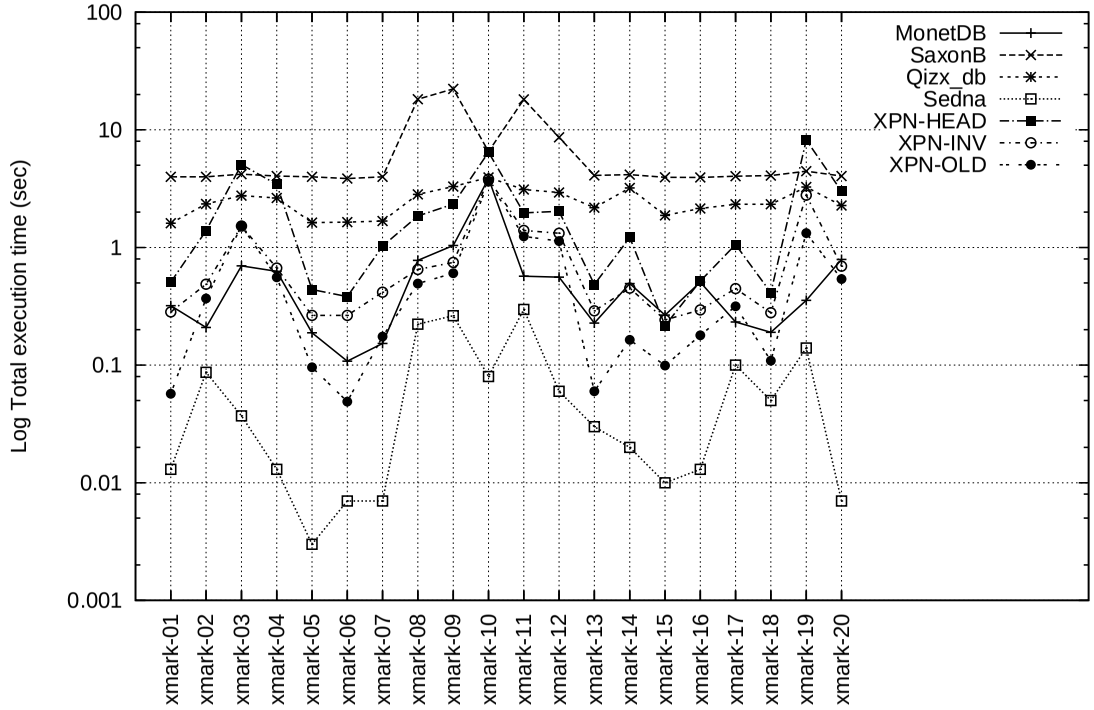

Figura 4.22: Tiempo Total de las Consultas, D0.1 (11.13 MB).

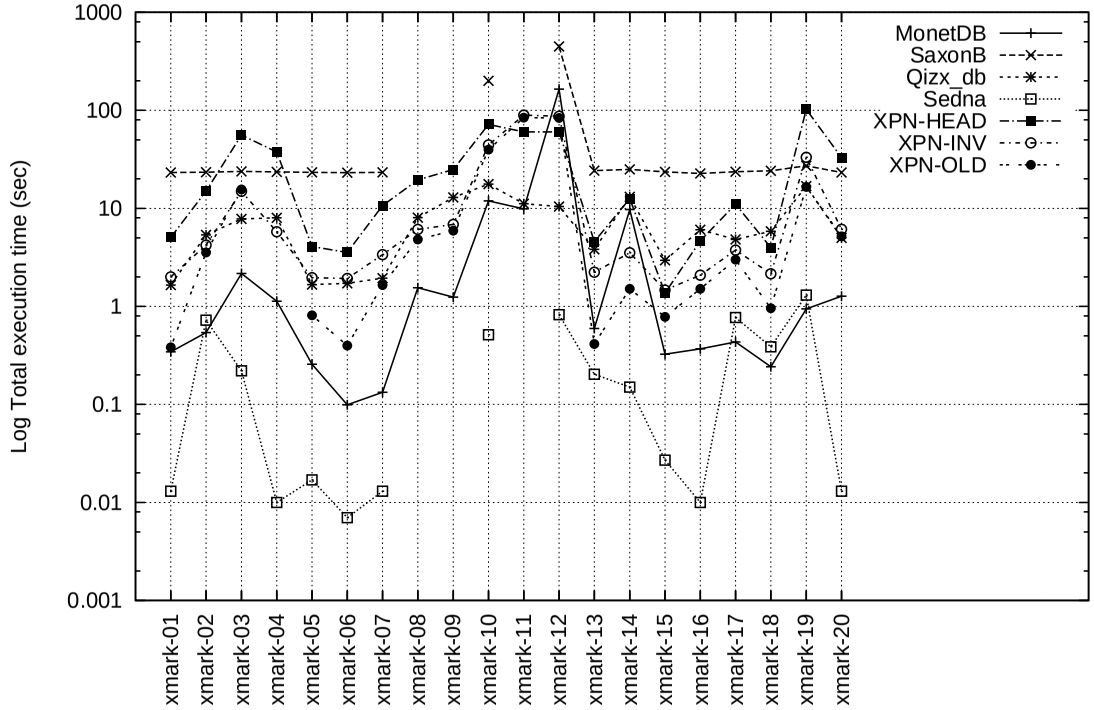

Figura 4.23: Tiempo Total de las Consultas, D1 (111.12 MB).

Capítulo 5

Discusión

Como se puede observar en las figuras de las secciones 4.3.2 y 4.3.3, XPN logra un buen desempeño en compresión a costa de perder un poco de desempeño en la consulta.

5.1. Desempeño en la Consulta XQuery

Es necesario destacar que XPN-HEAD no almacena el fichero en RAM como al parecer lo hace SEDNA. Este último, no es capaz de responder a consultas con joins intensivos en el fichero mayor (consultas xmark 8, 9 y 11), probablemente porque necesita configurar índices para optimizar las consultas, y porque empieza a agotar la memoria disponible.

Por otro lado, comparado con SaxonB, que procesa el fichero sólo para responder a la consulta, XPN-HEAD se demora mucho más en comprimir el fichero, pero en casi todas las consultas responde más rápido y alcanza a responder a los joins intensivos.

En cambio, XPN-HEAD no escala como lo hace SaxonB que con documentos más grandes debiera ser más rápido en responder las consultas. Sin embargo, por funcionar sobre Java, no es capaz de procesar el fichero `D10` [1] con toda la RAM disponible para la máquina virtual (1Gb).

En colecciones de tamaño pequeño XPN en todas sus variantes es superior a MonetDB, SaxonB y Qizx_db en la respuesta a las consultas. Esto sin considerar el tiempo requerido

[1] Correspondiente a un archivo de 1.1Gb no utilizado en las mediciones y generado con xmlgen con parámetro de tamaño igual a 10.

para comprimir (o indexar en el caso de XPN-OLD). En colecciones grandes, en cambio, si bien va degradando su tiempo de respuesta, XPN (en todas sus variantes) sigue siendo competitivo con los procesadores disponibles. Sus tiempos de serialización son incluso mejores en promedio que los de Qizx_db.

En la consulta xmark 4, que corresponde a un resultado vacío, la respuesta de XPN-HEAD y XPN-INV es inmediata, lo que muestra que los criterios de definición de qué es tiempo de compilación y qué de ejecución son arbitrarios según la aplicación. A medida que el documento crece, el costo de reconstrucción se va haciendo cada vez mayor, lo cual se explica por el funcionamiento intensivo de las secuencias de etiquetas provistas de Rank y Select (ver algoritmo 7), cuya complejidad temporal crece logarítmicamente tanto con el número de nodos como con el tamaño del vocabulario de etiquetas [CN08]. Además, muchas consultas hacen recuperación de valores para la identificación de nodos, lo cual se traduce en accesos puntuales a secciones no adyacentes del disco que castigan el desempeño global. Sin embargo, esto es esperable de una aplicación que responde sobre una colección comprimida, a diferencia de las alternativas.

En cuanto al desempeño de XPN entre sus variantes, XPN-HEAD logra un tiempo menor en el procesamiento de ficheros de tamaño pequeño a pesar que XPN-OLD sólo los indexa. A medida que el fichero crece, XPN-OLD supera a XPN-HEAD en la indexación aún cuando almacena índices parciales para el índice invertido de palabras y luego los mezcla en disco.

El tiempo dedicado a indexar por parte de XPN-INV es prohibitivo en colecciones de tamaño pequeño y no logra acortar distancia con ficheros mayores. La tasa de compresión alcanzada es aproximadamente un 10 % por sobre XPN-HEAD. Además, genera una gran cantidad de ficheros, de modo que es necesario configurar el sistema operativo para permitir la creación del índice sobre colecciones grandes.

XPN-INV tampoco logra sacar provecho del dataguide, como se puede observar al compararlo con XPN-OLD, pues el tiempo de compilación es demasiado alto. XPN-HEAD, en cambio, castiga los tiempos totales en la dificultad de recuperar la información a medida que el fichero crece. No obstante, sigue siendo competitivo y tiene la ventaja de la compresión.

La degradación del tiempo de compilación y consulta se debe probablemente a la dificultad en la creación de las estructuras auxiliares para la recuperación del documento original (dataguide y varias secuencias de bits). Además, los prototipos XPN-HEAD y XPN-INV poseen un proceso muy naive para sacar provecho del dataguide, el cual tiene complejidad $n \log n$ con el tamaño del dataguide. El procedimiento consiste en obtener el conjunto de todos los StructId potenciales para cada etiqueta y luego intersectarlos mediante los operadores estructurales que determina el plan de consulta. Esto se hace desde las hojas hacia la raíz y luego a la inversa, descartando todos aquellos StructId que no satisfacen la relación evaluada. De este modo, la obtención de los StructId potenciales empieza a sufrir el costo del escalamiento pues entre mayor el archivo, mayor cantidad de rutas posibles desde la raíz hacia cada etiqueta.

La solución debiera pasar por construir el árbol de consulta en análisis conjunto con el dataguide, y no en un proceso posterior de poda como el descrito pues es muy ineficiente. Al analizar las consultas XQuery se establecen las relaciones estructurales entre etiquetas porque se está determinando qué operadores usar, por tanto se puede aprovechar de resolver ambos problemas en conjunto.

Es muy importante destacar que la recuperación de estructuras y datos en XPN se hace desde memoria secundaria y comienza cada vez que se ejecuta una consulta, a diferencia de todos los demás procesadores utilizados. Por seguir el modelo cliente/servidor, estos últimos mantienen estructuras en memoria principal entre el procesamiento y la consulta, o como SaxonB, analizan el fichero de acuerdo a ella.

La única excepción a esta regla es MonetDB, que también es capaz de responder en forma estática, pero igualmente utiliza muchísima memoria principal (no es capaz de indexar `D10` en el equipo disponible sin hacer un intensivo proceso de swapping). Aún así, en las mediciones nunca se contabiliza el tiempo de partida del servidor.

Esto significa que XPN, si dispusiera de todas las estructuras en RAM, probablemente superaría a los demás procesadores. Por sus características de acceso local posee buenas características para hacer uso del caché de memoria. Además, con colecciones de tamaño

gigante (D10 y ESWIKI superan la RAM del equipo de medición), XPN es capaz de comprimir y consultar sin los problemas que plantean los demás procesadores. Cabe destacar, sin embargo, la naturaleza estática de las colecciones indexadas, lo cual es una desventaja para XPN en determinados contextos y no se percibe en estas comparaciones.

5.2. Desempeño en la Compresión

La tasa de compresión alcanzada por XPN (versión HEAD) es de aproximadamente un 60 % si lo observamos como un autoíndice, con fuertes variaciones dependiendo de la colección.

Si consideramos solamente la compresión del fichero sin capacidad de consulta, la tasa de compresión también varía fuertemente y es de aproximadamente un 25 %.

A continuación se hace un análisis por componentes.

5.2.1. Archivo

La ventaja del uso de ETDC radica en su gran velocidad de codificación, lo cual hace que la incorporación del documento al fichero tenga un impacto leve en el desempeño comparativo de la aplicación con respecto a su implementación previa, que no comprime nada y sólo parsea.

Con los stopwords agregados como separadores se obtiene tamaño pequeño para las palabras relevantes más comunes y también para los separadores más comunes, sea este un stopword o un separador propiamente tal.

5.2.2. Índice Invertido Estructural

En el índice invertido estructural original basado en listas de segmentos (ver sección 2.3.3) se almacenan implícitamente los mismos valores de diferencias de offset en más de una oportunidad. La diferencia de offsets de término y de comienzo de un nodo ascendiente cualquiera, representa a la suma de las diferencias de offsets de término y comienzo de todos sus descendientes directos. Recursivamente, cada nodo representa la suma de los tamaños de los segmentos de todos sus descendientes directos, en todos los niveles hacia abajo, lo cual

es muy ineficiente en términos de espacio, y justifica el nuevo enfoque "factorizado" que se logra con la secuencia de offset representada como sumas parciales.

5.3. Extensión del Conjunto de Etiquetas

Cabe destacar que puede parecer muy ineficiente la separación de los tipos de etiquetas con el dataguide para favorecer el *skipping*, si luego se procederá a unir los resultados. Sin embargo, la ganancia de contar con características específicamente convenientes para cada flujo al recorrer las listas de pre y post orden compensan esta operación extra.

Por otro lado, gracias a esta condición se determinan las relaciones Parent y Child entre etiquetas cuando se está procesando un plan de consulta XQuery, lo cual es de muchísima utilidad porque permite eliminar los `StructId` que no cumplan la relación que se pida en los caminos XPath.

La extensión del conjunto de etiquetas, también permite establecer claramente que los flujos de nodos que son resultado del operador Parent y Child no pierden su relación de orden. Y por último, permite optimizar el cálculo del ordenamiento de resultados intermedios, logrando complejidad lineal en muchos casos.

El detalle de estas características en el dominio XQuery de las consultas están fuera del alcance de esta memoria, pero son una potente razón para la existencia del dataguide como estructura que es necesario crear y almacenar para cada colección.

La ventaja del enfoque planteado es que, debido a que gran cantidad de documentos XML poseen un vocabulario pequeño de etiquetas, el tamaño y complejidad del esquema de relaciones es pequeño. Por ende el tamaño del hash para el dataguide y el dataguide mismo no serán un problema y por tanto su consulta tampoco.

Las soluciones existentes a árboles con aridades grandes en los cuales se necesita acceder al nodo hijo cuya etiqueta es dada [FR08, Jac89, MR01, BDM+05, RRR02], tienen la complejidad de la navegación asociada al grado del nodo. En nuestro enfoque, el árbol es una reducción al esquema, que si bien puede presentar problemas de aridad, en general, reducirá considerablemente la cantidad de relaciones, y en la práctica la consulta se restringe a

determinar existencia.

5.4. Dataguide vs XML Schema

El tipo de estructuras XML que pueden existir en la práctica en las diferentes colecciones disponibles está determinada fuertemente por el origen de la información. Una gran ventaja del enfoque de XPN es que no requiere tener conocimiento a priori sobre la estructura del repositorio para poder hacer una indexación conveniente. Otras bases de datos requieren la definición de índices por etiqueta para lograr un desempeño aceptable, sin embargo, esto sólo es posible si se conoce la estructura de la información y la semántica de las consultas. En XPN, la indexación es plana en el sentido de que ninguna etiqueta tiene ventaja sobre otra en el acceso, gracias a la implementación de Rank y Select. Esto significa que no es necesaria la incorporación de índices dedicados para optimizar las consultas, porque la aplicación detectará la estructura sacando provecho de ella.

En [ANd07a] se estudia un tipo de información que tiene un alto nivel de redundancia que no se encuentra las colecciones resultantes de una base de datos XML modelada y normalizada adecuadamente con XNF por ejemplo [AL02]. La colección forma parte de un repositorio de ficheros que no fueron modelados como fuente de información, sino más bien como reporte, y posee el mismo tipo de problema que puede presentarse en ficheros como Wikipedia; la redundancia en elementos, que corresponde a etiquetas con sus subárboles correspondientes, proviene de la no integración de las diversas fuentes en un sólo modelo, y produce repeticiones que caracterizan datos que no han sido normalizados. Esto se produce porque simplemente no es posible hacerlo de otro modo.

En este sentido, la flexibilidad en el dataguide de XPN permite adaptarse de mejor forma a las diversas fuentes de información en formato XML que puede ser agregada a la base de datos.

Una gran ventaja que posee el dataguide frente a los esquemas XML[2], es que contiene información efectivamente instanciada en el documento, lo cual significa que se restringe al

[2]http://www.w3.org/TR/xmlschema-0/.

dominio del documento o colección de un modo más ajustado que éste último. El esquema XML, en cambio, tiene como objetivo la estandarización de la información de acuerdo a reglas de producción que muchas veces no son todas obligatorias y por consiguiente no necesariamente existen todas en una colección dada. Más aún, el esquema XML es capaz incluso de representar estructuras recursivas no acotadas, lo cual no podría ser abordado con el modelo presentado aquí. El dataguide almacena la información estructural que efectivamente posee el documento o colección dado, aún cuando esta posea un esquema XML definido que lo determine.

Como se ha visto en [GW97], no es conveniente almacenar un grafo de relaciones donde cada etiqueta corresponderá a un sólo nodo, porque el problema aparece cuando existen etiquetas iguales en posiciones estructurales distintas, y puede ser necesario incluso diferenciar casos extremos como etiquetas anidadas o loops.

En resumen, el almacenamiento del dataguide permite establecer de manera compacta la información correspondiente a gran parte de la estructura del árbol XML (jerarquía, relación de etiquetas, esquema). Las listas invertidas (secuencias ordenadas por Pre y Post-orden provistas de Rank y Select) complementan dicha estructura para resolver las relaciones entre nodos en los operadores. Las sumas de prefijos proveen la información necesaria para recuperar la información (segmentos del texto comprimido). Finalmente, las características de la compresión del texto permite construir el resultado con acceso local y adyacente, y por ende con buen desempeño en memoria secundaria.

Esta estructura tiene dos importantes características frente a un enfoque como el que se plantea en [FLMM06], cuyo desempeño en compresión es excelente: Si bien el manipular separadamente la estructura y el contenido significa tener que recurrir a un puntero por cada nodo para enlazar con su contenido con la consecuente pérdida de eficiencia en la compresión, también es cierto que gracias a esta misma característica se logra: 1) la recuperación de la información de modo más eficiente, y 2) la búsqueda eficiente de patrones con calce parcial en consultas XPath del tipo $//a//b$.

Capítulo 6

Conclusiones y Recomendaciones

Considerando la gran relevancia que ha adquirido el formato XML y el crecimiento sostenido que han presentado diversas fuentes de información de carácter distribuido, es que el estudio de la información semi-estructurada cobra relevancia hoy en día.

Caracterizadas por una diversidad de fuentes, modelos y contenidos, los documentos XML (y otros formatos semi-estructurados como HTML y LaTeX), presentan un importante problema de almacenamiento y capacidad de hacer consultas, planteando la necesidad de disponer de dicha información de un modo adecuado.

En ese sentido, este estudio abordó el problema del almacenamiento en memoria secundaria de información semi-estructurada, a través de la implementación de la capa de almacenamiento de una aplicación particular llamada XPN, que almacena documentación en formato estándar XML. Sin embargo, muchas de las propuestas aquí realizadas son extensibles al caso general.

La implementación de las operaciones de consulta sobre los datos se abordó a través del modelo Proximal Nodes [NBY97], y la consulta sobre el contenido fue restringida a Lenguaje Natural.

Los resultados obtenidos son satisfactorios, pues la propuesta se presenta como una alternativa competitiva a las soluciones existentes que no realizan compresión de las colecciones, y supera largamente a las que sí lo hacen.

Por otro lado, el desarrollo de la aplicación está lejos de cerrarse gracias a la amplia gama

de extensiones que se pueden explotar.

6.1. Objetivos propuestos y cumplimiento de ellos

Como hemos visto en el capítulo anterior, se ha propuesto un novedoso sistema destinado al almacenamiento estático de colecciones XML, con muy buen desempeño en la compresión y con soporte sobre un subconjunto muy relevante de consultas XQuery.

A través de lo presentado en el Capítulo 4, se puede establecer que se ha cumplido con los objetivos propuestos en la Sección 1.2 del siguiente modo:

- Mediante el almacenamiento basado en palabras ETDC, la marca en una secuencia de bits para el tipo de palabra y la Suma Parcial para Offset de Fichero, se almacena de forma comprimida un documento semi-estructurado permitiendo hacer consulta local de modo eficiente.

- Mediante el Dataguide y las secuencias provistas de Rank y Select implementadas a través de Wavelet Trees, se obtiene un método para consultar eficientemente un documento semi-estructurado de acuerdo al análisis de sus características estructurales.

- Mediante el Índice Invertido de Palabras y la Suma Parcial para Offset de Palabra se obtiene un método para consultar de modo eficiente un documento semi-estructurado que permite recuperar contenido de acuerdo a una búsqueda por palabras.

XPN mantiene su ventaja sobre las alternativas para colecciones de tamaño pequeño y mediano (ver figuras 4.4 a 4.23).

6.2. Recomendaciones

Dadas las características de XPN, se puede enumerar una serie de aplicaciones prácticas estableciendo como justificación su desempeño en determinados contextos:

1. XPN se presenta como una buena alternativa para hacer consulta sobre colecciones comprimidas debido a su gran desempeño, pues ofrece un excelente balance entre compresión y capacidad de hacer consultas.

2. XPN también es conveniente en contextos donde el uso de RAM es restringido o cuando el tamaño de la colección impide el manejo eficiente en ella, pues está orientado principalmente al almacenamiento en disco.

3. XPN no es apropiada para un contexto dinámico donde se requiere modificación permanente de la información, pues no provee capacidad de inserción o borrado sobre la colección.

4. XPN es conveniente tanto en colecciones heterogéneas como en colecciones estructuradas. Gracias al comportamiento general que proveen las secuencias de etiquetas provistas de Rank y Select sobre cualquier tipo, la aplicación no requiere de la programación de índices para acelerar las consultas.

5. Por su enfoque orientado al almacenamiento comprimido en memoria secundaria, XPN se presenta como una aplicación de tipo one-shot, es decir, como un procesador de consulta no sistemática.

6. Debido al énfasis de búsqueda en texto de lenguaje natural, XPN no es apropiado para un contexto de búsqueda en texto completo. Sin embargo, el enfoque es muy apropiado en el contexto del etiquetado XML, pues el vocabulario es mucho más restringido.

6.3. Trabajo Futuro

XPN posee aún varias líneas de investigación que no han sido desarrolladas. La más interesante, probablemente, es la que corresponde a la implementación de búsqueda en texto plano. Una posibilidad es la aplicación de Re-Pair y el modelo presentado en [GN07a]. Otro interesante enfoque, probablemente con mayor eficiencia temporal, es la integración del LZ index [Nav08] con el índice estructural propuesto aquí.

Otra línea interesante de desarrollo, más enfocada al modelo XML, es utilizar las ideas presentadas en XMill [LS00] para comprimir el contenido de acuerdo al tipo de datos que determine el esquema XML. Esto debiera permitir obtener una ganancia en los datos numéricos, por ejemplo, y una ganancia en los joins de contenido. Sumado a la técnica de múltiples diccionarios planteada en [AdN05], esto debiese producir buenos resultados tanto con GZIP como con Re-Pair y con compresores dedicados también, permitiendo así lograr buena compresión combinada con consultas XQuery.

En este sentido, cabe destacar que la función de administrar varios diccionarios para diferentes contextos y compresores dedicados se complementa muy bien con una estructura como el dataguide aquí propuesto, por tanto la integración de las tres técnicas debiera ser fácil de lograr en la práctica.

Por otro lado, dada la facilidad de contextualizar nodos de texto gracias a las secuencias de Pre y Post del índice invertido estructural, puede ser ventajoso renunciar al índice invertido de contenido en algunos casos (p.e. palabras muy ocurrentes) y recurrir a una búsqueda secuencial acotada a los segmentos que se identifiquen como potenciales de resultado.

Una última importante línea de desarrollo en el área de XML se abre con la incorporación de los espacios de nombres. Separando los prefijos del nombre en la denominación completa que se hace de la etiqueta, se puede aplicar mejor la compresión basada en lenguaje natural para la estructura del documento.

Bibliografía

[ABMP07] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. XQueC: A query-conscious compressed XML database. *ACM Transactions on Internet Technologies*, 7(2), 2007. 2.2.8

[AdN05] J. Adiego, P. de la Fuente, and G. Navarro. Combining structural and textual contexts for compressing semistructured databases. In *Proc. 6th Mexican International Conference on Computer Science (ENC)*, pages 68–73. IEEE Computer Society, 2005. 2.2.8, 6.3

[AFMZ06] L. Afanasiev, M. Franceschet, M. Marx, and E. Zimuel. XCheck: A platform for benchmarking XQuery engines. In *Proc. 32nd International Conference on Very Large Data Bases (VLDB)*, pages 1247–1250. ACM, 2006. 3.2.1

[AL02] M. Arenas and L. Libkin. A normal form for XML documents. In *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 85–96. ACM, 2002. 5.4

[ANd07a] 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. 2.2.8, 4.1.3, 5.4

[ANd07b] J. Adiego, G. Navarro, and P. de la Fuente. Using structural contexts to compress semistructured text collections. *Information Processing and Management (IPM)*, 43(769–790), 2007. 2.2.1, 2.2.8

[ANd08] J. Adiego, G. Navarro, and P. de la Fuente. Un prototipo para la consulta sobre documentos transformados con LZCS. In *Actas de las XIII Jornadas de Ingeniería del Software y Bases de Datos*, 2008. To appear. 2.2.8

[Arr08] D. Arroyuelo. An improved succinct representation for dynamic k-ary trees. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029. Springer, 2008. 2.2.4

[BDM+05] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. 2.2.4, 5.3

[BFLN08] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 2008. 2.2.4, 2.2.6

[BFN+08] N. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS. Springer, 2008. To appear. 2.2.6, 4.1.1

[BFNP07] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007. 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 4.1.1

[BFNP08] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. New adaptive compressors for natural language text. *Software - Practice and Experience (SPE)*, 2008. To appear. 2.2.1, 2.2.1

[BGv+06] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 479–490. ACM, 2006. 2.3.3, 2.3.3

[BHMR07] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689. ACM, 2007. 2.2.4

[BLOL06] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Proc. 5th International Workshop on Experimental Algorithms (WEA)*, LNCS 4007, pages 146–157. Springer, 2006. 2.2.2

[BW94] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994. 1.1, 2.2.8, 2.2.8

[BWC89] T. C. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, 1989. 2.2.1

[BYN02] R. A. Baeza-Yates and G. Navarro. XQL and proximal nodes. *Journal of the American Society for Information Science and Technology (JASIST)*, 53(6):504–514, 2002. 2.3.1, 2.3.3, 2.3.3

[BYS05] R. A. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3772, pages 13–24. Springer, 2005. 2.2.2, 2.2.3

[CDZ04] Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: an efficient XPath processing system. In *Proc. 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 47–58. ACM, 2004. 2.2.8

[Che01] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proc. 11th Data Compression Conference (DCC)*, pages 163–172. IEEE Computer Society, 2001. 2.2.8

[CM05] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In Mariano P. Consens and Gonzalo Navarro, editors, *SPIRE*, LNCS 3772, pages 1–12. Springer, 2005. 2.2.1

[CM07] J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 137–148. Springer, 2007. 2.2.2

[CN04] J. Cheng and W. Ng. XQzip: Querying compressed XML using structural indexing. In *Proc. 9th International Conference on Extending Database Technology (EDBT)*, LNCS 2992, pages 219–236. Springer, 2004. 2.2.8

[CN07] F. Claude and G. Navarro. A fast and compact web graph representation. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 105–116. Springer, 2007. 2.2.1, 2.2.1

[CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS. Springer, 2008. To appear. 2.2.4, 4.1.2, 4.1.2, 5.1

[CW84] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, 1984. 2.2.8

[DLOM00] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752. ACM, 2000. 2.2.2

[DLOM01] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In Adam L. Buchsbaum and Jack Snoeyink, editors, *Proc. 3rd International Workshop on Algorithm Engineering and Experimentation (ALENEX)*, LNCS 2153, pages 91–104. Springer, 2001. 2.2.2

[dNZBY98] E. S. de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. 21st Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 298–306. ACM, 1998. 2.2.1, 4.1.1

[DRR07] O. Delpratt, N. Rahman, and R. Raman. Compressed prefix sums. In *Proc. 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 4362, pages 235–247. Springer, 2007. 2.2.5

[DRR08] O. Delpratt, R. Raman, and N. Rahman. Engineering succinct dom. In Alfons Kemper, editor, *Proc. 11th International Conference on Extending Database Technology (EDBT)*, volume 261 of *ACM International Conference Proceeding Series*, pages 49–60. ACM, 2008. 2.2.8

[FG02] N. Fuhr and N. Gövert. Index compression vs. retrieval time of inverted files for XML documents. In *Proc. of the 11th International Conference on Information and Knowledge Management (CIKM)*, pages 662–664. ACM, 2002. 2.2.7

[FLMM06] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proc. 15th International Conference on the World Wide Web (WWW)*, pages 751–760. ACM, 2006. 2.2.8, 5.4

[FLMM07] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. http://homepage3.nifty.com/wpage/links/xbw.html, 2007. 2.2.4, 2.2.4, 2.2.8

[FNP08] A. Fariña, G. Navarro, and J. Paramá. Word-based statistical compressors as natural language compression boosters. In *Proc. 18th Data Compression Conference (DCC)*, pages 162–171. IEEE Computer Society, 2008. 2.2.6

[FR08] P. Ferragina and S. S. Rao. Tree compression and indexing. In *Encyclopedia of Algorithms*, pages 374–386. 2008. 2.1.4, 2.2.4, 2.2.4, 5.3

[FSC+03] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 1077–1080. ACM, 2003. 2.4.2

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. ACM, 2003. 2.2.4

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373. ACM, 2006. 2.2.4

[GN07a] R. González and G. Navarro. A compressed text index on secondary memory. In *Proc. 18th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 80–91. College Publications, UK, 2007. 2.2.6, 6.3

[GN07b] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227. Springer, 2007. 2.2.6

[GN08] R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In *Proc. 8th Latin American Symposium (LATIN)*, LNCS 4957, pages 374–386. Springer, 2008. 2.2.5

[GNP+06] S. Grabowski, G. Navarro, R. Przywarski, A. Salinger, and V. Mäkinen. A simple alphabet-independent FM-index. *International Journal of Foundations of Computer Science (IJFCS)*, 17(6):1365–1384, 2006. 2.2.6

[GvT03] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. 29th International Conference on Very Large Data Bases (VLDB)*, pages 524–525. ACM, 2003. 2.2.3, 8, 2.3.3

[GW97] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. 23th International Conference on Very Large Data Bases (VLDB)*, pages 436–445. ACM, 1997. 2.2.7, 5.4

[HSS03] W. Hon, K. Sadakane, and W. Sung. Succinct data structures for searchable partial sums. In *Proc. 14th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 2906, pages 505–516. Springer, 2003. 2.2.5

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 2.2.1

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE Computer Society, 1989. 2.2.4, 5.3

[LM00] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, November 2000. 2.2.1, 2.2.1

[LM01] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. 27th International Conference on Very Large Data Bases (VLDB)*, pages 361–370. ACM, 2001. 2.1.4

[LS00] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 153–164. ACM, 2000. 2.2.8, 6.3

[May99] W. May. Information extraction and integration with FLORID: The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik, Germany, 1999. Available from `http://www.informatik.uni-freiburg.de/~may/Mondial/`. 4.3.1

[MN08] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008. 38 pages. 2.2.5

[MPC03] J. Min, M. Park, and C. Chung. XPRESS: A queriable compression for XML data. In *Proc. 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 122–133. ACM, 2003. 2.2.8

[MR01] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. 2.2.4, 5.3

[MRR98] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. In *Proc. 18th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1530, pages 186–196. Springer, 1998. 2.2.4

[MRS01] J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succinctly. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 529–536. ACM, 2001. 2.2.4

[Mun96] J. I. Munro. Tables. In *FSTTCS*, LNCS 1180, pages 37–42. Springer, 1996. 2.2.4

[Nav08] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics (JEA)*, 2008. To appear. 6.3

[NBY95] G. Navarro and R. A. Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proc. 18th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 93–101. ACM, 1995. 2.3.1

[NBY97] G. Navarro and R. A. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *ACM Transactions on Information Systems (TOIS)*, 15(4):400–435, 1997. 1, 2.3.1, 2.3.2, 2.3.3, 6

[NLC06] W. Ng, W. Y. Lam, and J. Cheng. Comparative analysis of XML compression technologies. *World Wide Web Journal*, 9(1):5–33, 2006. 2.2.8, 2.4.3

[NLWL06] W. Ng, W. Y. Lam, P. T. Wood, and M. Levene. XCQ: A queriable XML compression system. *Knowledge and Information Systems (KAIS)*, 10(4):421–452, 2006. 2.2.8

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 61 pages, 2007. 2.2.6

[NO02] G. Navarro and M. Ortega. Implementación de un lenguaje de consultas para XML basado en Proximal Nodes. Memoria de Ingeniería, 2002. Universidad De Chile, Facultad De Ciencias Fisicas Y Matemáticas, Departamento De Ciencias De La Computación. 1.3, 2.1.2

[NO03] G. Navarro and M. Ortega. IXPN: An Index-Based XPath Implementation. Technical Report 5, Universidad De Chile, Facultad De Ciencias Fisicas Y Matemáticas, Departamento De Ciencias De La Computación, Santiago, Chile, 2003. 1.1, 2.3.2, 2.3.2, 2.3.3, 2.3.3

[NR08] G. Navarro and L. Russo. Re-pair achieves high-order entropy. In *Proc. 18th Data Compression Conference (DCC)*, page 537. IEEE Computer Society, 2008. Poster. 2.2.1

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242. SIAM Press, 2002. 2.2.4, 2.2.5, 4.1.2, 4.1.2, 5.3

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003. 2.2.6

[Sad08] K. Sadakane. The ultimate balanced parentheses. Technical report, 2008. Unpublished, Personal Communication. 2.2.4

[Sak08] S. Sakr. An experimental investigation of XML compression tools. *The Computing Research Repository (CoRR)*, abs/0806.0075, 2008. 2.2.8

[SGS07] P. Skibiński, S. Grabowski, and J. Swacha. Effective asymmetric XML compression. *Software - Practice and Experience (SPE)*, 38(10):1027–1047, 2007. 2.2.8

[ST07] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th International Workshop on Algorithm Engineering and Experimentation (ALENEX)*, pages 71–83. SIAM, 2007. 2.2.2, 2.3.3

[SWK+01] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, I. Manolescu, and R. Busse. Why and how to benchmark XML databases. *ACM SIGMOD Record*, 35(3):27–32, 2001. 3.2.1

[SWK+02] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proc. 28th International Conference on Very Large Data Bases (VLDB)*, pages 974–985. ACM, 2002. 3.2.1

[Teu07] J. Teubner. Pathfinder: XQuery compilation techniques for relational database targets. In *Datenbanksysteme in Business, Technologie und Web (BTW)*, pages 465–474. GI, 2007. 2.2.8

[TH02] P. M. Tolani and J. R. Haritsa. XGRIND: A Query-Friendly XML compressor. In *Proc. 18th International Conference on Data Engineering (ICDE)*, pages 225–234. IEEE Computer Society, 2002. 2.2.8

[WLS07] R. K. Wong, F. Lam, and W. M. Shui. Querying and maintaining a compact XML storage. In *Proc. 16th International Conference on World Wide Web (WWW)*, pages 1073–1082. ACM, 2007. 2.4.1

[ZKÖ04] N. Zhang, V. Kacholia, and M. T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proc. 20th International Conference on Data Engineering (ICDE)*, page 54. IEEE Computer Society, 2004. 2.1.2, 2.4.1

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. 1.1

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. 1.1