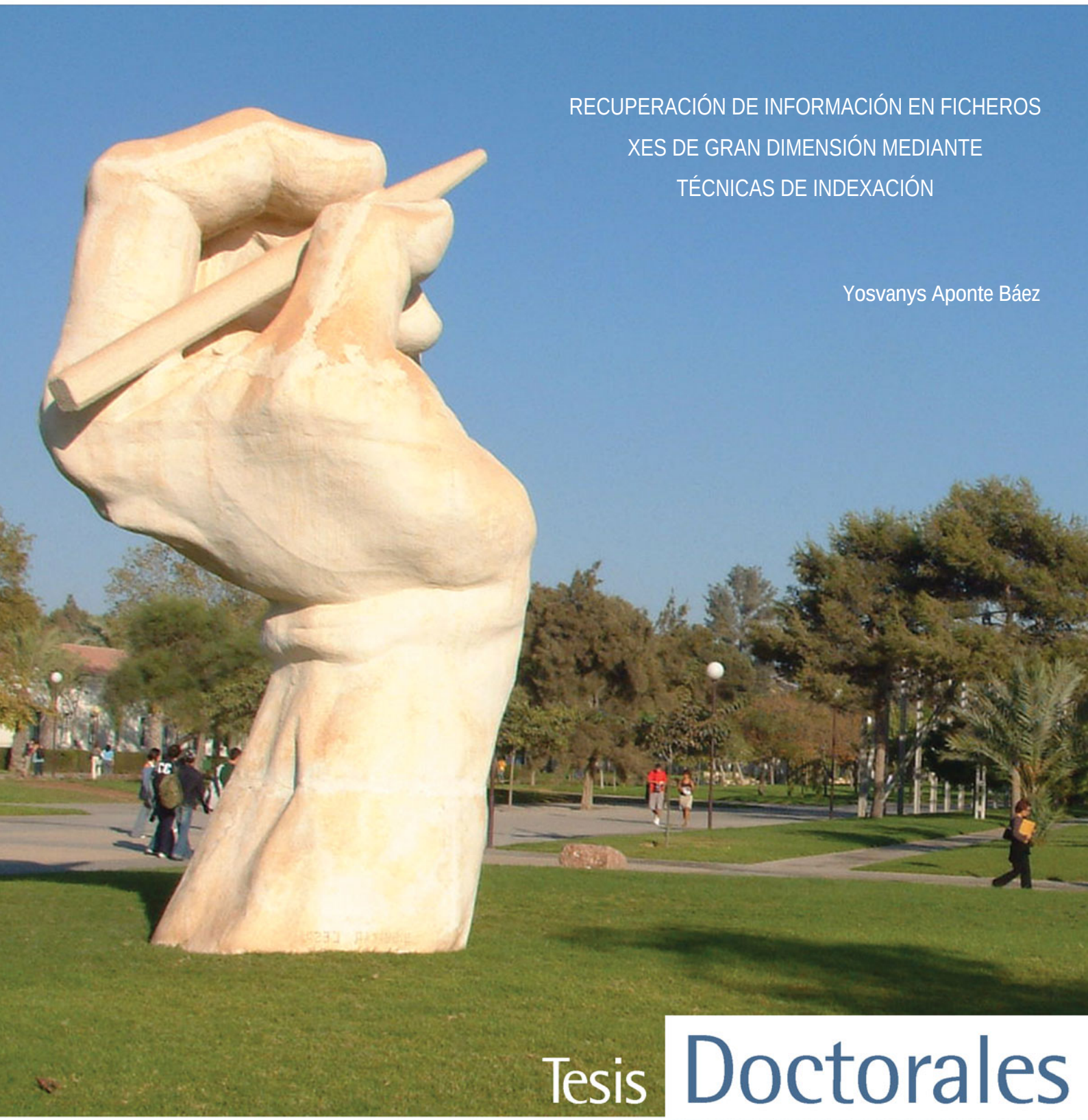
Universitat d'Alacant
Universidad de Alicante

RECUPERACIÓN DE INFORMACIÓN EN FICHEROS XES DE GRAN DIMENSIÓN MEDIANTE TÉCNICAS DE INDEXACIÓN

Yosvanys Aponte Báez



Tesis Doctorales

www.eltallerdigital.com

UNIVERSIDAD de ALICANTE

UNIVERSIDAD DE ALICANTE
Departamento de Lenguajes y Sistemas Informáticos

RECUPERACIÓN DE INFORMACIÓN EN FICHEROS XES DE GRAN DIMENSIÓN MEDIANTE TÉCNICAS DE INDEXACIÓN

Universitat d'Alacant
Universidad de Alicante

Tesis doctoral

Autor: Yosvanys Aponte Báez

Directores: Dr. Alexander Sánchez Díaz
Dr. Jesús Peral Cortés

Noviembre 2015

Universitat d'Alacant
Universidad de Alicante

TESIS DOCTORAL

RECUPERACIÓN DE INFORMACIÓN EN FICHEROS XES DE GRAN DIMENSIÓN MEDIANTE TÉCNICAS DE INDEXACIÓN

Este documento contiene una síntesis del trabajo realizado por Yosvanys Aponte Báez, bajo la dirección de los profesores Alexander Sánchez Díaz y Jesús Peral Cortés para optar al grado de Doctor en Informática.

Universitat d'Alacant
Universidad de Alicante

Noviembre 2015

Este trabajo ha sido parcialmente financiado por la Fundación Carolina y por el Departamento de Lenguajes y Sistemas Informáticos.

Universitat d'Alacant
Universidad de Alicante

Dedicatoria

Dedicado a
mi esposa, padres y hermano.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Agradecimientos

Me gustaría agradecer a mi tutor Alexander Sánchez Díaz. En lo profesional por su dedicación y guía en la realización de esta investigación, sobre todo en la escritura del documento de tesis. En el orden personal, gracias amigo por la paciencia y por supuesto gracias a Giselle por soportarme tantas veces en casa.

No podía faltar agradecer a mi esposa Yuselys, que en todo momento me decía que sí se puede, hasta en los momentos más difíciles de estos largos años. Agradecer a mi mamá y mi papá que han sido una guía para mi vida profesional y personal, sin ellos no hubiera llegado hasta aquí.

Quisiera dar las gracias al Departamento de Lenguajes y Sistemas Informáticos, especialmente por el trabajo inicial y por todas las ideas aportadas a esta investigación al profesor Rafael C. Carrasco Jiménez.

Agradecer también al profesor Manuel Marco Such por su apoyo constante pese a la distancia. Mercedes gracias también, sin ti este doctorado tampoco hubiera sido posible. A Annia y Antihus gracias por la preocupación en todo momento. Astrid, a ti mil gracias, por tus palabras de aliento que siempre estuvieron presentes. Agradecer también a Jesús Peral por el paso final y decisivo para la aprobación de esta investigación. Adrielys gracias por tu ayuda en la revisión de este documento. Agradezco también el apoyo brindado por Adel Balbisi en la elaboración de las pruebas experimentales.

Agradecer al departamento de Informática, a todos los profesores y amigos dentro y fuera de la universidad que han puesto su granito de arena en esta investigación. Y finalmente agradecer a todo aquel que pensó que esto era posible algún día haberlo terminado.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Índice general

Universitat d'Alacant
Universidad de Alicante

Índice de figuras

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Índice de tablas

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Índice de algoritmos

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Prefacio

Después de tantos años ha llegado el momento de escribir este prefacio, que trataré de hacerlo de la mejor manera. El desarrollo de la presente investigación surgió en el contexto del grupo de investigación PRONEG de la Universidad Agraria de La Habana. El problema fue identificado cuando una serie de algoritmos de descubrimiento de modelos para la Minería de Procesos, que se habían implementado en nuestro grupo de investigación, presentaban limitaciones cuando manipulaban grandes volúmenes de trazas. Aspecto este que afectaba en gran medida el desempeño de los algoritmos de descubrimiento de modelos y por tanto la detección de anomalías en tiempo real. El objetivo estaba en cómo lograr un análisis eficiente de los ficheros XES que almacenaban las trazas con grandes volúmenes de información. Por otra parte, desde los inicios en mi doctorado, he venido investigando en el área de la recuperación de información estructurada y sus técnicas de indexación. Por lo tanto, de conjunto con el profesor Alexander Sánchez Díaz y el resto de los integrantes del grupo de investigación, nos planteamos la tarea de desarrollar esta investigación mediante el empleo de técnicas de indexación para manejar eficientemente la información contenida en ficheros XES de gran dimensión, aprovechando su modelo lógico.

Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. Introducción

> El éxito no se logra sólo con cualidades especiales. Es sobre todo un trabajo de constancia, de método y de organización.
>
> J.P. Sergent

En este capítulo se explica la problemática del manejo eficiente de documentos XML de gran dimensión en el área de la recuperación de información en documentos estructurados. Esta problemática también está presente en la Minería de Procesos, específicamente en la necesidad del manejo eficiente de los archivos XES, que representan los datos de entrada de los algoritmos de descubrimiento de modelos. Los archivos XES en escenarios reales, contienen grandes volúmenes de trazas y son considerados como documentos estructurados de gran dimensión.

Se exponen además las principales técnicas de indexación desarrolladas hasta la fecha, que pueden ser extendidas para recuperar la información almacenada en ficheros XES de gran dimensión, ajustadas a su modelo lógico. Por último, se describe la motivación de la investigación, sus objetivos principales y la estructura del documento de tesis.

1.1 Documentos estructurados

Un documento estructurado es aquel que contiene una estructura predefinida a través del uso de etiquetas, con el objetivo de mostrar la información relevante del mismo. El estándar por excelencia para la representación de un documento estructurado es el XML[1] (*Extensible Markup Language* en

[1] http://www.w3.org/XML

```
<?xml version="1.0" encoding="UTF-8"?>
<log xes.version="1.0">
  <trace>
    <string key="concept:name" value="2"/>
    <event>
      <string key="concept:name" value="register"/>
      <string key="org:resource" value="Jones"/>
      <string key="activity" value="register request"/>
      <date key="time:timestamp" value="2012-01-05"/>
    </event>
  </trace>
</log>
```

Figura 1.1: Documento XML.

inglés). Se define como un metalenguaje que nos proporciona una manera sencilla de definición de lenguajes de etiquetas estructuradas, o también como un conjunto de reglas semánticas que nos permite la organización de información de distintas maneras (Harold, 1998). En la figura 1.1 se muestra un ejemplo de un documento XML.

Como plantean Gou and Chirkova (2007), se identifican dos modelos de datos básicos para representar la estructura jerárquica de un documento XML: el modelo de datos basado en el etiquetado de las aristas del árbol y el modelo basado en el etiquetado de los nodos del árbol. En la figura 1.2 se muestra un ejemplo del modelo de etiquetado basado en los nodos del árbol y en la figura 1.3 otro ejemplo del modelo de etiquetado basado en las aristas del árbol.

Existen básicamente tres tipos de nodos en un árbol XML. Por una parte, los nodos elementos y nodos atributos, que se conocen también como nodos internos del árbol y se corresponden con las etiquetas y los atributos de un XML, y los nodos valores o nodos hojas que contienen los datos en un documento XML.

Para que un documento XML se considere válido debe ajustarse a su DTD (*Document Type Definition* en inglés), donde se definen los elemen-

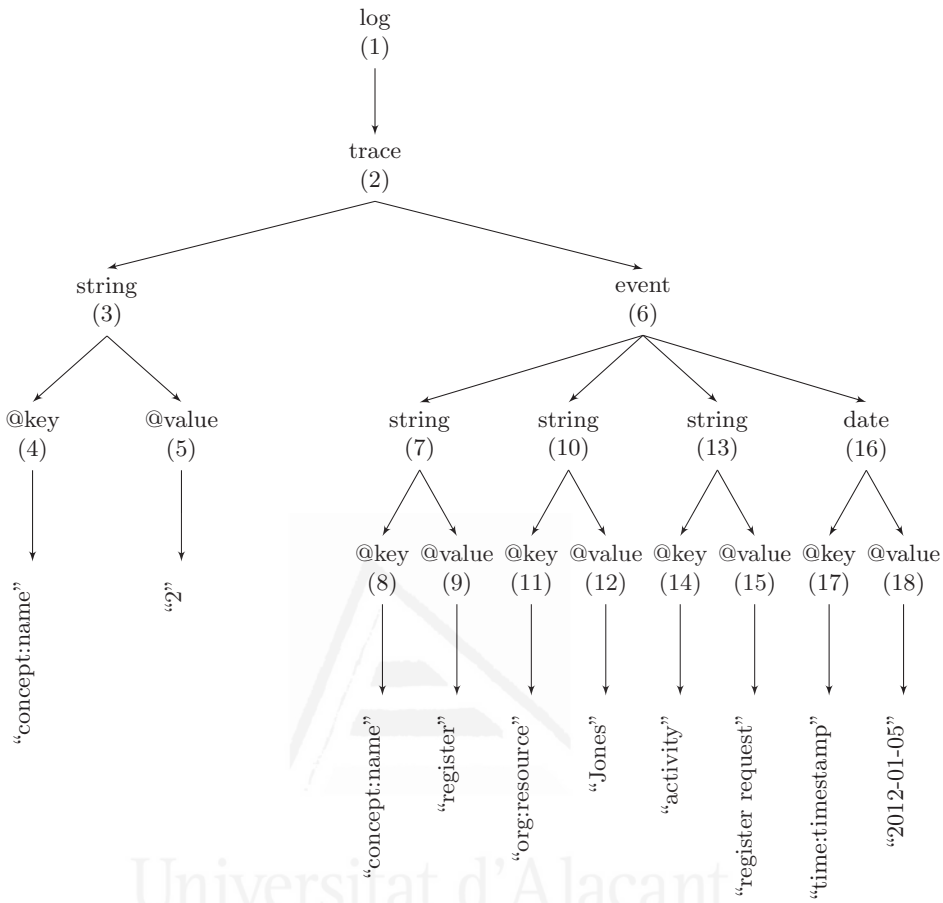
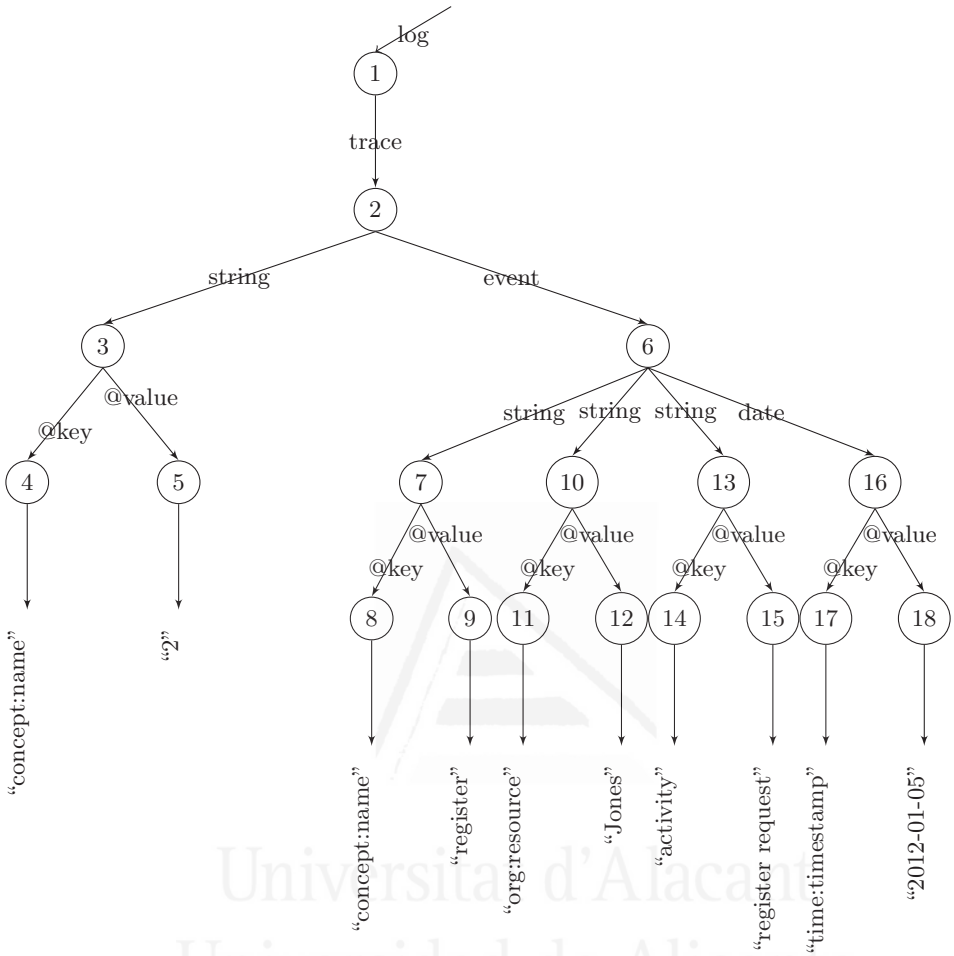


Figura 1.2: Modelado del XML basado en los nodos del árbol.

tos, atributos, entidades y las relaciones permitidas en un documento XML. Debido a diferentes limitaciones, surge *XML Schema*[2], considerado el sucesor de DTD. Una de las mejoras más importantes de *XML Schema* es el soporte de los tipos de datos, ya que permite especificar los valores de los elementos y atributos. Los tipos de datos atómicos más comunes son: *xs:string*, *xs:decimal*, *xs:integer*, *xs:boolean*, *xs:date* y *xs:time*. Una gran for-

[2]http://www.w3.org/XML/Schema



Universitat d'Alacant
Universidad de Alicante

Figura 1.3: Modelado del XML basado en las aristas del árbol.

taleza del *XML Schema* es que su sintaxis se escribe en el formato XML, por lo que editores y analizadores existentes pueden ser utilizados para su interpretación.

El estándar XML fue creado por la W3C[3], y ha sido ampliamente acep-

[3]http://www.w3.org

tado y adoptado para la representación e intercambio de información estructurada, motivando así, cada vez más la necesidad de almacenar dicha información. Esto conllevó a que una inmensa comunidad investigadora se dedicara al estudio y desarrollo de numerosas estrategias para el almacenamiento y la recuperación de información estructurada.

Según Chaudhri et al. (2003), se ha planteado la distinción entre los XML "centrados en datos" en contraposición con los "centrados en documentos". Los primeros están caracterizados por una estructura relativamente homogénea y un contenido textual fuertemente tipado. En cambio, los segundos se caracterizan por la presencia de contenido mixto, es decir, estructuras muy variables, típicas de los documentos con predominio de contenido textual.

Entre las primeras propuestas, para el almacenamiento de documentos XML, se pueden mencionar las bases de datos tradicionales, que se dividen en bases de datos relacionales y bases de datos orientadas a objetos. Para el almacenamiento en una base de datos se requiere del uso de un esquema predefinido. La estructura del documento debe ser trasladada a las diferentes tablas, por lo que se pierde el orden de los elementos insertados. Además se requiere de un tiempo elevado para la ejecución de los algoritmos de uniones entre las diferentes tablas, sobre todo cuando se procesan consultas complejas. Ejemplos de esta clasificación, basados en bases de datos relacionales, son STORED (Deutsch et al., 1999), XREL (Yoshikawa et al., 2001) y XISS/R (Harding et al., 2003).

Para evitar los altos costes de uniones entre tablas surgieron las llamadas bases de datos nativas XML, las cuales son utilizadas con frecuencia para documentos XML "centrados en documentos", pero en algunos casos, dependiendo de las características del XML y las necesidades del usuario, se pueden emplear para documentos XML "centrados en datos".

Las bases de datos nativas contienen como unidad fundamental de almacenamiento el modelo lógico del XML e implementan técnicas de indexación optimizadas y algoritmos de búsqueda que aceleran el acceso a la información almacenada. Tamino (Schöning, 2001), TIMBER (Jagadish et al., 2002), eXist (Meier, 2003) y Natix (Brantner et al., 2005), son ejemplos de este tipo de bases de datos.

Por último, están las propuestas híbridas que almacenan partes o el documento completo en su forma nativa en diferentes estructuras. Además los usuarios pueden recuperar datos relacionales o datos estructurales utilizando consultas SQL o consultas *XQuery*[4] respectivamente. Todas estas propuestas tienen en común la implementación de técnicas de indexación y búsqueda para la recuperación eficiente de información en documentos estructurados. System RX (Vanja et al., 2005) y las implementaciones realizadas por Mlynkova (2008) y Hall and Strömbäck (2010), constituyen ejemplos de esta clasificación.

1.2 Recuperación de documentos estructurados

Según la definición dada por Lalmas and Baeza-Yates (2009), la recuperación en documentos estructurados trata sobre la recuperación de fragmentos de documentos. La estructura del documento, que sea explícitamente proporcionada por un lenguaje de marcado o derivado, es explotada para determinar el fragmento más relevante del documento y retornarlo como respuesta a la consulta. La identificación del fragmento más relevante del documento puede ser utilizado, por sí mismo, para la identificación del documento más relevante a retornar como respuesta a la consulta realizada.

Para recuperar información en un documento XML se han definido diferentes lenguajes de consultas tales como: *XPath*[5] y *XQuery*. Por una parte, *XPath* (*XML Path Language* en inglés) fue introducido en su versión 1.0 en el año 1999 por la W3C. Es un lenguaje que permite construir expresiones que recorren y seleccionan los nodos de un documento XML. La idea es parecida a las expresiones regulares, para seleccionar partes de un texto sin atributos. *XPath* ofrece una multitud de ejes para navegar en un documento XML, donde el punto inicial de la navegación es el nodo contexto. En la tabla 1.1 se resumen los principales ejes definidos por *XPath*.

En el contexto del área de recuperación de información estructurada, como plantea Hammerschmidt (2006), tiene sentido clasificar las expresiones

[4] http://www.w3.org/TR/xquery
[5] http://www.w3.org/TR/xpath

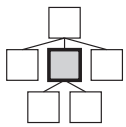
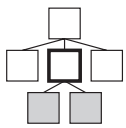
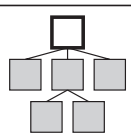
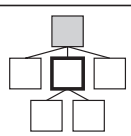
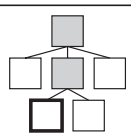
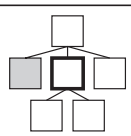
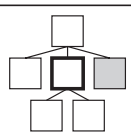
Eje	Representación (simplificada)	Ilustración
Nodo contexto	self::(.)	
Hijos del nodo contexto	child::(/)	
Descendientes del nodo contexto	descendant::(//)	
Padre del nodo contexto	parent::(..)	
Antecesores del nodo contexto	ancestor::	
Nodos que están antes del nodo contexto	preceding::	
Nodos que están después del nodo contexto	following::	

Tabla 1.1: Principales ejes de XPath.

Universitat d'Alacant
Universidad de Alicante

XPath en diferentes tipos de consultas:

- consultas estructurales o consultas puras de caminos (/log/trace/event/string/@key),
- consultas con valores o con predicado (/log/trace/event/string[@value ="Pete"]),
- consultas con comodines (/log/trace/event/*/@key) y
- consultas con descendencias (/log//string/@key).

Por otra parte, *XQuery* es un lenguaje de programación funcional que utiliza expresiones *XPath* para seleccionar nodos. *XQuery* es considerado un superconjunto de *XPath*. Esto implica que cada expresión *XPath* es ya una expresión *XQuery* válida y devolverá el mismo resultado. En contraste con *XPath*, que solo selecciona los nodos de un documento dado, *XQuery* permite transformar los nodos y crear nuevas estructuras con plantillas XML.

En comparación con los sistemas de recuperación de información tradicionales, la recuperación de información en documentos estructurados ha generado múltiples retos en cuanto a los tipos de consultas que se pueden ejecutar (Liu et al., 2004). Por ejemplo, los sistemas de recuperación de información tradicionales se centran en las consultas solo de contenido (*content only queries* en inglés), mientras que las técnicas de recuperación de información en documentos estructurados se centran en las consultas de contenido y en las consultas de contenido y estructura (*content and structure queries* en inglés). En la próxima sección se describen los principales retos en el área de recuperación de información en documentos estructurados.

1.2.1 Principales retos en el área de recuperación de información en documentos estructurados

Desde los inicios de la recuperación de información en documentos estructurados, diferentes autores han discutido múltiples retos o desafíos a tener en consideración. Por ejemplo, definición de las partes del documento a

recuperar y partes del documento a indexar, tratamiento de los elementos anidados, empleo de términos estadísticos, heterogeneidad del esquema, entre otros (Fuhr and Lalmas, 2006).

Como plantea Lalmas and Baeza-Yates (2009), un área abierta de gran importancia en la recuperación de documentos estructurados se basa en la parte del documento a recuperar. Para enfrentar esto, diferentes soluciones utilizan el principio básico de la recuperación de información en documentos estructurados: *"un sistema siempre debe recuperar la parte más específica de un documento como respuesta a una consulta"* (Chiaramella et al., 1996).

Otro de los desafíos, según Lalmas and Baeza-Yates (2009), consiste en cómo describir las diferentes unidades de indexación, es decir, las partes del documento a indexar. Entre las principales técnicas que abordan este desafío se encuentran:

- Agrupación de nodos en pseudo-documentos no superpuestos (desventaja: los pseudo-documentos generados pueden no tener sentido para el usuario, es decir, unidades sin coherencia).
- Indexación de arriba-abajo (desventaja: la relevancia de los elementos mayores no es un buen predictor de los menores subelementos dentro de ellos).
- Indexación de abajo-arriba (desventaja: la relevancia de un elemento hoja a menudo no es un buen predictor de la relevancia de los elementos donde está contenido).
- Indexación de todos los elementos (desventaja: se indexan muchos elementos que no son relevantes en la búsqueda y además pueden contener redundancia).

El tratamiento de los elementos anidados en un documento XML también ha sido un gran reto en esta área y por lo tanto, motivo de estudio y discusión. Las estrategias a seguir pueden ser las siguientes:

- descartar los elementos pequeños,

Universitat d'Alacant
Universidad de Alicante

- descartar los tipos de elementos que los usuarios no utilizan a menudo,
- descartar los tipos de elementos que los asesores no juzgan como relevantes y
- solo dejar los tipos de elementos que han sido categorizados como importantes para la búsqueda.

Aún con todas estas estrategias los documentos pueden continuar con elementos anidados, por lo que se plantea la utilización de términos estadísticos como el modelo *tf-idf* (*term frequency-inverse document frequency* en inglés), para distinguir los diferentes contextos de un término dentro de una colección de documentos (Lalmas, 2009).

Para el caso del uso de términos estadísticos, (Lalmas and Trotman, 2009) describen cómo obtener sus elementos y la recolección de datos estadísticos mediante el empleo del modelo *tf-idf*. Por una parte, el uso de estadísticas de relaciones determina qué subelementos contribuirán mejor al contenido de su nodo padre o viceversa. Además, discuten cómo estimar estadísticas de relación de nodos (por ejemplo: tamaño, número de hijos, profundidad y distancia). Por otra parte, el uso de estadísticas basadas en la estructura del árbol permite determinar que elementos son una buena unidad de indexación y cómo estimar las estadísticas de estructura (frecuencia, tamaño y profundidad).

Lo ideal para un documento XML es que presente un solo esquema y que el usuario esté familiarizado, pero en la práctica esto es poco común. Existen esquemas complejos que pueden cambiar con el paso del tiempo, por lo tanto la ejecución de consultas estructurales por parte del usuario, en ocasiones, puede presentarse como un aspecto crítico.

El estudio y discusión de estos y otros retos, han llevado consigo al desarrollo e implementación de diferentes técnicas de indexación y búsqueda que inciden en una recuperación de información en documentos estructurados más eficiente. A continuación se describen las diferentes técnicas de indexación sobre bases de datos nativas XML y cómo se han clasificado a lo largo del tiempo.

Universitat d'Alacant
Universidad de Alicante

1.2.2 Técnicas de indexación para bases de datos nativas XML

Las técnicas de indexación juegan el rol principal durante la manipulación y recuperación de información en documentos estructurados en las bases de datos nativas XML. Según Zemmar et al. (2011), entre los criterios más importantes se encuentran: estructura de datos utilizada por el índice, estructura del índice y los parámetros de entrada y salida del índice. Con respecto al segundo criterio, numerosos autores han propuesto diferentes clasificaciones de las técnicas de indexación. En la presente investigación se utiliza una clasificación basada en los criterios de Hammerschmidt (2006) y Mohammad and Martin (2010a):

- técnicas basadas en caminos,
- técnicas basadas en el etiquetado de los nodos,
- técnicas basadas en secuencias y
- técnicas híbridas.

Las técnicas de indexación basadas en caminos (Yan and Liang, 2005; Grimsmo, 2008; Mohammad and Martin, 2010b) crean resúmenes estructurales de todos los caminos mediante el recorrido del árbol XML desde el nodo raíz hasta los nodos hojas. Presentan muy buenas prestaciones en la resolución de consultas simples o que comienzan desde el nodo raíz. Su principal debilidad está dada, en ocasiones, con el rápido crecimiento del índice cuando se manejan documentos XML heterogéneos.

Las técnicas de indexación basadas en nodos (O'Neil et al., 2004; Lu et al., 2011; Assefa, 2012) emplean esquemas de etiquetado que utilizan las posiciones de los nodos dependiendo del recorrido del árbol XML. Se consideran un eslabón fundamental para los algoritmos de uniones estructurales y para los índices de resúmenes estructurales. La principal debilidad que presentan estas técnicas es la cantidad de uniones estructurales que es necesario establecer para procesar consultas complejas.

Estas técnicas también son conocidas en la literatura como: esquema de etiquetado, esquema de codificación o esquema de numeración. A diferencia de las técnicas de indexación de caminos, estas pueden determinar eficientemente relaciones jerárquicas entre diferentes nodos.

Las técnicas de indexación basadas en secuencias (Rao and Moon, 2004; Ferragina and Manzini, 2005; Li et al., 2013) tienen como objetivo principal disminuir el tiempo de ejecución de los algoritmos de uniones estructurales. Estas técnicas representan, tanto a los documentos como a las consultas por secuencias y subsecuencias, de modo que una consulta puede ser procesada mediante el cotejo de la secuencia del documento y de la consulta. Estas técnicas tienen como fortaleza fundamental el procesamiento eficiente de consultas estructurales, ya que no es necesario la utilización de algoritmos de uniones estructurales.

Finalmente, las técnicas de indexación híbridas (Liao et al., 2010; Hsu et al., 2012b; Hsu and Liao, 2013) presentan de una forma u otra, características de las diferentes técnicas descritas anteriormente. Estas técnicas por lo general, incluyen un índice estructural y un índice de valores o de contenidos.

Todas las técnicas de indexación antes mencionadas no son lo suficientemente adecuadas para la manipulación de grandes volúmenes de datos. Por ejemplo, los archivos en formato XES (Günther and Verbeek, 2009) utilizados en la Minería de Procesos por los algoritmos de descubrimiento, en ocasiones almacenan trazas de procesos informáticos muy densas, por el alto volumen de información. Estas técnicas de indexación requieren ser optimizadas para gestionar eficientemente la información almacenada en los ficheros XES. En escenarios reales donde las anomalías se deberían predecir y analizar en un corto período de tiempo es una tarea crítica.

Diferentes investigadores (Hsu et al., 2012a, 2014) han planteado el diseño de nuevas técnicas secuenciales o el rediseño de las ya existentes a parelelas/distribuidas, para el manejo de grandes volúmenes de documentos estructurados. Un estudio detallado de la estructura lógica del estándar XES, permite el diseño de técnicas de indexación más eficientes y personalizadas para la manipulación de grandes volúmenes de información en los diferentes escenarios de la Minería de Procesos.

Universidad de Alicante

1.3 Minería de procesos

Para las empresas es de vital importancia obtener información detallada sobre sus procesos de negocios, basada fundamentalmente en su funcionamiento y permitiéndoles encontrar anomalías y posibles mejoras. Una solución eficiente a este problema se encuentra en el área de la Minería de Procesos, donde sus técnicas principales tienen como objetivos: descubrir, mejorar y monitorear los procesos de negocios, a través de la extracción de conocimientos de los registros de eventos que se encuentran disponibles en los actuales sistemas de información. Es una tecnología relativamente joven y a pesar de esto múltiples empresas la están incorporando con el objetivo de mejorar sus procesos de negocios.

Los registros de eventos están compuestos por instancias de procesos y cada instancia contiene información relacionada con la ejecución de una acción u operación específica que se ha ejecutado en un sistema y su marca de tiempo correspondiente (Verbeek et al., 2011). Desde el año 2010 la IEEE[6] (*Institute of Electrical and Electronics Engineers* en inglés) adoptó un nuevo formato para el almacenamiento de los registros de eventos: el estándar XES (*eXtensible Event Stream*, Günther and Verbeek, 2009), considerado el sucesor del MXML.

El principal propósito del estándar XES es ofrecer un formato de intercambio de registros de eventos entre herramientas y dominios de aplicaciones (Arturo et al., 2015). En un XES los elementos *log*, *trace* y *event* solo definen la estructura del documento, ellos no contienen ninguna información por sí solos. Para almacenar información en un XES es necesario la utilización de atributos. Cada atributo está basado en una llave que define el tipo de dato y su valor correspondiente. Los tipos de datos pueden ser: *string*, *date*, *integer*, *float*, *boolean* e *ID*. La semántica precisa de un atributo se define por su extensión, lo que podría ser o bien una extensión estándar o alguna extensión definida por el usuario. En la figura 1.4 se muestra el modelo lógico del estándar XES.

Por otra parte, los clasificadores de eventos se definen en el elemento

[6] http://www.ieee.org

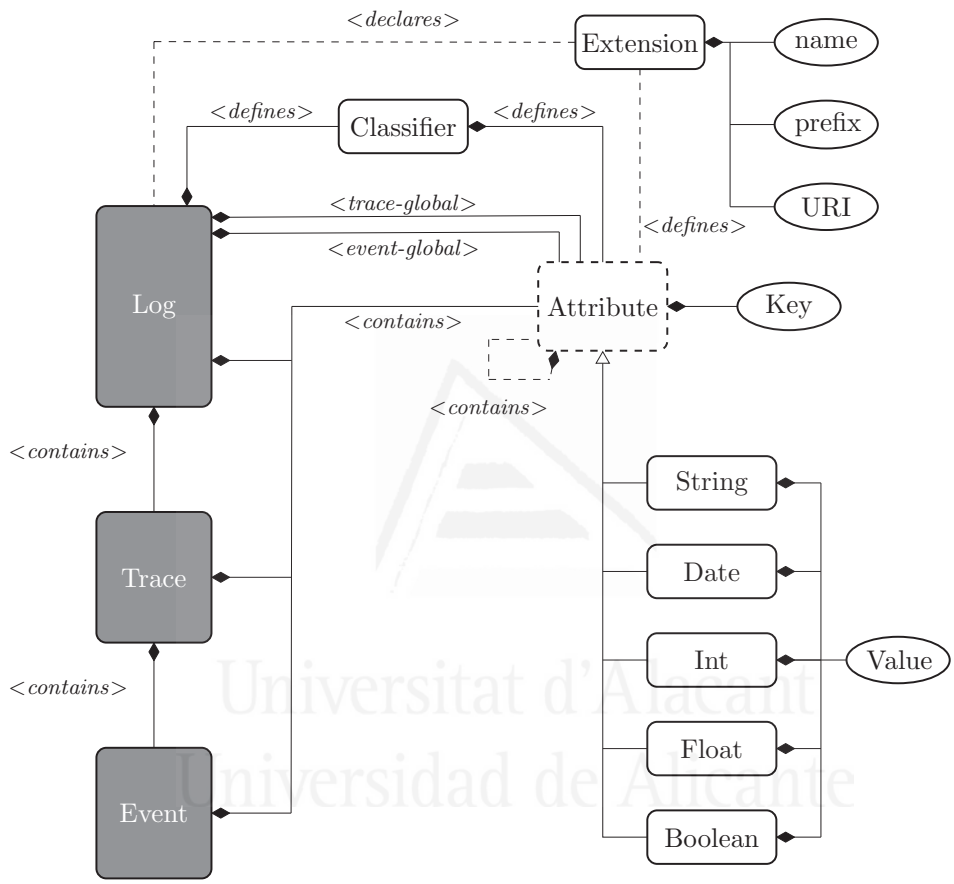


Figura 1.4: Modelo lógico del XES (Günther and Verbeek, 2009).

log, los cuales asignan una identidad a cada evento, que lo hace comparable con el resto. Los clasificadores son también definidos por un conjunto de atributos. El hecho de que ciertos atributos tienen valores bien definidos para cada traza y/o evento, es producto de su definición a nivel global en el elemento *log*.

Los registros de eventos contenidos en los ficheros XES, son la entrada de los algoritmos de descubrimiento utilizados por algunas de las herramientas especializadas como PROM (Günther and van der Aalst, 2006) y XESame (Buijs, 2010), donde su eficiencia es proporcional al tamaño del fichero. En algunos casos donde los procesos de negocios son muy densos (decenas de millones de trazas) y sea necesario la detección de anomalías en tiempo real, es preciso entonces, la utilización de analizadores sintácticos de ficheros XES eficientes y escalables.

En el marco del proyecto[7] desarrollado en la Universidad Agraria de la Habana se implementaron diferentes algoritmos de descubrimiento de modelos que se pusieron en práctica en diferentes escenarios económicos del país. Las prestaciones de los algoritmos se vieron afectadas cuando manipulaban archivos XES de gran dimensión. A partir de esto surgió la necesidad de diseñar e implementar estrategias de análisis para el manejo eficiente de estos archivos XES de gran dimensión. Se implementaron soluciones que utilizan modelos DOM para la carga en memoria del XES y el empleo de analizadores *lazy*. Dichas soluciones no fueron del todo eficiente para escenarios que manipulan grandes volúmenes de datos en tiempo real. En la siguiente sección se describe cómo es tratado en la literatura el manejo de grandes volúmenes de información o *Big Data*.

1.4 Tratamiento de grandes volúmenes de datos

El término *Big Data* es utilizado como: recolección, almacenamiento, gestión, visualización y la vinculación de grandes volúmenes de datos. Se incluyen además, los sistemas y las herramientas que se utilizan para analizar

[7]Proyecto PRONEG con el Centro de Investigaciones de Tecnologías Integradas (período de ejecución 2013-2015).

el valor de la información. Otros aplican a *Big Data* para toda aquella información que no puede ser procesada o analizada utilizando procesos o herramientas tradicionales (Dominguez and Yeja, 2015).

Big data plantea, básicamente, tres retos fundamentales sobre el flujo de datos (Schönberger and Cukier, 2013):

- Volumen: saber cómo gestionar e integrar grandes volúmenes de datos, procedentes de fuentes heterogéneas.

- Velocidad: poder acceder a la plataforma desde cualquier lugar, de forma autónoma por cualquier usuario de negocio, para mejorar y agilizar la toma de decisiones mediante la automatización: programación de acciones, eventos y alarmas.

- Variedad: conseguir unificar contenidos dispersos y no estructurados, con datos históricos, actuales y/o predictivos para un manejo óptimo de los mismos y para extraer de ellos información de valor.

Muchos investigadores y grandes empresas han afrontado la problemática del *Big Data* desde diferentes ángulos. Desde luego, el ángulo que actualmente tiene mayor liderazgo, en términos de popularidad, para analizar enormes cantidades de información es la plataforma de código abierto Hadoop (Lam, 2010). A pesar de esto, existen plataformas que compiten con Hadoop en el escenario de *Big Data*. Por ejemplo, el proyecto Spark[8], también de código abierto, y soluciones de HPCC Systems[9] y Pervasive Software[10] son otras de las propuestas que se encuentran en el mercado.

Hadoop es una plataforma que permite procesar datos usando el modelo de programación MapReduce (Dean and Ghemawat, 2010). Está diseñado para procesar grandes volúmenes de información de manera eficiente a través de la computación distribuida, conectando ordenadores y coordinándolos para trabajar en paralelo. Hadoop está inspirado en el proyecto

[8] http://spark.apache.org
[9] http://hpccsystems.com
[10] http://www.pervasive.com

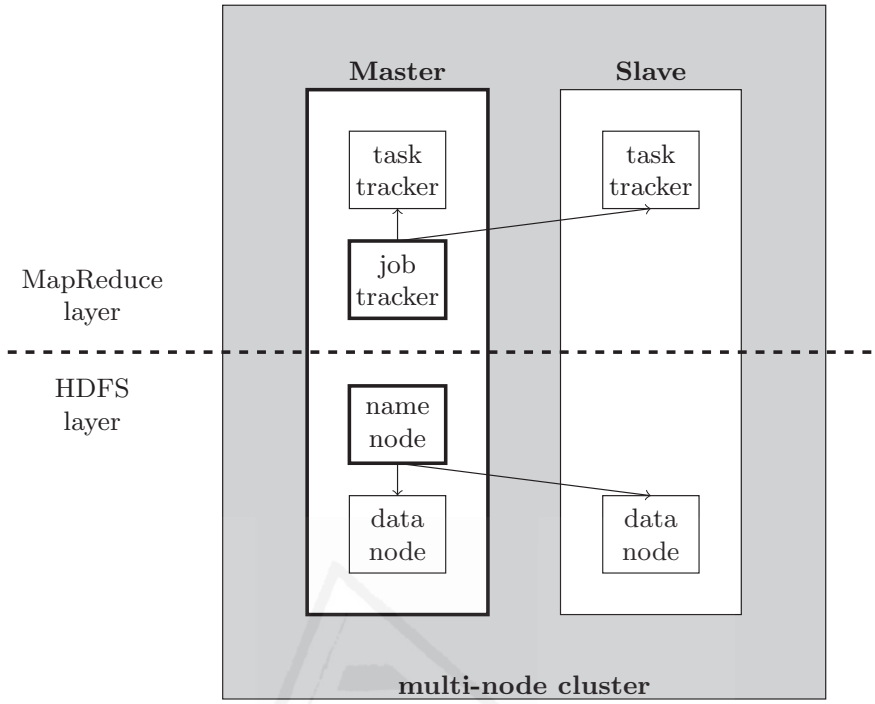


Figura 1.5: Arquitectura de la plataforma Hadoop (Lam, 2010).

de Google File System (Ghemawat et al., 2003) y basado en una arquitectura maestro/esclavo (*master/slave* en inglés). Contiene un nodo maestro y varios nodos esclavos (véase la figura 1.5).

Un nodo maestro contiene un trabajador (*worker* en inglés) que distribuye las diferentes tareas por el nodo maestro y por los diferentes nodos esclavos. Presenta dos capas fundamentales: la capa donde se ejecutan los procesos MapReduce y la capa del sistema de archivo HDFS. Los archivos en Hadoop son divididos en pequeñas piezas llamadas bloques y distribuidas en todo el clúster, de esta manera el proceso MapReduce puede ser ejecutado en pequeños subconjuntos, y esto provee la escalabilidad necesaria para el procesamiento de grandes volúmenes de información.

Hadoop implementa un paradigma computacional llamado MapReduce, el cual divide el procesamiento en dos funciones: *map* y *reduce*. Cada

una de ellas utiliza pares clave-valor como entradas y salidas. MapReduce se puede definir también como un modelo de programación distribuida que permite el procesamiento masivo de datos a gran escala de manera paralela. Desarrollado como alternativa escalable y tolerante a fallos para el procesamiento masivo de datos (Calle Jaramillo and Parrales Bravo, 2010).

La función *map* trabaja con grandes volúmenes de datos divididos en dos o más partes. Cada una de estas partes contienen listas de registros. Una función *map* es ejecutada para cada parte por separado, y calcula un conjunto de valores intermedios basados en el procesamiento de cada registro. MapReduce agrupa los valores de acuerdo a la clave intermedia y posteriormente los envía a la función *reduce*. Por otra parte, la función *reduce* se ejecuta para cada elemento de cada lista de valores intermedios que recibe. El resultado final se produce a través de la recopilación e interpretación de los resultados de todos los procesos ejecutados y es almacenado en el sistema de archivo HDFS.

En la actualidad muchos clientes utilizan la tecnología *Big Data* mediante la plataforma Hadoop para sus necesidades de negocios, donde la mayor actividad se registra alrededor de proyectos que requieren análisis en tiempo real, en los que el costo de procesamiento es alto y hay que analizar los grandes flujos de datos para la toma de decisiones en tiempo real. Por tal motivo es posible utilizar la plataforma Hadoop, para distribuir un índice con los contenidos textuales almacenados en un archivo XES de gran dimensión, que permita a los algoritmos de descubrimiento de modelos gestionar la información en tiempo real.

Universitat d'Alacant
Universidad de Alicante

1.5 Motivación, objetivos y estructura de la tesis

A pesar de lo positivo de las técnicas de indexación para la recuperación eficiente de información en documentos XML antes mencionadas, pueden presentarse algunas limitaciones. Primero, la necesidad de grandes volúmenes de memoria para el almacenamiento de la estructura del índice, donde en algunos casos puede ser mayor que el documento original (Chen et al., 2005). En otros casos, para reducir al mínimo el tamaño de la estructura del

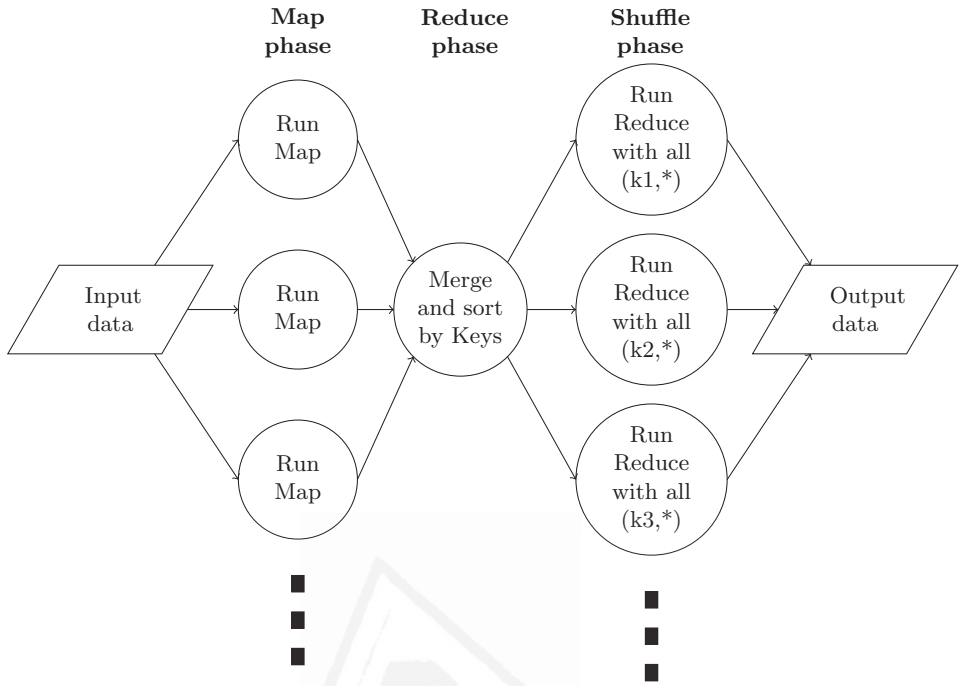


Figura 1.6: Paradigma de programación MapReduce (White, 2012).

índice es necesario un tiempo de ejecución crítico y generalmente su rendimiento se reduce cuando se analizan documentos XML de gran dimensión.

Además, la mayoría de las técnicas de indexación están diseñadas para el manejo de pequeñas colecciones de datos que se ejecutan de forma secuencial, por lo tanto el manejo de grandes volúmenes de documentos XML se ha presentado, hasta el momento, como un área crítica. El estudio de Liao et al. (2010) plantean la creación de nuevas técnicas de indexación o la adaptación de las ya creadas a entornos paralelos/distribuidos para lograr una manipulación eficiente de grandes volúmenes de datos.

En el marco del proyecto PRONEG se ha planteado la ejecución de esta tesis. A partir de una revisión extensa de la literatura en el área de recuperación de información en documentos estructurados se ha detectado la problemática sobre la necesidad del manejo eficiente de documentos

XML de gran dimensión. Además, la exigencia de un análisis eficiente de los archivos XES densos de trazas, que son la entrada de los algoritmos de descubrimiento de modelos en la Minería de Procesos, conllevó a la definición del problema de investigación siguiente: ¿cómo reducir los costes de procesamiento de archivos densos de trazas para favorecer el análisis de modelos de procesos en tiempo real?

Para dar solución a esta problemática se planteó el objetivo general siguiente: diseñar nuevos mecanismos de indexación para el manejo eficiente de consultas en tiempo real en archivos densos de trazas en formato XES.

Para dar total cumplimiento al objetivo general, se plantearon los siguientes objetivos específicos:

1. Proponer una estructura de índice basada en el modelo lógico del XES y que se pueda paginar en memoria.
2. Definir métodos que optimicen el tiempo de construcción del índice apoyados en un esquema de compresión del XES.
3. Optimizar el coste temporal de ejecución de las consultas sobre el índice, mediante el uso de estructuras basadas en arreglo de sufijos, árbol ternario de búsqueda y diccionario rank/select.
4. Proponer variantes paralelas/distribuidas de los algoritmos de construcción de índices creados y evaluar su eficiencia, escalabilidad y aceleración.
5. Definir mecanismos de consulta que aprovechen la estructura del índice.

Universitat d'Alacant
Universidad de Alicante

La novedad científica de esta investigación radica en el diseño de nuevas estrategias basadas en el manejo eficiente de índices estructurales y de contenidos para recuperar información en tiempo real en archivos XES de gran dimensión. Además presenta las siguientes novedades prácticas:

- Implementación de algoritmos de optimización mediante técnicas paralelas/distribuidas para reducir los tiempos de construcción del índice de contenidos del XES y su almacenamiento en memoria.

- Diseño de nuevos métodos para el manejo de archivos XES de gran dimensión utilizando índices, mediante la adaptación de estructuras de datos utilizadas en sistemas tradicionales de recuperación de información estructurada.

- Estudio comparativo de los costes temporales de las versiones secuenciales y paralelas/distribuidas.

Las diferentes técnicas de indexación, tanto secuenciales como paralelas/distribuidas, se revisan en el capítulo 2. En el capítulo 3 se estudia una propuesta de índice estructural basado en un arreglo de sufijos y complementado con un árbol ternario de búsqueda. A continuación, en el capítulo 4 se propone una optimización del índice de contenidos utilizando una estructura *rank/select* y se plantea una propuesta de construcción del mismo utilizando la plataforma paralela/distribuida Hadoop. Finalmente, se realizan las pruebas experimentales en el capítulo 5 y las conclusiones y trabajos pendientes en el capítulo 6.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

2. Estado de la cuestión

> Investigar es ver lo que todo el mundo ha visto, y pensar lo que nadie más ha pensado.
>
> Albert Szent Gyorgi

En este capítulo se analizan las diferentes técnicas de indexación y búsqueda para bases de datos nativas XML, que se ejecutan de forma secuencial y paralela/distribuida. Se describe la evolución en el tiempo y cómo han afrontado los diferentes retos en el área de la recuperación de información en documentos estructurados en las últimas dos décadas. También se describen las estructuras de datos utilizadas, los requisitos para su implementación y los tipos de consultas que soportan. Para cada una de las técnicas se emiten además valoraciones de las ventajas y desventajas que presentan.

Universitat d'Alacant
Universidad de Alicante

2.1 Técnicas secuenciales

Las técnicas de indexación son utilizadas para mejorar la eficiencia y la escalabilidad del proceso de consultas sobre cualquier documento XML, ya que reducen el espacio de búsqueda. Siguiendo los criterios de Hammerschmidt (2006) y Mohammad and Martin (2010a), las técnicas de indexación para bases de datos nativas XML se pueden clasificar en cuatro grandes grupos: basadas en los caminos del árbol, basadas en los nodos del árbol, basadas en secuencias e híbridas. Todas estas técnicas tienen en común que fueron diseñadas para ejecutarse de forma secuencial en un solo ordenador. Seguidamente se describen las principales técnicas que pertenecen a cada

```
<?xml version="1.0" encoding="UTF-8"?>
<log xes.version="1.0">
  <trace>
    <string key="concept:name" value="1"/>
    <event>
      <string key="concept:name" value="register request"/>
      <string key="org:resource" value="Pete"/>
      <string key="activity" value="register request"/>
      <date key="time:timestamp" value="2011-01-04"/>
    </event>
  </trace>
</log>
```

Figura 2.1: Documento XML.

clasificación. Todos los ejemplos mostrados en este capítulo, se basan en el documento XML de la figura 2.1.

2.1.1 Técnicas de indexación basadas en caminos

Las técnicas de indexación basadas en caminos crean resúmenes estructurales de todos los caminos mediante el recorrido del árbol del XML desde el nodo raíz hasta los nodos hojas. Una de las primeras técnicas en esta categoría la ocupa *DataGuide* (Goldman and Widom, 1997), donde cada camino almacenado es único. Existen dos variantes: *Minimal Dataguide* (véase la representación gráfica en la figura 2.2) y *Strong Dataguide* (véase la representación gráfica en la figura 2.3).

La primera variante es más compacta en cuanto a espacio, ya que contiene menos caminos y es más difícil de mantener, específicamente durante la inserción y actualización de los nodos. Por otra parte, la segunda variante difiere de la primera en cuanto a los mismos nodos que son alcanzados por caminos diferentes. En este caso, es creado un nuevo camino, en vez de agregar una referencia al nodo en común. Cada nodo del árbol en las dos variantes presenta una extensión con el nodo correspondiente en el XML original.

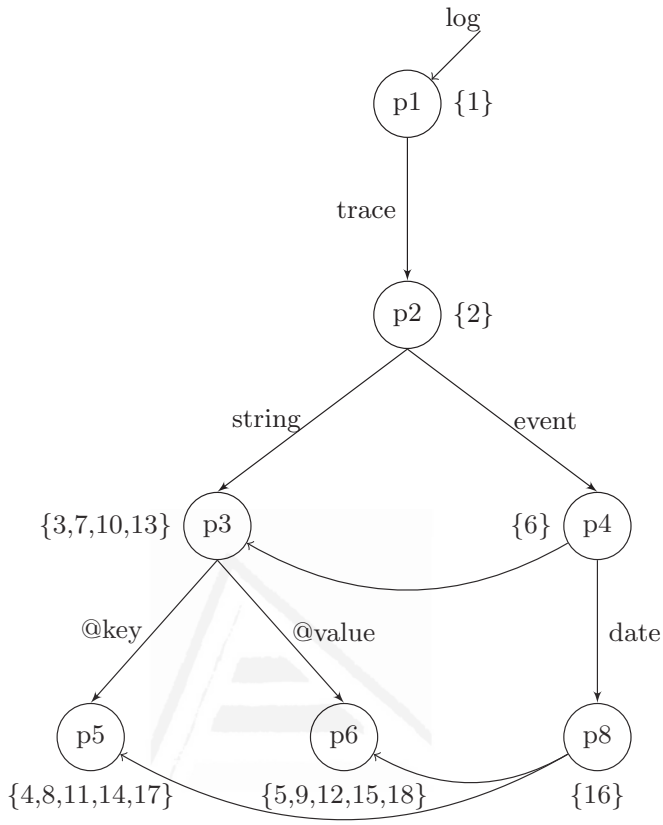


Figura 2.2: El índice Minimal DataGuide.

Universitat d'Alacant
Universidad de Alicante

La construcción de un *DataGuide* se puede ejecutar en tiempo lineal cuando los XML presentan una estructura arbórea. Cuando el XML presenta una estructura en forma de grafo puede llegar hasta orden exponencial, en el peor de los casos. Para estructuras irregulares un índice *DataGuide* puede ocupar más espacio que el ocupado por el documento original. Otro aspecto negativo es que solo se pueden ejecutar consultas con relaciones directas. Para las consultas con relaciones indirectas, la estructura de índice no guarda información de la jerarquía de los nodos, por lo tanto se necesita recorrer el XML original completo.

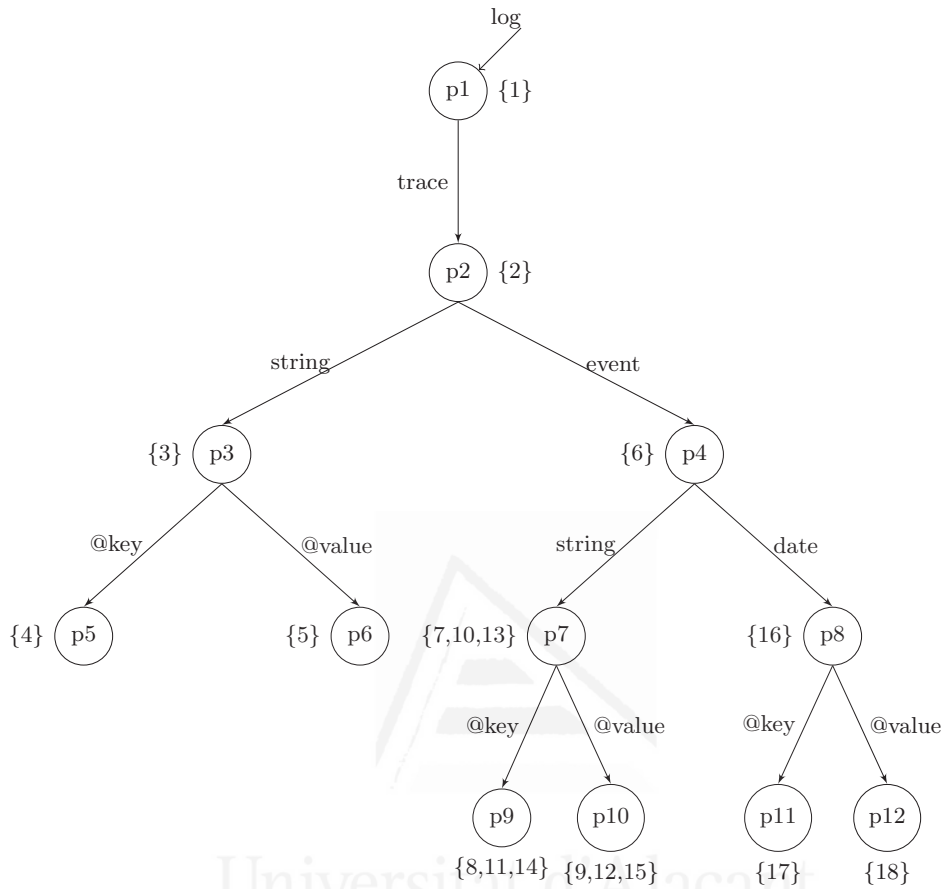


Figura 2.3: El índice Strong DataGuide.

Universitat d'Alacant
Universidad de Alicante

Para contrarrestar algunas de las limitantes anteriores, fue diseñado *ToXin* (Rizzolo and Mendelzon, 2001). En este caso se mantienen en el índice todos los valores de los caminos y la información de la navegación de los nodos, logrando así una mejor ejecución de las consultas con relaciones indirectas y ramificadas.

Por otra parte, *IndexFabric* (Cooper et al., 2001) utiliza una estructura arbórea para almacenar todos los caminos y emplea una clave que corresponde a una secuencia de caracteres para la organización y la navegación

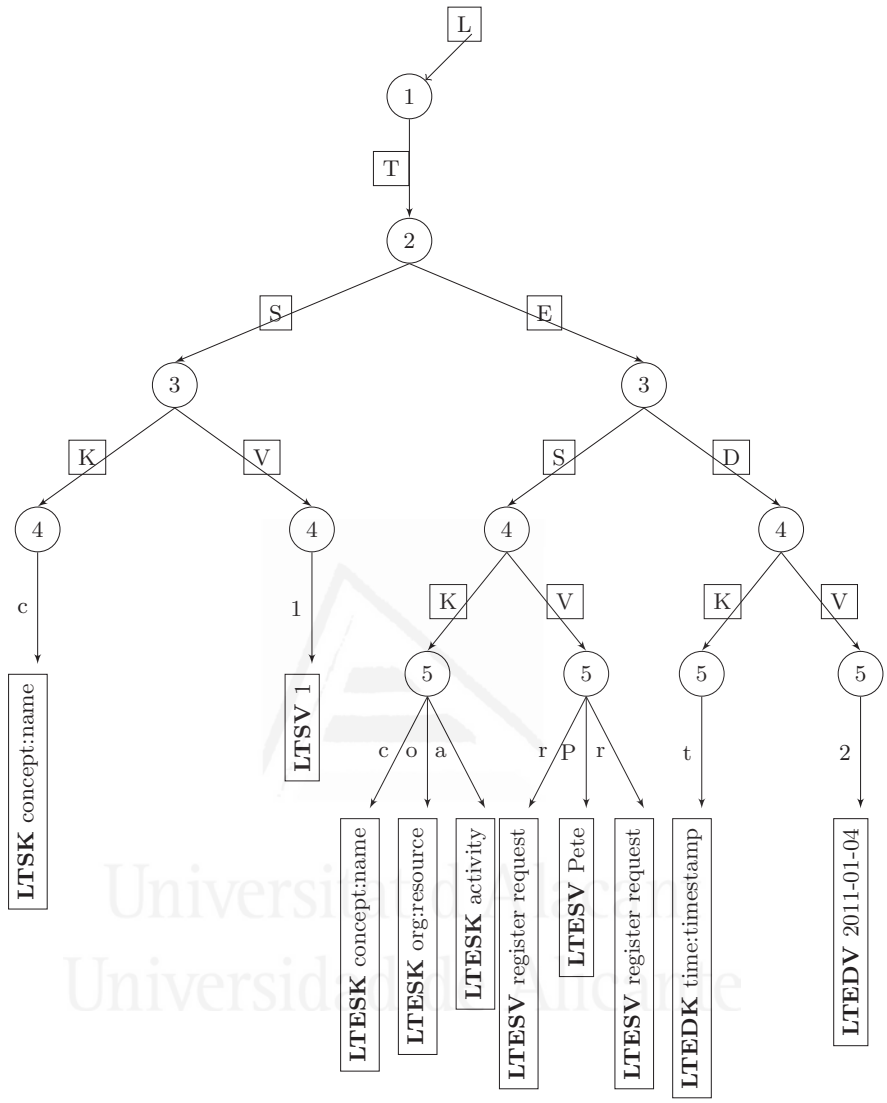


Figura 2.4: Representación de Index Fabric.

en el árbol. Para realizar una búsqueda en el mismo, se comienza desde el nodo raíz y se visita cada arista donde el caracter coincida con la clave de búsqueda. Cada nodo recorrido es codificado utilizando un designador

Etiqueta	Designador
log	L
trace	T
string	S
event	E
@key	K
@value	V
date	D

Tabla 2.1: Diccionario de designadores.

único y almacenado en un diccionario (véase un diccionario de designadores en la tabla 2.1).

Por ejemplo, según la figura 1.1 el nodo *log* se codifica como L, el nodo *trace* como T, el nodo evento como E y así sucesivamente. Por lo tanto, el camino *log trace event string @key* es codificado como $L\ T\ E\ S\ K$ y se almacena en la tabla 2.2 con su contenido textual correspondiente (*"concept:name"*). Una limitante de esta propuesta es su rápido crecimiento en espacio.

Los mismos autores utilizaron más tarde las estructuras de datos *Patricia Tries* (del acrónimo en inglés *Practical algorithm to retrieve information codec in alphanumeric*) y el balanceo de las mismas mediante la creación de múltiples capas. No obstante, mantienen algunas deficiencias con respecto al espacio del índice y al balanceo del árbol. Por otra parte, *IndexFabric* no almacena la información de los elementos XML que no contienen datos, por lo tanto, la estructura de índice es ineficiente para consultas parciales o también para las llamadas subconsultas, es decir, que no comienzan desde la raíz del documento XML.

Para enfrentar las limitaciones del coste espacial de las técnicas antes mencionadas, Chung et al. (2002) diseñaron APEX (*Adaptative Path indEX* en inglés). Mientras los índices antes discutidos mantienen los caminos desde el nodo raíz, esta estructura de índice utiliza los caminos más frecuentes mediante la aplicación de técnicas de minería de secuencias de patrones.

APEX contiene dos estructuras principales. Un grafo G_{APEX} que re-

Camino
L T S K concept:name
L T S V 1
L T E S K concept:name
L T E S V register request
L T E S K org:resource
L T E S V Pete
L T E D K time:timestamp
L T E D V 2011-01-04

Tabla 2.2: Camino desde el nodo raíz hasta los nodos hojas.

presenta un resumen estructural del documento XML y una tabla hash (H_{APEX}) que representa las etiquetas entrantes de los caminos a los respectivos nodos. Cada etiqueta G_{APEX} se almacena como un nodo en H_{APEX} y es conocido como *hnodo*. A su vez, cada *hnodo* referencia a un nodo de G_{APEX} o a una tabla hash (véase la figura 2.5).

Además, cada nodo en G_{APEX}, conocido como *xnodo*, coincide con la entrada de un *hnodo* de H_{APEX}. Por último, cada extensión contiene un conjunto de aristas del grafo y es asignada a cada *xnodo*. APEX a pesar de que no presenta problemas con la inserción y actualización de nodos y muestra buenas prestaciones para consultas simples, no puede responder directamente a consultas con expresiones de caminos mayor que uno y no permite la ejecución de consultas ramificadas.

Universidad de Alicante

Todas las técnicas discutidas anteriormente, son consideradas en la literatura como técnicas clásicas de indexación de caminos, no obstante se han propuesto otras que a continuación se describen. Por ejemplo, MIS (Lian et al., 2005), indexa las estructuras poco frecuentes de una colección de documentos XML. Además de los caminos indexados, almacena otras estructuras de alta selectividad, que mantienen la eficiencia del espacio del índice. MIS a pesar de utilizar caminos poco frecuentes, tiene la facilidad de podar gran cantidad de nodos rápidamente. Los mismos autores propusieron adicionarle técnicas de minería de datos para mejorar la eficiencia del procesamiento de las consultas (Lian et al., 2007).

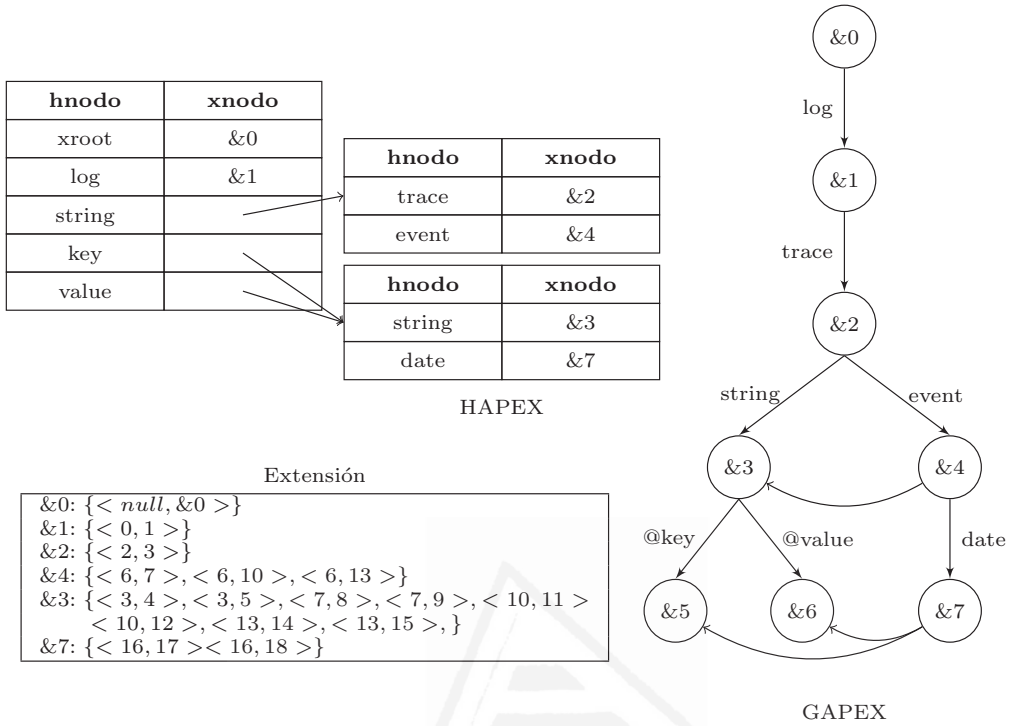


Figura 2.5: El índice APEX.

Por otra parte, MXI (Yan and Liang, 2005) permite la ejecución de consultas de caminos basada en su DTD. El índice es construido a partir de los documentos XML y de su correspondiente DTD, a través de un sistema de codificación propio. MXI para la ejecución de las consultas realiza el apareamiento contra el árbol de su DTD y utiliza el mismo método que el sistema XISS (Harding et al., 2003), donde la consulta es descompuesta en múltiples caminos simples.

En otra investigación Grimsmo (2008) propuso dos técnicas de indexación residentes en memoria. La primera, combina listas invertidas, expansión selectiva, expansión de aciertos y búsqueda por fuerza bruta. La segunda, utiliza los árboles de sufijos con estadísticas adicionales y múltiples puntos de entradas a las consultas. Estos métodos tienen como aspecto

positivo que pueden ser almacenados en disco cuando no sea posible que residan en memoria.

LTIX (Mohammad and Martin, 2010b) por su parte, combina características de un *DataGuide* con la información de los niveles de los nodos de un documento XML. Esta información de los niveles es esencial para la ejecución de las consultas, ya que permite determinar la jerarquía entre los nodos, así como la eliminación de los nodos que violan los diferentes niveles dentro de una consulta.

2.1.2 Técnicas de indexación basadas en nodos

Un buen esquema de etiquetado debe de ser conciso en términos de tamaño, eficiente en el etiquetado y en el tiempo de ejecución de las consultas, asegurando la persistencia de las etiquetas únicas y el dinamismo en la actualización de los nodos sin necesidad de reetiquetar el árbol del XML. Además, debe de identificar directamente los tipos de relaciones estructurales. En términos generales, los esquemas de etiquetado que generan etiquetas de pequeño tamaño, o bien no proporcionan información suficiente para identificar todas las relaciones estructurales entre los nodos o no son dinámicos. Por otro lado, los sistemas de etiquetado dinámicos necesitan más capacidad de almacenamiento que se traduce en una disminución del rendimiento durante el procesamiento de una consulta (Assefa, 2012).

Universidad de Alicante

Las técnicas de indexación basadas en nodos emplean esquemas de etiquetado que utilizan las posiciones de los nodos dependiendo del recorrido del árbol del XML. Se consideran un eslabón fundamental para los algoritmos de uniones estructurales y para los índices de resúmenes estructurales. La principal debilidad que presentan estas técnicas es la cantidad de uniones estructurales que son necesarias establecer para procesar consultas complejas y ramificadas. Estas técnicas también son conocidas en la literatura como: esquema de etiquetado, esquema de codificación o esquema de numeración. A diferencia de las técnicas de indexación de caminos estas pueden determinar eficientemente relaciones jerárquicas entre diferentes nodos.

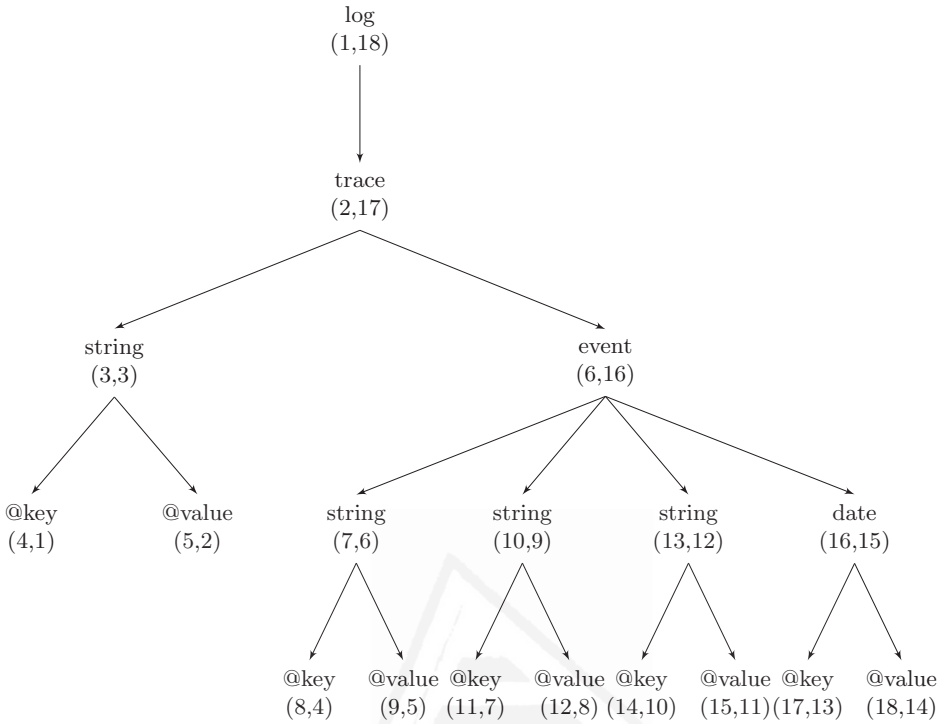


Figura 2.6: Esquema de etiquetado basado en el orden del recorrido.

Universitat d'Alacant
Universidad de Alicante

El primer esquema de etiquetado que se conoce en la literatura es el recorrido basado en el orden (*Traversal Order Based*, Dietz, 1982). Este esquema utiliza números enteros cuyos valores coinciden con el recorrido en preorden y postorden del árbol, donde cada etiqueta de los nodos presenta la forma <*pre,post*> (véase la figura 2.6). Por ejemplo, un nodo Y es descendiente del nodo X si $Y.pre > X.pre$ y $Y.post < X.post$. Este esquema presenta dos desventajas. Primero, las etiquetas no guardan suficiente información para determinar correctamente las relaciones padre-hijo y los órdenes entre nodos hermanos. Segundo, el esquema no es eficiente para los XML dinámicos, es decir, para insertar un nodo en el árbol se necesita un reetiquetado completo del mismo.

Para enfrentar la limitante anterior, Li et al. (2001) propusieron un nue-

vo esquema nombrado *Extended Preorder Traversal* (en español recorrido extendido en preorden). Este esquema utiliza un etiquetado extendido en preorden para garantizar las futuras inserciones en el árbol. Cada etiqueta de los nodos presenta la forma <*pre,tamaño*>, donde *pre* es el recorrido en preorden del árbol y *tamaño* es un número entero arbitrario mayor que el número de descendientes que tiene el nodo (véase la figura 2.7). Sin embargo, la asignación del tamaño no es sencilla, en ocasiones reservándose un tamaño lo suficientemente grande, puede ser alcanzado por las inserciones de nodos. En este esquema las relaciones de descendencia se pueden identificar de la siguiente forma:

$$pre(x) < pre(y) \leq pre(x) + tamaño(x) \Rightarrow ancestro(x, y) \tag{2.1}$$

Dewey ID, implementado por Tatarinov et al. (2002), es un esquema de etiquetado basado en prefijos y fue adaptado del sistema de clasificación decimal Dewey para la organización de colecciones bibliotecarias. En este esquema al identificador posicional del *n-ésimo* hijo le es asignado el valor entero *n* y es concatenado con la etiqueta del padre con el separador ".". Un aspecto positivo de este esquema es que las relaciones estructurales entre los nodos se pueden obtener con solo inspeccionar las etiquetas. Además se pueden actualizar e insertar en el árbol en los nodos más a la derecha sin tener que realizar una nueva codificación. En caso de que se inserte un nodo a la izquierda o en el centro, es necesario realizar una nueva codificación de las etiquetas del árbol.

Por otra parte, O'Neil et al. (2004) diseñaron un esquema de etiquetado también basado en prefijos llamado ORDPATH, con características similares conceptualmente a *Dewey ID*. Este esquema codifica las relaciones padre-hijo extendiendo la etiqueta del padre con la del hijo. Presenta algunas ventajas con respecto a *Dewey ID*. Por ejemplo, el uso de los números impares permite la inserción futura de nodos. Este esquema de etiquetado permite la utilización de enteros negativos para la codificación de los nodos. A pesar de esto, todos los espacios disponibles se pueden agotar si la cantidad de nodos a insertar es considerable, por lo que en ocasiones es necesario un reetiquetado completo del árbol.

Universidad de Alicante

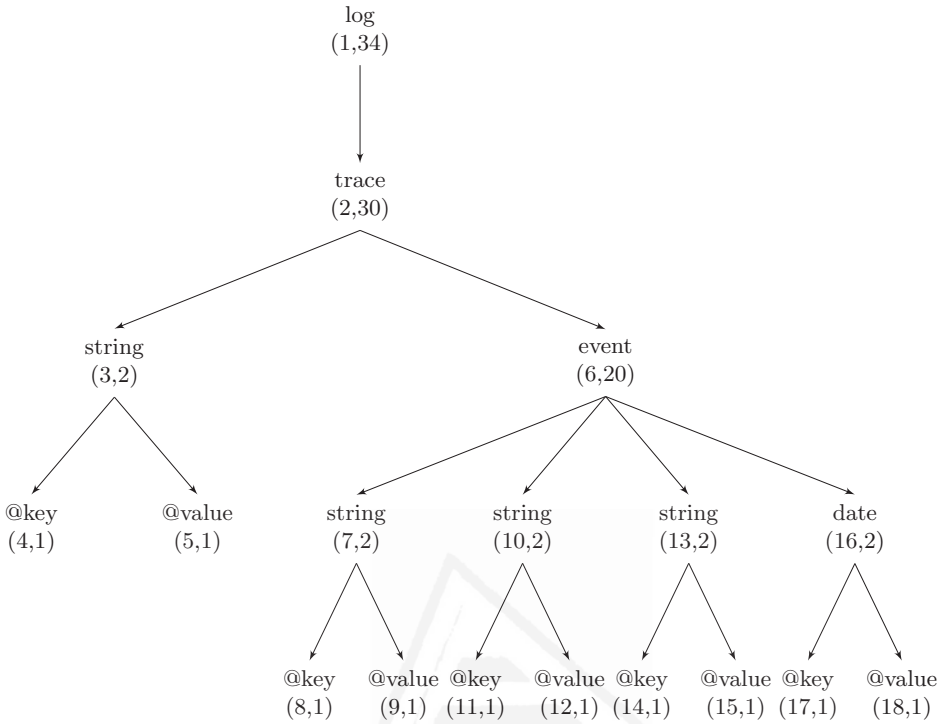


Figura 2.7: Esquema de etiquetado basado en el recorrido extendido en preorden.

Universitat d'Alacant
Universidad de Alicante

Unique Identifier (Lee et al., 1996) es un esquema de etiquetado basado en multiplicaciones. Dicho esquema enumera los nodos y asigna a cada uno un valor K de nodos virtuales, para lograr un balanceo total del árbol. Además, cada nodo es identificado con un valor entero que comienza en *"1"* mediante el recorrido de arriba-abajo y de izquierda-derecha. Este esquema presenta la siguiente propiedad:

$$Parent(i) = \frac{i-2}{K} + 1 \tag{2.2}$$

El valor de K debe ser calculado previo a la construcción del esquema, por lo que es necesario ser lo más preciso posible. Un valor de K demasiado

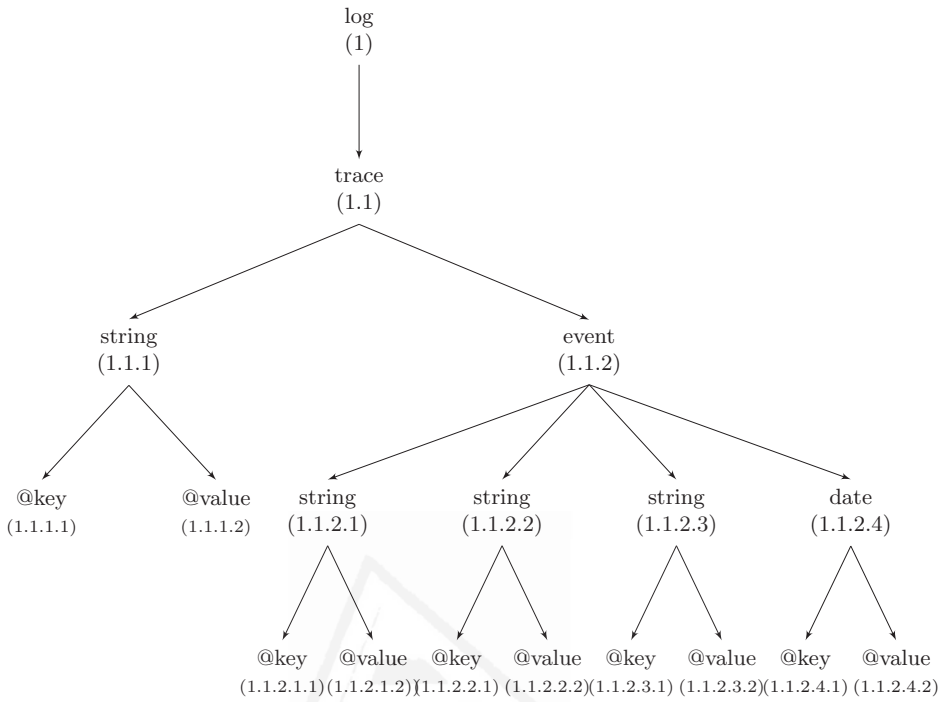


Figura 2.8: Esquema de etiquetado de ORDPATH.

Universitat d'Alacant
Universidad de Alicante

grande produce un aumento considerable de los nodos virtuales y por lo tanto un excesivo espacio en memoria para el almacenamiento del índice. Por otra parte, un valor muy pequeño de K, conlleva a un agotamiento temprano de los nodos virtuales y el reetiquetado completo del árbol.

Prime Number (Wu et al., 2004), por su parte, es un esquema multiplicativo que utiliza como etiquetado de cada nodo a los números primos. Existen dos variantes: una de abajo-arriba y la otra de arriba-abajo. La primera, tiene la particularidad de que cada nodo se construye a partir de la multiplicación de los valores primos de cada hijo. Las relaciones padre-hijo se pueden calcular por la división de ambos nodos y las relaciones ancestro-descendiente por el cálculo del módulo de la división de ambos nodos. Este esquema presenta la limitante de que los nodos cercanos a la raíz del árbol

se codifican con números primos muy grandes.

En la variante de arriba-abajo a cada nodo se le asigna la multiplicación del valor del padre con la del propio nodo, que es también un valor primo único. Las relaciones padre-hijo y ancestro-descendiente pueden ser calculadas como en la variante anterior. Este esquema permite la actualización dinámica de documentos XML.

Otro esquema de etiquetado multiplicativo es BIRD (Weigel et al., 2005). Se basa en el etiquetado de números enteros para responder a las consultas estructurales y presenta, entre otras estructuras de datos, un índice *DataGuide* que almacena el resumen estructural llamado *Ind(DB)*. Siempre en un índice *BIRD*, N es la cantidad de nodos del documento fuente y M es la cantidad de nodos de *Ind(DB)*. Para cada nodo $m \in M$ en *Ind(DB)* se le calculan dos pesos. Un peso balanceado de los nodos hijos y un pre-peso del nodo. Para todo $n \in N$, n es ancestro de n' si:

$$ID(n) < ID(n') < ID(n) + peso(n) \tag{2.3}$$

Se puede calcular el ID del padre de cualquier nodo n, donde b es el peso del nodo padre, con la siguiente fórmula:

$$ID(n) - (ID(n) \bmod b) \tag{2.4}$$

Universitat d'Alacant
Universidad de Alicante

Este esquema de etiquetado es muy eficiente para determinar las relaciones padre-hijo y ancestro-descendiente. Soporta inserciones/actualizaciones hasta que no se sobrepase el peso asignado a cada nodo. La principal limitante de este esquema es que la construcción de etiquetas tiene un alto coste computacional y cuando el número de inserciones/actualizaciones sobrepasa el peso asignado al nodo, se necesita una completa redistribución de los IDs.

Por otra parte, Lu et al. (2011) rediseñaron el esquema de codificación *Dewey ID* y lo nombraron *Dewey ID extendido*. Como característica importante de este esquema tenemos que con solo conocer la etiqueta de un nodo podemos saber cuales son los nodos que le anteceden hasta la raíz

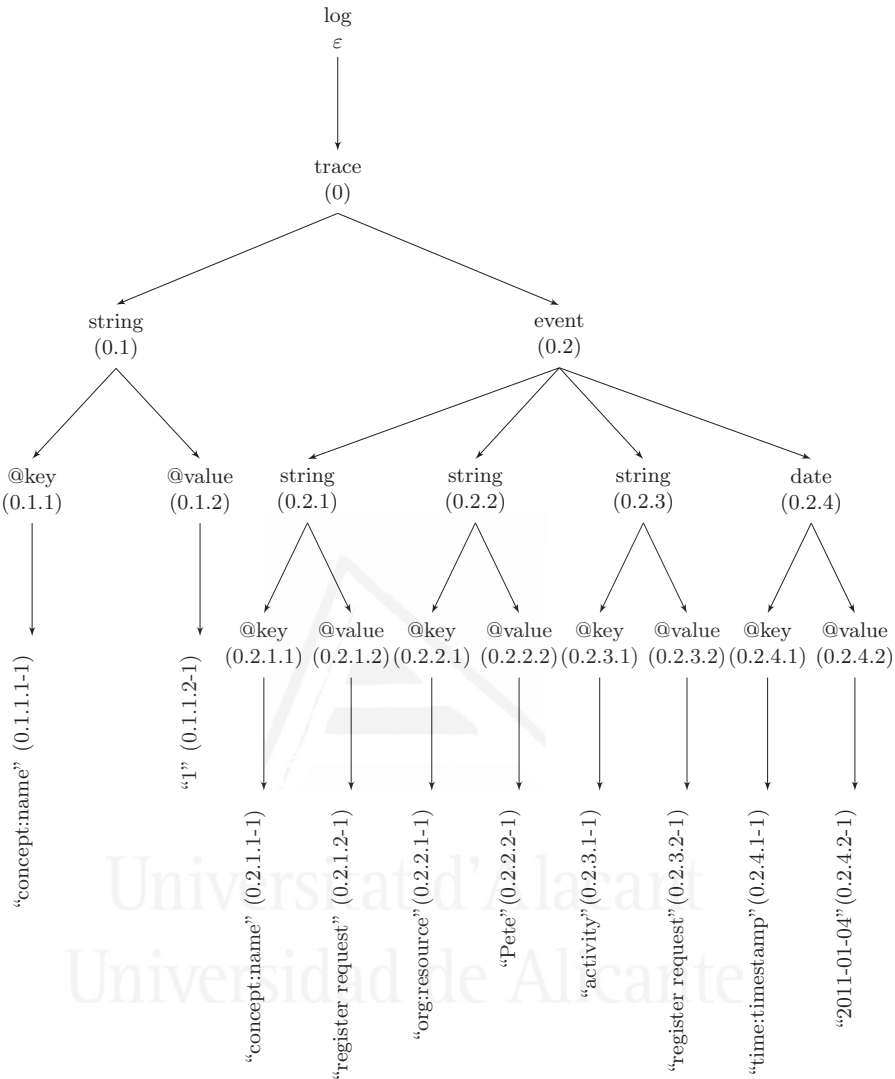


Figura 2.9: Esquema de etiquetado de Dewey ID extendido.

del documento. Un ejemplo de este esquema de etiquetado se muestra en la figura 2.9.

Finalmente, el esquema ORDERBASED (Assefa, 2012), permite un eti-

quetado dinámico que se basa en la combinación de letras y números. Cada etiqueta presenta la forma <*nivel,orden,orden-padre*>, donde *nivel* representa la distancia desde el nodo raíz, *orden* se corresponde con el caracter que representa la distancia horizontal desde el nodo más a la izquierda en el mismo *nivel* y el *orden-padre* es el *orden* del nodo padre. Lograr mantener la misma etiqueta del nodo en caso de nuevas inserciones o actualizaciones es una de las principales fortalezas de este esquema.

2.1.3 Técnicas de indexación basadas en secuencias

Las técnicas de indexación basadas en secuencias tienen como objetivo principal disminuir el tiempo de ejecución de los algoritmos de uniones estructurales. Estas técnicas representan tanto a los documentos como a las consultas por secuencias y subsecuencias, de modo que una consulta puede ser contestada mediante el cotejo de la secuencia del documento y de la consulta.

Por ejemplo, en VIST (*Virtual Suffix Tree*, Wang et al., 2003) tanto el documento XML como las consultas son transformadas en secuencias mediante el recorrido en preorden del documento XML. Para ejecutar una consulta se cotejan las coincidencias entre ambas secuencias. Cada secuencia consiste en una lista de pares (*símbolo,prefijo*), es decir, $(a_1, p_1)(a_2, p_2) \ldots (a_n, p_n)$, donde a_n representa al nodo del documento XML y p_n representa el camino desde el nodo raíz hasta el nodo a_n.

Una limitación presente en VIST es que el índice alcanza tamaño cuadrático en el peor de los casos si el documento XML es muy profundo. Además, en ocasiones pueden aparecer falsas alarmas y falsos despidos en la respuesta a una consulta. Por ejemplo, según la tabla 2.3 que representa las secuencias tanto para el documento como para las consultas; las consultas 1 y 2 representan un caso de falsa alarma y las consultas 1 y 3 falso despido.

Para dar solución a esta deficiencia fue diseñado PRIX (Rao and Moon, 2004), que se basa en la utilización de secuencias Prüfer. PRIX transforma cada documento XML en una secuencia de etiquetas utilizando el método Prüfer, que consiste en una relación de uno a uno entre la secuencia y el documento XML. Básicamente, los elementos del nivel superior del XML

son compartidos con los elementos del nivel más bajo por ser sus padres o nodos ancestrales. La codificación se realiza borrando repetidamente el nodo hoja que tiene la etiqueta más pequeña y se agrega la etiqueta de su padre a la secuencia. PRIX se basa en un árbol B+ y es construido de una manera similar a VIST. La transformación de abajo-arriba juega el rol principal en la reducción del tiempo de ejecución de las consultas. PRIX a pesar de que elimina las falsas alarmas utilizando series complejas de refinamiento, produce en algunos casos falsos despidos, como se aprecia en la tabla 2.3, las consultas 1 y 3.

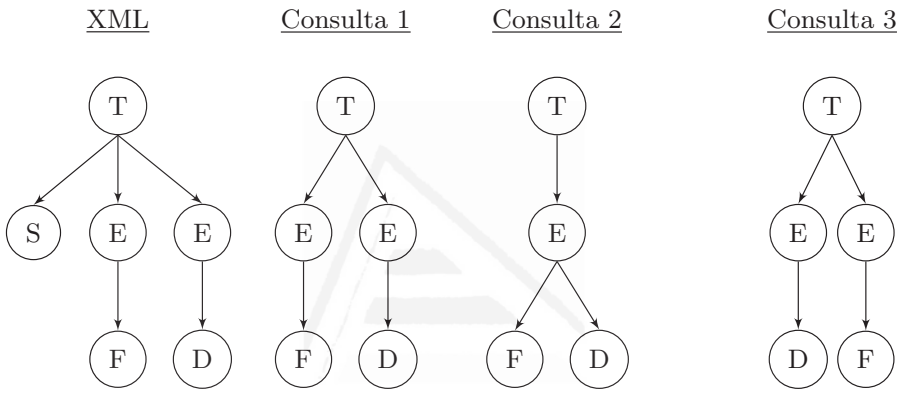


Figura 2.10: Documento XML y ejemplos de consultas.

Universitat d'Alacant
Universidad de Alicante

Métodos	**Origen**	**Secuencias**
VIST	XML	(T,null)(S,T)(E,T)(E,T)(F,E,T)(D,E,T)
	Consulta 1	(T,null)(E,T)(E,T)(F,E,T)(D,E,T)
	Consulta 2	(T,null)(E,T)(F,E,T)(D,E,T)
	Consulta 3	(T,null)(E,T)(E,T)(D,E,T)(F,E,T)
PRIX	XML	STFETDET
	Consulta 1	FETDET
	Consulta 2	FEDET
	Consulta 3	DETFET

Tabla 2.3: Falsas alarmas y falsos despidos.

Para enfrentar el reto de lograr la disminución del tamaño del índice, se definió un esquema de indexación comprimido basado en las transformaciones Burrows-Wheeler (Ferragina and Manzini, 2005). Esta técnica utiliza la ordenación de caminos y la agrupación de los nodos del árbol y los almacena en dos arreglos: uno para la estructura y el otro para las etiquetas.

En este esquema de indexación, dado un árbol T se construye una estructura S, donde cada elemento es una tripleta que se obtiene a partir del recorrido del árbol T en preorden. Para cada nodo u la tripleta es $s[u] = (last[u], \alpha[u], \pi[u])$, donde $last[u]$ es un valor binario igual a "1"si es el último nodo hijo de su padre, $\alpha[u]$ es la etiqueta del nodo y $\pi[u]$ es la cadena que concatena los nodos recorridos desde el nodo padre hasta la raíz. Las operaciones principales que se pueden ejecutar sobre estas secuencias arbitrarias son *rank* y *select*. Este esquema de indexación comprimido tiene algunas deficiencias. En primer lugar, no soporta la indexación de XML dinámicos; en segundo lugar, la construcción de las etiquetas presenta un alto coste computacional.

Por otra parte, investigaciones realizadas por Wang and Meng (2005) propusieron más tarde una técnica basada en restricciones de secuencias para evitar la falsa alarma causada por nodos hermanos idénticos como sucedía en VIST. Esta técnica a pesar de dar solución al problema de la falsa alarma, provoca un efecto negativo en el rendimiento del proceso de consultas. Este pobre rendimiento, unido a la falsa alarma provocada por VIST y las deficiencias de PRIX, motivó a la implementación de un nuevo esquema de indexación basado en secuencias llamado RP (*Region Path*, Li et al., 2013). Este nuevo esquema está compuesto por la unión de un esquema de secuencias basado en caminos y un esquema de codificación basado en regiones.

2.1.4 Técnicas de indexación híbridas

Las técnicas de indexación híbridas presentan características de las diferentes técnicas descritas anteriormente o de otras que se referencian en la literatura. Estas técnicas por lo general, incluyen un índice estructural y un índice de valores o de contenidos. Por ejemplo, NCIM (*Node Clustering*

Indexing Method, Liao et al., 2010) realiza un etiquetado de cada nodo del árbol del XML con la tripleta $(nivel, n^{\vdash}, n^{\dashv})$ para el caso de los nodos internos y la tupla $(nivel, n^{\vdash})$ para los nodos hojas. Para ambos casos el *nivel* es la profundidad del nodo n, $n^{\vdash}$ es el valor inicial a través del recorrido primero en profundidad del árbol del XML y $n^{\dashv}$ es el valor final después de realizar el recorrido completo por todos los nodos.

El empleo de este esquema de etiquetado permite determinar diferentes relaciones estructurales. Por ejemplo, dado dos nodos (x, y) si $x^{\vdash} \in (y^{\vdash}, y^{\dashv}]$ se sabe que x es descendiente de y. Si $nivel_x - nivel_y = 1$, entonces existe una relación padre-hijo (x es padre de y). Por otro lado, si $nivel_x - nivel_y > 1$, entonces existe una relación ancestro-descendiente (x es ancestro de y).

Etiq.,Nivel	**Inicio**	**Fin**				
log,1	1	26				
trace,2	2	25				
string,3	3	6				
event,3	7	24				
date,4	20	23				
string,4	8	11	12	15	16	19

Figura 2.11: Índice de los nodos internos de NCIM.

Universitat d'Alacant
Universidad de Alicante

Etiq.,Nivel	**Inicio**	**Fin**						
@key,1	4	"concept:name"						
@value,4	5	"1"						
@key,5	9	"concept:name"	9	"org:resource"	9	"activity"	9	"time:stamp"
@value,5	10	"register ..."	10	"Pete"	10	"register ..."	10	"2011-01-04"

Figura 2.12: Índice de los nodos hojas de NCIM.

Toda la información generada durante la fase de etiquetado de este método de indexación es almacenada en cuatro tablas hash, dos para el índice de nodos (véanse las figuras 2.11 y 2.12) y dos para el índice de niveles (véanse las figuras 2.13 y 2.14). Esta estructura de datos permite un

rápido acceso a la información almacenada, especialmente para determinar las relaciones ancestro-descendiente. El alto coste espacial de este índice es una de sus principales limitantes.

Para reducir el coste espacial de NCIM, Hsu et al. (2012b) desarrollaron PCIM (*Path Clustering Indexing Method* en inglés), basado en la indexación de un resumen estructural. PCIM es almacenado en dos tablas hash, una que almacena la parte estructural del árbol y la otra almacena el contenido textual. Sus resultados experimentales demuestran la eficiencia en el tiempo de ejecución de las consultas y en el tamaño ocupado por el índice. El alto coste temporal durante la construcción del índice es una de sus principales limitantes.

Ambas técnicas PCIM y NCIM utilizan el esquema de etiquetado basado en regiones. PCIM utiliza cadenas para representar las etiquetas y NCIM utiliza números enteros donde sea posible. Según Hsu and Liao (2013), en la mayoría de los casos NCIM supera a PCIM por dos razones fundamentales. Primero, PCIM almacena el contenido textual en otras tablas aparte de la estructura y NCIM lo realiza en el índice de los nodos hojas dentro de su etiqueta correspondiente, lo que significa una reducción en el tiempo de ejecución de las consultas con contenido y estructura. Segundo, las comparaciones de valores enteros siempre son más rápidas que la comparación de cadenas.

Universitat d'Alacant
Universidad de Alicante

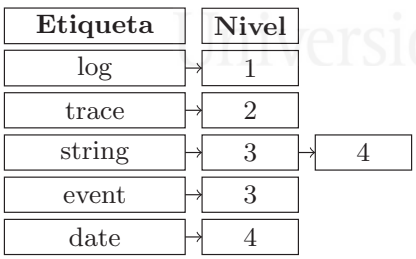


Figura 2.13: Índice de niveles de los nodos internos de NCIM.

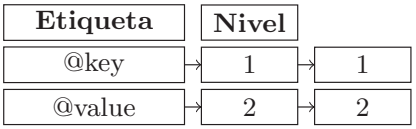


Figura 2.14: Índice de niveles de los nodos hojas de NCIM.

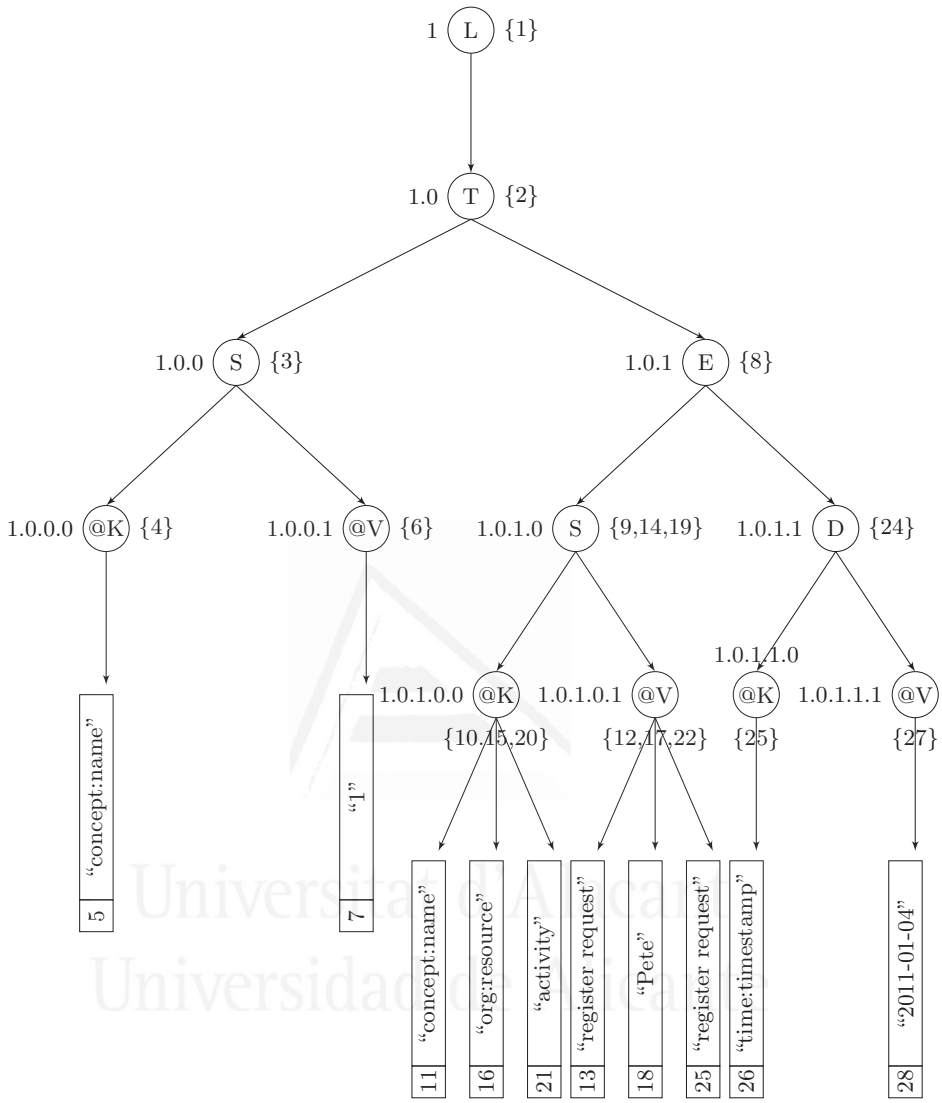


Figura 2.15: Resumen estructural y etiquetado de CIS-X.

Con el objetivo de superar algunas de las deficiencias de estas dos técnicas Hsu and Liao (2013) diseñaron CIS-X (*Compact Index Scheme for XML* en inglés), que extiende de los resúmenes estructurales de caminos (*Data-*

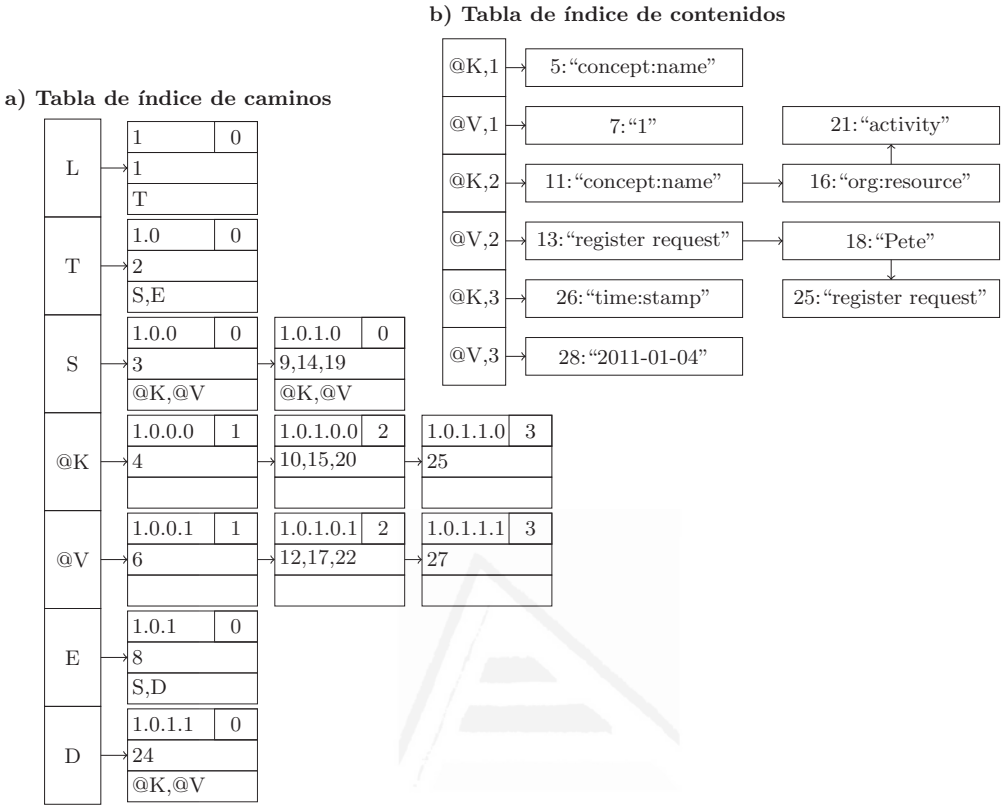


Figura 2.16: Arquitectura de CIS-X.

Universitat d'Alacant
Universidad de Alicante

Guide). El proceso de etiquetado se realiza mediante el recorrido en preorden y unido a los resúmenes estructurales aplicados se asemeja en gran medida al esquema de etiquetado basado en regiones (véase su representación gráfica en la figura 2.15). La diferencia más importante entre el resumen estructural de la figura 2.15 y un índice *DataGuide*, es que la primera incluye a los nodos hojas o nodos textuales.

Este resumen estructural no es eficiente para las búsquedas parciales. Su información se almacena en dos tablas hash, una para el índice de contenido y otra para el índice de caminos (véase la figura 2.16). Utiliza además el esquema de etiquetado *Dewey ID* para representar las posiciones de los

nodos dentro del árbol del XML y determinar las relaciones padre-hijo y ancestro-descendiente. Para la ejecución de las consultas utilizan el algoritmo *Twig-List* (Qin et al., 2007), que fue modificado para las consultas con contenido y estructura. Utilizan también una pila temporal que almacena los caminos activos recorridos, aspecto que incide en el tiempo de construcción del índice. CIS-X claramente retiene toda la información en esta estructura compacta, que ocupa menos espacio que el documento original y es capaz de construir un documento XML como respuesta si es conocido el nodo raíz.

Una limitación importante que presenta este esquema compacto de indexación es que el índice debe de ser reconstruido completamente cuando se actualizan los elementos del documento XML. Por otra parte, cuando se manipulan documentos XML de gran dimensión, el tamaño del índice sobrepasa la memoria principal del ordenador. Según Liu et al. (2013) una solución podría ser el análisis en paralelo de los XML o el almacenamiento del índice en memorias secundarias.

2.2 Técnicas paralelas/distribuidas

Hasta aquí se han discutido disímiles técnicas de indexación para acelerar el proceso de construcción del índice y de ejecución de consultas sobre documentos XML. Todas estas técnicas han sido diseñadas para ejecutarse en un ordenador de forma secuencial, donde la memoria es proporcional al tamaño del documento. Pero en los casos que esta memoria sobrepase la memoria principal, se necesita que las técnicas sean rediseñadas o que se implementen nuevas.

Si el trabajo de un gran número de programas computacionales pudiera ser dividido en pequeños programas y procesado en paralelo en miles de ordenadores distribuidos, el problema del tratamiento de documentos XML de gran dimensión pudiera ser resuelto. El paradigma de programación MapReduce es una solución eficiente. Esta área de investigación es relativamente joven en comparación con las técnicas secuenciales de indexación de documentos XML. No obstante en la presente sección describimos las

principales contribuciones de los últimos años que utilizan este paradigma de programación, implementados sobre la plataforma Hadoop.

Las técnicas que a continuación se describen se centran en tres casos fundamentales. Por una parte, en la paralelización/distribución del proceso de construcción del índice. Otros en la paralelización/distribución del proceso de ejecución de consultas. Por último, en la paralelización/distribución del proceso de construcción del índice y de ejecución de las consultas.

Por ejemplo, Li and Tao (2012) implementaron una técnica de indexación que utiliza el algoritmo SLCA (*Smallest Lowest Common Ancestor* en inglés) y la búsqueda por palabras claves en un documento XML. La arquitectura de búsqueda planteada utiliza dos procesos MapReduce. Uno para la construcción de las tablas hash con la información del recorrido en preorden de todas las palabras claves del documento con su identificador único. El otro para el proceso de ejecución de las consultas y para el cálculo del SLCA, que permite determinar todos los ancestros de un nodo. Un aspecto positivo de esta investigación consiste en el diseño de un algoritmo que reconstruye la estructura del documento XML, que se destruye durante la etapa de preprocesamiento del mismo.

Una técnica similar que utiliza el algoritmo SLCA para la búsqueda por palabras claves la desarrollaron Zhou et al. (2012). Entre sus contribuciones principales se encuentran: el diseño de un algoritmo para particionar documentos XML de gran dimensión sin afectar la estructura definida en el *XML Schema*; también el rediseño del algoritmo SLCA para la plataforma Hadoop y la creación de índices invertidos sobre la información estructural y el contenido de cada nodo para la aceleración del proceso de consultas.

Por otra parte, Choi et al. (2012) implementaron técnicas para el procesamiento de algoritmos de uniones estructurales sobre la plataforma Hadoop y su paradigma de programación MapReduce. El proceso MapReduce propuesto cuenta de dos fases. En la primera fase, el documento XML es dividido en trozos y es escaneado contra las consultas de entrada. A continuación los caminos son enviados a la segunda fase MapReduce para fusionar la respuesta final. Una de las debilidades es la pérdida de la forma integral para la generación de las soluciones de caminos. Otra debilidad de la propuesta es que no es posible resolver consultas estructurales con

relaciones indirectas para documentos XML con nodos recursivos.

Un trabajo similar lo desarrolló Wu (2014), que se centró en la paralelización de las técnicas de uniones estructurales para el procesamiento de consultas en documentos XML. En vez de distribuir los grandes documentos XML hacia un clúster de computadoras, lo que realiza es la distribución de las listas invertidas generadas a partir de la información de los documentos originales. Hasta lo que conocemos esta técnica es la primera en discutir la paralelización/distribución de listas invertidas sobre la plataforma Hadoop.

Estas listas invertidas ocupan menos espacio que los documentos originales y por supuesto inciden en el tiempo de ejecución de las consultas y en la transferencia de los datos hacia el sistema de archivo HDFS. La idea básica de esta técnica es dividir el espacio computacional del conjunto de uniones estructurales en varios subespacios. La función *map* toma una serie de etiquetas de las listas invertidas como entrada y emite un *ID* por cada etiqueta con el subespacio asociado (*e-id*). Luego la función *reduce* toma las etiquetas agrupadas de cada lista invertida, las reordena y le aplica los algoritmos de uniones estructurales para encontrar las respuestas a las consultas. Una limitante es el alto coste temporal durante la construcción de las listas invertidas.

En trabajos más recientes, Chen et al. (2015) propusieron un mecanismo eficiente para el procesamiento de grandes documentos XML utilizando igualmente la plataforma Hadoop. Los autores logran sacar ventaja de las bondades del paradigma de programación MapReduce y de la caché de las técnicas distribuidas. Se basan en el principio de que cualquier documento XML está formado por relaciones padre-hijo y que en las operaciones MapReduce estas relaciones pueden ser dañadas en la etapa de división de los documentos. Por lo tanto, se plantearon un algoritmo que reconstruye estas relaciones, con el objetivo de extraer todos los caminos desde el nodo raíz y almacenarlos en la plataforma HBase[1].

Otras aportaciones importantes se basan en la adaptación de las técnicas de indexación secuenciales a paralelas/distribuidas. Por ejemplo, Hsu et al. (2012a) modificaron el índice NCIM y diseñaron una nueva arquitectura de

[1] http://hbase.apache.org

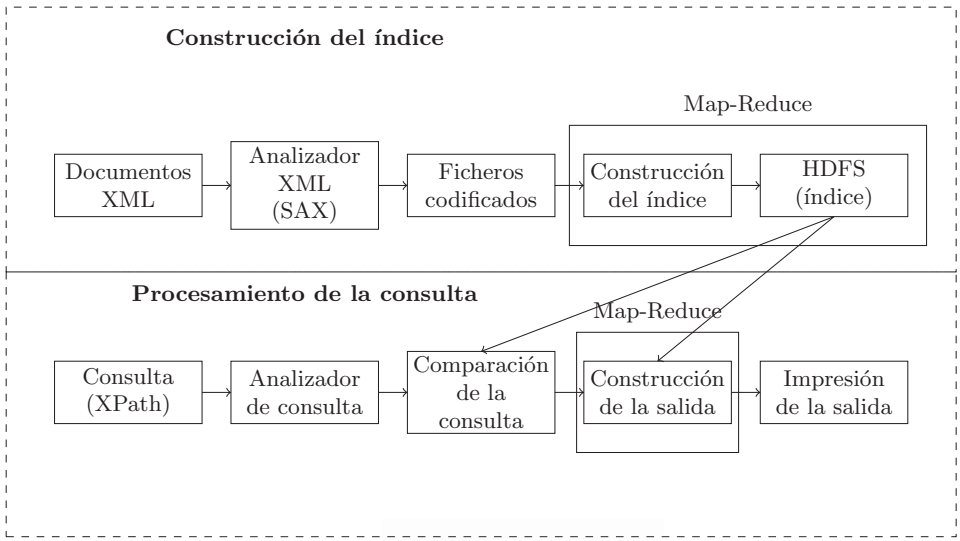


Figura 2.17: Arquitectura MapReduce de CIS-X.

indexación y consulta para el manejo de un gran número de documentos XML utilizando la plataforma Hadoop. A diferencia que en NCIM, que se mantenía el índice completo en memoria a través de cuatro tablas hash, en esta nueva versión se almacena el índice en el sistema de archivo HDFS. La etapa de análisis de los ficheros y de construcción del índice reside en la función *map*, la cual produce una lista de pares $clave{-}valor$, donde cada *clave* a su vez es un par $<etiqueta, nivel>$ y el *valor* es la etiqueta de cada nodo. Luego se agrupan todos los pares que presenten la misma *clave*. La función *reduce* es aplicada en paralelo a cada nodo y crean grupos con la misma *clave*. El resultado final es almacenado en varios ficheros en el sistema de archivo HDFS.

Igualmente, Hsu et al. (2014) rediseñaron el índice CIS-X sobre la plataforma Hadoop para manipular grandes volúmenes de documentos XML y lograr el procesamiento de consultas eficientes a través de la programación paralela en la nube. Esta técnica de indexación cuenta con dos módulos principales: generación del índice y evaluación de las consultas (véase la figura 2.17).

En el primer módulo los autores consideraron como muy necesario la división de los documentos XML, cuando estos ocupen más que la memoria principal y que estos sean los parámetros de entrada para el procesamiento MapReduce. Además que las partes del índice generadas por cada fase MapReduce, tienen que estar habilitadas para ser unidas tomando en consideración la política de división escogida.

El segundo módulo cuenta de dos fases. La primera fase consiste en encontrar los nodos objetivos y la segunda en generar la salida final. A diferencia de muchas otras técnicas de indexación, CIS-X para generar la salida final no necesita visitar el documento original nuevamente. Se puede generar a través de los índices almacenados en la plataforma Hadoop mediante las operaciones MapReduce. Esta fase consume en ocasiones gran cantidad de tiempo, debido a que todos los índices de los nodos descendientes tienen que ser visitados y luego construir el árbol correspondiente como respuesta a la consulta.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

3. Propuesta de un índice estructural

> La excelencia escrita se logra cuando se conduce al lector suavemente, de idea en idea.
>
> Sharon Morey

En este capítulo se presenta una propuesta de índice estructural para archivos en formato XES. Se analizan las propiedades y las ventajas del uso de una estructura de este tipo. Se describe además la etapa de análisis sobre un fichero XES y cómo es posible mejorar esta etapa mediante la aplicación de una estrategia de compresión del esquema del XES, basado en la repetición de los atributos *string* en los eventos de cada traza. Se detalla también cómo el índice estructural es posible almacenarlo eficientemente en un arreglo de sufijos y en un árbol ternario de búsqueda para reducir el coste temporal de las consultas.

Universitat d'Alacant
Universidad de Alicante

3.1 Índice estructural XES

El formato XES, es un estándar basado en XML para el almacenamiento e intercambio de los registros de eventos. Un registro a su vez contiene un determinado número de trazas y cada traza un determinado número de eventos. Los registros, trazas y eventos contienen además un conjunto de atributos de tipo: *string*, *date*, *int*, *float*, *boolean* o *ID*. En la figura 3.1 se aprecia un archivo XES, formado por una traza y un evento. La traza contiene un atributo de tipo *string* y el evento está compuesto por 3 atributos de tipo *string* y un atributo de tipo *date*.

```
<log>
  <trace>
    <string key="id" value="1"/>
    <event>
      <string key="id" value="1.1"/>
      <string key="name" value="Pete"/>
      <date key="date" value="2011-01-04"/>
      <string key="activity" value="login user"/>
    </event>
  </trace>
</log>
```

Figura 3.1: Ejemplo de un archivo XES.

En el área de la recuperación de documentos estructurados, Felix Weigel en su tesis doctoral define a un resumen estructural (structural summary, Weigel, 2006) como una estructura compacta sobre un documento XML, donde las propiedades estructurales se pueden inferir sin acceder al documento original. Los resúmenes estructurales pueden ser centralizados o no.

Un resumen estructural típico centralizado se representa como un árbol que contiene información de todas las etiquetas, los niveles de cada nodo con estas etiquetas y la forma en que son unificadas. Un sistema no centralizado incluye un esquema de etiquetado que identifica a cada nodo individualmente y las relaciones de los nodos del árbol usando una serie finita de información.

Universitat d'Alacant
Universidad de Alicante

Definición 1. *Un camino estructural C_n es una secuencia formada por los nodos $x_1, x_2, \ldots, x_k$ y se denota $C_n = x_1, x_2, \ldots, x_k$, donde x_1 es el nodo raíz. Y si $C_n = x_1, x_2, \ldots, x_k, x_{k+1}$ es un subcamino, entonces x_{k+1} es un nodo hoja. En ese caso x_{k+1} se denomina valor textual de C_n.*

Al menos dos tipos de relaciones se pueden establecer entre los nodos de un mismo camino estructural:

- *directas o de padre-hijo* (*parent-child* en inglés), significa que dos nodos x_1 y x_2 son consecutivos en el camino y se denota como x_1/x_2.

- *indirectas o ancestro-descendiente* (*ancestor-descendent* en inglés), significa que el nodo x_1 precede del nodo x_2 y se denota como $x_1//x_2$.

Por ejemplo, un camino sobre el documento XES de la figura 3.1 es *log trace event string @key*. Para este caso, una relación directa es *trace/event* y una relación indirecta es *trace//string*.

Es posible listar de manera ordenada los valores textuales de los nodos hojas del árbol del documento T, haciendo un recorrido por todos los nodos en preorden, postorden o entreorden. De esa forma una palabra w que aparece como valor textual de un determinado nodo hoja, se le asigna una posición j de ocurrencia en la secuencia ordenada. La secuencia ordenada ascendentemente según el orden de aparición de los valores textuales, la denominamos secuencia textual del árbol T.

Por ejemplo, según el documento XES de la figura 3.1, a los nodos hojas *"id"* y *"1"* le corresponde la secuencia textual [1,1] y [2,2]. En ocasiones, como ocurre en el nodo hoja *"login user"*, que contiene más de una palabra, las posiciones de ocurrencias no coinciden. En este caso le corresponde la secuencia textual [10,11].

Definición 2. *Un resumen estructural de caminos es una secuencia ordenada de pares* $<C_i, D_i>$*, donde* C_i *es un camino estructural único de un archivo XES y* D_i *es la secuencia de posiciones de cada uno de los valores textuales del camino estructural* C_i *con respecto a la secuencia textual obtenida para todo* $1 \leq i \leq n$*, donde* n *es el número total de caminos estructurales. De manera abreviada si las palabras asociadas al valor textual aparecen en el orden* $k, k+1, \ldots, k+n$*, en lugar de la secuencia de posiciones se especifica el intervalo [k, k+n].*

Siguiendo las definiciones anteriores se puede identificar en la figura 3.2 un resumen estructural que se crea a partir del documento XES de la figura 3.1. El par $<C_4, D_4>$ corresponde con el camino *log trace event string @value* y su secuencia de posiciones [4,4][6,6][10,11].

Definición 3. *Un índice estructural XES se define como un índice formado por un resumen estructural de caminos para representar de una manera compacta la estructura de un documento XES.*

```
log trace event date @key [7,7]
log trace event date @value [8,8]
log trace event string @key [3,3] [5,5] [9,9]
log trace event string @value [4,4] [6,6] [10,11]
log trace string @key [1,1]
log trace string @value [2,2]
```

Figura 3.2: Resumen estructural de caminos.

Es posible almacenar un índice estructural XES en diferentes estructuras de datos. Seguidamente se describe cómo se almacena el mismo en un arreglo de sufijos S y en un árbol ternario de búsqueda TST –como estructura complementaria para acelerar el procesamiento de las consultas– donde los nodos internos almacenan todos los sufijos de S y los nodos hojas el rango máximo de posiciones de cada sufijo en S. La estructura resultante la llamaremos a partir de aquí SATST (*Suffix Array and Ternary Search Tree* en inglés). Para su construcción, se definen las siguientes etapas:

1. Análisis sintáctico de los archivos XES.
2. Construcción del arreglo de sufijos.
3. Construcción del árbol ternario de búsqueda.
4. Procesamiento de las consultas.

Universitat d'Alacant
Universidad de Alicante

En la figura 3.3 se aprecia el flujo de construcción de un índice estructural XES con sus diferentes etapas. Los archivos XES componen uno de los parámetros de entrada y a través del análisis sintáctico es generado un resumen estructural de caminos. El proceso continúa con la construcción del arreglo de sufijos y luego la construcción del árbol ternario de búsqueda como estructura auxiliar para optimizar el procesamiento de las consultas. Las consultas constituyen otro parámetro de entrada del índice. Son procesadas y ejecutadas sobre el arreglo de sufijos o el árbol ternario de búsqueda indistintamente. El proceso culmina con la impresión de los resultados de la consulta.

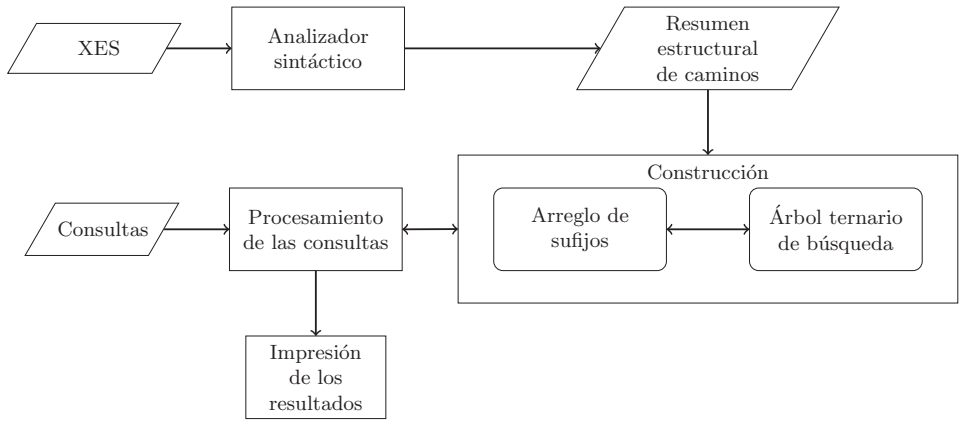


Figura 3.3: Flujo de construcción de un índice estructural XES.

3.2 Analizador sintáctico de archivos XES

La presente etapa tiene como objetivo principal la creación del resumen estructural de caminos, basado en un recorrido de arriba-abajo (top-down en inglés) de cada nodo del árbol del archivo XES. Primero, se genera un archivo de texto con los pares $<C_n, D_n>$, donde C_n es un camino estructural y D_n es la secuencia de posiciones del valor textual del camino estructural C_n. Por ejemplo, en la figura 3.1 el primer camino de la colección según el orden de aparición es C_1=*log trace string @key* y su rango de posiciones del valor textual es [1,1].

El archivo de texto generado es también procesado y se crea otro que contiene el resumen estructural de caminos. Cada línea del resumen estructural de caminos contiene los pares $<C_i, D_i>$, donde C_i es un camino estructural único del XES y D_i es la secuencia de posiciones de cada uno de los valores textuales del camino estructural C_i con respecto a la secuencia textual obtenida. Por ejemplo, la figura 3.2 representa un resumen estructural de caminos que contiene 6 caminos, donde el tercer camino de la colección es C_3=*log trace event string @key* y su secuencia textual es [3,3][5,5][9,9].

3.2.1 Estrategia de compresión del XES

La definición del esquema de un XES (Günther and Verbeek, 2009) establece varios tipos de datos para almacenar los valores de los atributos de las trazas y sus eventos: *string*, *date*, *integer*, *float*, *boolean* e *ID*. Cuando se utilizan archivos XES con decenas de millones de trazas y eventos, estos tipos de datos presentan una alta frecuencia de aparición, por lo que establecer una estrategia de compresión de estos atributos, permite disminuir los costes temporales durante la etapa de análisis de un XES. Seguidamente se define una nueva estrategia de compresión basada en la unificación de los atributos *string* a nivel de los nodos evento. La estrategia cuenta de dos pasos:

1. Definición de los nuevos atributos globales para trazas y eventos.
2. Transformación de los nodos del árbol del XES.

```
<log xes.version="1.0">
<global scope="trace">
  <string key="id" value="1"/>
</global>
<global scope="event">
  <string key="id" value="1.1"/>
  <string key="name" value="Pete"/>
  <string key="activity" value="Login"/>
  <date key="time" value="19:06:02"/>
</global>
</log>
```

Figura 3.4: Definición de los atributos globales de un archivo XES.

El primer paso de la estrategia se basa en la definición del esquema de un XES, donde el nodo registro del árbol de un XES contiene la definición de dos listas de atributos globales. Una a nivel de los nodos traza y otra a nivel de los nodos evento. Los atributos globales siempre son debidamente definidos para cada nivel en un documento XES.

Por ejemplo, en la figura 3.4 se define a nivel de traza el atributo *id* y a nivel de evento los atributos: *id*, *name*, *activity* y *date*. Como se aprecia, existen tres tipos de atributos *string* que se utilizan en la descripción de todos los eventos de un XES.

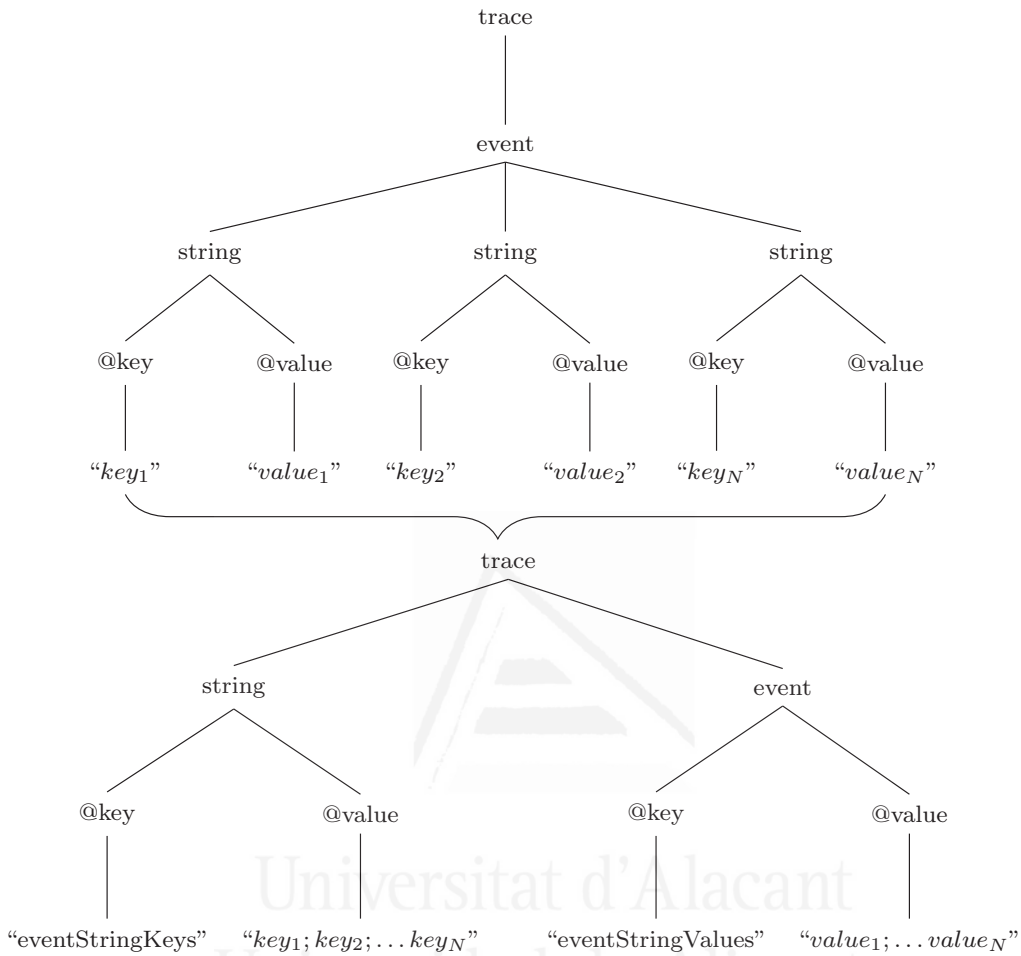
Estos tres atributos son eliminados de la definición y se inserta otro donde el tipo de dato *string* y el nivel al que pertenece se utilizan como prefijo para la definición del nuevo nombre del atributo a nivel de cada traza y evento. Siguiendo el ejemplo anterior, la nueva definición de los atributos globales quedaría como se muestra en la figura 3.5.

```
<log xes.version="1.0">
<global scope="trace">
  <string key="id" value="1"/>
  <string key="eventStringKeys" value="key1;key2"/>
</global>
<global scope="event">
  <string key="eventStringValues"
      value="value1;value2"/>
  <date key="time" value="19:06:02"/>
</global>
</log>
```

Figura 3.5: Definición de los atributos globales de un archivo XES comprimido.

Universitat d'Alacant
Universidad de Alicante

En el segundo paso se realiza un recorrido de todos los nodos del árbol del XES y se transforman basados en la nueva definición de los atributos globales planteada en el paso anterior. Por lo tanto, se unifican todos los nodos a nivel de evento que compartan el mismo atributo *string* y se crea un nuevo nodo a nivel de cada traza. El proceso de transformación para un XES genérico (representado por una traza y un evento con tres atributos *string*) se aprecia en la figura 3.6, donde se evidencia como la cantidad de nodos se reduce significativamente en comparación con el XES original.



Universitat d'Alacant
Universidad de Alicante

Figura 3.6: Transformación de los nodos del árbol de un XES basada en la estrategia de compresión.

3.3 Construcción del arreglo de sufijos

Los términos prefijo y sufijo son muy utilizados en el área de la recuperación de información no estructurada. Por ejemplo, dada una cadena $A = A_1, A_2, \ldots, A_N$ un **sufijo** $\sigma_k(A)$ se define como $A[k], A[k+1], \ldots, A[N]$

con longitud $\mu_k = N - k + 1$ y un **prefijo** $\rho(A)$ como $A[1], A[2], \ldots, A[M]$ donde $M \leq N$ con longitud $\mu = M$.

Por otra parte, para definir la relación de orden lexicográfico (representado por el símbolo $\prec$), consideremos dos sufijos $\sigma_k(A) = A[k], A[k+1], \ldots, A[N]$ y $\sigma_k(B) = B[k], B[k+1], \ldots, B[N]$. Decimos que $\sigma_k(A) \prec \sigma_k(B)$ si ambos sufijos tienen un prefijo común de largo i, y el primer valor diferente es $\sigma_k(A) < \sigma_k(B)$, donde $k = i + 1$. Un prefijo puede ser de longitud nula. Si un sufijo es prefijo del otro, se dice que el de menor longitud es el menor.

Definición 4. *Sea $A = (A_1, A_2, \ldots, A_m)$ la lista con todos los caminos estructurales de un XES ordenados lexicográficamente, donde el caracter "/" es utilizado siempre al inicio de cada camino. Un arreglo de sufijos S es un arreglo de enteros con las posiciones de inicio de cada sufijo de A en orden lexicográfico. Esto significa que $S[i]$ contiene la posición inicial del i-ésimo sufijo más pequeño en A y por tanto se cumple que para todo $1 < i \leq m$: $A[S[i-1], m] < A[S[i], m]$.*

El arreglo de sufijos S que aquí se propone se construye utilizando un método básico de ordenación mediante la comparación de los m sufijos de A (véase en el algoritmo 1 el método *compare*). El método tiene como parámetros de entrada dos punteros a los sufijos: $sufijo_1$ y $sufijo_2$. Estos sufijos se comparan y se retorna como salida *"-1"* si el $sufijo_1 < sufijo_2$, *"1"* si el $sufijo_1 > sufijo_2$ o *"0"* si son iguales. Este método requiere $\mathcal{O}(n \log n)$ comparaciones entre sufijos, pero una comparación entre dos sufijos se puede realizar en un tiempo $\mathcal{O}(n)$, entonces el tiempo completo de ejecución de este método es $\mathcal{O}(n^2 \log n)$.

Por ejemplo, dado el camino */ log trace event*, su arreglo de sufijos es $S[0, 3, 1, 2]$, donde los sufijos en orden lexicográfico son los siguientes: */ log trace event* $\prec$ *event* $\prec$ *log trace event* $\prec$ *trace event*.

El índice estructural resultante, utilizando los caminos C_3 y C_4 de la figura 3.2, se muestra en la figura 3.7. El índice incluye una estructura de datos auxiliar R, que contiene la secuencia de posiciones de los contenidos textuales para cada camino, donde R_n son los intervalos asociados de todos los nodos textuales para el camino n representado en A. Por lo tanto,

Algoritmo 1 `compare`$(offset1, offset2)$

Entrada: dos punteros a los $suffix1$ y $suffix2$

Salida: -1 si $suffix_1 < suffix2$, 1 si $suffix1 > suffix2$ o 0 si $suffix1 = suffix2$

1: **si** $suffix1 = suffix2 \wedge offset1 = offset2$ **entonces**
2: **devolver** 0
3: **si no**
4: $n1 \leftarrow offset1$
5: $n2 \leftarrow offset2$
6: **mientras** $n1 < |suffix1| \wedge n2 < |suffix2| \wedge suffix1.get(n1)equals\ (suffix_2.get(n2))$ **hacer**
7: $n1 \leftarrow n1 + 1$
8: $n2 \leftarrow n2 + 1$
9: **fin mientras**
10: **si** $n1 < |suffix1|$ **entonces**
11: **si** $n2 < |suffix2|$ **entonces**
12: **devolver** -1
13: **si no**
14: **devolver** 0
15: **fin si**
16: **si no**, **si** $n2 < |suffix2|$ **entonces**
17: **devolver** 1
18: **si no**
19: **devolver** $suffix1.get(n1).compareTo(suffix2.get(n2))$
20: **fin si**
21: **fin si**

Universitat d'Alacant
Universidad de Alicante

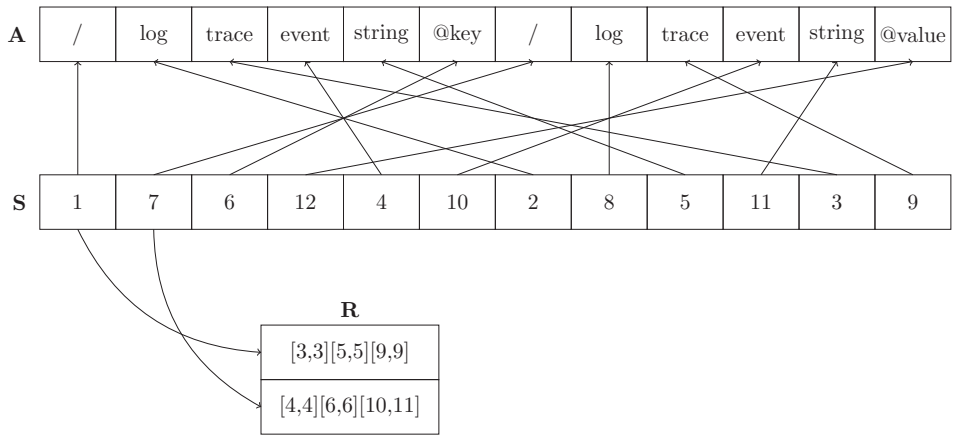


Figura 3.7: Índice estructural XES utilizando un arreglo de sufijos.

existe una correspondencia de los N primeros elementos en S con todos los elementos de R.

3.3.1 Procesamiento de consultas

Sobre cualquier documento XML es posible realizar un conjunto de consultas estructurales *XPath*. En un índice estructural XES, una consulta estructural directa $\alpha(q)$ se representa como un conjunto de nodos $y_1 y_2 \dots y_n$ que establecen relaciones directas entre ellos, separados por el caracter "/". Una consulta estructural indirecta $\beta(q)$ es un conjunto de consultas estructurales directas $\alpha(q_1)\alpha(q_2)\dots\alpha(q_i)$ separadas por "//", donde i es el total de consultas estructurales directas.

Ejecutar una consulta $\alpha(q)$ en el índice estructural XES, equivale entonces a buscar en A todos los sufijos de los cuales $\alpha(q)$ es prefijo, los cuales estarán en posiciones consecutivas de S. Por lo tanto, solo es necesario realizar dos búsquedas binarias en S, para encontrar las posiciones de inicio y fin. Hallar el subarreglo de longitud m en la lista A de longitud n toma un tiempo $\mathcal{O}(m \log n)$, dado que solo se necesita una comparación de sufijos para comparar m elementos.

Algoritmo 2 `findsubArray`$(subarray, bias)$

Entrada: subarreglo a buscar y (verdadero/falso) si es (primera/última) coincidencia

Salida: posición de inicio/fin en el arreglo de sufijos

1: $low \leftarrow 0$
2: $high \leftarrow |S| - 1$
3: **mientras** $high > low$ **hacer**
4: **si** $bias = true$ **entonces**
5: $mid \leftarrow low + (high - low)/2$
6: **si no**
7: $mid \leftarrow low + (high - low + 1)/2$
8: **fin si**
9: $ref \leftarrow A.starts(subarray, S.get(mid)$
10: **si** $ref = 0$ **entonces**
11: **si** $bias$ **entonces**
12: $high \leftarrow mid$
13: **si no**
14: $low \leftarrow mid$
15: **fin si**
16: **si no**
17: **si** $ref < 0$ **entonces**
18: $low \leftarrow mid + 1$
19: **si no**
20: $high \leftarrow mid - 1$
21: **fin si**
22: **fin si**
23: **fin mientras**
24: **si** $low = |S| \vee high < 0 \vee !A.startsWith(subarray, S.get(low))$ **entonces**
25: **si** $bias$ **entonces**
26: $low \leftarrow |S|$
27: **si no**
28: $low \leftarrow -1$
29: **fin si**
30: **fin si**
31: **devolver** low

Universitat d'Alacant
Universidad de Alicante

Cada búsqueda binaria consiste en comparar el elemento a buscar con el elemento central. Si el valor de éste es mayor que el elemento buscado se repite el procedimiento en la parte del arreglo que va desde el inicio de éste hasta el elemento tomado. En caso contrario se toma la parte del arreglo que va desde el elemento tomado hasta el final del arreglo. De esta manera obtenemos intervalos cada vez más pequeños, hasta que se obtenga un intervalo indivisible. Si el elemento no se encuentra dentro de éste último, entonces el elemento buscado no se encuentra en todo el arreglo. Si el elemento es encontrado se retorna la posición en S.

Las dos búsquedas binarias encuentran el mayor subarreglo $(S_i, S_{i+1}, \ldots, S_j)$ en S, tal que para todo $k = i, \ldots j$ el sufijo $\sigma_k = (A_{s_k}, A_{s_{k+1}}, \ldots, A_M)$ comienza con $\alpha(q)$, si $\alpha(q)$ es un prefijo de σ_k (véase el algoritmo 2 y el método *findsubarray*).

Luego, para cada sufijo σ_k, otra búsqueda binaria es ejecutada en $(S_1, \ldots S_N)$, cuando la consulta no comienza con el nodo raíz, para saber a qué camino pertenece el prefijo $\alpha(q)$ en σ_k, por lo tanto: $\alpha(q)$ pertenece al camino n-ésimo almacenado en S si $S_n \leq S_k < S_{n+1}$ y $1 \leq n \leq N$.

Cuando la consulta contiene relaciones indirectas, se divide en varias subconsultas a través del separador "//". Cada subconsulta es tratada como una consulta con relaciones directas. Con los resultados de cada una de las subconsultas se aplica un algoritmo de unión estructural para construir la respuesta final.

Se diseñaron dos algoritmos de unión estructural. El primero, (véase del algoritmo 3 el método *isContained*) con el resultado de la primera subconsulta –la posición inicial en el arreglo A– se compara si las posiciones devueltas por el resto de las consultas se encuentran después de esa posición. En caso afirmativo se devuelve la nueva posición y se repite el proceso con la subconsulta siguiente. En caso de que la posición no coincida se incrementa la posición y se repite hasta que se encuentre el valor "0" en el arreglo A, es decir, hasta que se encuentre el inicio de otro camino definido por el caracter "/".

El segundo algoritmo es similar al utilizado por Zuopeng et al. (2007). Con la posición obtenida de las dos primeras subconsultas, se verifica si existe el caracter "/" entre ambas posiciones en A (véase del algoritmo 4

Algoritmo 3 `isContained`$(pos, subquery)$

Entrada: la posición en la lista original de caminos y la subconsulta simple
Salida: 0 si la subconsulta no está después de la posición sino la posición de inicio de la subconsulta

```
1: mientras pos ≤ |A| and A.get(pos) ≠ 0 hacer
2:    si matches(pos, subquery) entonces
3:       devolver pos
4:    si no
5:       pos ← pos + 1
6:    fin si
7: fin mientras
```

el método *hasSlash*). Mientras que el resultado sea verdadero, el algoritmo se repite con el resto de las subconsultas (líneas 3-8).

Algoritmo 4 `hasSlash`(i, j)

Entrada: dos posiciones i and j en la lista original de caminos
Salida: (verdadero/falso) si (tiene/o no tiene) el caracter "/" entre i y j

```
1: result ← false
2: k ← i
3: mientras k ≤ j and result ≠ false hacer
4:    si A.get(k) ≠ 0 entonces
5:       result ← true
6:    fin si
7:    k ← k + 1
8: fin mientras
9: devolver result
```

Por ejemplo, si tenemos la consulta siguiente: $\alpha(q)$=*trace event* y se ejecuta sobre la estructura de índice que se muestra en la figura 3.7. Las dos búsquedas binarias en S retornan $S[11, 12]$ que corresponde con los sufijos σ_{11}=*trace event string @key* y σ_{12}=*trace event string @value*). Luego otra búsqueda binaria en S retorna los identificadores de caminos 1 y 2 y el

texto relevante $R_1 \bigcup R_2 = \{[3,3][5,5][9,9][4,4][6,6][10,11]\}$.

3.4 Construcción del árbol ternario de búsqueda

Un árbol ternario de búsqueda se define como una estructura de datos arbórea que almacena todos los sufijos de A, donde la raíz contiene tres subárboles distintos denominados *izquierdo*, *central* y *derecho*. Por lo tanto $TST(x)$ define un árbol ternario cuya raíz es x. $I(x)$, $C(x)$ y $D(x)$ son los subárboles *izquierdo*, *central* y *derecho* respectivamente. Además para todo $y \in TST(x)$:

- $y \in L(x) \rightarrow y < x$
- $y \in C(x) \rightarrow y = x$
- $y \in R(x) \rightarrow y > x$

La integración de un arreglo de sufijos con un árbol ternario de búsqueda acelera el procesamiento de las consultas (Aponte and Carrasco, 2012). El coste temporal de construcción de este árbol es $\mathcal{O}(n)$ y la búsqueda presenta un coste temporal $\mathcal{O}(m + \log n)$, donde m es el tamaño de la consulta y n la cantidad de nodos del árbol. En el árbol ternario que aquí se propone en cada nodo interno se almacenan los subcaminos o sufijos $\sigma_k = (A_{s_k}, A_{s_{k+1}}, \ldots, A_M)$ y en los nodos hojas las posiciones de inicio y fin de cada sufijo en S.

Según Bentley and Sedgewick (1998), para insertar elementos en un árbol ternario de búsqueda de forma balanceada, la lista a insertar debe de estar ordenada. En nuestro caso, el arreglo de sufijos S como se ha explicado anteriormente, contiene los valores enteros que representan las posiciones de inicio de cada sufijo de A en orden lexicográfico. Por tanto, primero se inserta el sufijo referenciado por el centro de S dada la fórmula $m = M/2$, donde M es la cantidad de sufijos. Luego el procedimiento continúa con las siguientes posiciones en S hacia la izquierda y derecha recursivamente.

Un sufijo siempre es dividido en μ_k nodos para la inserción en el árbol ternario. El método que inserta un sufijo en el árbol ternario se muestra

k	S_k	Prefijo	Rango en S
6	10	*event string @value #*	$[6, 7[$
3	6	*@key #*	$[3, 4[$
1	1	*/ log trace event string @key #*	$[1, 2[$
2	7	*/ log trace event string @value #*	$[2, 3[$
4	12	*@value #*	$[4, 5[$
5	4	*event string @key #*	$[5, 6[$
9	5	*string @key #*	$[9, 10[$
7	2	*log trace event string @key #*	$[7, 8[$
8	8	*log trace event string @value #*	$[8, 9[$
11	3	*trace event string @key #*	$[11, 12[$
10	11	*string @value #*	$[10, 11[$
12	9	*trace event string @value#*	$[12, 13[$

Tabla 3.1: Orden de inserción de cada sufijo en el árbol ternario.

en el algoritmo 5 y su método *insert*. De la línea (21-30) el método inserta un nuevo sufijo en los nodos internos del árbol $TST(x)$ y su rango de posiciones en S en los nodos hojas. Siempre los nodos hojas contienen como valor el caracter # y su nodo central contiene entonces los rangos en S. Si el árbol $TST(x) \neq null$ se comprueba entonces la posición para insertar el nuevo sufijo (líneas 6-20). Para ello se realiza un recorrido por el árbol ternario comenzando por el nodo raíz, mediante las propiedades que a continuación se describen. Si el nodo a insertar $y < TST(x)$ se dirige hacia el subárbol izquierdo I(x), si $y = TST(x)$ se dirige hacia subárbol central C(x) y finalmente si $y > TST(x)$ se dirige hacia el subárbol derecho D(x).

El orden de inserción de todos los sufijos σ_k en el árbol ternario se observa en la tabla 3.1. Esta tabla contiene en la primera columna la posición k en S, luego el inicio de los σ_k en A, le sigue el sufijo σ_k con el símbolo de fin de cadena # y finalmente el máximo rango de los *k-valores* tal que σ_k comience con el mismo prefijo. En la figura 3.8 se muestra el árbol ternario correspondiente al arreglo de sufijos representado en la figura 3.7 tras la

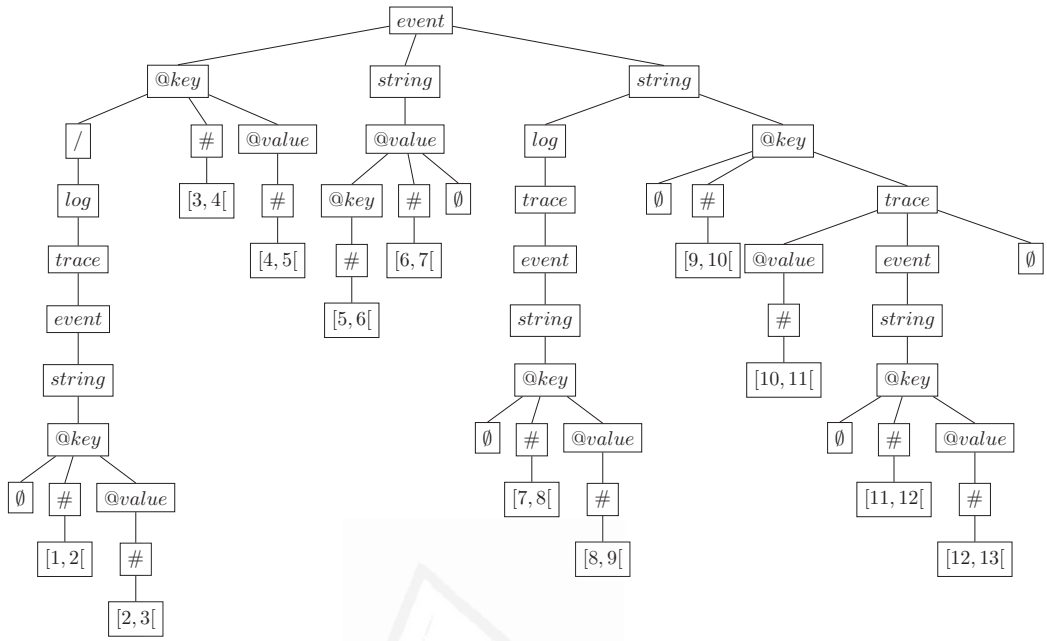


Figura 3.8: Árbol ternario de búsqueda.

inserción de todos los sufijos.

3.4.1 Procesamiento de consultas

Para buscar el contenido asociado a una consulta con relaciones directas $\alpha(q) = y_1 y_2 \dots y_k$, primero se visita el nodo raíz, si $y_1 < TST(x)$ se transfiere el recorrido hacia el subárbol izquierdo, si $y_1 > TST(x)$ se transfiere hacia el subárbol derecho y si $y_1 = TST(x)$ hacia el subárbol central. El procedimiento se repite para y_{i+1}, donde $1 \leq i < k$ (véase del algoritmo 6 el método *search*). El algoritmo utiliza los valores enteros "1" para referirse al nodo central, "0" para el nodo izquierdo y "2" para el nodo derecho. La llamada al método *getContent* devuelve los valores de todos los nodos hojas que se encuentran por debajo del nodo contexto.

Por ejemplo, si tenemos la consulta $\alpha(q) =$ */ log trace event string @key*. La búsqueda sobre el árbol ternario TST retorna el nodo hoja con el rango

Algoritmo 5 `insert`$(suffix, range)$

Entrada: sufijo a insertar y rango de posiciones en el arreglo de sufijos
Salida: nodo insertado

1: $node \leftarrow root$
2: $prev \leftarrow null$
3: $len \leftarrow 0$
4: $test \leftarrow 0$
5: $suffix.add(0)$
6: **mientras** $node \neq null$ **hacer**
7: $test = node.compare(suffix.get(len))$
8: $prev \leftarrow node$
9: **si** $test < 0$ **entonces**
10: $node \leftarrow node.getChild(1)$
11: **si no**, **si** $test > 0$ **entonces**
12: $node \leftarrow node.getChild(2)$
13: **si no**
14: $node \leftarrow node.getChild(0)$
15: **fin si**
16: **si** $++len == suffix.size()$ **entonces**
17: $node \leftarrow new\ LeafNode(range)$;
18: **devolver** $node$;
19: **fin si**
20: **fin mientras**
21: **mientras** $len < path.size()$ **hacer**
22: $node \leftarrow new\ InternalNode(suffix.get(len))$;
23: **si** $root == null$ **entonces**
24: $root \leftarrow node$
25: **si no**, **si** $prev \neq null$ **entonces**
26: $prev.setChild(Integer.signum(test) + 1, node)$
27: **fin si**
28: $prev \leftarrow node$
29: $test = 0$
30: $++len$
31: **fin mientras**
32: **si** $prev \neq null$ **entonces**
33: $prev.setChild(0, node)$
34: **fin si**
35: **devolver** $node$

Universitat d'Alacant
Universidad de Alicante

Algoritmo 6 `search`(*path*)

Entrada: Camino a buscar

```
1: node ← root
2: si path.isEmpty() entonces
3:    devolver root.getContent()
4: fin si
5: mientras node! = null hacer
6:    test = node.compare(path.get(len))
7:    si test < 0 entonces
8:       node ← node.getChild(1)
9:    si no, si test > 0 entonces
10:      node ← node.getChild(2)
11:   si no
12:      node ← node.getChild(0)
13:   fin si
14:   si ++len == path.size() entonces
15:      node.getContent()
16:   fin si
17: fin mientras
18: devolver null
```

de posiciones $[1, 2[$ en S. $S[1]$ corresponde con el sufijo σ_1=(*/ log trace event string @key*). En este caso, la consulta comienza con el nodo raíz, por lo tanto, la posición en S corresponde con el número del camino estructural y su secuencia textual es $R_1 = \{[3, 3][5, 5][9, 9]\}$.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

4. Propuesta de un índice de contenidos

> No tengo grandes ideas. A veces tengo pequeñas ideas que parecen funcionar
>
> Matt Mullenweg

Como parte de una extensión a lo expuesto en el capítulo anterior y aprovechando las propiedades del XES, en este capítulo se describe una propuesta de índice de contenidos para el manejo eficiente de los valores textuales de un archivo XES. Esta propuesta surge por la necesidad de manipular archivos XES a través de los clasificadores de eventos, los cuales definen clases de identidad para cada evento dentro de un registro de trazas. Una solución secuencial se basa en la construcción y manejo del índice y sus consultas utilizando una estructura o diccionario *rank/select*. Finalmente, se propone una solución paralela/distribuida para la construcción del índice y el procesamiento de las consultas utilizando la plataforma Hadoop y el paradigma de programación MapReduce.

Universidad de Alicante

4.1 Índice de contenidos XES

La propuesta parte de la generación de una secuencia ordenada S sobre los contenidos textuales de los nodos hojas del árbol que representa el archivo XES. Cada elemento de la secuencia está representado por el par $<C_i, V_i>$, donde C_i es el valor correspondiente al clasificador de evento y V_i es una cadena en formato JSON con los atributos de la traza y del evento asociado al valor del clasificador C_i. El total de elementos de S es igual al producto

de la cantidad de clasificadores y la cantidad de eventos. En lo adelante la secuencia ordenada la llamaremos lista invertida.

```
<log>
  <classifier name="activity" keys="activity"/>
  <classifier name="originator" keys="name"/>
  <trace>
    <string key="id" value="1"/>
    <event>
      <string key="id" value="1.1"/>
      <string key="name" value="Pete"/>
      <date key="date" value="2011-01-04"/>
      <string key="activity" value="login"/>
    </event>
    <event>
      <string key="id" value="1.2"/>
      <string key="name" value="Pete"/>
      <date key="date" value="2011-01-04"/>
      <string key="activity" value="logout"/>
    </event>
  </trace>
</log>
```

Figura 4.1: Ejemplo de un documento XES.

Universitat d'Alacant
Universidad de Alicante

Por ejemplo, dado el documento XES de la figura 4.1 y el clasificador de evento *activity*, un elemento de dicha lista invertida pudiera ser <login,trace: {id:"1", event:[{id:"1.1", name:"Pete", date: "2011-01-04", activity: "login" }]}>, donde *login* es el valor del clasificador del evento y el resto es la cadena en formato JSON, que contiene los atributos de la traza y el evento correspondiente.

Definición 5. *Un índice de contenidos XES se define como una estructura que contiene la secuencia* $<C_j, L_j>$*, tal que para cada valor único del clasificador* C_j*, se le asocia una lista* L_j *con los atributos de las trazas y de los eventos correspondientes al valor del clasificador* C_j*.*

4.2 Construcción del índice de contenidos con diccionarios rank/select

Si se tiene en cuenta que para cada V_i de la secuencia ordenada S, le corresponde una cadena en formato JSON con los atributos de la traza y del evento asociado al valor del clasificador C_i, a cada palabra w que aparece en V_i es posible asignarle una posición j de ocurrencia en S. Por ejemplo, en <login, trace: {id:"1",event:[{id:"1.1", name:"Pete", date: "2011-01-04", activity:"login"}]}>, el valor del clasificador *login* está asociado al evento que se encuentra en el rango de posiciones [3,10]. Basado en el archivo XES de la figura 4.1, el rango de posiciones para todos los valores de los clasificadores de eventos en orden lexicográfico se aprecia en la tabla 4.1.

Valor	**Rangos**
login	[3,10]
logout	[13,20]
Pete	[3,10][13,20]

Tabla 4.1: Valores de los clasificadores de eventos y su rango de posiciones.

Por otra parte, los diccionarios o estructura sucintas *rank/select*, son utilizados fundamentalmente para la indexación de textos planos y comprimidos. Entre las soluciones para estructuras de datos planas se encuentra la implementación desarrollada por Vigna (2008), que la utilizamos a lo largo de esta sección. Para toda estructura de datos *rank/select*, dada una secuencia binaria $B[1, n]$, se definen las siguientes operaciones:

- $rank_b(B, i)$ equivale al número de ocurrencias del bit b en la secuencia de bits $B[1, i]$.
- $select_b(B, i)$ equivale a la posición de ocurrencia del i-ésimo bit b en B.

Por ejemplo, si $B = 011011010101011010110$, $select_1(B, 5) = 8$ y $rank_1(B, 5) = 3$.

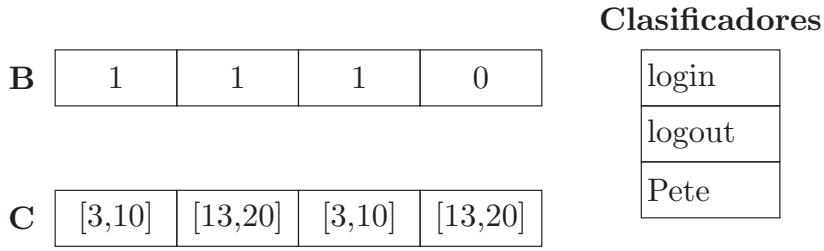


Figura 4.2: Índice de contenidos con la estructura rank/select.

Es posible utilizar este tipo de estructura para indexar todos los valores textuales de un XML (Báez et al., 2015). En este caso, para la construcción de un índice de contenidos utilizando un diccionario *rank/select*, primero se ordenan y se almacenan lexicográficamente en una lista todos los valores de los clasificadores de eventos. Se crea también una secuencia binaria B, donde cada "1" representa la primera aparición de ese valor del clasificador de evento y cada "0" las restantes apariciones. Paralelo a esto se construye una secuencia C con los rangos de apariciones de cada evento dentro de la lista invertida. La longitud de la secuencia binaria B y de la secuencia de rangos de apariciones de cada evento C es la misma. La estructura del índice resultante se observa en la figura 4.2.

Universitat d'Alacant
Universidad de Alicante

4.2.1 Procesamiento de consultas

Para el procesamiento de una consulta sobre este índice de contenidos, se parte de la consulta que se desea procesar. Por ejemplo, si se quieren recuperar todos los eventos cuyo originador haya sido *"Pete"*, primero se realiza una búsqueda binaria sobre la lista que contiene todos los valores de los clasificadores de eventos. El resultado de la búsqueda binaria es un índice i que corresponde con la posición de aparición en la lista. Luego mediante la operación $select_1(B, i)$ se retorna la posición de ocurrencia del *i-ésimo* bit "1" en B. Por lo tanto, para el ejemplo en cuestión sería $select_1(B, 3)$, que retorna la posición de inicio del tercer "1" dentro de la secuencia binaria y $select_1(B, 4) - 1$ retorna la posición final del último "0" después de la

posición de inicio. La respuesta final corresponde con el rango de posiciones $C_3 \bigcup C_4$={[3,10][13,20]}.

4.3 Construcción del índice de contenidos con la plataforma Hadoop

Una forma eficaz para tratar grandes volúmenes de información es mediante la utilización de uno de los conceptos fundamentales de la informática, divide y vencerás. La idea básica es dividir los grandes problemas de la computación en pequeños subproblemas, los cuales pueden ser atendidos de forma paralela. Por ejemplo, en diferentes hilos en un núcleo de un procesador, en diferentes núcleos en un procesador multinúcleo, en múltiples procesadores en un solo ordenador o en muchos ordenadores dentro de un clúster. En esta sección se describe el flujo de construcción del índice de contenidos en la plataforma Hadoop.

En la figura 4.3 se muestra el flujo de construcción de un índice de contenidos XES. El flujo comienza con el análisis sintáctico de los archivos XES, donde es generada una lista invertida. Cada elemento de la lista invertida contiene el par $<C_i, V_i>$. Esta lista invertida se almacena en el sistema de archivo HDFS de la plataforma Hadoop. El sistema de archivo es el encargado de particionarla en N bloques de tamaño fijo llamados *splits*. Estos bloques se distribuyen y se replican en los nodos del clúster con el objetivo de brindar balance de carga y tolerancia a fallas. El tamaño de un *split* típicamente es de 16 o 64 Mb, aunque puede ser modificado por el usuario. El paso siguiente del flujo es el procesamiento MapReduce, que se lleva a cabo a través de las funciones *map* y *reduce*.

Conceptualmente, las funciones *map* y *reduce* (véase una representación gráfica de ambas funciones en las figuras 4.4 y 4.5) se definen como se muestra a continuación:

- La función *map* recibe un par clave-valor y devuelve un conjunto de pares clave-valor intermedio: $map(k1, v1) \rightarrow lista(k2, v2)$

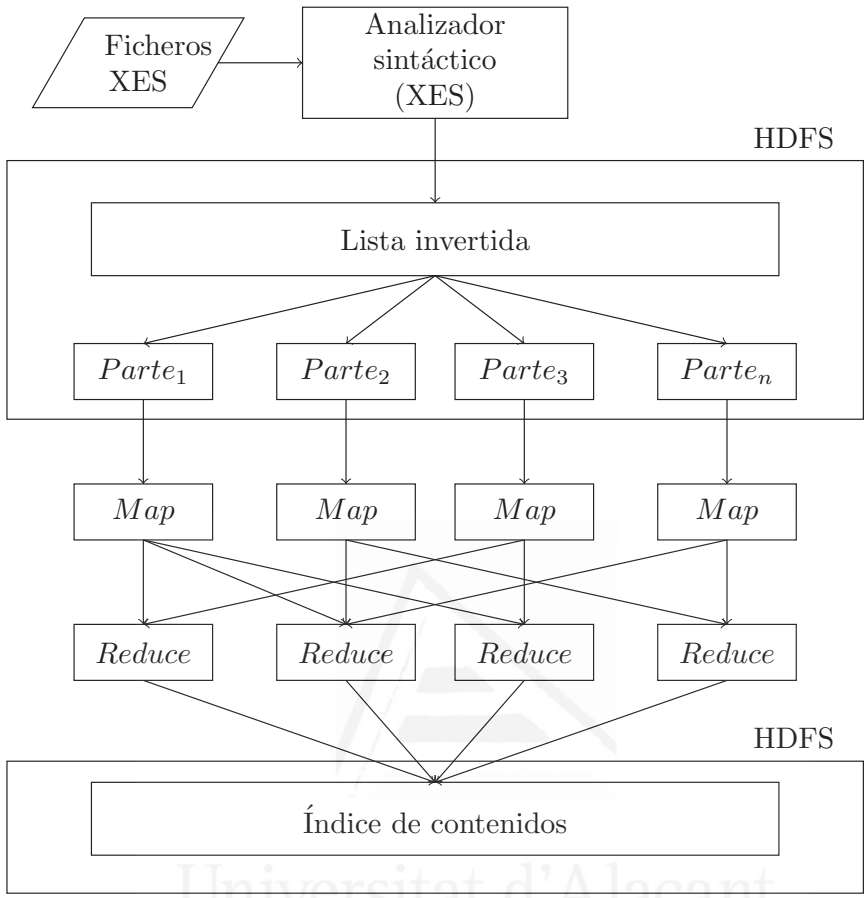


Figura 4.3: Flujo de construcción del índice de contenidos XES en la plataforma Hadoop.

Universitat d'Alacant
Universidad de Alicante

- La función *reduce* recibe una clave y su conjunto de valores asociados y los fusiona para formar un conjunto de valores posiblemente más pequeños: $reduce(k2, lista(v2)) \rightarrow lista(k3, v3)$

El ambiente MapReduce replica las funciones *map* y *reduce* en los nodos del clúster de tal manera, que las réplicas de ambas funciones se ejecutan al mismo tiempo en nodos distintos. Los datos de entrada –lista invertida–

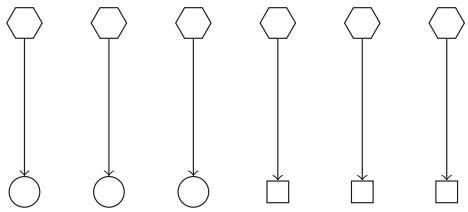


Figura 4.4: La función map.

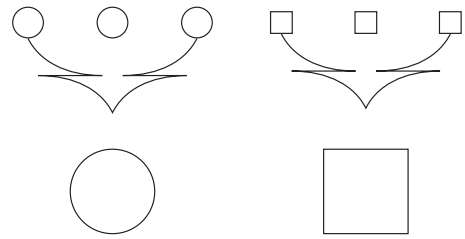


Figura 4.5: La función reduce.

se procesan por medio de la función *map* que lee el archivo de entrada de manera iterativa a través de un par de parámetros llamados *clave* y *valor*.

```
Pete&{trace:{id:''1'',event:[{id:''1.1'',name:''Pete'',
    date:''2011-01-04'',activity:''login''}]}}
login&{trace:{id:''1'',event:[{id:''1.1'',name:''Pete'',
    date:''2011-01-04'',activity:''login''}]}}
Pete&{trace:{id:''1'',event:[{id:''1.2'',name:''Pete'',
    date:''2011-01-04'',activity:''logout''}]}}
logout&{trace:{id:''1'',event:[{id:''1.2'',name:''Pete'',
    date:''2011-01-04'',activity:''logout''}]}}
```

Figura 4.6: Ejemplo de una lista invertida.

Universitat d'Alacant
Universidad de Alicante

Por ejemplo, en la figura 4.6 se muestra una lista invertida generada a partir del archivo XES de ejemplo de la figura 4.1, donde cada línea es representada por un par $<C_i, V_i>$, separado por el caracter "&". Por cada llamada de la función *map* se lee una línea de datos y a la función se le envía como *clave* el valor del clasificador del evento y como *valor* la lista en formato JSON con los atributos de la traza y el evento asociado al valor del clasificador C_i. Por cada llamada de la función *map* se realiza el procesamiento especificado en el algoritmo 7 y al final se emite una lista con los pares clave-valor.

Cuando se terminan de ejecutar todas las tareas *map* que emiten los pares intermedios clave-valor, una tarea *reduce* recibe los datos de diferentes

Algoritmo 7 `map`$(key, value, context)$

Entrada: el par (clave-valor) y el contexto de salida

```
1: tokens ← line.split("&")
2: key ← tokens[0]
3: value ← tokens[1]
4: context.write(key, value)
```

nodos, entonces la operación *merge* se encarga de crear la lista de valores asociada con una misma clave. La función *reduce* es llamada una vez por cada clave distinta generada en la etapa anterior en el orden establecido. La función *reduce* puede iterar a través de los valores asociados con esa clave y obtener como resultado cero o más valores que son almacenados en el sistema de archivo HDFS.

El pseudocódigo de la función *reduce* se muestra en el algoritmo 8. En la figura 4.7 se muestra un ejemplo del proceso de construcción del índice de contenidos XES en la plataforma Hadoop y el paradigma MapReduce, dada la lista invertida de la figura 4.6.

Algoritmo 8 `reduce`$(key, values, context)$

Entrada: una clave con su lista de valores y el contexto de salida

```
1: para todo (value in values) hacer
2:     out ← add(value)
3: fin para
4: context.write(key, out)
```

Universitat d'Alacant
Universidad de Alicante

4.3.1 Procesamiento de consultas

Para ejecutar cualquier consulta sobre el índice, es necesario ejecutar dos procesos MapReduce, uno para seleccionar los eventos que coinciden con la clave de búsqueda y el otro para seleccionar los eventos que pertenecen a una misma traza. Como se mencionó en la sección anterior, el índice de contenidos es almacenado en el sistema de archivo HDFS. Para el primer

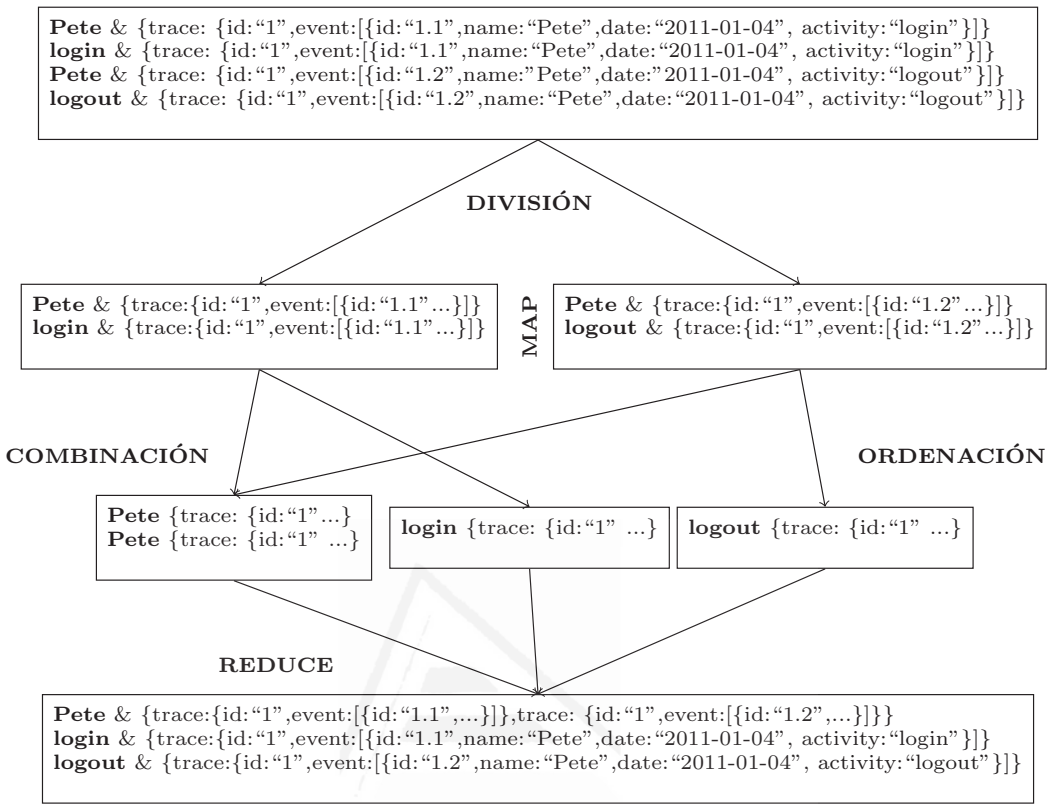


Figura 4.7: Ejemplo de construcción del índice de contenidos XES en la plataforma Hadoop.

Universitat d'Alacant
Universidad de Alicante

proceso MapReduce, por cada llamada de la función *map* se realiza el procesamiento especificado en el algoritmo 9 y al final se emite una lista con los pares clave-valor, donde cada clave coincide con el valor de la consulta.

Por otra parte la tarea *reduce*, igual que en la construcción del índice de contenidos, recibe los datos de los diferentes nodos y la operación *merge* se encarga de crear la lista de valores asociada con una misma clave y ejecuta la función *reduce*. El resultado final de la tarea *reduce* produce una lista con los pares clave-valor y es almacenada igualmente en el sistema de archivo HDFS. Se puede utilizar la misma función *reduce* descrita en el proceso de

construcción del índice de contenidos (algoritmo 8).

Un ejemplo de procesamiento de una consulta sobre un índice de contenidos XES, se muestra en la figura 4.8. En el mismo se recuperan todos los eventos cuyas actividades fueron ejecutadas por el usuario *"Pete"*.

Durante el segundo proceso MapReduce, como se mencionó anteriormente, se seleccionan los eventos que pertenecen a una misma traza, en este caso la función *map* emite como *clave* los identificadores de cada traza y como valor los atributos de la traza y su evento correspondiente. Al igual que en los casos anteriores, la tarea *reduce* recibe los datos de los diferentes nodos y a través de la operación *merge* se crea una lista de valores asociadas al mismo identificador de traza. La lista final generada por la tarea *reduce*, es también almacenada en el sistema de archivo HDFS.

Por ejemplo, si la respuesta a una consulta es {trace: {id:"1",event: [{id:"1.1"}]}} y {trace: {id:"1", event: [{id:"1.1"}]}}, donde ambos eventos pertenecen a la misma traza. El resultado luego de aplicar el segundo proceso MapReduce es {trace: {id:"1", event:[{id:"1.1"}], event:[{id:"1.2"}]}}. En la figura 4.8 esta etapa de procesamiento es representada de forma simplificada con líneas discontinuas.

Algoritmo 9 `mapQuery`$(key, value, context, query)$

Entrada: el par (clave-valor), el contexto de salida y la consulta

1: $tokens \leftarrow line.split(''\&'')$
2: $key \leftarrow tokens[0]$
3: $value \leftarrow tokens[1]$
4: **si** $key = query$ **entonces**
5: $\quad context.write(key, value)$
6: **fin si**

Universitat d'Alacant
Universidad de Alicante

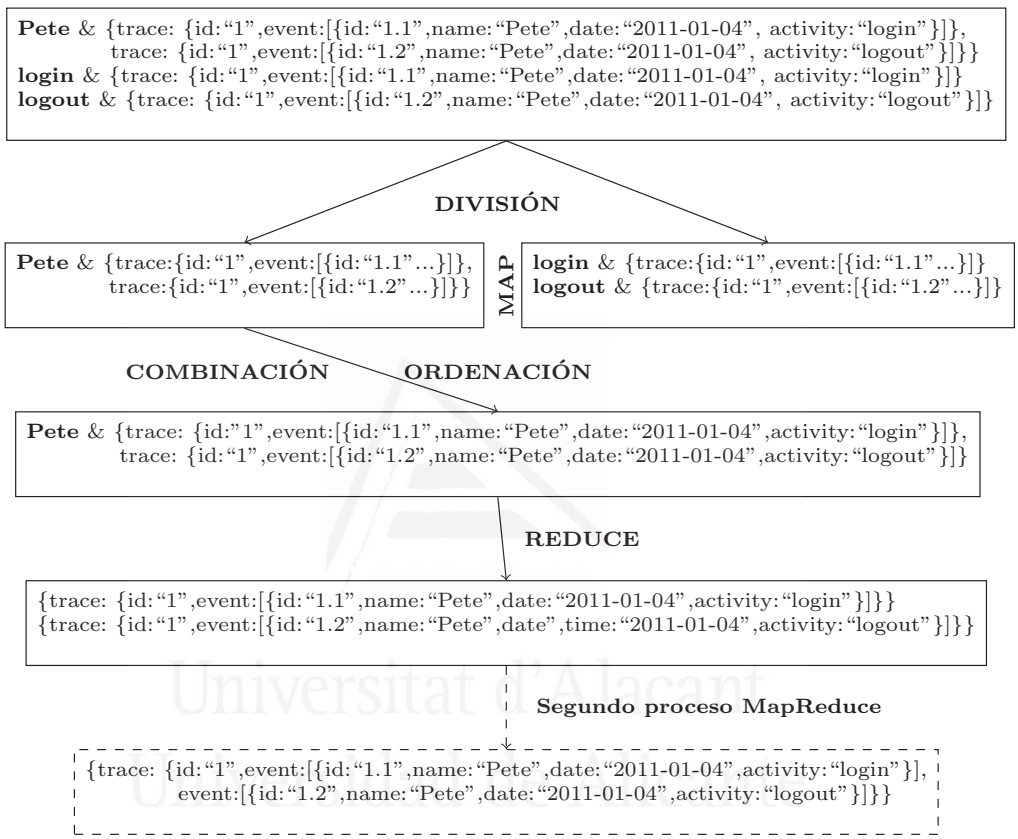


Figura 4.8: Ejemplo de procesamiento de una consulta sobre el índice de contenidos XES en la plataforma Hadoop.

Universitat d'Alacant
Universidad de Alicante

5. Experimentación

> Recordarás algo de lo que leas, bastante de lo que oigas, mucho de lo que veas, y todo lo que hagas
>
> John Little B.

En este capítulo se presentan los resultados obtenidos durante la realización de diferentes experimentos. El primer objetivo ha sido generar aleatoriamente un conjunto de ficheros XES que sirven de base para ejecutar los experimentos.

En el primer experimento se analiza el comportamiento durante la carga en memoria de un conjunto de trazas almacenadas en un archivo XES. Para ello se comparan dos propuestas: el índice estructural XES (SATST), descrito en el capítulo 3, con la propuesta que utiliza un analizador *lazy*. Esta última propuesta ha sido utilizada por todos los algoritmos de descubrimiento de modelos, implementados en el grupo de investigación PRONEG.

En el segundo experimento se analiza el radio de compresión del índice estructural XES. Para su ejecución se extraen los contenidos textuales del XES y se mantiene solo la información estructural. El resultado se compara con otras propuestas como: XQEngine[1], PCIM y NCIM. En el tercer experimento se evalúa el comportamiento del peso de un fichero XES tras aplicar la estrategia de compresión que se describe en la sección 3.2.1. En el cuarto y último experimento, se analiza el coste temporal de construcción del índice de contenidos en la plataforma Hadoop y de ejecución de diferentes consultas.

[1] http://xqengine.sourceforge.net/

Trazas	Peso (GB)			Caminos
	Estructura	Contenido	Total	(comunes)
10 000	0.56	0.79	1.35	1×10^8 (20)
20 000	1.11	1.60	2.71	2×10^8 (20)
30 000	1.63	2.45	4.08	3×10^8 (20)
40 000	2.23	3.22	5.45	4×10^8 (20)
50 000	2.80	4.03	6.83	5×10^8 (20)
60 000	3.29	4.73	8.02	6×10^8 (20)

Tabla 5.1: Ficheros XES variando el número de trazas.

5.1 Configuración de la experimentación

Para poder ejecutar los diferentes experimentos se implementó una herramienta que genera aleatoriamente diferentes archivos XES, basado en los parámetros: extensiones, atributos globales a nivel de traza y evento, clasificadores de eventos, cantidad de originadores, cantidad de actividades, entre otros. Los archivos XES generados aleatoriamente contienen toda la información necesaria (profundidad y anchura del árbol) para la correcta ejecución de los experimentos. En la figura 5.1 se muestran algunos de estos parámetros que contiene el encabezamiento de un documento XES.

Se generaron diferentes archivos XES, y se tomó una muestra de 16 de ellos, 8 variando la cantidad de trazas (véase la tabla 5.1) y 8 variando el número total de eventos por traza (véase la tabla 5.2). Cada traza se identifica por un atributo *string* y cada evento representa una actividad que contiene 4 atributos *string* y un atributo *date*. Todos los valores que contienen los atributos son generados aleatoriamente.

Como se muestran en las tablas 5.1 y 5.2, para ambos conjuntos de ficheros XES se registraron los indicadores siguientes: eventos, eventos por traza, peso, total de caminos y caminos comunes. El peso total de cada fichero se compone del peso de la estructura del documento y del peso del contenido textual. De manera general el peso de la estructura es aproximadamente un 40 % de su peso total.

```
<?xml version="1.0" encoding="UTF-8" ?>
<log xes.version="1.0" xes.features="nested-attributes"
    openxes.version="1.0RC7"
    xmlns="http://www.xes-standard.org/">
 <extension name="Lifecycle" prefix="lifecycle"
     uri="http://www.xes-standard.org/lifecycle.xesext"/>
 <extension name="Organizational" prefix="org"
     uri="http://www.xes-standard.org/org.xesext"/>
 <extension name="Time" prefix="time"
     uri="http://www.xes-standard.org/time.xesext"/>
 <extension name="Concept" prefix="concept"
     uri="http://www.xes-standard.org/concept.xesext"/>
 <extension name="Semantic" prefix="semantic"
     uri="http://www.xes-standard.org/semantic.xesext"/>
 <global scope="trace">
  <string key="concept:name" value="__INVALID__"/>
 </global>
 <global scope="event">
  <string key="concept:name" value="__INVALID__"/>
  <string key="lifecycle:transition" value="complete"/>
 </global>
 <classifier name="MXML Legacy Classifier"
     keys="concept:name lifecycle:transition"/>
 <classifier name="Event Name" keys="concept:name"/>
 <classifier name="Resource" keys="org:resource"/>
 <string key="source" value="Rapid Synthesizer"/>
 <string key="concept:name" value="excercise1.mxml"/>
 <string key="lifecycle:model" value="standard"/>
</log>
```

Figura 5.1: Encabezado de un documento XES.

Por otra parte, la cantidad de caminos para un fichero XES puede llegar hasta 659 millones y de ellos solamente 20 caminos comunes. Esto se debe a que un documento XES por definición es un documento XML estrecho en profundidad y la anchura depende entonces del número de trazas y eventos que contiene, lo que provoca un alto factor de repetición de los caminos del árbol.

Universidad de Alicante

Eventos por traza	Peso (GB)			Caminos (comunes)
	Estructura	Contenido	Total	
110	1.39	1.99	3.38	1.1×10^8 (20)
120	1.53	2.21	3.74	1.2×10^8 (20)
130	1.67	2.40	4.07	1.3×10^8 (20)
140	1.81	2.61	4.42	1.4×10^8 (20)
150	1.95	2.80	4.75	1.5×10^8 (20)
160	2.09	3.00	5.09	1.6×10^8 (20)

Tabla 5.2: Ficheros XES variando el número de eventos por traza.

Para la experimentación de las técnicas de indexación secuenciales (tres primeros experimentos), se utilizó un ordenador personal: Intel Celeron con CPU G1830 a 2.80 Ghz, 4 GB de memoria RAM, 1 TB de disco duro y Sistema Operativo Window 8.1. Para la experimentación en un ambiente paralelo/distribuido (cuarto experimento), se creó un clúster Hadoop con un total de 5 ordenadores (1 maestro y 4 esclavos). Cada ordenador presenta las siguientes características: Dual Core con 4GB de memoria RAM, 1 TB de disco duro y Sistema Operativo Ubuntu 14.04.

Universitat d'Alacant
Universidad de Alicante

5.2 Evaluación de la recuperación de trazas

El primer experimento consistió en cargar en memoria un XES completo para evaluar el tiempo de ejecución. Se utilizó el juego de datos de la tabla 5.1, donde se incrementa el número de trazas hasta un total de 60000. En este experimento se comparan dos estrategias. Por una parte, la utilización de un analizador *lazy* que carga todo el XES en una estructura de datos que almacena todos los atributos de las trazas con sus eventos correspondientes, utilizado por el algoritmo *Fuzzy Minner* (Günther and Van Der Aalst, 2007). Por otra parte, una estrategia que utiliza el índice estructural SATST. Esta última estrategia además de analizar sintácticamente el fichero XES, construye un índice estructural con los caminos comunes

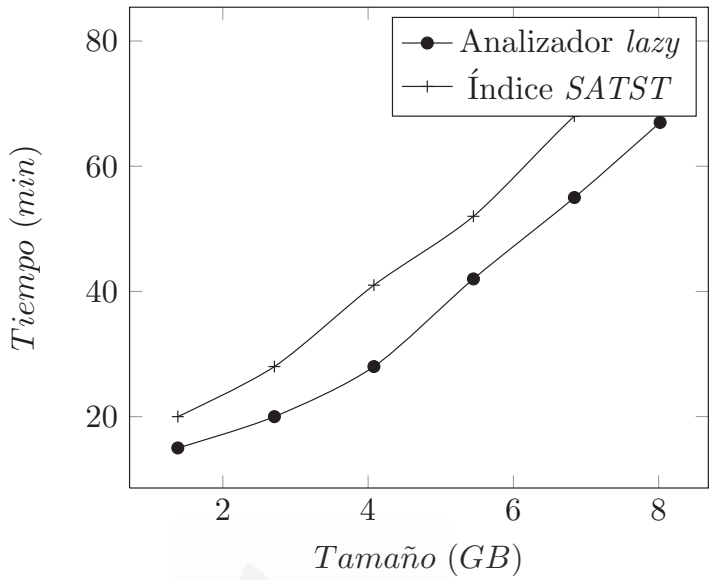


Figura 5.2: Cantidad de trazas analizadas.

del árbol. Todos los tiempos de ejecución fueron registrados y se graficaron según se muestra en la figura 5.2.

En la gráfica anterior (véase la figura 5.2) se observa que ambas estrategias tienen un comportamiento similar en cuanto a los costes temporales. Se aprecia gráficamente que la segunda estrategia consume mayor tiempo que la primera, ya que para la creación del índice estructural es preciso realizar un análisis sintáctico del XES y la ejecución de diferentes algoritmos de uniones estructurales.

Es válido destacar que esta última estrategia tiene como principal ventaja que se pueden ejecutar consultas estructurales con relaciones directas e indirectas. El tiempo promedio de ejecución de una consulta estructural con relaciones directas, en una estructura de este tipo, es aproximadamente 4μs (Aponte and Carrasco, 2012).

5.3 Análisis del radio de compresión del índice SATST

En el segundo experimento se compara el radio de compresión de la estrategia SATST con respecto al peso que ocupa la parte estructural del archivo XES. El resultado obtenido se compara con las estrategias XQEngine, PCIM y NCIM. Como XQEngine es la única entre las cuatro estrategias que construye el índice de contenidos junto al índice estructural del documento, se decidió extraer para todos los ficheros XES su contenido textual para calcular el radio de compresión con mayor precisión. Se utilizó la siguiente fórmula:

$$Radio = \frac{Tamaño\,sin\,comprimir\ -\ Tamaño\,comprimido}{Tamaño\,sin\,comprimir} \times 100\,\% \quad (5.1)$$

Se evaluaron los diferentes radios de compresión para cada estrategia y los resultados se graficaron según muestra la figura 5.3. Como se observa XQEngine presenta el menor radio de compresión. PCIM por su parte logra mayor radio que NCIM debido a que el primero utiliza un resumen estructural y almacena el índice en menos estructuras de datos que NCIM. Por otra parte, la propuesta SATST presenta un ligero radio de compresión mayor que PCIM. Esta propuesta utiliza también resúmenes estructurales y emplea un arreglo de sufijos y un árbol ternario de búsqueda, las cuales siempre se han reportado como estructuras eficientes en el espacio de almacenamiento.

Universidad de Alicante

5.4 Análisis del esquema de compresión de un XES

Para el tercer experimento se implementó un algoritmo basado en la estrategia de compresión planteada en la sección 3.2.1. Luego de ejecutar el algoritmo para cada archivo XES, se registraron los nuevos pesos, así como

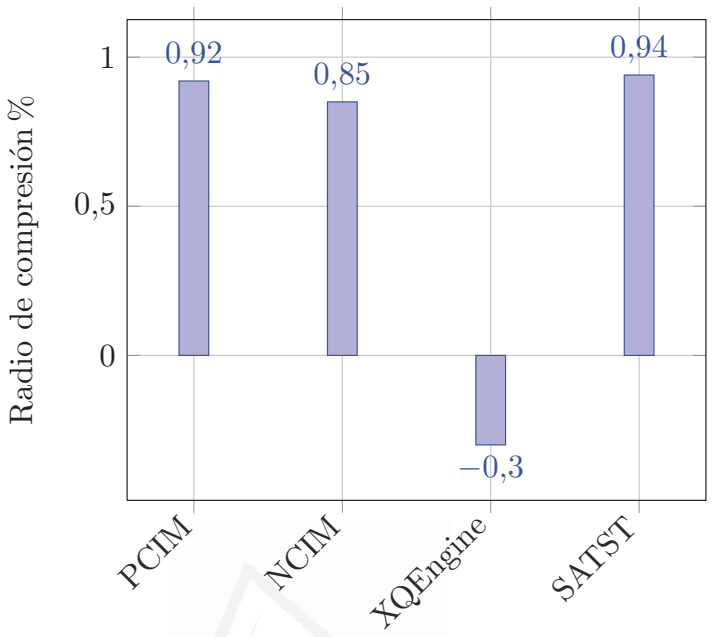


Figura 5.3: Radio de compresión.

Universitat d'Alacant
Universidad de Alicante

la cantidad de caminos eliminados para cada fichero XES. Los resultados se registraron en las tablas 5.3 y 5.4.

Para una referencia gráfica de los resultados, se construyeron también las gráficas que se muestran en las figuras 5.4 y 5.5, donde se observa que para todos los casos la función que pertenece al peso comprimido, siempre se encuentra por debajo de la función que representa al peso original de cada archivo XES. Finalmente, con la aplicación de esta estrategia se logró un radio de compresión como promedio de un 62 %.

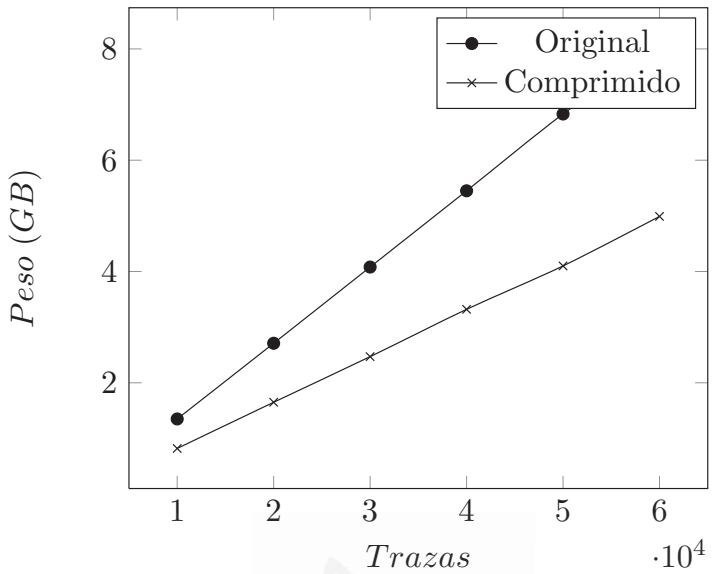


Figura 5.4: Comportamiento del peso de los ficheros XES de la tabla 5.1.

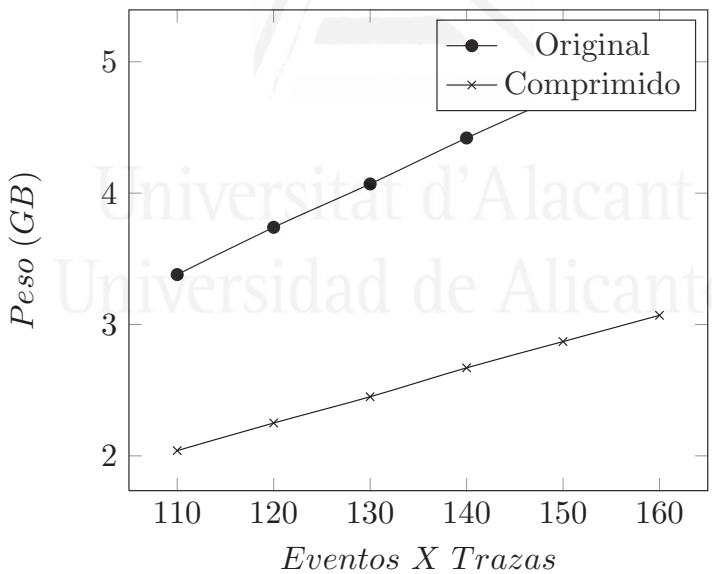


Figura 5.5: Comportamiento del peso de los ficheros XES de la tabla 5.2.

Universitat d'Alacant
Universidad de Alicante

Peso (GB)	Peso comprimido	Caminos eliminados
1,35	0,82	$59{,}9\times10^6$
2,71	1,65	$119{,}9\times10^6$
4,08	2,47	$179{,}9\times10^6$
5,45	3,32	$239{,}9\times10^6$
6,83	4,10	$299{,}9\times10^6$
8,02	4,99	$359{,}8\times10^6$

Tabla 5.3: Ficheros XES comprimidos variando el número de trazas.

Peso (GB)	Peso comprimido	Caminos eliminados
3,38	2,04	$65{,}8\times10^6$
3,74	2,25	$71{,}8\times10^6$
4,07	2,45	$77{,}8\times10^6$
4,42	2,67	$83{,}8\times10^6$
4,75	2,87	$89{,}8\times10^6$
5,09	3,07	$95{,}8\times10^6$

Tabla 5.4: Ficheros XES comprimidos variando el número de eventos por traza.

Universitat d'Alacant
Universidad de Alicante

5.5 Evaluación del tiempo de construcción del índice de contenidos

El cuarto y último experimento consistió en la evaluación del tiempo de construcción del índice de contenidos sobre la plataforma paralela/distribuida Hadoop. El primer paso consistió en la implementación de las tareas MapReduce según los algoritmos descritos en el capítulo 4 y se desplegaron en la plataforma Hadoop. Se decidió evaluar el tiempo de construcción del índice de contenidos siguiendo las métricas cantidad de nodos y peso de los archivos XES. También se evaluaron los tiempos promedios de ejecución de un conjunto de consultas.

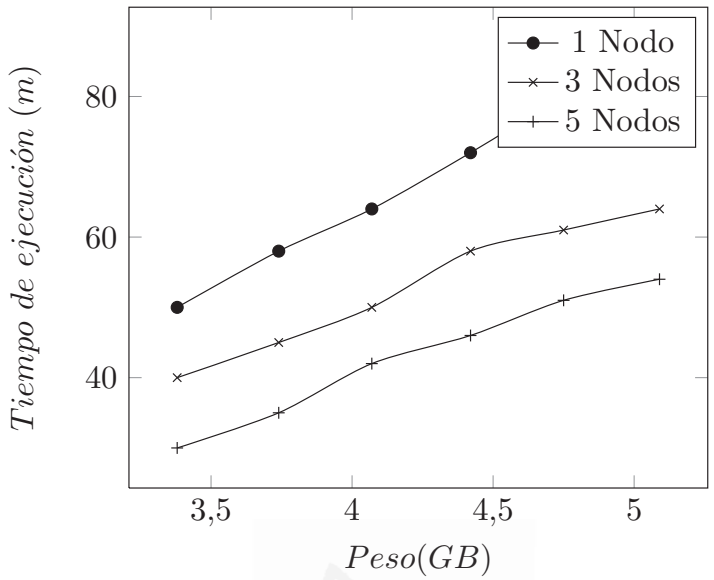


Figura 5.6: Tiempo de construcción del índice de contenidos.

Para evaluar el tiempo de construcción del índice de contenidos y comprobar la escalabilidad de la estrategia, se utilizó el segundo juego de datos, donde se incrementa el número de eventos por traza para cada fichero XES. El tiempo de construcción analizado incluye el análisis del fichero XES, construcción de la lista invertida, la ejecución del proceso MapReduce y el almacenamiento del índice de contenidos en el sistema de archivo HDFS. Los tiempos registrados se graficaron según muestra la figura 5.6.

En las gráficas representadas en la figura 5.6 se puede observar que con el incremento de la cantidad de nodos que atienden las tareas MapReduce, ambas gráficas consumen menor tiempo que la representada por un solo nodo, ya que las tareas son distribuidas por los diferentes nodos del clúster. Por lo tanto, mediante una distribución de los datos por los diferentes nodos del clúster, no se ve afectado el tiempo de construcción del índice de contenidos cuando se incrementa el tamaño de los archivos XES.

Debemos resaltar además que la plataforma Hadoop emplea mayor tiem-

po en la ejecución de las tareas *reduce*, donde el sistema de archivo HDFS crea múltiples réplicas de cada bloque de datos y los distribuye por todos los nodos del clúster. Finalmente, se ejecutaron un total de 1000 consultas sobre el índice de contenidos creado. Se promediaron los tiempos de respuesta y se alcanzó un tiempo promedio alrededor de los 80 segundos, incluyendo la ejecución de la consulta sobre el índice de contenidos y la construcción del fichero XES como respuesta final.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

6. Conclusiones

> Sólo aquellos que se arriesgan a ir muy lejos, pueden llegar a saber lo lejos que pueden ir
>
> Thomas Stearns Eliot

Para concluir, resumimos las principales contribuciones realizadas en esta tesis. Su fundamentación se basa en un análisis de las estructuras de índices propuestas, las cuales permiten reducir el tiempo de consulta en archivos en formato XES usados en la Minería de Procesos. Esta facilidad permite aplicar la Minería de Procesos en escenarios donde los archivos de trazas son densos o de gran dimensión. Se resume también cómo influye en la eficiencia de los algoritmos de descubrimiento de modelos el manejo de índices, creados a partir de la información almacenada en los registros de eventos. Por último, se mencionan las posibles extensiones con las cuales daremos continuidad al trabajo de investigación realizado.

Universitat d'Alacant
Universidad de Alicante

6.1 Aportaciones principales

A partir del estudio de la estructura lógica del estándar XES, se diseñaron diferentes estrategias de indexación, que permiten consultar la información almacenada en los registros de eventos de una manera más eficiente. De esta forma se facilita el análisis en tiempo real de los algoritmos de descubrimiento de modelos usados en la Minería de Procesos.

Las estrategias se basan en la construcción de índices estructurales e índices de contenidos, ajustados al modelo lógico de datos del estándar XES, lo que permite manejar grandes volúmenes de información contenida en los registros de eventos. Se definió el esquema de un índice estructural,

formado por un resumen de los caminos del árbol, según el esquema de un XES, desde el nodo raíz hasta los nodos hojas. Este índice representa de una forma compacta la información estructural almacenada en un archivo en formato XES.

Se describió cómo es posible implementar este índice estructural utilizando arreglo de sufijos, donde cada elemento del arreglo corresponde con una referencia al sufijo más pequeño del resumen estructural de caminos en orden lexicográfico. El índice permite la ejecución de consultas estructurales con relaciones directas e indirectas. Una consulta estructural con relaciones directas, implica realizar dos búsquedas binarias en el arreglo de sufijos.

Para disminuir los costes temporales de las dos búsquedas binarias es posible manejar un árbol ternario de búsqueda como estructura de datos auxiliar. Este árbol ternario almacena en los nodos internos los sufijos del resumen estructural de caminos y en los nodos hojas se almacena el rango máximo de posiciones en el arreglo de sufijos. Se comprobó que el empleo de una secuencia de posiciones textuales para cada camino, permite que durante la recuperación de los eventos de cada traza se obtengan en orden cronológico las actividades asociadas a los eventos.

Se ha definido una estrategia de compresión para los ficheros XES, apoyada en la repetición de los atributos *string*. Esta se basa en la unificación de todos los nodos del árbol de tipo *string* que describen los atributos de un evento. Para lograrlo se crea un nuevo atributo a la traza en la cual están contenidos los eventos. Se comprobó que el empleo de esta estrategia de compresión disminuye la cantidad de nodos y con ellos los costes temporales en la etapa de análisis sintáctico de un fichero en formato XES.

Se definió el esquema de un índice de contenidos XES, basado en los valores de los clasificadores de eventos. Se presentó una implementación secuencial mediante el empleo de las estructuras de datos sucintas o diccionarios *rank/select*. Presentamos además una estrategia paralela/distribuida para la construcción del índice de contenidos XES, sobre la plataforma Hadoop y su paradigma de programación MapReduce. La estrategia planteada es capaz de construir fragmentos de un documento XES como respuesta a una consulta, sin necesidad de acceder al documento original. Se verificó que

el incremento de los nodos del clúster Hadoop disminuye los costes temporales de construcción del índice de contenidos. Se comprobó que mediante una distribución de los datos, no se ve afectado el tiempo de construcción del índice de contenidos cuando se incrementa el tamaño del archivo XES.

Finalmente, se ha demostrado que el empleo de estrategias de índices que aprovechan la estructura del XES, son más eficientes que los modelos que cargan el XES completamente en memoria y que los modelos que utilizan analizadores *lazy*. Además, un índice de este tipo generado a partir de la estructura lógica del XES, permite que su construcción sea optimizada y personalizada para los diferentes algoritmos de descubrimiento de modelos en la Minería de Procesos (por ejemplo, el algoritmo *Fuzzy Minner*).

6.2 Trabajo pendiente

Sería interesante aplicar el esquema de compresión del XES a otros tipos de atributos definidos para las trazas y los eventos y comprobar los nuevos radios de compresión. Por ejemplo, si contamos con un archivo XES como se muestra en la figura 6.1, donde se repiten los atributos *date* e *int*, valdría la pena comprobar si puede ser aplicada la misma estrategia que para los atributos *string*.

Universitat d'Alacant
Universidad de Alicante

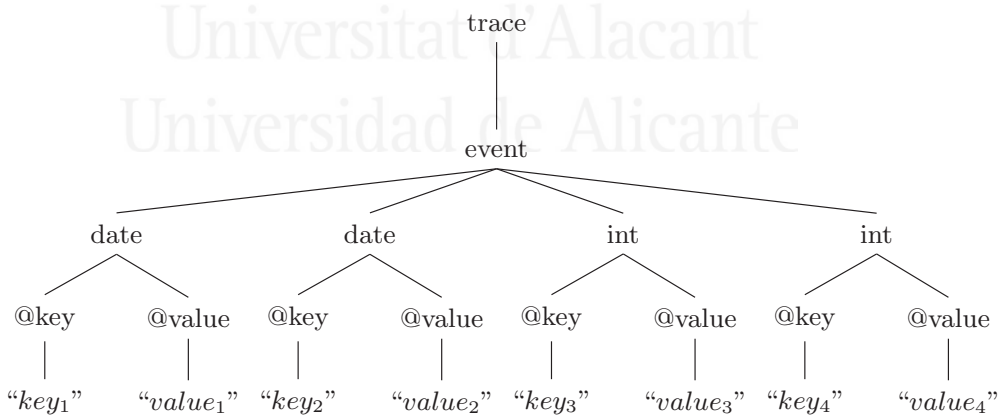


Figura 6.1: XES genérico.

Investigaciones futuras deberán tener en cuenta que es posible mejorar el coste temporal de construcción del arreglo de sufijos y de ejecución de las consultas. Por ejemplo, mediante la inclusión del cálculo del LCP (*Longest Common Prefix* en inglés) como lo reportaron Fischer (2011) y Nong et al. (2011).

Proponemos, además, evaluar el procesamiento y ejecución de las consultas estructurales y de contenido y comparar sus resultados con estrategias recientes de la literatura.

Sería conveniente también diseñar un proceso MapReduce para la construcción de la lista invertida de manera paralela/distribuida. Como se plantea en Vasilenko and Kurapati (2014), en la plataforma Hadoop es posible diseñar un algoritmo que lea cada registro del XES y se delimiten las etiquetas de inicio y fin correspondientes. De esta manera, es posible construir la secuencia que contiene los pares $<C_i, V_i>$ en menor tiempo.

Universitat d'Alacant
Universidad de Alicante

Referencias

Aponte, Y. and Carrasco, R. C. (2012). Indexing structured documents with suffix arrays. In *Computational Science and Its Applications (ICCSA), 2012 12th International Conference on*, pages 43–48. IEEE.

Arturo, O. G., Osvaldo Ulises, L. A., and Damián, P. A. (2015). Generador de registros de eventos para el análisis de procesos en el sistema de información hospitalaria xavia his. In *Convención Salud 2015*.

Assefa, B. G. (2012). Order based labeling scheme for dynamic xml (extensible markup language) query processing.

Bentley, J. and Sedgewick, B. (1998). Ternary search trees. *Dr. Dobb's Journal*, 23(4).

Báez, Y. A., Díaz, A. S., and Such, M. M. (2015). Optimized indexes for data structured retrieval. *International Journal of Advanced Research in Computer Science and Software Engineering*, 5(4):124–129.

Brantner, M., Helmer, S., Kanne, C.-C., and Moerkotte, G. (2005). Full-fledged algebraic xpath processing in natix. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 705–716. IEEE.

Buijs, J. (2010). Mapping data sources to xes in a generic way. *Maters Thesis*.

Calle Jaramillo, M. G. and Parrales Bravo, F. R. (2010). Evaluación de mapreduce, pig y hive, sobre la plataforma hadoop.

Universidad de Alicante

Chaudhri, A., Zicari, R., and Rashid, A. (2003). *XML Data Management: Native XML and XML Enabled DataBase Systems*. Addison-Wesley Longman Publishing Co., Inc.

Chen, S.-Y., Chen, H.-M., and Zeng, W.-C. (2015). Efficient xml data processing based on mapreduce framework. In Park, J. J. J. H., Pan, Y., Kim, C., and Yang, Y., editors, *Future Information Technology - II*, volume 329 of *Lecture Notes in Electrical Engineering*, pages 97–104. Springer Netherlands.

Chen, Z., Gehrke, J., Korn, F., Koudas, N., Shanmugasundaram, J., and Srivastava, D. (2005). Index structures for matching xml twigs using relational query processors. In *Data Engineering Workshops, 2005. 21st International Conference on*, pages 1273–1273.

Chiaramella, Y., Mulhem, P., and Fourel, F. (1996). A model for multimedia information retrieval. Technical report, Citeseer.

Choi, H., Lee, K.-H., Kim, S.-H., Lee, Y.-J., and Moon, B. (2012). Hadoopxml: a suite for parallel processing of massive xml data with multiple twig pattern queries. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2737–2739. ACM.

Chung, C.-W., Min, J.-K., and Shim, K. (2002). Apex: An adaptive path index for xml data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 121–132. ACM.

Cooper, B. F., Sample, N., Franklin, M. J., Hjaltason, G. R., and Shadmon, M. (2001). A fast index for semistructured data. In *VLDB*, volume 1, pages 341–350.

Dean, J. and Ghemawat, S. (2010). Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77.

Universitat d'Alacant
Universidad de Alicante

Deutsch, A., Fernandez, M., and Suciu, D. (1999). Storing semistructured data with stored. In *ACM SIGMOD Record*, volume 28, pages 431–442. ACM.

Dietz, P. F. (1982). Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127. ACM.

Dominguez, A. H. and Yeja, A. H. (2015). Acerca de la aplicación de mapreduce+ hadoop en el tratamiento de big data. *Revista Cubana de Ciencias Informáticas*, 9(3):49–62.

Ferragina, P. and Manzini, G. (2005). Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581.

Fischer, J. (2011). Inducing the lcp-array. In Dehne, F., Iacono, J., and Sack, J.-R., editors, *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer Berlin Heidelberg.

Fuhr, N. and Lalmas, M. (2006). Advances in xml retrieval: The inex initiative. In *Proceedings of the 2006 international workshop on Research issues in digital libraries*, page 16. ACM.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM.

Goldman, R. and Widom, J. (1997). Dataguides: Enabling query formulation and optimization in semistructured databases. In Jarke, M., Carey, M. J., Dittrich, K. R., Lochovsky, F. H., Loucopoulos, P., and Jeusfeld, M. A., editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445. Morgan Kaufmann.

Universidad de Alicante

Gou, G. and Chirkova, R. (2007). Efficiently querying large xml data repositories: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(10):1381–1403.

Grimsmo, N. (2008). Faster path indexes for search in xml data. In *Proceedings of the nineteenth conference on Australasian database-Volume 75*, pages 127–135. Australian Computer Society, Inc.

Günther, C. W. and van der Aalst, W. M. (2006). A generic import framework for process event logs. In *Business Process Management Workshops*, pages 81–92. Springer.

Günther, C. W. and Van Der Aalst, W. M. (2007). Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In *Business Process Management*, pages 328–343. Springer.

Günther, C. W. and Verbeek, E. (2009). Xes standard definition v1. 0.

Hall, D. and Strömbäck, L. (2010). Generation of synthetic xml for evaluation of hybrid xml systems. In *Database Systems for Advanced Applications*, pages 191–202. Springer.

Hammerschmidt, B. C. (2006). *KeyX: Selective Key-Oriented Indexing in Native XML-Databases*, volume 93. IOS Press.

Harding, P. J., Li, Q., and Moon, B. (2003). Xiss/r: Xml indexing and storage system using rdbms. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 1073–1076. VLDB Endowment.

Harold, E. R. (1998). *XML: extensible markup language*. IDG Books Worldwide, Inc.

Hsu, W., Liao, I., and Shih, H. (2012a). A cloud computing implementation of XML indexing method using hadoop. In *Intelligent Information and Database Systems - 4th Asian Conference, ACIIDS 2012, Kaohsiung, Taiwan, March 19-21, 2012, Proceedings, Part III*, pages 256–265.

Hsu, W.-C. and Liao, I.-E. (2013). Cis-x: A compacted indexing scheme for efficient query evaluation of xml documents. *Information Sciences*, 241:195–211.

Hsu, W.-C., Liao, I.-E., and Lu, P.-H. (2012b). Supporting efficient xml query evaluation using double indexes. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 197–202. IEEE.

Hsu, W.-C., Shih, H.-C., and Liao, I.-E. (2014). A scalable xml indexing method using mapreduce. In *Innovative Computing Technology (INTECH), 2014 Fourth International Conference on*, pages 81–86.

Jagadish, H. V., Al-Khalifa, S., Chapman, A., Lakshmanan, L. V., Nierman, A., Paparizos, S., Patel, J. M., Srivastava, D., Wiwatwattana, N., Wu, Y., et al. (2002). Timber: A native xml database. *The International Journal on Very Large Data Bases*, 11(4):274–291.

Lalmas, M. (2009). Term statistics for structured text retrieval. In *Encyclopedia of Database Systems*, pages 3036–3037. Springer.

Lalmas, M. and Baeza-Yates, R. (2009). Structured document retrieval. In *Encyclopedia of Database Systems*, pages 2867–2868. Springer.

Lalmas, M. and Trotman, A. (2009). Xml retrieval. In *Encyclopedia of Database Systems*, pages 3616–3621. Springer US.

Lam, C. (2010). *Hadoop in action*. Manning Publications Co.

Lee, Y. K., Yoo, S.-J., Yoon, K., and Berra, P. B. (1996). Index structures for structured documents. In *Proceedings of the first ACM international conference on Digital libraries*, pages 91–99. ACM.

Li, Q., Moon, B., et al. (2001). Indexing and querying xml data for regular path expressions. In *VLDB*, volume 1, pages 361–370.

Li, W., Yang, J., Sun, G., and Yue, S. (2013). A new sequence-based approach for xml data query. In *Proceedings of 2013 Chinese Intelligent Automation Conference*, pages 661–670. Springer.

Li, Z.-f. and Tao, S.-q. (2012). A xml keyword search algorithm based on mapreduce. *International Journal of Digital Content Technology & its Applications*, 6(17).

Lian, W., Cheung, D. W., and Yiu, S.-M. (2007). Maintenance of maximal frequent itemsets in large databases. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 388–392. ACM.

Lian, W., Mamoulis, N., Cheung, D. W.-l., and Yiu, S.-M. (2005). Indexing useful structural patterns for xml query processing. *Knowledge and Data Engineering, IEEE Transactions on*, 17(7):997–1009.

Liao, I.-E., Hsu, W.-C., and Chen, Y.-L. (2010). An efficient indexing and compressing scheme for xml query processing. In *Networked Digital Technologies*, pages 70–84. Springer.

Liu, J., Ma, Z., and Yan, L. (2013). Efficient labeling scheme for dynamic xml trees. *Information Sciences*, 221:338–354.

Liu, S., Zou, Q., and Chu, W. W. (2004). Configurable indexing and ranking for xml information retrieval. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 88–95. ACM.

Lu, J., Meng, X., and Ling, T. W. (2011). Indexing and querying xml using extended dewey labeling scheme. *Data & Knowledge Engineering*, 70(1):35–59.

Meier, W. (2003). exist: An open source native xml database. In *Web, Web-Services, and Database Systems*, pages 169–183. Springer.

Mlynkova, I. (2008). Standing on the shoulders of ants: towards more efficient xml-to-relational mapping strategies. In *Database and Expert*

Universidad de Alicante

Systems Application, 2008. DEXA'08. 19th International Workshop on, pages 279–283. IEEE.

Mohammad, S. and Martin, P. (2010a). Index structures for xml databases. *Advanced Applications and Structures in XML processing: Label Streams, Semantics Utilization and Data Query Technologies. IGI Global*, pages 98–124.

Mohammad, S. and Martin, P. (2010b). Ltix: a compact level-based tree to index xml databases. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, pages 21–25. ACM.

Nong, G., Zhang, S., and Chan, W. H. (2011). Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484.

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., and Westbury, N. (2004). Ordpaths: insert-friendly xml node labels. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908. ACM.

Qin, L., Yu, J. X., and Ding, B. (2007). Twiglist: make twig pattern matching fast. In *Advances in Databases: Concepts, Systems and Applications*, pages 850–862. Springer.

Rao, P. and Moon, B. (2004). Prix: Indexing and querying xml using prufer sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 288–299. IEEE.

Rizzolo, F. and Mendelzon, A. O. (2001). Indexing xml data with toxin. In *WebDB*, volume 1, pages 49–54.

Schönberger, V. M. and Cukier, K. (2013). *Big data: la revolución de los datos masivos*. Turner.

Schöning, H. (2001). Tamino-a dbms designed for xml. In *icccn*, page 0149. IEEE.

Universitat d'Alacant
Universidad de Alicante

Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E., and Zhang, C. (2002). Storing and querying ordered xml using a relational database system. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215. ACM.

Vanja, K. B. R. J. C., Guy, J. J. K. G. L., Lyle, L. B., Seemann, F. Ö. H. P. N., Van der Linden, T. T. B., Vickery, B., and Zhang, C. (2005). System rx: One part relational, one part xml.

Vasilenko, D. and Kurapati, M. (2014). Efficient processing of xml documents in hadoop map reduce. *International Journal on Computer Science and Engineering*, 6(9):329.

Verbeek, H., Buijs, J. C., Van Dongen, B. F., and Van Der Aalst, W. M. (2011). Xes, xesame, and prom 6. In *Information Systems Evolution*, pages 60–75. Springer.

Wang, H. and Meng, X. (2005). On the sequencing of tree structures for xml indexing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 372–383. IEEE.

Wang, H., Park, S., Fan, W., and Yu, P. S. (2003). Vist: a dynamic index method for querying xml data by tree structures. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 110–121. ACM.

Weigel, F. (2006). Structural summaries as a core technology for efficient xml retrieval.

Weigel, F., Schulz, K. U., and Meuss, H. (2005). The bird numbering scheme for xml and tree databases–deciding and reconstructing tree relations using efficient arithmetic operations. In *Database and XML Technologies*, pages 49–67. Springer.

White, T. (2012). *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".

Universidad de Alicante

Wu, H. (2014). Parallelizing structural joins to process queries over big xml data using mapreduce. In Decker, H., LhotskÃ¡, L., Link, S., Spies, M., and Wagner, R., editors, *Database and Expert Systems Applications*, volume 8645 of *Lecture Notes in Computer Science*, pages 183–190. Springer International Publishing.

Wu, X., Lee, M. L., and Hsu, W. (2004). A prime number labeling scheme for dynamic ordered xml trees. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 66–78. IEEE.

Yan, L. and Liang, Z. (2005). Multiple schema based xml indexing. In *Networking and Mobile Computing*, pages 891–900. Springer.

Yoshikawa, M., Amagasa, T., Shimura, T., and Uemura, S. (2001). Xrel: a path-based approach to storage and retrieval of xml documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141.

Zemmar, I., Benouareth, A., and Souici-Meslati, L. (2011). A survey of indexing techniques in natives xml databases.

Zhou, M., Hu, H., and Zhou, M. (2012). Searching xml data by slca on a mapreduce cluster. In *Universal Communication Symposium (IUCS), 2010 4th International*, pages 84–89. IEEE.

Zuopeng, L., Kongfa, H., Ning, Y., and Yisheng, D. (2007). An efficient index structure for xml based on generalized suffix tree. *Information Systems*, 32(2):283–294.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Reunido el Tribunal que suscribe en el día de la fecha acordó otorgar, por a la Tesis Doctoral de Yosvanys Aponte Báez la calificación de

, de de

El Presidente, El Secretario,

UNIVERSIDAD DE ALICANTE
CEDIP

La presente Tesis de Yosvanys Aponte Báez ha sido registrada con el no. del registro de entrada correspondiente.

, de de

El encargado del registro,

Universitat d'Alacant
Universidad de Alicante