UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

JREPORTS: HERRAMIENTA DE SOFTWARE PARA LA RECUPERACIÓN DE INFORMACIÓN

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA (COMPUTACIÓN)

P R E S E N T A :

GUSTAVO CORRAL GUILLÉ

DIRECTOR DE TESIS: DRA. AMPARO LÓPEZ GAONA

MÉXICO, D.F. 2009

Agradecimientos

A la UNAM, por darme una formación de calidad y permitirme crecer en muchos aspectos de mi vida.

A mis padres que me alientan a continuar y me enseñan a valorar mis logros.

A mi Directora de tesis, Dra. Amparo López Gaona, por su estímulo para seguir creciendo intelectualmente.

A mis sinodales, por su disposición en todo momento para aclarar mis dudas y por sus substanciales sugerencias durante la redacción de la tesis.

A Elvira, por mostrar interés por lo que hago y por su compañía que aligero este largo trabajo.

En fin, con tesis me titulo yo, pero en realidad no lo habría podido llevar a cabo sin el apoyo y las aportaciones de todos estos amigos.

Contenido

Lista de Figuras

Introducción

La idea de desarrollar el sistema aquí presentado surgió dada la necesidad de contar con una plataforma para el almacenamiento y recuperación de los reportes parciales de los proyectos de investigación realizados en el Centro de Ciencias de la Atmósfera, así como la generación del reporte final de un proyecto específico, a partir de un conjunto de reportes parciales. El objetivo principal es facilitar el acceso a los datos, la distribución de información y generación de los reportes.

El planteamiento de que un sistema de recuperación de información era una solución para este problema se originó durante el semestre final de la maestría, una vez que cursé un seminario sobre dicha temática en la Facultad de Ciencias y un curso sobre corpus textuales en el Instituto de Ingeniería. Al contar con todos estos antecedentes, resultó interesante realizar un trabajo en esta área y la problemática arriba descrita resultó una buena aplicación práctica.

Esta tesis es entonces, una referencia sobre recuperación de información y propone desarrollar un sistema que, haciendo uso de ésta, apoye a los investigadores en la generación de reportes finales que conjunten los reportes parciales de los colaboradores a lo largo de la investigación.

JReports busca reducir los costos de desarrollo y ofrecer una independencia tecnológica de software propietario, por lo tanto su desarrollo se realizó usando tecnología libre y estándares abiertos.

El término *recuperación de información* puede ser bastante amplio; sin embargo, des-

de un punto de vista académico, la recuperación de información, podría definirse de la siguiente manera ([1] , [3]):

La recuperación de información (RI) consiste en encontrar material (usualmente documentos) de una naturaleza no estructurada (usualmente texto), de entre grandes colecciones (usualmente almacenadas en computadoras), que satisface una necesidad de información.

En el pasado, y bajo esta definición, la recuperación de información solía ser una actividad limitada a unas pocas personas en áreas como: bibliotecología, derecho y medicina, entre otras. Ahora, es posible hablar de cientos de millones de personas que cotidianamente realizan recuperación de información, ya sea realizando una búsqueda en Internet, buscando un libro en el catálogo digital de la biblioteca, revisando su correo electrónico. La recuperación de información se ha convertido rápidamente en el método más utilizado para el acceso a la información, superando incluso a las consultas a los sistemas de bases de datos tradicionales (donde se pueden presentar situaciones como no poder recuperar una orden de compra si no se tiene el ID de la compra).

La recuperación de información puede también incluir otros tipos de datos y problemas de información más allá de los especificados en la breve definición de arriba. El término *datos no estructurados* se refiere a los datos que no tienen una estructura fácil de procesar para una computadora. Es decir, son el opuesto de los *datos estructurados*, cuyo ejemplo más significativo sería una base de datos relacional, usadas comúnmente en las compañias para mantener inventarios de productos o registros de personal.

En realidad, casi ningún dato carece totalmente de estructura, pues la mayoría de los textos contienen encabezados, párrafos, notas a pie de página, etc. los cuales se representan comúnmente en los documentos mediante un marcado explícito (tal como la forma en que se codifican las páginas web). La RI se suele usar también para resolver problemas con datos *semiestructurados*, los cuales generalmente son irregulares o incompletos y cuya estructura puede cambiar de forma rápida o impredecible, como por ejemplo, en-

contrar los documentos cuyo título contenga la palabra desequilibrio y en el cuerpo del documento se encuentre la palabra ecología.

Actualmente se pueden encontrar sistemas de recuperación de información operando a diferentes escalas, desde los sistemas personales de recuperación de información que en los últimos años han sido integrados en diversos sistemas operativos como el Spotlight de Mac OS X o la Búqueda Instantánea de Windows Vista, pero un ejemplo aún más antiguo se puede encontrar en los clientes de correo electrónico que no únicamente permiten buscar correos, sino que también es posible clasificarlos proporcionando herramientas para que los diferentes correos puedan ser colocados directamente dentro de una carpeta particular. En el otro extremo, se encuentran los motores de búsqueda de la web, los cuales deben ocuparse de resolver consultas sobre billones de documentos almacenados en millones de computadoras, además de lidiar con dificultades como son la profundidad del hipertexto o el tratamiento del contenido de las páginas web con la intención de elevar su posición en los resultados del motor de búsqueda, dada la importancia comercial que ahora tiene la web.

En medio de los dos extremos recién mencionados se encuentran los sistemas de recuperación de información de un dominio específico en un marco empresarial o institucional. En este tipo de sistemas, es necesaria la recuperación de documentos dentro de colecciones (sistemas de archivos o bases de datos) tales como los documentos internos de una corporación, una base de datos de patentes o artículos y reportes de investigación. Aunque en principio el sistema desarrollado para este proyecto pertenece a este último rubro, los conceptos presentados a lo largo de este trabajo, se encuentran presentes en cualquier sistema de recuperación de información.

El objetivo de este proyecto es desarrollar un sistema que, aplicando técnicas de recuperación de información codificada en XML, apoye a los investigadores o coordinadores de algún proyecto para extraer un subconjunto de la colección de reportes almacenados en una base de datos nativa XML, a partir de los cuales se generará el reporte final de dicho proyecto.

Los módulos del sistema facilitarán y agilizarán el proceso de generación de los reportes; así como el almacenamiento persistente y seguro de éstos. Además, facilitará la recuperación y el intercambio de los mismos entre los diversos organismos de la institución.

Resulta relevante también, el hecho de que el sistema aprovechará una de las grandes ventajas que ofrece eXist: un eficiente procesamiento de las consultas basado en índices; convirtiéndose en algo más que un simple repositorio de documentos XML.

Las metas planteadas para el proyecto incluyen:

1. Implementar diferentes métodos y estrategias de extracción de información, que en conjunto resultarán en un sistema efectivo de recuperación a partir del contenido y la estructura de documentos.

2. Proporcionar a los investigadores una interfaz para realizar las consultas a la base de datos con extensiones para ejecutar diversas técnicas de recuperación de la información como son: búsqueda tolerante, búsqueda por zonas o campos; a partir de los metadatos de los documentos y; búsqueda por frase, entre otros.

3. Los resultados arrojados por el sistema servirán como entrada para generar el reporte general, por lo que será necesario combinar fragmentos de estos documentos en un mismo documento XML de manera que éste siga siendo válido de acuerdo a la DTD propuesta y posteriormente pueda ser transformado a algún otro formato.

La tesis se encuentra organizada en tres capítulos, en los cuales se tratan las ideas y los conceptos que en el fondo están presentes en el sistema, de manera que la lectura de la misma proporcionará un panorama general de dichos conceptos:

- En el capítulo 1 se exponen los puntos medulares de la recuperación de información, los algoritmos y estructuras de datos más extendidos en esta disciplina. Se presenta también un bosquejo de lo que es la recuperación de información XML. El capítulo finaliza dando a conocer la API de Lucene con una breve descripción para familiarizar al lector con esta biblioteca.

- El capítulo 2, por su parte, presenta la *anatomía* de las bases de datos nativas XML, en particular una de código abierto llamada eXist. Se enfoca en aspectos relacionados al almacenamiento en estos sistemas de bases de datos, por ejemplo, el indexamiento de la información, así como su lenguaje de consulta, en este caso, XQuery.

- En el capítulo 3 se presenta, de la manera más clara y concisa posible el funcionamiento del sistema desarrollado, llamado JReports, descibiendo cada uno de los módulos que esta aplicación contiene y el desarrollo que se llevó a cabo.

- En la sección de Conclusiones se resumen las principales aportaciones que se han realizado con el desarrollo del presente trabajo de maestría, así como lo novedoso de las herramientas usadas en el mismo. Igualmente se presentan trabajos futuros y aspectos que valen la pena seguir investigando.

- Finalmente, este trabajo incluye cuatro apéndices. El apéndice A es un breve manual del usuario para JReports con indicaciones sobre la instalación y cómo utilizar la herramienta. El apéndice B es el manual de instalación con los pasos a seguir para tener la aplicación funcionando de manera local. El Apéndice C describe un método de evaluación para el sistema basado en los estándares de la Text Retrieval Conference para la evaluación de sistemas de recuperación de información. Aquí aparecen aclaraciones importantes sobre como interpretar mejor los resultados del sistema. Finalmente, el Apéndice D puntualiza el trabajo realizado en este proyecto y menciona el trabajo pendiente para futuras versiones del mismo.

Capítulo 1

Sistemas de Recuperación de Información

"Lo esencial es invisible para los ojos..."

El Principito, Antoine De Saint-Exupéry

1.1. Panorama General

La recuperación de información (RI) es una disciplina de gran utilidad para diversas áreas del conocimiento, pues debido al aumento de la producción de documentos en formato electrónico resulta cada vez más necesaria la implementación de métodos eficientes para la obtención de aquellos documentos que satisfagan una necesidad informativa dada.

Al buscar el origen de la recuperación de información, se tendría que situar en las bibliotecas y centros de documentación. Estos lugares realizan búsquedas bibliográficas de libros y artículos de revistas para satisfacer las necesidades de información de sus usuarios, proporcionándo la información pertinente en el menor tiempo y esfuerzo posible. Para que el usuario de una biblioteca pueda obtener el libro o revista que desea, debe consultar primero una colección de fichas (elaboradas a mano o de forma electrónica) que contienen una serie de campos con información representando al documento en cuestión, como su título, autor, fecha de publicación, etc. También es común incluir algunos términos que dan una idea de su contenido, éstos son conocidos como palabras clave.

La información almacenada en estas fichas recibe el nombre de índice . Los usuarios que consultan el sistema de recuperación para buscar información deben traducir su necesidad informativa en una consulta adecuada al sistema mediante un conjunto de términos que expresen semánticamente esta necesidad.

En este trabajo se trata con recuperación de información automatizada ; sin embargo, pueden existir algunos procedimientos manuales en etapas intermedias de la recuperación, así como en la preparación y almacenamiento del documento, también incluidos en el concepto de recuperación de información. Preparación del documento se refiere a la inserción de una serie de características que permiten su posterior procesamiento y recuperación.

Un repositorio de documentos de este tipo, carecerá de utilidad, si no cuenta con técnicas que faciliten la recuperación de la información que almacena, así como criterios y bases para su organización. Este es, precisamente, el principal objetivo del campo de la recuperación de información que actualmente representa el medio preferido de la gente para el acceso a la misma.

Una parte de las necesidades de recuperación de información para los usuarios de una base de datos documental se puede satisfacer con la implementación de algún lenguaje de recuperación de datos en el texto completo. Sin embargo, en diversas situaciones, estos presentan carencias en aspectos tales como la organización de información, la cuál llega a ser de una magnitud imposible de manejar para los usuarios finales, pues seguramente habrá muchos documentos que satisfagan la consulta en alguna parte de su contenido. Por el contrario, el usuario de un sistema de recuperación de información desea recuperar información en algún lugar específico del documento y no todos los documentos que satisfagan una consulta. Además los resultados deben ser realmente relevantes.

Para poder llevar a cabo este filtrado de los documentos entregados como resultado, el sistema de recuperación de información debe interpretar el contenido de los documen-

tos para poder cuantificar su grado de relevancia respecto a la consulta del usuario. Esta interpretación involucra aspectos sintácticos y semánticos del documento, que permiten, además, que el sistema sea tolerante a errores tipográficos en la consulta, usando los conceptos de *distancia de edición* y *n-gramas*.

1.1.1. Evaluación de un Sistema de RI

La noción de relevancia es precisamente el aspecto fundamental de la recuperación de información. De hecho, el objetivo de un sistema de recuperación de información es recuperar todos los documentos relevantes a la consulta del usuario, minimizando el número de documentos no relevantes [5].

Para evaluar la capacidad del sistema de recuperación y organización de la información para proporcionar documentos relevantes se utiliza una métrica llamada recall o exhaustividad, que mide el volumen de documentos relevantes recuperados respecto al total de relevantes disponibles en la colección, independientemente de que éstos, se recuperen o no. A mayor recall, el sistema devolverá una mayor proporción de documentos relevantes.

$$\text{Exhaustividad} = \frac{\text{Documentos relevantes recuperados}}{\text{Documentos relevantes}}$$

Figura 1.1: Recall o exhaustividad de un Sistema de Recuperación de Información. Tomada de [5]

Claro, que si el sistema devolviera todos los documentos de la colección, obviamente recuperaría también todos los documentos relevantes de la colección, conseguiendo así una exhaustividad de 1. Para evitar este tipo de confusiones, se utiliza también otra métrica, llamada precisión , que mide el nivel de relevantes recuperados respecto al total de documentos recuperados, relevantes o no. Es decir, da una idea de que tan *buenos* son los documentos que devuelve el sistema.

$$\text{Precisión} = \frac{\text{Documentosrelevantesrecuperados}}{\text{Documentosrecuperados}}$$

Figura 1.2: Precisión de un Sistema de Recuperación de Información. Tomada de [5]

Un valor de 1 en la precisión del sistema significa que todos los documentos recuperados fueron relevantes para una consulta, pero no permite saber si todos los documentos relevantes de la colección para esa consulta fueron recuperados. Por otro lado, un valor de 1 para la exhaustividad del sistema significa que todos los documentos relevantes para una consulta fueron recuperados, pero no dice nada sobre cuántos documentos no relevantes fueron también recuperados.

En la siguiente gráfica se observa un ejemplo de la relación de estas dos métricas:

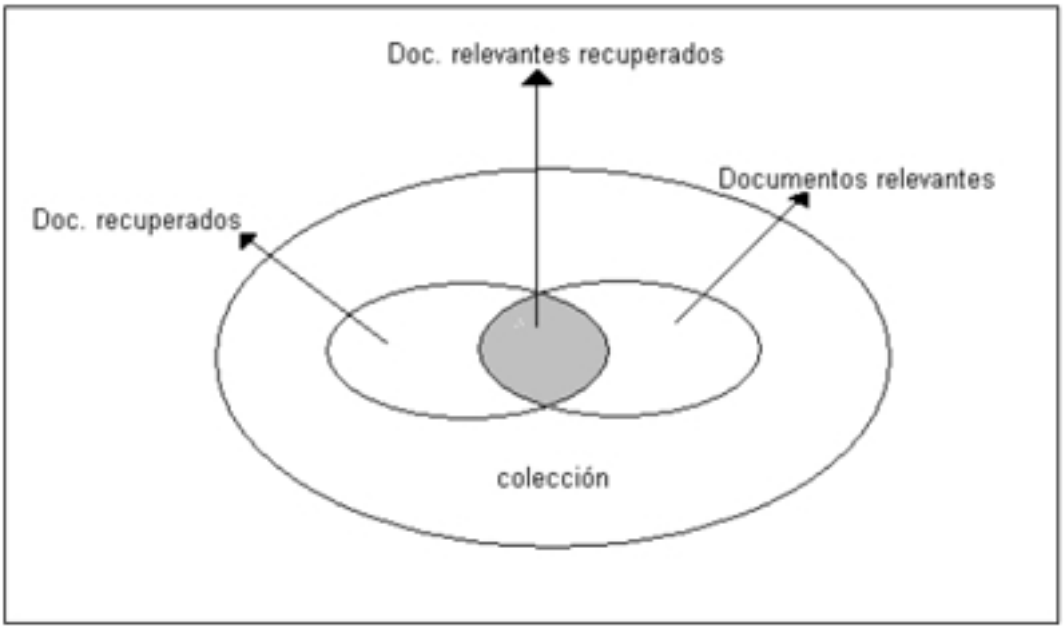

Figura 1.3: Relación Precisión/Exhaustividad. Tomada de [5]

Para los sistemas de recuperación de información que ordenan los documentos de mayor a menor relevancia de acuerdo a la consulta realizada, la exhaustividad y la precisión de cada consulta serán inversamente proporcionales. De esta manera, si se recuperan los *n* documentos de mayor relevancia , se tendrá una alta precisión y una baja exhaustividad

si el valor de *n* es pequeño, así como una baja precisión y alta exhaustividad si la *n* es grande. Se suele buscar un equilibrio entre ambas métricas, aunque en ocasiones hay que decidir a cual se le quiere dar preferencia para darle prioridad a las necesidades del usuario del sistema. Por ejemplo, un médico preferirá mayor recall, mientras que un usuario de *Google* suele preferir mayor precisión.

1.1.2. Modelos Clásicos de Recuperación de Información

Modelo Booleano

Un sistema de RI que no proporcione una medida para cuantificar la relevancia de los documentos obtenidos como resultado de una consulta tendrá una baja exhaustividad por alta precisión. Un sistema de este tipo sería el modelo booleano que se basa en la agrupación de documentos, vistos éstos como conjuntos de términos y en la concepción de las preguntas como expresiones booleanas.

La principal característica del modelo booleano es la consideración de la relevancia como un carácter puramente binario (relevante o no relevante) sin ninguna posibilidad de satisfacción parcial. Por lo tanto, para el sistema será igual de relevante un documento que contenga 1 ó 100 veces las palabras de la consulta, o que cumpla una, dos o todas las cláusulas de un OR. Tampoco considera como relevantes aquellos documentos que presentan una coincidencia parcial como podría ser el cumplir con todas las cláusulas de un AND excepto una. Por eso se puede considerar más como un modelo de recuperación de datos que de información [3].

Modelo Vectorial

Debido a lo antes mencionado, se han desarrollado técnicas de recuperación que permiten un emparejamiento parcial entre los documentos recuperados y la consulta, entre ellos el más exitoso se conoce como modelo vectorial el cual asigna pesos no binarios a los términos índice de la consulta dada y de cada documento. Estos pesos de los términos se usan para calcular el grado de similitud entre cada documento y la consulta del usuario.

Para esto, el modelo vectorial construye una matriz cuyos renglones serían los documentos y las columnas serían los términos índice, de manera que las filas de esta matriz serán vectores de frecuencias de términos para el documento en ese renglón. Pensando de manera similar, también es posible ver a la consulta como un vector expresado en función de la aparición de los *n* términos en la expresión de búsqueda. Así, para recuperar los documentos más relevantes a una consulta se debe calcular la similitud entre el vector pregunta y los vectores que representan a los documentos de la colección.

La forma más conocida para calcular esta similitud es la función del coseno, que consiste en calcular el producto escalar de dos vectores (A y B) y dividirlo entre el producto de las normas de ambos vectores (la raíz cuadrada de la sumatoria de los componentes del vector A multiplicada por la raíz cuadrada de la sumatoria de los componentes del vector B).

$$\cos\theta = \frac{A \cdot B}{||A||||B||}$$

Una vez calculada la similitud entre cada documento de la colección y la consulta, es posible ordenar todos los documentos de la colección recuperados en orden decreciente de su grado de similaridad con la consulta, incorporando así a los resultados aquellos documentos que satisfacen sólo parcialmente los términos de la consulta.

El Esquema tf-idf

La anterior es la idea básica detrás del modelo vectorial, pero el usar simplemente la frecuencia absoluta de aparición de un término en un documento como componente de los vectores presenta un problema crítico: la importancia de un término en función de su distribución puede llegar a ser desmesurada (por ejemplo, una frecuencia de 2 es un 200 % más importante que una frecuencia de 1, cuando la diferencia aritmética es sólo de una unidad). Por este motivo, se introdujo un mecanismo para que la importancia de un término dentro de la colección sea vista en su conjunto, no en un único documento. Tal mecanismo da mayor importancia a la presencia de aquellos términos que aparecen en menos documentos frente a los que aparecen en todos o casi todos, ya que realmente los muy frecuentes discriminan poco o nada a la hora de la representación del contenido de

un documento.

La medida que se propuso para este propósito se conoce como **idf** (frecuencia inversa de documento) y se define como sigue:

$$idf_t = \log \frac{N}{df_t}$$

donde,

N = número total de documentos

df_t = número de documentos que contienen el término *t*

Finalmente, para calcular el factor de peso (w) de un término en un documento se utiliza la combinación de la frecuencia del término (tf), y la frecuencia inversa del documento (idf). Para calcular el valor de la j-ésima componente del vector representando el documento i, se emplea el siguiente esquema de asignación de pesos dado por:

$$w_{j,i} = tf_{j,i} \times idf_j$$

Así, el peso de un término en un documento es alto si éste aparece varias veces en este documento en particular y es bajo si aparece más a menudo en todos los demás documentos.

Estos dos paradigmas forman parte de los llamados modelos clásicos de recuperación de información, los cuales, debido a la popularización de Internet han cobrado un nuevo auge tratando de combinar en un mismo sistema de recuperación las ventajas ofrecidas por cada uno de ellos. Actualmente, se investigan nuevas formas de mejorar la precisión y exhaustividad de los sistemas, pero tratando de contar con mayor participación del usuario para evaluar (de una manera subjetiva) los resultados de los sistemas.

1.2. Índice Invertido, Conceptos y Algoritmos

Antes de continuar es conveniente indicar que es habitual utilizar el término *documento* cuando se habla de recuperación de información para referirse a cualquier unidad sobre

la que se haya decidido implementar el sistema de recuperación de información. Los documentos podrían ser entonces, e-mails, capítulos de libros, documentos completos, etc.

La forma más simple de recuperar los documentos que satisfacen una determinada consulta o necesidad de información quizás sea el realizar una búsqueda lineal a lo largo de cada uno de los documentos dentro de la colección (también conocida como corpus); sin embargo, esta solución presenta varias desventajas como son:

- El alto costo computacional que implicaría procesar grandes colecciones de documentos, si es que esto es posible.
- No es posible la implementación de operadores más flexibles dentro de la consulta. Por ejemplo, un operador `NEAR` que permita especificar a qué distancia debe aparecer un término de otro dentro del mismo documento, para que éste sea relevante a la consulta.
- No es posible determinar de una manera fiable la relevancia de cada documento recuperado.

Es por esto que es necesario encontrar un equilibrio entre el número de términos utilizados para representar el contenido de cada documento y el costo computacional asociado para obtener la salida en un tiempo razonable.

La mayoría de los sistemas de recuperación de información tratan de resolver este problema *indexando* los documentos de la colección en una fase de preprocesamiento. Este concepto se ha convertido en la idea central de la RI y se puede pensar como la transformación de un documento de texto a una representación de texto.

La forma más eficiente de construir un índice se conoce como índice invertido y se trata de una estructura de datos que almacena un mapeo del contenido del documento (términos) al conjunto de documentos en los que cada uno de estos términos aparece (lista de destinos). De esta manera, en lugar de listar los términos para cada documento, el índice invertido lista los documentos para cada término. La Figura 1.4 muestra un

ejemplo de un índice invertido sobre una colección de documentos. También incluye una lista de *stop words* o palabras comunes, concepto que se describe a continuación.

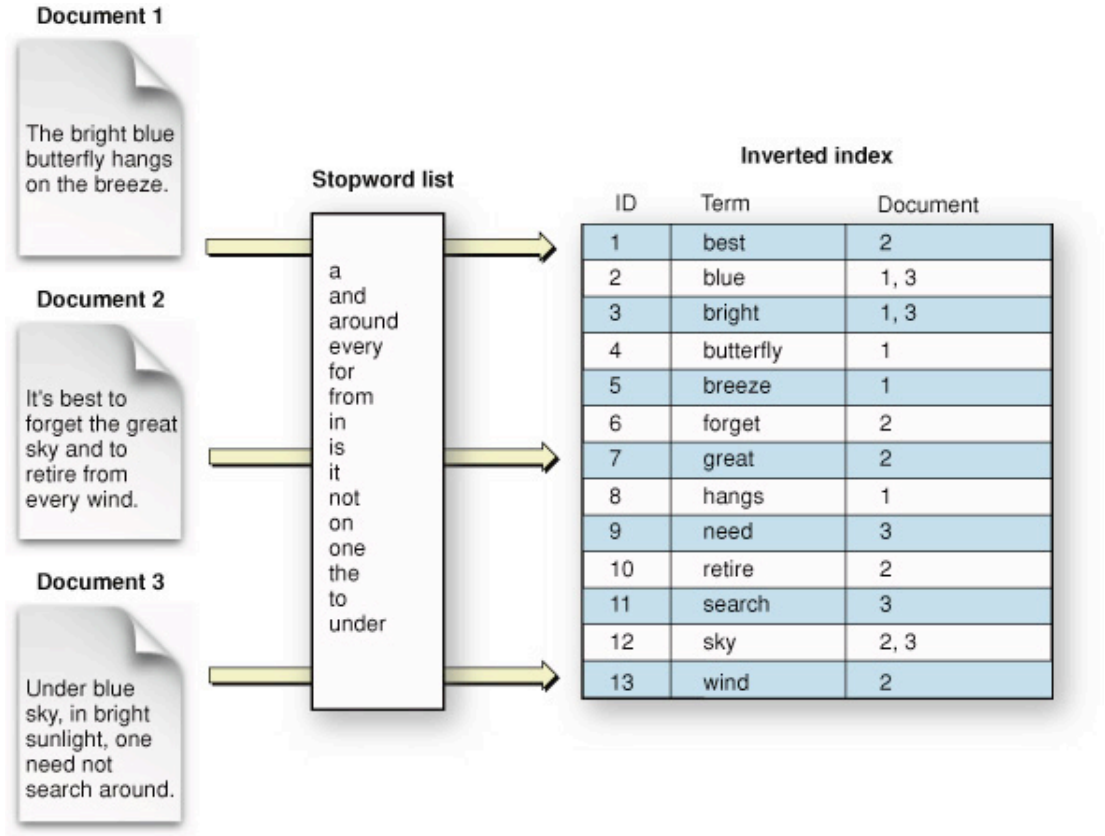

Figura 1.4: Ejemplo de un índice invertido. Tomada de [2]

1.2.1. Determinando los Términos a Indexar

Para generar el índice invertido, es necesario realizar una serie de operaciones con el texto, que pueden incluir la descomposición del texto en pequeñas piezas, llamadas *tokens* (término de difícil traducción al español, por lo que se usará el anglicismo), descartar signos de puntuación, remover acentos, convertir todo el texto a minúsculas, remover palabras demasiado comunes (stop words) o reducir las palabras extraidas a su forma raíz (stemming). A este proceso se le conoce como *tokenización*.

Es importante hacer distinción entre los tokens y los términos a indexar. Un *token* es una instancia de una secuencia de caracteres en un documento particular, agrupados como

una unidad semántica atómica. A la clase de todos los tokens con la misma secuencia de caracteres se le conoce como *tipo*. Un *término* que se indexará y, por lo tanto, formará parte del vocabulario del sistema de recuperación de información. El conjunto de términos a indexar, puede ser entonces completamente distinto al de los tokens. Por ejemplo, para indexar la frase *llámenos para preparar todo para su fiesta*, se encuentran 7 tokens, pero únicamente 6 tipos (hay 2 instancias de *para*). Sin embargo, si durante el proceso de indexamiento se eliminaran las palabras comunes *para*, *todo* y *su* (ver siguiente sección), entonces habrá únicamente 3 términos: *llámenos*, *preparar* y *fiesta*.

El determinar la forma de cómo llevar a cabo esta fase de tokenización no es sencillo y es una decisión específica del lenguaje, pues cada uno tiene características únicas. Otro factor a considerar al momento del análisis para indexar los documentos es el *dominio interdisciplinario* de los textos a analizar, pues de éste dependerá la terminología, acrónimos y abreviaciones. Para ciertos dominios sería importante considerar como términos a las fechas, para otros dominios serían importantes las URL, para otros, como la biología, los términos compuestos, por ejemplo *ácido desoxirribonucléico*.

1.2.2. Palabras Comunes o Stop Words

No todas las palabras de un documento son descriptivas del contenido del mismo, y por lo tanto, no es necesario utilizarlas como términos en el índice. Para cada idioma existen palabras más importantes que otras de acuerdo a su categoría gramatical, pero no es sencillo determinar la importancia de cada una de ellas. También resultan poco útiles para la recuperación aquellas palabras que se repiten con mucha frecuencia en la colección de documentos, pues no sirven para discriminar en relación con una consulta dada. (En [6] se encuentran listas de estas palabras comunes o stop words).

El enfoque más sencillo de filtrar del vocabulario estas palabras comunes como son pronombres, adverbios, artículos, preposiciones y conjunciones es considerarlas vacías de contenido semántico, y por lo tanto, poco útiles para el análisis de la RI, para simplemente eliminarlas antes del procesamiento de la colección. Sin embargo, actualmente muchas herramientas de recuperación de información evitan usar una lista de palabras comunes

para soportar búsquedas por frase, pues muchas de las palabras de este tipo incorporan un nivel semántico adicional al nombre o verbo que acompañan, y por tanto debieran tenerse en cuenta. Así que en realidad ninguna palabra del lenguaje es vacía de contenido.

1.2.3. Algoritmo de Stemming

Los documentos suelen usar diferentes formas de una misma palabra, bien gramaticalmente (afijos flexivos), por ejemplo *libro*, *libros*, bien semánticamente (afijos derivativos), tal como *democracia*, *democrático* y *democratización*. Por esta razón, en muchas ocasiones resulta útil para una consulta con alguna palabra de esta forma regresar además los documentos que contengan diferentes flexiones y derivaciones de dicha palabra.

El anterior es precisamente el objetivo de los llamados algoritmos de *stemming*: reducir las formas de flexión y derivación de una palabra a una forma base común. Se debe hacer notar que se usa el stemming con la finalidad de mejorar el rendimiento de los sistemas de RI, pues gramaticalmente, este algoritmo suele cometer muchos errores.

Este algoritmo será de utilidad en aquellos idiomas en los que las palabras tienden a ser constantes en la forma base (raíz o lexema) y a variar en el final (sufijo), como en el ejemplo de la siguiente tabla:

	-a
	-amos
	-an
abarc	-ar
	-ará
	-arán
	-ó

El algoritmo de stemming más común se conoce como *algoritmo de Porter*. Este algoritmo fue pensado inicialmente para la lengua inglesa; sin embargo, existen ya versiones del mismo para diferentes idiomas, entre ellos el español. El algoritmo completo es demasiado largo y complicado para presentarlo aquí, pero en su manera más general consiste en aplicar una serie de pasos de reducción de palabras, aplicadas de manera secuencial. Dentro de cada paso hay varias convenciones para seleccionar la regla que se aplicará, tal

como seleccionar al más largo entre un grupo de sufijos.

Por ejemplo, el paso 0 dice:

- Seleccionar el más largo entre los siguientes sufijos:
  me se sela selo selas selos la le lo las les los nos
- y borrarlo si es que viene después de una de las siguientes opciones:

  a) iéndo ándo ár ér ír

  b) ando iendo ar er ir

  c) yendo después de **u**

Para ver la descripción completa del algoritmo véase [4] que es la implementación de Snowball del proceso de stemming de Porter.

Obviamente estas eliminaciones *ciegas* de ciertos sufijos, sin tomar en cuenta el contexto de la palabra producen anomalías en el intento de obtención de la raíz, tanto por exceso (hiperlematización), pues si se eliminan muchos caracteres, se pueden agrupar términos que no están relacionados, como por defecto (hipolematización), ya que si se eliminan pocos caracteres, se puede no agrupar términos que están relacionados y, por este motivo, actualmente existen diversas investigaciones para encontrar alternativas a este algoritmo de Porter, destacando los que presentan un enfoque probabilístico.

1.2.4. Recuperación Tolerante

Un problema muy común de la recuperación de información es que los usuarios cometan errores ortográficos en las consultas, por ejemplo, es muy probable que cuando en una consulta aparece el término “efecto inbernadero”, el usuario en realidad busca los documentos que contengan el término “efecto invernadero”. Por tal motivo, es necesario dotar a los sistemas de RI con técnicas de corrección de errores ortográficos en dichas consultas, ya sea para un término a la vez, o para la cadena completa con los términos de la consulta.

A continuación se describen dos de estas técnicas: una basada en el algoritmo llamado *distancia de edición* o *distancia Levenshtein* y otra basada en la idea de los *k-gramas*.

Los algoritmos de corrección ortográfica suelen seguir dos principios básicos en su implementación para encontrar una sugerencia a un posible error ortográfico en la consulta.

1. De las palabras correctamente escritas y que son posibles sugerencias a un supuesto error ortográfico, elegir la más *cercana* a la palabra erronea de la consulta. Este es el concepto de *proximidad* entre dos cadenas.

2. Cuando existen dos palabras correctas ortográficamente que pueden ser una alternativa a una palabra incorrecta, seleccionar la que es más común. La manera más sencilla para determinar esta idea de la más común podría ser el número de ocurrencias de cada término en la colección. Otra alternativa, que es más comunmente utilizada en los motores de búsqueda en la Web es el usar la corrección que es más común entre las consultas de los usuarios.

Un sistema de recuperación de información puede utilizar estos algoritmos de corrección ortográfica para devolver al usuario los documentos recuperados de maneras muy diversas, entre las que destacan:

- Siempre devolver los documentos que contengan la palabra ortográficamente incorrecta, así como los documentos que contengan cualquiera de las palabras correctamente escritas y que resultan una alternativa a la consulta.

- El mismo procedimiento anterior, pero sólo cuando la consulta original regrese menos de un número fijo de documentos.

- Cuando la consulta original devuelve menos de un número fijo de documentos, la interfaz del sistema presenta al usuario una sugerencia ortográfica con el o los términos ortográficamente correctos. Esta alternativa es conocida en el área de recuperación de información como *Did you mean*. En [11] se encuentra una descripción muy completa de este enfoque.

Es importante mencionar que los algoritmos de corrección ortográfica aquí descritos no toman en cuenta el contexto de los términos en la consulta, ya que tratan de corregir uno a uno los términos que en ella aparecen. Existen otras técnicas de corrección sensibles al contexto que no se tratan en este trabajo.

Distancia de edición

La distancia de edición o distancia de Levenshtein se define como el número mínimo de *operaciones de edición* requeridas para transformar una cadena de caracteres en otra, entendiendo por operación la inserción, eliminación o substitución de un carácter [1]. Por ejemplo, la distancia entre las palabras *grande* y *granos* es 2 porque se necesitan al menos dos operaciones de edición para cambiar una en la otra.

1. grande : granoe (substitución de *d* por *o*)
2. granoe : granos (substitución de *e* por *s*)

Usando programación dinámica es posible calcular la distancia de edición entre dos palabras s_1 y s_2 en tiempo $O(|s_1| \times |s_2|)$, donde $|S|$ denota la longitud de la cadena S. El algoritmo consiste en llenar la matriz de $m \times n$, donde m es la longitud de la cadena s_1 y n es la longitud de s_2, de manera que la entrada (i, j) de la matriz contenga la distancia de edición entre la subcadena formada por los primeros i caracteres de s_1 y la subcadena de los primeros j caracteres de s_2.

A continuación se presenta una decripción paso a paso del algoritmo.

1. Sea m la longitud de s_1.
   Sea n la longitud de s_2.
   Si $m = 0$, el algoritmo regresa m.
   Si $n = 0$, el algoritmo regresa n.
   En caso contrario, crear una matriz de m renglones y n columnas.
2. Inicializar el primer renglón con los valores de $0..n$.
   Inicializar la primera columna con los valores de $0..m$.
3. Sean i de $1..m$ y j de $1..n$.

4. Para el siguiente paso, el costo está definido de la siguiente manera:

   - Si $s_1[i]$ es igual a $s_2[j]$, el costo es 0.
   - Si $s_1[i]$ es diferente a $s_2[j]$, el costo es 1.

5. La entrada $d[i, j]$ de la matriz será igual al mínimo de:

   a. d[i-1,j] + 1.

   b. d[i,j-1] + 1.

   c. d[i-1,j-1] + costo.

6. Al finalizar la ejecución del algoritmo, la distancia de edición entre s_1 y s_2 se encuentra en la entrada $d[m, n]$.

Sin embargo, el problema de corrección ortográfica es más complejo, pues consiste en encontrar dentro de un conjunto S de cadenas (en este caso todos los términos de la colección) , aquella cuya distancia de edición con la cadena de la consulta q sea menor y para evitar hacer una búsqueda exhaustiva de todos los términos se usan algunas heurísticas (como se describe en [3]). La Tabla 1.1 muestra un ejemplo de la ejecución del algoritmo.

		C	O	M	A
	0	1	2	3	4
C	1	0	1	1	1
E	2	1	1	1	1
N	3	1	1	1	1
A	4	1	1	1	0

		C	O	M	A
	0	1	2	3	4
C	1	0	1	2	3
E	2	1	1	1	1
N	3	1	1	1	1
A	4	1	1	1	0

		C	O	M	A
	0	1	2	3	4
C	1	0	1	2	3
E	2	1	1	2	3
N	3	1	1	1	1
A	4	1	1	1	0

		C	O	M	A
	0	1	2	3	4
C	1	0	1	2	3
E	2	1	1	2	3
N	3	2	2	2	3
A	4	1	1	1	0

		C	O	M	A
	0	1	2	3	4
C	1	0	1	2	3
E	2	1	1	2	3
N	3	2	2	2	3
A	4	3	3	3	**2**

Tabla 1.1: Ejemplo de la ejecución del algoritmo dinámico para calcular la distancia de edición entre las palabras *cena* y *coma*

N-grama

El otro modelo en el que se suelen basar los algoritmos de corrección de errores se conoce como *n-grama*. Un n-grama es una subsecuencia de n elementos de una secuencia dada. A un n-grama de tamaño dos se le conoce como *bigrama*, a uno de tamaño 3 como *trigrama* y a los de tamaño 4 o más se les denomina *n-grama*. Por ejemplo, la palabra *atmósfera* tiene 7 trigramas: *atm*, *tmo*, *mos*, *osf*, *sfe*, *fer*, *era*.

La idea es la siguiente: la experiencia indica que los errores ortográficos únicamente afectan a unos cuantos de los n-gramas que conforman una palabra, así que es posible reconocer la palabra correcta deseada buscando en el índice de las palabras correctas aquellas que comparten una alta proporción de n-gramas con la palabra incorrecta. Una manera típica de calcular esta medida de similitud es con un índice invertido donde cada n-grama tiene un apuntador a las palabras que lo contienen, donde las palabras son todos los términos que conforman la colección de documentos. Para devolver los términos del

vocabulario que tienen más n-gramas en común con el término de la consulta sólo se debe realizar una búsqueda lineal en este índice de los n-gramas que componen la cadena de la consulta q.

El índice de bigramas (2-gramas) de la Tabla 1.2 muestra un fragmento de la lista de destinos de los cuatro bigramas en la consulta *gaces* (claramente incorrecta). Si se quieren recuperar los términos que contienen al menos dos de esos bigramas basta con ejecutar una búsqueda lineal en el índice. En el ejemplo de la Tabla 1.2 estos términos serían *gases*, *maceta*, *cesto* y *acero*.

ga	→	fugas galleta gases
ac	→	saca acero acapulco matraca maceta
ce	→	acero maceta alce cesto
es	→	escapar cesto gases

Tabla 1.2: Bigramas de la consulta *gaces*

1.2.5. Proceso de Indexación

Así pues, el primer paso para la indexación es analizar el texto del documento para determinar qué términos lo representan mejor y además permiten diferenciar unos respecto de otros, éstos serán los términos índice. Este proceso que tiene como entrada el texto completo del documento y produce como salida un conjunto considerable de términos índice que lo representan, consiste en la siguiente sucesión de pasos:

Paso	**Objetivo**
Análisis léxico del texto	Convertir el texto completo del documento en un conjunto de palabras, y realizar el procesamiento sobre números, signos de puntuación, tratamiento de mayúsculas y/o minúsculas, nombres propios, etc. En este proceso es necesario llevar a cabo un análisis detallado de cómo proceder.

continúa. . .

. . .continuación

Eliminación de palabras comunes (stop words)	Filtrar aquellos términos con valores poco significativos para la recuperación. Es también una buena forma de reducir el tamaño de los archivos del sistema que almacenan el índice, y así mejorar el tiempo de respuesta del sistema.
Aplicación de stemming	Eliminar variaciones morfosintácticas, tanto de flexión como de derivación, y obtener la raíz de la palabra. Como se mencionó anteriormente, consiste en elegir convencionalmente una forma de una palabra para remitir a ella todas las de su misma familia por razones de espacio y tiempo.

Tabla 1.3: Descripción del proceso de indexación para un sistema de RI.

En la Figura 1.5 se ilustran los pasos básicos para la extracción de texto de la Tabla 1.3, comenzando con un documento con cierto formato y produciendo como resultado un índice de texto completo.

1.3. Recuperación de Información XML

Los sistemas de recuperación de información se utilizan para recuperar documentos basándose en su contenido. Tradicionalmente, estos sistemas recuperan información de textos no estructurados, es decir, sin ninguna marca indicando explícitamente la estructura del documento, por lo cual, cuando el usuario busca un conjunto de términos, el sistema le devuelve una lista de documentos sin tener en cuenta aspectos estructurales, ni en la formulación de la consulta ni en la respuesta respectiva del sistema. Sin embargo, muchos documentos son mejor modelados si están conformados con base a una estructura. La búsqueda sobre documentos estructurados se conoce como recuperación estructurada [3].

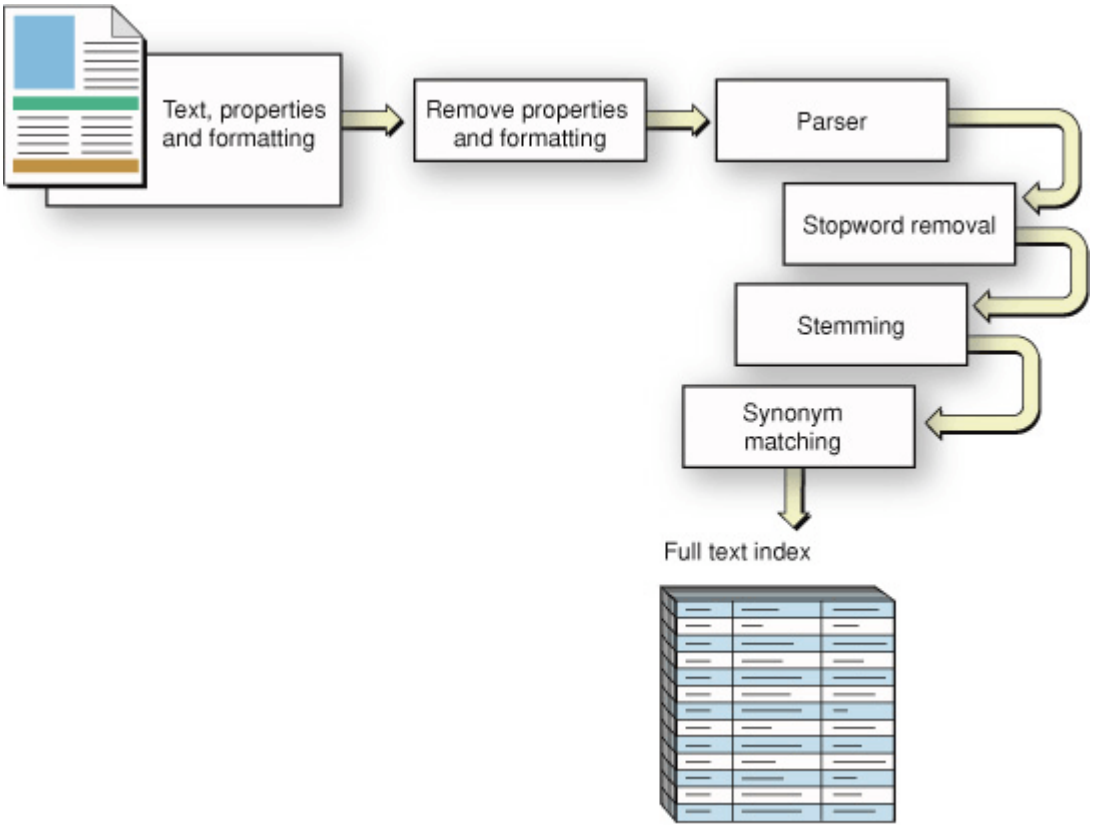

Figura 1.5: Extracción de texto en un índice invertido. Tomada de [2]

1.3.1. Documentos Estructurados

En un documento estructurado, los usuarios pueden buscar, por ejemplo, palabras clave localizadas en el título, en el cuerpo, o en la misma meta-información del documento. Se podría también optar por recuperar únicamente los documentos modificados en ciertas fechas, por ejemplo: las últimas dos semanas, dos meses o dos años.

En el ámbito académico, es muy común el intercambio de información basada en la estructura. Esto se debe a que los reportes solicitados por diversas instituciones se organizan habitualmente de una manera estructurada (título, autor, fecha de publicación, cuerpo del documento, referencias, etc.).

Recientemente se han hecho adaptaciones que permiten aplicar las técnicas de los sistemas de recuperación de información tradicionales a documentos estructurados, donde

las consultas combinan criterios del contenido y criterios estructurales.

El elaborar colecciones de documentos estructurados, resulta cada vez más factible gracias a la utilización de los lenguajes de etiquetado estándar, como puede ser XML. Para estas colecciones, el contenido y la estructura de los documentos se debe indexar y recuperar con el fin de adaptarse a las necesidades de información del usuario.

Las funcionalidades de búsqueda y recuperación se deben implementar usando una estrategia de indexación adecuada, de manera que se utilice el contenido y la estructura para estimar la relevancia de los documentos, permitiendo también al usuario explotar la estructura del documento, pues para la consulta por contenido y estructura es necesario especificar qué se está buscando (contenido) y en dónde se debe encontrar en los documentos (estructura).

Un sistema de recuperación de información para documentos estructurados, en principio, podría no sólo recuperar documentos sino también los elementos estructurales que los componen, por ejemplo, únicamente recuperar el título de los documentos o referencias bibliográficas que satisfagan la consulta. Para esto, es necesario definir y diseñar un índice invertido que considere los elementos estructurales que forman los documentos estructurados como entidades atómicas de manera que se puedan recuperar como respuesta a consultas por contenido o estructura, permitiendo resolver consultas del estilo ¿qué documentos fueron escritos por el autor Rousseau?, ¿qué documentos tienen resumen? o ¿qué documentos fueron modificados en los últimos 6 días?, suponiendo que autor, resumen y fecha de publicación se encuentran representados como elementos en la descripción de la estructura y sintaxis del documento, mediante la DTD (Document Type Definition).

Otra cuestión que se debe tomar en cuenta es la jerarquía de los elementos pues una entidad atómica (elemento estructural) que se encuentre en un nivel jerárquico n podrá estar compuesta a su vez por un conjunto de entidades atómicas (elementos estructurales) presentes en un nivel $n + 1$.

1.3.2. Estructura de XML

Este trabajo se enfoca únicamente en un estándar de codificación de documentos estructurados con XML (Extensible Markup Language), que actualmente es el más usado, aunque lo que aquí se plantea, se aplica para cualquier lenguaje de marcado.

La estructura de un documento XML puede modelarse como un árbol que tiene nodos hoja conteniendo texto y nodos internos que definen los roles de los nodos hoja dentro del documento. En el ejemplo en la Figura 1.6, algunas de las hojas mostradas son Efecto Invernadero, 10 Agosto 2006 y José Berruecos y los nodos internos codifican ya sea la estructura del documento (docTitle, titlePart) o metadatos del documento (docDate y docAuthor).

Este modelo del documento XML es un árbol ordenado y etiquetado. Cada nodo del árbol es un elemento XML representado con una etiqueta que abre y una que cierra. Un elemento puede tener uno o más atributos XML. Por ejemplo, el elemento titlePart de la Figura 1.6 estaría delimitado por las dos etiquetas `<titlePart>` y `</titlePart>` tiene un atributo `type` con el valor *main* y su nodo hijo es un nodo hoja con el valor *Efecto Invernadero.*

Al utilizar un lenguaje de marcado para codificar una colección de documentos, es posible eliminar la necesidad de manejar datos en distintos formatos, utilizando metalenguajes como una DTD o un XML-Schema que especifiquen las restricciones en la estructura de los documentos XML válidos para una aplicación particular, con lo cual también se estandariza su estructura. Lo anterior, permite el intercambio de la información entre los distintos actores de forma segura y precisa.

Por mencionar un ejemplo, es posible escribir una gran cantidad de informes de naturaleza muy diversa; sin embargo, la estructura en general será similar. Los informes pueden contener el nombre del autor, el destinatario del informe, quizás un número de referencia, fecha de creación, título del informe, títulos de secciones, tablas, etc.

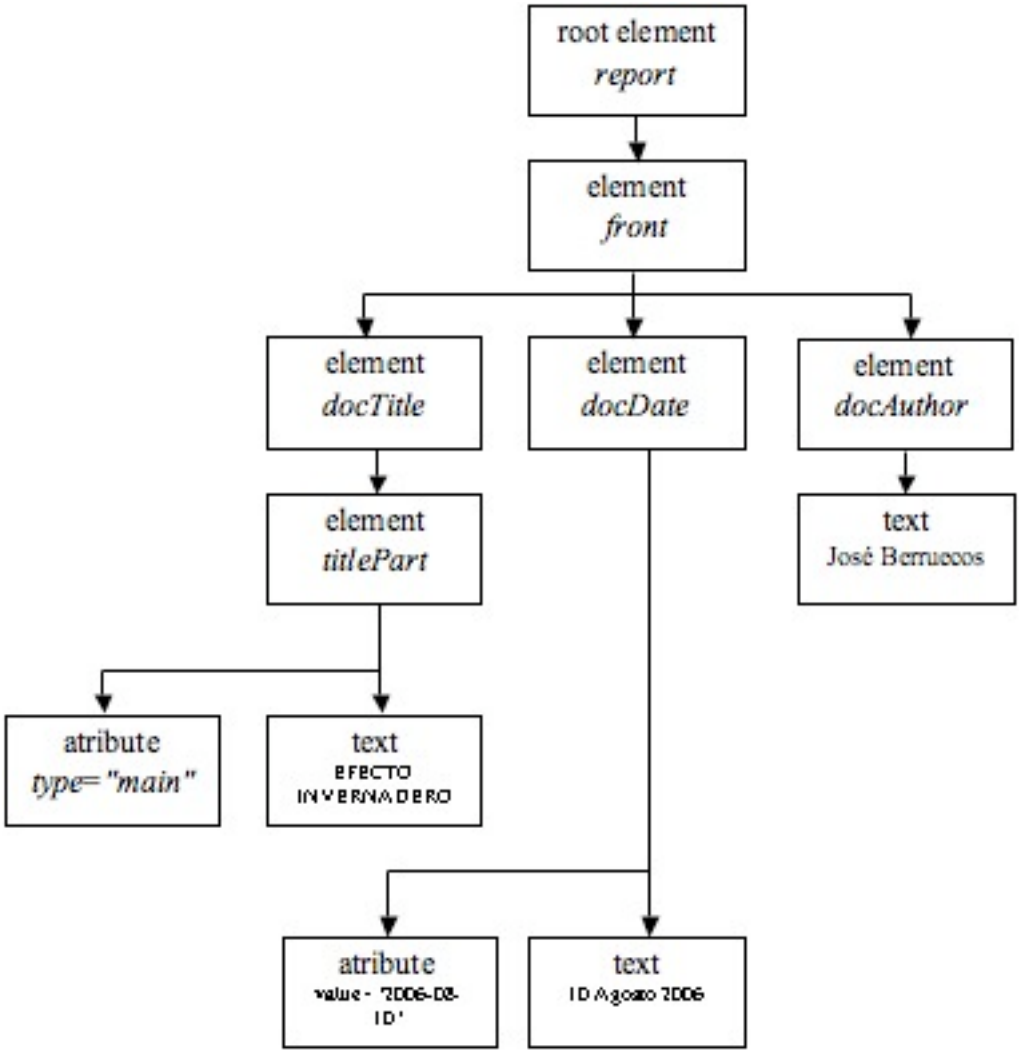

Figura 1.6: Estructura de un documento XML.

1.3.3. XPath

XPath (XML Path Language) es un lenguaje que permite construir expresiones que recorren y procesan un documento XML teniendo en cuenta su estructura jerárquica. XPath permite direccionar partes de un documento XML o acceder a cada una de las partes que lo componen, haciendo una clara distinción entre los elementos (Véase [13], [14]) .

XPath modela un documento XML como un árbol de nodos. Los nodos que componen este árbol son de diferentes tipos, incluyendo nodos elemento, nodos atributo, nodos de espacio de nombres y nodos texto. Algunos de los nodos tienen un nombre consistente

en una parte local y un espacio de nombres (quizá vacío).

Dos conceptos muy importantes de XPath y que se ocuparán en varias ocasiones a lo largo de esta sección son los de:

1. Nodo de contexto: Nodo del árbol representando el documento XML que se toma como punto de partida de la expresión.

2. Nodo actual: Nodo que se obtiene al final de una expresión, o más precisamente cada uno de los nodos de la secuencia devuelta por la expresión.

La entidad básica en XPath es la expresión. Las expresiones son evaluadas y producen un resultado de alguno de los cuatro tipos básicos: conjunto de nodos (una colección desordenada de nodos sin duplicados), booleano (verdadero o falso), numérico (un número en punto flotante), cadena (una secuencia de caracteres).

Un tipo importante de expresión es el camino de localización (path). Un camino de localización selecciona un conjunto de nodos relativo al nodo de contexto. El resultado de evaluar un camino de localización es el conjunto de nodos seleccionados por dicho camino. El conjunto de nodos resultante puede filtrarse mediante expresiones pasadas recursivamente a los caminos de localización, a este tipo de expresiones se les conoce como predicados. Existen dos tipos de caminos de localización: los relativos y los absolutos.

Un camino de localización relativo consiste en una secuencia de uno o más pasos de localización separados por / evaluados de izquierda a derecha. Cada paso selecciona un conjunto de nodos relativos a un nodo de contexto. La secuencia inicial de pasos selecciona un conjunto de nodos relativos a un nodo de contexto. Cada nodo de ese conjunto se usa como nodo de contexto para el siguiente paso. Los distintos conjuntos de nodos identificados por ese paso se unen. El conjunto de nodos identificado por la composición de pasos es dicha unión. Por ejemplo, `child::div1/child::para` selecciona los elementos `para` hijos de los elementos `div1` hijos del nodo de contexto, o, en otras palabras, los elementos `para` nietos que tengan padres `div1`.

Un camino de localización absoluto consiste en el nodo raíz (/) seguido opcionalmente por un camino de localización relativo. Una / por si misma selecciona el nodo raíz del documento que contiene al nodo contextual. Si es seguida por un camino de localización relativo, entonces el camino de localización selecciona el conjunto de nodos que seleccionaría el camino de localización relativo al nodo raíz del documento que contiene al nodo de contexto.

Un paso de localización tiene tres partes:

- Un eje, que especifica la relación jerárquica entre los nodos seleccionados por el paso de localización y el nodo contextual.
- Una prueba de nodo, que especifica el tipo de nodo y el nombre (expandido) de los nodos seleccionados por el paso de localización.
- Cero o más predicados, que usan expresiones arbitrarias para filtrar el conjunto de nodos seleccionado por el paso de localización.

Por ejemplo, en `child::div1[position()=1]` , `child` es el nombre del eje, `div1` es la prueba de nodo y `[position()=1]` es un predicado. El conjunto de nodos seleccionado por el paso de localización es el que resulta de generar un conjunto de nodos inicial a partir del eje y la prueba de nodo, y a continuación filtrar dicho conjunto por cada uno de los predicados sucesivamente.

Ejes

La siguiente tabla muestra los ejes disponibles en XPath:

Eje	**Descripción**
child	Contiene los hijos del nodo de contexto.
descendant	Contiene los descendientes del nodo de contexto, donde descendientes puede ser hijo, nieto, etc.
parent	Contiene al padre del nodo de contexto, si lo hay.

continúa. . .

. . .continuación

ancestor	Contiene los ancestros del nodo de contexto; los ancestros consisten en el padre del nodo de contexto y el padre del padre, etc; así, el eje ancestor siempre incluirá al nodo raíz, salvo que el nodo de contexto sea el nodo raíz.
following-sibling	Hermanos que siguen al nodo de contexto.
preceding-sibling	Hermanos que preceden al nodo de contexto.
following	Todos los nodos que siguen al nodo de contexto.
preceding	Todos los nodos que preceden al nodo de contexto.
attribute	Atributos del nodo de contexto.
namespace	Nodos del espacio de nombres del nodo de contexto.
self	Nodo de contexto.
descendant-or-self	Nodo de contexto con todo y sus descendientes.
ancestor-or-self	Nodo de contexto con todo y sus ancestros.

Tabla 1.4: Descripción de los ejes del lenguaje XPath.

Prueba de Nodo

El tipo principal que aparezca en cada prueba de nodo se determina a partir del eje que se encuentre en esa posición del camino de localización, de manera que si un eje puede contener elementos, entonces el tipo principal de nodo es elemento; en otro caso, será el tipo de los nodos que el eje contiene. Así,

- Para el eje attribute, el tipo de nodo principal es atributo.
- Para el eje namespace, el tipo de nodo principal es espacio de nombres.
- Para los demás ejes, el tipo de nodo principal es elemento.

Una prueba de nodo `Qname` (nombre calificado), especifica una parte local y una URI de espacio de nombres, es verdadera, si y sólo si el tipo de nodo es del tipo principal de nodo y tiene un nombre calificado igual al Qname de la prueba. Por ejem-

plo, `child::sección` selecciona los elementos sección hijos del nodo de contexto y `attribute::número` selecciona el atributo número del nodo de contexto.

Una prueba de nodo `*` es verdadera para cualquier nodo del tipo principal al que contenga el eje. Por ejemplo, `child::*` seleccionará todo elemento hijo del nodo de contexto, y `attribute::*` seleccionará todos los atributos del nodo de contexto.

La prueba de nodo `text()` es verdadera para cualquier nodo de texto. Por ejemplo, `child::text()` seleccionará los nodos de texto hijos del nodo de contexto. Además existen pruebas de nodo como: `comment()` verdadera para cualquier nodo comentario, `processing-instruction()` verdadera para cualquier instrucción de procesamiento, `node()` verdadera para cualquier nodo de cualquier tipo que sea, entre otras.

Predicados

Un predicado filtra un conjunto de nodos con respecto a un eje para producir un nuevo conjunto de nodos.

Los predicados son expresiones que producen un valor booleano que indica si el nodo pasa el filtro o no y se construyen mediante los operadores XPath, entre los que se encuentran:

- instance of
- intersect, except
- union, |
- *, div, idiv, mod
- + , -
- to
- eq, ne, lt, le, qt, =, !=, <. <=, >, >+, is
- and

- or
- for, some, every, if

así como funciones que las implementaciones de XPath deben incluir siempre en la biblioteca de funciones que se usa para evaluar expresiones, como son las funciones de conjuntos de nodos, de cadenas, booleanas y numéricas.

XPath está diseñado principalmente para ser un componente que pueda utilizarse con otras especificaciones. Por ejemplo, XQuery y XPointer están implementados sobre expresiones XPath, así que es fundamental tener un sólido conocimiento de Xpath para sacar el máximo provecho de lo que ofrece XML. Toda la especificación de XPath se encuentra en [14].

1.4. Lucene. Herramienta para Recuperación de Información

En las secciones anteriores se presentó una introducción a la recuperación de información (RI), y en ésta se detalla una de las más flexibles y poderosas bibliotecas existentes para crear aplicaciones de RI. Esta biblioteca se llama Lucene y cuenta con gran éxito y escalabilidad en muchas aplicaciones.

En 1997, Doug Cutting inició el desarrollo de Lucene en Java, lo que le permite ser independiente de la plataforma. En el 2000 Cutting lo liberó como software de código abierto en SourceForge y en 2001 la Apache Software Foundation ofreció adoptar el proyecto Lucene como parte de la familia Jakarta de productos escritos en Java [7].

Actualmente, Lucene cuenta con un equipo activo de desarrolladores y ha sido traducido a gran cantidad de lenguajes de programación, incluyendo C++, C#, Perl, Python y se uiliza en aplicaciones como grupos de discusión, motores de búsqueda en la web a una escala de billones de páginas, buscadores de correo electrónico desarrollados por Microsoft, etc. y cuenta con una gran comunidad de usuarios alrededor del mundo.

1.4.1. Clases Fundamentales

A continuación se desglosan las clases que componen la API de Lucene:

Índice (Index): Almacena estadísticas acerca de los términos de manera que se pueda realizar una búsqueda más eficiente basada en estos términos. Se trata de un índice invertido, que como se mencionó asocia a cada término con una lista de documentos en los que aparece.

Documento (Document): Un documento es una secuencia de campos. En un índice se añaden entonces documentos (objetos de la clase Document).

Campo (Field): Son instancias de la clase Field. Cada campo consta a su vez de términos, formando así tuplas con el nombre del campo y un valor, que es usualmente texto.

Gracias a esta división en campos, es posible tratar con documentos que poseen cierta estructura. Por ejemplo, si se desea indexar un e-mail, se podrá dividir el documento en cuatro campos: el origen, el destino, el título y el contenido del correo respectivamente. Además de su nombre y su valor, en el constructor de la clase Field se puede indicar:

1. Si el valor del campo se almacenará o no en el índice (Field.Store) de forma no invertida. Si se almacena, posteriormente podrá obtenerse al hacer consultas, pero en tal caso, se estará incrementando el tamaño del índice.
2. Si el valor del campo se almacenará en el índice o no, y, en el caso en que se añada, si el contenido se procesará con un Analizador o si se mantiene inalterado (Field.Index). Si el valor del campo no se indexa, no va a ser posible realizar búsquedas sobre él. El caso en el que el valor se indexa pero no se trata por el Analizador es útil cuando se trabaja por ejemplo con paths, URLs o identificadores, sobre los que se quiere tener la posibilidad de realizar búsqueda pero que los valores no sean modificados por el Analizador.

Analizador (Analyzer): Procesa el texto para extraer el conjunto de términos a indexar. Lucene proporciona algunos analizadores ya implementados como por ejemplo el

SimpleAnalyzer que simplemente divide el texto en palabras y convierte todo a minúsculas o el StandardAnalyzer que además elimina stop words (por defecto, las del idioma inglés).

Resultado (Hit): Cada uno de los resultados obtenidos al realizar cierta consulta sobre el índice se representa como un objeto de la clase Hit. A través de estos objetos es posible acceder al documento al que se refiere el resultado y a través del documento a cada uno de los campos almacenados, por ejemplo, para mostrarlos al usuario. Lucene no sólo permite buscar los documentos sino que además calcula el peso de los mismos, ordenándolos de acuerdo a ése valor.

Consulta (Query): Clase abstracta padre para llevar a cabo la consulta. Entre las subclases de Query están: TermQuery, BooleanQuery, PhraseQuery, FilteredQuery y SpanQuery, etc.

1.4.2. Estructura de un Índice de Lucene

Un índice en Lucene se almacena de manera que incremente la eficiencia en las búsquedas, y para esto, el índice se divide en múltiples segmentos (sub-índices completamente independientes). Un índice crecerá cada vez que se añada un nuevo documento, pues se crearán nuevos segmentos.

En la Figura 1.7 se ilustra la estructura de archivos creada para almacenar un índice de Lucene. A continuación se detalla cada uno de los archivos de esa estructura.

Field names (.fnm): Contiene los nombres de todos los campos que aparecen en los documentos en este segmento. Se incluye una marca para indicar si se trata de un campo *indexado* y/o *vectorizado* o ninguno de ellos.

Term dictionary (.tis): Almacena todos los términos (campo-valor) en un segmento. Los términos son ordenados alfabéticamente primero sobre el nombre del campo y después sobre su valor. También almacena la *frecuencia de documentos* que es el número de documentos que contienen este término dentro del segmento. Los campos

que no sean indexados no estarán disponibles como términos y por lo tanto no aparecerán en este archivo.

Term frequencies (.frq): Lista la frecuencia de un término en cada documento. Este es uno de los factores para determinar la relevancia de los resultados, de manera que un documento con una alta frecuencia de un término que aparece en la consulta normalmente será más relevante.

Term positions (.prx): Lista la posición de cada término dentro de un documento. Esta información es usada, por ejemplo, para búsquedas por frase.

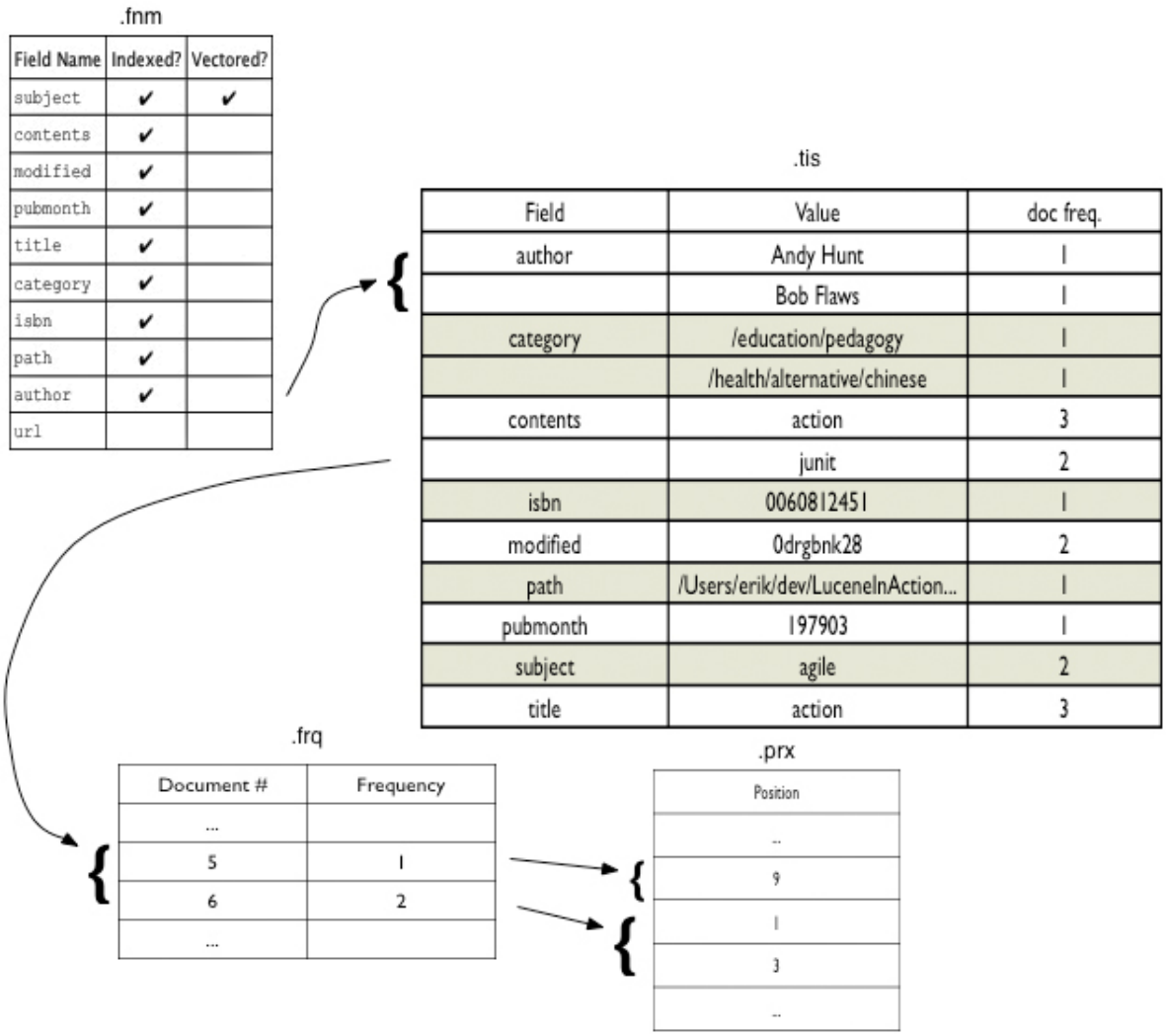

Figura 1.7: Una mirada al interior de un índice con el formato de Lucene. Tomada de [7]

1.4.3. Sintaxis para las Consultas en Lucene

Las consultas en Lucene se componen de términos, divididos en dos clases: términos simples y frases, donde una frase es una secuencia de términos simples encerrados entre comillas y operadores para modificar una consulta. A continuación se presentan ejemplos de cada unos de estos operadores para mostrar su utilidad [9], [8].

Búsqueda por campos: Usada para especificar la parte de la estructura del documento en la que se debe encontrar el término, es necesario anteponer el nombre del campo al término separados por dos puntos. Por ejemplo, para buscar todos los documentos que contengan en el campo *title* el texto "inventario nacional" y en el campo *keywords* el texto agricultura se utilizaría la consulta:

(title:"inventario nacional" AND keywords:agricultura)

Comodines: Lucene proporciona dos tipos de comodines para los términos, para uno o ningún carácter (0 ó 1) y para múltiples caracteres (de 0 a n). Para uno o ningún carácter se usa el símbolo "?"y para múltiples caracteres se usa el símbolo "*". Es importante mencionar, que estos comodines no se pueden utilizar como el primer carácter de la consulta.

c?ima, clima*, c*ima

Búsqueda difusa: Lucene soporta búsquedas difusas basadas en la distancia Levenshtein o algoritmo de distancia de edición. Para realizar una búsqueda difusa, se usa el símbolo de la tilde, "~" al final de una búsqueda de una sola palabra. Por ejemplo, para buscar un término que sea similar a *caza*:

caza~

y se pueden encontrar términos como raza y cazar.

Búsquedas por proximidad: Lucene soporta búsquedas de palabras dentro de una distancia determinada. Para esto se usa el símbolo de la tilde, "~" más el número indicando la distancia al final de una frase. Por ejemplo, para buscar en un documento "factores" y "climáticos" con una distancia de 6 palabras entre una y otra:

"factores climáticos"~6

Búsquedas por rango: Devuelve documentos cuyo valor del campo o campos se encuentre dentro del intervalo especificado por la consulta. Las consultas de rango pueden ser inclusivas o exclusivas en las cotas superior e inferior. La clasificación se hace lexicográficamente.

pubdate:[20040201 TO 20080707], author:{Berruecos TO Ruíz}

Incremento de relevancia de un término: Lucene asigna el nivel de relevancia de los documentos que satisfacen una consulta basado en los términos encontrados. Para incrementar la relevancia de un término se usa el símbolo "^" seguido de un factor numérico de incremento al final del término buscado. Mientras más alto sea este factor, más relevante será el término, permitiendo así controlar la relevancia de los documentos recuperados. El valor predeterminado de este factor es 1, pero este también puede disminuirse (ej. 0.2).

("inventario nacional"^3 "efecto invernadero")

Operadores booleanos: Sirven para combinar varios términos y formar consultas más complejas. Lucene acepta AND, "+", OR, NOT y "−"como operadores booleanos (Nota: los operadores booleanos deben estar COMPLETAMENTE EN MAYÚSCULAS).

- **OR**

  Este es el operador por defecto para conjunciones, por lo que si en la consulta no se incluye un operador booleano entre dos términos, se usará el operador OR. El operador OR enlaza dos términos y un documento satisface la consulta si alguno de los términos se encuentra en dicho documento. Esto es equivalente a una unión con conjuntos.

  Las consultas

  "factores climáticos" clima

  y

  "factores climáticos" OR clima

producirán los mismos resultados.

- **AND**

  El operador AND hace que un documento satisfaga la consulta si los dos términos enlazados están presentes en cualquier lugar del texto. Esto es equivalente a una intersección con conjuntos.

  “factores climáticos” AND “cambio climático”

- **+**

  Este operador exige que el término que aparece después de él exista en algún lugar de un campo del documento.

  +clima inventario

- **NOT**

  El operador NOT excluye los documentos que contengan el término que aparece después de él. Esto es equivalente a una diferencia con conjuntos.

  “efecto invernadero” NOT “cambio climático”

- **-**

  Este operador excluye los documentos que contengan el término que aparece después de él.

  −clima inventario

Agrupamiento: Lucene soporta el uso de paréntesis para agrupar cláusulas y formar sub-consultas. Esto puede resultar muy útil si se quiere controlar la lógica booleana en una consulta y para eliminar cualquier confusión.

(“luis gerardo ruiz” AND “manuel estrada”) OR “omar masera”

Agrupamiento de campos: Lucene soporta el uso de paréntesis para agrupar varias cláusulas en un único campo.

title:(+“inventario nacional” +gases)

Lucene también permite utilizar el caractér de escape “\” para los caracteres especiales que formen parte de la sintaxis de consulta.

1.4.4. Relevancia y Coincidencias

La forma en la que Lucene determina la relevancia de los documentos recuperados para una consulta del usuario es una combinación del modelo de espacio vectorial y el modelo booleano.

El modelo booleano se utiliza primero para recuperar los documentos que satisfacen la consulta del usuario mediante el uso de la lógica booleana especificada en la consulta. Una vez obtenidos dichos documentos se aplica el modelo vectorial a este subconjunto de documentos para poder ordenarlos por su relevancia.

El mecanismo de asignación de pesos de Lucene presenta algunas variantes con respecto al modelo que se explicó en la primera sección (tf-idf), para poder proporcionar soporte a las búsquedas booleanas y difusas. Esto será explicado en breve.

Lucene permite también influenciar los resultados de la búsqueda mediante *incentivos* a varios niveles:

A nivel documento: Cuando se añade el documento al índice se incrementa el valor de éste sobre los demás.

A nivel campo: Cuando se añade el campo al documento se incrementa el valor del campo sobre los demás.

A nivel consulta: En la búsqueda el usuario fija un factor de incremento para ciertos términos en la cláusula de la consulta.

1.4.5. Fórmula de Asignación de Pesos

En la clase `Similarity` de la API de Lucene, que junto con la clase `Query` forman el núcleo del mecanismo de asignación de pesos, se describe la forma en la que dichos pesos son calculados y el siguiente es un resumen [8], [9]:

La fórmula de asignación de pesos recibe como parámetros un documento y la consulta.

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t \in q} (tf(t \in d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d))$$

donde,

- q es una consulta, d un documento, t un término
- $coord(q,d)$ es un factor de peso que cuantifica los términos de la consulta que se encuentran en el documento especificado.
- $queryNorm(q)$ es un factor de normalización para hacer comparables los pesos de diferentes consultas (o incluso de diferentes índices).
- $tf(t \in d)$ se refiere a la $frecuencia\ de\ trmino$ y se define como el número de veces que un término t aparece en el documento d. Documentos con un mayor número de ocurrencias de un término dado recibirán un peso mayor.
- $idf(t)$ se refiere a la frecuencia inversa de documento y se refiere al inverso de $docFreq$ (número de documentos en que aparece el término t). Esto significa que los términos más raros tendrán una contribución mayor en el peso total.
- $t.getBoost()$ es un incentivo en tiempo de búsqueda del término t en la consulta q, ya sea que el usuario la haya especificado en la cláusula de la consulta o esté fija en la aplicación mediante el método $setBoost()$.
- $norm(t,d)$ encapsula algunos incentivos (guardados en tiempo de indexación) y otra información estadística.

1.4.6. Integrando Lucene en las Aplicaciones

Esta sección concluye con la Figura 1.8 que muestra la arquitectura de Lucene como motor de búsqueda dentro de las aplicaciones y quedará más claro cómo a Lucene no le importa cuál sea el origen, el formato ni el idioma de los documentos a indexar.

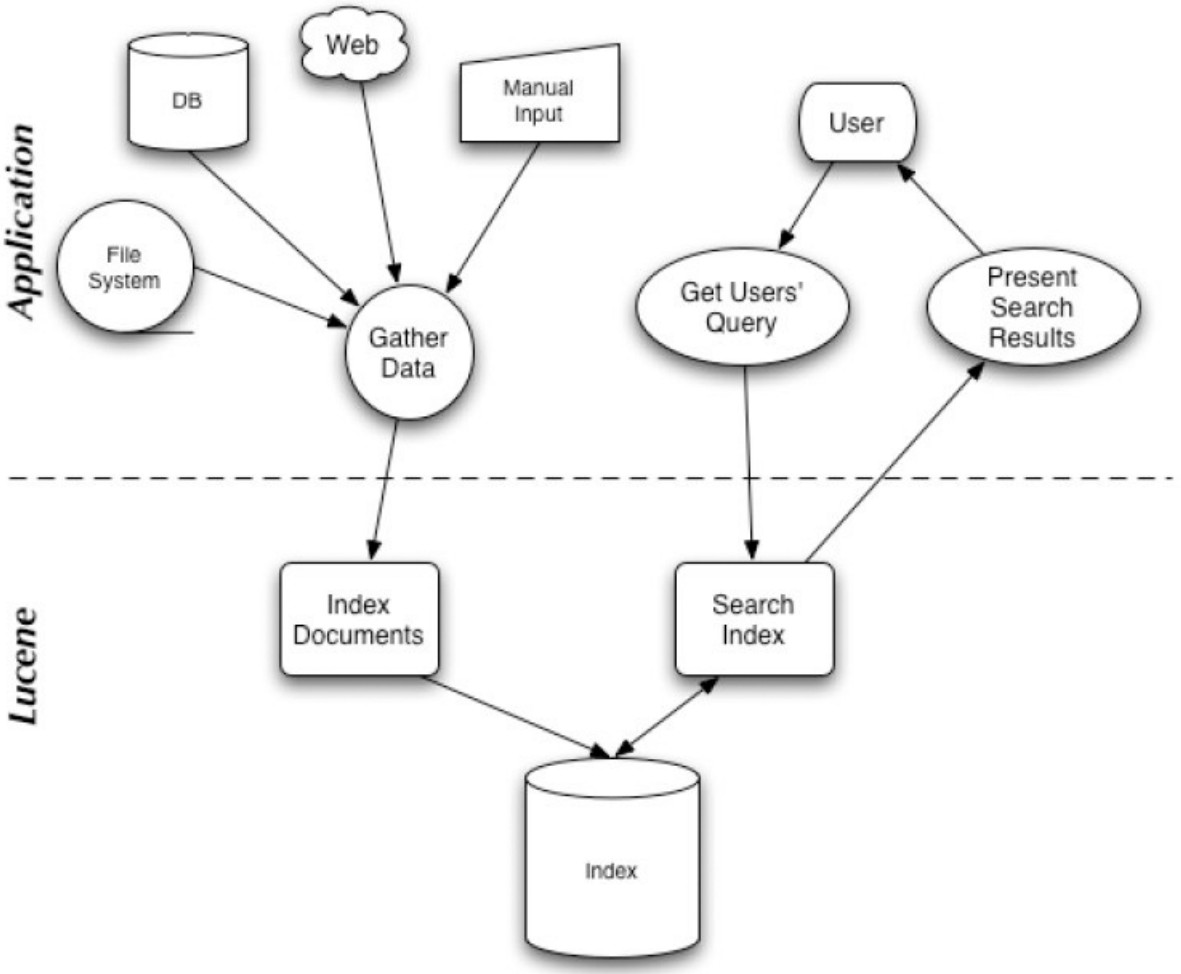

Figura 1.8: Arquitectura de Lucene. Tomada de [7]

Lucene puede usarse para indexar datos almacenados en archivos: páginas web en servidores remotos, documentos almacenados en un sistema de archivos local, archivos de texto plano, documentos de Word, HTML, PDF o XML, o cualquier otro formato que permita extraer texto. De igual manera, con Lucene se pueden indexar datos almacenados en una base de datos.

Así, una vez integrada la aplicación, el usuario podrá realizar búsquedas que Lucene procesará de manera transparente, para posteriormente devolver los resultados a dicha búsqueda independientemente del origen de los datos.

Capítulo 2

Bases de Datos Nativas XML

2.1. Introducción

El término *base de datos nativa XML (NXD, por sus siglas en inglés)* generalmente se refiere a bases de datos diseñadas para manejar contenido XML, que a diferencia de las bases de datos relacionales las cuales representan los datos y sus relaciones con tablas y columnas, basan su modelo de datos en la estructura jerárquica de los documentos y los agrupan en colecciones de documentos. (Para mayores referencias ver [22], [23]).

Este tipo de bases de datos están pensadas para almacenar datos menos predecibles que los que suelen guardarse en una base de datos relacional. Ejemplos de NXD, tanto comerciales como de código abierto son DB XML, eXist, MarkLogic Server, TigerLogic XDMS, XIndice y Monet XML. Todas ellas ofrecen las características de cualquier base de datos, como son almacenamiento de datos, índices, un lenguaje de consulta, inserción, eliminación, respaldo y recuperación de estos datos, además de que algunos proporcionan herramientas adicionales como búsqueda de texto completo, servicios de conversión de documentos o alguna interfaz de usuario.

Los documentos XML según su estructura se pueden agrupar en dos grandes categorías:

Centrados en los datos (data-centric): Usados para el intercambio de datos, ya que tienen una estructura bien definida.

Ejemplos: documentos de facturación típicos, órdenes de compra, etc.

Centrados en el documento (document-centric): tienden a ser más impredecibles en tamaño y contenido que los centrados en los datos los cuales son altamente estructurados, con tipos de datos de tamaño limitado y reglas menos flexibles para campos opcionales y contenido.

Ejemplos: reportes de trabajo, tesis, etc.

Al ver la necesidad de almacenar estos documentos XML, los productos de bases de datos relacionales comenzaron a agregar algún soporte para contenido XML, cuya principal característica consiste en que el contenido XML se fragmenta y almacena en las tablas del modelo relacional. Sin embargo, esto presenta serios problemas como el hecho de que durante la fragmentación se pierde la estructura del modelo DOM (Document Object Model), además de que para cierto contenido este mapeo resulta muy complejo. Estos son precisamente los problemas que resuelve una *NXD*.

2.2. XQuery. Más allá de XPath

Hoy en día, es muy común usar XML en diversos entornos de procesamiento de información y de programación, pero como consecuencia, también la complejidad y el tamaño de los proyectos va en aumento, así como la cantidad de los datos XML a almacenar.

Por este motivo, las herramientas tradicionales para procesar un árbol XML, como son los parsers SAX y DOM, dejan de ser prácticas para manejar estas grandes y complejas colecciones de datos. El problema radica en que para procesar a bajo nivel esta estructura de datos mediante estos parsers DOM y SAX es necesario escribir una gran cantidad de código que lleve a cabo el procesamiento. Para DOM, se tiene todo el árbol en memoria para que el parser pueda recorrerlo, para SAX, se tienen que especificar una serie de acciones a ejecutarse dependiendo de la etiqueta que se encuentre el parser.

Otra alternativa para el tratamiento de esta información XML es el estándar XSLT, cuya finalidad es definir transformaciones sobre estas colecciones XML a diversos formatos, como PDF o XHTML. Pero XSLT tiene también una gran limitante, sólo permite

transformar loa documentos de un formato a otro, pero no es posible utilizarlo como lenguaje de consulta para la localización de información concreta dentro de dicho conjunto de datos.

Un ejemplo claro de esta situación en que estas herramientas dejan de ser útiles se presenta justamente cuando se desea consultar la información en un documento o conjunto de documentos con gran cantidad de datos en XML almacenados en una NXD, o bien cuando se desea trabajar sólo sobre un subconjunto de todos los datos XML para evitar trabajar con la colección completa de datos.

Las NXD fueron diseñadas pensando principalmente en documentos XML centrados en el documento, como ya se mencionó; por tanto, la estructura de la información es de gran complejidad en muchos casos. Si se usara un parser SAX, sería necesario definir acciones más complejas a ejecutar cuando se encuentre una etiqueta, además de que el código será dependiente de la estructura de los datos en XML, con lo cual cualquier cambio en la consulta o en la estructura de la información obliga a modificar el código eliminando la posibilidad de reutilizar el código para consultas similares. El usar como solución un parser DOM añade, a los problemas anteriores el tener que cargar los datos en memoria para recorrer el árbol, lo cual sería imposible para grandes volúmenes de información.

Esta problemática hizo necesario el desarrollo de un lenguaje que permita realizar recorridos y especificar consultas sobre colecciones de datos en XML, a pesar de su complejidad y que devuelva todos aquellos nodos que satisfacen ciertas condiciones. Este lenguaje es XQuery [12], [13], [15].

Para realizar estas consultas, XQuery emplea en una medida muy importante expresiones XPath que fueron explicadas en el capítulo anterior. De hecho, XPath y XQuery comparten el mismo modelo de datos, operadores y funciones.

XQuery es un lenguaje funcional, es decir, sigue el paradigma de programación declarativa, cuya principal característica es que sólo existen valores y expresiones que de-

vuelven nuevos valores a partir de los declarados. Esto significa que, a diferencia de los lenguajes procedurales donde se ejecuta una lista de instrucciones, cada consulta es una expresión que es evaluada y devuelve un resultado, tal como funciona SQL. Estas expresiones se pueden combinar con otras expresiones para crear expresiones más complejas.

En XQuery se proporcionan las herramientas para la extracción y el tratamiento de datos de documentos XML o de cualquier origen que pueda visualizarse como XML, ya sean bases de datos relacionales con conversión de registros a XML, catálogos, servicios web, o incluso de procesadores de texto como OpenOffice y Office que permiten guardar un documento en formato XML. Es decir, XQuery es independiente del origen de los datos.

XQuery 1.0 y XPath 2.0 comparten el mismo modelo de datos y soportan las mismas funciones y operadores, pues el segundo es un subconjunto del primero como se puede ver en la Figura 2.1 de la familia de herramientas para la tratamiento o procesamiento de datos XML, así que si se cuenta con experiencia en XPath no debería de haber mayores problemas para comprender XQuery. XQuery 1.0 es una Recomendación del W3C desde el 23 de enero de 2007.

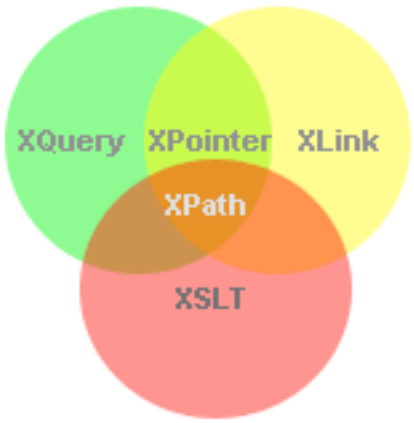

Figura 2.1: Familia del XML para la manipulación de datos. Tomada de [16]

Entre las características definidas por el grupo de trabajo en XQuery del W3C [15], destacan las siguientes:

- XQuery es un lenguaje declarativo. Hay que indicar que se quiere, no la manera de obtenerlo.
- Las expresiones se pueden combinar para generar expresiones más complejas.
- Puede extraer datos de documentos XML existentes y construir nuevos documentos XML.
- Las consultas y los resultados respetan el modelo de datos XML.
- Soporta tipos simples, como enteros y cadenas, y tipos complejos, como un nodo compuesto por varios nodos hijos.
- Es capaz de soportar XML-Schemas (esquemas XML) y DTD, además de poder trabajar sin ninguno de ellos.
- Puede trabajar con independencia de la estructura del documento, esto es, sin necesidad de conocerla.
- Es independiente del protocolo de acceso a la colección de datos. Una consulta en XQuery debe funcionar igual al consultar un archivo local que al consultar un servidor de bases de datos que al consultar un archivo XML en un servidor web.

En XML existe el concepto de jerarquía y orden de los datos que no está presente en el modelo relacional. Es por esto, que en XQuery el orden en el que se encuentran los datos es fundamental, pues no es lo mismo buscar un elemento *title* dentro de un elemento *bibl*, donde cada elemento *bibl* representa una referencia bibliográfica del documento procesado, que buscar el elemento *title* que representa el título del documento.

El lenguaje se basa en el modelo de árbol del documento XML, que se compone de siete tipos distintos de nodo: nodos elemento, nodos atributo, nodos de texto, comentarios, instrucciones de procesamiento, espacios de nombres y nodos documento.

Es importante mencionar que el sistema de tipos que implementa XQuery considera todos los valores como *secuencias*, de manera que un valor simple es en realidad una secuencia de un solo elemento. Los elementos de una secuencia pueden ser valores atómicos

(números enteros, cadenas de texto, valores booleanos, etc.) o nodos. La lista completa de los tipos considera todas las primitivas definidas en XML-Schema.

XQuery se construyó entonces, sobre la base de XPath para la localización de nodos y fragmentos de información en los árboles XML y usa en sus expresiones la sintaxis de este lenguaje para realizar la selección de información e iterar a través del conjunto de datos. Añade además unas expresiones similares a las usadas en SQL, mejor conocidas como expresiones FLWOR. Las expresiones FLWOR toman su nombre de los 5 tipos de sentencias de las que pueden estar compuestas: FOR, LET, WHERE, ORDER BY y RETURN. En la Tabla, 2.1 se describe la función de cada sentencia:

FOR	Crea una secuencia de tuplas mediante expresiones de XPath, esta secuencia iterará y cada tupla será vinculada a una variable.
LET	Vincula una variable a la secuencia resultante de una expresión.
WHERE	Filtra las tuplas eliminando todos los valores que no cumplan la expresión booleana dada.
ORDER BY	Ordena las tuplas según el criterio dado. Puede especificarse si el orden será ascendente o descendente.
RETURN	Se evalúa una vez por cada tupla.

Tabla 2.1: Posibles cláusulas en una consulta XQuery.

FOR y LET sirven para crear las tuplas con las que trabajarán las demás cláusulas de la consulta y pueden usarse las veces que sea necesario en una consulta, incluso dentro de otras cláusulas. Por otro lado, las cláusulas WHERE, ORDER BY y RETURN pueden declararse una sola vez.

Un ejemplo de una expresión XQuery siguiendo la norma FLWOR [12], que lista a los oradores que aparecen en cada uno de los actos de Hamlet de Shakespeare, codificada en un archivo hamlet.xml, es el siguiente:

```
for $act in doc("hamlet.xml")//ACT
  let $speakers := distinct-values($act//SPEAKER)
  return
    <span>
      <h1>{ $act/TITLE/text() }</h1>
      <ul>
      {
        for $speaker in $speakers
        return <li>{ $speaker }</li>
      }
      </ul>
    </span>
```

Ninguna de las cláusulas FLWOR es obligatoria en una consulta XQuery, como puede observarse en el ejemplo e incluso cualquier expresión XPath es una consulta válida y no contiene ninguna de las cláusulas FLWOR.

2.3. eXist

Un sistema de bases de datos nativo XML de código abierto desarrollado completamente en Java es eXist y puede integrarse en cualquier aplicación que maneje XML.

El desarrollo de eXist comienza en el año 2000 por Wolfgang Meier [18], [19] quien continúa trabajando como líder del proyecto actualmente. En septiembre de 2006, eXist publicó la versión 1.1 que incluía notorias mejoras con respecto a las versiones anteriores como la utilización de un nuevo esquema de indexamiento. La versión más recientemente liberada es la 1.2.3.

La base de datos está diseñada pensando principalmente en documentos XML centrados en el documento y para aplicaciones que manejarán colecciones de documentos no demasiado grandes y que no están en constante actualización. eXist proporciona para esto una serie de extensiones a las funciones estándar de XPath para procesar consultas eficientes de texto completo.

El lenguaje de consulta de eXist es XQuery, incluyendo éste a XPath y estas consultas se harán sobre las colecciones. Además eXist implementa un esquema de índices muy eficiente, permitiendo así una rápida identificación de las relaciones estructurales entre los nodos, como son padre-hijo, ancestro-descendiente, hermano anterior-hermano siguiente.

2.3.1. Arquitectura

Como se puede notar, en la definición de una NXD no existe ninguna restricción en cuanto al modelo físico de almacenamiento; sin embargo, eXist se planeó desde sus comienzos como una base de datos con el manejo de XML de forma nativa, tanto en su modelo lógico como en el físico, que para muchas aplicaciones presentaba un mejor rendimiento que si el modelo físico de almacenamiento fuera un manejador relacional.

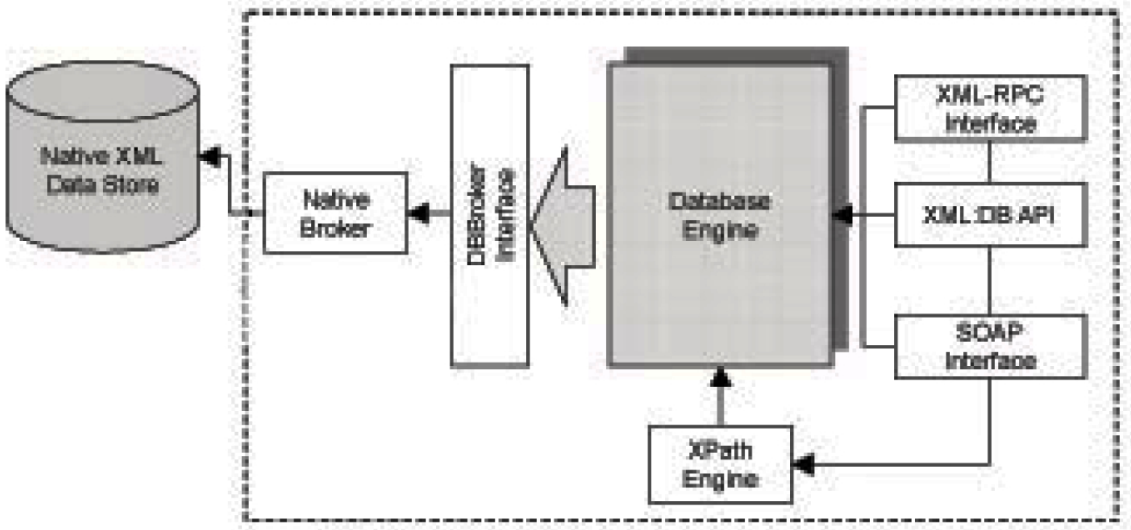

Figura 2.2: Arquitectura del manejador de eXist. Tomada de [18]

En eXist, los documentos son almacenados dentro de una jerarquía de colecciones y la única restricción para poder almacenar un documento es que éste esté bien formado, ya que sólo se validará contra una DTD o un XML-Schema al momento de indexar si se le indica explícitamente en los archivos de configuración.

eXist puede ejecutarse como aplicación *stand-alone* con interfaz HTTP y XML-RPC para acceso remoto, integrado en otras aplicaciones que se comunican con la base de datos

mediante la API XML:DB y dentro de un servidor de aplicaciones como Tomcat con lo cual se tiene acceso a XML-RPC, SOAP y WebDAV.

2.3.2. Índices en eXist

Muchos de los lenguajes de consulta para XML se basan en recorridos *top-down* y *bottom-up* del árbol XML para evaluar las expresiones de XPath. Para grandes colecciones de documentos este enfoque resulta muy ineficiente pues no hay modo de localizar los nodos descendientes desde un principio, lo que implica que en el recorrido se visitará una gran cantidad de nodos que no corresponden al nodo buscado.

Por ejemplo, para la expresión */reports//table/caption* siempre se partirá del nodo *reports* y se visitarán muchísimos nodos que no son el elemento *table*, lo que implica dos comparaciones: si se trata de un nodo elemento y si el nombre del elemento es *table*, de ahí la necesidad de una eficiente estructura de índices.

Los índices deben poder procesar selecciones estructurales (dentro de la estructura jerárquica del documento) y por valor. Para las selecciones por valor funcionan bien los esquemas tradicionales de indexamiento, tal como los árboles B+, pero esto resulta mucho más complejo para selecciones estructurales, para las cuales es necesaria una rápida identificación de las relaciones entre un par de nodos en el mismo documento. Es importante mencionar también, que para poder determinar estas relaciones no debe ser necesario cargar los nodos en memoria, pues ésta es una operación muy costosa.

Un esquema que en buena medida cumple con estos requerimientos se conoce como esquema de numeración que asigna un identificador único a cada nodo en el árbol del documento, por ejemplo, recorriéndolo en level-order o pre-order. Dichos identificadores son usados en el índice como referencia para el nodo actual y no es necesaria ninguna otra información para determinar si un nodo dado puede ser ancestro, descendiente o hermano de otro nodo.

Actualmente, eXist utiliza un algoritmo de numeración llamado numeración dinámica

por niveles (DLN) [20] que se basa en identificadores de longitud variable con lo cual no se necesita imponer un límite máximo en el tamaño de los documentos a indexar y como beneficio adicional permite que las actualizaciones en un documento sean rápidas y sin tener que reindexar.

Los identificadores usados en DLN son una secuencia jerárquica de valores numéricos separados por algún carácter. Así, la raíz tendrá el identificador 1 y todos los nodos debajo de ella tendrán un identificador formado con el identificador de su nodo padre como prefijo y un valor de nivel. Un ejemplo de este algoritmo de numeración se puede ver en la Figura 2.3.

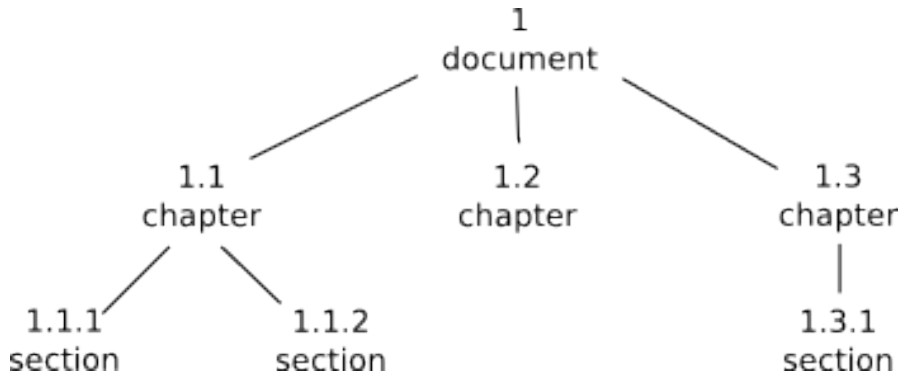

Figura 2.3: Ejemplo del algoritmo de numeración DLN. Tomada de [19]

En este caso, 1 representa el nodo raíz, 1.1 es el primer nodo del segundo nivel, 1.2 el segundo nodo del segundo nivel, etc. Nótese que el punto es el caractér separador.

Usando este esquema, para determinar la relación entre dos nodos, sólo es necesario fijarse en el prefijo de uno de ellos y esto funciona tanto para relaciones ancestro-descendiente, como para hermanos. La dificultad de este algoritmo radica en encontrar una codificación eficiente que:

1. No requiera de demasiados bits para representar un identificador.

2. Permita comparaciones binarias entre identificadores con respecto al orden del documento, pues si el documento tiene elementos anidados a una gran profundidad, los identificadores pueden ser muy largos.

La implementación de codificación elegida por eXist usa unidades de longitud fija (4 bits) para los identificadores de nivel y añade nuevas unidades si el número crece. Un identificador de nivel comienza con una unidad, usando los tres bits más bajos para codificar el número y el bit más alto será una bandera. Si se excede el rango de codificación cubierto por los tres bits (1...7), se añade una nueva unidad y el siguiente bit más alto se asigna a 1. Estos bits de bandera indican entonces el número de unidades que se empleó para codificar el identificador. La Tabla 2.2 ilustra este esquema de codificación, que como se puede ver tiene la ventaja de que el rango de identificadores que se pueden codificar crece exponencialmente por cada nueva unidad. De esta manera, un identificador tan grande como 1.54.70.7.322.1.16 se puede codificar con 48 bits.

No. de unidades	**Cadena de bits**	**Rango del identificador**
1	0XXX	(1...7)
2	10XX XXXX	(8...71)
3	110X XXXX XXXX	(72...583)
4	1110 XXXX XXXX XXXX	(584...4679)
⋮	⋮	⋮

Tabla 2.2: Esquema de codificación para el algoritmo DLN. Tomada de [19]

Además de que con este esquema desaparece la limitante en cuanto al tamaño del documento, permite insertar, eliminar o actualizar nodos de forma más fácil usando la idea de identificadores de subniveles planteada en el mismo algoritmo. Por ejemplo, si se desea insertar un nuevo nodo entre los nodos con identificadores 1.1 y 1.2, se le podría asignar a este nuevo nodo el identificador 1.1/1, donde "/" indica que este nuevo nodo será el hermano siguiente del nodo con identificador 1.1.

2.3.3. Organización de los Datos

Contar con un esquema de índices eficiente tiene como consecuencia un eficiente procesamiento de las consultas. En eXist este esquema se basa en los árboles B+ y está implementado en el núcleo de la capa física de almacenamiento mediante cuatro archivos:

1. *collections.dbx* maneja la jerarquía de colecciones.

2. *dom.dbx* almacena los nodos y les asocia un identificador único. Figura 2.4.

3. *elements.dbx* es el índice de los elementos y sus atributos. Figura 2.5.

4. *words.dbx* guarda un registro de cada palabra en el texto completo y se suele utilizar para consultas de texto completo.

El índice *dom.dbx* es el componente central en la arquitectura de almacenamiento nativo en el cual se almacenan todos los nodos del documento siguiendo el modelo DOM. Se usa un árbol B+ para asociar el identificador de un nodo con la dirección donde se almacena dicho nodo en la sección de datos.

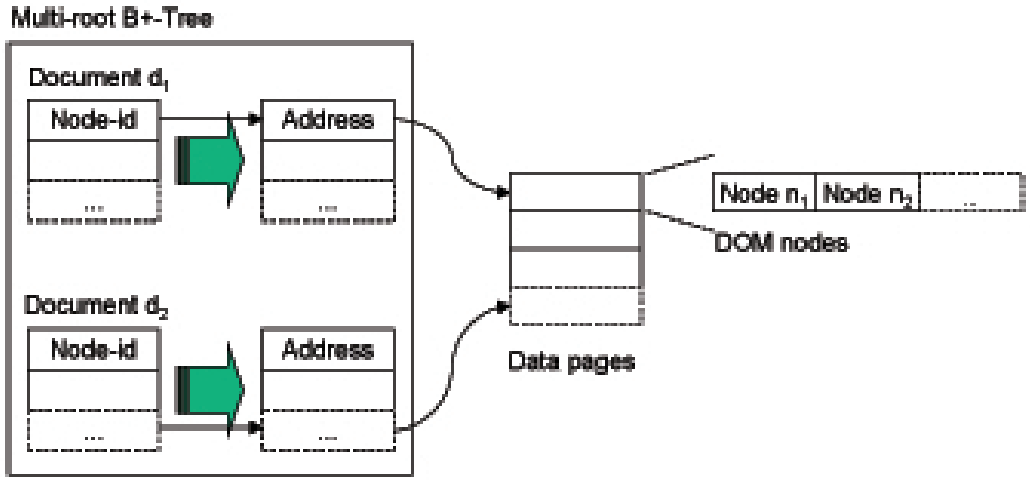

Figura 2.4: Organización en el índice de nodos del archivo dom.dbx. Tomada de [18]

Como se puede observar, no es necesario tener un apuntador para guardar las relaciones entre los nodos, ya que de esto se encarga el algoritmo DLN.

Para ahorrar espacio de almacenamiento en el índice *elements.dbx*, a los nombres de los nodos elemento y atributo se les asigna un identificador único en el índice, donde cada entrada consiste de una llave formada por una pareja <collection-id, name-id> y un arreglo ordenado de identificadores de documentos e identificadores de los nodos elemento y atributo que coinciden con el name-id de la llave.

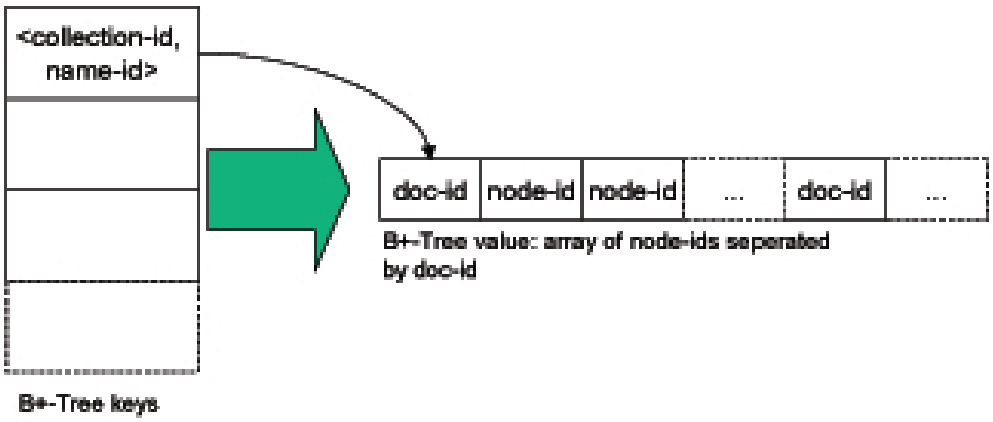

Figura 2.5: Organización en el índice de elementos y atributos del archivo elements.dbx. Tomada de [18]

Finalmente está el archivo *words.dbx* que es un índice invertido de todas las palabras en los documentos. Este índice invertido es similar a la estructura descrita en el capítulo 1 de recuperación de información, sólo que en lugar de almacenar las posiciones de las palabras, se utilizan los identificadores únicos de los nodos como registro de las ocurrencias de las palabras. Si se quisiera desarrollar un sistema de búsqueda con la opción de texto completo este índice será de gran utilidad, aunque no contará con los características que se mencionaron de un verdadero sistema de recuperación de información (eliminación de stop words, algoritmo de stemming, uso de tesauros, asignación de relevancia, etc.).

2.3.4. Procesamiento de Consultas en eXist

eXist usa un algoritmo conocido como *path join* [18] para procesar las consultas eficientemente y para ejecutar este algoritmo se auxilia del esquema de numeración para la asignación de identificadores único (DLN), que como ya se explicó proporciona la información de las relaciones estructurales entre nodos. De esta manera, puede verificarse si existe alguna relación entre cualquier par de nodos. El ejemplo siguiente muestra el funcionamiento del algoritmo *path join*.

Si se ejecutara la siguiente consulta siguiendo el algoritmo *path join*:
/report/titleStmt[title=“inventario nacional de gases de efecto invernadero”]//author.

1. Se buscan en el índice el elemento raíz *report* y los elementos *titleStmt*. En este paso se conservan los identificadores únicos de los elementos titlStmt que tengan una relación padre-hijo con algún elemento report.

2. Se buscan en el índice los elementos *title* y de la secuencia devuelta se seleccionan los identificadores de los elementos cuyo valor de texto sea igual a "inventario nacional de gases de efecto invernadero".

3. Ejecutar el algoritmo *path join* sobre los identificadores de los elementos *titleStmt* obtenidos en el paso 1 y los identificadores de los elementos *title* del paso 2. El resultado será una secuencia de identificadores con una relación ancestro-descendiente con los identificadores del elemento *title*.

4. Se buscan en el índice los elementos *author* y se ejecuta el algoritmo sobre los identificadores del conjunto devuelto como resultado y los identificadores del elemento *titleStmt* obtenidos en el paso 3. Los nodos deben tener una relación ancestro-descendiente. Todos los identificadores obtenidos representan nodos que satisfacen la consulta.

Para especificar consultas en eXist, en muchas ocasiones es preferible utilizar predicados de XPath que una expresión FLWOR con una cláusula WHERE equivalente, ya que la expresión FOR fuerza al motor de búsqueda a iterar paso a paso sobre la secuencia de entrada. Cuando eXist optimiza una expresión FLWOR internamente trata de procesar la cláusula WHERE como un predicado de XPath equivalente; sin embargo, para muchas expresiones esto puede ser muy complicado, así que un predicado de XPath usualmente garantiza una mejor eficiencia.

La realidad es que muchos usuarios que comienzan con XQuery tienden a utilizar expresiones FLWOR por su similitud con expresiones de SQL, cuando un predicado de XPath produciría los mismos resultados además de hacer más clara la cláusula de la consulta.

Por ejemplo, la expresión FLWOR

```
for $p in /people/person where $p/age > 30 return $p/name
```

es equivalente al sencillo predicado de XPath

```
/people/person[age > 30]/name
```

Por último, en eXist también es preferible construir consultas que impliquen calcular relaciones de abajo hacia arriba (bottom-up) en lugar de relaciones de arriba hacia abajo (top-down), ya que en muchas ocaciones resulta más económico subir hacia los ancestros de un nodo dado que recorrer el árbol partiendo de ese nodo y teniendo que visitar cada uno de los descendientes hasta encontrar el nodo buscado.

La consulta

```
for $section in collection("/db/articles")//section
for $match in $section//pcontains(., 'XML')
return
    <match>
        <section>{$section/title/text()}</section>
        {$match}
    </match>
```

es correcta con un enfoque top-down, pero ésta obliga al motor de búsqueda a evaluar la *subconsulta* $section//p[contains(., 'XML')] una vez por cada sección en cada documento de la colección. Esta consulta será optimizada por eXist, pero siempre será mejor reformular la expresión, en este ejemplo en particular encontrando los títulos de las secciones con el eje *ancestor*.

```
for $match in collection("/db/articles")//section
                        //pcontains(., 'XML')
return
    <match>
        <section>{$match/ancestor::title/text()}</section>
        {$match}
    </match>
```

Capítulo 3

JReports

3.1. Antecedentes

Una de las actividades que resultan más tediosas para el personal académico que labora en las diferentes dependencias de la Universidad Nacional Autónoma de México ha sido, desde hace muchos años, la elaboración de reportes de los proyectos de investigación en los que colabora.

La forma en la que se elabora el reporte definitivo de un proyecto determinado puede variar de acuerdo a la dependencia de la que se trate, pero en general el coordinador de dicho proyecto debe conjuntar los reportes parciales que le presentan sus colaboradores a lo largo de la investigación. Esta tarea puede llegar a consumir bastante tiempo, debido principalmente a que cada colaborador puede escribir su reporte en un formato que le parece más conveniente, utilizando también diferentes medios para la captura de la información.

Esta diversidad de medios y métodos empleados para obtener información provoca que ésta no pueda compartirse fácilmente y de forma estándar al interior de la dependencia, mucho menos entre las diferentes dependencias de la UNAM y hacia otras organizaciones, además de que no es posible formular resultados confiables referentes a la producción de cada dependencia en la universidad.

Para poder satisfacer estas necesidades, se implementó un sistema (que hace uso de tecnologías novedosas) para llevar a cabo una efectiva recuperación de información XML, tanto en contenido, como en estructura, restringiendo así la unidad de documento de una consulta (título, capítulo, bibliografía, etc.), aunque no es posible recuperar sólo ese fragmento de documento, ya que para los fines que se persiguen es necesario recuperar el documento completo.

El enfoque del sistema es una especie de *híbrido* de recuperación XML que combina características de un sistema de recuperación de información de texto completo, como es Lucene con las ventajas de la recuperación específica XML presentes en una base de datos nativa de XML, como es eXist.

Esta combinación presenta diversos beneficios para un sistema con requerimientos como el de este proyecto, entre los cuales destacan la eficiente implementación de una estructura de índice invertido en Lucene, el eficiente procesamiento de consultas basado en índices de eXist, así como el soporte de XQuery como su lenguaje de consultas. De los dos primeros puntos ya se habló a detalle en los capítulos anteriores, sin embargo el tercer punto también resultó ser de suma importancia para la implementación del sistema JReports facilitando en buena medida el desarrollo de la aplicación Web.

JReports puede servir incluso como un ejemplo de una aplicación que requiere procesar documentos XML complejos, pues XQuery permite descargar la lógica del negocio del lado del servidor, además de que las funciones de XQuery simplifican la codificación.

En el caso específico de sistemas como JReports que deben integrar múltiples campos de búsqueda, usando una entrada para cada criterio y ampliando así las opciones del usuario, las funciones de XQuery combinadas con expresiones XPath proporcionan un mecanismo de búsqueda poderoso para implementar tales requerimientos.

Con XQuery es posible además generar aplicaciones dinámicas mezclando fragmentos de código XML o HTML con programación, tal como PHP y la tecnología JSP de

Java. Sin embargo, XQuery es un lenguaje orientado a expresiones, con lo cual el código no termina siendo un caos entre la funcionalidad del programa y sentencias para generar la salida. Más importante aún, es posible tratar un fragmento de documento como entrada para una función, la cual posteriormente puede ser procesada por otra función y así cuanto sea necesario.

3.1.1. Necesidad de un Estándar

Se mencionó al principio de este trabajo que el objetivo del proyecto es construir un sistema orientado a facilitar el acceso y gestión de los datos, distribución de información y generación de reportes con una estructura definida apegada a un estándar que permita resolver tales necesidades.

Actualmente, cada colaborador en una investigación suele utilizar un medio diferente para elaborar el reporte de su parte en la investigación, por ejemplo, puede haber diferencias radicales entre el procesador de textos, la plataforma o la estructura misma de cada reporte parcial.

Cualquier procesador de textos dispone de un formato específico propiedad de su fabricante que puede ocasionar problemas en el caso de intercambio de datos a través de distintas plataformas y si la estructura de cada reporte parcial es diferente, entonces la generación del reporte final será una labor muy complicada para el coordinador del proyecto.

Entonces, si se desea intercambiar información, en forma segura y precisa, sin tener problemas con el formato, es necesario definir un estándar para que todos los reportes parciales y el reporte final tengan la misma estructura sin que ésta sea dependiente de la plataforma.

Los datos que se deben modelar y estandarizar son, entre otros:

- **Datos generales del documento:** título principal, título secundario, fecha de elaboración, autor del documento, destinatario del documento

- **Resumen**
- **Palabras clave**
- **Listas:** numeradas, ordenadas, no ordenadas
- **Prefacio**
- **Introducción**
- **Cuerpo del reporte:** dividido en secciones
- **Referencias bibliográficas**
- **Apéndices**
- **Meta**
- **Información del documento:** Con el formato MARC, campo, valor

3.2. Estructura de los Documentos

3.2.1. Estándar propuesto

El sistema basa su procesamiento de los documentos en el paradigma conocido como codificación estructural. Este tipo de codificación consiste en agregar a los documentos, claves o etiquetas que indican aspectos de su estructura, más que de su formato.

Entre los beneficios principales que ofrece este tipo de codificación se pueden mencionar:

- Una mejor separación entre el contenido y la presentación de los documentos.
- El proceso de análisis del contenido resulta más fácil de automatizar.

XML (eXtensible Markup Language) es un lenguaje de marcado de propósito general cuyo objetivo es facilitar la tarea de compartir datos entre diferentes sistemas de información, principalmente vía Internet.

Permite a los usuarios definir sus propias marcas a la vez que modelar la estructura de los documentos, de manera que se puede afirmar que las marcas denotan semántica al documento. XML resulta entonces una opción lógica para la codificación de los reportes.

La implementación de XML con la que serán marcados los documentos está basada en el modelo definido por TEI (Text Encoding Iniciative) [24], [25], que ofrece enormes posibilidades a la hora de clasificar y codificar los elementos de cualquier texto para representarlo de forma electrónica con ayuda de las directivas oficiales que ofrece TEI. Se trata de una DTD reducida que es en realidad, un subconjunto de los elementos incluidos en TEI Lite.

El siguiente es un ejemplo de cómo quedaría la codificación del encabezado de un reporte siguiendo el estándar propuesto.

```
<titlePage>
      <docTitle>
                <titlePart type="main">
   Inventario nacional de gases de efecto invernadero: 94-95
                </titlePart>
               <titlePart type="sub">
               Parte 5: Agricultura
               </titlePart>
     </docTitle>
     <docDate>18/08/2000</docDate>
     <from>Preparado por:
        <docAuthor>Luis Gerardo Ruiz Suarez</docAuthor>
        <docAuthor>Manuel Estrada</docAuthor>
     </from>
     <for>Preparado para:
       <docOwner>Instituto Nacional de Ecología</docOwner>
     </for>
</titlePage>
```

Una vez marcados los documentos en XML, puede aprovecharse la información del esquema de codificación y el motor de búsqueda podrá ofrecer al usuario la opción de realizar consultas solamente en ciertas partes de la estructura XML. Por ejemplo, en un documento que incluye una etiqueta `<docAuthor>` para indicar el autor del documento. El usuario tendrá la oportunidad de realizar la búsqueda de los documentos publicados por un autor en particular, en este caso en las etiquetas `<docAuthor>`.

3.2.2. Definición de la DTD

Una DTD (Document Type Definition) es una colección de reglas usadas con el propósito de establecer la estructura y sintaxis de un documento XML. Su función básica es la descripción del formato de datos, para usar un formato común y mantener la consistencia entre todos los documentos que utilicen la misma DTD. De esta forma, dichos documentos, pueden ser validados, conocer la estructura de los elementos y la descripción de los datos que trae consigo cada documento, y pueden además compartir la misma descripción y forma de validación dentro de un grupo de trabajo que usa el mismo tipo de información.

La DTD realiza entonces, las siguientes tareas:

- Definir los elementos (nombres de etiquetas) que pueden aparecer en el documento.
- Definir las relaciones entre los distintos elementos.
- Suministrar información adicional que puede ser incluida en el documento: atributos, entidades y dominios.
- Aportar comentarios e instrucciones para su procesamiento.

El punto de partida para la definición de la DTD de los reportes será la DTD y las directrices del Consorcio TEI, organización sin ánimo de lucro que sigue una filosofía similar a la del W3C (World Wide Web Consortium), para sostener y desarrollar TEI (Text Encoding Initiative), un estándar internacional e interdisciplinar de facto que ayuda a bibliotecas, museos, editores, eruditos e investigadores para la representación de textos en forma digital. TEI nació como aplicación de SGML (Standard Generalized Markup Language), pero se migró su DTD al más reciente XML, sucesor de SGML.

La DTD del TEI permite elegir codificar tanto o tan poco como se quiera de forma modular, es decir, el diseñador de la DTD puede elegir cómo combinar las marcas propuestas por el TEI.

La estructura básica consta de:

- **Encabezado:** de qué trata el texto, creador, etc.

- **Material del front:** portadas, cartas a modo de prólogo, etc.
- **Cuerpo:** texto generalmente es un conjunto de párrafos agrupados en capítulos, secciones, subsecciones, etc.

Cuenta además con elementos para indicar tipo de fuente, referencias, fechas y horas, tablas, gráficas e imágenes, entre otros. Abajo se presenta un fragmento de la DTD usada en JReports.

```
<!ELEMENT TEI2 ( teiHeader, text )>
<!ELEMENT teiHeader ( fileDesc )>
<!ELEMENT fileDesc ( titleStmt, publicationStmt )>
<!ELEMENT titleStmt ( title | part )*>
<!ELEMENT title ( #PCDATA )>
<!ELEMENT publicationStmt ( author+ )>
<!ELEMENT p ( #PCDATA | abbr | date | num | note | ref)*>
<!ELEMENT text ( front, body, back )>
<!ELEMENT front ( titlePage )>
<!ELEMENT body ( div1 | keywords | list | p )*>
<!ELEMENT div1 ( #PCDATA | def | div2 | head | listBibl | listInfo
        | note | p | table | term | list | formula | figure)*>
<!ATTLIST div1 n NMTOKEN #IMPLIED>
<!ATTLIST div1 name CDATA #IMPLIED>
<!ATTLIST div1 type NMTOKEN #REQUIRED>
<!ELEMENT author ( #PCDATA )>
<!ELEMENT back ( div1* )>
<!ELEMENT bibl ( #PCDATA | author | biblScope | date | pubPlace
        | title | publisher | url)* >
```

Existe otro metalenguaje para especificar la estructura de documentos en XML, los XML-Schema (esquemas XML). Los esquemas tienen principalmente dos aportaciones sobre la DTD: su sintaxis es también XML y el gran número de tipos de datos que incorpora. Sin embargo, para esta primera etapa de desarrollo de JReports no se van a usar muchas de las complejas características que ofrecen los esquemas, pues todo el modelo de los reportes está compuesto por expresiones simples, así que al menos por el momento, una DTD es suficiente para el poder expresivo requerido para esta aplicación. En un futuro, si los documentos a modelar se volvieran más sofisticados y fuera necesario hacer uso de especificaciones complejas no resultaría muy complicado generar un esquema a partir de la DTD con la que se cuenta.

3.3. Desarrollo

3.3.1. Entorno de Desarrollo

Para la implementación de la aplicación se empleó el entorno de desarrollo de código abierto *Eclipse*, totalmente desarrollado en Java. Se trata de implementar un módulo de extensión para eXist, así que para hacer más fácil el proceso de prueba/desarrollo se instaló el plugin Subclipse para Eclipse como cliente para conectarse a repositorios de Subversion (SVN) y así obtener una copia del proyecto en un repositorio dado y toda la funcionalidad con que cuenta un sistema de control de versiones como lo es Subversion.

Simplemente es necesario conectarse al repositorio SVN de eXist, seleccionar la raíz que se quiere obtener (ejemplo trunk/eXist), realizar la primera descarga (checkout) y se tendrá de manera local una copia de todo el código de eXist.

XQuery no es solamente un lenguaje poderoso de consulta, es también un lenguaje de programación funcional que puede usarse para implementar la lógica de una aplicación web y esta característica puede ser explotada a la perfección en eXist donde se han añadido diversos módulos como son:

- Obtener parámetros HTTP Request
- Asignar y obtener atributos de sesión
- Codificar direcciones URL
- Módulos de extensión para: enviar mails, recuperar datos de una base de datos relacional, manipular imágenes, etc.

Con estas características, es posible usar XQuery como lenguaje del lado del servidor (server-side), tal como son JSP o PHP con la ventaja de que para quien ya lo conoce resultará más fácil escribir todo en XQuery, además de que la aplicación completa utilizaría la tecnología XML, no sólo los datos a procesar.

La Figura 3.1 ilustra la arquitectura de una aplicación Web XQuery típica. La respuesta del servidor puede ser en cualquier formato que el cliente requiera, incluyendo HTML, XML, plain-text, etc. El motor de XQuery es el encargado de ejecutar las consultas predefinidas y el programador sólo define la lógica del negocio y la presentación de los resultados de la consulta. Los datos XML pueden venir de diversas fuentes como son un sistema de archivos, una URI o una base de datos.

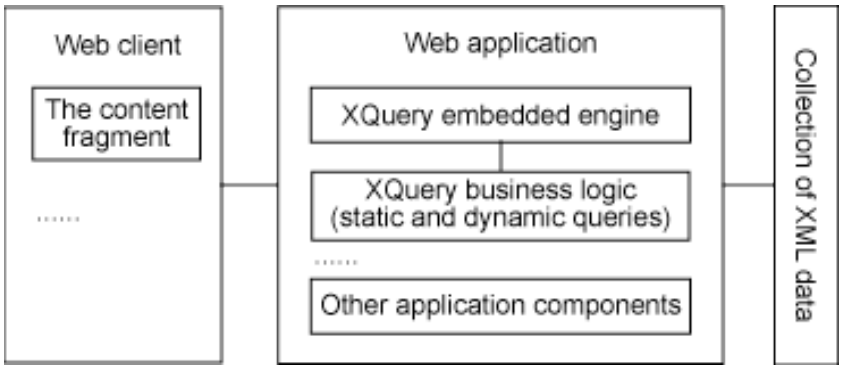

Figura 3.1: Arquitectura de una aplicación web XQuery. Tomada de [17]

En eXist es sencillo añadir funcionalidad a las aplicaciones mediante módulos de extensión implementados en Java, tales funciones pueden ser llamadas desde la aplicación principal XQuery [19].

Estos módulos de extensión permitirán añadir funciones adicionales utilizando XQuery a través de espacios de nombres. Los módulos de extensión tienen total acceso a la base de datos de eXist, al conjunto de funciones nativas definidas en el API, al contexto de ejecución de XQuery y a la sesión de HTTP (si ésta es necesaria).

El código fuente de un módulo de extensión que se vaya a desarrollar debe colocarse en una carpeta, creada desde eclipse y que estará dentro de

$$\$EXIST\_HOME/extensions/modules/src/org/exist/xquery/modules$$

para que de manera automática, al ejecutar el proyeto de eXist desde eclipse, la funcionalidad del módulo de extensión sea añadida.

3.3.2. Arquitectura del Sistema

La lógica del sistema JReports se implementa básicamente en dos módulos:

1. Un módulo para generar el índice inicial en Lucene.
2. Un módulo XQuery de Consulta y un Trigger de eXist.

La idea básica del proceso consiste en que una vez que eXist se encuentre corriendo en el sistema, se debe ejecutar la herramienta indexadora, la cual se conecta a la base de datos donde se encuentran almacenados los documentos y a partir de estos, genera con la ayuda de Lucene, un índice inicial de los documentos.

Cuando eXist está corriendo, el Trigger se encarga de actualizar el índice de Lucene en caso de que algún documento en la base de datos sea modificado, agregado o eliminado.

Finalmente, el módulo de extensión de XQuery proporciona la funcionalidad necesaria para consultar el índice de Lucene y recuperar los documentos almacenados en la base de datos que satisfagan una consulta.

3.3.3. Módulo Generador del Índice Inicial

Este módulo implementa la parte del sistema que genera el índice inicial con los documentos que se encuentran almacenados en la base de datos de eXist.

La clase `Indexer` del módulo recorre el contenido de un reporte codificado en XML e indexa la información que se encuentra en cada uno de los elementos definidos en el esquema de codificación.

La clase incluye métodos para añadir la información de las diferentes secciones en que se divide un documento (encabezado, materia inicial, cuerpo y materia final). Para extraer la información que contiene el documento XML se utiliza un parser de XML compatible con DOM. Los campos que se indexarán son los siguientes:

	id	Identificador único del documento indexado, en este caso se trata del nombre del archivo.
Header (Encabezado)	title author	Se trata de la información descriptiva del documento. Pueden considerarse los metadatos del documento.
Front (Materia Inicial)	docAuthor docOwner docDate docTitle titlePartMain titlePartSub figures summary abstract references	Existen etiquetas especializadas para transcribir portadas para asegurar que el software de proceso localice e identifique autor, título, y fecha del documento como aparecen en ella. El elemento <titlePage> (generalmente dentro de <front>) contiene, entre otros, los elementos <docTitle> (con subelementos <titlePart> que por su atributo "type" identifican partes concretas del título).
Body (Cuerpo del documento)	keywords	Almacena todo el contenido del cuerpo del documento, para poder realizar búsquedas de texto completo.
Back (Materia Final)	appendix glossary	También se podría incluir en esta sección información como el índice, notas o incluso codificar aquí las referencias bibliográficas.

Tabla 3.1: Campos que formarán parte de índice invertido

Analizador

Cuando se crea un índice en Lucene, es necesario pasar el tipo de analizador que se usará para indexar el contenido. Un analizador determina las reglas para buscar en el índice, por ejemplo, un analizador simple buscaría únicamente basándose en los espacios

en blanco entre las palabras.

Para la indexación y recuperación del texto de los documentos que se manejan en JReports no basta con utilizar alguno de los analizadores que proporciona por defecto Lucene, para mejorar las búsquedas de manera que no se produzca demasiado ruido (documentos poco significativos para la consulta) en el resultado y para cumplir el objetivo de buscar los documentos más cercanos a los criterios de búsqueda, es necesario que los documentos pasen por un filtro lo más exhaustivo posible.

SpanishAnalizer

Aunque Lucene provee varios analizadores por defecto, por ejemplo, el StandardAnalyzer que ya cuenta con una lista reducida de palabras comunes en inglés y a pesar de que es posible obtener una biblioteca con analizadores en bastantes idiomas (lucene-analizers), curiosamente el español no se encuentra entre ellos.

Aún con un analizador en el idioma requerido, debería implementarse uno propio para aumentar la lista de palabras comunes, si fuera necesario, además de poder especificar el algoritmo con el que se está realizando, si es que se realiza, el stemming.

Existen diversas páginas especializadas en recuperación de información o en minería de textos que cuentan con listas de stop words para el español, siendo quizás la más importante de ellas la página de Snowball (http://snowball.tartarus.org).

JReports implementa su propio analizador en la clase `SpanishAnalyzer` que hereda de la clase `Analyzer` de Lucene. Un atributo importante de SpanishAnalyzer es la constante SPANISH_STOP_WORDS, un arreglo de cadenas que contiene algunas de las palabras del español que usualmente no son útiles para una búsqueda y que serán consideradas como stop words. En esta clase también se implementa un filtro que convierte todo el contenido a minúsculas, además de un filtro para utilizar el stemmer para español de Snowball.

En la Figura 3.2 se presenta el diagrama de clases que describe la estructura de este módulo. En el diagrama se incluyen todas las operaciones de ambas clases, aunque se omiten los atributos por cuestiones de espacio, excepto los atributos collection e indexer debido a su importancia.

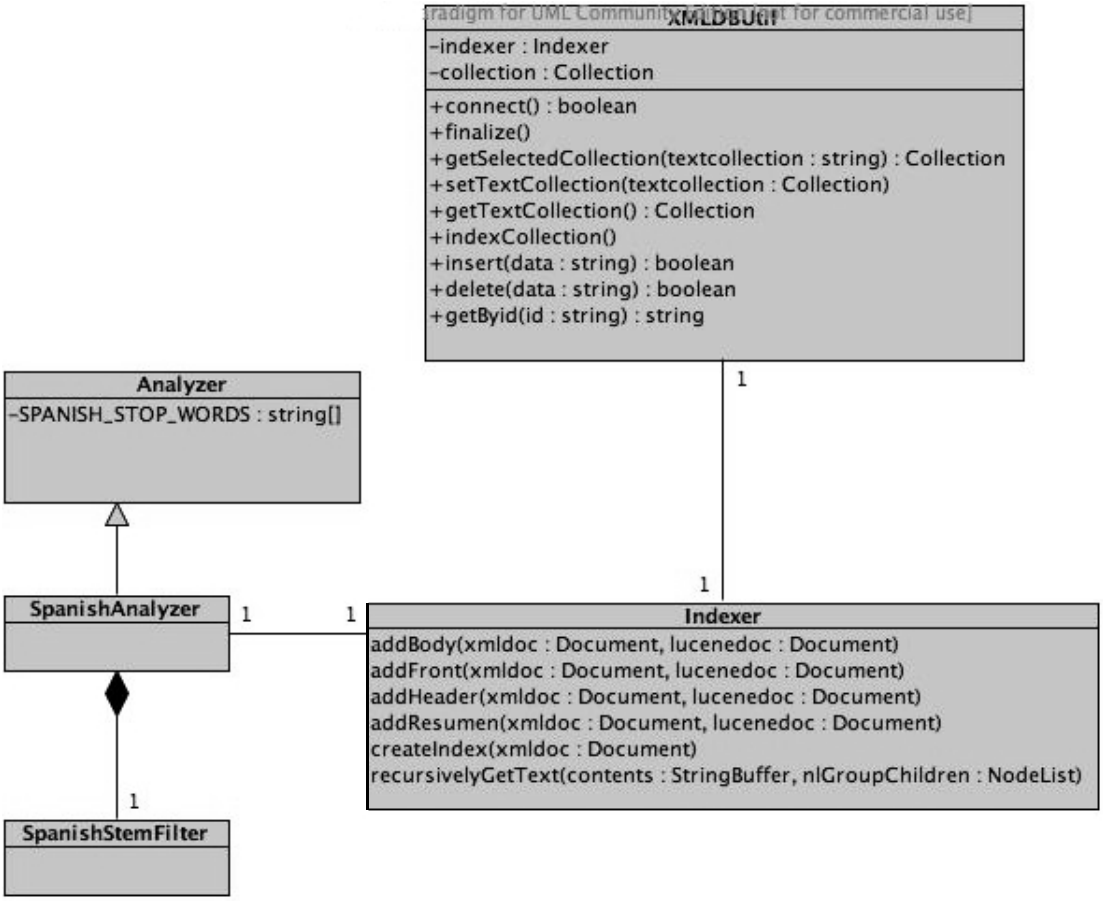

Figura 3.2: Diagrama de clases del módulo que genera el índice de Lucene.

Operaciones con la Base de Datos

La clase `XMLDBUtil`, contiene las operaciones necesarias para interactuar con eXist, para lo cual implementa métodos para:

- conectarse a la base de datos `(connect)`,
- obtener/asignar una colección (`getTextCollection`/`setTextCollection`),

- insertar/eliminar documentos de una colección (`insert`/`delete`),
- obtener un documento por su identificador en la colección (`getByid`) y
- finalmente implementa el método `indexCollection`, el cual crea una instancia de la clase `Indexer` para indexar todos los documentos que se encuentran almacenados en una colección.

3.3.4. Módulo XQuery de Consulta

En el paquete reports es donde se implementa toda la funcionalidad para la consulta del índice creado con Lucene y la recuperación de los documentos relevantes a la consulta desde la base de datos.

Incorporar funcionalidad adicional a la base de datos en eXist se logra mediante módulos definidos en Java usando XQuery de una manera relativamente sencilla. Estos módulos se conocen como módulos internos de extensión y en la página oficial de eXist se describe detalladamente la forma de crear dichos módulos y como agregarlos al API de eXist [19].

Para registrar un módulo de extensión, eXist necesita un espacio de nombres (una URI) con el cuál será identificado dicho módulo, así como la especificación de las funciones que este contiene. Todo esto se le pasa al motor de XQuery definiendo una clase *manejadora* que implemente la interfaz `InternalModule` o, que sea una subclase de la clase `AbstractInternalModule`, ambas forman parte de la API de eXist [21]. Siguiendo este lineamiento, JReports define, la clase `JReportsModule` que extiende a `AbstractInternalModule` y será entonces la clase manejadora del sistema.

El constructor de esta clase manejadora debe recibir un arreglo con la definición de las funciones que contiene el módulo que se va a registrar. En la API de eXist esta definición de funciones se especifica en la clase `FunctionDef` mediante dos propiedades: la firma de la función (una instancia de la clase FunctionSignature) y la clase de Java donde se implementa la función. Esta clase `FunctionDef`, es entonces usada por los módulos

para definir las funciones que éste tiene disponibles.

Las funciones definidas para la clase manejadora de JReports son dos. La primera de ellas se llama `keyword-search`, la cual construye la cadena final de consulta para el índice de Lucene, a partir de los argumentos de la función, se ejecuta la búsqueda en Lucene la cual regresa un arreglo de cadenas con los nombres de los reportes que satisfacen el criterio de búsqueda, para finalmente, regresar una secuencia de los documentos obtenidos. Así, el usuario introducirá en el sistema las palabras clave que está buscando y las secciones del documento en que éstas deben aparecer. Por ejemplo, para una consulta con la frase "Efecto invernadero" dentro del título y el autor "Alfredo Gutierrez", el sistema deberá regresar un arreglo con los documentos que contienen dicha frase en el título y que tienen como autor a Alfredo Gutierrez. Para esto, *keyword-search* realiza las búsquedas en el índice de Lucene. La segunda función se llama *print-query* que únicamente regresa la cadena que representa la consulta introducida por el usuario en el formato aceptado por Lucene, para el ejemplo anterior esta cadena sería: titulo:"Efecto invernadero" AND autor:"Alfredo Gutierrez".

Estas funciones se encuentran implementadas en las clases `SearcherFunction` y `PrintQueryFunction`, respectivamente y ambas extienden a la clase abstracta `BasicFunction` de la API de eXist.

La firma de estas funciones se indica en el atributo *signature* de las clases: `SearcherFunction` y `PrintQueryFunction`.

Este atributo es una instancia de la clase `FunctionSignature` cuyo constructor recibe como argumentos:

- Una instancia de la clase `QName` que permite identificar a la función
- Una cadena de caractéres para describir brevemente a la función
- Los tipos y cardinalidad de las secuencias de cada argumento
- El tipo de la secuencia del valor de regreso

En las clases `SearcherFunction` y `PrintQueryFunction` también se sobreescribe el método *eval* que es el encargado de procesar la función. Este método recibe dos argumentos: un arreglo de los valores de todos los argumentos pasados a la función (cuyos tipos ya se ha checado que correspondan a los definidos en la firma de la función) y la secuencia del contexto actual.

Para añadir la función `keyword-search` al nuevo módulo, es necesario, como ya se mencionó, proporcionar una clase *manejadora*, donde se definirá un espacio de nombres y un prefijo para el módulo. Las funciones se registran pasándole al constructor un arreglo de instancias de `FunctionDef`.

Los módulos creados de esta manera podrán ser utilizados en cualquier script de XQuery importándolos como se ilustra en el fragmento de código 3.1:

```
xquery version "1.0";

import module namespace jreport="http://exist-db.org/xquery/jreports"
at "java:org.exist.xquery.modules.reports.JReportsModule";
```

Código 3.1: Importando el módulo

El motor de XQuery reconoce el prefijo *java:* y asume que lo que sigue a continuación es el nombre completo de la clase manejadora de este módulo. Ya que se importó el módulo, es posible llamar a las funciones que este tiene disponibles con su *nombre calificado*, que consiste en un prefijo, al cual se le asignó la URI del espacio de nombres en la línea anterior y en el nombre local de la función, por ejemplo `keyword-search` con sus respectivos parámetros.

```
jreport:keyword-search($keyword, $author, $resumen, $abstract, true())
```

Código 3.2: Llamando a la función keyword-search

A continuación, se incluye el diagrama de clases del módulo XQuery de consultas.

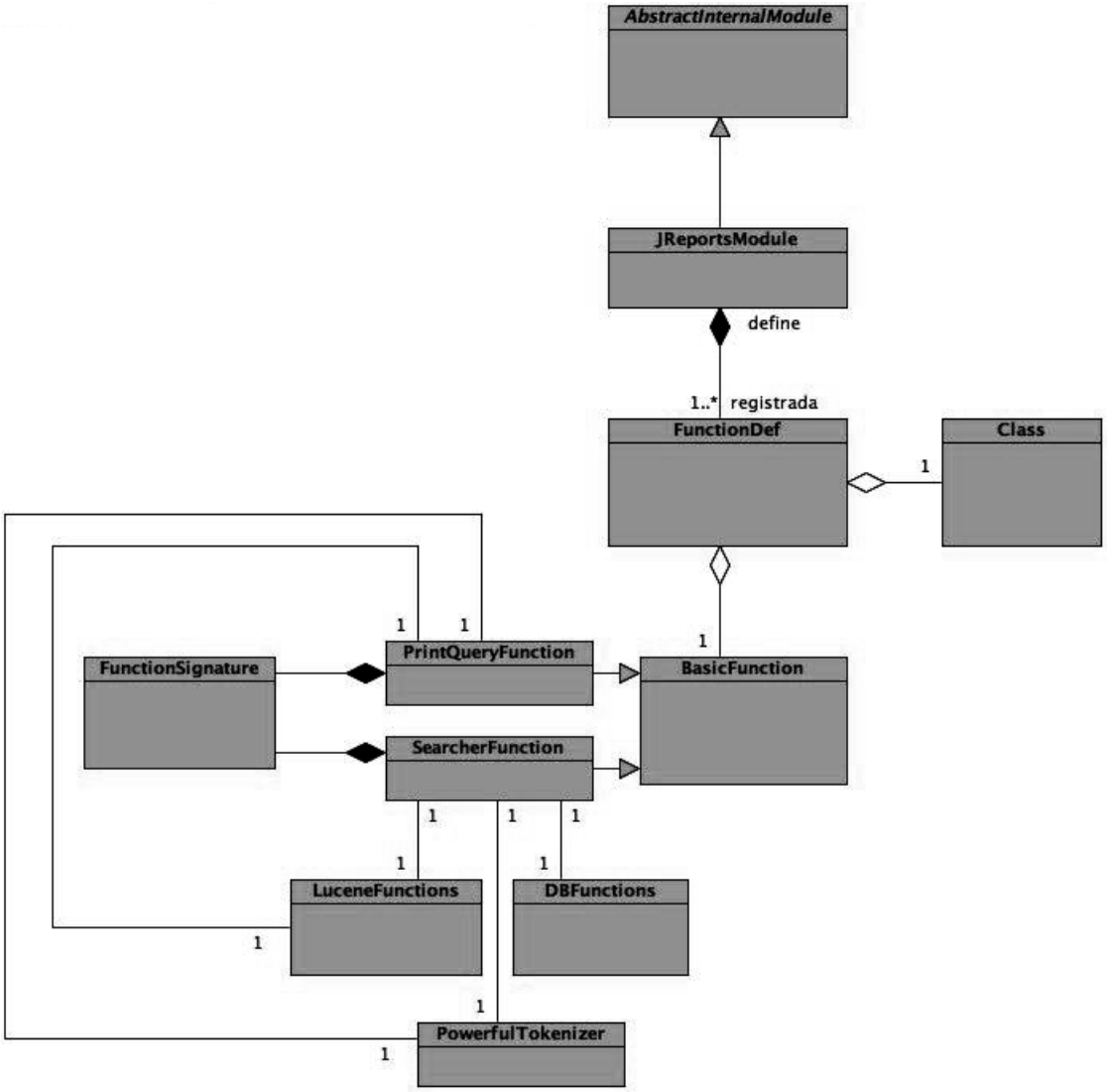

Figura 3.3: El diagrama incluye las clases más significativas del módulo XQuery.

Índice de n-gramas

En el capítulo 1 se describieron las dos alternativas para implementar una búsqueda tolerante en un sistema de recuperación de información, distancia de edición e índice inverso de n-gramas. En esta sección se tratará la manera en la que JReports implementa el módulo de búsqueda tolerante, que en el fondo utiliza ambas técnicas en conjunto.

Como se vió previamente, una de las opciones para manejar los errores ortográficos cometidos por el usuario en la consulta es presentar una consulta alterna como sugerencia y en cierto modo tratando de interpretar lo que el usuario quiso decir en los términos que

en principio presentan errores ortográficos y por lo tanto difícilmente se encontrarán en algún documento. Este, es precisamente el enfoque empleado por JReports.

El código de búsqueda tolerante de JReports se basa en gran parte en el artículo *Did You Mean: Lucene?* [11]. La idea consiste básicamente en crear una instancia de la clase `SpellChecker` del paquete spell de Lucene para generar un índice de n-gramas a partir del índice invertido original para cada palabra que en él aparece.

En JReports se implementa la clase `DidYouMeanIndexer` para crear este índice de n-gramas, que únicamente tiene el método `createSpellIndex`. Este método crea un objeto de la clase `LuceneDictionary` para iterar sobre las palabras en un campo dado del índice original. En este caso, itera sobre las palabras en el campo *didyoumean* del índice original. Después se crea la instancia de la clase `SpellChecker`, cuyo constructor recibe como argumento el directorio donde se creará el índice de n-gramas para el diccionario.

Para poder proveer a JReports de una búsqueda tolerante como se ha descrito, se implementan las clases `DidYouMeanSearchEngine` y `CompositeTolerantParser`.

`DidYouMeanSearchEngine` solicita a `CompositeTolerantParser` la mejor sugerencia si no se recupera una cantidad mínima de documentos para una consulta. Esta tarea la realiza el método *suggest* de `CompositeTolerantParser` que regresa una instancia de la clase `Query`.

En `CompositeTolerantParser` nuevamente se construye una instancia de la clase `SpellChecker` para leer el índice de n-gramas y, para cada término de la consulta verificar si este existe en el índice, de ser así, no se hace ninguna sugerencia, pues se asume que la palabra está correctamente escrita. En caso contrario, se solicita una sugerencia a la instancia de `SpellChecker` llamando al método *suggestSimilar*. En caso de que se encuentre una sugerencia, esta se añadirá al objeto `Query` que se regresará como sugerencia final.

El algoritmo que implementa el método *suggestSimilar* es bastante intuitivo. Básicamente, cada palabra del diccionario (todos los términos de la colección de documentos que se encuentran en el índice invertido) es dividida en segmentos de tres y cuatro caracteres y almacenada en un índice de Lucene con los siguientes campos:

Campo	Descripción	Ejemplo
word	La palabra del diccionario.	cancer
gram3	Campo múltiple que apunta a todos los 3-gramas que componen la palabra.	can, anc, nce, cer
gram4	Campo múltiple que apunta a todos los 4-gramas que componen la palabra.	canc, ance, ncer
start3	Los primeros tres caracteres de la palabra.	can
end3	Los últimos tres caracteres de la palabra.	cer
start4	Los primeros cuatro caracteres de la palabra.	canc
end4	Los últimos cuatro caracteres de la palabra.	ncer

Tabla 3.2: Estructura del índice de n-gramas en Lucene.

A los n-gramas de principio y fin de palabra se les asigna un mayor peso que a los demás n-gramas, porque se asume que la gente comete menos errores al principio y final de una palabra. Otra razón para indexar por separado los n-gramas de principio y fin de palabra es su función posicional, a diferencia de los demás n-gramas. Por ejemplo, las palabras *cosa* y *saco* tienen casi los mismos conjuntos de unigramas y bigramas (gram1:c gram1:o gram1:s gram1:a gram2: co gram2:sa gram2:os gram2:ac), así que los campos *start* y *end* son necesarios para distinguir entre ambas palabras (start1:c end1:a start2:co end2:sa para *cosa* y start1:s end1:o start2:sa end2:co para *saco*).

Una vez que se cuenta con el índice de n-gramas, *suggestSimilar* transforma cada término de la consulta del usuario en una consulta booleana con la sintaxis de Lucene cuyos términos serán los n-gramas del término original que aparecen en el índice (en este caso

los 3-gramas y 4-gramas) y ejecuta la consulta en el índice de n-gramas en búsqueda de posibles sugerencias. Para la palabra incorrecta ortográficamente *cancer* la consulta se vería como sigue:

start3:can^2.0 end3:cer gram3:can gram3:anc gram3:nce gram3:cer
start4:canc^2.0 end4:ncer gram4:canc gram4:ance gram4:ncer

El método *suggestSimilar* devuelve entonces, una lista de sugerencias ordenadas por la distancia de edición, con lo cual, la palabra más similar a la palabra incorrecta será la primera en la lista. Opcionalmente, es posible indicarle al método que también se tome en cuenta la frecuencia de la sugerencia en el índice.

El módulo de JReports que implementa la búsqueda tolerante y las relaciones entre las diferentes clases que lo componen se ilustra en la Figura 3.4.

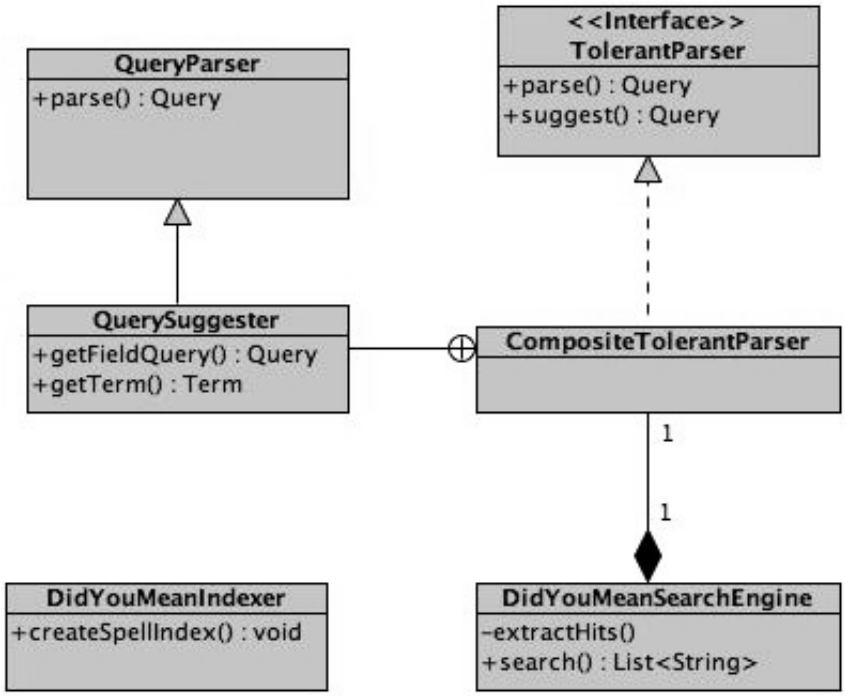

Figura 3.4: Diagrama de clases del módulo que implementa la búsqueda tolerante.

Una vez que se obtienen los nombres de los documentos que satisfacen la consulta

original del usuario o la consulta sugerida por el sistema, éstos pueden ser recuperados de la base de datos, obviamente en el formato XML, pero con XQuery también será posible aplicar las transformaciones necesarias para darle a cada documento una presentación más clara para el usuario; por ejemplo, HTML.

3.3.5. Trigger de eXist

Primero conviene volver a plantear las necesidades del sistema, para poner las cosas en perspectiva. El sistema a implementar utilizará las facilidades de indexación y recuperación de información que ofrece la biblioteca Apache Lucene sobre una colección de documentos XML.

Se usa eXist como base de datos nativa XML (NXD) para tener los beneficios de XQuery (ausente en muchos productos comerciales) para generar los reportes finales, así que resulta lógico pensar en que se puede generar el índice de Lucene de un documento al mismo tiempo que se almacena en eXist. De esta manera, no se tendría que volver a extraer el documento recién almacenado en eXist para luego indexarlo en Lucene.

La implementación de este procedimiento resulta más fácil de lo que se podría pensar con la ayuda de uno de los conceptos más importantes en una base de datos: los *triggers* (*disparadores* en español, aunque aquí utilizaré el término en inglés).

Los triggers son funciones que se ejecutan de forma automática en respuesta a ciertos eventos que ocurren en la base de datos. En eXist, los triggers pueden ser configurados por el usuario para responder a los siguientes eventos a nivel documento o nivel colección:

store: Lanzado cuando un documento es almacenado en la colección o sub-colección.
update: Lanzado cuando un documento es modificado en la colección o sub-colección.
remove: Lanzado cuando un documento es eliminado de la colección o sub-colección.
create: Lanzado cuando una sub-colección es creada.
rename: Lanzado cuando una sub-colección es renombrada.
delete: Lanzado cuando una sub-colección es eliminada.

Un trigger puede implementarse en XQuery o en Java y se disparará dos veces, una vez antes del evento y otra vez después del evento. Los triggers de eXist que son desarrollados en Java deben implementar la interfaz Ttigger o extender alguna de las clases que la implementan como la clase `FilteringTrigger` que implementa a su vez la interfaz `DocumentTrigger`.

La interfaz `DocumentTrigger` especifica los métodos *prepare* (disparado antes del evento) y *finish* (disparado después del evento). En este método *finish* es donde se verifica el tipo de evento que ha ocurrido y se ejecutan las operaciones que correspondan a este evento.

Mediante este trigger se garantiza que los datos que se encuentren indexados en Lucene, siempre serán consistentes con los documentos de la base de datos, pues las modificaciones se harán de manera conjunta.

Para ilustrar la forma en la que se define un trigger en eXist y la interacción con las demás clases e interfaces de la API se incluye en la Figura 3.5 el diagrama de clases del Trigger desarrollado para JReports.

Como se puede observar, la clase `ReportTrigger` es donde JReports implementa el trigger. Esta clase hereda de `FilteringTrigger`. Para ser consistentes con la indexación de la colección con el módulo generador del Índice, `ReportTrigger` utiliza también la clase `SpanishAnalyzer` con el filtro `SpanishStemFilter`.

3.3.6. Generación de XML y HTML usando XQuery

Cuando se trata de definir XQuery, inmediatamente se piensa en un lenguaje de consulta para una NXD. Pero XQuery ofrece un valor agregado bastante útil, también es un lenguaje para generar XML y HTML, desarrollando así páginas Web dinámicas. En esta sección se presenta a XQuery desde este ángulo.

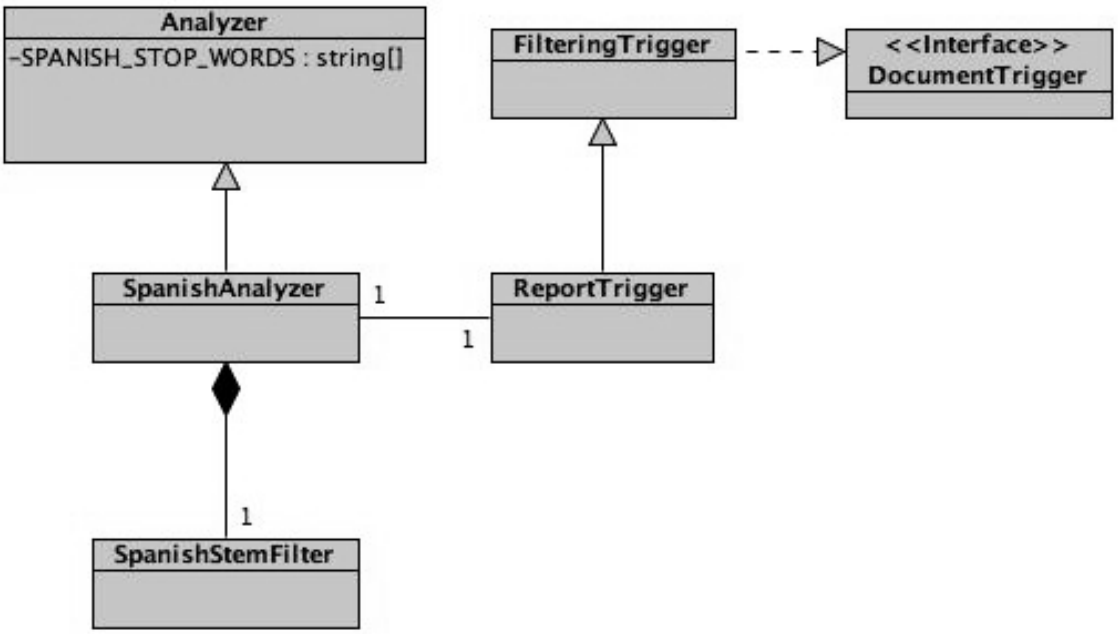

Figura 3.5: Diagrama de clases de la implementación de ReportTrigger.

Herramientas para generar páginas Web dinámicas

Existen diversas herramientas para generar páginas Web de forma dinámica. Muchas de estas herramientas basan su funcionamiento en el uso de plantillas: se escribe una página HTML y se le insertan expresiones que son procesadas por el servidor. Entre éstas destacan:

- JavaServer Pages (JSP) es una tecnología Java que permite la utilización de código Java mediante scripts. Los JSP son en realidad servlets. Cuando se compila un JSP se crea una clase java que se ejecuta en el servidor como los servlets. Un servidor de aplicaciones Web como Tomcat se puede configurar para ejecutar el servlet cada que reciba una petición HTTP. Un sencillo ejemplo de JSP sería:

```
<p>3*4 es: <%= 3*4%>.</p>
```

Código 3.3: JSP

- PHP Hypertext Pre-processor (PHP) es un lenguaje interpretado de propósito general ampliamente usado y que está diseñado especialmente para desarrollo web y puede ser embebido dentro de código HTML. Generalmente se ejecuta en un servi-

dor web, tomando el código en PHP como su entrada y creando páginas web como salida. Así se escribiría el ejemplo anterior con PHP:

```
<p>3*4 es: <?php echo 3*4?>.</p>
```

Código 3.4: PHP

- Active Server Pages (ASP) es una tecnología de Microsoft del tipo *lado servidor* para páginas web generadas dinámicamente, que ha sido comercializada como un anexo a Internet Information Services (IIS) y por tanto, está estrechamente relacionada con el modelo tecnológico de su fabricante. Las páginas pueden ser generadas mezclando código de scripts del lado del servidor con HTML. Otra vez el mismo ejemplo, ahora con ASP:

```
<p>3*4 es: <%= 3*4>.</p>
```

Código 3.5: ASP

- XQuery permite hacer lo mismo de la siguiente manera:

```
<p>3*4 es: {3*4}.</p>
```

Código 3.6: XQL

Existe, sin embargo, una diferencia importante entre XQuery y las herramientas antes mencionadas. Sólo con XQuery es posible anidar código HTML dentro de las expresiones del lenguaje, como en este ciclo:

```
for $i in (1 to 10) return
   <p>{$i}*4 is: {$i*4}</p>
```

Código 3.7: HTML dentro de código XQuery

Las otras alternativas únicamente permiten anidar expresiones en el lenguaje de programación dentro de HTML, como los ejemplos del listado anterior.

En XQuery es posible además definir funciones que regresen fragmentos HTML, los cuales a su vez pueden ser pasados a otras funciones, mientras que con la mayoría de las tecnologías basadas en plantillas sólo es posible generar estos fragmentos HTML como cadenas.

Comparación de XSLT y XQuery

XSLT (Extensible Stylesheet Language Transformation) es un lenguaje potente para transformar documentos XML a documentos en diversos formatos como son XML, HTML, PDF, o texto plano.

La diferencia más *visible* entre XSLT y XQuery está en que un programa XSLT (llamado también hoja de estilo) es también un documento XML. Esto en muchos casos resulta bastante útil, pero también puede representar una gran desventaja, pues a pesar de su simplicidad, una hoja de estilo XSLT puede resultar difícil de leer.

Por otro lado, la diferencia más significativa entre XSLT y XQuery está en el modelo de ejecución, específicamente en el control de flujo. En XQuery depende de la implementación el orden en el que se evaluarán las expresiones, mientras este orden se apegue al estándar.

En XSLT, este control de flujo está dado por el orden del documento, es decir, su modelo de ejecución automáticamente recorre el árbol que modela al documento XML y conforme va encontrando los nodos, de un conjunto de plantillas previamente definidas aplica aquella cuyo patrón se ajuste más al nodo en turno. Este proceso puede repetirse de manera recursiva con la instrucción <xsl:apply-templates>.

El uso de patrones para conducir la ejecución de esta manera resulta muy conveniente, sobre todo cuando se están realizando conversiones relativamente simples y que pueden ser expresadas usando patrones. Para situaciones más complejas XSLT resultaría rápidamente incómodo y poco elegante.

En la elaboración de JReports se usó únicamente XQuery para la transformación a HTML de los documentos XML almacenados en eXist. Sin embargo, en muchas situaciones se volvió complicado expresar la tarea a realizar con la sintaxis de XQuery, y en tal caso una opción sería usar XSLT. De cualquier modo, a continuación se presenta una lista con las características de ambos lenguajes para diferentes situaciones que se pueden

presentar en un aplicación:

- Si los datos se encuentran en una base de datos, es más recomendable usar XQuery
- Copiar un documento con pequeños cambios es más fácil con XSLT
- Extraer una pequeña cantidad de información de los documentos es más fácil con XQuery
- XQuery es más fácil de aprender y más simple para trabajos pequeños
- Para desarrollar componentes reutilizables es necesario utilizar XSLT

Para finalizar este capítulo, en la Figura 3.6 se muestra la arquitectura del módulo JReports y el enfoque combinado de Lucene y eXist para la recuperación XML. Primero se genera con Lucene el índice invertido con los documentos almacenados en eXist y posteriormente, para cada consulta, JReports la transforma a la sintaxis comprensible por Lucene, ésta es procesada para devolver los documentos más relevantes.

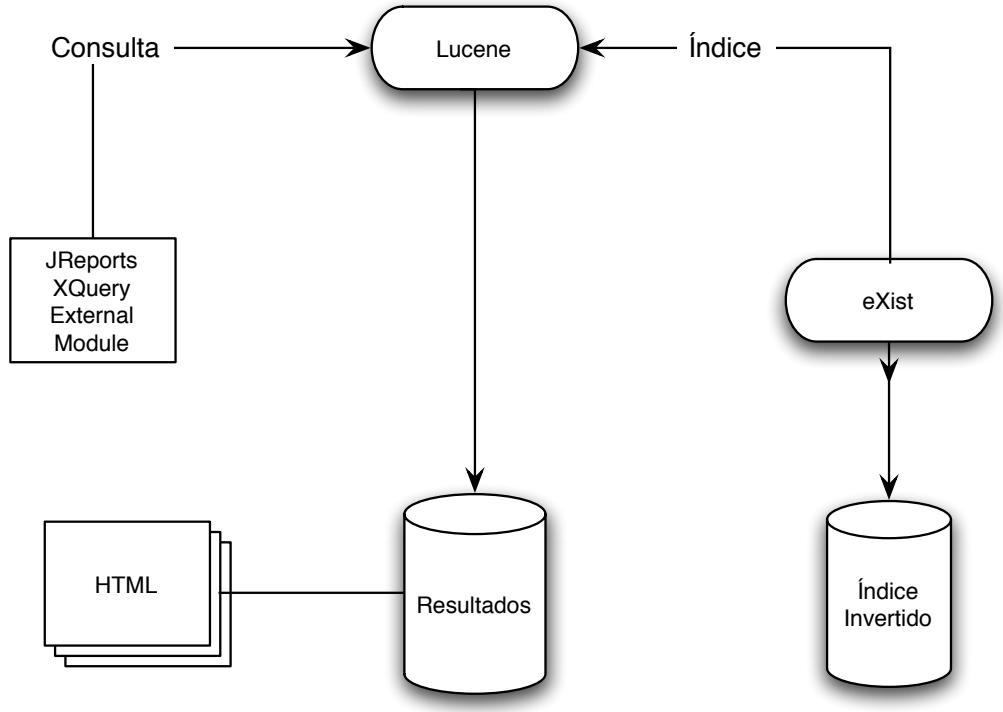

Figura 3.6: Arquitectura de JReports..

Conclusiones

Desde hace ya bastantes años el área de la recuperación de información se había enfocado mayormente a datos en texto plano. Actualmente, en el mundo de la información, los documentos en formato de texto plano están disminuyendo aceleradamente y en su lugar, se puede encontrar cada vez en mayor medida información presentada en documentos de *formato enriquecido*. Por ejemplo:

- En entornos corporativos es muy frecuente trabajar con documentos PDF, Microsoft Word, or Excel.
- La web normalmente contiene datos en HTML.
- Las aplicaciones de software usan, cada vez más, XML para intercambiar datos.

Una vez cumplidos los objetivos planteados al inicio de este trabajo, es posible concluir que JReports resulta un buen ejemplo de la manera en la que se puede implementar un sistema de recuperación de información estructurada combinando las ventajas que ofrecen un sistema de recuperación de información de texto completo como es Lucene y una base de datos nativa XML (NXD).

En este trabajo también se muestra, que aprovechando el hecho de que con Lucene es posible extraer e indexar el texto de cualquiera de estos formatos enriquecidos, los distintos centros de investigación que planeen utilizar una aplicación como JReports podrán generar un estándar de marcado en XML para facilitar el intercambio de información con otras dependencias para su posterior procesamiento.

Con el uso de diversas tecnologías novedosas como son XQuery, bases de datos nativas XML y Lucene es posible implementar toda la funcionalidad del sistema, eliminando

la necesidad de herramientas que implican tareas de procesamiento más complejas y que consumen más recursos de la máquina como un parser de DOM o SAX o transformaciones con XSLT, que a su vez requieren un mayor número de líneas de código. Almacenar los documentos en eXist, ahorra también la complicación que representa el hacer un mapeo entre los documentos XML y una base de datos relacional adaptada para XML como Oracle.

Al estar completamente desarrollado en Java, JReports es una herramienta multiplataforma, lo que significa que corre en cualquier sistema operativo como Unix, MacOS y Windows, cumpliendo entonces con una característica tan importante para cualquier aplicación de software como lo es la portabilidad.

Es importante mencionar que en México, la bibliografía especializada sobre eXist y Lucene es prácticamente inexistente, por lo que el desarrollo de JReports está basado principalmente en ejemplos de comunicaciones y de recursos en Internet relativos a dichos temas. En prácticamente todas las comunidades de software libre, la mejor manera de solicitar ayuda o información es mediante listas de correo electrónico y foros virtuales y esta no es la excepción en el caso de eXist y Lucene. Sin embargo, todos estos recursos se encuentran en inglés, de allí la importancia de haber concluido con éxito los objetivos de este proyecto, pues más allá de la experiencia adquirida por el desarrollador del mismo, de alguna manera JReports puede considerarse el resultado de la colaboración de una comunidad amplia y transnacional de programadores voluntarios. Sólo es cuestión de formular preguntas a la comunidad de manera inteligente y práctica. Ahora, este trabajo podrá servir como una referencia en español para futuros desarrollos de sistemas que conjunten las ventajas de ambas herramientas.

La intención de este sistema, desde su desarrollo inicial, era recuperar aquellos documentos que fueran más relevantes para una consulta del usuario, así que el resultado devuelto eran los documentos completos, quedando pendiente para futuras revisiones la implementación de un módulo que permita devolver no el documento en su totalidad, sino los componentes del documento que más se ajusten a la necesidad de información

del usuario, por ejemplo, un capítulo, una sección, una página, etc. característica necesaria para un completo sistema de recuperación de información en contenido y estructura (CAS).

Como el sistema aún no es capaz de identificar la granularidad deseada para la respuesta final, es necesario continuar con el trabajo de investigación y el desarrollo para implementar los métodos y estrategias para una recuperación CAS.

Por otro lado, es muy común que se quiera dar prioridad a ciertos contextos XML en la consulta. En muchas ocasiones, los usuarios le dan más importancia a algunas partes de la consulta que a otras; por ejemplo, se puede dar un mayor peso al campo autor cuando no se tiene la seguridad de recordar el título correcto. Esta asignación de prioridades puede hacerse en la interfaz de usuario como un parámetro más de la consulta.

El Apéndice D puntualiza en que etapa del desarrollo se encuentra actualmente JReports y se destacan los requerimientos funcionales que cubre por ahora el sistema. Sin embargo, al tratarse de un software libre se proporcionan también algunas claves sobre las tareas que quedan pendientes por desarrollar y sugerencias de lo que debería hacer cualquier persona interesada en continuar con el desarrollo.

Apéndice A

Manual de Usuario

Ingresar al Sistema

El ingreso a la aplicación es a través de cualquier navegador web (Firefox, Safari, Explorer, etc.) mediante la dirección electrónica:

```
http://localhost:8080/exist/xquery/registro.xql
```

Iniciar Sesión

Inmediatamente, el navegador desplegará la pantalla mostrada en la Figura a.1.

Figura a.1: Pantalla para iniciar sesión.

Esta es la ventana de inicio de sesión para poder realizar consultas en los documentos almacenados en una colección de eXist. Si se es usuario de eXist, entonces se debe ingresar el mismo **Usuario** y **Contraseña** de eXist y presionar (click con el ratón) en el botón

Ingresar y se mostrará la pantalla de la Figura a.2.

Si todavía no se está dado de alta como usuario de eXist, entonces es necesario solicitar al administrador de la base de datos que cree una nueva cuenta desde el panel de administración de eXist.

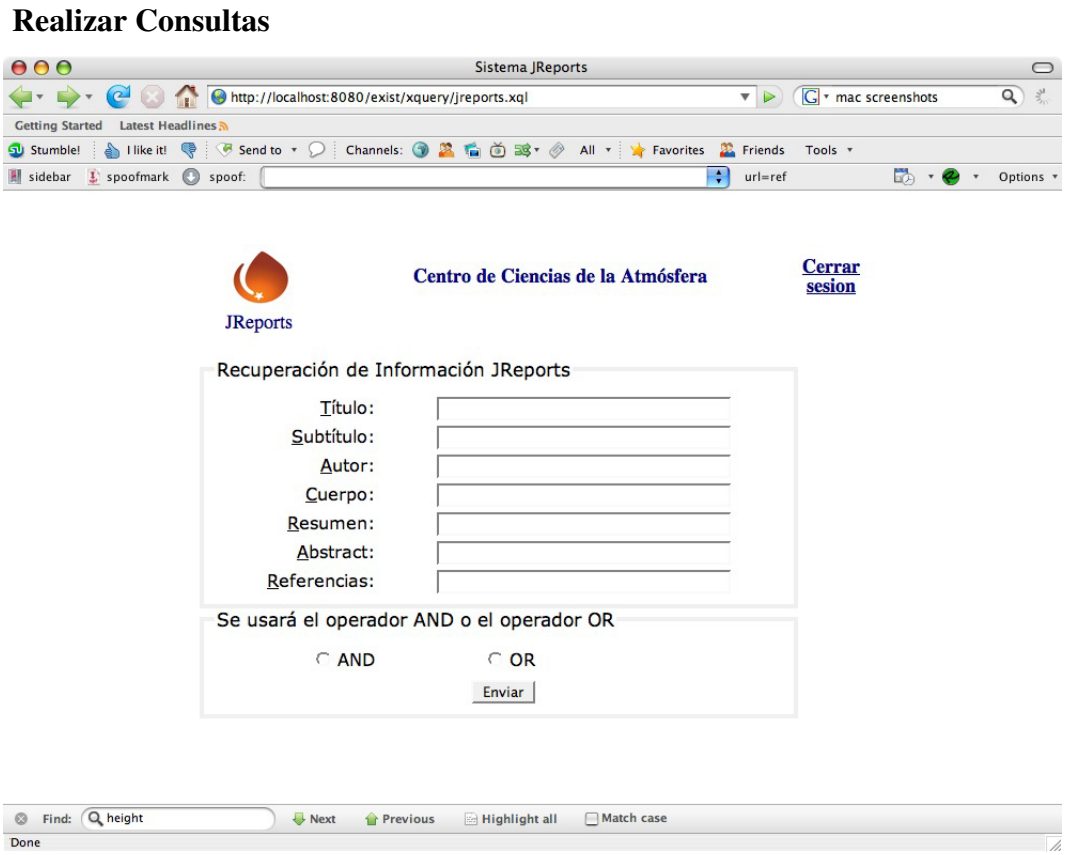

Figura a.2: Pantalla para realizar consultas en JReports.

En la pantalla de la Figura a.2 ya es posible introducir los criterios de búsqueda con los cuales deben coincidir los documentos que recuperará el sistema. El sistema permite introducir términos para los campos título, subtítulo, autor, resumen, abstract y referencias en sus respectivas áreas de texto. En esta pantalla también se puede especificar el operador booleano que se aplicará a los términos introducidos en cada campo, ya sea OR o AND, siendo OR el operador por defecto.

La Figura a.3 ejemplifica una consulta en JReports con los términos “efecto invernadero” para el campo título y el término “luis gerardo ruiz” para el campo autor.

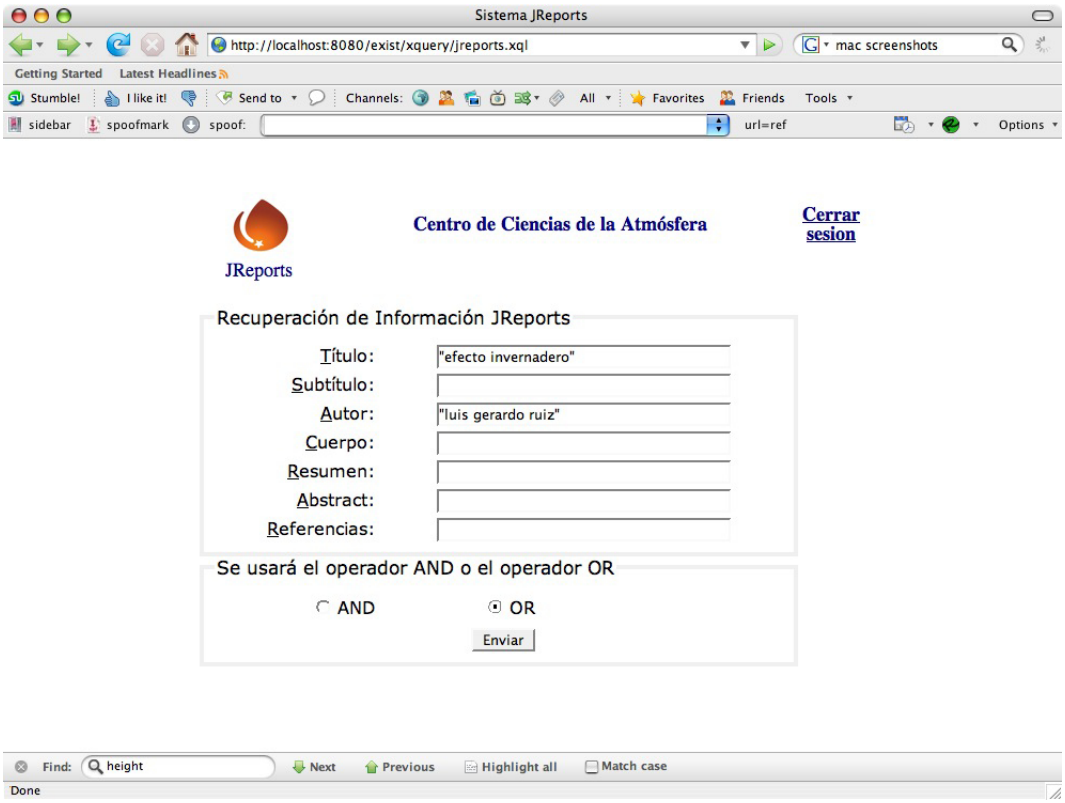

Figura a.3: Ejemplo de consulta para JReports.

Consultas con errores ortográficos

Si el usuario comete algún error ortográfico en los términos que conforman su consulta, como en la Figura a.4, donde se tecleó “efecro inbernadero” en lugar de “efecto invernadero”, el sistema devolverá como sugerencia una cadena con los términos correctos que más se aproximen a los términos de la consulta original. En la pantalla de la Figura a.5 se puede observar que el sistema devuelve como sugerencia en la línea que dice *Quizás quiso decir* el término correcto “efecto invernadero”.

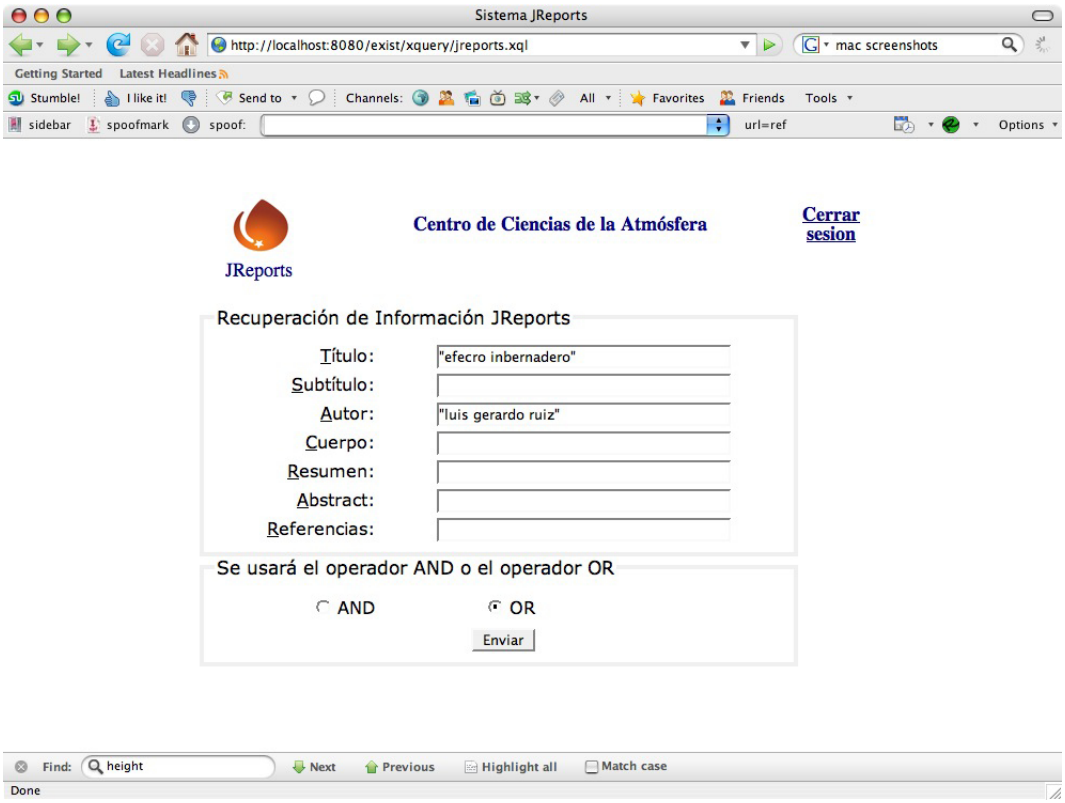

Figura a.4: Errores ortográficos en algunos términos de la consulta.

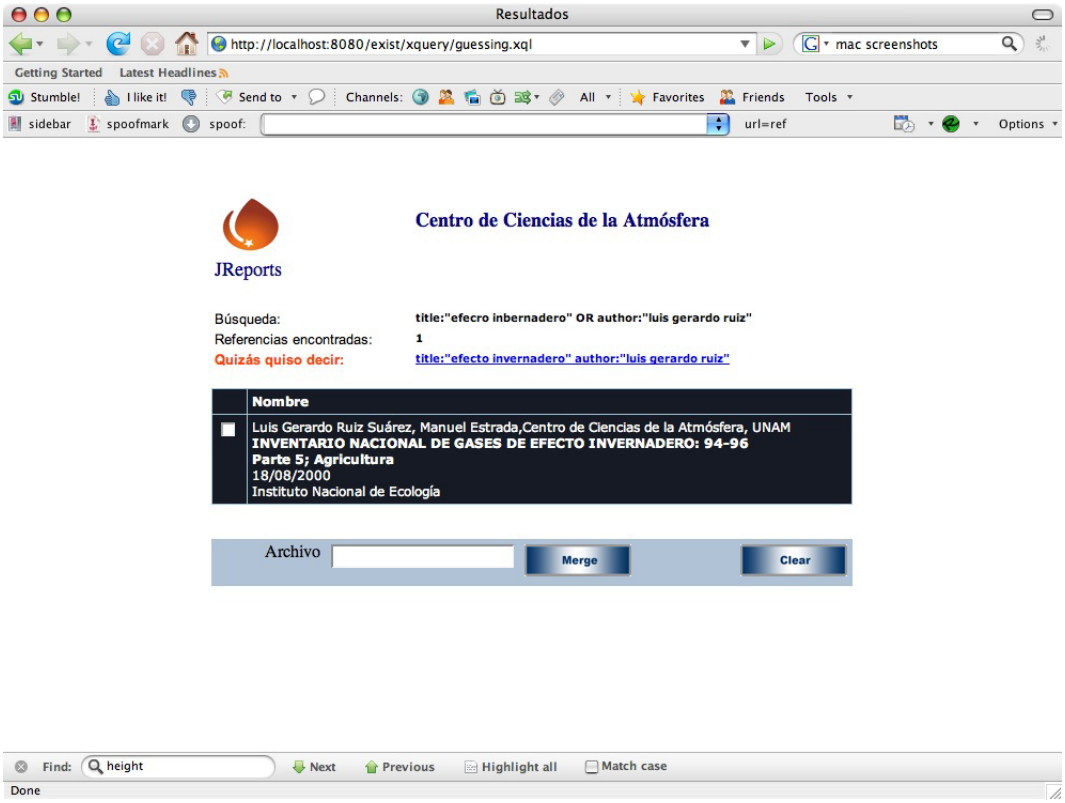

Figura a.5: Si hay algún error ortográfico en la consulta, el sistema tratará de hacer alguna sugerencia.

Desplegar Resultados

Una vez introducidos todos los términos de la consulta en sus respectivos campos y seleccionado el operador booleano, se debe presionar el botón **Enviar** y se mostrará la pantalla de la Figura a.6 donde se despliegan los documentos encontrados por JReports que coinciden con la consulta dada.

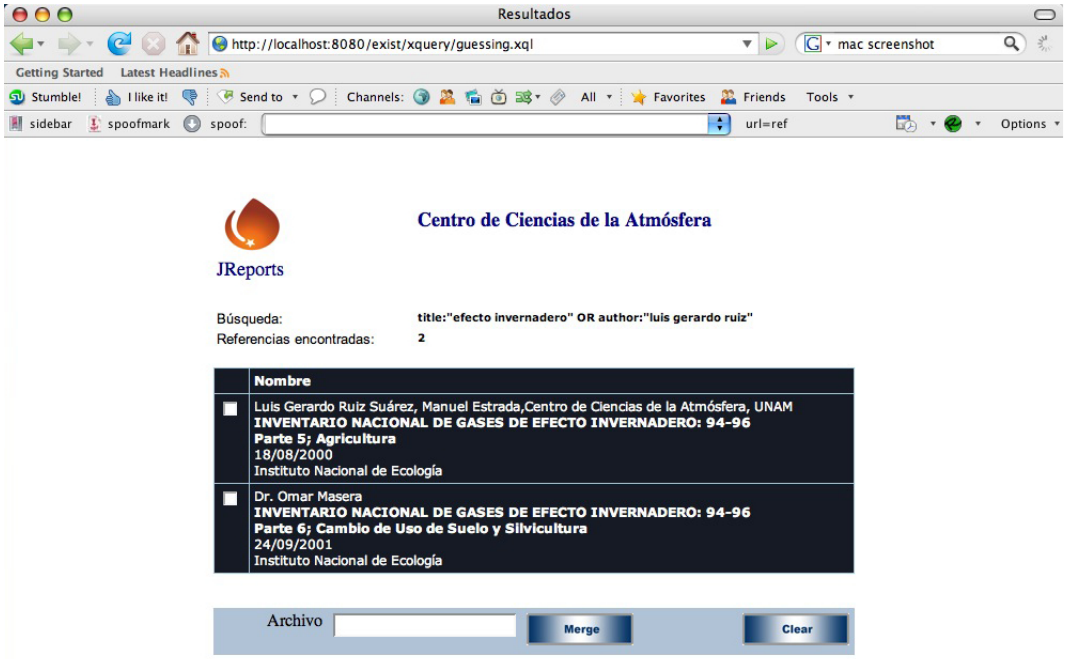

Figura a.6: Resultados que coinciden con los criterios de la consulta.

Mostrar Contenido del Documento

Para poder visualizar el contenido de alguno de los documentos presentes en la lista de resultados, es necesario posicionar el cursor sobre el documento de interes y presionar el ratón, e inmediatamente se abrirá una nueva ventana desplegando el contenido del documento, después de que éste haya sido transformado de XML a HTML. Por ejemplo, el contenido del primer documento de la lista de resultados de la pantalla anterior se muestra en la Figura a.7.

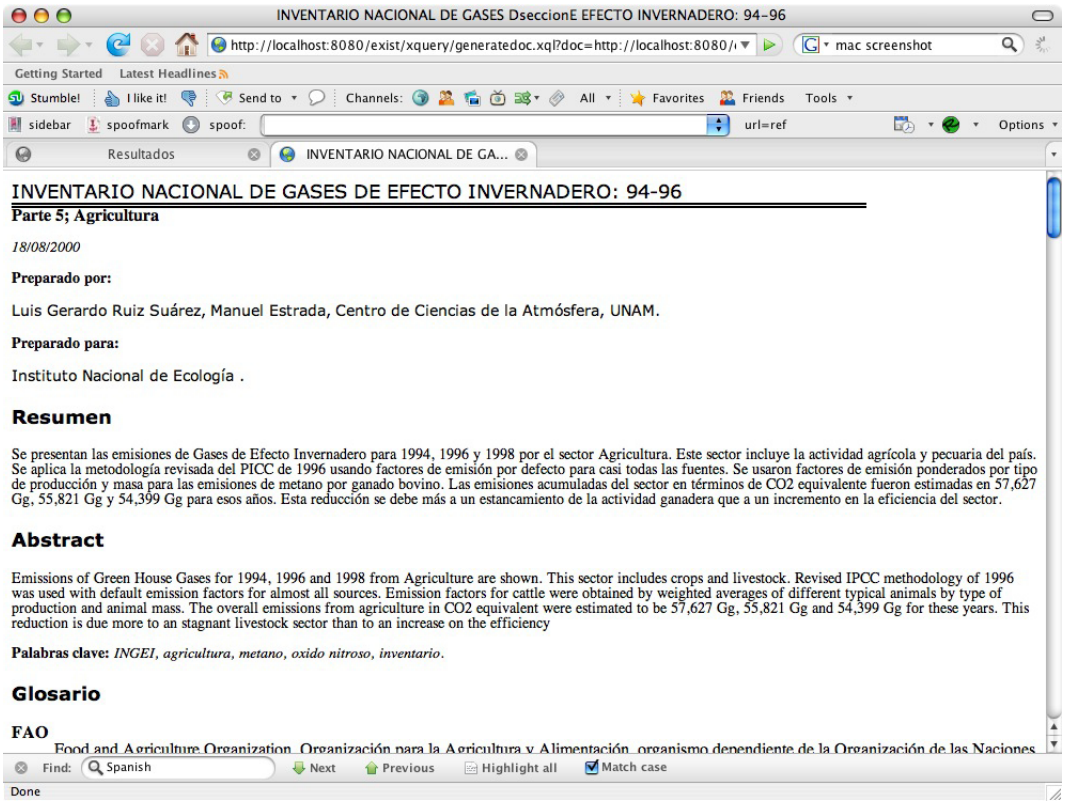

Figura a.7: Resultados que coinciden con los criterios de la consulta.

Cerrar sesión

Para terminar la sesión de usuario de manera voluntaria, finalizando así el uso del sistema, es necesario presionar la liga de **Cerrar sesión** a la derecha de la pantalla de la Figura a.2.

La acción de cerrar de sesión también puede ser automática. En general, cuando transcurre un determinado período de tiempo sin actividad, la sesión se cierra automáticamente por cuestiones de seguridad. Igualmente cuando se cierra el navegador, se termina la sesión.

Apéndice B

Manual de Instalación

Como se ha mencionado, JReports es un módulo de extensión del código de eXist, así que es necesario contar con toda la estructura de archivos de eXist y el módulo en el sistema de archivos.

JReports se distribuye entonces con todo el código fuente de eXist, para que se pueda importar como un nuevo proyecto de eclipse y poder generar el índice inverso de los documentos que se van a almacenar en la base de datos, además de que quien así lo desee puede revisar y/o corregir el código o agregar nuevos módulos.

Para importar el código de eXist con JReports que se distribuye en el CD que contiene esta tesis dentro de su espacio de trabajo de eclipse, se deben seguir los siguientes pasos:

1. Crear un nuevo proyecto de eclipse llamado JReports. Asegurarse de que el proyecto está seleccionado.
2. Seleccionar *File* → *Import*.
3. En la ventana de Import que se despliega, seleccionar *File System* y dar click en *Next*.
4. Localizar el directorio JReports del CD usando el botón *Browse*.
5. En la ventana que se despliega checar la casilla de verificación con el nombre de JReports y dar click en *Finish*.

6. Si aparecen mensajes preguntando si se desea sobreescribir el *.classpath* y el *.project* dar click en *Yes*. Eclipse comenzará ahora a importar toda la estructura de archivos de eXist al nuevo proyecto y al finalizar ya estará la aplicación en nuestro espacio de trabajo de eclipse.

El siguiente paso es recompilar la aplicación para generar un archivo Web (WAR), para poder hacerla portable. Las indicaciones para hacer esto se pueden encontrar a mayor detalle en la siguiente URL:

http://atomic.exist-db.org/blogs/dizzzz/Howto create installers/IZPackAndLaunch4J

Mover al directorio $TOMCAT_HOME/webapps el archivo WAR que se genera en el paso anterior y renombrarlo a exist.war para mayor comodidad. Para evitarnos este complicado paso, en el CD también se distribuye el archivo WAR ya recompilado, así que si así se desea, basta con copiar este archivo a $TOMCAT_HOME/webapps.

Ahora es necesario *levantar* el servidor de Tomcat para poder ingresar a la interfaz de eXist y comenzar a almacenar los archivos XML en la base de datos para posteriormente generar el índice inicial. Para este paso es recomendable utilizar el plugin para eclipse *Sysdeo Tomcat* disponible en www.eclipsetotale.com/tomcatPlugin.html, pero también se puede hacer manualmente.

Una vez que Tomcat se esté ejecutando, ingresar a la dirección

http://localhost:8080/exist

y se desplegará la interfaz de administración de eXist. Ir al menú *Administration* a la izquierda y abajo de la pantalla y dar click en *Admin*. Se desplegará la pantalla de *Login*. Seguir las indicaciones para autentificarse.

En el menú seleccionar la opción *Browse Collections* y en la nueva pantalla que aparece a la derecha teclear el nombre de la colección que vamos a crear, que en este caso se

llamará *verificados*[1] y dar click en el botón *Create Collection*.

Ingresar a la colección recién creada y comenzar a agregar los archivos XML que siguen la DTD de JReports. Paraleleamente al desarrollo de JReports, en otro trabajo de tesis se estaba implementando una aplicación para generar los archivos XML que sirven como entrada a JReports, lo que facilitará muchísimo esta tarea.

Ya que se tienen almacenados todos los archivos en la colección, se debe generar el índice inicial y esto es muy sencillo desde eclipse, siguiendo los siguientes pasos:

1. Dar click derecho sobre el proyecto recién creado JReports, click en *Run As* y luego en *Open Run Dialog*.
2. En *Name* poner un nombre distintivo como *Indexer*.
3. Localizar el proyecto JReports usando el botón *Browse*.
4. Localizar la clase XMLDBUtil usando el botón *Search*.
5. Dar click en *Run*.
6. Entonces se generará el índice inicial con el vocabulario de los documentos que se encuentren almacenados en eXist.

[1]Por el momento JReports no permite seleccionar el nombre de la colección y por default la colección se llama verificados. Para cambiar el nombre se debe modificar el valor de la propiedad *textCollection* del archivo *xmldatabase-mapping.properties* en el paquete `reports.indexer` de JReports.

Apéndice C

Evaluación del Sistema

Dentro de la lista de documentos recuperados por un sistema de RI se encontarán *buenos* y *malos* documentos para la consulta del usuario. La evaluación de la calidad de los resultados obtenidos por el sistema, normalmente se mide en términos de la proporción de documentos relevantes encontrados en la lista, la posición de los documentos relevantes recuperados en relación con los no relevantes también recuperados y la proporción de documentos relevantes que no fueron recuperados.

En una situación ideal, un sistema de RI debe recuperar todos los documentos relevantes y únicamente los relevantes. Se diría entonces, que tal sistema tiene una gran calidad con una alta *precisión* y *exhaustividad*, las dos medidas de evaluación mencionadas en el capítulo 1. Lamentablemente, en la actualidad, los sistemas de RI sólo son capaces de recuperar algunos de los documentos relevantes de la colección junto con algunos no relevantes.

El problema radica en que el texto de las consultas no proporciona una completa especificación de la *necesidad de información* del usuario. Cada necesidad de información tiene una intención implícita que la consulta no puede describir completamente y, por lo tanto, no conocerá el sistema.

Por ejemplo, si el usuario desea buscar todos los documentos de la colección relacionados con tratamientos para enfermedades virales, es muy probable que el sistema

recupere documentos que no hablen sobre un tratamiento específico para este tipo de enfermedades, sino que únicamente mencionen a lo largo del documento los términos *tratamiento* o *enfermedades virales*. Estos documentos serán considerados como falsos positivos y el sistema los devolverá, a pesar de no ser relevantes para la necesidad de información del usuario. Es muy complicado escribir una mejor consulta que regrese únicamente los documentos relevantes.

El determinar la relevancia de un documento es un asunto subjetivo, pues se trata de una interpretación del lenguaje natural; sin embargo, es necesario realizar este *juicio* de relevancia de todos los documentos de la colección para poder medir estadísticamente la calidad del sistema.

Balance entre Precisión y Exhaustividad

Como se explicó, exhaustividad y precisión son dos medidas útiles para evaluar la calidad de los resultados devueltos por el sistema. Se podría decir que la precisión permite cuantificar la *utilidad* de la lista de resultados y la exhaustividad cuantifica la *completez* de la lista.

Un sistema de RI debe tener una alta exhaustividad para ser evaluado satisfactoriamente, por lo que lo más importante de un sistema de RI es ofrecer un incremento en la precisión sin sacrificar la exhaustividad. Por ejemplo, un buscador Web como *google*, generalmente tienen una buena exhaustividad, pero baja precisión, es decir, dentro de los resultados que presentan para una consulta se encuentran muchos documentos relevantes, pero también regresan demasiados documentos no relevantes.

Precisión en *n* documentos

La precisión y la exhaustividad como se han presentado hasta ahora presentan una

debilidad importante, no permiten evaluar si el sistema ha ordenado correctamente la lista de los documentos devueltos como resultado de acuerdo a su relevancia. Es decir, los documentos más relevantes deberían estar en las primeras posiciones de la lista regresada para una consulta. Con esta finalidad, de evaluar el orden de los documentos recuperados, se propuso como medida el cálculo de la precisión en diferentes *intervalos* de longitud *n* de la lista.

Por ejemplo, si todos los documentos en las primeras cinco posiciones de la lista son relevantes para la consulta y ninguno de los siguientes cinco es relevante, el sistema de RI tendría una precisión de 100 % en un intervalo de 5 documentos, pero una precisión de 50 % en un intervalo de 10 documentos. Este orden de la lista de resultados es muy bueno, pues todos los documentos relevantes se encuentran en las primeras posiciones, por encima de cualquiera no relevante.

Resultados para JReports

Para llevar a cabo la evaluación de cualquier sistema de RI, es escencial contar con una colección de prueba. En principio, estas colecciones de prueba contienen cientos o miles de documentos, pero al no contar con una colección de prueba en español, para evaluar JReports fueron utilizados los mismos documentos que se encuentran almacenados en eXist.

Los componentes de una colección de prueba son los siguientes:

- Corpus o conjunto de documentos
- Conjunto de necesidades de información
- Juicios de relevancia que relacionan las necesidades de información con los documentos del corpus

La evaluación de JReports se llevó a cabo con el uso de la herramienta *trec_eval*, un programa que proporciona la Conferencia de Recuperación de Texto (TREC) para evaluar

los resultados producidos por un sistema de RI usando los procedimientos de evaluación del Instituto Nacional de Normas y la Tecnología (NIST).

El programa *trec_eval* recibe como parámetros dos archivos:

qrels.txt Este archivo contiene los juicios de relevancia determinados manualmente por alguna persona. Su formato contiene 4 columnas, por ejemplo:

```
001 Q0 AGENTESETIOLOGICOS 0
001 Q0 AMALGAMACION 1
qid iter docno rel
```

donde **qid** representa el identificador de la consulta, **iter** es un entero, que es obligatorio, pero ignorado por trec_eval, **docno** es el identificador del documento y **rel** un entero (0 ó 1) indicando si el documento es relevante o no para la consulta en cuestión.

results.txt Salida producida por JReports para 4 consultas. Su formato contiene 6 columnas, por ejemplo:

```
001 Q0 AGRICULTURA 1 0.89891136 jreports
qid iter docno rank sim run_id
```

donde **qid**, **iter** y **docno** son como se describió en el inciso anterior, **rank** es un entero comenzando en 0 y representa la posición del documento en la lista de resultados, aunque también es ignorado por el programa y **sim** es el valor de similitud entre el documento y la consulta.

Para generar los archivos para *trec_eval* se ejecutaron 4 consultas con JReports sobre los documentos de la colección y para tales consultas se determinó la relevancia de cada documento.

Las consultas fueron las siguientes:

1. title:efecto OR keywords:gases
2. (keywords:aid OR keywords:"de los Estados Unidos")
3. (keywords:drogas OR keywords:gastroenterologia) OR (referencias:chagas OR referencias:gastroenterologia)
4. keywords:enfermedades OR keywords:virales

La siguiente tabla muestra los resutados arrojados por *trec_eval*. Cabe recalcar que la colección con la que se cuenta para este trabajo tiene únicamente 30 documentos, y obviamente estos resultados sirven únicamente como referente para el sistema y para describir brevemente la manera en la que se suelen hacer las evaluaciones de los sistemas de RI.

exhaustividad	0.8182
precisión	0.7105
precisión 5	0.8000
precisión 10	0.5750
precisión 15	0.4000

Tabla c.1: Exhaustividad, precisión y precisión en n de la evaluación.

Para comprobar como se degrada la precisión del sistema conforme se consulta la lista de documentos recuperados se suele graficar la precisión interpolada contra los 11 valores de exhaustividad entre 0.0 y 1.0. Por ejemplo, una exhaustividad del 50 % se refiere a la posición de la lista de documentos recuperados en que se ha encontrado el 50 % de los documentos relevantes. En otras palabras, cuantifica el número de documentos que el usuario debe consultar antes de encontrar un porcentaje determinado de documentos relevantes.

La curva de una gráfica *Precisión -vs- Recall* generalmente tiene una forma cóncava. La Figura c.1 muestra la gráfica para la evaluación realizada a JReports. Como se puede

observar, esta gráfica evidencía la compensación que existe entre precisión y exhaustividad, es decir, al aumentar la exhaustividad tendrá como consecuencia una cantidad mayor de documentos no relevantes dentro de la lista de recuperados, lo que reduciría la precisión. Por otro lado, para incrementar la precisión, generalmente disminuye la exhaustividad pues se pierden documentos relevantes.

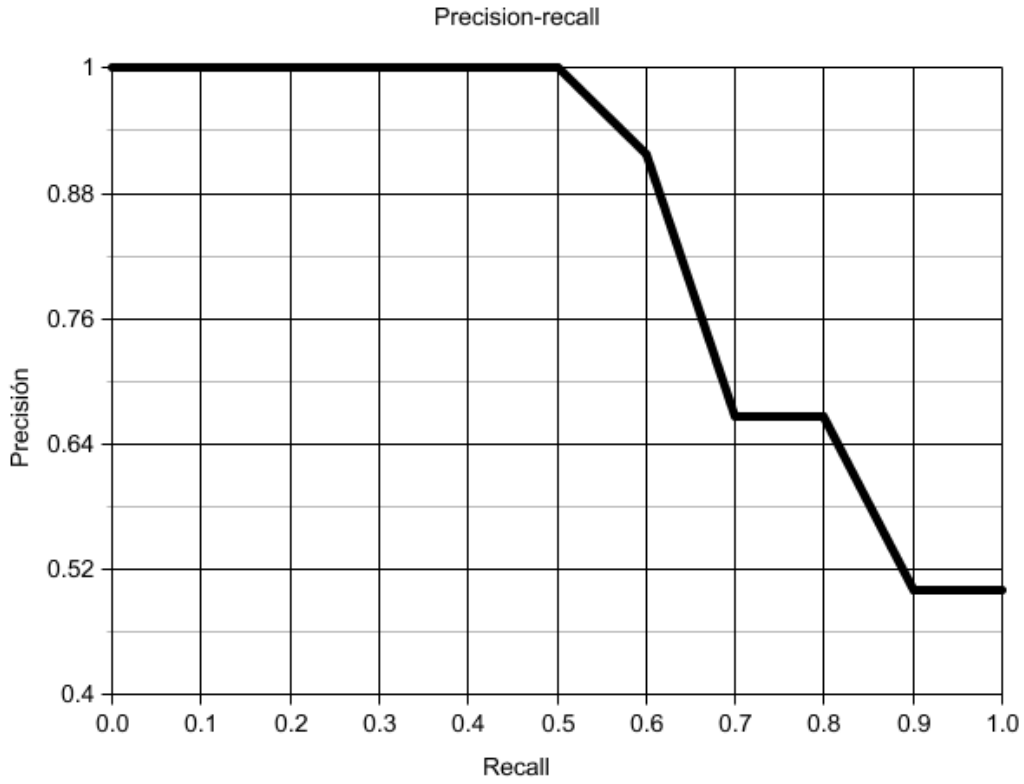

Figura c.1: Precisión-Recall de la evaluación.

Apéndice D

Trabajo Pendiente

En esta primera versión, JReports ya presenta todas las características que debe tener cualquier sistema de recuperación que se precie de serlo. Entre estas funcionalidades destacan:

- Asignación de relevancia para los resultados, con lo cual los documentos devueltos aparecerán en orden de relevancia.
- Eliminación de palabras comunes, que aunque tiene sus desventajas para ciertos sistemas de recuperación, en este caso, es útil, pues no serán muy comunes las consultas de frase que contengan estas palabras.
- La implementación del algoritmo de stemming de Snowball Porter para el español, de manera que las palabras se reducen a su raíz.
- Se incluye también un motor de búsqueda que se encarga de hacer sugerencias al usuario final del sistema, en caso de que los resultados tengan una baja relevancia. Este es uno de los enfoques de la tolerancia a errores ortográficos en sistsemas de RI.
- El sistema ya conjunta perfectamente las dos tecnologías que sustentan su arquitectura. Lucene y eXist trabajan conjuntamente de manera transparente para el usuario.

JReports es un módulo externo de eXist liberado como software libre, para cualquiera que esté interesado en el tema de la recuperación de información, en particular, recupera-

ción XML usando Lucene y eXist como software de apoyo.

Al ser libre bajo licencia GPL, cualquiera cuenta con las libertades básicas que esta licencia exige:

Libertad 0: La libertad de usar el programa, con cualquier propósito.

Libertad 1: La libertad de estudiar cómo funciona el programa, y adaptarlo a tus necesidades. El acceso al código fuente es una condición previa para esto.

Libertad 2: La libertad de distribuir copias, con lo que puedes ayudar a tu vecino.

Libertad 3: La libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie. El acceso al código fuente es un requisito previo para esto.

Entonces, para cualquiera que desee colaborar y mejorar JReports, se incluyen a continuación las siguientes sugerencias para comenzar:

- Estudiar la documentación que se encuentra en la página oficial de eXist, principalmente la forma en que se crean los módulos para eXist en Java.
  Clases: JReportsModule.java, SearcherFunction.java, PrintQueryFunction.java

- Estudiar la forma en como funciona Lucene y checar en el módulo cómo se implementa la creación del índice invertido y la recuperación de los documentos almacenados en eXist.
  Clases: Indexer.java, XMLDBUtil.java y LuceneFunctions.java, SpanishAnalyzer.java

- Estudiar el artículo Did You Mean: Lucene? para comprender como se adaptó esa implementación en este módulo.
  http://today.java.net/pub/a/today/2005/08/09/didyoumean.html?page=1

Hay muchas cosas que se pueden mejorar, pues esta es apenas una primera versión del sistema, además de que se pueden agregar bastantes funcionalidades nuevas. A continuación, lo que podría considerarse prioritario:

- La clase VitroQueryParser del paquete org.exist.xquery.modules.reports no la ocupo, pero servirá para permitir que el stemming no deje sin funcionar las consultas con comodines (expresiones regulares), así que hay que reemplazar el QueryParser por esta clase.

- Ofrecer más transformaciones de los documentos recuperados y el reporte final que se genera. Por el momento sólo se ofrece la transformación a XML y HTML, pero se pueden generar transformaciones a PDF, RTF, etc. mediante XSL.

- Agregar a la interfaz la posibilidad de que el usuario elija la colección donde se encuentran almacenados los documentos. Por el momento el nombre de la colección lo toma el sistema del archivo xmldatabase-mapping.properties.

- Si se quisiera trabajar con documentos en diversos idiomas, sería necesario generar analizadores y los índices para cada uno de ellos.

- Para que JReports sea un verdadero sistema de recuperación estructurado, es necesario agregar la funcionalidad de devolver en lugar del documento completo, el nodo seleccionado. Por ejemplo: devolver únicamente los títulos, las secciones, subsecciones, etc. Para mayor referencia buscar en la red el documento: RMIT INEX experiments: XML Retrieval using Lucy/eXist

Bibliografía

[1] Ricardo Baeza-Yates y Berthier Ribeiro-Neto. *Modern Information Retrieval*. ISBN: 0-201-39829-X. Addison-Wesley, Harlow, England, 1999.

[2] *Search Kit Programming Guide*. Apple Developer Connection.

[3] Christopher D. Manning, Prabhakar Raghavan y Hinrich Schütze, *Introduction to Information Retrieval*, ISBN: 0-521-86571-9. Cambridge University Press. 2008.

[4] Sitio especializado sobre el algoritmo de stemming de Porter (Snowball). Disponible en http://snowball.tartarus.org.

[5] *La evaluación en recuperación de la información*. Autor: Raquel Gómez Díaz Disponible en http://www.hipertext.net/web/pag238.htm.

[6] Sitio especializado sobre listas de stopwords y stemmers. Disponible en http://members.unine.ch/jacques.savoy/clef/index.html.

[7] Gospodnetic Otis y Hatcher Erik. *Lucene in Action*. ISBN: 1-932394-28-1. Manning Publications, 2005.

[8] Sitio oficial del proyecto de Apache Lucene. Disponible en http://lucene.apache.org/java/docs/index.html.

[9] Wiki con documentación adicional sobre Lucene. Disponible en http://wiki.apache.org/lucene-java.

[10] Lista de correos de Lucene. Disponible en http://mail-archives.apache.org/mod_mbox/lucene-java-user.

[11] Artículo sobre búsqueda tolerante con Lucene.
Disponible en http://today.java.net/pub/a/today/2005/08/09/didyoumean.html.

[12] Walmsley, Priscilla. *XQuery*. ISBN: 0-596-00634-9. O'Reilly Media Inc., 2007.

[13] Chamberlin Don, Draper Denise, et. al., *XQuery from the Experts: A Guide to the W3C XML Query Language*. ISBN: 0-321-18060-7, Addison Wesley Professional, Agosto 2003.

[14] *XML Path Language (XPath) 2.0*, Anders Berglund, et. al., W3C Recommendation, Enero 23 de 2007.
Disponible en http://www.w3.org/TR/xpath.

[15] XQuery 1.0: An XML Query Language S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, W3C Recommendation, Enero 23 de 2007.
Disponible en http://www.w3.org/TR/xquery.

[16] *XQuery Tutorial*.
Disponible en http://www.w3schools.com/xquery/default.asp

[17] *Power your mashups with XQuery*. Ning Yan.
Disponible en
http://www.ibm.com/developerworks/xml/library/x-xquerymashup/index.html

[18] Meier, W. (2003) *eXist: An Open Source Native XML Database*. Web, Web-Services and Database Systems. NODE 2002 Web and Database Related Workshops. 2593.

[19] Sitio oficial de eXist.
Disponible en http://exist.sourceforge.net/

[20] Böhme, T.; Rahm, E. *Supporting Efficient Streaming and Insertion of XML Data in RDBMS*. Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb), 2004.

[21] *eXist Javadocs*.
Disponible en http://exist.sourceforge.net/api/index.html

[22] Akmal B. Chaudri, Awais Rashid y Roberto Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. ISBN: 0-201-84452-4. Addison Wesley Professional, Marzo, 2003.

[23] *XML and Databases*.
Disponible en http://www.rpbourret.com/index.htm.

[24] Burnard Lou, *TEI Lite: Encoding for Interchange: an introduction to the TEI Revised for TEI P5 release*. Febrero 2006.
Disponible en
http://www.tei-c.org/release/doc/tei-p5-exemplars/html/teilite.doc.html.

[25] Sitio oficial de TEI (Text Encoding Initiative).
Disponible en http://www.tei-c.org/index.xml.

[26] *Agile Web Development with Rails*. Dave Thomas, David H. Hansson. The Pragmatic Bookshelf. 2006.

Índice Analítico