Un Lenguaje de Consultas sobre Estructura y Contenido en Bases de Datos de Texto

Gonzalo Navarro

Depto. de Ciencias de la Computación
Universidad de Chile
Blanco Encalada 2120 - Santiago - Chile
email: gnavarro@dcc.uchile.cl

Resumen

Las bases de datos textuales estructuradas son relativamente nuevas en el área de recuperación de información. No existe aun consenso acerca de cómo debería ser el modelo de estructuración. Las soluciones actuales se centran en los problemas o de expresividad o de eficiencia. Más aun, no hay un modelo formal y completo para analizar esta expresividad.

Nuestro objetivo es encontrar un modelo de estructuración y un lenguaje de consultas que sea suficientemente expresivo y eficientemente implementable. Para ello, se evaluaron las soluciones actuales, se definió un modelo con las características deseadas, se comparó su expresividad con los otros en términos teóricos y prácticos, se definieron algoritmos para implementarlo y se estudió su complejidad de peor caso en tiempo y espacio, y se desarrolló un prototipo para evaluar heurísticas y estudiar tiempos promedios en casos reales.

Este trabajo permite concluír que un lenguaje de consultas orientado a conjuntos y basado en operaciones sobre componentes estructurales cercanas de una o más jerarquías es suficientemente expresivo y eficientemente implementable. También presenta un marco para comparar expresividad, que da una idea de hasta dónde se puede enriquecer ésta sin degradar la performance. Por último, sugiere varias líneas de investigación futuras.

Abstract

Structured text databases are relatively new in the field of information retrieval. There is still no consensus on how the structuring model and the query language should be. Current approaches focus strongly either on expressivity or on efficiency issues. Moreover, there is no formal and complete foundation to analyze this expressivity.

Our goal is to find a structural model and a query language that is expressive and efficiently implementable. To achieve it, previous work was first evaluated, a model with the desired characteristics was defined, its expressivity was compared against the others, formally and practically, algorithms were defined and their worst-case space and time complexity analyzed, and a real prototype was developed, to evaluate heuristics and draw average running times.

This work leads to the conclusion that a set-oriented query language based on operations on nearby structure elements of one or more hierarchies is quite expressive and efficiently implementable. It also presents a framework to compare expressivity that gives an idea of up to where it can be enriched without degrading the performance. Finally, it suggests some research directions.

Este trabajo ha sido parcialmente financiado por los proyectos FONDECYT 1930765, 1940271 y 1950622.

1 Introducción

Las bases de datos textuales están recibiendo cada vez más atención, debido a sus múltiples aplicaciones: bibliotecas, automatización de oficinas, ingeniería de software, diccionarios automatizados, y en general cualquier problema basado en mantener y recuperar información textual [13, cap. 1].

El propósito de una base de datos textual es almacenar documentos de texto, estructurados o no. De modo que está compuesta de dos partes: contenido y estructura (si la hay). El contenido es el texto mismo, mientras que la estructura relaciona distintas partes de la base de datos con algún criterio.

El objetivo de cualquier sistema relacionado con recuperación de la información es ayudar a los usuarios de la base de datos a encontrar lo que buscan. Las bases de datos de texto no son como las relacionales [10], en las que la información está ya formateada y preparada para ser recuperada mediante "claves". La información está allí, pero no es fácil recuperarla. El usuario debe especificar lo que desea, ver los resultados, reformular la consulta, y así, hasta estar satisfecho con el resultado. Cualquier cosa que se haga para para ayudar a los usuarios a encontrar lo que buscan merece ser considerado.

Tradicionalmente, las bases de datos textuales han permitido a los usuarios buscar o en el contenido (palabras, frases, etc.) o en la estructura (por ejemplo, mirando el índice de un libro). Sin embargo, esto no es totalmente satisfactorio. Por ejemplo, una persona puede recordar que lo que quiere estaba escrito en itálicas, cerca de una figura que decía algo sobre "Chile". Buscar la palabra "Chile" puede ser una mala idea, como puede serlo buscar todos los textos en itálicas o todas las figuras del libro. Lo que realmente ayudaría sería un lenguaje que permitiera decir "quiero un texto en itálicas, cerca de una figura que contenga la palabra 'Chile' ". Esta consulta habla del contenido y de la estructura de la base de datos al mismo tiempo.

Mezclar el contenido y la estructura nos permite expresar consultas muy potentes, mucho más expresivas que las de cada mecanismo por separado. Usando un lenguaje que integre ambos tipos de consultas, podemos potenciar la calidad de recuperación de las bases de datos textuales. Sin embargo, ambos aspectos presentan sus propios desafíos.

Por un lado, la información almacenada como "contenido" no está formateada, sino en forma de lenguaje natural. Eso implica que ninguna metodología basada en información formateada (como el modelo relacional) es adecuada para extraer la información deseada; y que ninguna metodología que asuma objetos no interpretados y se base en sus atributos, también formateados (por ejemplo bases de datos multimedia [4]) es suficientemente potente para expresar el complejo modelo de información que el texto representa. La información debe ser extraída del texto, pero no en forma rígida (véase también [28]).

Por otro lado, no hay consenso acerca de cómo debería ser el modelo de estructuración de una base de datos. Hay varios posibles modelos, que van desde no estructurar en absoluto hasta complejas redes de relaciones. Decidir usar un modelo de estructuración implica elegir también qué clase de consultas se pueden hacer a la estructura. Hay un compromiso entre proveer un modelo de estructuración muy poderoso y poderlo implementar eficientemente.

El objetivo de este trabajo es presentar un modelo para estructurar y consultar bases de datos textuales, que sea suficientemente expresivo y eficientemente implementable. No existe en este momento, que sepamos, ninguna solución que satisfaga ambos objetivos. Para comparar expresividad, es necesario definir un marco en el que ubicar todos los modelos, que en este momento no existe. Para comparar eficiencia, es necesario describir los algoritmos que construyen los índices y procesan las consultas, analizar su complejidad en términos de espacio y tiempo, y obtener resultados experimentales a partir de un prototipo real.

La Tesis está organizada de la siguiente forma. En el Capítulo 2 se repasan los trabajos relacionados con el problema. En el Capítulo 3 se presenta informalmente nuestro modelo. En el Capítulo 4 se analiza su expresividad. En el Capítulo 5, se definen y analizan los algoritmos y estructuras de datos para implementar el modelo. En el Capítulo 6 se presenta una implementación real y se obtienen valores reales de tiempos promedios. Finalmente, en el Capítulo 7, se presentan las conclusiones y futuras líneas de investigación.

En este resumen se presenta una sección por cada capítulo del trabajo. Distintos aspectos de este trabajo has sido publicados en [27] (descripción del modelo), [3] (survey de modelos) y [26] (análisis de expresividad), y se han enviado para publicar una exposición de los algoritmos y una versión journal exponiendo el modelo.

2 Trabajos Relacionados

Cubriremos aquí los trabajos relacionados con el problema de consultar una base de datos textual. Veremos primero los modelos tradicionales y luego las ideas más novedosas. Una comparación más detallada se puede ver en [3].

2.1 Modelos Tradicionales

En esta sección se presentan cuatro líneas principales en la solución del problema, que constituyen los modelos "tradicionales" para encararlo. Estos modelos son: el modelo relacional [10] y sus extensiones a texto [33, 11], el modelo tradicional de recuperación de la información [31, 30, 32] (llamado "RI tradicional"), el modelo de texto completo [13, 2, 25], e hipertextos y lenguajes orientados a objetos [6, 17, 34, 7, 22].

Si bien estos modelos no son lo suficientemente potentes como para extraer la información que queremos de las bases de datos de texto, resuelven otros problemas que los lenguajes puramente orientados a la estructura no resuelven en general (por ejemplo, tuplas y joins, atributos, etc.).

En [28] se sugiere que es mejor poner una capa integrando una base de datos tradicional con una textual que pretender diseñar una base de datos que incluya todas las características. Por ejemplo, en [8] se muestra que las consultas relacionadas con la estructura se manejan mejor con una maquinaria que sepa acerca de la semántica de las jerarquías, que si se usa un lenguaje de base de datos orientado a objetos cualquiera.

Nosotros nos basamos en esta idea. Diseñamos un lenguaje orientado a explotar las características textuales y estructurales. Otras cosas, como tuplas y joins, deben añadirse mediante integrar este lenguaje con otro orientado a esa clase de operaciones, por ejemplo una base de datos relacional.

Por otro lado, no resolvemos el problema de unir consultas de estructura con otras que involucran operaciones como ordenamiento por relevancia (por ejemplo, obtener las secciones donde la palabra "computador" es relevante). Véanse en [28] algunas ideas sobre este tema.

2.2 Nuevos Modelos

Estos se caracterizan por dos hechos importantes: la estructura de los documentos es más o menos jerárquica (lo cual es razonablemente expresivo) y el lenguaje de consultas integra en una sintaxis uniforme las consultas sobre texto y sobre estructura. En el trabajo se cubren varios modelos, que aquí sólo mencionamos brevemente:

Modelo híbrido [2]: Combina campos (zonas del texto, como "título" o "apéndices") con un sistema tradicional de texto completo. Los campos se utilizan para restringir las búsquedas al texto dentro de algún campo dado. Toda respuesta a una consulta consiste de un conjunto de documentos y un conjunto de calces dentro de los documentos (donde se encontró lo que se pedía). Este es uno de los modelos más simples dentro de los que permiten mezclar consultas sobre contenido y estructura. Gracias a esa simplicidad, puede implementarse muy eficientemente.

PAT expressions [29]: En este modelo la estructura se define principalmente mediante expresiones de texto, de modo que en un texto marcado al estilo SGML se puede decir que una sección es lo que va desde donde hay un `<seccion>` hasta donde hay un `</seccion>` en el texto. Esto permite indizar en forma flexible, pero impone una estructura plana en los índices (una sección no podría contener otra sección). Gracias a estas restricciones, el modelo también puede implementarse muy eficientemente.

Listas solapadas [5]: Es una evolución a partir de PAT expressions, que resuelve los problemas semánticos de aquél (relacionados con la posibilidad de que surjan solapamientos de resultados en las operaciones) mediante permitir que los segmentos de una estructura solapen en el texto, aunque no que se contengan. Se obtiene mucha mayor expresividad y una implementación igualmente eficiente.

Listas de referencias [24]: Este modelo permite expresar, además de relaciones jerárquicas, conexiones estilo hipertexto entre los documentos y atributos, de modo que sólo se considera de él la parte relacionada con la estructura. La respuesta a una consulta es una lista de referencias (punteros) a la base de datos, de modo que se puede unir en forma elegante a listas de referencias obtenidas por mecanismos hipertextuales. Si bien permite una estructura jerárquica, las respuestas deben ser planas, y además del mismo tipo de constructor (por ejemplo, sólo secciones).

Parsed strings [15]: Este es en realidad un modelo de manipulación de datos, más que de consulta, de modo que se lo deja de lado al momento de comparar expresividad. Se basa en ver la base de datos jerárquica como descrita por una gramática libre de contexto (que representa el esquema de la base). Las operaciones realizan transformaciones sobre ése árbol, obteniéndose así un lenguaje muy flexible y potente, pero de implementación muy ineficiente.

Matching de árboles [21]: Este modelo consta de una única primitiva: la consulta es un árbol, y la respuesta a la consulta son todos los puntos de la estructura (árbol) de la base de datos donde la consulta calza, en el sentido de ser la raíz de un subárbol de la base donde la consulta puede ser embebida respetando los labels, las relaciones de descendencia y (opcionalmente) de orden entre hermanos. Esto se enriquece mediante un lenguaje estilo Prolog, con variables lógicas y búsqueda exhaustiva de todas las combinaciones de calces, para obtener un lenguaje de consulta potente, pero de difícil implementación [20]. Todo lo que involucre variables lógicas o matching sin respetar orden es NP-Completo, y fuera de eso, si bien es polinomial, debe recorrerse toda la base de datos buscando los calces.

3 Nuestro Modelo

3.1 Conceptos Fundamentales

En esta sección se exponen las ideas generales de cómo debería ser el modelo de estructuración y el lenguaje de consultas para alcanzar los objetivos de expresividad y eficiencia al mismo tiempo. Luego se define un modelo siguiendo estos principios. Una exposición completa del modelo se puede ver en [27].

Como queremos definir un lenguaje totalmente composicional, podemos considerar a las expresiones de consulta como árboles sintácticos, donde los nodos representan operaciones a realizar y sus subárboles los operandos.

Un primer punto es que queremos un lenguaje orientado a conjuntos, porque se han mostrado exitosos en otras áreas (como el modelo relacional), y porque si hay que extraer el conjunto completo de las respuestas, es posible encontrar algoritmos que recuperen los elementos a un costo muy bajo por elemento.

Para obtener el conjunto de respuestas queremos además evitar una solución "top-down", donde las respuestas se buscan en todo el árbol. Preferiremos una estrategia "bottom-up": la idea es poder encontrar rápidamente un conjunto pequeño de candidatos para las respuestas, y de ahí eliminar los que no cumplen con el criterio de búsqueda.

Nuestras operaciones toman y devuelven entonces conjuntos de nodos. Estos son subconjuntos del conjunto de todos los nodos del árbol de la base de datos. El único momento en que se resuelve una consulta de matching de texto o se designa por su nombre a un elemento de la estructura (por ejemplo, "sección") es en las hojas del árbol sintáctico de la consulta. Estas hojas deben ser resueltas con alguna clase de índice, y convertidas en un conjunto de nodos. De ahí en más, todos los operadores reciben y devuelven conjuntos de nodos (véase la Figura 1).

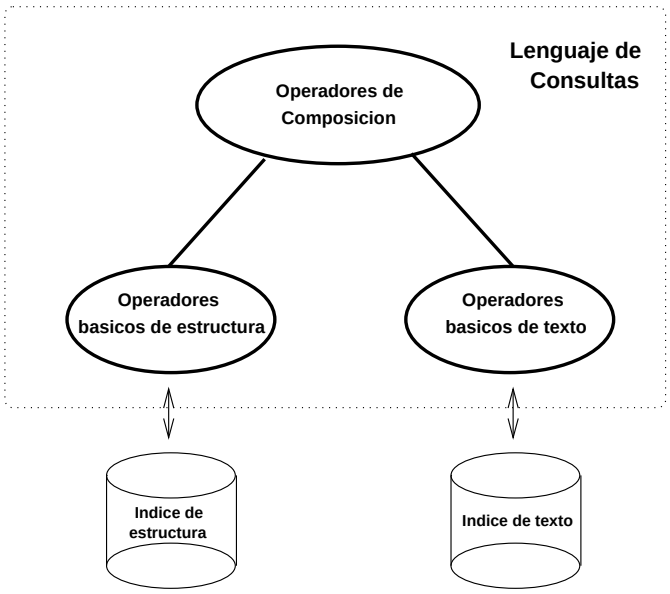

Figura 1: Diagrama inicial de cómo opera nuestro modelo.

Con ésta idea, usamos índices para recuperar los nodos que satisfacen una consulta de matching

de texto, o los nodos de una componente estructural dada (o "constructor"). Esos conjuntos deben ser obtenidos sin recorrer la base de datos completa.

Una vez que convertimos las hojas de la consulta en conjuntos, todas las otras operaciones toman esos conjuntos y los operan. Normalmente un conjunto va a contener los candidatos para el resultado de la operación. Obsérvese que de esta forma nunca debemos recorrer la estructura completa.

Por otro lado, estas operaciones de conjuntos pueden ser tan caras como recorrer la base de datos. Para evitar esto se imponen dos condiciones: que los nodos tengan un *segmento* (par de números representando una porción contigua del texto) asociado, y que las operaciones trabajen en función de los nodos con segmentos cercanos de ambos operandos.

La asociación de nodos con segmentos permite usar estructuras de datos eficientes para disponer los nodos basándose en los segmentos que representan (formando un árbol, por ejemplo). En otros modelos [21, 15], hay una conexión débil entre los nodos y el segmento que poseen en el texto, de modo que necesitan recorrer todo el árbol para encontrar lo que necesitan.

El operar con nodos cercanos permite aplicar técnicas de "dividir y reinar", para reducir drásticamente el área en la cual buscar los nodos a operar. De este modo se evitan las operaciones de "todos contra todos" como en [20], y obtener una implementación eficiente. Este esquema nos permite tener más de una estructura jerárquica, si son independientes.

Por otro lado, debemos mostrar que muchos operadores interesantes son del tipo que permitimos, en el sentido de operar con nodos cercanos y sólamente necesitar de la identidad de los nodos y del segmento asociado.

3.2 Modelo de Datos

Una base de datos textual se compone de dos partes:

- Texto, que se ve como una (larga) secuencia de símbolos. Que este texto se almacene como se lo ve, o que sea filtrado para ocultar marcado o componentes sin interés, no es importante para el modelo, dado que sólo usamos la visión lógica del texto. Además, los símbolos pueden ser caracteres, palabras, o lo que sea.

- Estructura, que se organiza como un conjunto de jerarquías independientes (ortogonales). Cada jerarquía tiene sus propios tipos de nodos, y las áreas cubiertas por los nodos de distintas jerarquías pueden solapar, si bien esto no puede ocurrir dentro de una misma jerarquía.

Nuestro modelo de utilización es que el texto es estático, las jerarquías se construyen sobre él una vez (o esporádicamente) y las consultas son frecuentes. Esto se tiene en cuenta a la hora de elegir qué operaciones implementar eficientemente.

Cada jerarquía (o árbol) se llama *view*, que como su nombre sugiere es una forma independiente de ver el texto (por ejemplo, capítulos/secciones/párrafos o páginas/líneas). Se entiende que la raíz de la view contiene toda la base de datos.

Cada view tiene una serie de *constructores*, que denotan tipos de nodos en el árbol correspondiente (por ejemplo páginas, capítulos, etc.). Los conjuntos de constructores de dos views distintas son disjuntos.

Cada nodo del árbol correspondiente a una view tiene un constructor asociado, y un *segmento*, que como se explicó antes es una porción contigua del texto. El segmento de un nodo debe incluír a los de sus hijos en el árbol (aunque no tiene por qué ser una inclusión estricta).

Cualquier conjunto de segmentos disjuntos se puede ver como perteneciente a una *view de texto* especial, en la que los nodos pertenecen a un *constructor de texto* y tienen una estructura plana (todos los nodos en el segundo nivel del árbol). Así, la *view de texto* tiene un nodo por cada posible segmento del texto. La idea es usar esa view para modelizar consultas de matching de texto, a las cuales imponemos que retornen una secuencia de segmentos disjuntos. Esta imposición no es esencial, dado que perfectamente podría generarse una jerarquía. Sin embargo, asumimos que la estructura es plana para algunas operaciones sobre consultas de texto, las que no serían aplicables en otro caso.

3.3 Lenguaje de Consultas

En esta sección definimos un lenguaje de consultas que opera sobre la estructura definida previamente, incluyendo también consulta por contenido. No pretendemos definir un lenguaje monolítico ni completo, dado que los requerimientos varían mucho según la aplicación. Incluír todas las alternativas en un mismo lenguaje de consulta lo haría demasiado complejo. En cambio, mostramos una cantidad de características que pueden ser útiles, de modo de seleccionar un subconjunto apropiado a cada necesidad.

Lo que es importante en este conjunto es que las operaciones sólo necesiten operar con nodos cercanos, y baciones en la identidad de los nodos y el segmento que les corresponde. Las operaciones que no cumplen con eso difícilmente se puedan implementar en forma eficiente.

Definimos un álgebra de operadores sobre conjuntos de nodos. Cada consulta retorna un conjunto de nodos del árbol de alguna jerarquía (no un subárbol), de modo que no se puede manipular subárboles ni ver las consultas como "views" (en el sentido relacional). En este sentido, el modelo es más débil que [15].

Decidimos no unir nodos de distintas views en un mismo resultado de consulta por dos razones: por un lado no queda claro, siendo las views formas independientes y distintas de ver el texto, si esto tiene algún sentido (por ejemplo, "capítulos o páginas con una figura"); y por otro, la implementación es mucho más eficiente si cada conjunto presenta una jerarquía estricta. En [5] se prefiere en cambio permitir solapamientos pero no anidamientos en las respuestas.

Si bien no es posible recuperar subárboles, el álgebra permite seleccionar nodos basándose en su *contexto* en la jerarquía, como [21]. Obsérvese que este lenguaje no es para usuarios finales, sino una representación operacional de las consultas.

En la Figura 2 se esquematizan las operaciones. Hay operaciones básicas de matching de texto y de obtención de constructores, operaciones para combinar los conjuntos resultantes de otras operaciones, para seleccionar elementos basándose en criterios de inclusión, cercanía y en relaciones de ascendencia y descendencia directa en la view. A continuación se presenta una descripción de las operaciones más comunes, en la Tesis se detallan varias más:

Sublenguaje de matching: Es el único que accede al contenido de la base de datos, y es ortogonal al resto del lenguaje.

Matches: El lenguaje de matches genera un conjunto de segmentos no solapados ni anidados, que se introducen en el modelo como pertenecientes a la *view de texto*, como se explicó. Por ejemplo `'computador'` genera el árbol plano de todos los segmentos donde esa palabra aparece en el texto. Obsérvese que el lenguaje de matches puede permitir expresiones mucho más complejas (cercanía, expresiones regulares, matching con errores, etc.).

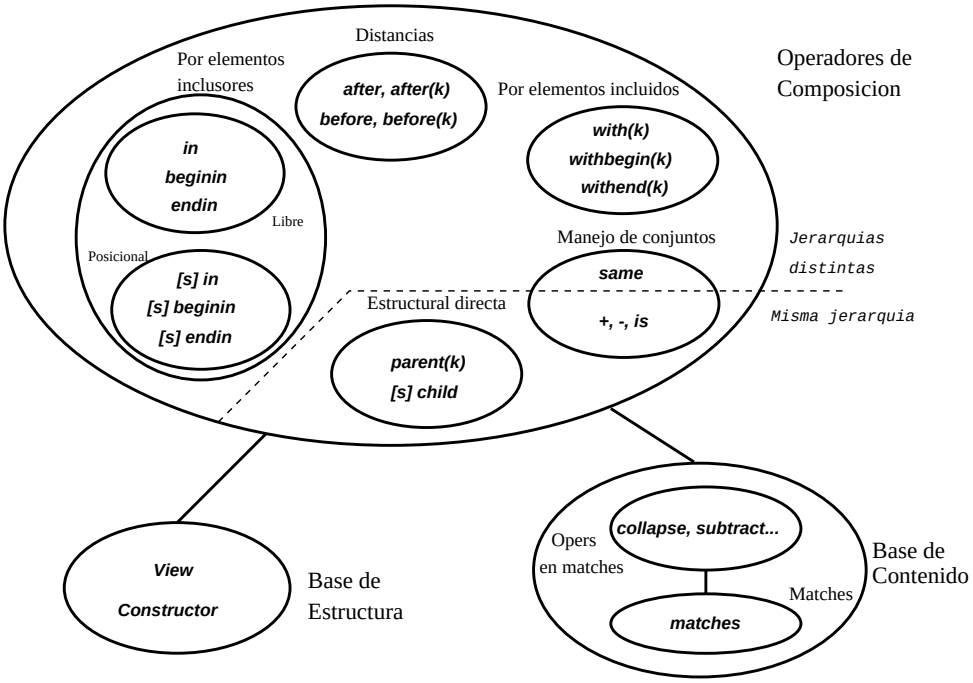

Figura 2: Las operaciones de nuestro modelo, clasificadas por tipo.

Operaciones sobre matches: Son aplicables sólamente a subconjuntos de la view de texto, y hacen transformaciones a los segmentos. Este punto y el anterior se consideran conjuntamente como el mecanismo para generar consultas de matching de texto, y no restringimos el lenguaje a ningún sublenguaje específico para esto. En la Tesis mostramos una cantidad de buenos candidatos para este conjunto, aquí incluímos sólo un ejemplo:

M collapse M': Es el conjunto de segmentos obtenidos al superponer ambos conjuntos de segmentos; si resulta un solapamiento se unen las dos áreas. Por ejemplo, `'Juan Perez' collapse 'J. Perez'` obtiene las apariciones de Juan Pérez en el texto, en sus dos formas.

Operadores básicos de estructura: Son la otra clase de hojas del árbol sintáctico de la consulta, que se refieren a componentes estructurales.

Nombre de constructor: (consultas "**Constr**"). Es el conjunto de todos los nodos de un constructor dado. Por ejemplo `capitulo` recupera todos los capítulos de un libro.

Nombre de view: (consultas "**View**"). Es el conjunto de todos los nodos de la view dada. Por ejemplo, `Formateado` recupera la view completa relacionada con aspectos de formateado. El mismo efecto se obtiene sumando (operador "+") todos los constructores de la view.

Operadores incluído-en: Seleccionan elementos del primer operando que están, en algún sentido, incluídos en el segundo.

Inclusión libre: Seleccionan cualquier elemento incluído, por ejemplo:

P in Q: Es el conjunto de todos los nodos de P incluídos en un nodo de Q. Por ejemplo, `citacion` **in** `tabla` recupera todas las citas hechas desde dentro de una tabla.

Inclusión posicional: Seleccionan sólo los elementos incluídos en una posición dada. Para definir posición, sólo se tiene en cuenta el top-level de los elementos incluídos en cada inclusor. Por ejemplo:

[s] **P in Q:** Lo mismo que **in**, pero sólo califican los nodos que descienden de un nodo de Q en la posición (de izquierda a derecha) considerada en s. Para linealizar la posición, para cada nodo de Q se considera sólo el top-level de los nodos de P no disjuntos con el nodo de Q, y los que solapan se descartan junto con sus descendientes. El lenguaje para expresar posiciones (valores de s) es también independiente. Consideramos que expresar uniones finitas de $i..j$, $último{-}i..último{-}j$, y $i..último{-}j$ alcanza para la mayoría de los propósitos. Por ejemplo, `[3..5]` `seccion` **in** `capitulo` recupera la 3ra, 4ta y 5ta sección de todos los capítulos. Si las secciones incluyeran otras secciones, sólo se considerarían las secciones top-level.

Operadores inclusores: Seleccionan los elementos del primer operando que incluyen en algún sentido elementos del segundo, por ejemplo.

P with(k) Q: Es el conjunto de nodos de P que incluye al menos k nodos de Q. Si (k) no está presente, se asume 1. Por ejemplo, `seccion` **with**(5) `'computador'` selecciona las secciones en las que la palabra "computador" aparece 5 veces o más.

Operadores de estructura directa: Seleccionan elementos del primer operando basados en criterios estructurales directos, es decir por relaciones de padre/hijo directos en el árbol de la view.

[s] **P child Q:** Es el conjunto de nodos de P que son hijos (en el árbol de la view) se algún nodo de Q, en una posición considerada en s (o sea, s-ésimos hijos). Ambos conjuntos deben ser de la misma view, que no puede ser la view de texto. Si $[s]$ no está presente, se asume $1..último$. Por ejemplo, `titulo` **child** `capitulo` recupera los títulos de todos los capítulos (y no títulos de secciones dentro de los capítulos).

P parent(k) Q: Es el conjunto de nodos de P que son padres (en el árbol de la view) de al menos k nodos de Q. Ambos conjuntos deben ser de la misma view, que no puede ser la view de texto. Si (k) no está presente, se asume 1. Por ejemplo, `capitulo` **parent**(3) `seccion` selecciona capítulos con 3 o más secciones.

Operadores posicionales: Seleccionan del primer operando los elementos que están a una cierta distancia de algún elemento del segundo operando, bajo ciertas condiciones adicionales. Por ejemplo:

P after/before Q (C): Es el conjunto de nodos de P cuyos segmentos empiezan/terminan después/antes que el fin/comienzo de un segmento de Q. Si hay más de un candidato de P para un mismo nodo de Q, se considera el más cercano (si están a la misma distancia, entonces uno de ellos incluye al otro y se elige el más alto). Para que un nodo de Q se considere candidato para un nodo de P, el nodo minimal de C que lo contiene debe ser el mismo que el del nodo de P, o no existir en ambos casos. Por ejemplo, `tabla` **after** `figura` (`capitulo`) recupera las tablas más cercanas a una figura que las precede, dentro del mismo capítulo.

Operadores de manipulación de conjuntos: Manipulan ambos operandos como conjuntos, implementando unión, diferencia, intersección, etc. Ejemplos:

P – Q: Es el conjunto diferencia entre P y Q. Ambos deben ser de la misma view, y esta view no debe ser la de texto. Por ejemplo, `capitulo` – (`capitulo` **with** `figura`) son los capítulos sin figuras.

P same Q: Es el conjunto de nodos de P cuyo segmento es el mismo de algún nodo de Q. P y Q pueden ser de diferentes views. Por ejemplo, `titulo` **same** `'Introduccion'` recupera los títulos que dicen (exactamente) "Introducción".

En la Figura 3 se puede ver un ejemplo de una consulta.

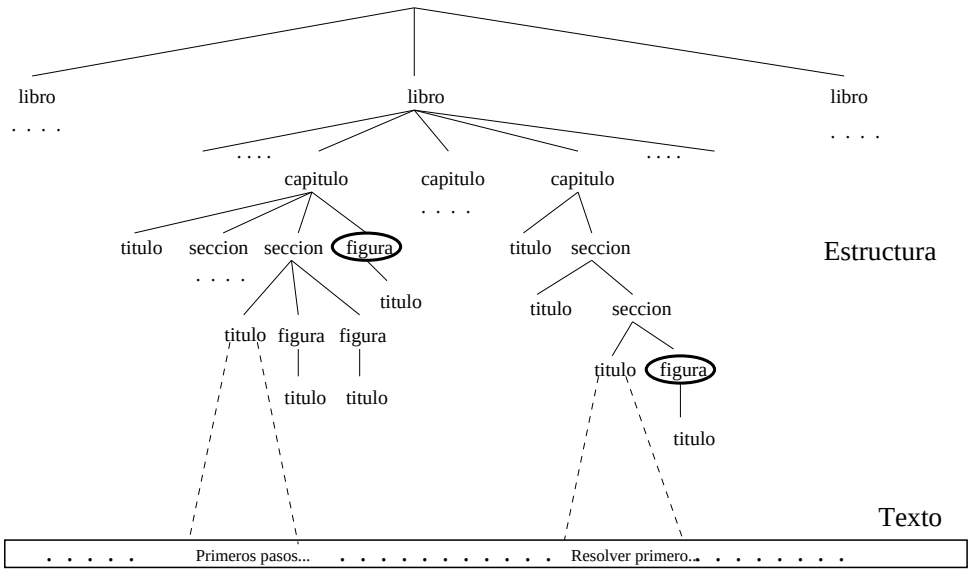

Figura 3: Efecto de la consulta `[ultimo] figura` **in** (`capitulo` **with** (`seccion` **with** (`titulo` **with** `'primer*'`))). Los círculos destacados son los nodos seleccionados.

3.4 Una Arquitectura de Software

En esta sección se describe una posible arquitectura de software para un sistema basado en nuestro modelo.

Los usuarios interactúan con el sistema vía una interfaz, en la cual definen lo que quieren en algún lenguaje amigable. Esa interfaz debe entonces convertir esa consulta en nuestro árbol sintáctico, es decir en el lenguaje que se ha definido. El árbol es entonces enviado al módulo de consulta.

Este módulo optimiza la consulta y genera un plan de acción inteligente para evaluarla (linealiza el árbol en una secuencia de operaciones a realizar). Las hojas del árbol de la consulta involucran extraer componentes de la jerarquía por su nombre (constructores) y expresiones de matching de texto. Las primeras se resuelven accediendo al índice de estructura para obtener el conjunto completo de nodos de ese constructor (un conjunto de identificadores de nodos y sus segmentos).

Las segundas se envían al módulo de texto, que retorna una lista de segmentos correspondientes a las porciones del texto que calzaron. De ahí en adelante, el resto de las operaciones se efectúan internamente, hasta que se entrega el resultado final (un conjunto de nodos) a la interfaz.

La interfaz está a cargo de la visualización los resultados. Para ello debe acceder al contenido de la base de datos, a las porciones del texto indicadas por los segmentos a mostrar. Esto se hace a través de un pedido al módulo de texto, dado que sólo él sabe cómo acceder al texto. Este módulo es el único responsable de de la visión que tenemos sobre el texto, y el que puede filtrar marcas y stopwords, reemplazar sinónimos, etc.

Véase en la Figura 4 un diagrama de cómo sería un sistema basado en este esquema. La "capa documental" está pensada para soportar un manejo de documentos más sofisticado, como colecciones de documentos. Traslada todas las referencias a colecciones (como las vería el usuario) a referencias a documentos (como están indizados en niveles inferiores).

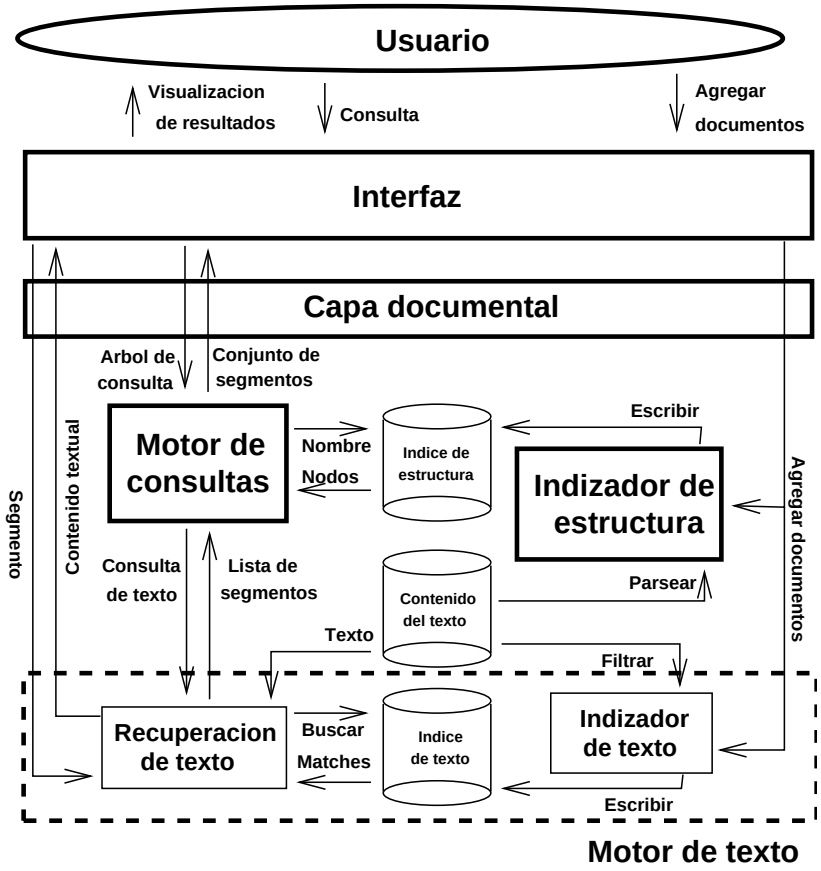

Figura 4: La arquitectura de un sistema según nuestro modelo. Las flechas muestran el proceso de resolver una consulta (lado izquierdo) o de incorporar documentos (lado derecho).

La Figura también muestra el proceso de indización. Si el módulo de texto está completamente separado, puede haber dos procesos de indización. Uno de ellos (realizado en el módulo de texto) indiza el texto para responder a preguntas de matching de texto. El otro extrae la estructura de alguna forma a partir del texto (parsing, reconocimiento de marcas, etc.), y crea el índice de estructura, que es accedido más tarde por el módulo de consultas. De hecho, ambos indizadores

deben colaborar, pues el marcado usado por el segundo para reconocer la estructura (si lo hay) debe ser filtrado por el primero cuando se presenta el texto a las capas superiores.

4 Expresividad

En este capítulo de la Tesis se estudia la expresividad que tiene nuestro modelo en relación con otros similares. Un análisis más detallado de la expresividad se puede ver en [26].

Lo primero que se hace es definir formalmente nuestro modelo de datos y las operaciones, de modo de tener una definición precisa para poder compararlo con otros.

Luego se procede a comparar formalmente el modelo con cada uno de los estudiados antes, salvo *p-strings*, que es un modelo de manipulación más que de consulta. Esta comparación procede mediante definir mapeos que lleven de expresiones de un lenguaje a expresiones del otro. De esta forma se puede tener una idea de cuáles son las cosas de un lenguaje que pueden ser representadas en el otro, y cuáles no. Otro parámetro interesante es cuán largas pueden resultar las consultas traducidas con respecto a las originales, y si se necesita más información en los índices.

De esta comparación de nuestro modelo con cada uno de los otros, se concluye que nuestro modelo es superior en la mayoría de los aspectos al modelo híbrido [2], a PAT expressions [29], a listas solapadas [5] y a listas de referencias [24], mientras que matching de árboles [21] es superior en general al nuestro.

De esta comparación surgen ciertos criterios acerca de cómo debería ser analizado un lenguaje de este tipo. Se define una metodología para analizar estos lenguajes, que se subdivide en tres aspectos: mecanismo de estructuración, consulta por contenidos y consulta por estructura. La metodología se ve como una serie de "preguntas" que hay que hacerse a la hora de medir la expresividad de un lenguaje de este tipo.

A partir de considerar los modelos similares con esta metodología se obtiene un diagrama donde se muestran los temas más importantes a considerar como puntos, y los modelos como conjuntos que contienen los puntos si es que los incluyen adecuadamente (obsérvese que se trata de consideraciones, hasta cierto punto, informales). Con ésto se puede apreciar cuánto de lo importante incluye cada modelo (Figura 5).

5 Implementación

En esta sección se describen brevemente las ideas expuestas en el trabajo para implementar el modelo, y la eficiencia resultante.

Dado que el lenguaje es composicional, el problema es, dados los conjuntos de nodos resultantes de las sub-operaciones, cómo obtener el conjunto resultante de la operación a aplicar.

Una primera y fundamental disyuntiva es si usar evaluación completa ("full") o retardada ("lazy"). Evaluación full significa, antes de aplicar el operador actual, obtener completos los conjuntos sobre los que operar, y luego generar completo el conjunto resultante. Evaluación lazy significa tratar de obtener el resultado del operador más alto elemento por elemento, donde la extracción de cada elemento genera la necesidad de obtener elementos de los operandos, y así hasta las hojas, que van evaluando lo que se les va pidiendo. De esta forma, sólo la parte necesaria de los conjuntos resulta evaluada.

Considérese, por ejemplo, la consulta "`seccion` **with**(10) `'Computador'` ". Si, en vez de obtener toda la jerarquía de secciones y luego descartar las que no contienen 10 veces la palabra "Computador", se obtienen primero las secciones top-level, se descartan las que no contienen la

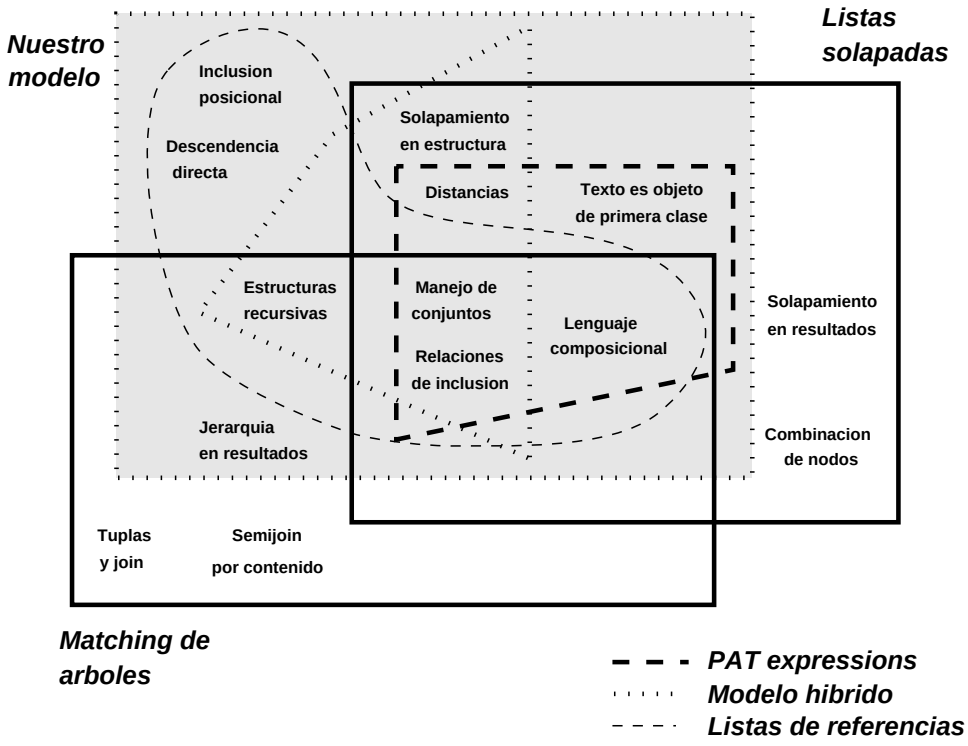

Figura 5: Representación gráfica de la comparación entre modelos.

palabra 10 veces y sólo se expanden las que sí la contienen, nos ahorramos de expandir inútilmente las secciones que resultan eliminadas en el primer paso.

Por otro lado, la evaluación lazy puede forzarnos a usar algoritmos menos eficientes, de modo que no es inmediato cuál alternativa es mejor. Es se determina experimentalmente.

Para evaluación full, se proponen dos técnicas distintas, que llamamos *search-operate* (buscar y operar) y *merge* (mezclar). La última se modifica para obtener evaluación lazy.

Finalmente, se pueden usar distintas clases de índices, algunos soportando mejor ciertas operaciones que otros, al costo de cierto requerimiento de espacio.

5.1 Indización

En esta sección de la Tesis se estudian distintas alternativas para implementar los índices. Todos los índices representan, de alguna forma, al árbol de la view. Dado que la view es estática, este árbol se mapea sobre un arreglo, nivel por nivel, de modo de ahorrar espacio de punteros y de algunas medidas que resultan implícitas si se pone nivel por nivel. La estructura de datos, cualquiera que sea, debe permitir hallar eficientemente el hermano y el primer hijo de cualquier nodo.

Primeramente se considera un índice completo, es decir, que permita realizar eficientemente todas las operaciones descritas en el modelo. Este índice ocupa 7 palabras por nodo, por lo cual deben estudiarse formas de mantenerlo en el disco. El problema con el disco es que si se almacena el arreglo de la view completo puede realizarse mucho seek para recuperar los nodos de un constructor. Para solucionar esto se propone tener un puntero al próximo elemento en el disco del mismo constructor (reemplazando otro puntero que se revela prescindible en la práctica),

o tener un archivo separado por constructor. Esta última técnica es mejor para consultas tipo **Constr**, pero peor para las tipo **View**. Se considera que las primeras son mucho más comunes. Finalmente, algunas consideraciones de reindización indican que sería beneficioso tener un archivo separado para cada nivel del árbol (en cualquier esquema).

Otro caso en que se puede acceder en forma aleatoria al índice en disco es para consultar si un nodo dado es padre directo de otro (para operaciones **parent** y **child**). Estos accesos aleatorios se pueden evitar mediante determinar sintácticamente (en la consulta) qué nodos pueden llegar a necesitar información de padre/hijo, y leer esa información al momento de resolver las hojas, cuando se leen esos nodos del disco y almacenar esa información no cuesta nada.

Si se tiene que no puede haber dos segmentos iguales en la jerarquía y se descartan las operaciones **parent** y **child**, es posible obtener un índice de sólo dos palabras por nodo, almacenando un arreglo con las posiciones iniciales de los segmentos y otro con las finales, ordenadas. Este índice no es bueno para evaluación lazy, porque obliga a expandir todo un subárbol para obtener sólo los hijos de un nodo (la operación normal de evaluación lazy). Para este caso es preferible eliminar información en el esquema de índice completo con un arreglo por constructor, para obtener un índice de tres palabras por nodo.

Para tener una idea de los tamaños involucrados, si se indiza código C con todos los detalles finos de parsing, un archivo de 1 Mb de código daría lugar a una estructura de 300 mil nodos, o sea 8 Mb para el índice completo y 2.3 Mb para el parcial. En cambio, si se indiza la estructura de un libro normal hasta los párrafos, un libro de 300 páginas (aproximadamente 1 Mb de texto) puede dar lugar a unos 1000 nodos, necesitando 28 Kb el índice completo y 8 Kb el parcial.

5.2 Consultas

En esta sección de la Tesis se detallan los algoritmos para efectuar cada operación y su análisis de tiempo en el peor caso. Primeramente se describe la estructura de datos con la que se almacenan los conjuntos de nodos. Esta estructura resulta ser un árbol también, en que las relaciones de descendencia se definen justamente teniendo en cuenta las posiciones en la view. Es decir, un nodo desciende de otro en el árbol si y sólo si desciende en la view (véase Figura 6).

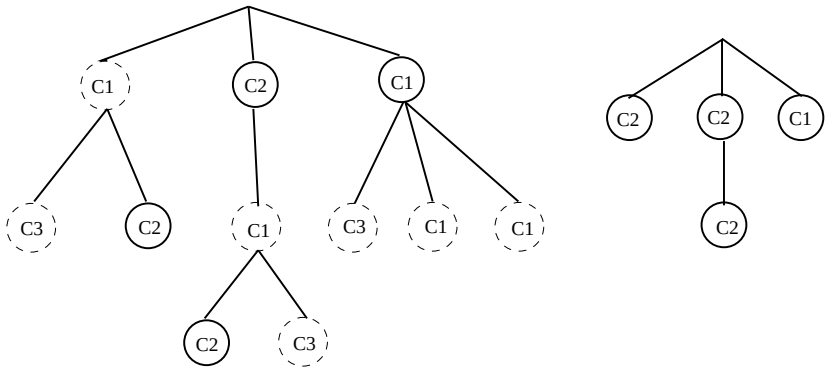

Figura 6: Ejemplo de una consulta (subconjunto de la view) y su representación como árbol.

Seguidamente se estudia el problema de generar un plan de acción inteligente a partir del árbol de la consulta, atendiendo a problemas como minimizar el espacio máximo que se requiere para mantener los resultados intermedios, políticas de swapping cuando no todos los resultados

Operación	Search-Operate	Merge	Lazy
$+/-$	$n \log d..n \log^2 d$	n	$n \min(d,h)$
same	$n \min(n, h \log d)$	n	n
in	$n \log d$	$\min(n, d^2 h)$	$\min(n, d^2 h)$
$[s]$**in**	$n \min(n, d + h \log d)$	$n \min(d,h)$	$n \min(d,h)$
with(k)	$n \min(n, h \log d)$	$k = 1 \ ? \ n \ : \ nh$	$n \min(n, k + dh)$
$[s]$**child**	$n \log d..n \log^2 d$	n	n
parent(k)	$n \log d..n \log^2 d$	n	$n \min(d,h)$
after/before(C)	$n \min(n \log d, dh)$	$n \min(n, dh)$	$n \min(n, dh)$

Tabla 1: Complejidad para las distintas versiones de algunos de los operadores estudiados en la Tesis.

intermedios caben en memoria, compartir conjuntos obtenidos por subexpresiones comunes, etc. Este tema deja en general una serie de líneas futuras de investigación que no se resuelven en este trabajo.

No vamos a reproducir aquí la descripción de los algoritmos. Sólo vamos a explicar la filosofía general de las operaciones tipo *search-operate*, *merge* y *lazy*.

En la primera, se selecciona el menor de los dos operandos, y para cada elemento E de su top-level, se busca este elemento dentro del segundo (los niveles se mantienen en arreglos para permitir búsqueda binaria en cada nivel), hasta hallar la zona donde debe operar. Además de operar en el lugar, el subárbol del elemento E debe a su vez ser operado con el subárbol de la zona encontrada por E. Puede darse que en los subproblemas haya cambiado la relación de tamaño y se elija buscar el segundo operando en el primero.

En las versiones *merge*, se recorren ambos árboles en forma sincronizada, buscando en cada caso contra qué elementos debe ir operando el elemento corriente de uno de los operandos. El método se llama así por la semejanza con el merging de dos listas ordenadas, donde se las recorre en forma sincronizada.

Las versiones *lazy* son muy similares a las *merge*, sólo que se expande un nivel del resultado por vez, de modo que a veces en vez de realizar todo el trabajo, se deja indicado para ser resuelto más adelante, si es requerido. Esto hace que haya que re-atravesar algunas listas, y la complejidad sea más alta que para las versiones *merge* correspondientes.

Esto es sólo una descripción general, los algoritmos son variaciones sobre la idea inicial. La descripción de los algoritmos se encuentra a todo lo largo del Capítulo 5. En cada caso se analiza el costo del algoritmo en el peor caso, basándose en tres parámetros de los operandos: n (el tamaño del árbol), h (la altura máxima) y d (la aridad máxima). En el caso de *lazy*, también se analiza cuánto se puede llegar a trabajar en una sóla invocación lazy, y cuánto se puede llegar a expandir de ambos operadores.

Véase en la Tabla 1 los resultados resumidos de las complejidades de todos los algoritmos. Como puede verse, las versiones *merge* son casi invariablemente las mejores, siendo lineales en la mayor parte de los casos.

6 Un Prototipo

Esta sección describe brevemente el prototipo desarrollado para probar el modelo en situaciones reales. El prototipo sigue el modelo propuesto para la arquitectura.

El motor de texto se basa en la API de SearchCity [1], un sistema de búsqueda en texto que implementa parcialmente el modelo híbrido [2] usando PAT o suffix arrays [13, cap. 5] y [25].

Para indizar la estructura, se crearon indizadores ad-hoc para varios "lenguajes" de estructuración: SGML [18, 14], LaTeX [23], código C [19] y DDIF [12]. En cada caso debió implementarse un parser que obtuviera la estructura a partir del texto y un filtro que eliminara las marcas de estructura al verse el texto desde un nivel más alto. Se implementó la versión más simple del índice de estructura (completo y en memoria).

El motor de consultas incluye todos los operadores representativos, en sus tres versiones (*search-operate*, *merge* y *lazy*).

Finalmente, la interfaz y la capa documental son aun rudimentarias.

Con este prototipo se condujeron varias pruebas, las más importantes son:

- Una prueba operador por operador, con tamaños de operandos entre 100 y 10.000 nodos, con operadores de árboles planos y profundos. Esto totalizó un total de 1300 pruebas. El objetivo principal era conocer los tiempos promedio de las operaciones y cuándo es mejor utilizar *merge*, *search-operate* o evaluación *lazy*.

- Una prueba de *lazy* vs *merge*, con diez consultas reales, más complejas que las de un sólo operador hechas en la prueba anterior. Se pretendía determinar cuánto tarda la evaluación *lazy* con respecto a *merge* en casos reales, y qué porcentaje del total de nodos eran expandidos en promedio.

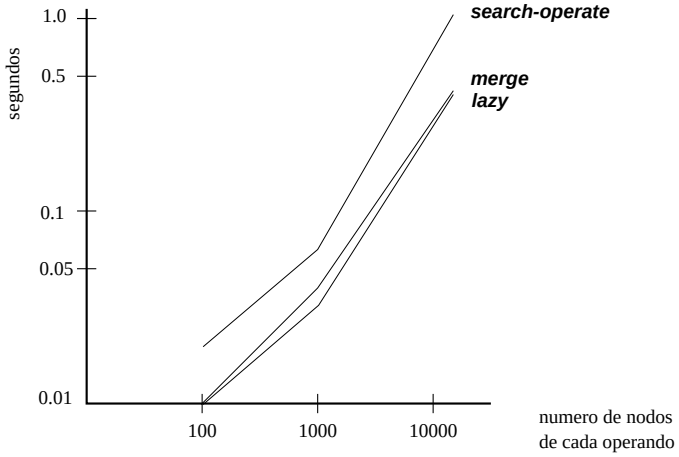

Figura 7: Tiempos típicos para operandos de igual tamaño. Observar que se usa una escala logarítmica.

La Figura 7 muestra los tiempos para una caso típico con un operador. Los tiempos corresponden a una Sun SparcClassic con 16 Mb de RAM, corriendo SunOS 4.1.3_U1. La velocidad de CPU es aproximadamente 26 SpecMark. Los principales resultados que se extraen de las pruebas son los siguientes:

- Los algoritmos *merge* son mucho mejores que los *search-operate*, para operandos de más de 1000 nodos (con menos nodos son similares). Los algoritmos *search-operate* han resultado de peor complejidad, más complejos de escribir y de analizar, y finalmente de peor comportamiento real: mayor promedio y varianza. De modo que deben ser definitivamente descartados.

- Los algoritmos *merge* y *lazy* son decididamente lineales en el tamaño de los operandos, y los *merge* sumamente predecibles en sus tiempos. Esto es bueno para tareas interactivas. Las versiones *lazy*, en cambio, tienen una varianza mucho mayor, aunque en promedio están cercanas a las *merge*.

- Los tiempos de *merge* para operaciones simples corresponden a unos 50.000 nodos procesados por segundo.

- En consultas más complejas (y más reales) las versiones *lazy* son generalmente mejores que las *merge*, aunque la fracción es sumamente variable. Un intervalo de confianza razonable es [0.25..0.90] (fracción de lo que tarda *lazy* con respecto a *merge*), y un valor común el 0.5. El porcentaje de nodos expandidos es también muy variable, yendo del 40% al 100% del total.

- En consultas complejas, el tiempo del *merge* es proporcional a la suma de los tamaños de todos los resultados intermedios. Una estimación más gruesa es $(2q+1) \times$ *tamaño promedio de operando*, donde q es la cantidad de nodos del árbol de la consulta. La constante en nuestra máquina resulta ser de $[4..7] \times 10^4$ nodos procesados por segundo.

7 Conclusiones y Trabajo Futuro

Se ha analizado el problema de consultar una base de datos de texto por su contenido y su estructura. Los modelos existentes se encontraron demasiado enfocados en expresividad o en eficiencia, pero no en ambas.

Se definió un modelo para estructurar documentos y un lenguaje de consulta que resultaron suficientemente expresivos y eficientemente implementables. La idea principal es una organización jerárquica de los nodos y un lenguaje de conjuntos basado en operaciones sobre nodos cercanos.

Para evaluar la expresividad se comparó el modelo propuesto con otros afines, mostrando qué aspectos de cada uno podía ser representado con el otro; y se definió una metodología para evaluar expresividad en modelos afines.

Para evaluar eficiencia se analizó el comportamiento de nuestros algoritmos en términos de tiempo y espacio, y se implementó un prototipo para obtener tiempos promedios y evaluar las distintas soluciones propuestas.

Los resultados de este trabajo son un nuevo modelo para estructurar y consultar bases de datos de texto, que tiene una implementación eficiente y buena expresividad, una metodología para comparar expresividad, y un estudio teórico y práctico del comportamiento del modelo en términos de tiempo y espacio.

Si bien se pueden agregar muchas operaciones al lenguaje, este trabajo muestra que algunas no pueden agregarse sin degradar la performance (por ejemplo, semijoin por contenidos). Finalmente, se insiste en que el lenguaje es un álgebra operacional y no está diseñado para usuarios finales. Sirve como representación intermedia de las operaciones.

Véase en la Figura 8 una comparación gráfica (e informal) de la ubicación de los distintos modelos, atendiendo a la eficiencia y a la expresividad.

Figura 8: Comparación entre modelos similares, atendiendo a la eficiencia y la expresividad.

Este trabajo da un paso en la dirección de obtener una perspectiva unificadora acerca de cómo debería ser un lenguaje de consultas para bases de datos textuales, qué poder expresivo debería tener y cuán bien se puede implementar. También deja abiertas varias líneas de investigación que merecen ser seguidas:

- Encontrar más operaciones eficientemente implementables en nuestro modelo.
- Definir un lenguaje de consultas orientado a usuarios finales, para mapear sobre nuestra álgebra operacional.
- Refinar el mecanismo lazy, permitiendo expandir sólo partes de un nivel, dado que en este momento el mecanismo expande de una vez todos los hijos de un nodo y eso no es bueno para árboles poco profundos (y la mayoría lo son).
- Implementar las técnicas propuestas para mantener el índice y los operandos en disco, para estudiar el comportamiento de diferentes métodos y políticas.
- Diseñar un parser de consultas que genere planes de acción inteligentes, implementando las técnicas propuestas (árboles, DAGs y políticas de dealocación).
- Extender el prototipo, convirtiéndolo en una aplicación real, con manejo de índices en disco, swapping, generación de planes de acción, etc. Una posibilidad interesante es usar el mecanismo lazy como el motor de un sistema que combine consultas de conjuntos con navegación.

Para terminar, indicamos una serie de problemas abiertos que han aparecido a lo largo de este trabajo, y merecen un serio estudio aparte. Algunas son:

- El diseño de técnicas de optimización de consultas para nuestro lenguaje, basado en identidades algebraicas o en atomizar operaciones compuestas muy comunes.

- La integración entre este tipo de modelo y otros, como el relacional o el de RI tradicional. Este tema no se ha considerado aquí, dado que nos centramos en el problema de la estructura. Relacionado con esto está generalizar el problema para manejar estructuras no jerárquicas, como un hipertexto. No se ha hallado aun un lenguaje expresivo y eficiente para consultar estas estructuras.

- Hallar un modelo formal en que comparar expresividad. En este trabajo se hizo algo en ese sentido, pero el objetivo a largo plazo es una jerarquía formal y bien fundada. Véase [9, 16] como ejemplos. Otro problema es obtener consenso sobre un estándar práctico de expresividad, lo que es un problema de maduración.

Referencias

[1] Ars Innovandi, Santiago, Chile. *Search City 1.1. Text Retrieval for Windows Power Users*, 1992.

[2] R. Baeza-Yates. An hybrid query model for full text retrieval systems. Technical Report DCC-1994-2, Dept. of Computer Science, Univ. of Chile, 1994. `ftp://sunsite.dcc.uchile.cl/-pub/users/rbaeza/hybridmodel.ps.gz`.

[3] R. Baeza-Yates and G. Navarro. *Integrating contents and structure in text retrieval. ACM SIGMOD Record*, Mar. 1996. To appear. `ftp://sunsite.dcc.uchile.cl/pub/users/-gnavarro/sigmod.ps.gz`.

[4] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM TOIS*, 6(1):1–41, Jan. 1988.

[5] C. Clarke, G. Cormack, and F. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 1995.

[6] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, Sept. 1987.

[7] M. Consens and A. Mendelzon. Hy$^+$: A hygraph-based query and visualization system. In *Proc. ACM SIGMOD'93*, pages 511–516, 1993. Video presentation summary.

[8] M. Consens and T. Milo. Optimizing queries on files. In *Proc. ACM SIGMOD'94*, pages 301–312, 1994.

[9] M. Consens and T. Milo. Algebras for querying text regions. In *Proc. PODS'95*, 1995.

[10] C. Date. *An Introduction to Database Systems*. Addison-Wesley, 6th edition, 1995.

[11] B. Desai, P. Goyal, and S. Sadri. A data model for use with formatted and textual data. *Journal of ASIS*, 37(3):158–165, 1986.

[12] Digital Equipment Corporation. *CDA - DDIF Technical Specification*, June 1991.

[13] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[14] C. Goldfarb. *The SGML Handbook*, 1990.

[15] G. Gonnet and F. Tompa. Mind Your Grammar: a new approach to modelling text. In *Proc. VLDB'87*, pages 339–346, 1987.

[16] M. Gyssens, J. Paredaens, and D. V. Gucht. A grammar-based approach towards unifying hierarchical data models. In *Proc. ACM SIGMOD'89*, pages 263–272, 1989.

[17] R. Hull and R. King. Semantic database modelling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[18] International Standards Organization. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*, 1986. ISO 8879-1986.

[19] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[20] P. Kilpeläinen and H. Mannila. Grammatical tree matching. In *Proc. CPM'92*, pages 162–174, 1992.

[21] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proc. ACM SIGIR'93*, pages 214–222, 1993.

[22] W. Kim and F. Lochovski, editors. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, 1989.

[23] L. Lamport. LaTeX*: A Document Preparation System*. Addison-Wesley, 1986.

[24] I. MacLeod. A query language for retrieving information from hierarchic text structures. *The Computer Journal*, 34(3):254–264, 1991.

[25] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proc. ACM-SIAM'90*, pages 319–327, 1990.

[26] G. Navarro and R. Baeza-Yates. Expressive power of a new model for structured text databases. In *Proc. PANEL'95*, pages 1151–1162, 1995. `ftp://sunsite.dcc.uchile.cl/pub/users/gnavarro/clei.ps.gz`.

[27] G. Navarro and R. Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proc. ACM SIGIR'95*, pages 93–101, 1995. `ftp://sunsite.dcc.uchile.cl/pub/users/gnavarro/sigir.ps.gz`.

[28] R. Sacks-Davis, T. Arnold-Moore, and J. Zobel. Database systems for structured documents. In *Proc. ADTI'94*, pages 272–283, 1994.

[29] A. Salminen and F. Tompa. PAT expressions: an algebra for text search. In *COMPLEX'92*, pages 309–332, 1992.

[30] G. Salton. Another look at automatic text retrieval systems. *CACM*, 29:648–656, 1986.

[31] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[32] G. Salton and M. McGill. *Automatic text processing*. Addison-Wesley, 1989.

[33] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman. Document processing in a relational database system. *ACM TOIS*, 1(2):143–158, Apr. 1983.

[34] J. Tague, A. Salminen, and C. McClellan. Complete formal model for information retrieval systems. In *Proc. ACM SIGIR'91*, pages 14–20, 1991.