Índices en Memoria Secundaria para Búsquedas en Texto

Gonzalo Navarro
Departamento de Ciencias de la Computación
Universidad de Chile, Chile
gnavarro@dcc.uchile.cl

Nieves Rodríguez Brisaboa
Facultad de Informática
Universidad de A Coruña, España
brisaboa@udc.es

Norma Herrera, Carina Ruano, Ana Villegas
Departamento de Informática
Universidad Nacional de San Luis, Argentina
{nherrera, cmruano, anaville}@unsl.edu.ar

Resumen

Una base de datos de texto es un sistema que mantiene una colección grande de texto y que provee acceso rápido y seguro al mismo. Los *arreglos de sufijos* y *árboles de sufijos* son efectivos para manejar cadenas de longitud no limitada, pero esta eficiencia se degrada considerablemente si el texto es lo suficientemente grande como para que el índice resida en memoria secundaria. En este trabajo estamos interesados en índices dinámicos en memoria secundaria. Específicamente estamos trabajando sobre el *Compact Pat Tree* y el *String B-Tree* , dos índices que conservan las facilidades de búsqueda de los arreglos y árboles de sufijos, son dinámicos y tienen un buen desempeño en memoria secundaria.

1. Introducción

Una base de datos de texto es un sistema que mantiene una colección grande de texto y que provee acceso rápido y seguro al mismo. Las tecnologías tradicionales de bases de datos no son adecuadas para manejar este tipo de bases de datos dado que no es posible organizar una colección de texto en registros y campos. Además, las búsquedas exactas no son de interés en este contexto. Sin pérdida de generalidad, asumiremos que la base de datos de texto es un único texto que posiblemente se encuentra almacenado en varios archivos.

Las búsquedas en una base de texto pueden ser búsquedas sintácticas, en las que el usuario especifica la secuencia de caracteres a buscar en el texto, o pueden ser búsquedas semánticas en la que el usuario especifica la información que desea recuperar y el sistema retorna todos los documentos que son relevantes. En este trabajo estamos interesados en búsquedas sintácticas.

Una de las búsquedas sintácticas más sencilla en bases de datos de texto es la *búsqueda de un patrón*: el usuario ingresa un string P (*patrón de búsqueda*),y el sistema retorna todas las posiciones del texto donde P ocurre. Para resolver este tipo de búsqueda podemos o trabajar directamente sobre el texto sin preprocesarlo o preprocesar el texto para construir un índice que será usado posteriormente para acelerar el proceso de búsqueda. En el primer enfoque encontramos algoritmos como Knuth-Morris-Pratt [7] y Boyer-Moore [2], que básicamente consisten en construir un autómata en base al patrón P que guiará el procesamiento secuencial del texto; estas técnicas son adecuadas cuando el texto ocupa varios megabytes. Si el texto es demasiado grande se hará necesario la construcción de

T: a b c c a b c a $

$=00 a=01 b=10 c=11

Pos.	Sufijo
9	$
8	a$
5	a b c a $
1	a b c c a b c a $
6	b c a $
2	b c c a b c a $
7	c a $
4	c a b c a $
3	c c a b c a $

Figura 1: Un ejemplo de un texto y sus correspondiente sufijos ordenado lexicográficamente.

un índice. Entre los índices más populares encontramos el *arreglo de sufijos* [8] y el *árbol de sufijos* [11].

Dado un texto $T = t_1, \ldots, t_n$ sobre un alfabeto Σ, donde $t_n = \$ \notin \Sigma$ es un símbolo menor en orden lexicográfico que cualquier símbolo de Σ, entonces $T_{i,n} = t_i, \ldots, t_n$ es un sufijo de T para todo $i = 1..n$. La figura 1 muestra un ejemplo de un texto y sus correspondientes sufijos ordenados lexicográficamente, suponiendo que la codificación es $\$ = 00, \mathrm{a} = 01, \mathrm{b} = 10, \mathrm{c} = 11$. Un *arreglo de sufijos* $A[1, n]$ es una permutación de los números $1, 2, \ldots, n$ tal que $T_{A[i],n} \prec T_{A[i+1],n}$, donde $\prec$ es la relación de orden lexicográfico. Un patrón P ocurre en el texto si es prefijo de algún sufijo del texto. En consecuencia, las ocurrencias de P en T pueden determinarse por dos búsquedas binarias que identifiquen el segmento del arreglo A que contiene todas la ocurrencias de P.

Un *árbol de sufijos* es un Pat-Tree [6] construido sobre el conjunto de todos los sufijos de T. Cada nodo interno mantiene el número de bit del patrón que corresponde usar en ese punto para direccionar la búsqueda y las hojas contienen una posición (sufijo) del texto. La figura 2 muestra el árbol y el arreglo de sufijos del ejemplo dado en la figura 1.

Los arreglos y árboles de sufijos son efectivos para manejar cadenas de longitud no limitada, pero esta eficiencia se degrada considerablemente si el texto es lo suficientemente grande como para que el índice resida en memoria secundaria. Los índices clásicos como las Listas Invertidas [4] y Prefix B-Tree [1] tienen un muy buen desempeño en memoria secundaria pero su eficiencia degrada considerablemente cuando las claves de búsqueda tienen una longitud arbitrariamente grande, como ocurrirá si intentamos indexar todos los sufijos del texto. En la actualidad, el diseño de índices para textos que tengan una buena performance en memoria secundaria es un tema de creciente interés.

En este trabajo estamos interesados en índices dinámicos en memoria secundaria. Específicamente estamos trabajando sobre el Compact Pat Tree [3] y el String B-Tree [5], dos índices para memoria secundaria que conservan las facilidades de búsqueda de los arreglos y árboles de sufijos y son dinámicos.

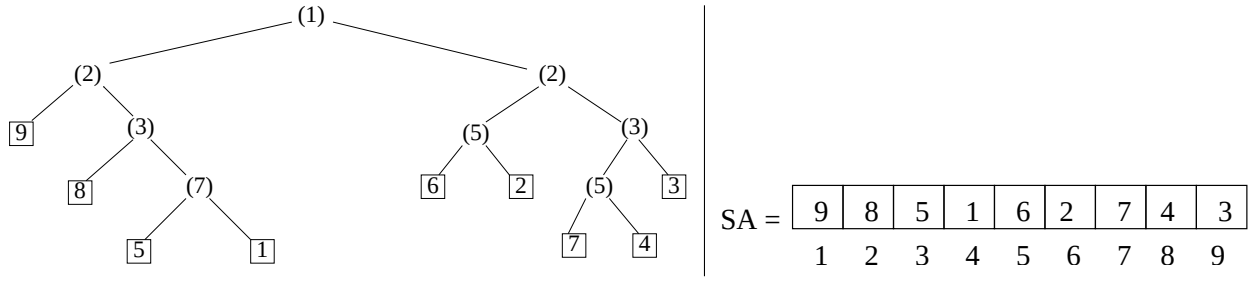

Figura 2: Árbol de sufijos (izquierda) y arreglo de sufijos (derecha) para el ejemplo de la figura 1.

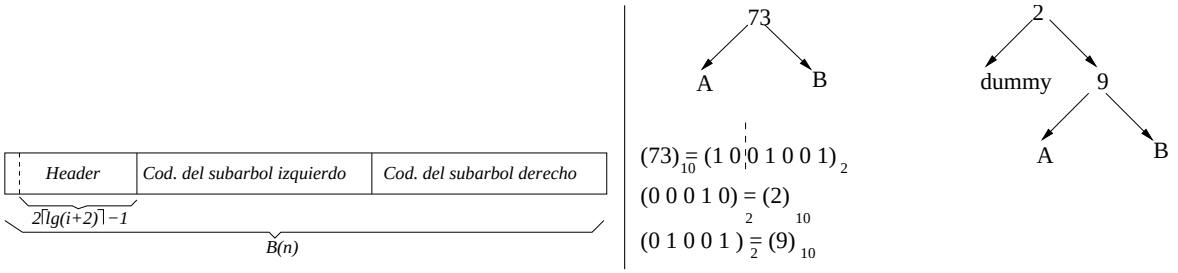

Figura 3: Representación compacta de la forma del árbol (izquierda) y de los valores de salto (derecha).

Comenzaremos explicando el índice Compact Pat Tree, detallando la estrategia utilizada para obtener una representación compacta, luego presentamos el String B-Tree detallando su construcción y finalizamos describiendo el trabajo que actualmente estamos realizando en esta temática.

2. Compact Pat Trees

La información almacenada en un árbol de sufijos puede dividirse en tres categorías: la forma del árbol, los valores de salto en los nodos internos y los desplazamientos de los sufijos en los nodos hojas. Un *Compact Pat Tree(CPT)* [3] consiste de una representación compacta de cada una de estas componentes del árbol más una estrategia para manejar esta representación en memoria secundaria.

Representación compacta de la forma del árbol

En el índice CPT se representa la forma de cada árbol somo un string binario formado por un *header*, seguido de la codificación del subárbol izquierdo y de la codificación del subárbol derecho (ver figura 3, izquierda). El campo *header* se forma con dos valores: un único bit que indica cuál de los dos subárboles es el más pequeño y un código que indica el tamaño, en cantidad de nodos, del subárbol más pequeño. Este código ocupa $2\lceil \lg(i+2) \rceil - 1$ bits y se forma concatenando la codificación unaria de $\lfloor \lg(i+1) \rfloor$ con la representación binaria de $i+1$. Esta técnica permite construir un código prefijo tal que ningún valor codificado es prefijo de otro. Por ejemplo, si el subárbol más pequeño es el izquierdo y este subárbol tiene 2 nodos, el campo *header* sería 1011 donde el primer bit 1 significa *izquierdo* y los últimos tres bits son el resultado de codificar el número 2 como $(\log 3)_1 \centerdot (3)_2 = 011$.

Para asegurar que las operaciones de navegación sobre el árbol puedan implementarse eficientemente, la codificación de un árbol de n nodos se aumenta de manera tal que ocupe el mismo espacio que el máximo requerido para codificar un árbol de n nodos. Sabiendo que el campo *header* ocupa $2\lceil \lg(i+2) \rceil$ bits, se puede demostrar que el tamaño de la codificación de un árbol de n nodos ocupa $B(n) = 3n - 2\lceil \lg(n+1) \rceil - 2v_2(n+1) + 2$ bits, es decir, ocupa menos de 3 bits por nodo [3].

Representación compacta del valor de salto

La compresión de los valores de salto se realiza de la siguiente manera: se reserva una pequeña cantidad fija de bits para almacenar los valores de salto en los nodos internos. En caso de que un valor de salto ocupe más bits de los que hay reservados (overflow), se crea un nuevo nodo interno y se distribuye el valor entre el nodo original y el nuevo nodo creado. Además. también se crea una nueva hoja *dummy* a fin de completar el nuevo nodo interno creado. La figura 3 (derecha) muestra un ejemplo de un caso de overflow, en el que se han reservado 5 bits para el valor de salto. El valor de salto 73 se ha dividido en los valores 2 y 9 teniendo en cuenta la representación binaria del 73. La nueva hoja *dummy* debe tener algún valor especial que permita reconocerla para evitar comparaciones con ella. Si un valor de salto es demasiado grande, se crearán tantos nodos internos y y nodos dummy como sean necesarios.

Representación compacta de los desplazamientos

Los desplazamientos de los sufijos almacenado en las hojas admiten compresión si estamos dispuestos a sacrificar la performance. La técnica usada en CPT es la misma que se usa en Shang's PaTries [10]. Se omiten los l bits de menor orden en cada uno de los desplazamientos, logrando así ahorrar nl bits en el índice completo. Luego, durante una búsqueda, cada vez que se requiera conocer el desplazamiento exacto, se deberá buscar entre 2^l sufijos posibles seleccionando aquel cuya búsqueda finalice en la hoja correcta. Esto implica un costo adicional de 2^l durante la búsqueda y un factor multiplicativo de 2^l para convertir un nodo en su lista de desplazamientos posibles.

Método de paginación

Para controlar la cantidad de accesos a memoria secundaria realizados durante una búsqueda en el *CPT*, se particiona el árbol en componente conexas, a las que llamaremos *partes*. Cada una de las partes se almacena en una página de disco usando la representación vista en las secciones anteriores, con la única diferencia que los desplazamientos en las hojas de cada parte pueden ahora ser o bien desplazamientos reales dentro del texto o bien punteros a otra parte del árbol. En consecuencia, se debe agregar un bit en las hojas para poder distinguir ambos casos.

El algoritmo propuesto por los autores es un algoritmo greedy que procede en forma bottom-up tratando de condensar en una única parte un nodo con uno o los dos subárboles que dependen de él. En este proceso de particionado las decisiones se toman en base a la profundidad de cada nodo involucrado, donde la profundidad de un nodo a es la cantidad máxima de páginas que se deben acceder en un camino que comience en a y termine en una hoja del subárbol con raíz a.

3. String B-Tree

Un String B-Tree (SBT) [5] es un árbol B^+ en el que las claves son los sufijos del texto a indexar. Dado que los sufijos tienen distintas longitudes en este árbol B^+ se almacena un puntero lógico al sufijo correspondiente a fin de lograr que todas las claves tengan exactamente la misma longitud. Denotaremos con $SUF(T)$ al conjunto de sufijos del texto a indexar, y denotaremos con $\mathcal{K} = K_1, K_2, \ldots K_n$ a los strings de $SUF(T)$ ordenados lexicográficamente. Luego, $K_1, K_2, \ldots K_n$ estarán en las hojas del árbol y sólo algunos de ellos se trasladarán a los nodos internos a fin de servir como referencia para direccionar la búsqueda en el árbol.

Vamos a denotar con Π a un nodo del árbol, S_Π al conjunto de strings asociados a Π, $L(\Pi)$ al string que se encuentra más a la izquierda de Π (el menor de S_Π en orden lexicográfico) y con $R(\Pi)$ al string que se encuentra más a la derecha de Π (el mayor de S_Π en orden lexicográfico). Cada nodo Π se almacenará en una página de disco. Si suponemos que $\mathbf{B}$ es el tamaño de la página de disco luego $b \leq |S_\Pi| \leq 2b$ con $b = \Theta((B)$.

Para construir un SBT sobre $SUF(T)$ se procede de la siguiente manera. Se particiona $\mathcal{K}$ en grupos de $2b$ sufijos cada uno, excepto posiblemente el último grupo que puede contener menos. Cada uno de estos grupos formará una hoja del SBT, es decir, será el conjunto S_Π para una hoja Π. Luego, se promueven al nivel superior los sufijos $L(\Pi)$ y $R(\Pi)$ de cada nodo hoja Π, con los que se procede de igual manera. En un árbol B^+ tradicional, cada nodo del árbol es una lista secuencial ordenada. En el caso del SBT, se utiliza dentro de cada nodo un Pat-Tree. Los autores presentan estrategias de *split* y *join* de árboles Pat-Tree, lo que permite que el índice SBT sea dinámico siguiendo la estrategia tradicional de inserción y eliminación en un B^+-Tree.

4. Trabajo Futuro

En la actualidad estamos diseñando estrategias que permitan mejorar, tanto en tiempo como en espacio, la performance del CPT y del String B-Tree conservando sus capacidades dinámicas.

Con respecto al CPT una primera mejora consiste en aumentar la cardinalidad de cada parte a fin de lograr bajar la profundidad total del árbol. Para lograr esto, se almacena en un archivo separado todas las hojas del árbol original. En cada página mantenemos sólo aquellas hojas que son punteros a otras páginas y un *bitmap* de tantas posiciones como hojas haya en esa parte que identifica la situación de cada hoja: 1 significa que la hoja correspondiente es un puntero a página y 0 significa que la hoja correspondiente es un puntero a un sufijo del texto y por lo tanto corresponde buscarla en el archivo separado. La situación de cada hoja puede determinarse con operaciones *rank* [9] sobre el *bitmap*.

Con respecto al String B-Tree, persiguiendo el mismo objetivo, la estrategia diseñada consiste en usar en cada nodo del árbol un Pat-Tree compactado siguiendo las ideas usadas en el CPT. De esta manera se espera aumentar la cardinalidad de cada Pat-Tree y mejorar así la cantidad de accesos a disco. Para lograr esto sin perder la capacidad dinámica del String B-Tree, se están estudiando distintas alternativas que permitan adaptar las operaciones de split y join (necesarias para inserciones y eliminaciones) sobre el Pat-Tree subyacente a la representación compacta del mismo.

Estas actividades se están desarrollando en el ámbito de la línea *Técnicas de indexación para datos no estructurados* del proyecto *Tecnologías Avanzadas de Bases de Datos (UNSL)*.

Referencias

[1] R. Bayer and K. Unteraurer. Prefix B-trees. *ACM Trans. Database System*, pages 11–26, 1977.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[3] D. Clark and I. Munro. Efficient suffix tree on secondary storage. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[4] C. Faloutsos. Access methods for text. *Computing Surveys*, 17(1):49–74, 1985.

[5] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[6] G. H. Gonnet, R. Baeza-Yates, and T. Snider. *New indices for text: PAT trees and PAT arrays*, pages 66–82. Prentice Hall, New Jersey, 1992.

[7] D. E. knuth, J. H. Morris, and V. R.Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[8] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[9] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.

[10] H. Shang. Trie methods for text and spatial data structure on secondary storage. *PhD Thesis*, 1995.

[11] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium Switching Theory and Automata Theory*, pages 1–11, 1973.