Búsquedas Indexadas 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, Darío Ruano, Ana Villegas, Susana Esquivel
Departamento de Informática
Universidad Nacional de San Luis, Argentina
{nherrera, cmruano, dmruano, anaville, esquivel}@unsl.edu.ar

Resumen

Uno de los principales problemas al que nos enfretamos al indexar una base de datos de texto es que el índice ocupa más espacio que el texto a indexar, pudiendo alcanzar de 4 a 20 veces el tamaño del mismo. Una alternativa para reducir el espacio ocupado por el índice es buscar una representación compacta del mismo. Pero en grandes colecciones de texto, el índice aún comprimido suele ser demasiado grande como para residir en memoria principal. En estos casos, la cantidad de accesos a discos realizados durante el procesamiento de una consulta resulta crítica para la performance del índice. Nuestro ámbito de investigación es el estudio de índices comprimidos y en memoria secundaria para búsquedas en texto.
Palabras claves: Bases de Datos de Texto, Índices, Compresión, Memoria Secundaria.

1. Contexto

El presente trabajo se desarrolla en el ámbito de la línea Técnicas de Indexación para Datos no Estructurados del Proyecto Tecnologías Avanzadas de Bases de Datos, cuyo objetivo principal es realizar investigación básica en problemas relacionados al manejo y recuperación eficiente de información no tradicional.

2. Introducción

Una base de datos de texto es un sistema que mantiene una colección grande de texto, y provee acceso rápido y seguro al mismo. Las búsquedas en la que el usuario ingresa un *patrón de búsqueda* y el sistema retorna todas las posiciones del texto donde el patrón ocurre, es una de las búsquedas más comunes en este tipo de bases de datos. Las tecnologías tradicionales de bases de datos no ofrecen una solución viable en este contexto, dado que no es posible organizar una colección de texto en registros y campos. Sin pérdida de generalidad, asumiremos que la base de datos de texto es un único texto T posiblemente almacenado en varios archivos.

Dado un texto $T = t_1, \ldots, t_n$ sobre un alfabeto Σ de tamaño σ, donde $t_n = \$ \notin \Sigma$ es un símbolo menor en orden lexicográfico que cualquier otro símbolo de Σ, denota-

	1	2	3	4	5	6	7	8	9
Texto=	**a**	**b**	**c**	**c**	**a**	**b**	**c**	**a**	**$**

Arreglo de Sufijos:

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 $

Arbol de Sufijos:

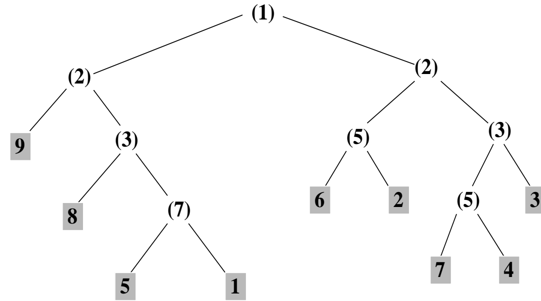

Figura 1: Un ejemplo de un arreglo de sufijos y un árbol de sufijos.

remos con $T_{i,j}$ a la secuencia $t_i, \ldots, t_j$, con $1 \leq i \leq j \leq n$. Un sufijo de T es cualquier string de la forma $T_{i,n} = t_i, \ldots, t_n$ y un prefijo de T es cualquier string de la forma $T_{1,i} = t_1, \ldots, t_i$ con $i = 1..n$. Un patrón de búsqueda $P = p_1 \ldots p_m$ es cualquier string sobre el alfabeto Σ.

Si el texto es pequeño, la búsqueda de patrones puede resolverse eficientemente sin indexar el texto. Si el texto es demasiado grande se debe preprocesar el texto para construir un índice. Entre los índices más populares para resolver búsqueda de patrones encontramos el *arreglo de sufijos*, el *trie de sufijos* y el *árbol de sufijos*. Estos índices se construyen basándose en la observación de que un patrón P ocurre en el texto si es prefijo de algún sufijo del texto.

Arreglo de sufijos: 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 [11]. Buscar un patrón P en T equivale a buscar todos los sufijos de los cuales P es prefijo, los cuales estarán en posiciones consecutivas de A. El proceso de búsqueda consiste entonces en dos búsquedas binarias. La figura 1 muestra un ejemplo de un arreglo de sufijos.

Trie de Sufijos: un trie de sufijos es un *Trie* [4] construido sobre el conjunto de todos los sufijos del texto, en el cual cada hoja mantiene el índice del sufijo que esa hoja representa [15]. El trie de sufijos resuelve eficientemente búsquedas de patrones en un texto basándose en la observación anterior y utilizando la eficiencia del Trie para resolver búsquedas de prefijos en un conjunto de string.

Árbol de sufijos: un árbol de sufijos es un Pat-Tree [4] construido sobre el conjunto de todos los sufijos de T codificados sobre alfabeto binario. 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 del texto que representa al sufijo que se inicia en dicha posición [15]. La figura 1 muestra un ejemplo, suponiendo que la codificación binaria de los símbolos es $\$ = 00, \mathrm{a} = 01, \mathrm{b} = 10, \mathrm{c} = 11$.

Contrariamente a lo que sucede en las bases de datos tradicionales, los índices en las bases de datos de texto generalmente ocupan más espacio que el texto a indexar, pudiendo alcanzar de 4 a 20 veces el tamaño del mismo.

Una alternativa para reducir el espacio ocupado por el índice es buscar una representación compacta del mismo, manteniendo las facilidades de navegación sobre la estructura [5, 6] Pero en grandes colecciones de texto, el índi-

C	$C_{1,1}$ $C_{1,2}$	$C_{2,1}$	$C_{3,1}$ $C_{3,2}$ $C_{3,3}$	$C_{4,1}$ $C_{4,2}$	$C_{5,1}$

A_1	$C_{1,1}$	$C_{2,1}$	$C_{3,1}$	$C_{4,1}$	$C_{5,1}$
B_1	1	0	1	1	0

A_2	$C_{1,2}$	$C_{3,2}$	$C_{4,2}$
B_2	0	1	0

A_3	$C_{3,3}$
B_3	0

Figura 2: Representación de una secuencia de códigos de longitud variable usando DAC.

ce aún comprimido suele ser demasiado grande como para residir en memoria principal. Por esta razón, el estudio de índices comprimidos y en memoria secundaria para búsquedas en texto es un tema de creciente interés en la comunidad de bases de datos.

3. Líneas de Investigación

El objetivo principal de esta línea de investigación es el diseño de índices comprimidos en memoria secundaria, que resulten eficientes para la búsqueda de patrones. Damos a continuación una reseña de las líneas de estudio que estamos desarrollando en este momento.

3.1. Códigos DAC

Dentro de la temática de compresión de datos, un problema central es la asignación de códigos de longitud variable a los símbolos de alfabeto del texto que se está comprimiendo. Los métodos de compresión estadísticos, como por ejemplo Huffman, asignan códigos más corto a los símbolos más frecuentes y códigos más largos a los menos frecuentes. El principal problema de estos métodos es que no permiten acceder eficientemente al i-ésimo símbolo en la secuencia codificada. La solución típica a este problema implica un overhead en tiempo y espacio que ocasiona perder parte del espacio ganado al comprimir.

El *Directly Addressable Variable-Length Code (DAC)*, presentado en [1], es una técnica que permite acceso aleatorio y eficiente a cada código en una secuencia de códigos de longitud variable.

Dada una secuencia de códigos de longitud variable $C = C_1, C_2, \ldots, C_k$, se divide cada código C_i en bloques de b bits. Luego se crea un arreglo A_1 conteniendo la concatenación de los primeros bloques de cada símbolo, y un mapa de bits (*bitmap*) B_1 de k bits, donde el i-ésimo bit está en 1 si el código C_i está formado por más de un bloque. Se continua con la creación de un arreglo A_2 conteniendo la concatenación de los segundos bloques de cada símbolo y un bitmap B_2 con 1 en aquellos bits correspondientes a los códigos con más de dos bloques. Se continua así hasta alcanzar la máxima cantidad de bloques. La figura 2 muestra un ejemplo para una secuencia C formada por 5 códigos de longitud variable, $C_{i,j}$ representa el j-ésimo bloque del i-ésimo código de la secuencia.

Para acceder al código C_i de la secuencia original primero buscamos su primer bloque en $A_1[i]$. Si el i-ésimo bit de B_1 está en 0, se finaliza retornando $C_i = A_1[i]$. Caso contrario, continuamos buscando el segundo bloque del código C_i en $A_2[rank_1(B_1, i)]$, donde $rank_1(B_1, i)$ es la cantidad de unos en $B_1[1..i]$. Este proceso continúa hasta llegar al último nivel de arreglos o hasta encontrar el bit del correspondiente bitmap en cero.

3.1.1. Aplicación de Códigos DAC

Entre los índices para texto en memoria secundaria más relevantes encontramos el String B-Tree [3] y el Compact Pat Tree [2].

El String B-Tree es un índice dinámico para búsquedas de patrones en memoria secundaria. Básicamente consiste en una combinación de dos estructuras: el B-Tree y el Pat-Tree [4]. No es un índice comprimido y su versión

estática requiere en espacio de 5 a 6 veces el tamaño del texto. Estamos trabajando en la implementación de una variante comprimida de este índice que consiste en comprimir la representación de los Pat-Tree involucrados. Para ello se utiliza la representación de paréntesis [12] para mantener la topología del árbol y los códigos DAC para la representación de los valores de salto del Pat-Tree.

El Compact Pat Tree (CPT) consiste en representar un árbol de sufijos en memoria secundaria y en forma compacta. Si bien no existen desarrollos teóricos que garanticen el espacio ocupado por este índice y el tiempo insumido en la búsqueda, en la práctica tiene un muy buen desempeño. Los autores proponen una representación comprimida de los valores de salto que implica la creación de hojas adicionales especiales llamadas *dummy*. Hemos diseñado una implementación del CPT en la que se incorporan los códigos DAC para la representación de los valores de salto de este árbol, lo que evita la creación de hojas dummy. Los primeros experimentos realizados son alentadores en cuanto al espacio ocupado por el índice.

3.2. Locally Compressed SA

Un arreglo de sufijos A construido sobre un texto T de longitud n es compresible si T lo es. La entropía de orden k de T (H_k) se refleja en A formando secuencias largas $A[i, i + l]$, denominadas *pseudo-repeticiones* que aparecen en otro lugar $A[j, j + l]$ con todos los valores incrementados en uno, es decir: $A[j + s] = A[i + s] + 1$ con $0 \leq s \leq l$.

Si particionamos A en *pseudo-repeticiones* de tamaño maximal, el número de partes que obtendríamos sería a lo más $nH_k + \sigma^k$, para algún k [13]. Esta propiedad ha sido usada por varios autores para comprimir un arreglo de sufijos A [9, 10]. El Locally Compressed Suffix Array (LCSA) [6] es un técnica para compresión de arreglos de sufijos que consiste en convertir las *pseudo-repeticiones* en repeticiones reales, que luego son factorizadas usando Re-Pair [8].

3.2.1. Aplicación de LCSA

En [7] hemos presentado un modificación en el diseño del CPT que permite mantener la representación del arreglo de sufijos subyacente en el CPT separada de la representación del árbol propiamente dicho. Esto nos permite como próximo paso reducir el espacio total requerido por el índice comprimiendo dicho arreglo de sufijos. Para ellos estamos trabajando en la incorporación de la técnica LCSA en el CPT.

3.3. Trie de Sufijos

El trabajo sobre este índice apunta a lograr una representación en memoria secundaria de un trie de sufijos, de manera tal que el mismo resulte competitivo en cantidad de accesos a disco. Hemos diseñado una técnica de paginación del mismo la que surge como una extensión para arboles r-arios del paginado utilizado en el CPT. Esta técnica fue presentada en [14], encontrándonos actualmente en la etapa de evaluación experimental de la misma.

4. Resultados Esperados

Se espera obtener índices con las mismas funcionalidades que los originales pero reduciendo el espacio necesario para su representación. En el caso particular del trie, el aporte además será contar con una versión eficiente del mismo en memoria secundaria. El desempeño de los índices obtenidos será medido tanto analíticamente como en forma empírica. Para esto último se cuenta con un conjunto de textos de prueba usados y aceptados por la comunidad científica del área de estudio.

5. Recursos Humanos

El trabajo desarrollado en esta línea de investigación forma parte del desarrollo de dos Trabajos Finales de la Licenciatura en Ciencias de la Computación, dos Tesis de Maestría en Ciencias de la Computación y una Tesis de Doctorado en Ciencias de la Computación, todos realizados en el ámbito de la Universidad Nacional de San Luis, con el asesoramiento del Dr. Gonzalo Navarro de la Universidad de Chile y de la Dra. Nieves Brisaboa de la Universidad da Coruña, España.

Referencias

[1] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Directly addressable variable-length codes. In *SPIRE*, pages 122–130, 2009.

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

[3] 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.

[4] 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.

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

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

[7] N. Herrera and G. Navarro. Árboles de sufijos comprimidos en memoria secundaria. In *Proc. XXXV Latin American Conference on Informatics (CLEI)*, Pelotas, Brazil, 2009.

[8] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *DCC '99: Proceedings of the Conference on Data Compression*, page 296, Washington, DC, USA, 1999. IEEE Computer Society.

[9] V. Mäkinen. Compact suffix array: a space-efficient full-text index. *Fundam. Inf.*, 56(1,2):191–210, 2002.

[10] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.

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

[12] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

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

[14] D. Ruano, N. Herrera, C. Ruano, and A. Villegas. Representación en memoria secundaria del trie de sufijos. In *Actas del XVI Congreso Argentino de Ciencias de la Computación*, Buenos Aires, 2010.

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