Índices para Bases de Datos de 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
Departamento de Informática
Universidad Nacional de San Luis, Argentina
{nherrera, cmruano, dmruano, anaville}@unsl.edu.ar

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, diseñando nuevos algoritmos de indexación que permitan realizar búsquedas eficientes sobre datos no estructurados.

Resumen

Mientras que en bases de datos tradicionales los índices ocupan menos espacio que el conjunto de datos indexados, en bases de datos de texto el índice generalmente ocupa más espacio que el texto pudiendo necesitar 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. 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 memoria secundaria realizados durante el proceso de búsqueda es un factor crítico en la performance del índice. En este trabajo estamos interesados en el diseño de índices comprimidos y en memoria secundaria para búsquedas en texto, un tema de creciente interés en la comunidad de bases de datos.

Palabras claves: *Bases de Datos de Texto, Índices, Compresión, 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 posiblemente 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

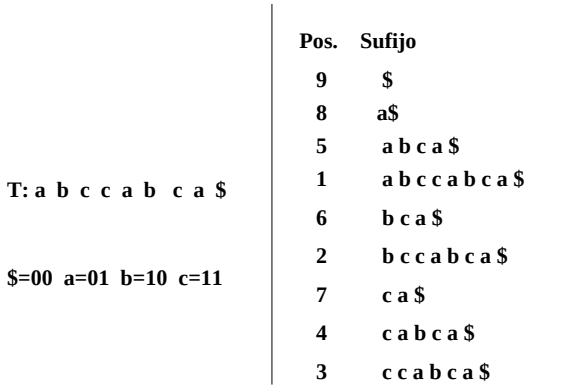

Figura 1: *Un ejemplo de un texto y sus correspondientes sufijos ordenados lexicográficamente.*

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 [12] 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 un índice.

Construir un índice tiene sentido cuando el texto es grande, cuando las búsquedas son más frecuentes que las modificaciones (de manera tal que los costos de construcción se vean amortizados) y cuando hay suficiente espacio como para contener el índice. Un índice debe dar soporte a dos operaciones básicas:

Count : consiste en contar el número de ocurrencias de un patrón P en un texto T.

Locate : consiste en ubicar todas las posiciones del texto T donde el patrón de búsqueda P ocurre.

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, denotaremos 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 $i = 1..n$.

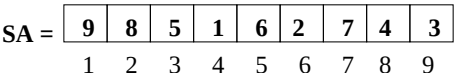

Figura 2: *Arreglo de sufijos para el ejemplo de la figura 1.*

Un patrón de búsqueda $P = p_1 \ldots p_m$ es cualquier string sobre el alfabeto Σ. La figura 1 muestra un ejemplo de un texto y sus correspondientes sufijos ordenados lexicográficamente, suponiendo que la codificación de los símbolos del alfabeto es $\$ = 00, \mathrm{a} = 01, \mathrm{b} = 10, \mathrm{c} = 11$.

Entre los índices más populares para resolver búsqueda de patrones encontramos el *arreglo de sufijos* [15] y el *árbol de sufijos* [24]. 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. 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 que identifiquen el segmento del arreglo A que contiene todas las posiciones de T donde P ocurre. La figura 2 muestra el arreglo de sufijos para el texto de la figura 1.

Árbol de sufijos: un árbol de sufijos es un PatTree [8] 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. La figura 3 muestra el árbol de sufijos para el texto de la figura 1.

Mientras que en bases de datos tradicionales los índices ocupan menos espacio que el conjunto de datos indexados, en bases de datos de texto el índice generalmente ocupa más espacio que el texto pudiendo necesitar de 4 a 20 veces el tamaño del mismo [8, 15]. Una alternativa para reducir el espacio ocupado por el índice es buscar una re-

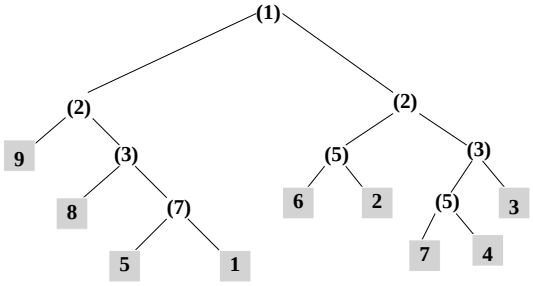

Figura 3: *Árbol de sufijos para el ejemplo de la figura 1.*

presentación compacta del mismo, manteniendo las facilidades de navegación sobre la estructura [6, 7, 9, 10, 13, 19, 20, 22]. Una línea de investigación reciente se enfoca en construir índices que no necesitan almacenar de manera explícita el texto que indexan. Estos índices, denominados ***autoíndices***, mantienen información que permite reconstruir el texto indexado [3, 11, 21, 22].

Pero en grandes colecciones de texto, el índice aún comprimido suele ser demasiado grande como para residir en memoria principal. Como un ejemplo de este caso podemos nombrar las bases de datos conteniendo secuencias de ADN y secuencias de proteínas, que requieren la construcción de un índice de texto completo y cuyo tamaño implicará que el índice resida en memoria secundaria. En estos casos, la cantidad de accesos a memoria secundaria realizados durante el proceso de búsqueda es un factor crítico en la performance del índice [23].

Entre los índices para texto en memoria secundaria más relevantes encontramos:

String B-Tree [5]: consiste básicamente en un B-Tree en el que cada nodo es representado como un Pat-Tree [8]. Este índice requiere tanto para *count* como para *locate* $O(\frac{m+occ}{b}+\log_b n)$ accesos a memoria secundaria en el peor caso, donde occ es la cantidad de ocurrencias de P en T y b es el tamaño de páginas de disco medido en enteros. No es un índice comprimido y su versión estática requiere en espacio de 5 a 6 veces el tamaño del texto más el texto.

Compact Pat Tree [4]: representa un árbol de sufijos en memoria secundaria y en forma compacta. Si bien no existen desarrollos teóricos que garanticen el espacio ocupado por el índice y el tiempo insumido en resolver la búsqueda, en la práctica el índice tiene un muy buen desempeño requieriendo de 2 a 3 accesos a memoria secundaria tanto para *count* como para *locate*, y ocupando entre 4 y 5 veces el tamaño del texto más el texto.

Disk-based Compressed Suffix Array [14]: adapta el autoíndice comprimido para memoria principal presentado en [22] a memoria secundaria. Requiere $n(H_0+O(\log\log\sigma))$ bits de espacio (donde $H_k \leq \log\sigma$ es la entropía de orden k de T [16]). Para la operación *count* realiza $O(m \log_b n)$ accesos. Para la operación *locate* realiza $O(\log n)$ accesos lo cual es demasiado costoso.

Disk-based LZ-Index [1]: adapta a memoria secundaria el autoíndice comprimido para memoria principal presentado en [19]. Utiliza $8\ n\ H_k(T) + o(n\log\sigma)$ bits; los autores no proveen límites teóricos para la complejidad temporal, pero en la práctica es muy competitivo.

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.

2. LÍNEAS DE INVESTIGACIÓN Y DESARROLLO

El objetivo principal de esta línea de investigación es el diseño de índices comprimidos en memoria secundaria. Para lograr este objetivo hemos tomado como base dos índices: el *Compact Pat Tree* (CPT), un índice comprimido dinámico para memoria secundaria, y el *String B-Tree*(SBT) un índice dinámico para memoria secundaria.

Sobre el SBT el objetivo principal es lograr una reducción en el espacio utilizado por el mismo manteniendo los costos de búsquedas de la versión original. Para ello, se han diseñado dos variantes que consisten en modificar la representación de cada nodo del árbol B subyacente. Una de las variantes consiste en usar un Pat-Tree como originalmente proponen los autores para los nodos pero usando representación de paréntesis para el mismo [17]. La otra variante consiste en

representar cada nodo con la representación de arreglos propuesta en [18] que ofrece las mismas funcionalidades que un Pat-Tree pero que tienen las características necesarias como para permitir una posterior compresión de los mismos.

Sobre el CPT hemos diseñado e implementado los algoritmos de creación y búsqueda en un CPT en el cual se reemplaza la codificación de la forma del árbol originalmente propuesta por los autores, por la representación de paréntesis propuesta en [17]. Esta representación utiliza $2n$ bits para codificar un árbol de n nodos en lugar de los $B(n)$ bits (con $2 < B(n) < 3n$) del CPT original. Si bien esta representación es menos eficiente en búsquedas, como sólo se buscará en memoria principal sobre subárboles pequeños (limitados por el tamaño de página de disco) el desempeño del índice no se verá afectado. Los tiempos inclusive podrían mejorar dado que, reducir el espacio usado para codificar la forma del árbol, permite que cada página mantenga subárboles más grandes y la altura total del CPT (en cantidad de páginas) disminuya.

Se ha diseñado y se está implementado una segunda modificación al CPT que consiste en almacenar la hojas del árbol que son índices de sufijos en un archivo separado. El objetivo de esta modificación es lograr que las partes contengan subárboles más grandes y, en consecuencia, la altura total en cantidad de páginas sea menor.

Se está analizando además como extender el algoritmo de paginación propuesto por los autores del CPT para árboles r-arios con el fin de poder paginar otras estructuras para búsquedas en texto.

3. RESULTADOS OBTENIDOS/ESPERADOS

Se espera que los cambios en el diseño del CPT y del SBT permitan obtener índices con las mismas funcionalidades que los originales pero reduciendo el espacio necesario para su representación. 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 ampliamente usados y aceptados por la comunidad científica del área de estudio; los mismos se encuentran disponibles en el sitio `http://pizzachili.dcc.uchile.cl`.

4. FORMACIÓN DE RECURSOS HUMANOS

El trabajo en curso forma parte del desarrollo de un Trabajo Final 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] D. Arroyuelo and G. Navarro. A lempel-ziv text index on secondary storage. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 83–94, 2007.

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

[3] N. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS. Springer, 2008.

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

[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] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[7] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007.

[8] 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.

[9] 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.

[10] 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.

[11] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA'03)*, pages 841–850, 2003.

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

[13] V. Mäkinen and G. Navarro. *Compressed Text Indexing*, pages 176–178. Springer, 2008.

[14] V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching - efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. 15th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 3341, pages 681–692. Springer, 2004.

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

[16] G. Manzini. An analysis of the burrows—wheeler transform. *J. ACM*, 48(3):407–430, 2001.

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

[18] Joong Chae Na and Kunsoo Park. Simple implementation of string b-trees. In Alberto Apostolico and Massimo Melucci, editors, *SPIRE*, volume 3246 of *Lecture Notes in Computer Science*, pages 214–215. Springer, 2004.

[19] G. Navarro. Indexing text using the ziv-lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.

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

[21] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *ISAAC '00: Proceedings of the 11th International Conference on Algorithms and Computation*, pages 410–421, London, UK, 2000. Springer-Verlag.

[22] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[23] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

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