Compresión de Índices para Bases de Datos Textuales

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

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. 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.

2. Introducción

La información disponible en formato digital aumenta día a día su tamaño de manera exponencial. Gran parte de esta información se representa en forma de texto, es decir, secuencias de símbolos que pueden representar no sólo lenguaje natural, sino también música, códigos de programas, secuencias de ADN, secuencias de proteínas, etc. Debido a que no es posible organizar una colección de textos en registros y campos, cada vez que se utiliza este tipo de datos para expresar información, las tecnologías tradicionales de bases de datos no son las más adecuadas para su manejo. Además, las búsquedas exactas provistas por las tecnologías tradicionales no son de interés cuando se maneja este tipo de datos.

Una base de datos de texto es un sistema que mantiene una colección grande de texto, y provee acceso rápido y seguro al mismo. Sin pérdida de generalidad, asumiremos que la base de datos de texto es un único texto, el cual está posiblemente almacenado en varios archivos.

Una de las búsquedas más comunes 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. Los métodos existentes para resolver este tipo de búsquedas pueden clasificarse en dos categorías: secuencial e indexada. Los métodos secuenciales, no requieren un preprocesamiento del texto, estos preprocesan sólo el patrón de búsqueda. En esta categoría encontramos algoritmos como Knuth-Morris-Pratt [14] y Boyer-Moore [2]. Estos algoritmos consisten básicamente en construir un autómata en base al patrón P que guiará el procesamiento secuencial sobre el texto. Por otro lado se encuentran los métodos indexados, estos preprocesan el texto para construir un índice diseñado para resolver búsqueda de patrones en el texto de manera eficiente.

Los métodos secuenciales son adecuados cuando el texto ocupa varios megabytes, mientras que si el texto es demasiado grande será necesaria la construcción

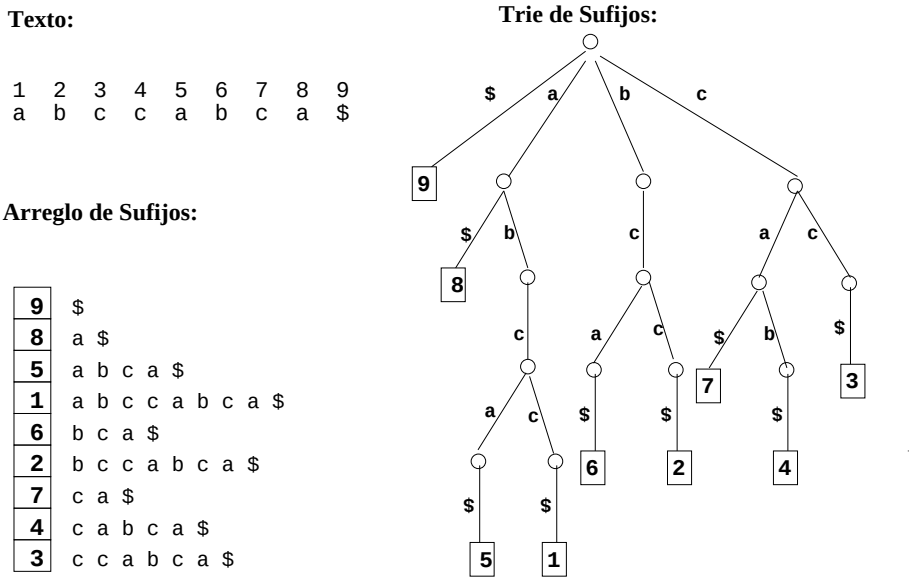

Figura 1: *Un ejemplo de un arreglo de sufijos y de un trie de sufijos.*

de un índice. Construir un índice tiene sentido, además, cuando las búsquedas son más frecuentes que las modificaciones (de manera que los costos de construcción se vean amortizados) y cuando hay suficiente espacio como para contener el índice. Los índices deben dar soporte a dos operaciones básicas: *Count* y *Locate*. *Count* consiste en contar el número de ocurrencias de un patrón P en un texto T y *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 $i = 1..n$. Un patrón de búsqueda $P = p_1 \ldots p_m$ es cualquier string sobre el alfabeto Σ.

Entre los índices más populares para resolver búsqueda de patrones encontramos el *arreglo de sufijos* [17] el *trie de sufijos* 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 1 muestra un ejemplo de un arreglo de sufijos; en el ejemplo se ha indicado junto con cada valor del arreglo el sufijo que ese valor representa.

Trie de Sufijos: un árbol digital o trie es un árbol que permite almacenar un conjunto finito de strings. En este árbol, cada rama está rotulada por un símbolo del alfabeto y cada hoja representa un string del conjunto almacenado en el árbol. El conjunto total de strings se obtiene recorriendo todos los caminos posibles desde la raíz hasta una hoja y concatenando los rótulos de las ramas que forman cada uno de esos caminos. Un trie de sufijos es un trie construido sobre el conjunto de sufijos del texto, en el cual cada hoja mantiene el índice del sufijo que esa hoja representa. Para encontrar todas las ocurrencias de P en T, se busca en el trie utilizando los caracteres de P para direccionar la búsqueda. Si la misma finaliza en un nodo interno x, entonces todas las hojas del subárbol con raíz x forman la respuesta. Si la búsqueda finaliza en una hoja, es necesario realizar una comparación entre P y el sufijo que empieza en la posición del texto indicada por la hoja para determinar si esa hoja es la respuesta. La figura 1 muestra el trie de sufijos para el texto dado. Notar que si recorremos de izquierda a derecha las hojas del trie de sufijos obtenemos el arreglo de sufijos.

Árbol de sufijos: un árbol de sufijos es un Pat-Tree [9] 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

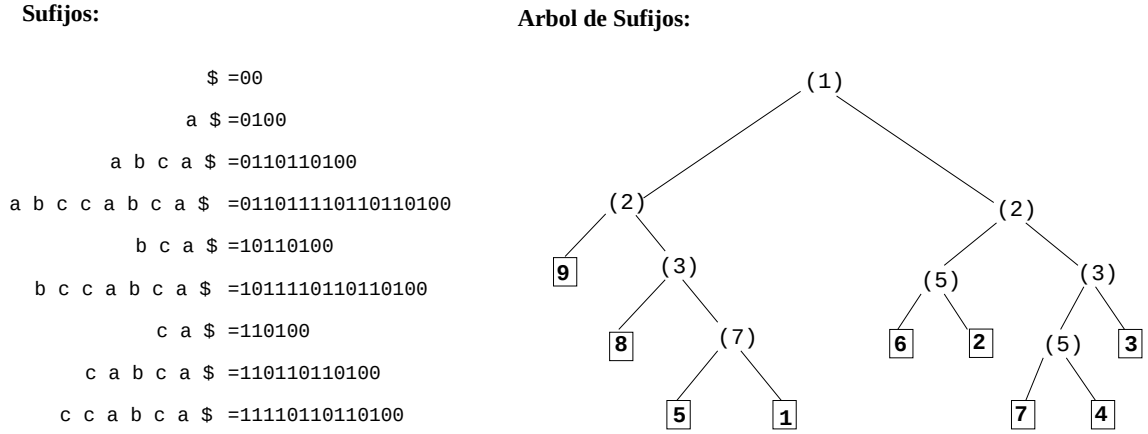

Figura 2: *Codificación binaria de los sufijos (ordenadas lexicográficamente) y su correspondiente árbol de sufijos.*

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 2 muestra el árbol de sufijos para el mismo texto dado en el ejemplo de la figura 1. Para cada sufijo del texto se muestra cuál es su representación binaria suponiendo que la codificación binaria de los símbolos es $\$ = 00, \mathrm{a} = 01, \mathrm{b} = 10, \mathrm{c} = 11$.

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 [5] 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.

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 [9, 17].

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 [7, 8, 10, 11, 16, 20, 21, 23]. 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, 12, 22, 23].

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.

Si bien el almacenamiento secundario cada vez es más económico, el acceso al mismo sigue siendo muy costoso en tiempo. Por lo tanto, el problema real en las bases de datos de texto no es solamente el ahorro de espacio sino también lograr eficiencia en la cantidad de accesos a memoria secundaria requeridos para resolver una búsqueda. 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 y Desarrollo

El objetivo principal de esta línea de investigación es el diseño de índices comprimidos en memoria secundaria. Actualmente estamos trabajando en base a los siguientes índices:

String B-Tree [6]: es un índice dinámico para memoria secundaria capaz de indexar string de tamaño variable. Consiste básicamente en una combinación de dos estructuras: el B-Tree y el Pat-Tree [9]. El B-Tree (usando como claves punteros a los sufijos del texto) forma parte de la estructura general y un Pat-Tree es usado para para representar cada nodo del String B-Tree.

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 espacio 5 a 6 veces el tamaño del texto sólo para la representación del índice. El objetivo principal

del trabajo sobre este índice es reducir el espacio requerido para su almacenamiento sin degradar los costos de búsquedas.

Las modificaciones diseñadas hasta el momento se basan en reemplazar la representación original de cada nodo, por variantes que minimizan el espacio requerido para almacenar el mismo. Una de la variantes modifica la representación original del Pat-Tree para utilizar la representación de paréntesis presentada en [18]. En esta variante se modifica además, la representación de los saltos mediante la utilización de técnicas de compresión. Otra de las variantes consiste en reemplazar el Pat-Tree por la representación de arreglos propuesta en [19]. Esta representación se compone de 1 arreglo de enteros y 1 arreglo de caracteres y utiliza un branching algorithm para resolver las búsquedas. Una de las ventajas de esta representación es que posee características apropiadas para permitir una posterior compresión de los arreglos.

Todas estas variantes se centran en disminuir la cantidad de espacio requerido para representar cada nodo, esto permite aumentar la cantidad de elementos incluidos en el nodo, y como consecuencia la altura del árbol tiende a disminuir. Esta es la base que nos permite suponer que éstas modificaciones afectarán de manera positiva no sólo en el ahorro de espacio requerido por el índice, sino también en la cantidad de accesos a memoria secundaria realizados para resolver operaciones de *count* y *locate*.

Compact Pat Tree [4]: 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 requiriendo de 2 a 3 accesos a memoria secundaria tanto para *count* como para *locate*, y ocupando entre 4 a 5 veces el tamaño del texto. Uno de los principales problemas de este índice es el espacio desperdiciado del total de espacio ocupado por el índice. En los experimentos reportados en [4] el desperdicio de espacio varía entre el $40\,\%$ para textos de 900KB hasta el $60\,\%$ para textos de 100MB. Los autores proponen varias técnicas para la reducción de este desperdicio pero no reportan resultados de las mejoras obtenidas con estas técnicas. En [13] se presenta una modificación en el diseño del CPT que ha permitido bajar el desperdicio de espacio al $20\,\%$ en el peor caso manteniendo la eficiencia del índice. Esta modificación consiste en mantener el arreglo de sufijos subyacente del CPT en un archivo separado del archivo que contiene la estructura del árbol.

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. Para lograr esta representación hay dos aspectos principales a tener en cuenta: la paginación del árbol y la representación secuencial del trie en páginas de disco. Las técnicas de paginación y de representación de árboles propuestas para el CPT y las técnicas propuestas en [15] sirven de base para lograr el objetivo propuesto.

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 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`.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] A. Thomo M. Barsky *, U. Stege. A survey of practical algorithms for suffix tree construction in external memory. In *Software: Practice and Experience*, 2010.

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

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

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

[19] 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.

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

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

[22] 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.

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

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