Indexando Texto en Memoria Secundaria

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

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

La próxima generación de administradores de bases de datos deberá ser capaz de indexar datos no estructurados (datos multimedia) y responder consultas sobre estos datos con tanta eficiencia como actualmente responden consultas de búsqueda exacta sobre bases de datos relacionales. Si bien existen numerosas técnicas de indexación diseñadas para esta problemática, mejorar la eficiencia de las mismas es de vital importancia. Nuestro ámbito de investigación es el estudio de índices eficientes para datos no estructurados.

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 (22/F014), cuyo objetivo es realizar investigación básica en problemas relacionados al manejo y recuperación eficiente de información no tradicional.

2. Introducción

Las bases de datos actuales han incluido la capacidad de almacenar datos no estructurados tales como imágenes, sonido, texto, video, datos geométricos, etc. La problemática de almacenamiento y búsqueda en estos tipos de base de datos difiere de las bases de datos clásicas, dado que no es posible organizar los datos en registros y campos, y aun cuando pudiera hacerse, la búsqueda exacta carece de interés. Es en este contexto donde surgen nuevos modelos de bases de datos capaces de cubrir las necesidades de almacenamiento y búsqueda de estas aplicaciones. Nuestro interés se basa en el diseño de índices para estas nuevas bases de datos, centrándonos en bases de datos de texto.

Un 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 $T = t_1, \ldots, t_n$ posiblemente almacenado en varios archivos. Asumiremos que T está formado por símbolos de 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.

Construir un índice sobre T tiene sentido cuando T 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*, que consiste en contar el número de ocurrencias de un patrón P en un texto T y *locate*, que consiste en ubicar todas las posiciones del texto T donde el patrón de búsqueda P ocurre.

Entre los índices más populares para texto 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 [12]. 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.

Trie de Sufijos: un trie de sufijos es un *Trie* construido sobre el conjunto de todos los sufijos del texto, en el cual cada hoja mantiene el índice del sufijo que esa hoja representa [18]. 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 utilizar 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 [18].

Mientras que en bases de datos tradicionales los índices ocupan menos espacio que el conjunto de datos indexado, en las bases de datos de texto el índice ocupa más espacio que el texto, pudiendo necesitar de 4 a 20 veces el tamaño del mismo [4, 12]. 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. Es por ello que el desarrollo de índices comprimidos en memoria secundaria es un tema de creciente interés. Entre los índices para texto en memoria secundaria más relevantes encontramos:

String B-Tree [3]: consiste básicamente en un B-Tree en el que cada nodo es representado como un Pat-Tree [4]. 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 [2]: 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 [11]: adapta el autoíndice comprimido para memoria principal presentado en [17] 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). 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 [15]. 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.

3. Líneas de Investigación

Nuestra principal línea de trabajo es el estudio de algoritmos de indexación sobre bases de datos no estructurados, centrándonos principalmente en el diseño de índices para bases de datos textuales. Describimos a continuación las líneas de investigación que actualmente estamos desarrollando.

3.1. Locally Compressed SA

El Locally Compressed Suffix Array (LCSA) [5] es una técnica para compresión de arreglos de sufijos. 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 [16]. Esta propiedad ha sido usada por varios autores para comprimir un arreglo de sufijos A [9, 10]. El LCSA 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].

Re-Pair es un compresor basado en diccionario que permite una rápida descompresión local usando solamente el diccionario. La técnica consiste en encontrar el par de símbolos más frecuente y reemplazarlo con un nuevo símbolo. Podemos resumir el proceso realizado por Re-Pair en los siguientes pasos:

1. Encontrar el par de símbolos ab más frecuente en T.
2. Crear un nuevo símbolo s mayor que cualquier símbolo en T y agregar al diccionario R la regla $s \rightarrow ab$.
3. Reemplazar en T toda ocurrencia de ab por s.
4. Repetir hasta que todos los pares tengan frecuencia 1 .

El resultado de este algoritmo de compresión es el diccionario de reglas R más una secuencia de símbolos C (símbolos originales y nuevos) que es el texto T ya comprimido. Notar que podemos representar R en un vector de pares de manera tal que la regla $s \rightarrow ab$ esté representada en $R[s-\sigma] = a : b$.

Cualquier segmento de C puede ser rápida y fácilmente descomprimido de la siguiente manera: para descomprimir $C[i]$ primero verificamos el valor de $C[i]$. Si $C[i] < \sigma$, entonces es un símbolo original de T, por lo tanto no corresponde hacer nada mas. Caso contrario obtenemos los símbolos que corresponden a $C[i]$ en $R[C[i]-\sigma]$ y los expandimos recursivamente. Esto permite reproducir u caracteres de T en $O(u)$ unidades de tiempo.

3.2. LCSA + CPT

Como mencionamos anteriormente, el Compact Pat Tree (CPT) consiste en representar un árbol de sufijos en memoria secundaria y en forma compacta.

En [7] hemos presentado una 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 reducir el espacio total requerido por el índice comprimiendo dicho arreglo de sufijos. Para ello estamos trabajando en la incorporación de la técnica LCSA en el CPT.

Como primer paso se deben diseñar los algoritmos de construcción en memoria secundaria. Para lograr algoritmos eficientes en memoria secundaria es necesario que los mismos tengan alta localidad de referencia. El algoritmo de construcción de LCSA tiene una muy baja localidad de referencia dado que recorre A usando la función Ψ, donde $\Psi[i] = j$ si $A[j] = A[i] + 1$. El algoritmo de construcción del LCSA en memoria secundaria fue propuesto en [6]. Allí se presenta el diseño de dicho algoritmo y el desarrollo de complejidad del mismo, pero sin realizar la implementación y la evaluación empírica del algoritmo. No hay aún resultados experimentales sobre cómo se comporta esta implementación, por lo cual es posible que aún requiera de ajustes para lograr un rendimiento aceptable. En este momento hemos finalizado la implementación del algoritmo de construcción del LCSA en memoria secundaria encontrándonos en la etapa de evaluación empírica del mismo.

3.3. String B-Tree

El String B-Tree(SBT) [3] 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.

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 e implementado 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 [13]. La otra variante consiste en representar cada nodo con la representación de arreglos propuesta en [14] que ofrece las mismas funcionalidades que un Pat-Tree pero que tienen las características necesarias como para permitir una posterior compresión de los mismos. Nos encontramos en la etapa de evaluación experimental de estas versiones para su posterior ajuste de parámetros.

4. Resultados Esperados

Se espera obtener índices en memoria secundaria eficientes, tanto en espacio como en tiempo, para el procesamiento de consultas en bases de datos textuales. Los mismos serán evaluados tanto analíticamente como empíricamente.

5. Recursos Humanos

El trabajo desarrollado en esta línea forma parte del desarrollo de un Trabajo Final de la Licenciatura, dos Tesis de Maestría y una Tesis de Doctorado, todas ellas en el ámbito de Ciencias de la Computación en la Universidad Nacional de San Luis.

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

[6] R. González and G. Navarro. A compressed text index on secondary memory. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 71:127–154, 2009.

[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] 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.

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

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

[14] J. Na and K. 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.

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

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

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

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