Indexando Bases de Datos de Texto

Norma Herrera, Carina Ruano, Darío Ruano, Susana Esquivel

Departamento de Informática

Universidad Nacional de San Luis, Argentina

{nherrera, cmruano, dmruano, 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.

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

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 [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.

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 [14]. 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 [3] 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 [14].

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 [3, 11]. 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

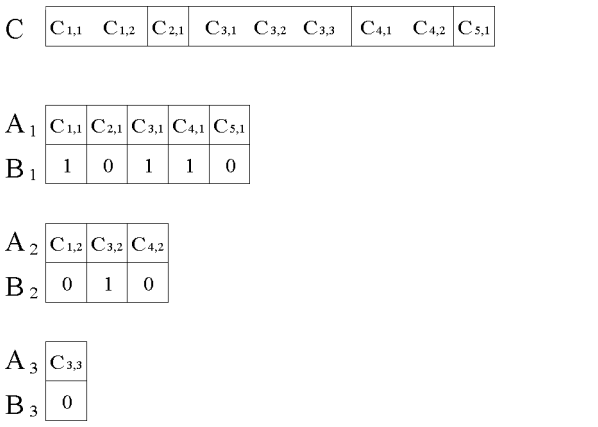

Figura 1: *Representación de una secuencia de códigos de longitud variable usando DAC.*

índices comprimidos en memoria secundaria es un tema de creciente interés.

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. 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 1 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.

3.2. Trie de Sufijos

Un ***trie de sufijos*** es un índice que permite resolver eficientemente las operaciones *count* y *locate* pero que necesita en espacio 10 veces el tamaño del texto indexado. Por esta razón es importante contar con una técnica de paginación que permita mantener el índice en memoria secundaria pero resolviendo eficientemente las búsquedas sobre el texto indexado. Para lograr esto, como primer paso debemos contar con una representación que sea adecuada para memoria secundaria, es decir, una representación que secuencialice la estructura del árbol.

La representación habitual de un trie consiste en mantener en cada nodo los punteros a sus hijos, junto con el rótulo correspondiente a cada uno de ellos. Existen distintas variantes de representación que consisten en organizar estos punteros a los hijos sobre una lista secuencial, sobre una lista vinculada o sobre una tabla de hashing [8]. Una de las propuestas de representación que mejor desempeño tiene en memoria principal es la de Kurtz, quien basándose en la idea de la representación sobre una lista vinculada, propuso que cada nodo mantenga un apuntador al primer hijo y almacenar los nodos hermanos en posiciones consecutivas de memoria. Esto permite durante una búsqueda, realizar una búsqueda binaria sobre los rótulos para decidir por cual hijo seguir.

Nuestra propuesta de representación de un trie de sufijos surge como una extensión de la propuesta hecha en [2, 6] a árboles r-arios. Dicha representación permitirá por una lado reducir el espacio necesario para almacenar el índice, dado que no existirán los punteros a los hijos, y por otro facilitará un posterior proceso de paginado.

El objetivo principal es lograr una representación del trie de sufijos que permita un posterior proceso de paginación en disco. El proceso de paginación de un índice consiste en dividir el mismo en partes, cada una de las cuales se aloja en una página de disco. Luego el proceso de búsqueda consiste en ir cargando en memoria principal una parte, realizar la búsqueda en memoria principal sobre esa parte, para luego cargar la siguiente y proseguir la búsqueda.

Cuando un índice se maneja en disco, el costo de búsqueda queda determinado por la cantidad de accesos a disco realizadas [13]. Aun así, es importante no descuidar las operaciones que se hacen en memoria principal a fin de lograr un funcionamiento eficiente del índice. Es por esta razón que es necesario evaluar el desempeño en memoria principal de la representación que hemos propuesto.

Hemos implementado y evaluado experimentalmente la representación de Kurtz y nuestra propuesta de representación secuencial. Los resultados obtenidos nos han permitido concluir que la representación secuencial

logra mejorar en espacio a la representación de Kurtz pero no así en tiempo. Sin embargo la representación secuencial tiene la ventaja de permitir un posterior paginado del índice.

Actualmente estamos implementando la técnica de compresión *Directly Addressable Variable-Length Code (DAC)* [1], para reducir aún mas el espacio ocupado por la representación secuencial. El objetivo es analizar la reducción de espacio lograda y el impacto que tiene en los tiempos de count y locate. Posterior a ello, implementaremos una técnica de paginación, rediseñando los algoritmos de creación y búsqueda para esta nueva versión del trie.

3.3. Locally Compressed SA

El Locally Compressed Suffix Array (LCSA) [4] 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 [12]. 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 [7].

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.

El Compact Pat Tree (CPT) consiste en representar un árbol de sufijos en memoria secundaria y en forma compacta.

En [6] 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 Ψ[i] = j si A[j] = A[i] +1.

El algoritmo de construcción del LCSA en memoria secundaria fue propuesto en [5]. 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.

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] 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] 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.

[4] 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.

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

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

[7] 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.

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

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

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

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