Indexando Bases de Datos no Convencionales para Memorias Jerárquicas *

Cristian Bustos, Verónica Ludueña, Nora Reyes
Departamento de Informática, Universidad Nacional de San Luis.
Tel. 420822 int. 257 – Fax 430224
Ejército de los Andes 950, San Luis
{cjbustos,vlud,nreyes}@unsl.edu.ar
y
Gonzalo Navarro
Departamento de Ciencias de la Computación, Universidad de Chile.
gnavarro@dcc.uchile.cl

1. Introducción y Motivación

La alarmante velocidad de crecimiento de los datos disponibles en forma digital, se condice con un paralelo crecimiento de las capacidades de almacenamiento a precios más moderados. Por otro lado, mientras la velocidad de procesamiento de la CPU se ha duplicado cada 18 meses, la de los almacenamientos masivos ha progresado poco. Sin embargo han aparecido memorias caché con mayor capacidad, más rápidas y más pequeñas, aunque más costosas, que las memorias RAM; algunas tienen incluso varias capas de memorias que poseen diferencias significativas de eficiencia de un nivel al siguiente. Antes almacenar datos en forma comprimida conllevaba un costo en términos de velocidad de procesamiento por la descompresión. Hoy en día, la diferencia entre los tiempos de la CPU y disco es tan significativa que el esfuerzo de descompresión se paga a cambio de una pequeña disminución en el tiempo de I/O. Además, la transferencia de datos sobre una red local cuesta aproximadamente lo mismo que la transferencia al disco, por lo cual ésta se ve favorecida con la compresión. Este panorama ha promovido varias líneas de investigación las cuales tienen en cuenta estas arquitecturas: las **estructuras de datos compactas** con distintas variantes (las *sucintas* y las *comprimidas*) y las **estructuras de datos con I/O eficiente**.

Nuestro objetivo es contribuir a estas líneas de investigación, diseñando estructuras de datos más eficientes para memorias jerárquicas, haciendo uso de la compacticidad o la I/O eficiente. Particularmente nos centraremos en las estructuras de datos capaces de manipular los siguientes tipos de datos: secuencias, textos, árboles, grafos, y espacios métricos, entre otros, y en estudiar los problemas desde ambos puntos de vista teórico y empírico. Además de diseñar estructuras de datos estáticas, planeamos investigar otros aspectos tales como la construcción eficiente (en espacio o en términos de la I/O), el dinamismo (es decir actualizaciones eficientes) y operaciones de búsqueda complejas (más allá de las básicas soportadas por las estructuras de datos).

2. Estructuras de Datos Compactas

Las estructuras de datos *compactas*, son variantes de sus contrapartes clásicas funcionando en espacio reducido y se pueden dividir en dos grupos: *sucintas* y *comprimidas*. Una estructura de datos se denomina sucinta si necesita espacio asintóticamente despreciable sobre los datos en bruto. Una estructura de datos se dice comprimida cuando toma espacio proporcional al de la secuencia comprimida (donde hay una cierta libertad para elegir un método razonable de compresión). Una medida popular de compresibilidad de secuencias es la entropía de orden k-ésimo (H_k), según lo definido por Manzini [10], que mide la complejidad del espacio para cada secuencia individual, sin ninguna suposición sobre la entrada. La medida H_k es un límite inferior para el número de bits de salida cuando se com-

*Este trabajo ha sido financiado parcialmente por el Núcleo Milenio Centro de Investigación de la Web, Proyecto P04-067-F, Mideplan, Chile (último autor).

prime S con cualquier compresor que codifique cada símbolo de la entrada que dependa solamente de los k símbolos precedentes. Esto abarca los estándares populares tales como PPM, la familia de Lempel-Ziv (gzip, zip, arj, winzip, etc.) y compresores basados en Burrows-Wheeler (bzip2).

Sea s el alfabeto de una secuencia de símbolos S, $n(i)$ el número de ocurrencias del i-ésimo símbolo, y n la longitud de S. Entonces $H_0(S) = \Sigma(n(i)\log(n/n(i))$. Sea w una secuencia de longitud k y $w(S)$ la subsecuencia de símbolos de S que siguen a w, entonces $H_k(S) = 1/n\Sigma(|w|H_0(w))$ sobre todos los posibles w. Esta medida es popular también en las estructuras de datos comprimidas por ejemplo existen índices comprimidos que codifican un texto T de n símbolos sobre un alfabeto s usando espacio de $H_k(T)+o(n\log s)$ bits y además pueden buscar eficientemente patrones en el texto. Observar que la codificación llana del texto toma $n\log s$ bits. La idea de las estructuras de datos compactas, a diferencia de la compresión pura, es la capacidad de manipular los datos en forma comprimida, sin tener que descomprimirlos primero. En la actualidad las estructuras de datos compactas pueden manipular secuencias de bits o de símbolos generales; árboles, grafos,colecciones de texto, permutaciones y mapping, sumas parciales, búsqueda por rango, y así sucesivamente.

Si consideramos la complejidad de las operaciones de I/O en el modelo más simple, asumimos que los datos están almacenados en bloques de disco y los bloques se acceden como unidades atómicas: el costo de I/O es el *número de bloques accedidos* durante un cómputo. Esto abstrae las características esenciales de la tecnología de memoria secundaria actual, aún desatendiendo datos más precisos tales como los accesos secuenciales. Esta simplificación se debe al interés de tener un modelo más fácil de analizar, y porque en la práctica es difícil controlar dónde se ubican los bloques realmente en disco.

3. Búsqueda en Texto con Estructuras Comprimidas

El problema de la *búsqueda aproximada de un patrón en el texto* puede verse como: dado un *texto* $T = T[1, n]$ y un *patrón* $P = P[1, m]$ en el alfabeto Σ (con $m << n$) y un entero k, se desea encontrar todas las posiciones i en el texto tal que una ocurrencia aproximada de P, con a lo más k diferencias termine en i. La diferencia entre dos strings α y β es medida con la *distancia de edición* d; $d(\alpha, \beta)$ es el mínimo número de *operaciones de edición* (inserciones, eliminaciones y/o sustituciones de caracteres) que se deben realizar para convertir β en α. La búsqueda aproximada de patrones tiene aplicaciones tales como la recuperación de texto, biología computacional, comunicaciones de datos y data mining, entre otras.

Filtración con q-gramas. La solución básica al problema de búsqueda aproximada de patrones se basa en la *programación dinámica* [9, 14], que tiene una complejidad de $O(kn)$ tiempo lo cual es lento para textos grandes. Se puede mejorar introduciendo algún esquema de *filtración* para extraer las áreas del texto con potenciales coincidencias.

Potenciales coincidencias aproximadas pueden ser reconocidas por la distribución de los caracteres [6], localizando bloques de coincidencias del patrón en el texto [17], recuperando substrings de coincidencia maximales del patrón en el texto [3, 15], o aún aplicando la idea de Boyer-Moore de búsqueda exacta de un patrón [1] a la búsqueda aproximada de un patrón [13]. Una familia muy popular de esquemas de filtración, llamada la *q-famlia*, establece la condición de filtración en términos de *q-gramas*. Por q-grama nos referiremos a cualquier (sub)string de q caracteres. Una coincidencia aproximada se parece al patrón original, además cualquier coincidencia aproximada debe contener al menos algún q-grama común con el patrón buscado. De esta forma, la aplicabilidad de los q-gramas en una condición de filtración es intuitivamente clara.

Cada miembro de la q-familia se basa en su condición de filtración característica, aunque todos ellos deben encontrar en el texto q-gramas que coincidan con los q-gramas del patrón, esto se puede realizar de dos maneras: *dinámica* que escanea el texto en tiempo lineal, y *estática* que utiliza un *índice de q-gramas*. La propuesta estática es superior cuando el mismo índice puede usarse varias veces.

Implementación de índices de q-gramas. Hay distintas implementaciones para índices de q-gramas para los métodos estáticos. La básica es un arreglo de punteros de tamaño $|\Sigma|^q$. Cada posición del arreglo referencia a la lista de ocurrencias del correspondiente q-grama en el texto. El índice puede construirse en tiempo $O(n + |\Sigma|^q)$ [7], aunque puede resultar en índices inútilmente grandes. Una mejora a este esquema de indexación es reducir el tamaño del arreglo de punteros por medio de un hashing, llevando el tamaño y el tiempo de construcción a $O(n)$ sin

una demora significativa en las búsquedas.

Otra solución usa una estructura de trie. Los q-gramas del texto son almacenados en un trie, en el que cada hoja contiene la lista de ocurrencias del correspondiente q-grama en el texto. El trie puede construirse en tiempo $O(n)$ si se usa un algoritmo similar al de la creación del árbol de sufijos [16].

Las implementaciones anteriores encuentran todas las ocurrencias de un q-grama dado en tiempo óptimo de $O(q+l)$, donde l es el número de ocurrencias encontradas. Pero todas ellas sufren el mismo inconveniente: el tamaño del índice se vuelve impráctico al crecer la longitud del texto, porque el espacio utilizado es al menos lineal. Actualmente ambas estructuras pueden implementarse en $O(M)$ espacio, donde M es el número de q-gramas distintos del texto; y se requiere $O(n)$ espacio para las listas de ocurrencias.

Un índice que usa compresión. En este índice *Lempel-Ziv* (LZ) para q-gramas las listas de ocurrencias se reemplazan con una estructura de datos más compacta. La estructura (hashing o trie) para los distintos q-gramas, ahora llamada *índice primario*, es aún necesaria para proveer un punto de comienzo para las búsquedas.

La representación compacta de las listas de ocurrencias toma ventaja de las *repeticiones* en el texto. La primera ocurrencia del string será llamada *definición* y las siguientes se llamarán *frases*. Luego, cada ocurrencia de un q-grama es o el primero de su clase o parte de alguna frase. La primera ocurrencia se almacenará en el índice primario y las demás serán encontradas usando la información sobre repeticiones.

El índice LZ encontrará todas las ocurrencias de un q-grama dado en tiempo $O(q + l)$, el mismo utilizado por los índices de q-gramas tradicionales. El tamaño es $O(M + N)$, siendo M el número de q-gramas distintos en el texto y N el de repeticiones necesarias para cubrir todos los q-gramas (salvo la primer ocurrencia de cada q-grama distinto). $M+N$ es a lo sumo n, pero puede ser mucho menor.

4. Optimización de Estructuras

Existen numerosas estructuras para búsquedas por similitud en espacios métricos, pero sólo unas pocas trabajan eficientemente en espacios de alta o mediana dimensión, y la mayoría no admiten dinamismo, ni están diseñadas para trabajar sobre grandes volúmenes de datos; es decir, en memoria secundaria. Por lo tanto, estudiamos distintas maneras de optimizar algunas de las estructuras que han mostrado buen desempeño, con el fin de optimizarlas teniendo en cuenta la jerarquía de memorias.

4.1. *SATD*

Hemos desarrollado una estructura para búsqueda por similitud en espacios métricos llamado *Árbol de Aproximación Espacial Dinámico (SATD)* [12] que permite realizar inserciones y eliminaciones, manteniendo un buen desempeño en las búsquedas. Muy pocos índices para espacios métricos son completamente dinámicos. Esta estructura se basa en el *Árbol de Aproximación Espacial* [11], el cual había mostrado un muy buen desempeño en espacios de mediana a alta dimensión, pero era completamente estático.

El *SAT* está definido recursivamente; la propiedad que cumple la raíz a (y a su vez cada uno de los nodos) es que los hijos están más cerca de la raíz que de cualquier otro punto de S. La construcción del árbol se hace también de manera recursiva. De la definición se observa que se necesitan de antemano todos los elementos para la construcción y que queda completamente determinado al elegirle una raíz, lo cual en el *SAT* original se realizaba al azar.

4.1.1. Elección de la raíz en el *SATD*

El *SATD* se construye incrementalmente, tomando como raíz el primer elemento insertado. Sin embargo, ya hemos visto que se obtienen mejores resultados en las búsquedas si se elige la raíz con más información sobre el espacio métrico considerado. Luego, pensamos adaptar los distintos métodos de selección de raíz que mostraron buen desempeño en el *SAT*, y estudiar su comportamiento en la versión dinámica. En este caso deberíamos demorar la selección de la raíz hasta conocer un razonable número de objetos de la base de datos, pero sin perder el dinamismo.

Esperamos con esto que, tal como sucedió en el *SAT*, logremos brindar una estructura dinámica más eficiente en las búsquedas gracias a tener una mejor raíz para el árbol.

4.1.2. Actualización de *SATD*

Para el desarrollo del *SATD* [12] se han estudiado distintas maneras de realizar incorporaciones de nuevos elementos sobre el árbol, pero se optó por un método que inserta un nuevo elemento en un punto

determinado del árbol, manteniendo la aridad acotada. Gracias a esos trabajos, quedó demostrado que existen otros posibles puntos de inserción válidos, aunque no se analizó cuál de ellos sería el mejor.

Por lo tanto, tiene sentido considerar si los otros puntos posibles de inserción para un elemento consiguen mejorar aún más los costos de búsqueda. Así, como resultado se espera brindar una estructura que mejore el comportamiento durante las búsquedas gracias a elegir adecuadamente los puntos de inserción de los nuevos elementos.

4.1.3. *SATD* con Clustering

El *SATD* es una estructura que realiza la partición del espacio considerando la proximidad espacial; pero, si el árbol agrupara los elementos que se encuentran muy cercanos entre sí, lograría mejorar las búsquedas al evitar recorrerlo para alcanzarlos.

Podemos pensar entonces que construimos un *SATD*, en el que cada nodo representa un grupo de elementos muy cercanos ("clusters") y relacionamos los clusters por su proximidad en el espacio. La idea sería que en cada nodo se mantenga el centro del cluster correspondiente, y se almacenen los k elementos más cercanos a él; cualquier elemento a mayor distancia del centro que los k elementos, pasaría a formar parte de otro nodo en el árbol.

Esperamos así obtener una estructura más adecuada a espacios en los que se sabe de antemano que pueden existir "clusters" de elementos, aprovechándonos de la existencia de los mismos para mejorar las búsquedas, lo que también nos podría permitir mejorar su alojamiento en memoria secundaria.

4.1.4. *SATD* en Memoria Secundaria

Hemos desarrollado una versión del *SATD* que funciona adecuadamente en memoria secundaria y actualmente estamos evaluando su desempeño en las búsquedas y su dinamismo, comparándolo contra otras estructuras que poseen estas propiedades.

Así, eligiendo la estructura más apta, sería posible pensar en extender apropiadamente el álgebra relacional y diseñar soluciones eficientes para los nuevos operadores, teniendo en cuenta aspectos no sólo de memoria secundaria, sino también de concurrencia, confiabilidad, etc. Algunos ejemplos de las operaciones que podrían ser de interés resolver son: join espacial, operaciones de conjuntos y otras operaciones de interés en bases de datos espaciales tales como los operadores topológicos. Algunos de estos problemas ya poseen solución en las bases de datos espaciales, pero no en el ámbito de los espacios métricos.

En este caso no sólo se busca minimizar la cantidad de cálculos de la función de distancia, sino también la cantidad de operaciones de E/S, lo que permitiría utilizarla en aplicaciones reales, tales como búsquedas en la web.

5. Dimensión Intrínseca

En los espacios de vectores, la "maldición de la dimensionalidad" describe el fenómeno por el cual el desempeño de todos los algoritmos existentes se deteriora exponencialmente con la dimensión. En espacios métricos generales la complejidad se mide como el número de cálculos de distancias realizados, pero la ausencia de coordenadas no permite analizar la complejidad en términos de la dimensión.

En los espacios vectoriales existe una clara relación entre la dimensión (*intrínseca*) del espacio y la dificultad de buscar. Se habla de "intrínseca", como opuesta a "representacional". Los algoritmos más ingeniosos se comportan más de acuerdo a la dimensión intrínseca que a la representacional. Hay varios intentos de medir la dimensión intrínseca en espacios de vectores, como la transformada de *Karhunen-Loève (KL)* y otras más fáciles de computar; medidas tales como *Fastmap* [5]. Otro intento útil para medir la dimensión intrínseca, de espacios de vectores no uniformemente distribuidos, es la *dimensión fractal* [2].

Existen sólo unas pocas propuestas diferentes sobre cómo estimar la dimensión intrínseca de un espacio métrico tales como el *exponente de la distancia* (basada en una ley de potencias empírica observada en muchos conjuntos de datos) [8], y la medida de dimensión intrínseca como una medida cuantitativa basada en el histograma de distancias [4]. Además, también parece posible adaptar algunos de los estimadores de distancia para espacios de vectores para aplicarlos a espacios métricos generales, como por ejemplo *Fastmap* y *dimensión fractal*.

En aplicaciones reales de búsqueda en espacios métricos, sería muy importante contar con un buen estimador de la dimensión intrínseca porque nos permitiría decidir el índice adecuado a utilizar en función de la dimensión del espacio. Además, tener una buena estimación de la dimensión nos permitiría, en algunas ocasiones, elegir la función de distancia de manera tal que se obtenga una menor dimensión.

6. Trabajos Futuros

Como trabajo futuro de esta línea de investigación se consideran varios aspectos relacionados al diseño de estructuras de datos que concientes de que existe una jerarquía de memorias, saquen el mejor partido de la misma haciendólas eficientes tanto en espacio como en tiempo. Vamos a trabajar en particular con estructuras de datos compactas para textos y en optimización de estructuras de datos para espacios métricos.

Implementaremos un índice LZ para q-gramas y estudiaremos su comportamiento en comparación con los índices de q-gramas tradicionales. Además, consideraremos varias maneras de optimizar el *SATD*, una estructura de datos para espacios métricos, que ha mostrado ser competitiva por su desempeño en las búsquedas y que también es dinámica. Intentaremos mejorarlo adaptándolo más al espacio métrico particular considerado y también al nivel de la jerarquía de memorias en que se deba almacenar. Es importante destacar que estos estudios sobre espacios métricos y sobre el *SATD* nos permitirán no sólo mejorar el desempeño del *SATD* sino que también muchos de los resultados que obtengamos podrán eventualmente aplicarse a otras estructuras de datos para espacios métricos.

Referencias

[1] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[2] Francesco Camastra. Data dimensionality estimation methods: a survey. *Pattern Recognition*, 36(12):2945–2954, 2003.

[3] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.

[4] E. Chávez and G. Navarro. Towards measuring the searching complexity of metric spaces. In *Proc. International Mexican Conference in Computer Science (ENC'01)*, volume II, pages 969–978, Aguascalientes, México, 2001. Sociedad Mexicana de Ciencias de la Computación.

[5] Christos Faloutsos and King-Ip Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 163–174. ACM Press, 1995.

[6] Roberto Grossi and Fabrizio Luccio. Simple and efficient string matching with k mismatches. *Inf. Process. Lett.*, 33(3):113–120, 1989.

[7] Petteri Jokinen and Esko Ukkonen. Two algorithms for approximate string matching in static texts. In *MFCS*, pages 240–248, 1991.

[8] Caetano Traina Jr., Agma J. M. Traina, and Christos Faloutsos. Distance exponent: A new concept for selectivity estimation in metric trees. In *ICDE*, page 195, 2000.

[9] Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.

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

[11] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[12] G. Navarro and N. Reyes. Fully dynamic spatial approximation trees. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, LNCS 2476, pages 254–270. Springer, 2002.

[13] Jorma Tarhio and Esko Ukkonen. Approximate boyer-moore string matching. *SIAM J. Comput.*, 22(2):243–260, 1993.

[14] Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.

[15] Esko Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.

[16] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[17] Sun Wu and Udi Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.