Métodos de Acceso por Similitud *

Edgar L. Chávez
Escuela de Ciencias Físico - Matemáticas
Universidad Michoacana - México
elchavez@fismat.umich.mx

Norma E. Herrera, Carina M. Ruano, Ana V. Villegas
Departamento de Informática
Universidad Nacional de San Luis
{nherrera,cmruano,anaville}@unsl.edu.ar
Fax: +54 (2652) 430224

1 Introducción

La búsqueda de elementos cercanos o similares a uno dado es un problema que aparece en diversas áreas. Este concepto fue motivado como una extensión natural del concepto de búsqueda exacta, ante el surgimiento de nuevos tipos de bases de datos tales como base de datos de imágenes, de sonido, de texto, etc. Estructurar estos tipos de datos en registros para adecuarlos al concepto tradicional de búsqueda exacta, es difícil en muchos casos y hasta imposible si la base de datos cambia más rápido de lo que se puede estructurar (como por ejemplo la web). Aún cuando pudiera hacerse, las consultas que se pueden satisfacer con la tecnología tradicional están limitadas a variaciones de la búsqueda exacta.

Nos interesan las búsquedas en donde se puedan recuperar objetos similares a uno dado. Este tipo de búsqueda, se conoce con el nombre de *búsqueda por proximidad o búsqueda por similitud*, y surge en áreas tales como reconocimiento de voz, reconocimiento de imágenes, compresión de texto, recuperación de texto, biología computacional, etc. La necesidad de una respuesta rápida y adecuada, y un uso eficiente de memoria, hacen necesaria la existencia de estructuras de datos especializadas que incluyan estos aspectos.

La problemática de búsquedas por similitud en bases de datos no tradicionales puede formalizarse por medio del modelo de *espacios métricos*. Un espacio métrico es un par $(\mathcal{X}, d)$, donde $\mathcal{X}$ es un conjunto de objetos y $d : \mathcal{X} \times \mathcal{X} \rightarrow R^+$ es una función de distancia que modela la similitud entre los elementos de $\mathcal{X}$. La función d cumple con las propiedades características de una función de distancia: $\forall x, y \in \mathcal{X},\ d(x, y) \geq 0$ (positividad), $\forall x, y \in \mathcal{X},\ d(x, y) = d(y, x)$ (simetría), $\forall x, y, z \in \mathcal{X},\ d(x, y) \leq d(x, z) + d(z, y)$(desigualdad tringular). La base de datos es un conjunto finito $\mathcal{U} \subseteq \mathcal{X}$.

Una de las consultas típicas que implica recuperar objetos similares de una base de datos es la *búsqueda por rango*, que denotaremos con $(q, r)_d$. Dado un elemento de consulta q, al que llamaremos *query* y un radio de tolerancia r, una búsqueda por rango consiste en recuperar aquellos objetos de la base de datos cuya distancia a q no sea mayor que r.

*Este trabajo es parcialmente subvencionado por CYTED VII.19 RIBIDI Project (todos los autores) y por CONACyT 36911-A (Edgar Chávez)

Una forma trivial de resolver este tipo de búsquedas es examinando exhaustivamente la base de datos, es decir, comparando cada elemento de la base datos con la query. Pero en general, esto es demasiado costoso para aplicaciones reales. Para evitar esta situación, se preprocesa la base de datos por medio de un *algoritmo de indización* con el objetivo de construir una *estructura de datos o índice*, diseñada para ahorrar cálculos en el momento de resolver una búsqueda.

El tiempo total de resolución de una búsqueda puede ser calculado de la siguiente manera:

T= #evaluaciones de $d\times$ *complejidad(d) + tiempo extra de CPU + tiempo de I/O*

En muchas aplicaciones la evaluación de la función d es tan costosa que las demás componentes de la fórmula anterior pueden ser despreciadas. Éste es el modelo usado en la mayoría de los trabajos de investigación hechos en esta temática. Sin embargo, hay que prestar especial atención al tiempo extra de *CPU*, dado que reducir este tiempo produce que en la práctica la búsqueda sea más rápida, aún cuando estemos realizando la misma cantidad de evaluaciones de la función d. De igual manera, el tiempo de *I/O* puede jugar un papel importante en algunas aplicaciones, dependiendo de la memoria principal disponible y del costo relativo de computar la función d. Los trabajos sobre espacios métricos, generalmente, se han enfocado en algoritmos para descartar elementos en tiempo de búsqueda, dejando de lado las consideraciones sobre el tiempo de *I/O*. La única excepción, ha sido el *MTree* [7], diseñado específicamente para memoria secundaria.

Otro punto importante, es que la mayoría de las estructuras para espacios métricos se construyen bajo el supuesto de que el conjunto de datos es estático. En muchas aplicaciones esto no es razonable, dado que los elementos son insertados y eliminados dinámicamente. Algunas estructuras toleran inserciones pero muy pocas eliminaciones.

Básicamente existen dos enfoques para el diseño de algoritmos de indización en espacios métricos: uno está basado en particiones compactas o tipo Voronoi y el otro está basado en pivotes [5]. Nuestro trabajo se ha centrado en los algoritmos basados en pivotes.

La idea subyacente de los algoritmos basados en pivotes es la siguiente. Se seleccionan k pivotes $\{p_1, p_2, \ldots, p_k\}$, y se le asigna a cada elemento a de la base de datos, el vector o firma $\Phi(a) = (d(a,p_1), d(a,p_2), \ldots, d(a,p_k))$. Ante una búsqueda $(q,r)_d$, se usa la desigualdad triangular junto con los pivotes para filtrar elementos de la base de datos sin medir su distancia a la query q. Para ello se computa la firma de la query q, $\Phi(q) = (d(q,p_1), d(q,p_2), \ldots, d(q,p_k))$, y luego se descartan todos aquellos elementos a, tales que para algún pivote $p_i \mid d(q,p_i) - d(a,p_i) \mid > r$.

2 Fixed Queries Trie

La familia de estructuras *FQ* (FQT [2], FHQT [2, 1], FQA [4], FQtrie [6]) forman parte de las estructuras basadas en pivotes, y son algunas de las pocas que soportan tanto inserciones como eliminaciones. En el caso particular del FQtrie, si bien en su concepción original permite inserciones y eliminaciones, hasta el momento no se ha realizado una implementación del mismo en la que se incorporen estas operaciones.

El punto de partida de nuestro trabajo es el FQtrie. Esta estructura fue presentada recientemente [6] y por consiguiente no se ha realizado hasta el momento un estudio exhaustivo de la misma. El objetivo es lograr una implementación eficiente; no sólo en términos de cantidad de evaluaciones de la función de distancia d sino también en tiempo extra de *CPU*, totalmente dinámica y con manejo de

espacios métricos cuyo índice completo exceda la capacidad de la memoria principal.

Detallamos a continuación el trabajo que hemos realizado hasta el momento.

2.1 Selección de pivotes

Los mejores pivotes son aquellos que durante una búsqueda permiten filtrar una mayor cantidad de objetos dado que, mientras mayor sea la cantidad de elementos filtrados, menor es la cantidad la cantidad de evaluaciones de la función d que debemos realizar para responder la consulta.

Por lo tanto la política usada para seleccionar los pivotes afecta notablemente la performance de la búsqueda [5, 3, 8, 10]. Esto significa que si tenemos dos conjuntos de pivotes del mismo tamaño elegir el mejor de los dos permite reducir el tiempo de búsqueda. Por otro lado, un grupo pequeño de pivotes bien elegidos puede resultar tan eficiente como un grupo de mayor cantidad de pivotes pero elegidos aleatoriamente. En consecuencia, el tema de selección de un buen grupo de pivotes para indizar un determinado espacio métrico está siendo ampliamente estudiado.

Nuestra propuesta aquí es una optimización local al problema de selección de un buen grupo de pivotes. En lugar de seleccionar durante la construcción del índice un grupo de pivotes que sea efectivo para todo el espacio métrico, *seleccionamos durante la búsqueda un grupo de pivotes que sea efectivo para la query* q.

Para ello, construimos varios índices sobre el espacio con distintos grupos de pivotes (elegidos aleatoriamente). En el momento de realizar una búsqueda $(q, r)_d$ seleccionamos aquel índice que sea más adecuado a q de acuerdo al conjunto de pivotes con el que fue construido. Esta idea de optimización local a una query q permite además realizar consultas en paralelo, si cada uno de los índices creados se mantiene en distintas máquinas de una red.

Hemos definido e implementado varias políticas de selección del índice adecuado bajo la perspectiva de optimización local de una query q. Una explicación detallada de estas políticas de selección de pivotes, como así también resultados experimentales de la evaluación de las mismas, pueden consultarse en [9]. Esperamos publicar estos resultados en futuros congresos.

2.2 Tiempo extra de CPU

Las *tablas lookup*, propuestas en [6], han demostrado ser una buena opción para mejorar el desempeño del FQA y del *scan secuencial*.

Recordemos que, dada una búsqueda $(q, r)_d$, los elementos no relevantes para la query son aquellos a tales que para algún p_i se cumple que $\mid d(q, p_i) - d(a, p_i) \mid > r$. Esto significa que si $d(a, p_i)$ se codifica en b_i bits, debemos realizar operaciones de enmascaramiento y corrimiento para evaluar la condición anterior. Una tabla lookup es una estrategia de representación de $\Phi(q)$, que permite realizar comparaciones entre palabras de máquina completas en lugar de hacerlas por grupos de b_i bits (donde b_i es la cantidad de bits necesitados para codificar $d(a, p_i)$). Hemos realizado una implementación del FQTrie, utilizando las tablas lookup para mejorar su desempeño.

Otro de los parámetros que afecta el tiempo extra de CPU, y que es necesario establecer en el momento de construcción de la estructura, es la cantidad de bits que utiliza cada pivote. En este sentido, el objetivo es caracterizar la cantidad de bits que se le asignará a cada pivote, y el rango de

cada intervalo correspondiente. No es claro cuál es la mejor selección, ni si cada pivote debería tener el mismo número de bits y los mismos intervalos.

2.3 Tiempo de I/O

Tal como lo mencionáramos anteriormente otro objetivo del trabajo es lograr una implementación totalmente dinámica y con manejo de espacios metricos cuyo índice completo exceda la capacidad de la memoria principal.

Para este último caso, en lugar de modificar el FQtrie para que sea eficiente su manejo en disco, particionamos el espacio metrico de manera tal que el índice de cada una de la partes entre en memoria principal. Luego, una búsqueda $(q, r)_d$ se resuelve buscando separadamente en cada uno de los índices, lo que puede ser hecho en memoria principal y en paralelo.

Con respecto al particionamiento, hemos implementado dos estrategias para dividir el espacio. En la primera se agrupan elementos en una partición hasta completar una página de disco. De esta forma nos aseguramos que los índices correspondientes a tales particiones pueden ser fácilmente contenidos en memoria principal. En la segunda se agrupan elementos en una partición mientras el índice correspondiente a los elementos agrupados hasta el momento no supere el tamaño de una página de disco.

También al particionar se analiza la posibilidad de que los elementos puedan ser mayores que el tamaño de una página de disco. Con dicha posibilidad la primer política se vuelve obsoleta. Para solucionar este problema, en esos casos se trabaja con apuntadores a los objetos reales.

El realizar inserciones y eliminaciones puede provocar desbalances en las cardinalidades de las partes. Esto debe ser tomado en cuenta para decidir el momento y el modo apropiado para reorganizar las particiones. Hasta el momento se han planteado e implementado dos opciones: una consiste en reorganizar las partes en períodos de tiempo establecidos, y la otra consiste en mantener organizadas las particiones a medida que se producen los cambios.

3 Trabajo Futuro

El trabajo futuro se puede resumir en los siguientes puntos:

- Establecer experiementalmente la cantidad de bits que se le debe asignar a cada pivote y el rango de cada intervalo correspondiente. Esperamos que los resultados de estos experimentos nos permitan establecer alguna relación entre la bondad del pivote y el espacio (cantidad de bits) que se le debe asignar al mismo.
- Realizar la evaluación experimental de las técnicas de particionamiento y de las estrategias de reorganización de las particiones, a fin de seleccionar la más adecuada.
- Continuar con el estudio de políticas de selección de pivotes a fin de poder poder caracterizar buenos grupos de pivotes.

- Hasta el momento los experimentos se han realizado usando como espacios métricos diccionarios de palabras con la función de distancia de edición. Nos proponemos experimentar las técnicas propuestas sobre otros tipos de espacios métricos.

Referencias

[1] R. Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker Inc., 1997.

[2] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[3] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. In *Proc. of the XXI Conference of the Chilean Computer Science Society (SCCC'01)*, pages 33–40. IEEE CS Press, 2001.

[4] E. Chávez, J. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135, 2001.

[5] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[6] E. Chávez. Faster proximity searching using lookup tables. Technical report, Universidad Michoacana, México, 2002.

[7] Paolo Ciaccia, Marco Patella, Fausto Rabitti, and Pavel Zezula. Indexing metric spaces with m-tree. In *Sistemi Evolui per Basi di Dati*, pages 67–86, 1997.

[8] A. Faragó, T. Linder, and G. Lugosi. Fast nearest-neighbor search in dissimilarity spaces. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 15(9):957–962, 1993.

[9] N. Herrera. Diseño e implementación de estructuras optimizadas para búsquedas en espacios métricos. *Tesis de Maestría en Ciencias de la Computación, Universidad Nacional de San Luis*, 2003.

[10] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.