Métodos de Paginación Para Índices Métricos Basados en Pivotes *

Ana Valeria Villegas
Dpto de Informática
Univ. Nacional de San Luis
Argentina
anaville@unsl.edu.ar

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

Norma Edith Herrera
Dpto de Informática
Univ. Nacional de San Luis
Argentina
nherrera@unsl.edu.ar

Resumen

El problema de buscar objetos en una base de datos que sean similares a uno dado puede formalizarse por medio del modelo de *Espacios Métricos*. La mayoría de las soluciones existentes para búsquedas por similitud en espacios métricos suponen que tanto el espacio como el índice completo entran en memoria principal.

En este artículo presentamos una implementación del *Fixed Queries Trie (FQtrie)* que permite manejar espacios métricos cuyo índice completo y/o datos exceda la capacidad de la memoria principal. Para ello, en lugar de modificar la estructura para que sea eficiente en memoria secundaria, particionamos el espacio de manera tal que cada una de las partes entre en memoria principal, las que posteriormente se indexan en forma separada. Luego, una búsqueda se resuelve buscando en cada parte, lo que puede ser hecho en memoria principal y en paralelo.

Para particionar el espacio hemos diseñado un método basado en la distancia LCS (longest common subsequence). Mostramos experimentalmente que esta forma de particionar, ante una búsqueda, implica menor cantidad de accesos a disco que si el espacio se particiona en forma totalmente aleatoria.

Palabras claves: Base de Datos, Búsquedas por Similitud, Índices, Memoria Secundaria.

1. Introducción

La búsqueda de elementos cercanos o similares a uno dado es un problema que aparece en diversas áreas de ciencias de la computación. Este tipo de búsqueda, conocida con el nombre de *búsqueda por proximidad o búsqueda por similitud*, fue motivada como una extensión natural del concepto de búsqueda exacta ante el surgimiento de nuevos tipos de bases de datos, tales como las bases de datos multimedia (bases de datos de imágenes, de sonido, de texto, etc).

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

*Este trabajo ha sido parcialmente subvencionado por CYTED VII.19 RIBIDI Project, por el proyecto CONACyT 36911A y por el proyecto 22/F314 - UNSL

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 triangular). 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.

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*

Hay dos casos importantes a considerar: si el índice y los datos pueden ser mantenidos en memoria principal, o si es necesario utilizar memoria secundaria para los índices y/o datos. En el primer caso el objetivo central es reducir los cálculos de distancia realizados, es decir, un algoritmo se considera eficiente si puede contestar una consulta por similitud realizando un número pequeño de cálculos de distancia, sublineal respecto a la cantidad de elementos en la base de datos. Para los algoritmos en memoria secundaria, además de realizar pocos cálculos de distancia, se requiere que se realicen pocos accesos a disco. En este último tópico los avances han sido más lentos que los realizados para memoria principal.

Los trabajos sobre espacios métricos, generalmente, han supuesto que la función d es tan costosa que las demás componentes involucradas en T puede ser despreciadas. Una excepción ha sido el *MTree* presentado en [6]. Allí se propone e implementa una solución para memoria secundaria y es el único método disponible para usuario final.

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 familia de estructuras *FQ* (FQT [2], FHQT [2, 1], FQA [4], FQtrie [3]) forman parte del grupo de algoritmos basados en pivotes; cada una de ellas fue presentada como una mejora de la anterior.

El punto de partida de nuestro trabajo es el FQtrie. El objetivo es lograr una implementación con manejo de espacios métricos cuyo índice completo exceda la capacidad de la memoria principal. Para esto, en lugar de modificar el FQtrie para que su manejo en disco sea eficiente, particionamos el espacio métrico 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. Esto nos llevaría a pensar que el número de visitas a disco siempre es constante (la cantidad de partes generadas más la cantidad de páginas de índices). Sin embargo, al particionar la base se datos y agrupar en cada parte elementos similares, es posible evitar visitas a ciertas páginas de disco, debido a que la búsqueda en un índice puede indicar que en la parte correspondiente no hay elementos relevantes a la búsqueda, con lo que se evita cargar a memoria una parte.

Este artículo está organizado de la siguiente manera. En la sección 2 comenzamos dando una breve explicación de la estructura FQTrie. La sección 3 está dedicada a la explicación de la estrategia de partición diseñada. En la sección 4 presentamos la evaluación experimental de la misma donde mostramos que la técnica propuesta es competitiva. Finalizamos en la sección 5 dando las conclusiones y el trabajo futuro.

2. Fixed Queries Trie

Esta estructura fue presentada recientemente [3] y forma parte de los familia de estructuras basadas 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 computa $\Phi(q) = (d(q, p_1), d(q, p_2), \ldots, d(q, p_k))$. Luego, se descartan todos aquellos elementos a, tales que para algún pivote p_i, $\mid d(q, p_i) - d(a, p_i) \mid > r$, es decir $max_{1 \leq i \leq k} \mid d(q, p_i) - d(a, p_i) \mid = L_\infty(\Phi(a), \Phi(q)) > r$. Los elementos no descartados por la condición anterior, se comparan directamente con la query q; llamaremos a estos elementos lista de candidatos.

Esto significa que, todos los algoritmos basados en pivotes, proyectan el espacio métrico original en un espacio vectorial k dimensional con la función de distancia L_∞. La diferencia entre todos ellos, radica en cómo implementan la búsqueda en el espacio mapeado. En el FQtrie [3], se utiliza un *Árbol Digital o Trie* [**?**] para indexar las firmas de los elementos de la base de datos.

Un *Trie* es un árbol m-ario para búsqueda lexicográfica. En esta estructura, cada elemento se considera como una secuencia de caracteres sobre un alfabeto Σ; la cardinalidad del alfabeto determina la aridad del árbol, es decir $m = |\Sigma|$. Un nodo en un trie o es un nodo externo y contiene un elemento; o es un nodo interno y contiene m punteros a subtries. Dada una cadena, se usan los caracteres que la conforman para direccionar la búsqueda en el árbol. Estando en un nodo de nivel i, la selección del subtrie que le corresponde se realiza en función del i-ésimo caracter de la cadena. El nodo raíz usa el primer caracter, los nodos hijos de la raíz usan el segundo caracter, y así sucesivamente.

En un trie m-ario la búsqueda toma un tiempo que es proporcional a la longitud de la cadena, independientemente de cual sea el tamaño de la base de datos.

3. Manejo de Espacios Métricos en Memoria Secundaria

Uno de los factores que intervienen en el diseño de estructuras de datos es si la memoria principal tiene la capacidad suficiente como para contener la estructura completa, o si sólo podemos mantener una fracción de la misma. Los parámetros de construcción de una estructura varían dependiendo de esta condición dado que, si la estructura será manejada en disco, el objetivo será diseñarla de manera tal de reducir la cantidad de accesos a disco para recuperar partes de la estructura. Por ejemplo, en un *rebalse o hashing* la cantidad de ranuras (slots) por bucket deber ser 1 si la estructura se maneja en memoria principal, y debe ser mayor que 1 si la estructura se maneja en memoria secundaria.

Tal como ya lo mencionáramos, la mayoría de las investigaciones realizadas sobre espacios métricos han dejado de lado las consideraciones de I/O suponiendo que la memoria principal tiene la capacidad suficiente como para mantener tanto el espacio métrico completo como su índice. Esto puede no ser real y, en estos casos, es el sistema operativo el encargado de mantener en memoria principal las páginas requeridas.

En este artículo presentamos una estrategia para manejar espacios métricos cuyo índice y/o datos no entran en memoria principal; de esta forma podemos controlar los datos que se cargan en memoria principal y reducir así la cantidad de accesos a disco cuando se realiza una consulta. Nuestra propuesta es, en lugar de modificar el FQtrie para que su manejo en disco sea eficiente, *particionar el espacio métrico 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.

Esto significa que cada parte que conforma la partición del espacio se indexa separadamente de las restantes; luego, una búsqueda $(q, r)_d$ se resuelve realizando los siguientes pasos con cada una de

las partes:

- se carga el índice de la parte.
- se busca en él para obtener la lista de elementos candidatos.
- si la lista de candidatos no es vacía, se carga la parte correspondiente en memoria principal, para poder comparar cada elemento de la lista contra la query q.

Dado que la búsqueda en una parte es independiente de la búsqueda en las restantes, se podría resolver $(q, r)_d$ en paralelo si mantenemos el índice de cada una de ellas en distintos nodos de una red.

Un forma trivial de particionar el espacio es dividir el conjunto de datos en forma aleatoria. Si bien esto nos permitiría manejar espacios métricos que excedan la capacidad de memoria principal, podría suceder que ante una búsqueda $(q, r)_d$ tengamos que cargar todas las partes que conforman el espacio porque cada una de ellas contiene un elemento de la respuesta.

Para evitar esta situación, particionamos el espacio agrupando elementos que sean similares entre sí. De esta manera tenemos una mayor probabilidad de que la respuesta a una búsqueda $(q, r)_d$ implique cargar pocas páginas en memoria principal. Es decir, en cada parte o hay un gran porcentaje de elementos similares a q o la parte completa debería ser descartada a partir de la búsqueda en el índice. Explicamos a continuación el proceso de particionado que utiliza estas ideas.

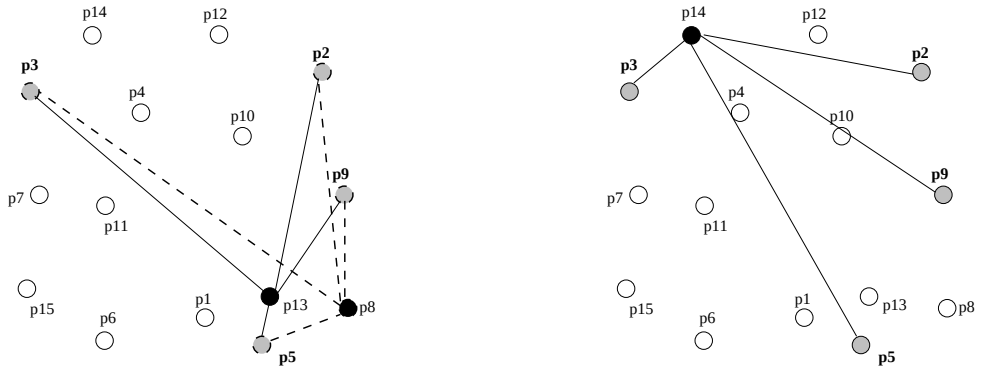
3.1. Particionado LCS

Este proceso de particionamiento usa la función de distancia *LCS (longest common subsequence)* para medir la similitud entre elementos. Esta función se aplica a cadenas y devuelve la longitud de la máxima subsecuencia común entre dos cadenas. Por ejemplo, si tenemos el alfabeto $\sum = \{a, b, c, d, e, f, g, h, i\}$, y dos cadenas $\alpha = aabcdeh$ y $\beta = abfgdih$, entonces $LCS(\alpha, \beta) = 4$, siendo la máxima subsecuencia común $abdh$.

En términos generales, el proceso de particionamiento consiste en identificar dos grupos: los *t_aceptables*, en donde quedarán todos los elementos del espacio que son similares según una tolerancia t, y los *no t_aceptables*. Como en el conjunto de los *t_aceptables* todos los elementos son parecidos, este conjunto se particiona aleatoriamente. Con el conjunto de los no *t_aceptables* se intenta aplicar el mismo proceso.

En forma detallada, el proceso es el siguiente:

1. Se eligen en forma aleatoria z elementos de la base de datos a los que llamaremos ***permutantes***.
2. Para cada elemento x de la base de datos (incluyendo a los permutantes), se crea un vector v_x de z componentes al que llamamos ***permutación***. En este vector se mantienen los permutantes ordenados según la distancia a x. Es decir, si v_{x_i} es elemento que se encuentra en la posición i de v_x, entonces $d(x, v_{x_i}) \leq d(x, v_{x_{i+1}})$ para $i = 1 \ldots z - 1$.
3. Se elige al azar una permutación como canónica a la que denotaremos con v_c.
4. Para cada elemento x, si $LCS(v_x, v_c) \geq t$ entonces x se agrega al conjunto de los t*_aceptables*; caso contrario integra el conjunto de los *no* t*_aceptables*
5. Con los elementos *t_aceptables* se arman al azar una cierta cantidad de partes. La cantidad de elementos en cada parte queda determinada por la cantidad de elementos que cubran el tamaño de una página de disco.



Permutación correspondiente a los puntos p13 y p8: *p5, p9, p2, p3*

Permutación correspondiente a los puntos p14: *p3, p2, p9, p5*

Figura 1: Permutaciones generadas por los permutantes p_5, p_9, p_2, p_3.

6. Los elementos del conjunto *t_aceptables* que no se ubicaron en ninguna parte junto con los *no t_aceptables* forman un nuevo grupo, sobre el que se repiten todos los pasos anteriores, teniendo cuidado de elegir como z permutantes un conjunto distinto a los ya usados.

El proceso de particionamiento detallado determina la similitud de dos elementos en función de la visión que estos elementos tienen de los permutantes. Es decir, dos elementos se consideran parecidos si ven a t o más permutantes en el mismo orden. La figura 1 ilustra estas ideas. En este ejemplo se han elegido 4 permutantes p_3, p_2, p_9 y p_5, obteniéndose las permutaciones para los puntos p_{13}, p_8 y p_{14}. Notar que tanto p_{13} como p_8 generan la misma permutación $v_{p_{14}} = v_{p_8} = (p_5, p_9, p_2, p_3)$, y en consecuencia $LCS(v_{p_{13}}, v_{p_5}) = 4$. Sin embargo, si miramos el punto p_{14}, la permutación generada es $v_{p_{14}} = (p_3, p_2, p_9, p_5)$, por lo tanto $LCS(v_{p_{13}}, v_{p_{14}}) = 1$, lo que implica que p_{13} no es similar a p_{14}.

Es posible que en algún momento nos quede un conjunto de elementos tan dispares que no sea posible clasificarlos con z permutantes, es decir, el conjunto *t_aceptables* queda vacío o con pocos elementos a partir de los cuales no es posible formar nuevas particiones. A este grupo de elementos, que llamaremos *outliers*, se lo particiona en forma totalmente aleatoria.

Notar que este algoritmo tiene tres parámetros: la tolerancia t, la cantidad de permutantes z y la cantidad de partes que se generan con cada conjunto *t_aceptables*.

4. Evaluación Experimental

La evaluación de la técnica de particionado se realizó usando como espacio métrico diccionarios de palabras con la función de distancia de edición. Esta función es discreta y calcula la mínima cantidad de caracteres que hay que agregar, cambiar y/o eliminar a una palabra para obtener otra.

Se usaron en total 4 diccionarios (Español, Inglés, Francés e Italiano). Para cada uno de ellos se eligieron al azar 500 palabras las que fueron utilizadas en todos los experimentos. Para cada palabra de este grupo, se realizaron búsquedas por rango usando como radio de búsqueda r los valores $1, 2, 3$ y 4.

Los experimentos se diseñaron teniendo como objetivo evaluar dos aspectos:

- Establecer los valores óptimos para los parámetros del algoritmo de partición basado en la distancia LCS.
- Analizar el desempeño de la técnica de partición comparándola con un particionado completamente al azar.

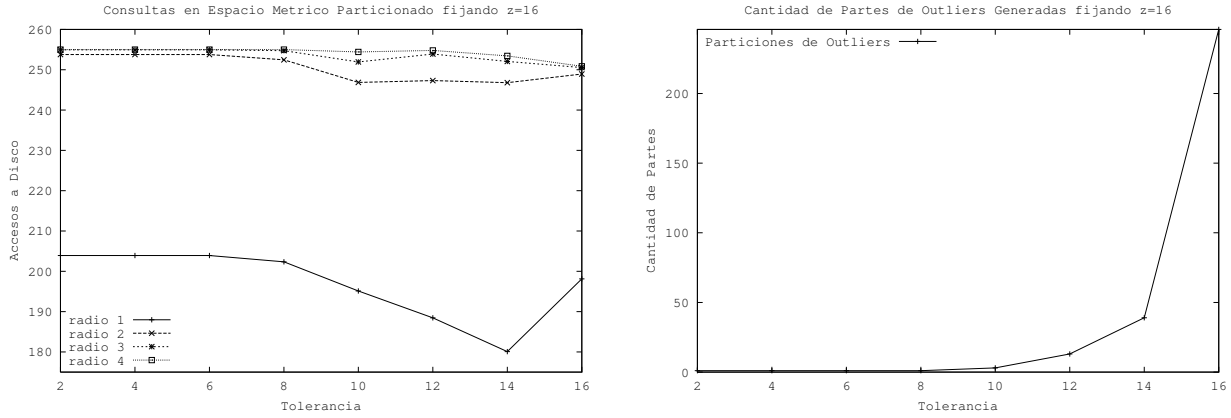


Figura 2: *Diccionario Español, sin limitar la cantidad de particiones generadas, variando t independiente de z*

Para indizar cada parte de usaron 10 pivotes elegidos aleatoriamente, con 4 bits por pivote.

4.1. Determinación de los valores óptimos para los parámetros

Recordemos que el algoritmo presentado tiene 3 parámetros: la cantidad de partes que se generan con cada conjunto de *t_aceptables*, la tolerancia t y la cantidad de permutantes z .

- Respecto de la cantidad de partes generadas con cada conjunto de *t_aceptables*, se probaron dos opciones: una es generar con cada conjunto la mayor cantidad de partes posibles, y la otra es generar sólo 2 partes a fin de permitir que los elementos se agrupen respecto de distintos puntos de referencia.

- Respecto de t y z en una primera etapa se hicieron experimentos que fijan el valor de z en 16 y varían el valor de t en $2, 4, 6, \ldots, 16$. Pero nuestra intuición nos decía que era más representativo elegir el valor de t cercano al valor de z; por ejemplo, no significa lo mismo una tolerancia de 3 cuando $z = 4$ que cuando $z = 9$; en el primer caso para que dos elementos sean similares estamos pidiendo que vean de la misma manera a 3 de 4 permutantes; en el segundo caso pedimos 3 de 9, lo que implicaría que dos elementos no parecidos pertenezcan a la misma parte. Por esta razón se realizó un segundo grupo de experimentos en los que varía t en función de z, esto es, $z = 4, 5, \ldots, 9$ y $t = z, z - 1$.

- Finalmente teníamos que establecer un límite en la cantidad de intentos para particionar antes de que un grupo de elementos fuera considerado conjunto de outliers. Para ello, se realizaron experimentos que permitieron determinar que 500 iteraciones era un límite seguro. Esto significa que si no logramos dividir un conjunto de elementos después de esa cantidad de intentos, los elementos son realmente dispares.

Comenzamos nuestros experimentos con un diccionario Español de $86{,}061$ palabras analizando la conveniencia o no de limitar la cantidad de partes generadas con cada conjunto de *t_aceptables*.

La figura 2 muestra los resultados obtenidos con este diccionario **cuando no se limitan la cantidad de partes generadas con cada conjunto *t_aceptables***, con t variando en forma independiente de z (para $z = 16$). En la figura de la izquierda se ha graficado la cantidad de accesos a disco en función de t para los distintos radios de búsqueda r. Se puede observar que el valor de t que produce resultados más competitivos varía dependiendo del radio de búsqueda considerado. Por ejemplo, para $r = 1$

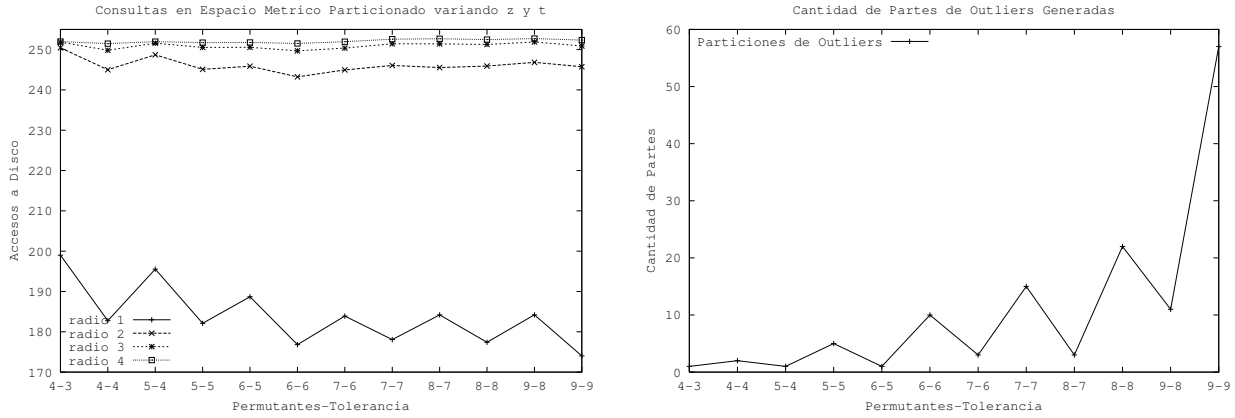


Figura 3: *Diccionario Español, sin limitar la cantidad de particiones generadas, variando t dependiendo de z*

los mejores resultados se obtienen con $t = 14$ y para $r = 4$ con $t = 16$. Sin embargo, si analizamos el comportamiento de $t = 14$ para todos los radios, vemos que o es el óptimo o está muy cercano al óptimo. Como las búsquedas se realizarán con distintos radios, en el momento de particionar no podemos realizar una elección que dependa de r. Por lo tanto elegimos $t = 14$ para la comparación con las demás variantes del particionado LCS.

Notar que mientras más lejano es t de z más cantidad de accesos a disco deben realizarse. Esto sucede debido a que para la mayoría de los elementos se cumple que su permutación está a distancia LCS mayor o igual que t de la permutación elegida como canónica, aún cuando estos elementos no son similares entre sí.

Algo similar sucede cuando t varía cercano a z, pero aquí el problema reside en que es poco factible que la permutación de un elemento esté a distancia LCS mayor o igual que t de la permutación canónica. Esto significa que los conjuntos *t_aceptables* contienen pocos elementos y generan pocas o ninguna parte, lo que provoca un conjunto de outliers demasiado grande. La afirmación anterior puede verificarse en la figura 2 (derecha), en la que se muestra la cantidad de partes que se generan en forma totalmente aleatorio a partir del conjunto de outliers, en función de t. Para el caso $t = 16$ se generan un 246 partes de outliers sobre un total de 251 partes, es decir un $98\,\%$ de partes provienen del conjunto de outliers.

La figura 3 muestra los resultados obtenidos cuando t varía en función de z. La figura de la izquierda muestra la cantidad de acceso a disco de cada combinación $z - t$, para los distintos radios de búsquedas. Nuevamente sucede que no existe una combinación que sea la más adecuada para todos los radios, por esta razón elegimos $z = 6$ y $t = 6$ para las futuras comparaciones dado que es la combinación más cercana al óptimo en todos los casos.

Hay dos hechos para destacar. Por una lado, si comparamos $t = z$ y $t = z - 1$ para cada valor de z, siempre sucede que los resultados más competitivos se logran en $t = z$. Nuestra intuición sobre este comportamiento es que los elementos agrupados con $t = z$ son realmente similares porque ven a todos los permutantes exactamente de la misma forma (recordemos el ejemplo presentado en la figura 1 de la sección 3.1).

Por otro lado, a medida que z crece se mejora la performance de la técnica. Esto sucede porque mientras más permutantes usemos más puntos de referencia existen para corroborar si dos elementos son realmente similares.

La afirmación anterior nos llevaría a seguir aumentando la cantidad de permutantes z. Pero si hacemos esto provocamos un aumento en la cardinalidad del conjunto de outliers. La figura 3 (derecha) muestra la cantidad de partes generadas a partir del conjunto de outliers, en donde se ve claramente

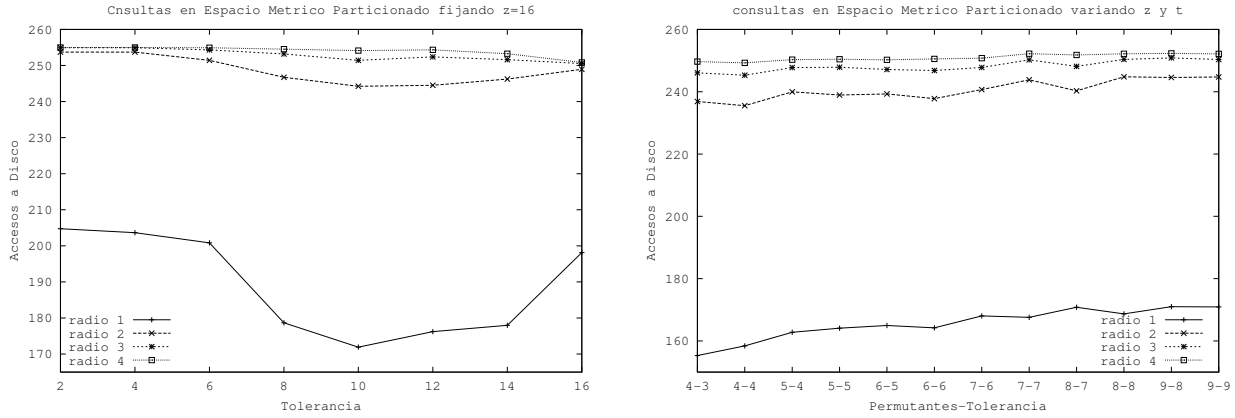


Figura 4: *Diccionario Español, limitando la cantidad de particiones generadas, variando t independiente de z (izquierda) y variando t en función de z (derecha)*

como influye z en este aspecto.

Realizamos ahora el análisis del comportamiento de particionado LCS **cuando limitamos la cantidad de partes que se generan con cada conjunto de *t_aceptables* en** 2. La figura 4 muestra la cantidad de accesos a disco que se realizan cuando t varía en forma independiente e z (izquierda) y cuando t varía dependiendo del valor de z (derecha). En el primer caso el valor más conveniente resulta ser $t = 10$, dado que es el más cercano al óptimo para todos los radios de búsquedas. En el segundo caso la combinación más conveniente es $z = t = 4$.

Notar que han disminuido los valores de z y t en los que se alcanza mejor desempeño, respecto del caso anterior (no limitar la cantidad de partes). Esto sucede porque si limitamos la cantidad de partes a generar con cada grupo de permutantes en 2, estamos forzando a que pocas partes dependan de los mismos puntos de referencia (los permutantes) evitando sesgar la clasificación con los primeros z permutantes. Este mismo comportamiento se lograba en el caso anterior aumentando la distancia LCS a su valor máximo ($t = z$), logrando así que cada conjunto de *t_aceptables* tenga pocos elementos y en consecuencia genere pocas partes.

Los experimentos hasta aquí presentados se repitieron con los diccionarios Inglés, Francés e Italiano, obteniéndose resultados similares.

4.1.1. Seleccionando la mejor variación

El análisis realizado en la sección anterior puede resumirse en los siguientes puntos:

1. Cuando no se limita la cantidad de partes generada con cada conjunto *t_aceptable* y se fija $z = 16$, la mejor opción es $t = 14$.

2. Cuando no se limita la cantidad de partes generada con cada conjunto *t_aceptable* y se varía t en función de z, la mejor combinación es $z = t = 6$.

3. Cuando se limita la cantidad de partes generada con cada conjunto *t_aceptable* a 2 y se fija $z = 16$, la mejor opción es $t = 10$.

4. Cuando se limita la cantidad de partes generada con cada conjunto *t_aceptable* a 2, y se varía t en función de z, la mejor combinación es $z = t = 4$.

La figura 5 (izquierda) muestra el comportamiento de estos cuatro puntos para el diccionario Español a medida aumenta el radio de búsqueda. A partir de ellas podemos concluir que la mejor

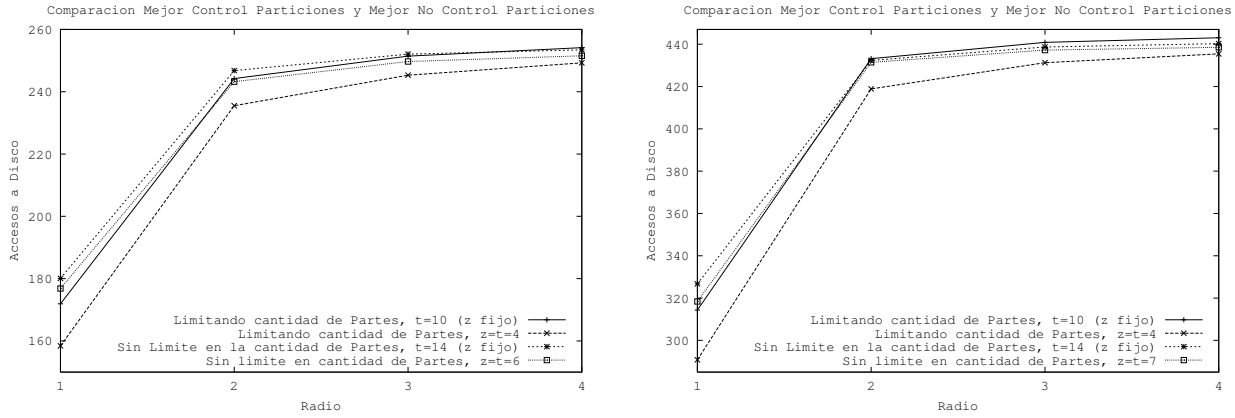


Figura 5: *Diccionario Español (izquierda), Diccionario Francés (derecha).*

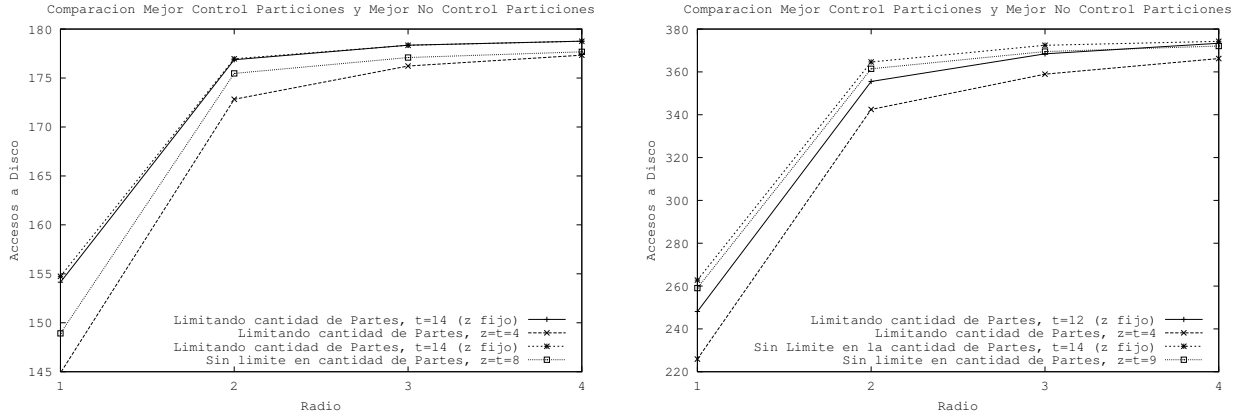


Figura 6: *Diccionario Inglés (izquierda), Diccionario Italiano (derecha).*

opción es *limitar la cantidad de partes generada con cada conjunto* t_aceptable *a* 2*, fijando* $z = 4$ *y* $t = 4$.

Con los demás diccionarios se repite este comportamiento. En las figuras 5 (derecha) y 6 se muestran las mismas gráficas para el resto de los diccionarios en las que se puede observar la similitud con la figura anterior; es decir,varían algunos valores de z y t respecto del diccionario Español en los puntos 1 a 3 pero la mejor opción resulta ser, en todos los casos presentados, la 4 donde $z = 4$ y $t = 4$.

4.2. Comparación con Particionado Aleatorio

Las figuras 7 y 8 muestran, para todos los diccionarios, la cantidad de accesos a disco realizados por el Particionado LCS (en su mejor variación) y la cantidad de acceso a disco realizados por un particionado totalmente aleatorio.

Puede observarse que, en todos los casos, la técnica de particionado LCS es superior al particionado aleatorio especialmente para búsquedas de alta selectividad, de modo que una parte importante del algoritmo está en la división adecuada del espacio.

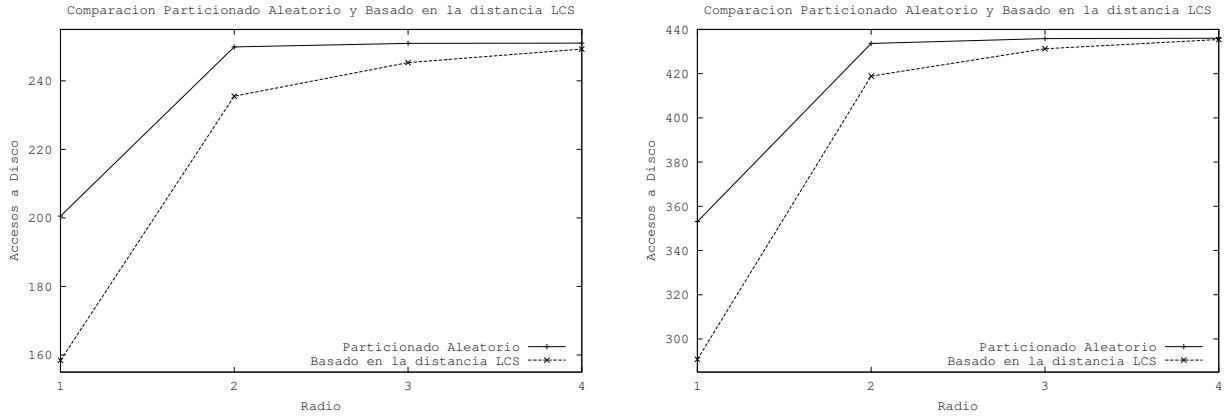


Figura 7: *Particionado LCS versus Particionado Aleatorio para los diccionarios Español (izquierda) y Francés (derecha).*

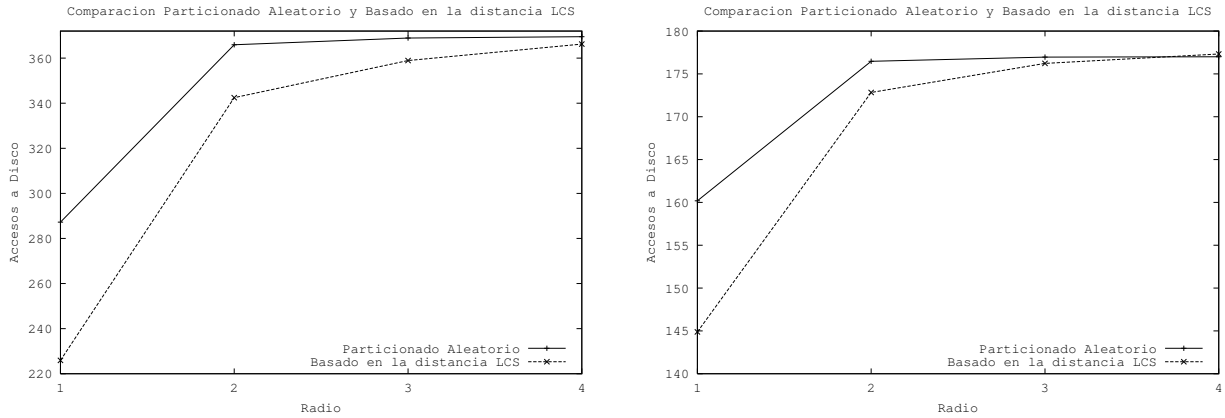


Figura 8: *Particionado LCS versus Particionado Aleatorio para los diccionarios Italiano (izquierda) e Inglés (derecha).*

5. Conclusiones y Trabajo Futuro

En este artículo hemos presentado una técnica que permite el manejo de espacio métricos cuyo índice completo y/o datos excedan la capacidad de memoria principal. Para ello, en lugar de modificar el *FQTrie* para que sea eficiente en memoria secundaria, particionamos el espacio de manera tal que cada una de las partes entre en memoria principal, las que posteriormente se indexan en forma separada. Luego, una búsqueda se resuelve buscando en cada parte, lo que puede ser hecho en memoria principal y en paralelo. Esto implicaría un número constante de visitas a disco (la cantidad de partes generadas más la cantidad de páginas que contengan índices). Sin embargo, al agrupar elementos similares en cada parte, se evitan visitas a ciertas páginas de disco, ya que buscar en el índice de una parte puede indicar que en esa parte no hay elementos relevantes a la búsqueda, y no es necesario cargarla, y es aquí donde se evita un acceso a disco.

La técnica de particionamiento diseñada se basa en la distancia LCS, lo que permite que el espacio se divida de manera tal que en cada parte queden elementos similares. Esta técnica detecta grupos de elementos parecidos, denominados *t_aceptables*, a partir de los cuales genera la partición.

Se estudió el comportamiento de esta técnica sobre diccionarios de palabras con la función de distancia de edición. Del análisis de resultados podemos concluir que, para obtener un buen desempeño de las consultas sobre el espacio particionado, debe limitarse la cantidad de partes generadas a partir de cada conjunto *t_aceptable* en 2, fijándose la cantidad de permutantes z en 4 y el radio de tolerancia t en 4.

La técnica de particionamiento basada en la distancia LCS ha demostrado ser competitiva, logrando disminuir la cantidad de accesos a disco en un $21\,\%$ respecto de un particionado totalmente aleatorio.

Con respecto al trabajo futuro nos proponemos estudiar el comportamiento de esta técnica sobre otros tipos de espacios métricos. Estamos trabajando además en integrar este trabajo con una versión eficiente del FQTrie en memoria principal, a fin de dejar disponible nuestra implementación para usuario final.

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] E. Chávez and K. Figueroa. Faster proximity searching in metric data. In *Proceedings of MICAI 2004. LNCS 2972, Springer*, Cd. de México, México, 2004.

[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] 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.