Indexación espacial de puntos empleando wavelet trees*

Nieves R. Brisaboa[1], Miguel R. Luaces[1], Gonzalo Navarro[2], and Diego Seco[1]

[1] Laboratorio de Bases de Datos, Universidade da Coruña
Campus de Elviña, 15071, A Coruña, España
{brisaboa,luaces,dseco}@udc.es
[2] Departamento de Ciencias de la Computación, Universidad de Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Resumen El desarrollo de estructuras de indexación que permitan recuperar objetos espaciales de manera eficiente ha sido un tema de interés en las últimas décadas. La mayoría de estas estructuras han sido diseñadas pensando en las características específicas de la memoria secundaria. Sin embargo, en los últimos años el precio de la memoria principal se ha reducido considerablemente y, por tanto, hoy en día es posible almacenar índices espaciales completos sin necesidad de acceder a disco.

En este trabajo presentamos una estructura para la indexación de puntos diseñada para memoria principal que mantiene un buena relación entre el espacio necesario para almacenar el índice y la eficiencia de las búsquedas. Nuestra estructura se basa en el *wavelet tree*, un árbol diseñado originalmente para indexar los caracteres de un texto, pero que recientemente se ha empleado con éxito para la construcción de auto-índices en áreas tan diferentes como la recuperación de información o la compresión de imágenes.

Key words: Índice espacial, métodos de acceso a puntos, wavelet tree

1. Introducción

La popularidad de los sistemas de información geográfica (SIG) [1] ha aumentado considerablemente en los últimos años. Una de las principales causas de este aumento de popularidad son las mejoras recientes en el hardware que han permitido que el desarrollo de este tipo de sistemas sea abordable por muchas organizaciones. Además, esta tecnología se emplea cada vez más en un mayor número de áreas distintas, como la generación de cartografía, la planificación urbanística, el márketing o la arqueología.

* Este trabajo ha sido financiado parcialmente por el Ministerio de Educación y Ciencia (PGE y FEDER) ref. TIN2006-15071-C03-03 y por la Xunta de Galicia ref. 2006/4 y ref. 08SIN009CT.

Una de las características más destacables de los sistemas de información geográfica es su capacidad para gestionar enormes cantidades de información. Esto hace que uno de los temas de investigación más importantes en el área sea el diseño de estructuras de indexación que permitan un acceso eficiente a la información. A lo largo de los años, se han propuesto muchas estructuras diferentes para lograr este objetivo. Dichas estructuras se pueden clasificar de manera general en métodos de acceso a puntos (PAMs, del inglés *Point Access Methods*) y métodos de acceso espacial (SAMs, del inglés *spatial access methods*). Los métodos del primer grupo se caracterizan por mejorar el acceso a colecciones de datos formadas por puntos espaciales. En cambio, los métodos del segundo grupo son más generales ya que trabajan con colecciones de todo tipo de objetos geográficos (puntos, líneas, polígonos, etc.).

La mayoría de los métodos propuestos, tanto en una categoría como en la otra, están orientados a su empleo en memoria secundaria. Esto se debe fundamentalmente a razones históricas. Hace años, las memorias eran pequeñas y caras, y esto convertía en inviable el pensar en el diseño de estructuras de indexación espacial que se pudiesen emplear en memoria principal. Sin embargo, en los últimos años, el precio de las memorias se ha reducido considerablemente, aumentando su tamaño también de manera notable. Por tanto, hoy en día es posible diseñar índices espaciales completos que puedan operar en memoria principal. Para ello, a la hora de diseñar nuevas estructuras de indexación no se debe pensar en optimizar tan sólo la eficiencia de las consultas sino también el espacio necesario para el almacenamiento de la estructura.

En este artículo presentamos un nuevo método de acceso a puntos que almacena tanto el índice como la colección de puntos en una estructura compacta. Esta estructura presenta una buena relación entre el espacio necesario para su almacenamiento y la eficiencia de las búsquedas, lo cual la hace adecuada para su empleo en memoria principal.

En los últimos años, la idea de almacenar de forma conjunta los datos y la estructura de indexación que permite un acceso eficiente a los mismos se ha empleado con éxito en varios campos de investigación. Estas estructuras que permiten almacenar los datos y el propio índice de manera conjunta se denominan auto-índices. Por ejemplo, en [2], se presenta una aproximación para indexar documentos empleando *wavelet trees*. Originalmente, el *wavelet tree* [3] se diseñó como un auto-índice organizado en forma de árbol binario para indexar los caracteres de un texto. En nuestro trabajo, analizamos esta estructura y la adaptamos a las características especiales de la información geográfica.

El resto de este artículo está organizado de la siguiente manera. En primer lugar, revisamos el trabajo relacionado en la sección 2. Luego, en la sección 3 describimos nuestra estructura de indexación. A continuación, en la sección 4, presentamos algunos experimentos que hemos realizado para comparar el rendimiento de nuestra estructura frente a otros métodos de acceso a puntos y métodos de acceso espacial en general. Finalmente, cerramos el artículo con las conclusiones y líneas de trabajo futuro en la sección 5.

2. Trabajo relacionado

Durante los últimos años se han propuesto muchos métodos de acceso espacial y muchos métodos de acceso a puntos diferentes. En [4] y [5] se puede encontrar una buena descripción de las estructuras más relevantes en cada categoría. El objetivo de los métodos en ambos grupos es mejorar la eficiencia a la hora de recuperar subconjuntos de datos que se ajustan a una consulta de búsqueda determinada. Una de las consultas de búsqueda más comunes que deben resolver los métodos de ambos grupos es la consulta de tipo región. Esta consulta define una región en el espacio y obtiene como resultado todos los objetos geográficos indexados que se solapan con dicha región. En la sección 4, comparamos la eficiencia de nuestra estructura frente a la de métodos representativos de las dos categorías de estructuras de indexación espacial a la hora de resolver este tipo de consultas.

El R-tree [6] es uno de los métodos de la categoría SAM más populares y se puede considerar un ejemplo paradigmático. Esta estructura se basa en un árbol balanceado derivado del B-tree que divide el espacio en rectángulos de cobertura mínima (MBRs, del inglés *Minimum Bounding Rectangles*) agrupados jerárquicamente y que pueden solaparse entre ellos o no. El número de nodos hijos de cada nodo interno varía entre un mínimo y un máximo. El árbol se mantiene balanceado dividiendo aquellos nodos que tienen un número de descendientes por encima del umbral de carga máxima y combinando aquellos otros que tienen un número de descendientes por debajo del umbral de carga mínima. Cada nodo hoja tiene asociado un MBR que delimita el área del espacio que cubre ese nodo. Además, los nodos internos también almacenan un MBR que delimita el área que cubren todos sus descendientes. La descomposición del espacio que proporciona el R-tree es adaptativa (es decir, dependiente de la distribución de los objetos geográficos indexados) y puede presentar solapes (es decir, los nodos del árbol pueden representar regiones no disjuntas). Sobre la propuesta original de A. Guttman [6] se han propuesto muchas variantes para mejorar su eficiencia (como el R+-tree o el R*-tree) y para adaptarlas a distintos problemas (como el STR R-tree para colecciones estáticas). En [7] se presenta un resumen de todas estas estructuras y de sus aplicaciones.

Por otra parte, el K-d-tree [8] es uno de los métodos de la categoría PAM que más se ha empleado, debido fundamentalmente a su sencillez y eficiencia. Cuando esta estructura se emplea para indexar colecciones de puntos se denomina más comúnmente Point k-d-tree. En general, el K-d-tree y los métodos derivados de él se basan en árboles de búsqueda binarios que representan una subdivisión recursiva del dominio basada en el valor de un único atributo en cada nivel del árbol. Las numerosas variantes de esta estructura se suelen identificar basándose en cómo particionan el espacio. En nuestros experimentos, empleamos una aproximación estática propuesta en [8] que asume que se conoce de antemano la colección completa de puntos a indexar. En esta variante, las líneas de partición deben pasar obligatoriamente por los puntos del conjunto de datos y los ejes deben alternarse cíclicamente en un orden constante.

Ambas estructuras están optimizadas para minimizar el tiempo necesario para resolver distintos tipos de consultas, fundamentalmente consultas de tipo región. Sin embargo, estas estructuras ignoran completamente el espacio necesario para su almacenamiento. En nuestra opinión, una buena relación entre el espacio necesario para almacenar la estructura y su eficiencia a la hora de realizar búsquedas es más interesante que únicamente la optimización de la eficiencia de las búsquedas. Para realizar esta afirmación nos basamos en que el tiempo de acceso a disco es varios ordenes de magnitud superior al tiempo de acceso a memoria principal. Por tanto, si conseguimos reducir el tamaño de la estructura de indexación, e incluso de los datos, lo suficiente como para que pueda operar en memoria principal con conjuntos reales de datos la eficiencia de nuestra estructura será mucho mejor que la de cualquier alternativa que tenga que trabajar en disco.

Como ya hemos mencionado, el *wavelet tree* [3] es una estructura compacta diseñada originalmente para indexar los caracteres de un texto. Esta estructura se ha empleado recientemente en otras áreas para almacenar e indexar distintos tipos de información de forma compacta. Por ejemplo, en [2] se emplea para indexar y recuperar documentos, mientras que en [9] se emplea para la indexación de imágenes. La herramienta básica empleada en el *wavelet tree* es la operación *rank* sobre vectores de bits: la consulta $rank(B, i) = rank_1(B, i)$ devuelve el número de bits establecidos a uno en el prefijo $B[1, i]$ de un vector de bits $B[1, n]$. De manera simétrica, $rank_0(B, i) = i - rank_1(B, i)$. La operación dual a $rank_1$ es $select_1(B, j)$, que permite obtener la posición del j-ésimo bit establecido a uno en B. La definición de $select_0$ es análoga. Por ejemplo, dado un bitmap $B =$ 1000110, $rank_1(B, 5) = 2$ y $select_0(B, 4) = 7$. Tanto la operación de *rank* como la de *select* se han estudiado mucho en los últimos años y existen implementaciones que permiten su ejecución en tiempo constante y con un consumo de espacio bastante pequeño. Además, la implementación de estas operaciones de forma eficiente continúa siendo un tema de interés hoy en día [10,11].

3. Descripción de la estructura

3.1. Construcción de la estructura

Podemos formalizar el problema que tratamos de abordar de la siguiente manera. Partimos de un conjunto de N puntos, $P = P_1 \dots P_n$, cada uno definido por dos coordenadas (por ejemplo, latitud y longitud) que identifican su posición en el espacio con respecto a un sistema de referencia. Podemos asumir que estos puntos están distribuidos en una matriz de $N \times N$ donde sólo hay un punto en cada fila y columna. Esto no supone una restricción muy importante ya que podemos ordenar los puntos en cada dimensión permitiendo duplicados. En la figura 1 mostramos el proceso de creación de la matriz representativa de los puntos de entrada. Es importante observar que lo único que mantiene la matriz representativa es el orden, no hay distancias ni proporciones. Esto es lo que nos permite crear una matriz que represente cualquier conjunto de puntos aunque algunos tengan coordenadas repetidas.

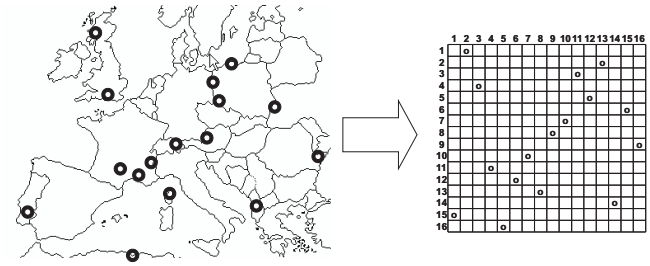

Figura 1. Creación de la matriz representativa de los datos

El *wavelet tree* es una estructura compacta que nos permite almacenar en poco espacio la matriz que acabamos de construir y con unas propiedades interesantes para realizar búsquedas en ella. Considerando una matriz de $N \times N$, podemos construir un *wavelet tree* con $log_2(N)$ niveles y N bits por nivel. Este *wavelet tree* permite almacenar la permutación del orden de los puntos en una dimensión (por ejemplo, longitud) al orden de los puntos en la otra dimensión (por ejemplo, latitud). Sean $X = P_{X1} \dots P_{Xn}$ e $Y = P_{Y1} \dots P_{Yn}$ dichas permutaciones donde los puntos están ordenados por sus longitudes y latitudes respectivamente. Por ejemplo, en la figura 1 podemos nombrar los puntos de izquierda a derecha (es decir, P_i es el i-ésimo punto empezando a contar por la izquierda). Por tanto, la permutación de las longitudes la podemos escribir como $X = P_1 \dots P_n$. En ese caso, la otra permutación la escribiríamos como $Y = P_2 P_{13} P_{11} \dots P_1 P_5$. Estas permutaciones nos indican, por ejemplo, que el punto P_1 es el primero en el orden de las longitudes y el penúltimo en el orden de las latitudes (es decir, el situado más a la izquierda y el penúltimo más abajo).

La raíz del *wavelet tree* es un bitmap $B = b_1 \dots b_n$ de la misma longitud que el conjunto de puntos (es decir, de N posiciones). Cada posición i del bitmap representa al punto en la i-ésima posición de la primera permutación (es decir, longitud). Siguiendo con el ejemplo, la posición 1 representa el punto $P_{X1} = P_1$, la 2 representa el punto $P_{X2} = P_2$, etc. El valor que se almacena en cada posición puede ser $b_i = 0$ si $P_{Xi} \in P_{Y1} \dots P_{Yn/2}$ o $b_i = 1$ si $P_{Xi} \in P_{Yn/2+1} \dots P_{Yn}$. Es decir, si el punto se encuentra en la primera mitad de la segunda permutación se almacena un 0 y en caso contrario se almacena un 1. La secuencia de puntos marcados con un 1 en el vector se procesan en el hijo derecho del nodo, mientras que aquellos marcados con 0 se procesan en el hijo izquierdo del nodo. En esta estructura, cada nodo indexa la mitad de símbolos que su nodo padre. Este proceso se repite recursivamente en cada nodo hasta que se alcanzan los nodos hoja donde la secuencia de símbolos indexados se corresponde con la permutación en la segunda dimensión (es decir, latitud). En la figura 2 mostramos el *wavelet tree* construido con el ejemplo de la figura 1. Para aportar claridad al ejemplo mostramos para cada elemento del bitmap a qué posición de la segunda permutación se corresponde (estos valores los tachamos para indicar que no se almacenan en la estructura).

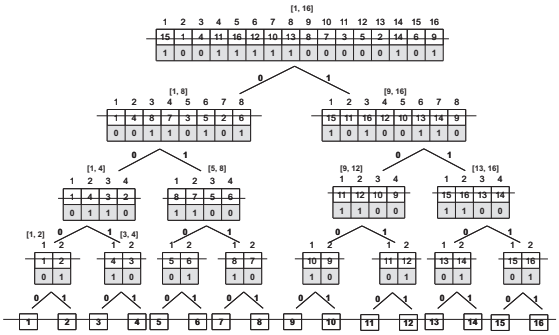

Figura 2. Creación del wavelet tree

3.2. Resolución de consultas

La estructura que acabamos de describir nos permite obtener de forma sencilla en qué posición de la segunda dimensión se encuentra un punto del cual se sabe su posición en la primera dimensión, simplemente descendiendo en el árbol. Para acceder desde una posición determinada de un nodo del árbol al siguiente nivel se emplea la operación *rank* y el valor almacenado en esa posición. El bit B_i del bitmap de un nodo indica si el punto correspondiente se indexa en el hijo izquierdo ($B_i = 0$) o en el hijo derecho ($B_i = 1$). Además, $rank_{Bi}(B, i)$ nos indica la posición en la que se almacena dicho punto en el nodo hijo. Este proceso se repite hasta que se alcanza un nodo hoja donde la posición señala el orden en la otra dimensión. Siguiendo con el ejemplo, para conocer en qué fila se encuentra el punto de la columna 6 se mira el valor que hay en la posición 6 del nodo raíz y se accede a la posición $rank_1(B, 6) = 4$ del nodo derecho (ya que en la posición 6 hay un 1). Este proceso se repite en todos los niveles del árbol hasta alcanzar el nodo hoja donde se obtiene la posición 12 que, como se puede comprobar en la matriz de la figura 1, es la fila correspondiente a la columna 6.

Del mismo modo, podemos conocer en qué posición de la primera permutación se encuentra un punto del cual se sabe su posición en la otra dimensión simplemente ascendiendo en el árbol. Para acceder al nivel anterior en el árbol desde un nodo se emplea la operación *select* y el valor que etiqueta la rama del árbol (es decir, el valor que da acceso a ese nodo). Este segundo valor no lo tenemos que almacenar ya que, al ser un árbol binario, es muy fácil de obtener a partir de la posición. En el ejemplo, si queremos conocer en qué columna se encuentra el punto de la fila 13 se mira con qué valor se accedió al nodo que contiene la posición 13 en el último nivel del árbol. Como el último nivel del árbol no se almacena, en dicho nodo se encuentran las posiciones 13 y 14. Además, podemos determinar cuál de las dos posiciones de ese nodo se corresponde con la 13 empleando los valores almacenados en ellas de tal modo que si están en orden en la primera habrá un 0 y en la segunda un 1, y al revés en caso contrario. Por tanto, como sabemos que se accedió a ese nodo con un 0 y que la posición 13 es la

primera del nodo (ya que están en orden) se accede a la posición $select_0(B, 1) = 3$ del nodo padre. En el siguiente nivel se calcula $select_1(B, 3) = 6$ (1 por ser un hijo derecho y 3 por ser el valor calculado en el paso anterior) y así hasta el último nivel donde se obtiene la posición $select_1(B, 6) = 8$ que nos dice que el punto se encuentra en la columna 8.

Sin embargo, para poder resolver consultas espaciales de tipo región empleando este índice necesitamos tres estructuras auxiliares: dos vectores con las coordenadas ordenadas en cada dimensión y los identificadores de los puntos ordenados en el mismo orden que uno de esos dos vectores. Esto resulta evidente ya que el *wavelet tree* sólo almacena la relación entre las permutaciones, sin tener para nada en cuenta las relaciones en el mundo real (por ejemplo, las coordenadas de los puntos o las distancias entre ellos). Los vectores con las coordenadas se emplean para traducir la consulta espacial a un rango que indica las columnas y las filas donde se encuentran los puntos que pueden formar parte de la solución de la consulta. Una vez realizada la traducción de la consulta, el rango de columnas (longitudes) indica el rango de posiciones válidas en el nodo raíz del *wavelet tree*. El descenso en el árbol se realiza tal y como acabamos de explicar para una posición aunque se optimiza considerablemente teniendo en cuenta que los puntos consecutivos se mantienen consecutivos en los nodos hijo. Por tanto, sólo se necesitan dos operaciones de *rank* (una para la primera posición del rango y otra para la última). Además, el rango de filas (latitudes) se puede emplear para podar el descenso en el árbol. Cada nodo en el *wavelet tree* contiene puntos que cubren un rango determinado de filas. Si ese rango no interseca con el rango de filas que indica la consulta, el algoritmo no tiene que continuar por esa rama.

En la figura 3, mostramos el *wavelet tree* del ejemplo que hemos empleado hasta ahora con los vectores de coordenadas ordenados y los identificadores. En cuanto a estos últimos, sólo necesitamos uno de los dos vectores IDs(X) e IDs(Y) pero la elección de uno u otro condiciona el algoritmo de resolución de consultas y tiene sus ventajas e inconvenientes como veremos a continuación. La figura representa la resolución de un ejemplo de consulta de tipo región $q = \{(27'53, 15'75), (30'71, 19)\}$. Esta consulta se traduce al rango de columnas de interés [6, 10] y al rango de filas de interés [9, 14]. El algoritmo de resolución de consultas comienza con el descenso en el árbol. Como ya hemos mencionado, sólo es necesario calcular los *ranks* del principio y del final de las posiciones consecutivas. De este modo, el proceso se iniciaría calculando $rank_0(B, 6) = 3$, $rank_0(B, 10) = 4$, $rank_1(B, 6) = 4$ y $rank_1(B, 10) = 6$ (en realidad sólo dos de ellos ya que $rank_0$ y $rank_1$ de la misma posición son dependientes y uno se puede derivar del otro). Por tanto, en el segundo nivel los rangos de interés son el [3, 4] en el hijo izquierdo y el [4, 6] en el derecho. Sin embargo, el hijo izquierdo sabemos que no puede contener soluciones ya que cubre el rango de filas [1, 8] que no interseca con las filas de interés para la consulta [9, 14]. Este proceso se repite en todo el descenso hasta alcanzar los nodos hoja.

Una vez que se alcanzan los nodos hoja el algoritmo difiere según el orden en el que se almacenen los identificadores. Si se almacenan en el mismo orden que los nodos hoja (es decir, en el orden de las latitudes), al alcanzar el último

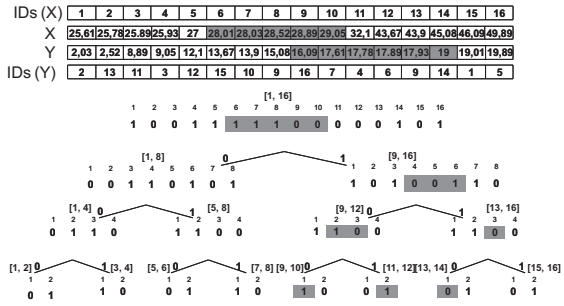

Figura 3. Resolución de consultas empleando el wavelet tree

nivel del árbol ya sabemos los identificadores que forman parte del resultado de la consulta. Esta versión de la estructura es más sencilla algorítmicamente y como veremos en la sección 4 también es más eficiente, a pesar de que tiene la desventaja de tener que alcanzar siempre los nodos hoja para obtener el resultado. Por el contrario, si la estructura almacena los identificadores en el mismo orden que el nodo raíz (es decir, en el orden de las longitudes) el algoritmo es más complicado ya que una vez se comprueba que es una latitud válida debe volver a ascender para obtener el identificador. Esta implementación tiene la ventaja de que dicha comprobación se puede cumplir en niveles altos del árbol y, por tanto, se puede detener el descenso y comenzar el ascenso. Sin embargo, los experimentos realizados demuestran que la eficiencia de esta versión es inferior.

4. Experimentos

4.1. Descripción de los experimentos

En esta sección presentamos los experimentos que realizamos para comparar la eficiencia de nuestra estructura frente a otros índices espaciales. Esta comparación tiene dos partes, en primer lugar comparamos los requisitos de espacio de las estructuras y, en segundo lugar, su eficiencia a la hora de resolver consultas espaciales. Como veremos, estos resultados demuestran que nuestra estructura presenta una muy buena relación entre el espacio necesario para almacenarla y la eficiencia a la hora de resolver las consultas.

En los resultados que vamos a presentar a continuación comparamos cuatro estructuras espaciales que trabajan en memoria principal. Las dos primeras se corresponden con las variantes de nuestra estructura de indexación que presentamos en la sección 3. En la primera de ellas, denominada PEW-tree (*efficient point wavelet tree)*, los identificadores de los puntos se almacenan siguiendo la permutación que representan los nodos hoja y, por tanto, sólo es necesario descender en el árbol para obtener los identificadores de los puntos. En la segunda, denominada PCW-tree (*classical point wavelet tree*), los identificadores se almacenan siguiendo la permutación representada en el nodo raíz y, por tanto, una

vez que se comprueba en el descenso que un punto pertenece al resultado hay que volver a ascender en el árbol para obtener su identificador. La tercera estructura es el R-tree clásico adaptado para trabajar en memoria principal. Aunque esta estructura se encuentra en la categoría de métodos de acceso espacial en general y no está optimizada para la indexación de puntos, es la más empleada en los sistemas de información geográfica que se desarrollan hoy en día y, por tanto, es importante ver cuánto se podrían beneficiar dichos sistemas empleando nuestra estructura. Por último, la cuarta estructura es un K-d-tree que representa los métodos de acceso a puntos. La variante de K-d-tree que hemos seleccionado es posiblemente la más eficiente ya que está optimizada para situaciones en las que se conoce *a priori* el conjunto de puntos a indexar.

Las colecciones de prueba sobre las que vamos a construir los índices espaciales son colecciones sintéticas con los puntos uniformemente distribuidos en el espacio. Experimentamos con colecciones de puntos con tamaños de 2^{19}, 2^{20}, 2^{23} y 2^{24}.

Para la realización de los experimentos, además de las estructuras de indexación y de las colecciones de puntos, necesitamos un conjunto de ventanas de consulta (es decir, de regiones donde buscar puntos que se encuentren en ellas). Para construirlas, implementamos un algoritmo de generación de ventanas de consulta espaciales de tamaño parametrizable. Dicho algoritmo está basado en los experimentos diseñados para evaluar el rendimiento del R*-tree tal y como presentaron sus autores en [12]. Las ventanas que generamos están uniformemente distribuidas en el espacio. Además, su proporción se determina de tal forma que el ratio entre la *extensión en x* y la *extensión en y* varía de manera uniforme entre 0,25 y 2,25.

4.2. Comparación de espacio

A la hora de diseñar estructuras de indexación espacial, el espacio necesario para almacenarlas ha sido un factor muy poco valorado ya que la mayoría de estas estructuras estaban pensadas para trabajar en disco. Recientemente, con el crecimiento de las memorias principales algunas de las estructuras clásicas se adaptaron para trabajar en memoria principal pero sin preocuparse demasiado del tamaño que ocupan. Por este motivo, creemos que el desarrollo de estructuras que presenten una buena relación entre el espacio que ocupan y la eficiencia a la hora de resolver consultas es un tópico de especial interés hoy en día.

La estructura que proponemos, en sus dos variantes, necesita almacenar las coordenadas de los N puntos (dos vectores de N elementos que son números en punto flotante), los identificadores (un vector de N elementos que son números enteros) y el *wavelet tree*. El *wavelet tree* es una estructura muy compacta que sólo necesita $N \times log_2(N)$ bits (se necesitan N bits por nivel, un bit por punto y hay $log_2(N)$ niveles). Además, para poder realizar las operaciones de *rank* y *select* en tiempo constante se necesitan unas estructuras auxiliares que ocupan un 37,5 % de espacio adicional sobre el tamaño del *wavelet tree*. Es decir, la estructura completa ocupa $20 \times N + (N \times log_2(N) \times 1,375)/8$ bytes.

En cuanto al espacio necesario para la construcción de un R-tree sobre una colección de N puntos, lo podemos estimar suponiendo un factor de ramificación (M) dado. Asumiremos un factor de ramificación de 30 entradas por nodo ya que, en nuestros experimentos, es el que presenta mejores resultados en términos de eficiencia. Además, dado que nuestra estructura es estática, supondremos que los nodos están completamente llenos (en realidad se suele estimar que de media los nodos están al 70 % de su capacidad). En este caso el R-tree consta de $N/M \times (M-1)$ nodos internos y N/M nodos hoja, donde cada nodo tiene un tamaño de $36 \times M$ bytes (los cuatro números en punto flotante que ocupa un MBR y un puntero). Además, hay que sumar el tamaño que ocupa en memoria la tabla de puntos ($20 \times N$ bytes).

Finalmente, un K-d-tree para indexar N puntos tiene una altura $h = \lceil log_2(N) \rceil$ y $2^h - 1 + (N \,\%\, 2^{\lfloor log_2(N) \rfloor})$ nodos, donde cada nodo tiene un tamaño de 16 bytes (un número en punto flotante y dos punteros). Al igual que en el caso del R-tree, hay que tener en cuenta los $20 \times N$ bytes de la tabla de puntos.

En la figura 4 mostramos una gráfica comparativa del espacio requerido por las diferentes estructuras de indexación espacial. Como nuestras dos alternativas ocupan exactamente el mismo espacio mostramos una única curva bajo el nombre de PW-tree. La principal conclusión a la vista de los resultados es que nuestra estructura necesita mucho menos espacio que cualquiera de las alternativas de índices espaciales clásicos por lo que es más adecuada para trabajar en memoria principal.

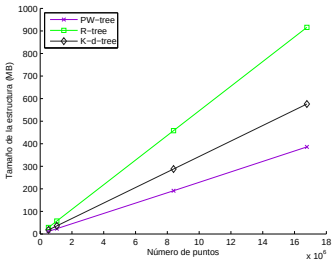

Figura 4. Comparación del espacio requerido por los índices espaciales

4.3. Comparación de tiempo

Para realizar la comparación de tiempos tenemos en cuenta dos parámetros que pueden influir en las pruebas: la selectividad de las consultas y el tamaño de las colecciones. La selectividad de las consultas viene indicado por el tamaño de las ventanas empleadas. En nuestros experimentos creamos ventanas de cuatro tamaños diferentes que representan el 0,01 %, el 0,1 %, el 1 % y el 10 % del área

total del espacio donde se encuentran representados los puntos. En la figura 5, mostramos varias gráficas donde se puede observar la influencia de estos factores en el tiempo necesario para resolver las consultas. Los tiempos se midieron en una máquina Intel(R) Pentium(R) 4 3.00GHz con 4GB DDR-400 RAM y sistema operativo Debian GNU/Linux (versión 2.4.27 del kernel).

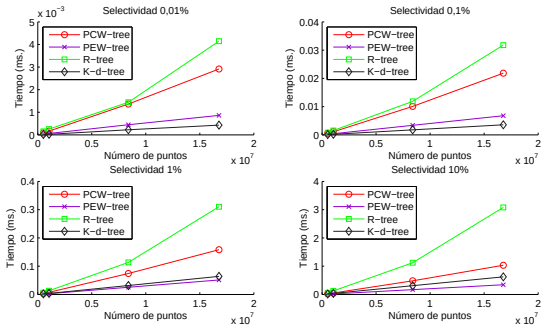

Figura 5. Comparación de tiempos

La conclusión más importante que se puede extraer a la vista de los resultados es que nuestra estructura es competitiva en cuanto a la eficiencia de resolución de consultas, llegando incluso a ganarle al K-d-tree en consultas poco selectivas. El K-d-tree se puede asegurar que es la estructura más eficiente pero, como veíamos en el apartado anterior, esa eficiencia se basa en un consumo de espacio mucho mayor. Como esperábamos, el R-tree no es tan competitivo ya que es un método mucho más general. Lo incluimos en estos resultados porque es el ejemplo paradigmático de estructura de indexación espacial y, además, se debe tener muy en cuenta ya que sigue siendo la estructura más empleada hoy en día. En cuanto a las dos versiones de nuestra estructura, la PEW-tree (la versión que sólo desciende en la estructura) es más eficiente que la PCW-tree (la versión que necesita ascender en el árbol).

5. Conclusiones y trabajo futuro

En este artículo hemos presentado un nuevo índice espacial en la categoría de los métodos de acceso a puntos. Este índice espacial está basado en el *wavelet tree*, una estructura compacta ampliamente utilizada en áreas como la recuperación de información para la creación de auto-índices. La principal ventaja de esta estructura frente a otros índices espaciales es que presenta una buena relación entre el espacio necesario para almacenarla y la eficiencia de las búsquedas. El poco espacio que necesita le permite operar en memoria principal aun con colecciones realmente grandes de puntos.

Actualmente, estamos trabajando en varias líneas que nos ha abierto esta investigación. En primer lugar, estamos desarrollando una nueva estructura basada en *wavelet trees* para la indexación de cualquier tipo de objeto geográfico empleando sus MBRs. Este índice, más general que el que presentamos en este trabajo, se enmarcaría en la categoría de métodos de acceso espacial. Además, estamos trabajando en la mejora de la estructura de indexación de puntos para permitir la inserción de manera dinámica de nuevos puntos una vez construida la estructura. Aunque consideramos que el ser una estructura estática no es un problema muy grave debido a la sencillez de la estructura y la consecuente rapidez de su construcción. También creemos que es posible mejorar la eficiencia a la hora de resolver consultas y acercarnos un poco más al K-d-tree ya que nuestra estructura se comporta bien en general y tiene un peor caso que se podría mejorar empleando búsquedas binarias. Otra línea de trabajo se centra en la implementación de algoritmos que nos permitan resolver otros tipos de consultas habituales como son las consultas del vecino más próximo (o los k vecinos más próximos) y el *join* espacial. Finalmente, estamos integrando la estructura en sistemas de información geográfica reales para comprobar como mejora la eficiencia de los mismos ya que, aunque se han propuesto muchas estructuras a lo largo de los años, la mayoría de los SIG reales continúan empleando el R-tree para la indexación de cualquier colección de objetos geográficos.

Referencias

1. Worboys, M.F.: GIS: A Computing Perspective. CRC (2004)
2. Brisaboa, N.R., Cillero, Y., Fariña, A., Ladra, S., Pedreira, O.: A new approach for document indexing usingwavelet trees. In: Proc. of DEXA'07. (2007) 69–73
3. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. of ACM-SIAM SODA'03. (2003) 841–850
4. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2) (1998) 170–231
5. Samet, H.: Multidimensional and Metric Data Structures. M. Kaufmann (2006)
6. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: Proc. of SIGMOD'84, ACM Press (1984) 47–57
7. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-Trees: Theory and Applications. Springer-Verlag New York, Inc. (2005)
8. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9) (1975) 509–517
9. Mäkinen, V., Navarro, G.: On self-indexing images - image compression with added value. In: Proc. of DCC'08, IEEE Computer Society (2008) 422–431
10. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comput. Sci. **387**(3) (2007) 332–347
11. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1) (2007)
12. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. SIGMOD Rec. **19**(2) (1990) 322–331